Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Model-Checking Multi-Threaded Distributed Java Programs

Scott D. Stoller[⋆]

Computer Science Dept., Indiana University, Bloomington, IN 47405-7104 USA. e-mail: `stoller@cs.sunysb.edu`

Received: date / Revised version: date

Abstract. State-space exploration is a powerful technique for verification of concurrent software systems. Applying it to software systems written in standard programming languages requires powerful abstractions (of data) and reductions (of atomicity), which focus on simplifying the data and control, respectively, by aggregation. We propose a reduction that exploits a common pattern of synchronization, namely, the use of locks to protect shared data structures. This pattern of synchronization is particularly common in concurrent Java programs, because Java provides built-in locks. We describe the design of a new tool for state-less state-space exploration of Java programs that incorporates this reduction. We also describe an implementation of the reduction in Java PathFinder, a more traditional state-space exploration tool for Java programs.

Key words: reduction – locks – model checking – partial-order methods – Java

1 Introduction

Development of correct software is difficult, especially for concurrent and distributed systems. Ideally, software would be generated from requirements, yielding programs that are correct by construction. Until that ideal is reached, software verification—checking whether a given program satisfies its requirements—will remain an important problem. A variety of techniques are being brought to bear on it. *State-space exploration* starts from the simple idea of exhaustive search of all possible behaviors—and hence all reachable states—of a system and checking, either during or after the search, whether the explored behaviors/states satisfy the requirements.

[⋆] Present address: Computer Science Dept., SUNY at Stony Brook, Stony Brook, NY 11794-4400. The author gratefully acknowledges the support of ONR under Grants N00014-99-1-0358 and N00014-01-1-0109 and the support of NSF under Grant CCR-9876058.

This approach is attractive because it is fully automatic. In practice, for most software systems, the state space is intractably large. This is especially true for programs written in standard programming languages, as opposed to simplified versions written in modeling languages.

Aggregation is commonly used to reduce the size of the state space. *Abstractions* simplify data by aggregating values into equivalence classes. *Reductions* simplify control by aggregating transitions into coarser-grained transitions. In both cases, the aggregation defines a transformed system that has fewer reachable states and whose correctness implies correctness of the original system (the converse sometimes also holds). State-space exploration is applied to the transformed system, yielding correctness results that hold for the original system as well.

We propose a reduction that exploits a common pattern of synchronization, namely, the use of locks to protect shared variables. This pattern of synchronization is particularly common in concurrent Java programs, because Java provides built-in locks [10]. It is also common in C programs that use the pthreads thread library [17]. In general, when exploring the state space of a concurrent program, context switches between threads must be allowed before each access to a shared variable. If that variable is protected by a lock—in other words, the lock is held whenever the variable is accessed—then it suffices to allow context switches before acquire operations on the lock, prohibiting them before accesses to the variable. Limiting the points where context switches may occur effectively increases the granularity of transitions. One can regard this as defining a *reduced system*, which is a coarser-grained version of the original system. The reduced system may have significantly fewer reachable states than the original system. For example, when Jigsaw, the World Wide Web Consortium's web server (www.w3.org/Jigsaw/), is serving small web pages, on average, every 35th bytecode performs an acquire operation (usually an invocation of a synchronized method). Such an increase in the granularity of transitions can have an even larger impact on the number of reachable states, depending on the number of threads and the pattern of requests.

The use of mutual exclusion to justify deducing properties of a system from properties of a reduced system is a well-known special case of several reduction theorems, such as [15,4,2]. When used to justify this lock-based reduction, all of these reduction theorems assume that one knows statically (*i.e.*, before state-space exploration) that selected variables are protected by locks. Static analyses like Extended Static Checking [6], type-based race detection [7], and protected variable analysis [5] can automatically provide a conservative approximation to this information.

For general finite-state systems, it might seem that the only way to automatically and accurately determine whether selected variables are protected by locks is state-space exploration of the original system, using a variant of the lockset algorithm [17] to keep track of which variables are protected by which locks. If this were the case, then the reduction would be almost pointless, because the goal is to avoid exploring the entire state-space of the original system. We show in Section 7 that one can determine exactly during state-space exploration of the *reduced* system whether all variables in a given set are protected by locks in the original system. Intuitively, this is possible because of commutativity properties of the operations used in the lockset algorithm, which reflect the fact that whether a variable is protected by a lock depends only on which accesses occur and which locks are held at each access, not on the order in which accesses occur. Actually, this is not true for the Eraser locking discipline [17], whose treatment of initialization is slightly too liberal and therefore order-dependent. Our result is for a slightly stricter locking discipline, introduced in Section 7. Our proofs are based on partial-order methods, specifically, on persistent sets [8].

The results in Section 7 cannot be derived from classic reduction theorems such as [15,4]. One might try to derive them by constructing a transformed system that is instrumented to halt immediately before it would violate the locking discipline (LD). The idea is that the transformed system would always satisfy LD, which would ensure that the hypotheses of the classic reduction theorem hold. This does not work, because the necessary instrumentation would itself perform accesses that violate LD.

Our framework handles distributed (*i.e.*, multi-process) multi-threaded systems. It combines and extends ideas in VeriSoft [9], which targets distributed systems of single-threaded processes and does not incorporate a lock-based reduction, and ExitBlock [2], which incorporates a lock-based reduction for single-process multi-threaded systems. A more detailed comparison with [2] appears in Section 2.

Section 3 provides background. Section 4 presents two partial-order methods, called persistent sets and sleep sets. Sections 5 and 7 describe our lock-based reductions. Section 6 gives an algorithm for computing persistent sets. This algorithm is not included in Section 4 because the lock-based reductions do not rely on it. This algorithm completes the picture of how to combine a lock-based reduction with general use of persistent sets.

Section 8 describes a prototype state-space exploration tool for single-process multi-threaded Java programs. It uses state-less search [9] and incorporates our reduction. *State-less search* systematically explores different schedules, without storing the set of states that have been visited. Section 9 describes a prototype implementation of our reduction in Java PathFinder (JPF) [1], a more traditional (state-based) state-space exploration tool for Java bytecode. The main advantage of state-less search is that it can be applied more easily to systems whose state is not easily captured and stored, *e.g.*, large systems written in a combination of Java and other programming languages. The main disadvantage is that it may unnecessarily explore some states multiple times. Section 10 contains some initial experimental results that compare the speed of the two implementations and measure the effectiveness of the reduction.

Sections 3–5, 7 and 10 form the core of the paper. Readers interested in implementation of reductions in state-based tools might want to read Section 9 as well. Readers interested in partial-order methods might want to read Section 6. Section 8 is mainly for readers interested in state-less search.

2 Related Work

Comparison to VeriSoft [9] and traditional reduction theorems [15,4] appears in Section 1.

Bruening's ExitBlock algorithm [2] corresponds roughly to the invisible-first state-less selective search (IF-SSS) algorithm in Section 5.1 without the use of persistent sets or sleep sets. ExitBlockRW corresponds roughly to IF-SSS with sleep sets and without persistent sets. ExitBlock and ExitBlockRW treat release as visible and acquire as invisible. This complicates deadlock detection in ExitBlock, and ExitBlockRW might miss deadlocks. The framework in [2] assumes LD is used for all shared variables. Bruening does not discuss whether ExitBlock or ExitBlockRW is guaranteed to find a violation of LD for systems that violate LD.

Corbett's protected variable reduction [5] is similar to the reduction in Section 5.2. Corbett's reduction allows context switches before all five of Java's synchronization operations (described in Section 3.2). Our reduction prohibits some of these context switches and hence can provide more benefit. Also, [5] does not provide results on checking LD during state-space exploration.

3 Background

Section 3.1 describes our system model. Sections 3.2 and 3.3 provide an informal introduction to and a more formal model of the relevant aspects of synchronization in Java. Section 3.4 expresses Eraser's locking discipline in our system model. Section 3.5 classifies operations, transitions, control points, and states as visible or invisible. Section 3.6 defines some conditions on systems.

3.1 System Model

We adopt Godefroid's model of concurrent systems [8], 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 four categories. 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 argument and the current value of the object and returns a return value and an updated value for the object. A *concurrent system* (or *system*, for brevity) is a tuple $\langle \Theta, \mathcal{O}_{unsh}, \mathcal{O}_{syn}, \mathcal{O}_{ld}, \mathcal{O}_{com}, s_{init}, \mathcal{T} \rangle$, where

Θ is a finite set of threads. A thread is a finite set of elements called *control points*. Threads are required to be pairwise disjoint.
$\mathcal{O}_{unsh}$ is the set of unshared objects, *i.e.*, objects accessed by at most one thread.
$\mathcal{O}_{syn}$ is the set of synchronization objects, defined in Section 3.3.
$\mathcal{O}_{ld}$ is the set of objects for which a locking discipline, defined in Section 3.4, is used.
$\mathcal{O}_{com}$ is a set of objects, called communication objects.
s_{init} is the initial state. State is defined below.
$\mathcal{T}$ 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 and denote them by $start(t)$ and $final(t)$, respectively.

This four-way classification of objects is the basis for classifying operations into two categories: visible and invisible. All operations on communication objects are visible, and operations on synchronization objects that may block are visible (details are given below). This classification of operations determines the reduced system; informally, in the reduced system, context switches are allowed only before transitions containing visible operations.

The objects from any Java program can be classified in this way. No assumptions are made about communication objects, so it is safe to classify all objects as communication objects. However, classifying more objects as communication objects means that more operations are visible and hence that the reduction provides less benefit. Therefore, objects should be put in the other categories whenever possible.

There is considerable flexibility in modeling Java programs as concurrent systems. For example, an object in such a model might correspond to a single storage location in the Java program (*e.g.*, a static field, an instance field of a Java object, or a variable on the call stack), or it might correspond to a collection of storage locations (*e.g.*, all fields of a Java object). Note that objects corresponding to variables on the call stack of a Java program are always unshared. There is also flexibility in the details of how to model references, exceptions, object creation, etc. Generally, this can be done as in [12], which describes a translation from Java to PROMELA (like [12], we assume Θ, $\mathcal{O}_{unsh}$, $\mathcal{O}_{syn}$, $\mathcal{O}_{ld}$, and $\mathcal{O}_{com}$ are large enough to accommodate all Java threads and objects that will be created). Such details are irrelevant here, because our implementations, described in Sections 8 and 9, do not actually construct such models: they work directly with Java bytecode. The important points about modeling are: (1) any Java program with a finite number of reachable states can be modeled as a concurrent system, and (2) one can determine which bytecodes in the Java program correspond to visible operations in the model.

Our framework handles distributed and multi-threaded systems. For example, a system containing Java processes communicating over sockets would contain some instances of java.net.Socket, which are in $\mathcal{O}_{ld}$, and an underlying socket, which is in $\mathcal{O}_{com}$. For our purposes, it is not necessary to explicitly model the division of the system into processes. What matters is how variables are accessed. For example, a variable shared only by threads in a single process could be regarded as a communication object (if accesses to that variable do not satisfy the locking discipline); conversely, if some form of distributed shared memory is used, then a variable shared by threads in multiple processes could be classified as an element of $\mathcal{O}_{ld}$.

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 an object o, we abuse notation and write $s(\theta)$ to denote the control point of thread θ in state s. Similarly, we write $s(o)$ to denote the value of o in 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 system $\mathcal{M}$, let $pending_{\mathcal{M}}(s, \theta)$ and $enabled_{\mathcal{M}}(s, \theta)$ denote the sets of transitions of thread θ that are pending and enabled, respectively, in state s (in system $\mathcal{M}$). Let $enabled_{\mathcal{M}}(s)$ 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 $\langle (L \backslash \{S\}) \cup \{F\}, apply(C, V) \rangle$, where $apply(C, V)$ denotes the values obtained by using the operations in C to update the values in V. Commands are assumed to be deterministic; non-determinism is modeled using multiple transitions. 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 system $\mathcal{M}$ is a finite or infinite sequence σ of transitions of $\mathcal{M}$ such that there exist states s_0, s_1, s_2, ... such that $s_0 = s_{init}$ and $s_0 \overset{\sigma(0)}{\rightarrow} s_1 \overset{\sigma(1)}{\rightarrow} s_2 \cdots$. Operations are deterministic, so the sequence of states $s_1, s_2, \ldots$ is completely determined by the sequence of transitions. When convenient, we regard the sequence of states as part of the execution. A state is *reachable* in $\mathcal{M}$ if it appears in some execution of $\mathcal{M}$. A control point is *reachable* if it appears in some reachable state.

3.2 Synchronization in Java

Java's concurrency model allows non-deterministic interleaving of the actions of different threads. Threads have priorities. "Threads with higher priority are generally executed in preference to threads with lower priority. Such preference is not, however, a guarantee..." [10, Section 17.12]. Our framework is designed to verify only guaranteed properties, not probabilistic properties, so we ignore such preference and hence priorities.

Java provides five built-in synchronization operations based on the classic operations on monitors [14]: acquire, release, wait, notify, and notifyAll. Java also provides bounded-time variants of wait; we do not consider them, because our system model is untimed.

One lock and one condition variable are implicitly associated with each object. A *synchronized statement* is used to acquire and release a lock. The statement

synchronized (*expr*) { *stmt* }

evaluates expression *expr* to an object reference o, acquires the lock associated with o, executes statement *stmt*, and then releases o's lock. Synchronized statements are compiled into bytecode that uses monitorenter and monitorexit instructions to acquire and release the lock, respectively.

Java also allows the keyword "synchronized" to be used as a modifier in method declarations. Declaring an instance method as synchronized is equivalent to replacing its body *b* with: synchronized (this) { *b* }.

The locks are recursive, *i.e.*, a lock is free iff each execution of the acquire operation has been matched by an execution of the release operation. Recursive locks conveniently support nested or recursive invocations of synchronized methods.

wait, notify, and notifyAll are final native methods of java.-lang.Object. They are inherited by all objects. They throw IllegalMonitorStateException if invoked by a thread that does not own the target object's lock; otherwise, they behave as follows. o.wait() adds the calling thread θ to o's wait set (*i.e.*, the set of threads waiting on o), releases o's lock, and suspends θ. When another thread notifies θ (by invoking notify or notifyAll), θ contends to re-acquire o's lock. When θ acquires the lock, the invocation of o.wait() returns. o.notify() non-deterministically selects a thread θ in o's wait set, removes θ from the set, and notifies θ. o.notifyAll() removes all threads from o's wait set and notifies each of them.

A waiting thread can also be awoken by a call to Thread.-interrupt. For simplicity, we do not consider this possibility in our model. Also, we do not consider Java's controversial weakly consistent memory model [10,16], which provides sequential consistency for objects protected by locks [2, Section 2.3.3] but not necessarily for other objects.

3.3 Synchronization Objects

In our framework, a synchronization object embodies the synchronization state that the JVM maintains for each Java object. In Java, every object contains its own synchronization state; there are no separate synchronization objects. This difference is inconsequential.

The fields of a synchronization object are: *owner* (the name of a thread, or *free*), *depth* (the number of unmatched acquire operations), and *waiters* (the set of waiting threads). We assume that the initial state so_{init} of each synchronization object has $owner = free$, $depth = 0$, and $waiters = \emptyset$. 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. For concreteness, we describe one representation here.

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 $o.owner \neq \theta$ and a command that throws an IllegalMonitorStateException, and one with guard $o.owner = \theta$ and command

$$o.depth{-}{-}; o.owner := newOwner(o.depth, \theta),$$

where $newOwner(depth, \theta)$ returns θ if $depth > 0$ and returns *free* otherwise.

Let $lkDepth_\theta$ denote an unshared object used by θ and whose domain is the natural numbers. 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

$$o.waiters.add(\theta); lkDepth_\theta = o.depth;$$
$$o.depth := 0; o.owner := free$$

followed by one with guard $o.owner = free \wedge \theta \notin o.waiters$ and command $o.owner := \theta; o.depth := lkDepth_\theta$.

Thread θ invoking 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 $o.owner = \theta \wedge o.waiters = \emptyset$ and a command that does nothing, and, for each $\theta' \in \Theta \setminus \{\theta\}$, a transition with guard $o.owner = \theta \wedge \theta' \in o.waiters$ and a command that removes θ' from $o.waiters$. 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 a command that makes $o.waiters$ empty.

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

A useful 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 own 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 which thread, if any, owns o's lock, and regardless of which threads, if any, are blocked waiting on o).

One might consider including synchronization objects in $\mathcal{O}_{ld}$ or $\mathcal{O}_{com}$ instead of treating them specially. They cannot be included in $\mathcal{O}_{ld}$, because operations on them access $o.owner$ in a way that violates LD. Classifying them as communication objects would mean that all operations on them are visible (see Section 3.5), which would increase the cost of the selective search.

To illustrate the definitions, consider a Java program with two threads that execute the following snippets of code, where z is a local variable, and $o1$ is an instance of a class with field x, and $o2$ is an instance of a class with field y:

Code for θ_1:
```
synchronized (o1) {o1.x++;};
synchronized (o2) {o2.y++;}
```

Code for θ_2:
```
synchronized (o2) { o2.y++;
                    synchronized (o1) {z=o1.x+o2.y;}}
```

This program is modeled by the concurrent system $\mathcal{M}_{ex} = \langle \Theta, \mathcal{O}_{unsh}, \mathcal{O}_{syn}, \mathcal{O}_{ld}, \mathcal{O}_{com}, s_{init}, \mathcal{T}\rangle$, where

$$\begin{array}{lll} \Theta = \{\theta_1, \theta_2\} & \theta_1 = \{a_1, \ldots, g_1\} & \theta_2 = \{a_2, \ldots, f_2\} \\ \mathcal{O}_{unsh} = \{z\} & \mathcal{O}_{syn} = \{\ell_1, \ell_2\} & \mathcal{O}_{ld} = \{x, y\} \\ \mathcal{O}_{com} = \emptyset & & \end{array}$$
$$s_{init} = \langle\{a_1, a_2\}, \{x \mapsto 0, y \mapsto 0, z \mapsto 0, \ell_1 \mapsto so_{init}, \ell_2 \mapsto so_{init}\}\rangle$$

and $\mathcal{T}$, in a self-explanatory shorthand with the operations of a transition sandwiched between the starting and final control points of the transition, is given by

Transitions of θ_1:

$\boxed{a_1}\ \ell_1.\text{acquire()}\ \boxed{b_1}\ x{+}{+}\ \boxed{c_1}\ \ell_1.\text{release()}\ \boxed{d_1}$

$\boxed{d_1}\ \ell_2.\text{acquire()}\ \boxed{e_1}\ y{+}{+}\ \boxed{f_1}\ \ell_2.\text{release()}\ \boxed{g_1}$

Transitions of θ_2:

$\boxed{a_2}\ \ell_2.\text{acquire()}\ \boxed{b_2}\ y{+}{+}\ \boxed{c_2}\ \ell_1.\text{acquire()}\ \boxed{d_2}$

$\boxed{d_2}\ z = x + y\ \boxed{e_2}\ \ell_1.\text{release()}; \ell_2.\text{release()}\ \boxed{f_2}$

For example, the first transition of θ_1 is officially

$$\begin{array}{l}\langle a_1, \\ \ell_1.owner \in \{free, \theta_1\}, \\ \ell_1.owner := \theta_1; \ell_1.depth{+}{+}, \\ b_1\rangle.\end{array}$$

If $\mathcal{M}_{ex}$ were obtained by systematic translation of a complete Java program, it would be much larger, due to modeling of creation and starting of threads in appropriate ThreadGroups, invocations of run methods, etc. Also, the granularity would be smaller, *e.g.*, θ_2's two releases would be separate transitions; we merged them into one transition to keep the state space small. Figure 1 shows the reachable states of $\mathcal{M}_{ex}$.

3.4 Locking Discipline

The locking discipline of [17] 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.

For an execution $\sigma = s_0 \overset{\sigma(0)}{\to} s_1 \overset{\sigma(1)}{\to} s_2 \cdots$, $endInit(\sigma, o)$ is the index of the first state in σ in which o is accessed by a second thread; formally, $endInit(\sigma, o)$ 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.

Locking Discipline (LD). An execution $\sigma = s_0 \overset{\sigma(0)}{\to} s_1 \overset{\sigma(1)}{\to} s_2 \cdots$ of a system $\langle \Theta, \mathcal{O}_{unsh}, \mathcal{O}_{syn}, \mathcal{O}_{ld}, \mathcal{O}_{com}, s_{init}, \mathcal{T}\rangle$ satisfies LD iff, for all $o \in \mathcal{O}_{ld}$, one of the following conditions holds:

LD-RO: o is read-only after initialization, *i.e.*, there exists a constant c such that for all $i \geq endInit(\sigma, o)$, $s_i(o) = c$.

LD-lock: o is lock-protected after initialization, *i.e.*, there exists a synchronization object $o_1 \in \mathcal{O}_{syn}$ such that, for all $i \geq endInit(\sigma, o)$, for all $\theta \in \Theta$, if $access(s_i, \theta, o)$, then θ owns o_1's lock in s_i.

A system satisfies LD iff all of its executions do. For example, $\mathcal{M}_{ex}$ satisfies LD.

Godefroid [8] 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 LD to depend on such uses. This motivates our definition of "accesses".

3.5 Visible and Invisible

Recall that operations are classified as visible or invisible. In the reduced system, context switches are allowed only immediately before transitions containing visible operations.

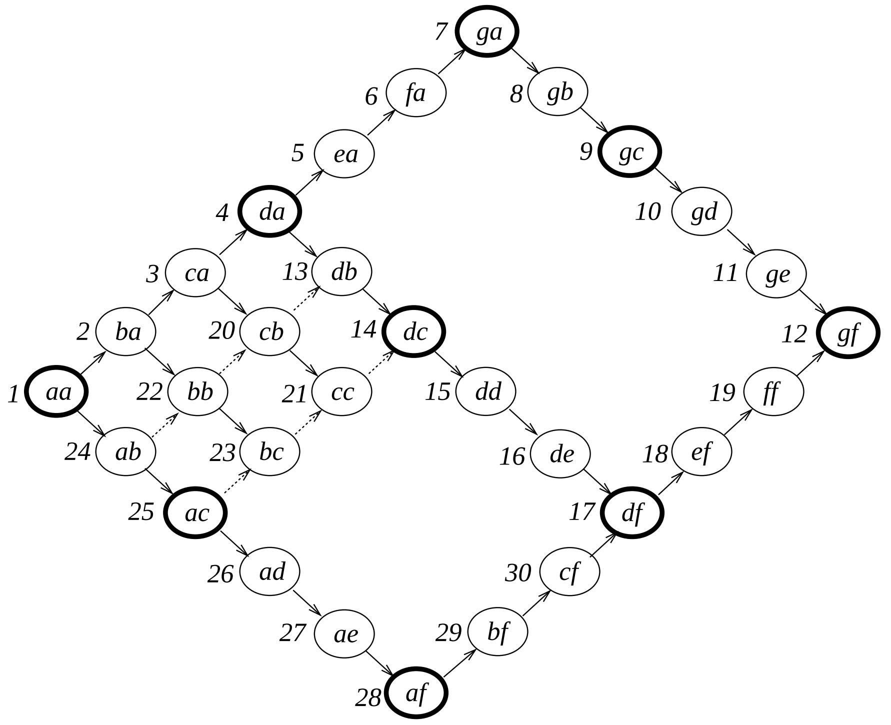

Fig. 1. State Space of $\mathcal{M}_{ex}$. The first and second letters within each state indicate the control points of θ_1 and θ_2, respectively; for example, ba abbreviates $\{b_1, a_2\}$. Values of objects are not shown but can easily be inferred. Edges represent transitions. Transitions of θ_1 and θ_2 point diagonally up and down, respectively. States are numbered in depth-first order. Thick circles denote visible states (see Section 3.5). The sleep set technique (see Section 4) avoids exploring the dotted edges.

All operations on communication objects are visible, as in [9]. Operations on synchronization objects that may block are visible; specifically, operations in the transitions for acquire and wait that do not throw exceptions are visible. All other operations are invisible. A transition t is visible if (1) t's command or guard contains a visible operation or (2) t is part of a non-deterministic choice, *i.e.*, in some reachable state, multiple transitions with starting control point $start(t)$ are enabled. An over-approximation of the second condition can be used when classifying transitions: unnecessarily classifying a transition as visible is safe (*i.e.*, all the theorems below still hold). For concurrent systems that model Java programs, a simple over-approximation of the second condition is that it holds for transitions corresponding to invocations of notify(), which non-deterministically chooses a waiting thread to awaken, and invocations of methods of java.util.Random, which we interpret as non-deterministic (see Section 8.3).

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 [9].

For example, in $\mathcal{M}_{ex}$, control points a_1, d_1, g_1, a_2, c_2, and f_2 are visible. In Figure 1, thick circles denote visible states of $\mathcal{M}_{ex}$.

If additional operations on synchronization objects were introduced, the invisibility of existing operations would need to be re-evaluated. For example, consider introducing a non-blocking operation on synchronization objects that returns true iff the lock is free. This would invalidate SyncWithoutLock and require re-classifying release as visible.

3.6 Conditions on Systems

We define some conditions on systems.

Separation (of visible and invisible transitions): 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.

InitVis (initial control locations are visible): For every thread θ, $s_{init}(\theta)$ is visible. This condition is inessential but convenient.

BoundedInvis (bound on invisible transition sequences): 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.

DetermInvis (determinism of invisible control points): For every reachable state s, for every thread θ, θ has at most one enabled invisible transition in s.

NonBlockInvis (non-blocking invisible control points): For every thread θ, for every invisible control point S of θ, for every reachable state s containing S, $enabled(s, \theta)$ is non-empty.

PureVis (pure visible transitions): Visible transitions do not contain operations on objects in $\mathcal{O}_{ld}$.

These conditions, except possibly BoundedInvis, are satisfied by all reasonable models of Java programs with the natural granularity in which a transition corresponds roughly to a bytecode or a fragment of a bytecode. Separation and DetermInvis follow from the definitions of visible transitions and visible control points. InitVis is easily enforced by classifying the first transition of each thread as visible; recall that classifying transitions as visible is always safe. NonBlockInvis holds because the only blocking operations are acquire, wait, and some operations on communication objects (*e.g.*, receive on a socket), and all of these operations are visible. PureVis holds for models in which transitions correspond roughly to bytecodes or fragments of bytecodes, because all bytecodes satisfy this condition. PureVis typically does not hold for coarse-grained models, such as $\mathcal{C}(\mathcal{M})$ in Section 5.2. BoundedInvis is checked during state-space exploration: if an executing thread does not execute a visible transition within a user-specified amount of time, an error ("possible violation of BoundedInvis") is reported.

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

4 State-less Selective Search

Two techniques used to make state-space exploration more efficient are persistent sets and sleep sets. They attempt to reduce the number of explored states and transitions by exploiting independence of transitions. They are called *selective search* techniques, because they justify exploring only a carefully selected subset of the enabled transitions from each visited state. We start with informal descriptions of these techniques and then give formal definitions. The material in this section is paraphrased from [8,9].

Informally, two transitions are *independent* if they commute. A set T of transitions enabled in a state s is *persistent* in s if everything the system can do from s while staying outside T (*i.e.*, while not executing transitions in T) is independent with transitions in T. In this case, transitions t' outside T can safely be deferred until after a transition t in T has executed, because t and t' are independent. Thus, in the first line of procedure DFS in the selective search algorithm in Figure 2, the set T of transitions to be explored is based on a persistent set PS(s), rather than $enabled(s)$. An algorithm for computing persistent sets, based on the static structure of the system, appears in Section 6.

Sleep sets are computed by the selective search algorithm in Figure 2 and reflect the history of the search. After exploring a transition t from a state s and recursively exploring all states reachable therefrom (note that the search is depth-first), t is propagated into the sleep sets of states s' that are explored on paths σ from s that start with a different transition, as long as the path σ contains only transitions that are independent with t. Informally, it is safe to defer execution of t along paths containing only transitions independent with t, because the transitions in such a path were already explored on some path from s that starts with t, so it is unnecessary to also explore interleavings in which t occurs immediately after those transitions. Consequently, t is not explored from such states s'; this is implemented by subtracting $sleep$ from PS(s) in the first line of procedure DFS. If a transition dependent with t is executed, t is removed from the sleep set; this is implemented by the calculation of $sleep'$ in procedure DFS.

Dependency Relation. Let $\mathcal{T}$ and $State$ be the sets of transitions and states, respectively, of a system $\mathcal{M}$. $D \subseteq \mathcal{T} \times \mathcal{T}$ is an *unconditional dependency relation* for $\mathcal{M}$ 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 $\mathcal{M}$ 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 [8, 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.

Consider $\mathcal{M}_{ex}$, introduced in section 3.3. For control points S and F, let t_{SF} denote the transition with starting control point S and final control point F, if any. The set $\{t_{b_1c_1}\}$ is persistent in state ba. To see this, note that σ in the definition of persistent set effectively ranges over non-empty prefixes of $\langle t_{a_2b_2}, t_{b_2c_2} \rangle$, and that these two transitions are independent with $t_{b_1c_1}$. We do not need to consider sequences σ containing $t_{c_2d_2}$, because such sequences must also contain $t_{b_1c_1}$ and $t_{c_1d_1}$ (because of contention for ℓ_1's lock), so the antecedent $\sigma(i) \notin T$ in the definition of persistent set is false.

Let PS(s) return a set of transitions that is persistent in s. Let D be an unconditional dependency relation.[1] Godefroid's state-less selective search (SSS) using persistent sets and sleep sets appears in Figure 2, where $exec$ and $undo$ are specified by: if $s \xrightarrow{t} s'$, then $exec(s,t) = s'$ and $undo(s',t) =$

[1] A conditional dependency relation can be used, but this involves some minor complications [18].

```
SSS() {                  DFS(sleep) {
  stack := empty;          $T := \mathrm{PS}(curState) \setminus sleep$;
  curState := $s_{init}$;        for each $t$ in $T$
  DFS($\emptyset$);                stack.push($t$);
}                              $curState := exec(curState, t)$;
                               // next line is in IF-SSS, not in SSS
                               $curState := execInvis_{\mathcal{M}}(curState, thread(t))$;
                               $sleep' := \{t' \in sleep \mid \langle t, t' \rangle \notin D\}$;
                               DFS($sleep'$);
                               stack.pop();   // pop $t$ from stack;
                               $curState := undo(curState, t)$;
                               $sleep := sleep \cup \{t\}$;
                         }
```

Fig. 2. State-less Selective Search (SSS) and Invisible-First State-less Selective Search (IF-SSS) using persistent sets and sleep sets. The two algorithms differ only in the presence of the indicated assignment statement. *stack* and *curState* are global variables. $stack$ is needed to report error traces to the user. Also, $stack$ is needed if *undo* is implemented using reset+replay, as discussed in Section 8.4.

s. Because SSS is state-less, it may visit states multiple times; persistent sets and sleep sets help reduce this redundancy.

Consider $\mathcal{M}_{ex}$. Suppose $\mathrm{PS}(s) = enabled(s)$, and the dependency relation used to compute sleep sets is: two transitions are independent if they use disjoint sets of objects. Then SSS explores all states in Figure 1, and all edges except the dotted ones. The depth-first numbering in Figure 1 reflects the order in which states of $\mathcal{M}_{ex}$ are explored for the first time during execution of SSS. Some states are explored again later in the search. Specifically, a state s is explored once for each path from s_{init} to s in Figure 1, ignoring paths containing dotted edges. To see that the sleep set technique avoids exploration of, *e.g.*, the dotted edge from state ab to state bb, consider the relevant steps of the algorithm: (1) initially, DFS($\emptyset$) is invoked in state aa, (2) in the second iteration of the **for** loop in that state, $sleep = \{t_{a_1b_1}\}$, (3) $t_{a_1b_1}$ is independent with $t_{a_2b_2}$, so DFS($\{t_{a_1b_1}\}$) is invoked in state ab, so $t_{a_1b_1}$ is not explored from that state.

SSS works best for acyclic state spaces. Two problems arise if the state space contains cycles. First, a state-less search cannot recognize that it is in a cycle, so a purely depth-first approach may get stuck in a cycle and never explore some states. This problem can avoided by using iterative deepening. Second, persistent sets might cause some enabled transitions to be permanently deferred (*i.e.*, not included in the persistent sets), causing the selective search to miss some states [8, Chapter 6]; this is called the *ignoring problem* [19]. This motivates definition of a state-less search that uses iterative deepening and sleep sets but not persistent sets (or, as we will see, uses persistent sets only in a restricted way). $\mathrm{SSS}_{\mathrm{bnd}}(d)$ denotes a depth-bounded variant of SSS that explores executions of length at most d and uses $\mathrm{PS}(s) = enabled(s)$. $\mathrm{SSS}_{\mathrm{cyc}}$ denotes an iterative-deepening variant of SSS, that calls $\mathrm{SSS}_{\mathrm{bnd}}$ with increasingly larger bounds until the longest execution explored by $\mathrm{SSS}_{\mathrm{bnd}}$ is shorter than the bound.

Following Godefroid [8] 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 5.2), 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 [8, chapter 7]. We say that a search algorithm explores a control point S if it explores a state containing S.

Theorem 1. *Let $\mathcal{M}$ be a system with a finite state space. (a) If $\mathcal{M}$'s state space is acyclic, then SSS explores all reachable deadlocks and control points of $\mathcal{M}$. (b) $\mathrm{SSS}_{\mathrm{cyc}}$ explores all reachable deadlocks and control points of $\mathcal{M}$.*

Proof: (a) This is a paraphrase of Theorem 2 of [9]. Assertion violations correspond to reachability of control points. (b) As discussed above, allowing cycles in the state space potentially causes two problems, and $\mathrm{SSS}_{\mathrm{cyc}}$ avoids both of them. Note that the claim holds even if $\mathrm{SSS}_{\mathrm{cyc}}$ diverges. □

Persistent sets and sleep sets can also be used in traditional (*i.e.*, state-based) selective search [8]. We omit details of traditional selective search algorithms, because the results for traditional selective search in Sections 5.2 and 7 are independent of those details.

5 Two Approaches to Lock-Based Reduction

Sections 5.1 and 5.2 describe two approaches to lock-based reduction, based on persistent sets and composition of transitions, respectively. Section 5.3 compares the two approaches.

5.1 Invisible-First State-less Selective Search

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

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

Proof: See Appendix. □

Suppose the system satisfies LD, 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 no enabled invisible transitions. BoundedInvis implies that this iteration terminates. Let $\mathit{execInvis}_{\mathcal{M}}(s, \theta)$ be the unique state obtained by performing this procedure starting from state s; if θ has no enabled invisible transitions in state s, then let $\mathit{execInvis}_{\mathcal{M}}(s, \theta) = s$. Specializing SSS to work in this way yields Invisible-First State-less Selective Search (IF-SSS), given in Figure 2. IF-SSS$_{\mathrm{cyc}}$ is defined analogously.

Theorem 3. *Let $\mathcal{M}$ be a system with a finite state space and satisfying LD, Separation, BoundedInvis, DetermInvis, and NonBlockInvis. (a) If $\mathcal{M}$'s state space is acyclic, then IF-SSS explores all reachable deadlocks and control points of $\mathcal{M}$. (b) IF-*SSS$_{\mathrm{cyc}}$ *explores all reachable deadlocks and control points of $\mathcal{M}$.*

Proof: See Appendix. □

Consider applying IF-SSS to $\mathcal{M}_{ex}$ from Section 3.3, with $\mathrm{PS}(s) = \mathit{enabled}(s)$ and with the same dependency relation for sleep sets as in Section 4. IF-SSS avoids exploring states *bb* and *cb* (see Figure 1). For example, the dotted inedge of *bb* is not explored because of sleep sets, and the solid inedge of *bb* is not explored because $\mathit{execInvis}_{\mathcal{M}_{ex}}(ba, \theta_1)$ explores only $t_{b_1 c_1}$.

5.2 Composing Transitions

Another approach to lock-based reduction is to aggregate a visible transition and the subsequent sequence of invisible transitions of the same thread into a single transition. Transitions are aggregated (composed) as follows. For a sequence σ of transitions, let $\mathit{cmd}_c(\sigma)$ be the sequential composition of the commands of the transitions in σ, and let $\mathit{guard}_c(\sigma)$ be the weakest predicate ensuring that when each transition t in σ is executed, t's guard holds. guard_c can be expressed in terms of the weakest-precondition predicate transformer wp [11]:[2]

$$\begin{aligned}\mathit{guard}_c(\sigma) = {} & \wedge\ \mathit{guard}(\sigma(0)) \\ & \wedge\ \textstyle\bigwedge_{0<i<|\sigma|} wp(\mathit{guard}(\sigma(i)), \\ & \qquad\qquad \mathit{cmd}_c(\sigma(0..i-1))),\end{aligned} \tag{1}$$

where $\mathit{guard}(\langle S, G, C, F\rangle) = G$. We assume $\mathit{guard}_c(\sigma)$ can be expressed in terms of the available operations on objects.

For a system $\mathcal{M}$ satisfying LD, Separation, BoundedInvis, and DetermInvis, the transformed system $\mathcal{C}(\mathcal{M})$ with composed transitions is the same as $\mathcal{M}$ except that the set of transitions is as follows. Let b be the bound in BoundedInvis for $\mathcal{M}$. For each visible transition $t = \langle S, G, C, F\rangle$ in $\mathcal{T}$, for each sequence σ of invisible transitions such that $|\sigma| \le b$ and $\mathit{guard}_c(\langle t\rangle \cdot \sigma) \ne \mathrm{false}$ and the intermediate control points match up (*i.e.*, $\mathit{final}(t) = \mathit{start}(\sigma(0))$ and for all $i < |\sigma| - 1$, $\mathit{final}(\sigma(i)) = \mathit{start}(\sigma(i+1))$) and $\mathit{final}(\mathit{last}(\sigma))$ is visible, $\mathcal{C}(\mathcal{M})$ has the transition $\langle S, \mathit{guard}_c(\langle t\rangle \cdot \sigma), \mathit{cmd}_c(\langle t\rangle \cdot \sigma), \mathit{final}(\mathit{last}(\sigma))\rangle$.

[2] We let a list of formulas bulleted with $\wedge$ or $\vee$ denote the conjunction or disjunction, respectively, of the formulas, using indentation to eliminate parentheses.

For example, $\mathcal{C}(\mathcal{M}_{ex})$ has exactly 9 reachable states. These are the visible states of $\mathcal{M}_{ex}$, denoted by thick circles in Figure 1. The transitions of $\mathcal{C}(\mathcal{M}_{ex})$ lead between these states. For example, the first transition $t_{a_1 d_1}$ of θ_1 in $\mathcal{C}(\mathcal{M}_{ex})$ is

$$\begin{aligned}&\langle a_1, \\ &\ell_1.\mathit{owner} \in \{\mathit{free}, \theta_1\}, \\ &\ell_1.\mathit{owner} := \theta_1; \ell_1.\mathit{depth}{+}{+}; x++; \ell_1.\mathit{depth}{-}{-}; \\ &\quad \ell_1.\mathit{owner} := \mathit{newOwner}(\ell_1.\mathit{depth}, \theta_1), \\ &d_1\rangle.\end{aligned} \tag{2}$$

The guard of the release in $t_{c_1 d_1}$ does not contribute to the guard of the composed transition, because

$$wp(\ell_1.\mathit{owner} = \theta,\ \ell_1.\mathit{owner} := \theta) = \mathrm{true}. \tag{3}$$

The command in (2) can be simplified; we do not consider this further.

Theorem 4. *Let $\mathcal{M}$ be a system satisfying LD, Separation, InitVis, BoundedInvis, and DetermInvis. $\mathcal{M}$ and $\mathcal{C}(\mathcal{M})$ have the same reachable visible states.*

Proof: See Appendix. □

An invisible control point S is defined to be reachable in $\mathcal{C}(\mathcal{M})$ if a composed transition passes through S in some execution of $\mathcal{C}(\mathcal{M})$.

Theorem 5. *Let $\mathcal{M}$ be a system with a finite state space and satisfying LD, Separation, InitVis, BoundedInvis, DetermInvis, and NonBlockInvis. $\mathcal{M}$ and $\mathcal{C}(\mathcal{M})$ have the same reachable deadlocks. $\mathcal{M}$ and $\mathcal{C}(\mathcal{M})$ have the same reachable control points.*

Proof: See Appendix. □

Theorems 1(a) (or 1(b)) and 5 imply that, for a system $\mathcal{M}$ satisfying their hypotheses, SSS (or SSS$_{\mathrm{cyc}}$) applied to $\mathcal{C}(\mathcal{M})$ explores all reachable deadlocks and control points of $\mathcal{M}$.

Theorem 5 is also useful with traditional selective search. For example, Theorems 5.2 and 6.14 of [8] together with Theorem 5 imply that, for a system $\mathcal{M}$ satisfying their hypotheses, traditional selective search with persistent sets and sleep sets [8, Figure 6.2] applied to $\mathcal{C}(\mathcal{M})$ explores all reachable deadlocks and control points of $\mathcal{M}$.

5.3 Comparison of the Two Approaches

The invisible-first approach (Section 5.1) is worthwhile for three reasons. First, Theorem 2 shows that this reduction is a special case of persistent sets, thereby demonstrating 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 could cause transitions to be removed from sleep sets. Third, the guards of composed transitions sometimes introduce dependencies that cause SSS applied to $\mathcal{C}(\mathcal{M})$ to explore more interleavings than IF-SSS applied to $\mathcal{M}$. For example, consider a thread θ that is ready to execute

$$\begin{aligned}&\mathrm{someVisibleOp}; \\ &\textbf{if } (x_1)\ \{\ \textbf{if } (x_2)\ c_1\ \textbf{else } c_2\ \} \\ &\textbf{else } \{\ \textbf{if } (x_3)\ c_3\ \textbf{else } c_4\ \}\end{aligned} \tag{4}$$

where the x_i are in $\mathcal{O}_{ld}$ and the c_i do not contain visible operations. Let S denote the starting control point of this statement. In the original system $\mathcal{M}$, θ does not access x_2 or x_3 at S. In $\mathcal{C}(\mathcal{M})$, 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 $\mathcal{C}(\mathcal{M})$, the access by θ to x_2 in a state s with $s(\theta) = S$ and $s(x_1) = \text{false}$ is an artifact of composition. Such accesses induce dependencies that could cause persistent sets to be larger in $\mathcal{C}(\mathcal{M})$ than $\mathcal{M}$. Whether this occurs depends partly on how persistent sets are computed. This would not occur if they are computed as described in Section 6, because $\mathit{pendInvisOps}(s, \theta)$ would be the same in $\mathcal{M}$ and $\mathcal{C}(\mathcal{M})$. This could occur if the calculation of $\mathit{pendInvisOps}$ exploited information from static analysis. In practice, the number of such code fragments that actually lead to smaller persistent sets is expected to be too small to have a significant impact on overall performance.

The composition approach (Section 5.2) is worthwhile because it sometimes achieves a stronger 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 $\mathcal{C}(\mathcal{M})$, each thread does this with a single composed transition, and the two composed transitions are independent (because atomically acquiring and then releasing a lock has no net effect on the state of the synchronization object), so SSS applied to $\mathcal{C}(\mathcal{M})$ could explore a single interleaving of these transitions. In $\mathcal{M}$, each thread does this with a sequence of three transitions, and the transitions that manipulate the lock are dependent, so IF-SSS applied to $\mathcal{M}$ explores multiple interleavings.

Which approach yields better performance depends mainly on whether the stronger reduction of the composition approach outweighs the cost of recording invisible operations. Recording invisible operations and using them in the computation of sleep sets typically does not consume a significant fraction of the overall running time, in part because the lockset algorithm is more expensive than this recording. Thus, the composing transitions approach typically has similar or better performance, compared to the invisible-first approach. The stronger reduction provided by the composing transitions approach is particularly attractive when used with state-less search, because avoiding exploration of one of the interleavings that leads to a state s can avoid one redundant exploration of the entire subgraph of the state space reachable from s.

6 Computing Persistent Sets

Computing persistent sets in a given state requires information about possible future behaviors of each thread. Static analysis or user-supplied annotations may be used to obtain an upper bound (with respect to the subset ordering) on the set of operations that each thread may perform. Let $\mathit{ops}(\theta)$ denote such an upper bound for thread θ. Let $\mathit{invisOps}(\theta)$ be the set of invisible operations in $\mathit{ops}(\theta)$. Throughout this section, we ignore operations on unshared objects.

To compute small persistent sets, an upper bound on the set of operations used by the pending transitions of a thread is also needed. Let $\mathit{usedVisOps}(t)$ be the set of visible operations used by a transition t. We assume that in each visible state s, for each thread θ, the following set is known:

$$\mathit{pendVisOps}(s, \theta) = \{\mathit{usedVisOps}(t) \mid t \in \mathit{pending}(s, \theta)\}.$$

$\mathit{pendVisOps}(s, \theta)$ is typically obtained by intercepting each visible operation. A non-trivial upper bound on the set of invisible operations used by pending transitions of θ in s can be obtained by exploiting LD. For concreteness, we describe how to calculate a bound from the data structures maintained by the lockset algorithm [17]. We assume in this section that the system satisfies LD; the lockset algorithm is used here only to obtain information about which locks protect each object in $\mathcal{O}_{ld}$. If that information is available from static analysis, then running the lockset algorithm during the search is unnecessary.

The lockset algorithm maintains the following values for each object o: $o.\mathit{mode}$, 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 o), or shared-modified (none of the above conditions hold); $o.\mathit{firstThread}$, which is the first thread that accessed o (*i.e.*, the thread that initializes o; $o.\mathit{firstThread}$ is undefined when o is in virgin mode); and $o.\mathit{candLkSet}$ ("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 $o.\mathit{mode}$ from exclusive to shared or shared-modified). $o.\mathit{candLkSet}$ contains all locks (*i.e.*, equals $\mathcal{O}_{syn}$) while o is in exclusive mode.

$\mathit{held}(s, \theta)$ is the set of synchronization objects whose locks are owned by θ in state s. $\mathit{acqing}(s, \theta)$ is the set of synchronization objects o such that $\mathit{pendVisOps}(s, \theta)$ contains an acquire operation on o. $\mathit{rdOnly}(o.op)$ holds if $o.op$ is read-only. $\mathit{mayInit}(s, \theta, o)$ holds if θ may be the first thread to access virgin object o in state s. For example, in Java, for instance variables, one might require that θ be the thread that allocated o; for static variables of a class C, one might require that θ be the thread that caused class C to be loaded. An upper bound on the set of invisible operations used by pending transitions of θ in s is

$$\begin{array}{l} \mathit{pendInvisOps}(s, \theta) = \\ \quad \{o.op \in \mathit{invisOps}(\theta) \mid \begin{array}[t]{l} \vee\ o \in \mathcal{O}_{syn} \\ \vee\ \exists o_1 \in \mathit{held}(s, \theta) \cup \mathit{acqing}(s, \theta) : \\ \quad \mathit{LDallows}(s, \theta, o_1, o.op)\} \end{array} \end{array}$$

$$\begin{array}{l} \mathit{LDallows}(s, \theta, o_1, o.op) = \\ \quad \vee\ o.\mathit{mode} = \text{virgin} \wedge \mathit{mayInit}(s, \theta, o) \\ \quad \vee\ o.\mathit{mode} = \text{exclusive} \wedge \begin{array}[t]{l} \vee\ \theta = o.\mathit{firstThread} \\ \vee\ \mathit{rdOnly}(o.op) \\ \vee\ o_1 \in o.\mathit{candLkSet} \end{array} \\ \quad \vee\ o.\mathit{mode} = \text{shared} \wedge (\mathit{rdOnly}(op) \vee o_1 \in o.\mathit{candLkSet}) \\ \quad \vee\ o.\mathit{mode} = \text{shared-modified} \wedge o_1 \in o.\mathit{candLkSet} \end{array}$$

We can obtain a tighter bound on $\mathit{pendInvisOps}$ (and hence potentially smaller persistent sets) if the system satisfies the following stricter version of LD-lock: 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 $\mathcal{M}$" 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 modification potentially makes $o.candLkSet$ and $pendInvisOps$ smaller.

Persistent sets can be computed using a variant of Algorithm 2 of [8], which is based on Overman's Algorithm. It uses the following relation.

Might-be-the-first-to-interfere-with. Operation op' might be the first to interfere with operation op from state s, denoted $op \rhd_s op'$, if [8, Definition 4.26]: (1) op and op' are operations on the same object and (2) there exists a sequence σ of transitions such that (2a) $s = s_0$ and $s_0 \overset{\sigma(0)}{\to} s_1 \overset{\sigma(1)}{\to} s_2 \cdots \overset{\sigma(n-1)}{\to} s_n \overset{\sigma(n)}{\to} s_{n+1}$, and (2b) for all $i \in [0..n-1]$, all operations in $\sigma(i)$ are independent with op in s_i, and (2c) $\sigma(n)$ uses op', and op and op' are dependent in s_n.

Algorithm 2-LD, our variant of Algorithm 2 of [8], is:

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

The might-be-the-first-to-interfere-with relation is determined manually for each kind of object and stored in a library [8, Section 4.7]. A might-be-the-first-to-interfere-with relation for synchronization objects appears in Figure 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). Let $\theta{:}op$ denote a synchronization operation op performed by thread θ. Figure 3 gives $\rhd_s$ only for operations on synchronization objects in transitions that do not throw IllegalMonitorStateException; for operations (in op_1 or op_2) in thetransitions that throw IllegalMonitorStateException, SyncWithoutLock implies that $op_1 \rhd_s op_2$ is false.

Justifying the relation in Figure 3 is not difficult. For example, consider the entry for $\theta_1{:}o.\mathrm{acquire} \rhd_s \theta_2{:}o.\mathrm{acquire}$. Consider a state s in which $o.owner \notin \{free, \theta_1\}$. Let σ be a sequence of transitions starting from s and ending with a transition that performs $\theta_2{:}o.\mathrm{acquire}$. As usual, let $s_0 \overset{\sigma(0)}{\to} s_1 \overset{\sigma(1)}{\to} s_2 \cdots \overset{\sigma(n-1)}{\to} s_n \overset{\sigma(n)}{\to} s_{n+1}$, with $s_0 = s$. It suffices to show that either an operation dependent with $\theta_1{:}o.\mathrm{acquire}$ occurs before $\theta_2{:}o.\mathrm{acquire}$ in σ, or $\theta_1{:}o.\mathrm{acquire}$ is independent with $\theta_2{:}o.\mathrm{acquire}$ in s_n. Consider two cases.

case: $s(o.owner) = \theta_2$. If $s_n(o.owner) = \theta_2$ holds, then $\theta_1{:}o.\mathrm{acquire}$ is disabled in s_n, and executing $\theta_2{:}o.\mathrm{acquire}$ in s_n leaves $\theta_1{:}o.\mathrm{acquire}$ disabled, so $\theta_1{:}o.\mathrm{acquire}$ and $\theta_2{:}o.\mathrm{acquire}$ are independent in s_n. If $s_n(o.owner) \neq \theta_2$, then θ_2 must free o's lock at some point during σ, *i.e.*, for some $i \in [0..n]$, $s_i(o.owner) = free$, and the release operation by θ_2 in the transition $\sigma(i-1)$ enables $\theta_1{:}o.\mathrm{acquire}$ and hence is dependent with it in s_{i-1}.

case: $s(o.owner) = \theta_3$, where $\theta_3 \in \Theta \setminus \{\theta_1, \theta_2\}$. In this case, θ_3 must free o's lock before $\theta_2{:}o.\mathrm{acquire}$ can occur; that release operation by θ_3 also enables $\theta_1{:}o.\mathrm{acquire}$ and hence is dependent with it.

Another example is $\theta_1{:}o.\mathrm{notifyAll} \rhd_s \theta_2{:}o.\mathrm{wait}_1$. These two operations are independent in all states, because in any state, at most one of them is enabled (because the guard of each requires o's lock), and executing the enabled one (if any) does not enable the other one. This argument blurs the distinction between guards and commands. The operations on o in the command of a transition t_2 that performs wait_1 do not commute with the operations in the command of a transition t_1 that performs notifyAll. Nevertheless, t_1 and t_2 are independent in all states, because of their guards, by the above argument. Thus, this dependency between the operations in the commands can safely be ignored, because it never induces dependency between transitions. Dependency between transitions is what matters, because it is the basis for defining persistent sets and sleep sets.

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

Proof: This follows from correctness of Algorithm 2 of [8]. □

For example, Algorithm 2-LD can compute non-trivial persistent sets for dining philosophers. Ignoring initialization, the heart of the code for each philosopher is

^{a}synchronized (lfork) { ^{b}synchronized (rfork) { eat } }

where a and b are labels representing control points, lfork and rfork are instance fields that refer to objects representing this philosopher's left and right forks, respectively, and eat is an operation on an unshared object. Consider using Algorithm 2-LD on a model of this program with three philosophers, corresponding to threads θ_0, θ_1, and θ_2. Let f_i be a synchronization object representing a fork. For $i \in [0..2]$, the left and right forks of philosopher i are f_i and $f_{i\oplus 1}$, respectively, where $\oplus$ denotes addition modulo 3. We trace the execution of Algorithm 2-LD in the unique reachable state s in which θ_0 is at label b, and θ_1 and θ_2 are at label a. The relevant sets of operations are

$$\begin{aligned} pendVisOps(s,\theta_0) &= \{f_2.\mathrm{acquire}\} \\ pendVisOps(s,\theta_1) &= \{f_2.\mathrm{acquire}\} \\ pendInvisOps(s,\theta_i) &= \{f_i.\mathrm{release}, f_{i\oplus 1}.\mathrm{release}\} \\ ops(\theta_2) &= \{f_2.\mathrm{acquire}, f_0.\mathrm{acquire}, \\ &\quad\ f_2.\mathrm{release}, f_0.\mathrm{release}\} \end{aligned}$$

Suppose we start with $T = \{\theta_0\}$. In step 2 with $\theta = \theta_0$ and $\theta' = \theta_1$, we have $\theta_0 : f_2.\mathrm{acquire} \rhd_s \theta_1 : f_2.\mathrm{acquire}$, so θ_1 is inserted in T. In step 2 with $\theta = \theta_0$ and $\theta' = \theta_2$, all of

op_1 \ op_2	acquire	release	wait$_1$	wait$_2$	notifyAll
acquire	$owner \in \{free, \theta_1\}$	$owner = \theta_2$	$owner = \theta_2$	$\vee\ \theta_2 \in waiters \wedge owner = \theta_1$ $\vee\ \theta_2 \notin waiters \wedge owner \in \{free, \theta_1\}$	
release					
wait$_1$	$owner = \theta_1$				
wait$_2$	$owner = free$				$owner \in \{free, \theta_1\}$
notifyAll					

Fig. 3. $op_1 \triangleright_s op_2$ holds only if the predicate in the appropriate box holds. An empty box denotes false. op_i is an operation performed by thread θ_i on synchronization object o, with $\theta_1 \neq \theta_2$. The row for $op_1 = \mathrm{notify}$ and the column for $op_2 = \mathrm{notify}$ are not shown; all entries in them are false. We do not consider $\theta_1 = \theta_2$, because Algorithm 2-LD does not depend on it. wait$_1$ and wait$_2$ correspond to the operations in the first and second non-exception-throwing transitions in wait, respectively. $owner$ and $waiters$ abbreviate $o.owner$ and $o.waiters$, respectively.

the relevant might-be-the-first-to-interfere-with relations are false, so θ_2 is not inserted in T. Next, step 2 is executed again, and there is a new combination of threads to consider, namely, $\theta = \theta_1$ and $\theta' = \theta_2$. All of the relevant might-be-the-first-to-interfere-with relations are false, so the algorithm terminates with $T = \{\theta_0, \theta_1\}$.

7 Checking LD on the Reduced System

If accesses to objects in $\mathcal{O}_{ld}$ are expected to satisfy LD, but no static guarantee is available, LD can be checked during state-space exploration using the lockset algorithm [17]. The results in Sections 5 and 6 do not directly apply in this case, because they assume that the original (unreduced) system satisfies LD. Here we extend those results to ensure that, if the original system violates a slightly stricter variant of LD, then state-space exploration of the reduced system finds a violation.

Savage *et al.* observe that their liberal treatment of initialization makes Eraser's checking undesirably dependent on the scheduler [17, p. 398]. For the same reason, IF-SSS might indeed miss violations of LD. Consider a system in which θ_1 can perform the sequence of two 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 $v \in \mathcal{O}_{ld}$ is an integer variable, $o \in \mathcal{O}_{syn}$, and $sem \in \mathcal{O}_{com}$ is a semaphore, initialized to zero. This system violates LD, because $v := 1$ can occur after $v := 2$, and θ_1 owns no locks when it executes $v := 1$. IF-SSS does not find a violation, because after $sem.Up()$, $execInvis$ immediately executes $v := 1$. Similarly, SSS applied to $\mathcal{C}(\mathcal{M})$ would not find a violation of LD.

We strengthen the locking discipline's 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 $\mathcal{O}_{ld}$ to Θ be included as part of the system. For $o \notin \mathrm{domain}(initThread)$, let $endInit'(\sigma, o) = 0$. For $o \in \mathrm{domain}(initThread)$, for $\sigma = s_0 \overset{\sigma(0)}{\to} s_1 \overset{\sigma(1)}{\to} s_2 \cdots$, let $endInit'(\sigma, o)$ be the second smallest i such that $(\exists \theta \in \Theta : access(s_i, \theta, o))$, or $|\sigma|$ if no such value exists. Let LD$'$ denote LD with $endInit$ replaced with $endInit'$, and extended with the requirement that for each object o in the domain of $initThread$, $initThread(o)$ is the first thread to access o.[3] The lockset algorithm can easily be modified to check LD$'$; we call the modified version the lockset$'$ algorithm. We assume guards of transitions in $\mathcal{C}(\mathcal{M})$ are constructed using short-circuiting conjunction, so the artifacts described in Section 5.3 do not affect the lockset$'$ algorithm (in contrast, persistent sets are calculated from the static structure of the system, with incomplete information about object values, so short-circuiting has limited impact on that calculation). We assume that accesses to objects in $\mathcal{O}_{ld}$ by the guard of a transition t are checked in each state in which t is pending (in other words, in each state, guards of all pending transitions are evaluated). It suffices to check accesses to objects in $\mathcal{O}_{ld}$ 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 LD$'$-lock is equivalent to LD$'$-lock, in the sense that it does not change the set of systems satisfying LD$'$:

> o is lock-protected after initialization, *i.e.*, there exists a synchronization object $o_1 \in \mathcal{O}_{syn}$ such that, for all $i \geq endInit'(\sigma, o)$, (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.

Let $\sigma; t$ denote execution of σ followed by evaluation of t's guard and, if t is enabled, execution of t's command. Recall that conditions on systems (Separation, BoundedInvis, etc.) are defined in Section 3.6.

Theorem 7. *Let $\mathcal{M}$ be a system satisfying Separation. For all threads θ, all reachable states s, and all executions σ_0 leading to s, if $enabled(s, \theta)$ contains an invisible transition, then either (a) $enabled(s, \theta)$ is persistent in s or (b) $enabled(s, \theta)$ contains a transition t such that either (b1)*

[3] It is easy to show that LD$'$ is stricter than LD. This observation does not enable simple proofs of the theorems in this section from the theorems in previous sections or *vice versa*.

$\sigma_0; t$ violates LD$'$ or (b2) $s \xrightarrow{t} s'$ and a violation of LD$'$ is reachable from s'.

Proof: See Appendix. □

Theorem 8. *Let $\mathcal{M}$ be a system with a finite state space and satisfying Separation, BoundedInvis, DetermInvis, NonBlockInvis, and PureVis. Assume $\mathcal{M}$ runs the lockset$'$ algorithm. (a) If $\mathcal{M}$'s state space is acyclic, $\mathcal{M}$ violates LD$'$ iff IF-SSS finds a violation of LD$'$. (b) $\mathcal{M}$ violates LD$'$ iff IF-SSS$_{\text{cyc}}$ finds a violation of LD$'$.*

Proof: See Appendix. □

Theorem 9. *Let $\mathcal{M}$ be a system with a finite state space and satisfying Separation, InitVis, BoundedInvis, DetermInvis, NonBlockInvis, and PureVis. Assume $\mathcal{M}$ runs the lockset$'$ algorithm. $\mathcal{M}$ violates LD$'$ iff $\mathcal{C}(\mathcal{M})$ violates LD$'$.*

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

($\Rightarrow$): Theorem 8(b) implies that IF-SSS$_{\text{cyc}}$ explores an execution σ of $\mathcal{M}$ that violates LD$'$. Composing sequences of transitions of $\mathcal{M}$ in σ to form transitions of $\mathcal{C}(\mathcal{M})$ yields an execution of $\mathcal{C}(\mathcal{M})$ that violates LD$'$. Note that $\mathcal{C}(\mathcal{M})$ is not expected to satisfy PureVis. □

The stricter constraints on initialization in LD$'$ allow the definition of $\mathit{pendInvisOps}$ to be tightened. Let $\mathit{pendInvisOps}'$ denote that variant of $\mathit{pendInvisOps}$. Let Algorithm 2-LD$'$ denote the variant of Algorithm 2-LD that uses $\mathit{pendInvisOps}'$.

Theorem 10. *Let $\mathcal{M}$ be a system that runs the lockset$'$ algorithm. In every state s of $\mathcal{M}$, Algorithm 2-LD$'$ returns a set P such that either P is persistent in s or P contains a transition t such that t violates LD$'$ in s.*

Proof: $\mathit{pendInvisOps}'$ is the only part of Algorithm 2-LD$'$ that depends on LD$'$. $\mathit{pendInvisOps}'(s, \theta)$ is invoked only for threads θ that have already been added to T. Suppose for all threads θ in T, all transitions in $\mathit{enabled}(s, \theta)$ satisfy LD$'$ in s. Then all invocations of $\mathit{pendInvisOps}'$ in this invocation of Algorithm 2-LD$'$ return accurate results, so Theorem 6 implies that P is persistent in s. Suppose there exists a thread θ in T such that some transition t in $\mathit{enabled}(s, \theta)$ violates LD$'$ in s. Then P contains t, and t violates LD$'$ in s. □

The results in Theorems 8–10 can easily be generalized to reflect that, if static analysis ensures that accesses to some objects in $\mathcal{O}_{ld}$ satisfy LD$'$, then it suffices to run the lockset$'$ algorithm only for the other objects in $\mathcal{O}_{ld}$.

Let $\mathcal{M}$ be a system with a finite and acyclic state space and satisfying Separation, InitVis, BoundedInvis, DetermInvis, NonBlockInvis and PureVis. Assume $\mathcal{M}$ runs the lockset$'$ algorithm. Consider applying IF-SSS with Algorithm 2-LD$'$ to $\mathcal{M}$. Theorems 8(a) and 10 imply that if no violation of LD$'$ is found, then $\mathcal{M}$ satisfies LD$'$ and hence LD. Theorem 3 implies that all reachable deadlocks and control points of $\mathcal{M}$ were explored.

Consider applying SSS with Algorithm 2-LD$'$ to $\mathcal{C}(\mathcal{M})$. Theorems 1(a) and 10 imply that if no violation of LD$'$ is found, then $\mathcal{C}(\mathcal{M})$ satisfies LD$'$. Theorem 9 implies that $\mathcal{M}$ satisfies LD$'$ and hence LD. Theorems 1(a) and 5 together imply that all reachable deadlocks and control points of $\mathcal{M}$ were explored.

Consider applying Godefroid's traditional selective search with persistent sets and sleep sets [8, Figure 6.2] to $\mathcal{C}(\mathcal{M})$, where $\mathcal{M}$ is as above except without the acyclicity requirement. By the same reasoning as in the previous paragraph, except with references to Theorem 1(a) replaced with references to [8, Theorem 6.12], if no violation of LD$'$ is found, then $\mathcal{C}(\mathcal{M})$ satisfies LD$'$ and all reachable deadlocks and control points of $\mathcal{M}$ were explored.

8 Implementation of State-less Search with Lock-Based Reduction

JavaChecker is a prototype implementation of state-less search for multi-threaded single-process Java programs. It incorporates our lock-based reduction and has been applied to some simple programs (*e.g.*, dining philosophers). It currently has several limitations, *e.g.*, array accesses are not intercepted, and Algorithm 2-LD$'$ and support for communication objects and RMI are unimplemented.

JavaChecker transforms Java class files (source code is not needed) by inserting calls to a scheduler at visible operations and inserting calls to the lockset$'$ algorithm at accesses to shared objects. The scheduler, written in Java, performs state-less selective search. Markus Dahm's Byte Code Engineering Library, available from bcel.sourceforge.net, 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.

JavaChecker exploits annotations indicating which objects are (possibly) shared. We use allocation sites as static names for (equivalence classes of) objects [3]. Specifically, object creation instructions (namely, the new instruction and invocations of java.lang.Class.newInstance and java.lang.Object.-clone) may 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 instructions are potentially shared; accesses to them are intercepted to check LD$'$ and, if necessary, are recorded to determine dependencies. Currently, annotations are provided by the user; escape analysis, such as [20], could provide some of them automatically.

8.1 Granularity

By default, classes have *field granularity*, *i.e.*, the intercepted operations are field accesses. For some classes, it is desirable to consider execution of a method to be one operation (or, for some blocking methods, a small number of operations) for purposes of checking LD′ and computing dependencies. We say that such classes have *method granularity*. For example, with semaphores, it is desirable for operations seen by the scheduler to be up (also called V or signal) and down (also called P or wait), not field accesses. Method granularity reduces overhead and allows use of class-specific dependency relations. User-supplied annotations indicate which classes have method granularity. Currently, method granularity is supported only for instance methods; static fields are always handled with field granularity.

When methods are considered as operations, the boundaries of the operation must be defined carefully, so that dependencies are defined and computed appropriately. In our framework, by default, an invocation i of an instance method of a class with method granularity represents accesses to `this` performed by i but not accesses to `this` performed by methods invoked within i; furthermore, it does not represent accesses to objects other than `this` or accesses to static fields. Accesses by i to other objects are intercepted based on the granularities of the classes of those objects. Thus, indicating that a class C has method granularity determines only how accesses to instances of C are intercepted. For a class C with method granularity, we require: (G1) all instance methods of C (including inherited methods) 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 an instance o of C has fields that are accessed by methods invoked on objects other than o, those field accesses would also need to be recorded. Therefore, for a class C with method granularity, we also require: (G2) all instance fields (including inherited fields) are private or final (accesses to final fields are not intercepted) and (G3) instance methods of C do not directly access fields of other instances of C (*i.e.*, instances other than `this`). If (G3) turns out to be undesirably restrictive (*e.g.*, for classes that use such accesses to implement comparisons, such as equals), we can deviate from this ideal and explicitly record such field accesses; a simple static analysis can identify getfield and putfield instructions that possibly access instances other than `this`.

8.2 Synchronization Operations

Synchronized methods 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 synchronized method m of C, the wrapper class contains a wrapper method that overrides m. The wrapper method yields control to the scheduler and then waits for permission to proceed. When the scheduler gives it ownership of the appropriate lock and lets it continue, it invokes super.m and then releases the lock.

An "invokevirtual $C.m$" instruction requires no explicit modification; the JVM's built-in method lookup mechanism efficiently determines whether the target object is shared (i.e., whether it is an instance of the original class or the wrapper class) and invokes the appropriate method. For method invocations on unshared instances, the overhead is negligible.

An obvious alternative approach, which we call Inline, is to insert near each invocation instruction some bytecode that explicitly tests whether the instance is shared and, if so, performs the steps described above. With Inline, 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 Inline, 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 that implement Runnable or extend Thread are special (we assume such methods are not invoked directly by the application): their first action is to block, waiting for permission to proceed.

The other visible synchronization operations are intercepted using Inline; this includes "invokespecial $C.m$", monitorenter, and invocations of java.lang.Object.wait. The bytecode inserted near these instructions must efficiently determine whether the target object is shared. Inserting a boolean field in java.lang.-Object would be a nice solution if it didn't give the JVM (in Sun JDK 1.2.1) a heart attack. Our transformation inserts 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". This works fine with Sun JDK 1.2, but causes the HotSpot VM in JDK 1.3 and JDK 1.4 to die. More portable but slower alternatives are to look up the object in a hash table or use java.lang.Object.getClass to check whether the object is an instance of a wrapper class.

Invocations of notify and notifyAll are intercepted, so that our system can keep track of which threads, if any, are currently waiting on an object; the JVM provides no direct way to determine this. Similarly, releases are intercepted, and our system keeps track of which thread, if any, owns an object's lock, because the JVM provides no direct way to determine this. Interception of monitorexit instructions is easy. Interception of the implicit release performed when a synchronized method returns is slightly tricky; in particular, if a synchronized method invoked with invokespecial throws an exception, the inserted code 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.

Our system maintains its own copy of all synchronization-related state, so executing the JVM's built-in synchronization operations would be redundant. Those operations are removed by the transformation.

8.3 Method Invocations and Field Accesses

Methods of classes with method granularity are intercepted in a similar way as synchronized methods. In short, invokevirtual instructions are intercepted using wrapper methods that call the lockset$'$ algorithm, record the operation (if necessary), and invoke super.m. Invokespecial instructions and field accesses (and, in principle, array accesses, though this is unimplemented) are intercepted using the Inline approach.

The lockset$'$ algorithm requires associating a candidate lock set, *etc*., with each potentially shared storage location. We maintain this information for each object and each static field, by inserting an instance field in each wrapper class and inserting appropriate static fields in each class. The prototype currently does not maintain this information separately for each instance field, so a violation of LD$'$ is reported if different locks protect different fields of an object.

java.util.Random is treated specially. Instances of it generate pseudo-random sequences of numbers. Instances that are created with a specified seed behave the same way in the original and transformed program. Instances created without a specified seed behave differently. In the original program, they are implicitly seeded from the real-time clock. In the transformed program, they are non-deterministic: the scheduler explores transitions corresponding to each possible return value. This is similar to VS_Toss in VeriSoft [9].

8.4 Undo

$undo(s, t)$, as used in SSS or IF-SSS, can be implemented in three ways: reverse computation, reset+replay, and checkpointing. Reverse computation is efficient but difficult to implement. JavaChecker uses reset+replay (like VeriSoft), mainly because it is relatively easy to implement. Specifically, undo is implemented by resetting the system to its initial state and replaying the sequence of transitions in $stack$ (see Figure 2). ExitBlock [2] and Java PathFinder [1] use checkpointing, which requires a custom JVM but is more efficient than reset+replay for systems with long executions.

9 Implementation of Lock-Based Reduction in Java PathFinder

Java PathFinder (JPF) [1] is based on a custom JVM, written in Java, that supports traditional selective search, as in Spin [13]. We implemented the lock-based reduction in JPF by (1) modifying the existing JPF implementation of the lockset algorithm to use a different notion of initialization, and (2) modifying the scheduler so that context switches are allowed only in visible states. The user supplies two files, .commun and .unshared, that classify objects into $\mathcal{O}_{unsh}$, $\mathcal{O}_{ld}$, and $\mathcal{O}_{com}$. Each file contains a list of class names and object creation instructions. Instances of those classes and objects created by those instructions are in the corresponding category. The classification based on object creation site takes precedence in case of conflicts. Objects not explicitly classified by .commun or .unshared are in $\mathcal{O}_{ld}$ by default. Correctness of the classification is fully checked during state-space exploration, using the modified lockset algorithm for objects in $\mathcal{O}_{ld}$, and using a straightforward algorithm for objects in $\mathcal{O}_{unsh}$.

Locksets are stored in a hash table, so at most one copy of each distinct lockset is stored, as in Eraser [17]. We plan to implement the memoization optimization described in [17], which caches the results of set intersection operations.

10 Experimental Results

We report two kinds of experiments. The first compares the execution speeds of state-less search in JavaChecker and traditional search in JPF. The second measures the benefit of the lock-based reduction, using the JPF implementation. The two kinds of experiments used Sun JDK 1.2.2 and Sun JDK 1.3.0, respectively. All experiments were done with a Sun UltraSPARC-II 300 MHz CPU. One should not generalize too much from the results of these few experiments. More experiments with a variety of larger programs are needed.

To compare the execution speeds of JavaChecker and JPF, we ran the usual deadlock-prone dining philosophers program in both systems. Due to differences in the granularity used by the two implementations for some system classes, the two tools do not explore exactly the same number of interleavings with multiple philosophers, so we performed this experiment with a single philosopher that executes its main loop (acquire both chopsticks, eat, release both chopsticks) 5,000 times. Context switches consume a small fraction of the running time, so the relative speed of the two implementations is about the same with any number of philosophers. The running times are 563.0 sec for JPF, and 5.79 sec for JavaChecker. Thus, JavaChecker's execution speed is about two orders of magnitude faster than JPF's. The reasons are simple. JPF incurs the cost of hashing and storing states and has interpretive overhead for execution of every bytecode instruction. JavaChecker does not hash or store states, and it incurs no overhead for accesses to unshared variables (*e.g.*, bytecodes that manipulate the operand stack) and relatively little overhead (only the cost of recording invisible operations) for accesses to unshared objects. However, in overall performance on non-trivial systems, this constant factor in favor of JavaChecker can very easily be outweighed by the cost of replay (which can be arbitrarily large, depending on the length of executions) and the cost of redundant exploration of states (which can be exponential in the number of distinct visited states, depending on the structure of the computation and the effectiveness of the partial-order reductions). Consequently, state-less search is useful in practice mainly for systems whose state is not easily captured and stored, *e.g.*, systems written in a combination of programming languages.

To measure the benefit of the reduction, we used two programs supplied by the developers of JPF. HaltException, described in [12], involves a producer thread and a consumer thread that exchange data items *via* a shared FIFO buffer. In our experiments, the threads exchange 10 items, and the buffer has capacity 6. Clean, which is roughly the same as [1, Figure 1], is based on code in NASA's Remote Agent and involves two threads that use bounded counters, synchronized methods, wait, and notifyAll. The measurements appear in Figure 4. In the Atomicity column, "bytecode" and "line" mean that a transition corresponds to execution of one bytecode instruction and one line of source code, respectively; "reduced" means that the lock-based reduction is used. The Lockset column indicates whether the (modified) lockset algorithm was turned on.

For HaltException, the reduction (with lockset on) reduces memory usage by a factor of 67, and running time by a factor of 2.6, compared to bytecode atomicity with lockset off. For Clean, the reduction (with lockset on) reduces memory usage by a factor of 28.2, and running time by a factor of 6.7, compared to bytecode atomicity with lockset off. If static analysis can show that objects in $\mathcal{O}_{ld}$ are lock-protected, then running the reduction with lockset off would be more appropriate. For both programs, if the user is interested in verifying absence of race conditions, and static analysis cannot verify this, then comparing the reduction with lockset on to bytecode atomicity with lockset on would be more appropriate.

HaltException and Clean are little more than synchronization skeletons. They perform few local computations. This is an unfavorable case for the reduction, which is more beneficial for programs that perform more local computation.

For both programs, the reduction in the number of visited states is larger than the reduction in memory, because JPF uses substructure sharing, so storing a new state that differs only slightly from the previous state requires only a small additional amount of memory.

A performance comparison of lock-based reduction and source-line atomicity has limited significance, because the former is a sound reduction (relative to bytecode atomicity), and the latter is not.

The lockset algorithm's data structures (described in Section 6) are part of the state, so turning on the lockset algorithm can increase the number of visited states as well as the amount of memory used. For Clean with bytecode atomicity, turning on the lockset algorithm decreases the final memory usage (reported in the table) by about 14%; we are investigating this anomaly. The time overhead of the lockset algorithm is significant but can be decreased using memoization, as mentioned in Section 9.

Acknowledgements. Gregory Alexander, Aseem Asthana, Sean Broadley, Sriram Krishnan, and Adam Wick implemented JavaChecker. Han Li implemented the reduction in JPF, with help and advice from Flavio Lerda and Willem Visser. I thank Patrice Godefroid for his comments about partial-order methods, and Ernie Cohen for his comments about reduction theorems.

Atomicity	Lockset	States	RAM	Time
bytecode	off	201,688	56.1	245.3
bytecode	on	323,854	92.3	1729.9
line	off	8,266	2.8	23.0
line	on	11,457	3.9	54.6
reduced	off	788	0.78	19.7
reduced	on	788	0.83	92.8

Atomicity	Lockset	States	RAM	Time
bytecode	off	58,370	15.9	77.9
bytecode	on	58,370	13.7	167.0
line	off	8,365	2.5	23.9
line	on	8,365	2.5	32.2
reduced	off	209	0.55	10.3
reduced	on	209	0.56	11.6

Fig. 4. Experimental results for HaltException (top) and Clean (bottom). The units for memory and running time are MB and seconds, respectively. Memory is "Memory used after gc" (garbage collection), as reported by JPF. Running time is user+system time. For measurements with bytecode and line atomicity, we used unmodified JPF version 0.9 from NASA.

References

1. G. Brat, K. Havelund, S.-J. Park, and W. Visser. Model checking programs. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 3–12, Sept. 2000.
2. D. 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.
3. D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 296–310. ACM Press, 1990.
4. E. Cohen and L. Lamport. Reduction in TLA. In D. Sangiorgi and R. de Simone, editors, *Proc. 9th Int'l. Conference on Concurrency Theory (CONCUR)*, volume 1466 of *Lecture Notes in Computer Science*, pages 317–331. Springer-Verlag, 1998.
5. J. C. Corbett. Using shape analysis to reduce finite-state models of concurrent Java programs. *ACM Transactions on Software Engineering and Methodology*, 9(1):51–93, Jan. 2000.
6. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq SRC, 1998. Extended Static Checking for Java is available at http://www.research.compaq.com/SRC/esc/.
7. C. Flanagan and S. Freund. Type-based race detection for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232. ACM Press, 2000.
8. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
9. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 174–186. ACM Press, 1997.
10. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 2000.
11. D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
12. K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4), Apr. 2000.

13. G. J. Holzmann and D. Peled. An improvement in formal verification. In *Proc. International Conference on Formal Description Techniques (FORTE)*, 1994.
14. B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):106–117, Feb. 1980.
15. R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.
16. W. Pugh. The Java memory model. Available at http://www.cs.umd.edu/~pugh/java/memoryModel/.
17. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.
18. S. D. Stoller. Model-checking multi-threaded distributed Java programs. Technical Report 536, Computer Science Dept., Indiana University, Jan. 2000. Revised May 2000.
19. A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.
20. J. Whaley and M. 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, Oct. 1999. Appeared in *ACM SIGPLAN Notices* 34(10).

Appendix

Proof of Theorem 2: Let σ be a sequence of transitions such that $s_0 \overset{\sigma(0)}{\rightarrow} s_1 \overset{\sigma(1)}{\rightarrow} s_2 \cdots \overset{\sigma(n-1)}{\rightarrow} s_n \overset{\sigma(n)}{\rightarrow} s_{n+1}$ 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 $s(\theta) = S$.

We first prove by induction that σ does not contain transitions of θ. Base case: $\sigma(0)$ is executed 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 current 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 $\mathcal{O}_{ld}$, where $0 \leq j \leq i$.

case: t_d's guard does not access o in s_{i+1}. Then $\sigma(j)$'s access to o does not affect t_d's enabledness.

case: t_d's guard accesses o in s_{i+1}. Since $S_d = S$, t_d's guard also accesses o in s. $\mathit{thread}(\sigma(j)) \neq \theta$, so o becomes shared at or before s_j, so $\sigma(j)$ is not part of initialization of o, so LD-RO or LD-lock holds for o at s_j and thereafter.

- case: LD-RO 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: LD-lock holds for o. This case is impossible. Let o_1 be the synchronization object whose lock protects accesses to o. $\sigma(j)$ is not part of initialization of o, so LD-lock implies $\mathit{thread}(\sigma(j))$ owns o_1's lock in state s_j, so θ does not own o_1's lock in s_j. By the induction hypothesis, $\sigma(j..i)$ does not contain transitions of θ, so θ does not own o_1's lock in s_{i+1}, so LD-lock implies that t_d (including t_d's guard) does not access o in s_{i+1}, a contradiction.

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.

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

case: $o \in \mathcal{O}_{syn}$. t is invisible, so op is release, notify, or notifyAll.

- case: θ owns o's lock in s_n. As shown above, $\mathit{thread}(\sigma(n))$ is not θ, so $\mathit{thread}(\sigma(n))$ does not own 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 $\mathit{thread}(\sigma(n))$ to hold o's lock, so SyncWithoutLock-2 implies that execution of op_n is unaffected by execution of op.
- case: θ does not own 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: $o \in \mathcal{O}_{ld}$. By hypothesis, θ and $\mathit{thread}(\sigma(n))$ both access o in s_n, and $\theta \neq \mathit{thread}(\sigma(n))$, so o is shared in s_n, so $\sigma(n)$ and t are not part of initialization of o.

- case: LD-RO holds for o. LD-RO implies that $\sigma(n)$ and t do not update o in s_n, so op_n and op are independent in s_n.
- case: LD-lock holds for o. This case is impossible. Let o_1 be the synchronization object whose lock protects accesses to o. By hypothesis, $\sigma(n)$ and t both access

o in s_n, so θ and $thread(\sigma(n))$ both own o_1's lock in s_n, a contradiction, because $\theta \neq thread(\sigma(n))$.

case: $o \in \mathcal{O}_{com}$. This case is impossible, because all operations on a communication object are visible, and t contains an operation on o and is invisible. □

Proof of Theorem 3: (a) 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, for invisible states, returns a singleton set containing an invisible transition; NonBlockInvis (defined in Section 3.6) implies that IF-SSS applies PS to visible states only, so there is no need to restrict the behavior of PS for visible states. 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. Separation, DetermInvis, and the above hypothesis about PS for invisible states together imply that $|T| = 1$ whenever T contains an invisible transition, so the argument of DFS never contains invisible transitions. Thus, inserting invisible transitions in sleep sets (in the last line of the **for** loop) does not reduce the number of transitions explored by SSS.

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 in $enabled(s) \setminus enabled(s, \theta)$; (2) in IF-SSS, when an invisible transition is executed, $sleep \subseteq enabled(curState)$ (similarly, in SSS, $sleep \subseteq enabled(curState)$ when $exec$ is called).

(b) BoundedInvis implies that the limited use of persistent sets in IF-SSS$_{\text{cyc}}$ does not introduce the ignoring problem, because a call to $execInvis$ defers transitions only until the next visible transition is explored, which occurs within a bounded number of steps. □

Proof of Theorem 4: Let s be a visible state. We show that s is reachable in $\mathcal{M}$ iff s is reachable in $\mathcal{C}(\mathcal{M})$.

($\Leftarrow$): This direction follows immediately from the observation that for every execution σ of $\mathcal{C}(\mathcal{M})$, expanding each transition t' in σ' into the sequence of transitions of $\mathcal{M}$ from which t' is composed yields an execution of $\mathcal{M}$.

($\Rightarrow$): Let s be a reachable visible state of $\mathcal{M}$; thus, there is an execution σ of $\mathcal{M}$ such that $s_0 \xrightarrow{\sigma(0)} s_1 \xrightarrow{\sigma(1)} s_2 \cdots \xrightarrow{\sigma(n-1)} s_n \xrightarrow{\sigma(n)} s_{n+1}$ 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 (if any) 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 $\sigma$ has $thread(t) \neq thread(t_i)$)
        swap $t_i$ and $t$ in $\sigma$;
```

We show that each swap preserves the fact that σ is an execution of $\mathcal{M}$. 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 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 $o \in \mathcal{O}_{ld}$. 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: LD-RO holds for o. LD-RO implies that t does not update o, a contradiction.

case: LD-lock holds for o. Let o_1 be the synchronization object whose lock protects accesses to o. LD-lock implies that $thread(t)$ and $thread(t_i)$ both own o_1's lock in s, and $thread(t_i)$ owns 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 $enabled(s, thread(t_i))$ 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. This completes the proof that each swap yields an execution of $\mathcal{M}$. In the remainder of this proof, σ denotes the re-arranged execution.

Let $j+1$ be the number of visible transitions in σ, and let $v \in [0..j] \rightarrow [0..n]$ 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 $v(0) = 0$. Let $w_i = \sigma(v(i)..v(i+1)-1)$. Let s'_{i+1} denote the state after execution of w_i in σ. We show that s'_{i+1} is visible. w_i changes the control point only of $thread(w_i(0))$, so it suffices to show that $s'_{i+1}(thread(w_i(0)))$ is visible.

case: w_i contains the last transition of $thread(w_i(0))$ in σ. Then visibility of s'_{i+1} follows from visibility of s.

case: w_i does not contain the last transition of $thread(w_i(0))$ in σ. The next transition of $thread(w_i(0))$ after w_i in σ is the first transition in some w_{i_1} and hence is visible, so Separation implies that $s'_{i+1}(thread(w_i(0)))$ is visible.

By definition of $\mathcal{C}(\mathcal{M})$, for each w_i, $\mathcal{C}(\mathcal{M})$ has a transition that is the sequential composition of the transitions in w_i; let

$\sigma'(i)$ equal that transition. Thus, $s_0 \overset{\sigma'(0)}{\to} s'_1 \overset{\sigma'(1)}{\to} s'_2 \cdots \overset{\sigma'(j-1)}{\to} s'_j \overset{\sigma'(j)}{\to} s'_{j+1}$ and $s'_{j+1} = s$, so s is reachable in $\mathcal{C}(\mathcal{M})$. □

Proof of Theorem 5: NonBlockInvis (defined in Section 3.6) implies that all deadlocks of $\mathcal{M}$ are visible, so Theorem 4 implies that that $\mathcal{M}$ and $\mathcal{C}(\mathcal{M})$ have the same reachable deadlocks.

For a control point S, we show that S is reachable in $\mathcal{M}$ iff S is reachable in $\mathcal{C}(\mathcal{M})$. The proof of the backward direction (⇐) is the same as for Theorem 4. Consider the forward direction (⇒). By hypothesis, S is reachable in $\mathcal{M}$, so there exists an execution σ_0 of $\mathcal{M}$ that ends in a state containing S. NonBlockInvis implies that σ_0 can be extended to form an execution σ of $\mathcal{M}$ that ends in a visible state. The construction in the proof of the forward direction of Theorem 4 shows that σ can be re-arranged by swapping transitions into an execution σ' of $\mathcal{C}(\mathcal{M})$. σ and σ' contain the same control points, so S is reachable in $\mathcal{C}(\mathcal{M})$. □

Proof of Theorem 7: Some observations about accesses: (O1) 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; (O2) 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 LD′: (O3) a transition t that is pending in a state s can violate LD′ in s even if t is disabled in s; (O4) after initialization, whether accesses to an object satisfy LD′-RO is independent of the order in which the accesses occur (what matters is whether the set contains a non-read-only operation); (O5) after initialization, whether accesses to an object satisfy LD′-lock is independent of the order of the accesses, because set intersection is commutative and associative.

Let s, θ, and σ_0 be as in the statement of this theorem. Let σ and t be as in the proof of Therorem 2. Note that $t \in \mathit{enabled}(s, \theta)$. Consider cases corresponding to the places in which a violation of LD′ could affect the proof of Theorem 2

case 1: for all $j \in [0..|\sigma|–1]$, $\sigma_0 \cdot \sigma(0..j)$ and $\sigma_0 \cdot \sigma(0..j–1); t$ both satisfy LD′. In this case, the proof of Theorem 2 goes through, so disjunct (a) in the statement of the theorem holds.

case 2: there exists $j \in [0..|\sigma| - 1]$ such that $\sigma_0 \cdot \sigma(0..j)$ or $\sigma_0 \cdot \sigma(0..j-1); t$ violates LD′. Let j denote the least such j. The proof of Theorem 2 goes through for $\sigma(0..j-1)$; specifically, for $i < j$, $\mathit{thread}(\sigma(i)) \neq \theta$ 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: $\sigma_0 \cdot \sigma(0..j-1); t$ violates LD′. The violation occurs when t accesses some object o in s_j.

case 2.1.1: $\sigma(0..j–1)$ contains an initialization transition $\sigma(i)$ for o. This implies o is in the domain of $\mathit{initThread}$. The definition of j implies that $\sigma_0 \cdot \sigma(0..i)$ does not violate LD′, so $\mathit{thread}(\sigma(i)) = \mathit{initThread}(o)$. LD′ requires that $\mathit{thread}(\sigma(i))$ be the first thread to access o, so $\sigma_0; t$ violates LD′, because $\mathit{thread}(\sigma(i)) \neq \theta$, and because t is enabled in s and hence accesses in s all of the objects that it accesses in s_j. Thus, disjunct (b1) holds. (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 O1–O5 and invisibility of t imply that $\sigma_0 \cdot \langle t \rangle \cdot \sigma(0..j-1)$ also violates LD′, since the same set of accesses to o with the same sets of held locks occur in $\sigma_0 \cdot \sigma(0..j–1) \cdot \langle t \rangle$ and $\sigma_0 \cdot \langle t \rangle \cdot \sigma(0..j–1)$, because $\mathit{thread}(\sigma(i)) \neq \theta$ for $i < j$. The violation might occur at any point in $\langle t \rangle \cdot \sigma(0..j-1)$; depending on when it occurs, disjunct (b1) or disjunct (b2) holds.

case 2.2: $\sigma_0 \cdot \sigma(0..j-1); t$ satisfies LD′. Thus, in $\sigma_0 \cdot \sigma(0..j)$, some access by $\sigma(j)$'s guard or command violates LD′.

case 2.2.1: after $\sigma_0 \cdot \sigma(0..j–1)$, some access by $\sigma(j)$'s guard violates LD′. Observations O1–O5 and invisibility of t imply that $\sigma_0 \cdot \langle t \rangle \cdot \sigma(0..j–1); \sigma(j)$ also violates LD′, even though $\sigma(j)$ might be disabled after $\sigma_0 \cdot \langle t \rangle \cdot \sigma(0..j–1)$. Thus, disjunct (b2) holds.

case 2.2.2: after $\sigma_0 \cdot \sigma(0..j–1)$, all accesses by $\sigma(j)$'s guard satisfy LD′. Thus, in $\sigma_0 \cdot \sigma(0..j)$, some access by $\sigma(j)$'s command violates LD′. As in the proof of Theorem 2, t does not affect $\sigma(j)$'s enabledness (this follows from the hypotheses of cases 2.2 and 2.2.2), so $\sigma(j)$ is enabled after $\sigma_0 \cdot \langle t \rangle \cdot \sigma(0..j-1)$. Observations O1–O5 and invisibility of t imply that $\sigma_0 \cdot \langle t \rangle \cdot \sigma(0..j)$ also violates LD′, so disjunct (b2) holds. □

Proof of Theorem 8: (a) The proof of the reverse direction (⇐) of the "iff" is straightforward. For the forward direction (⇒), we suppose $\mathcal{M}$ violates LD′ and show that IF-SSS finds a violation. The proof involves an invariant about states in the stack of the depth-first search. To express the invariant conveniently, we introduce a new local variable v of procedure DFS, and insert "*v* := *curState*" before the assignment to T. Also, we assume there is a global variable *violated* that is initially false and is set to true by the lockset algorithm when a violation of LD′ is encountered. Let *callStack* denote the call stack, ignoring frames for functions other than DFS. Let $f.x$ denote the value of a local variable x in a stack frame f. Let $\mathrm{violR}(s)$ denote that a violation of LD′ is reachable from s; because the system is running the lockset′ algorithm, the state contains the lockset′ algorithms data structures, so this property is a function of the state s, independent of the execution that led to s. Let $s \xrightarrow{t}_{eI}$ denote evaluation of t's guard in s and, if t is enabled in s, execution of t's command, leading to a state s', followed by $\mathit{execInvis}(s', \mathit{thread}(t))$, leading to a state s''; if t is enabled in s, then we write $s \xrightarrow{t}_{eI} s''$. It is sensible to ask whether $s \xrightarrow{t}_{eI}$ violates LD′, independent of the execution that led to s, for the same reasons. The invariant

I is

$$\begin{array}{l} \vee\ \mathit{violated} \\ \vee\ (\forall f \in \mathit{callStack} : \mathrm{violR}(f.v) \Rightarrow (\exists t \in f.T, s' \in \mathit{State} : \\ \qquad (f.v \xrightarrow{t}_{eI} \text{violates LD}')\ \vee\ (f.v \xrightarrow{t}_{eI} s' \wedge \mathrm{violR}(s')))) \end{array}$$

To avoid clutter, this formula relies on the pretense that creation of a stack frame f for DFS and the assignments to $f.v$ and $f.T$ occur atomically.

By hypothesis, $\mathcal{M}$ runs the lockset$'$ algorithm, so violation of LD$'$ corresponds to reachability of a control point. Theorem 1(a) can be generalized to show that selective search starting from any state s explores all control points reachable from s. This implies that if persistent sets and sleep sets are computed correctly, then I is preserved, and a violation of LD$'$ is found.

Consider a stack frame f, a (visible) transition $t_v \in f.T$, and a (invisible) transition t_i executed in a state s_i by the call to $\mathit{execInvis}$ after execution of t_v from $f.v$. Let σ_0 be the sequence of transitions that led to this visit to s_i. Violations of LD$'$ can potentially cause two problems when t_i is explored: (P1) $\{t_i\}$ might not be persistent in s_i; (P2) t_i might be dependent in s_i with a transition t' in sleep, in which case t' incorrectly remains in sleep. We show that these potential problems do not falsify I. Specifically, we suppose that $\mathrm{violR}(s_i)$ holds and show that one of the disjuncts in the last line of I holds.

(P1) Suppose $\{t_i\}$ is not persistent in s_i. Theorem 7, Separation (defined in Section 3.6), and DetermInvis imply that either $\sigma_0; t_i$ violates LD$'$, or $s_i \xrightarrow{t_i} s' \wedge \mathrm{violR}(s')$. The former implies $s \xrightarrow{t_v}_{eI}$ violates LD$'$, so the first disjunct in the last line of I holds. The latter, Separation, and DetermInvis imply that either $s \xrightarrow{t_v}_{eI}$ violates LD$'$ or there exists s' such that $s \xrightarrow{t_v}_{eI} s'$ and $\mathrm{violR}(s')$, so one of the disjuncts in the last line of I holds.

(P2) Separation and DetermInvis imply that all transitions in sleep are visible, so t' is visible, so PureVis implies that t' does not access any object in $\mathcal{O}_{ld}$. Thus, a LD$'$-violating access by t_i to an object in $\mathcal{O}_{ld}$ cannot cause dependency between t and t'. Thus, such errors in computing sleep sets are impossible.

Now we show that I implies that a violation is found. Since $\mathcal{M}$ violates LD$'$, the disjunct $(\exists t \in f.T)$ of I holds for the first frame for DFS. Consider the execution defined by repeatedly following the transition t that witnesses the existential in I. The state space is finite and acyclic, so the second disjunct in the existential cannot hold indefinitely, so eventually the first disjunct in the existential holds, as desired.

(b) The proof of the reverse direction ($\Leftarrow$) of the "iff" is straightforward. For the forward direction ($\Rightarrow$), we suppose $\mathcal{M}$ violates LD$'$, and show that IF-SSS$_{\mathrm{cyc}}$ finds a violation. The proof that I is an invariant is the same as in part (a), except that we refer to Theorem 1(b) instead of Theorem 1(a). The proof that I implies a violation is found is the same as in part (a), except for two points. First, we consider a call to SSS$_{\mathrm{bnd}}$ with a depth bound large enough for the search to reach a violation of LD$'$. Second, the presence of cycles in the state space means that it is possible for the second disjunct in the existential in I to hold along an infinite path. We need to show that the search does not get stuck in a cycle and miss violations outside the cycle. This holds, because BoundedInvis implies that every cycle contains a visible state, and in a visible state, even if LD$'$ is violated, IF-SSS$_{\mathrm{cyc}}$ explores all enabled transitions except those in the sleep set. (Recall that sleep sets are computed correctly, even if LD$'$ is violated.) $\square$