Exploiting Object Escape and Locking Information in Partial-Order Reductions for Concurrent Object-Oriented Programs[*]

Matthew B. Dwyer John Hatcliff Robby
Venkatesh Prasad Ranganath
Department of Computing and Information Sciences, Kansas State University [†]

March 23, 2004

Abstract

Explicit-state model checking tools often incorporate partial-order reductions to reduce the number of system states explored (and thus the time and memory required) for verification. As model checking techniques are scaled up to software systems, it is important to develop and assess partial-order reduction strategies that are effective for addressing the complex structures found in software and for reducing the tremendous cost of model checking software systems.

In this paper, we consider a number of reduction strategies for model checking concurrent object-oriented software. We investigate a range of techniques that have been proposed in the literature, improve on those in several ways, and develop five novel reduction techniques that advance the state of the art in partial-order reduction for concurrent object-oriented systems. These reduction strategies are based on (a) detecting heap objects that are *thread-local* (i.e., can be accessed by a single thread) and (b) exploiting information about patterns of lock-acquisition and release in a program (building on previous work). We present empirical results that demonstrate upwards of a hundred fold reduction in both space and time over existing approaches to model checking concurrent Java programs. In addition to validating their effectiveness, we prove that the reductions preserve LTL$_{-X}$ properties and describe an implementation architecture that allows them to be easily incorporated into existing explicit-state software model checkers.

1 Introduction

State-space exploration techniques such as model checking verify a property ϕ of a concurrent system Σ by exploring all possible interleavings of thread transitions from Σ while looking for violations of ϕ. This exhaustive exploration is very expensive with respect to both the time and space required to store visited system states. In many cases, the exploration process searches paths that differ only in unimportant ways – e.g., in the intermediate states produced or in the ordering of transitions – that cannot be distinguished by the property ϕ. Exploring multiple paths that are indistinguishable by ϕ is

[*]This work was supported in part by the U.S. Army Research Office (DAAD190110564), by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), by NSF under grant CCR-0306607, by Rockwell-Collins, and by Intel Corporation (Grant 11462).

[†]234 Nichols Hall, Manhattan KS, 66506, USA.
`{dwyer,hatcliff,robby,rvprasad}@cis.ksu.edu`

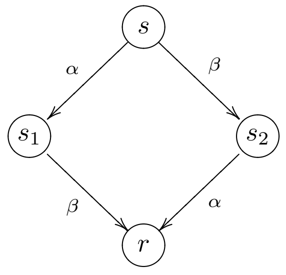

Figure 1: Independent transitions

not necessary for verification, and techniques called *partial-order reductions* (POR), e.g., [6, Chapter 10], have been widely applied to avoid this form of unnecessary exploration.

Figure 1 presents a state s with two outgoing transitions α and β. These transitions are *independent* in the sense that they *commute*: both paths $\alpha\beta$ and $\beta\alpha$ arrive at the same state r. If the property ϕ being verified cannot distinguish the intermediate states s_1 and s_2 nor the ordering of the transitions, then α and β are said to be *invisible* to ϕ, and it is safe to explore one path only. For example, α might represent the assignment of a constant value v_1 to variable x that is local to thread t_1, and β might represent the assignment of a constant value v_2 to variable y that is local to thread t_2. If the primitive propositions in ϕ do not refer to x and y, and if they do not refer to the control points associated with assignment states to x and y, then transitions are invisible to ϕ and the resulting sub-paths are *ϕ-equivalent*. Partial-order reduction strategies reduce both time and space for storing states by detecting independent transitions and by exploring a single representative sub-path (e.g., $\alpha\beta$) from a set of ϕ-equivalent sub-paths (e.g., $\{\alpha\beta, \beta\alpha\}$).

Recent work [7, 9, 33] has demonstrated the effectiveness of state-space exploration techniques for verifying temporal properties of concurrent object-oriented (OO) software systems. Exhaustive exploration of such systems is expensive due not only to the presence of concurrency, but also to the potentially large and complex heap structures that arise in OO systems. The semantics of OO languages, such as Java, require that heap-allocated data be treated by a concurrent thread as if other threads may access it, but Java programs may manipulate significant amounts of heap-allocated data that can be considered *local* to a thread. For conventional transition systems, strategies like partial-order reductions have proven effective in exploiting local data to significantly reduce cost of state-space search. In existing work on partial-order reductions, the transition independence conditions that enable reductions can be identified in static descriptions of transition system (e.g., in the Promela [20] model of a system) by a syntactic scan of the description (e.g., two transitions from different processes access program variables that are declared as locals in a thread [6, p. 157] as in the example above). However, independence of program actions is much more difficult to recognize in object-oriented systems where heap-allocated data are referenced through chains of dereferences, where aliasing makes it hard to determine ownership and visibility of objects, and where locking can enforce exclusive access to shared objects.

There is a rich body of work from the compiler community on reducing the run-time cost of heap-allocated data in programs. For example, recent work on object stack allocation [10] and synchronization removal [29] have identified a number of opportunities for exploiting the results of *escape analysis* to improve performance by eliminating object dereferencing, reducing garbage collection, and eliminating Java monitor statements. Escape analyses can be used to determine that an object instance is only manipulated by a single thread and thus can be treated as local to that thread. Those analyses can also be used to enable partial-order reductions for model checking OO programs. However, as originally designed, they make cost-benefit trade-offs that are appropriate for compilation speed and performance optimization, but are sub-optimal for model checking reductions. Specifically, the lack of context and object-sensitivity in many existing escape analyses means they will miss opportunities to identify objects as thread local. More fundamentally, for an object to be classified as thread-local, these analyses require *all* accesses to the object to come from a single thread. Exploiting information about the *run time* state of the program can enable significantly more precise analyses, as evidenced for a variety of analyses in [17]. For example, in our context, although an object is accessed by more than one thread over its lifetime, run time information can determine that it can still be classified as thread-local if no more than one thread can access the object in any given state encountered in the program execution. With this type of on-the-fly analysis and optimization, one must balance the run-time overhead against the potential benefit. For model checking, the size of the state-space is the dominant factor in the cost of analysis. Exponential state-space growth means that the potential benefit of precise on-the-fly analysis will often be significant and more than outweigh any increase in the analysis cost.

Another line of work has studied the extent to which the locking disciplines used in implementing concurrent object-oriented programs can be used to drive state-space reductions. Inspired by the work of Lipton [26], recent work [14, 31] exploits locking information to identify regions of program execution that can safely be modeled ass executing atomically. These methods rely on programmer annotations to identify the objects in the program that are governed by a specific locking discipline. Once annotated, the program can be analyzed on-the-fly to assure that the discipline is maintained. If the discipline is maintained, atomicity can be exploited to reduce the state-space during model checking [31].

In this paper, we propose a variety of partial-order reduction strategies for model checking concurrent object-oriented software that are based on detecting heap objects that are *thread-local* (i.e., can be accessed by a single thread only). Transitions that access such objects are independent, and identifying these allows the model checker to avoid exploring unnecessary interleavings. Locality of actions can be determined using both (a) analysis of the program heap structure to determine object escape information and (b) analysis of the locking patterns used to coordinate object access. We define and evaluate both a static and dynamic technique for approach (a) and several strategies for approach (b) as well as combinations of these techniques. Furthermore, we define and evaluate an additional atomicity-based reduction that can be combined with any of the preceding techniques.

We have implemented these reduction techniques in Bogor [27] [1] – a novel model checker that is the core of the next generation of the Bandera tool set for model checking concurrent Java systems [7]. Bogor is an extensible model checker designed to support checking of OO software with state-of-the-art strategies for heap-state representations and for achieving heap [22] and thread symmetry reductions [3]. This allows us to justify the effectiveness of our reductions empirically and to directly compare them with existing approaches to software model checking.

We begin our presentation in Section 2 with a collection of examples that provides the intuition behind our reduction strategies. Section 3 recalls the basic definitions of

[1]http://bogor.projects.cis.ksu.edu

the *ample set* [6] partial-order reduction framework. The sections that follow provide the contributions of our paper.

- (Section 4) We recall how the traditional notion of transition independence (which requires transitions to be independent for all states) to can be generalized to the notion of *conditional independence* [23] (which allows transitions to be independent in some states but not in others). We believe that this generalization to *state-sensitive independence* is necessary to treat effectively the dynamic nature of object-oriented software systems. Appendix B presents modifications to the correctness proofs for the ample set framework in [6] that account for the generalization to conditional independence.
- (Section 5) We describe how Java transition systems and independence properties for heap-based data can be cast in the ample sets partial-order reduction framework using both *static* and *dynamic* escape analysis. For static escape analysis, we summarize how Ruf's escape analysis [29] developed for removal of unnecessary synchronization can be adapted to provide information about object-sharing suitable for identifying independent transitions and driving partial-order reductions. For dynamic escape analysis, we give a novel technique for carrying out escape analysis on-the-fly during model checking that substantially improves upon the optimization enabled by the static escape analysis.
- (Section 6) We recall Stoller's locking-discipline based reduction approach [31] and then present alternate lock-based strategies that also incorporate dynamic thread-locality information. We note that neither our approach nor Stoller's subsumes the other, and then we present a strategy that combines Stoller's strategy with the best of our alternate strategies.
- (Section 7) We describe how to incorporate additional optimizations that avoid storage of unnecessary intermediate states – essentially by dynamically determining sequences of transitions such that each of these sequences can be expanded in a single atomic step.
- (Section 8) We report on experimental studies that show the relative performance of these reduction techniques and that in combination they can provide multiple order of magnitude state-space reductions over that provided by existing highly-optimized software model checking frameworks.

Section 9 presents related work, and Section 10 concludes. Appendix A presents the static analysis that we used to detect thread-local objects. Appendix B presents modifications to the correctness proofs for the ample set framework in [6] that account for the generalization to conditional independence. Appendix C presents the proofs that the thread-local and lock-based reductions we propose satisfy the conditions of the ample set framework. Appendix D recasts Stoller's reduction approach into the ample set framework and describes our implementation of the strategy, which we use as a point of comparison in our experimental studies.

Our approach and results are described in the context of Bogor, however, we believe that the relative simplicity of integrating the approach with explicit-state model checking algorithms and the potential for significant reductions that it brings suggest that other software model checkers should incorporate similar ideas.

(a) Independent Work (version 1)

```
public class IndependentWork1 {
  public static void main
      (String[] args) {
    new Process().start();
    new Process().start();
  }
}
class Node {
  int x; Node next;
  public Node(int x) {
    this.x = x;
  }
}
class Process extends Thread {
  Node head;
  public void run() {
    head = new Node(0);
    Node temp = head;
    for (int i = 1; i < 10; i++) {
      temp.next = new Node(i);
      temp = temp.next;
    }
    while (head != null) {
      head.x++;
      head = head.next;
    }
  }
}
```

(b) Independent Work (version 2)

```
public class IndependentWork2 {
  public static void main
      (String[] args) {
    new Process().start();
    new Process().start();
  }
}
class Node {
  int x; Node next;
  public Node(int x) {
    this.x = x;
  }
}
class Process extends Thread {
  Node head;
  public Process() {
    head = new Node(0);
    Node temp = head;
    for (int i = 1; i<10; i++) {
      temp.next = new Node(i);
      temp = temp.next;
    }
  }
  public void run() {
    while (head != null) {
      head.x++;
      head = head.next;
    }
  }
}
```

Figure 2: Java examples illustrating static vs. dynamic escape analysis

2 Motivating Examples

In the following subsections, we give examples to provide the intuition behind our reduction strategies.

2.1 Detecting independence using object escape information

Figure 2 presents two Java programs that we will use to illustrate why we want to consider both static and dynamic detection of thread-local objects. Both programs are similar: the main thread creates two `Process` threads, and for each `Process`, a list is built and then traversed while incrementing each list node's `x` field.

The difference: in Figure 2(a), the list is both built and traversed in the `run` method of the `Process` thread; in Figure 2(b), the list is built in the `Process` object's constructor, but the list is traversed in the `run` method of the `Process` thread. We would like an analysis to tell us if the lists in these examples are thread-local.

A conventional static escape analysis would establish the fact that the list does not escape the `Process` thread in Figure 2(a), and thus is thread-local. This information allows a model checker to apply partial-order reductions and avoid exploring interleavings with other threads at any transitions that access thread-local data — since the actions of other threads cannot interfere with the thread-local data, it is safe to explore only a path where there are no context switches, i.e., the current thread continues to execute. In fact, for Independent Work (version 1), static analysis is able to determine that the entire `run` method of the `Process` thread may execute as a sequence of thread-local transitions without considering any interleavings. This ends up reducing the number of states explored by a factor of twelve.

In contrast, the list in Figure 2(b) actually gets manipulated by two threads: the main thread executes the `Process` constructor (and thus builds the list), but the `Process` thread traverses the list. Thus, a conventional static escape analysis would tell us that the list does escape the control of both the main and the `Process` thread, and based on that information we could not consider the list to be thread-local. Thus, no reduction in the number of states explored is possible based on conventional static escape analysis information.

The dynamic escape analysis that we have implemented checks for the thread-local condition after each execution step. It clearly detects that the list of Figure 2(a) is thread-local at every program state. Moreover, it reveals an interesting fact about the list of Figure 2(b): the list is never reachable by more than one thread at any state in the program. When the main thread executes the `Process` constructor, it is the only thread that can reach the list. Furthermore, when the constructor's execution is complete, the main thread cannot reach the list anymore. Thus, when a `Process` thread is started, it is the only thread that can reach the list.[2] Thus, we can consider the list thread-local at all program states of Figure 2(b). This information allows the model checker to apply reductions and to process the `run` method of the `Process` as a sequence of thread-local transitions without considering any interleavings. We will demonstrate that the cost of dynamic escape analysis is negligible, and is almost always dramatically dominated by cost savings in number of states explored/stored, etc.

There are many other common coding patterns where dynamic escape analysis provides significant optimization of model checking costs. For example, many objects are often thread-local at the beginning of their life-times (and thus partial-order reductions can be applied during this portion of the execution) and then only later do they escape. In Section 5, we give a more rigorous explanation of the notions of thread-local object and state-sensitive transition independence that enable effective optimizations based on partial-order reductions.

2.2 Detecting independence using object locking information

Figure 3 presents a bounded-buffer implementation of a first-in first-out (FIFO) data structure that we use to illustrate the usefulness of lock-based partial-order reductions in addition to the thread-local object-based reductions of the previous example. There are two threads that are exchanging messages (objects) via the two instances of bounded-buffers. A thread can only take a message from a non-empty buffer; otherwise, it will block. Conversely, a thread can only add a message to a non-full buffer.

Because of the high degree of object-sharing in this example, escape-based information does not yield dramatic reductions. Conventional static escape analysis does not reveal any objects that are thread-local. Dynamic escape analysis reveals that the `BoundedBuffer` objects are thread-local during the execution of the associated constructors, and reductions based on this information reduces the number of explored states from 4641 to 3713. However, considering locking information and state storage heuristics can reduce the states stored by a factor of 57 (bringing the number of stored states down to 81).

The `BoundedBuffer` class is *thread-safe*[3], i.e., all accesses to its instance are mutually exclusive. The mutual exclusion is guaranteed by using the Java `synchronize` modifier on all of its methods. This style of implementation is commonly used, e.g., the `java.util.Vector` class and the `java.util.Hashtable` class in the Java Collection

[2]This is due to the fact that we optimize the model by using information from a dead variable analysis. After each `Process` is started, the JVM stack element that holds a reference of the `Process` object is considered dead.

[3]a term that is usually used in the Java community for synchronized classes

```
public class BBDriver {
  public static
      void main(String[] args) {
    BoundedBuffer b1 =
        new BoundedBuffer(3);
    BoundedBuffer b2 =
        new BoundedBuffer(3);

    b1.add(new Object());
    b1.add(new Object());

    (new InOut(b1, b2)).start();
    (new InOut(b2, b1)).start();
  }
}

class InOut extends Thread {
  BoundedBuffer in;
  BoundedBuffer out;

  public InOut(BoundedBuffer in,
               BoundedBuffer out) {
    this.in = in;
    this.out = out;
  }

  public void run() {
    Object tmp;

    while (true) {
      tmp = in.take();
      out.add(tmp);
    }
  }
}
```

```
class BoundedBuffer {
  protected Object[] buffer;
  protected int bound;
  protected int head;
  protected int tail;

  public BoundedBuffer(int b) {
    bound = b;
    buffer = new Object[bound];
    head = 0;
    tail = bound - 1;
  }

  public synchronized
      void add(Object o) {
    while (tail == head) {
      try {
        wait();
      } catch (InterruptedException e) {
      }
    }
    buffer[head] = o;
    head = (head + 1) % bound;
    notifyAll();
  }

  public synchronized Object take() {
    while (head == ((tail+1) % bound)) {
      try {
        wait();
      } catch (InterruptedException e) {
      }
    }
    tail = (tail + 1) % bound;
    notifyAll();
    return buffer[tail];
  }
}
```

Figure 3: BoundedBuffer example illustrating locking-discipline

Framework. The `java.lang.Object.wait()` method and the `java.lang.Object.notifyAll()` method are used to implement the blocking behavior of the bounded-buffer.

Notice that all accesses to the fields of a bounded-buffer object take place through its (synchronized) methods. Thus, when any field of a bounded-buffer is accessed by a thread t, this means that t holds the lock of the bounded-buffer object. This disciplined use of locks guarantees that no other threads can access the fields until the bounded-buffer's lock is released. We call an object that uses this pattern of locking *self-locking* since the object's fields are protected by acquiring the lock associated with the object. Based on this information, many interleavings can be safely avoided when exploring the state-space of this system. For example, there is no need to consider any interleaving of thread actions before the assignments to `head` in the `add` method and to `tail` in the `take` method since locking guarantees that no other threads would be able to execute statements that change the values of the `head` and `tail` fields.

Furthermore, notice also that all accesses to the array `buffer` that provides the storage for the buffer occur only in the bounded-buffer (synchronized) methods, because the array does not escape the bounded-buffer. Specifically, all access paths to the array are dominated by a `BoundedBuffer` object – no other object holds a direct reference to the array. Thus, a `BoundedBuffer` object lock must be acquired before the array is accessed. We refer to this pattern of locking as *monitor-based* locking discipline. In addition to the reductions due to self-locking above, this means that there is no need to consider interleavings before the assignments to the `buffer` arrays in the `add` and `take` methods.

For both the `BoundedBuffer` object fields and buffer array, threads follow a locking discipline to access data – there is always a specific lock (whether in the object itself our in an associated object) that must be acquired before accessing the data. In Stoller's lock-based reductions [31], users annotate the objects should satisfy this general notion of locking discipline. Then, the model checking algorithm with partial-order reductions uses this information to avoid interleavings as described above while simultaneously checking that the locking discipline is not violated.

While Stoller's approach covers most cases that yield opportunities for lock-based reductions, there are several common locking idioms that dynamically re-associate locks with a set of objects being protected. Some of these examples can be handled by our dominated/monitor based approach but not by Stoller's method. On the other hand, there are also examples where there is no link via references from a protecting object to the object being protected, and in these cases, Stoller's method enables reductions whereas ours does not. Finally, we have observed that combining lock-based reductions with thread-local-based reductions almost always yields some improvements over either type of strategy alone, and further combining these with heuristics for avoiding state storage yields dramatic reductions. Furthermore, we discuss how Stoller's approach can be combined with ours. In the latter sections of this paper, we attempt to clarify the relationships between these approaches, and we aim to determine which strategies should be considered for implementation in explicit-state software model checkers.

2.3 Heuristics to avoid state storage

In addition to reducing interleavings based on independence information derived from escape analysis and locking discipline, we propose variants to our algorithms that also apply heuristics to avoid storing states produced when there are no interleavings. Using these additional heuristics, for example, the number of states explored for Independent Work (version 2) drops from 6770 states using algorithm that does not consider escape information to a total of 2 states.

3 Background

3.1 State transition systems

We present our reduction strategies using the well-known partial-order reduction framework based on *ample sets* as presented in [6]. The presentation in [6] phrases systems in terms of *state transition systems* (Kripke structures with slight modifications). Transitions and state structures in Java systems are substantially more complex than those found in traditional model checking since Java systems include notions such as heap-allocated data, dynamic thread creation, exceptions, etc. Nevertheless, previous formal presentations on partial-order reductions for Java systems, e.g., the work of Stoller [31], provide strategies for phrasing Java systems (with a few simplifying constraints) as transition systems. Accordingly, in our work we will assume the existence of strategies for phrasing Java semantics in the form of state transition systems. We believe that the conditions on the state structure and transitions that are needed to describe the enabling/disabling of our reductions are simple enough to be accurately described without exposing significant details of the Java execution model in the formal structures we use for presenting our ideas.

A *state transition system* [6] Σ is a quadruple (S, T, S_0, L) with a set of states S, a set of transitions T such that for each $\alpha \in T, \alpha \subseteq S \times S$, a set of initial states S_0, and a labeling function L that maps a state s to a set of primitive propositions that are true at s.

For the Java systems that we consider, each state s holds the stack frames for each thread (program counters and local variables for each stack frame), global variables (i.e., static class fields), and a representation of the heap. Intuitively, each $\alpha \in T$ represents a statement or step (e.g., execution of a bytecode) that can be taken by a particular thread t. In general, α is defined on multiple "input states", since the transition may be carried out, e.g., not only in a state s but also in another state s' that only differs from s in that it presents the result of another thread t' performing a transition on s.

For a transition $\alpha \in T$, we say that α is *enabled* in a state s if there is a state s' such that $\alpha(s, s')$ holds. Otherwise, α is disabled in s. Intuitively, a transition α for a thread t may be disabled if the program counter for t is not at the bytecode represented by α, if α represents an entermonitor bytecode step that is blocked waiting to acquire a lock, or if t is currently in the *wait set* of an object o (i.e., t has surrendered the lock of o and put itself to sleep by calling `wait` on o). The set of transitions enabled in s is $\mathit{enabled}(s)$, and the set of transitions enabled in s belonging to thread t is $\mathit{enabled}(s, t)$. We denote the program counter of a thread t in a state s by $\mathit{pc}(s, t)$. We write $\mathit{current}(s, t)$ for the set of transitions associated the current control point $\mathit{pc}(s, t)$ of thread t (this set will include $\mathit{enabled}(s, t)$ as well as any transitions of t at $\mathit{pc}(s, t)$ that may be disabled). Also, $\mathit{current}(s)$ represents the union of current transitions at s for all active threads.

A transition is *deterministic* if for every state s there is at most one state s' such that $\alpha(s, s')$. When α is deterministic, we write $s' = \alpha(s)$ instead of $\alpha(s, s')$. Following [6], we will only consider deterministic transitions. Note that this does *not* eliminate non-determinism in a thread (e.g., as might result from abstraction) – the non-determinism is simply represented by multiple enabled transitions for a thread. For each transition α, we assume that we can determine, among other things, a unique identifier for a thread t that executes α, and the set of variables or heap-allocated objects that are read or written by α. A *path* π from a state s is a finite or infinite sequence such that $\pi = s_0 \overset{\alpha_0}{\rightarrow} s_1 \overset{\alpha_1}{\rightarrow} \ldots$ such that $s = s_0$ and for every i, $\alpha_i(s_i) = s_{i+1}$.

In order to simplify the presentation of some of the lock-based reduction algorithms described in the latter part of the paper, we make the following assumption.

Assumption 1 *We assume that at each control-point for a thread t and for any state s, if we take O to be the set of objects accessed in s by all transitions of t that are current for that control point, then O contains at most one object. This reflects the granularity of Java transitions at the bytecode level (i.e., each Java bytecode accesses at most one object).*

3.2 Partial-order reductions

POR techniques based on ample sets can be summarized as follows. For each state s where $\mathit{enabled}(s) = \{\alpha_0, \alpha_1, \ldots, \alpha_m, \beta_0, \beta_1, \ldots, \beta_n\}$, one attempts to determine a subset $\mathit{ample}(s) = \{\alpha_0, \alpha_1, \ldots, \alpha_m\}$ of $\mathit{enabled}(s)$ such that it is sufficient to only explore paths out of s that begin with transitions α_i from $\mathit{ample}(s)$. The fact that exploring these paths alone is sufficient can be established by showing that each of the other possible paths out of s (i.e., those beginning with one or more β_j) is equivalent (with respect to the property ϕ being checked) to one of the explored paths (i.e., those beginning with one or more α_i). Showing this equivalence relies on establishing three basic notions: (1) showing that ample transitions α_i are *independent* of (i.e., commute with) non-ample transitions β_j, which allows the α_i to be "moved to the front" of paths considered from s (the α_i are allowed to be dependent on each other), (2) showing that the relative ordering of α_i with respect to β_j and the intermediate states generated by α_i are *invisible* to the property ϕ, and (3) showing that no transition β with an ϕ-observable effect is delayed forever by a cycle in the ample transitions α_i.

We now consider both *independence* and *invisibility* in more detail, and then proceed with an outline of the ample sets state-exploration algorithm and conditions that guarantee that strategies for defining *ample* result in correct reductions.

3.2.1 Independence

In presentations of POR [6, p. 144], the notion of independent transition is usually expressed using an *independence relation* I between transitions. Specifically, $(\alpha, \beta) \in I$ if *for all states* s the execution order of α and β can be interchanged when they are both enabled in s. An independence relation is required to be symmetric and anti-reflexive and to satisfy the two conditions in the definition below. The first ensures that one transition does not disable the other (note the 'symmetric' requirement on I ensures that this condition only needs to be stated in one direction). The second ensures that executing the transitions in any order will lead to the same state.

Definition 1 (independence conditions) *For each* $(\alpha, \beta) \in I$*, and for all states* $s \in S$*:*

- Preservation of Enabledness: *If* $\alpha, \beta \in \mathit{enabled}(s)$ *then* $\alpha \in \mathit{enabled}(\beta(s))$,
- Commutativity: *If* $\alpha, \beta \in \mathit{enabled}(s)$ *then* $\alpha(\beta(s)) = \beta(\alpha(s))$.

It is convenient to refer to the *dependence relation* D that is defined as the complement of I.

Note that if a thread has multiple enabled transitions $\{\alpha_1, \ldots, \alpha_n\}$ at a particular control-point, this definition implies that those transitions cannot be independent of each other. This follows from the fact that if one of the α_i is executed, it will change the program counter for that thread, and will thus disable the remaining transitions (violating the first condition of the definition). Heuristics based on this observation play an important role in the choice of transitions to consider in partial-order reduction strategies.

3.2.2 Invisibility

The correctness of POR depends not only on the ability of certain transitions to commute with another, but also on the inability of the property ϕ being checked to distinguish the ordering of commuting transitions or the intermediate states generated by them. Let $L : S \rightarrow 2^{AP}$ be the function from a system Σ that labels each state with a set of atomic propositions. A transition $\alpha \in T$ is *invisible* with respect to a set of propositions $AP' \subseteq AP$ if α does not change the value of the propositional variables in AP', i.e., if for each pair of states $s, s' \in S$ such that $s' = \alpha(s)$, $L(s) \cap AP' = L(s') \cap AP'$. A transition is *visible* if it is not invisible [6, p. 144].

The well-studied property of *stuttering equivalence* can be used to establish that if a path π_2 can be obtained from path π_1 by commuting, adding, or removing invisible transitions, then no LTL$_{-X}$ (i.e., LTL without the 'next state' operator) formula can distinguish π_1 and π_2 [6, p. 146]. We do not present the details of stuttering equivalence here. Instead, we simply note that the conditions in the following section guarantee that only invisible transitions are included in *ample*(s) and that any path omitted as a result of following the POR strategy is stuttering equivalent to some path that is included in the reduced system. Thus, preservation of verification and refutation of LTL$_{-X}$ properties is guaranteed.

```
1 $seen := \{s_0\}$
2 $pushStack(s_0)$
3 $DFS(s_0)$

$DFS(s)$
4 $workSet(s) := ample(s)$
5 while $workSet(s)$ is not empty
6     let $\alpha \in workSet(s)$
7     $workSet(s) := workSet(s) \setminus \{\alpha\}$
8     $s' := \alpha(s)$
9     if $s' \notin seen$ then
10        $seen := seen \cup \{s'\}$
11        $pushStack(s')$
12        $DFS(s')$
13        $popStack()$
end $DFS$
```

Algorithm 1: Depth-first search with partial-order reduction [6, p. 143]

We previously used stuttering equivalence to prove the correctness of Bandera's approach to LTL$_{-X}$ property-preserving slicing [18], and notions of visibility/invisibility defined there along with their correctness justifications carry over to the present setting. Bandera's specification language [8] includes two forms of primitive propositions: location predicates and data state predicates. A location predicate P_l for control point l holds when a thread is at l. Thus, both the transition of a thread into l is visible (because the proposition value shifts from *false* to *true*) and the transition of thread out of l is visible (because the proposition shifts from *true* to *false*) – all other transitions are invisible to P_l. A data state predicate that references a set of memory cells holds when the values of those cells satisfy the predicate. Thus, any transition that modifies one or more of those cells is visible, and all others are invisible. In each of the two types of predicates, Bandera uses a simple static analysis to form a conservative approximation of the set of visible/invisible transitions.

3.3 Ample set conditions

Algorithm 1 presents the familiar depth-first search algorithm extended to include the ample-sets-based partial-order reduction [6, p. 143]. As we have noted, the essence of the idea is, for a given state s, only explore transitions from s that are in $ample(s) \subseteq enabled(s)$. When $ample(s) = enabled(s)$, s is said to be *fully expanded.*

Building on the notions of independence and invisibility above, [6, pp. 147–151] gives four conditions that $ample(s)$ must satisfy to achieve correct reductions when checking LTL$_{-X}$.

Condition **C0** requires that a state has no successor in the full search iff it has no successors in the reduced search. This simple condition is satisfied by any natural approach to computing $ample(s)$.

C0: For all $s \in S$, $ample(s) = \emptyset$ if and only if $enabled(s) = \emptyset$.

Of the four conditions, **C1** is the most interesting with respect to justifying our conditions for thread-local and lock-based independence. As shown in [6, pp. 148], it guarantees that every transition $\beta \in (enabled(s) - ample(s))$ is independent of the transitions in $ample(s)$. This (along with the subsequent conditions) enables one to show that every path π_f from s in a full (non-reduced) exploration can be shown equivalent to a path π_r in the reduced exploration by commuting independent transitions in π_f to obtain π_r.

C1: For all $s \in S$, every path that starts at s has one of the following forms:

- the path has a prefix $\beta_1\beta_2 \ldots \beta_m\alpha$ where $m \geq 0$ and $\alpha \in \mathit{ample}(s)$ and each β_i is not in $\mathit{ample}(s)$ and is independent of all transitions in $\mathit{ample}(s)$, or
- the path is an infinite sequence of transitions $\beta_1\beta_2 \ldots$ where each β_i is independent of all transitions in $\mathit{ample}(s)$.

The intuition behind condition **C2** is simple: if we omit some paths that contain different orderings of the transitions in $\mathit{ample}(s)$, then all transitions in $\mathit{ample}(s)$ should be invisible.

C2: For all $s \in S$, if s is not fully expanded, then every $\alpha \in \mathit{ample}(s)$ is invisible.

Condition **C3** ensures that an enabled transition β is never infinitely postponed due to a cycle of ample transitions.

C3: A cycle (in the reduced state graph) is not allowed if it contains a state in which some transition β is enabled, but is never included in $\mathit{ample}(s)$ for any state s on the cycle.

A sufficient condition for **C3** is that at least one state along each cycle is fully expanded [6, p. 155].

[6, Chapter 10] gives several strategies for implementing $\mathit{ample}(s)$ so that the conditions above are satisfied. We will refer to these strategies when outlining our implementation in subsequent sections.

4 State-Sensitive Independence

In conventional systems considered for model checking, requiring that the independence condition for (α, β) hold for *all* states (as in Definition 1 of the previous section) is reasonable since independence is usually detected by doing a *static* scan of the system description as written in, e.g., Promela, to detect that, e.g., two transitions from different processes are not interacting with the same channel or the same program variable [6, p. 157].

In our setting, due to the changing structure of the heap, it is useful to weaken the notion of independence relation to allow a pair of transitions to be independent in some states and dependent in others. For example, there will be situations where (a) a thread t is manipulating a non-escaped object o at control point p (and the manipulating transition will be independent of the transitions of other threads), (b) subsequent execution causes o to escape so that it can be accessed by other threads, and (c) control of t returns to p and now the transition at p is not guaranteed independent since o is now visible from other threads. Thus, we follow the notion of *conditional independence* introduced in [23] and modify the notion of independence relation to allow *state-sensitive independence* conditions by considering a family of relations $I_s \subseteq T \times T$ indexed by states $s \in S$. We write $I_s(\alpha)$ for the set of transitions related to α by I_s. Following the criteria for conventional independence relations [6, p. 144], we require each I_s to be symmetric and anti-reflexive and to satisfy the following three conditions below. The first two conditions are identical to those of Definition 1. The third condition, which we have added to incorporate conditional independence into the ample set reduction framework, ensures that the independence property between two transitions α and β_1 is not dissolved by a transition β_0 from another thread that is independent to α (if β_0 and β_1 are from the same thread, they are not independent of each other, and thus one is allowed to dissolve independence properties of the other).

Definition 2 (state-sensitive independence conditions) *For each state $s \in S$, and for each $(\alpha, \beta) \in I_s$, thread$(\alpha) \neq$ thread(β) and*

- Preservation of Enabledness: *If $\alpha, \beta \in$ enabled(s) then $\alpha \in$ enabled$(\beta(s))$.*
- Commutativity: *If $\alpha, \beta \in$ enabled(s) then $\alpha(\beta(s)) = \beta(\alpha(s))$.*
- Preservation of Independence:
  If $\{\beta_0, \beta_1\} \subseteq I_s(\alpha)$ and thread$(\beta_0) \neq$ thread(β_1) and $s' = \beta_0(s)$, then $\{\beta_1\} \subseteq I_{s'}(\alpha)$.

Of the conditions **C0 - C1** for ample set corrections, only **C1** refers the concept of independence. Accordingly, we need to modify it to account for the notion of state-sensitive independence (the underlined text indicates the additions to the condition).

C1: For all $s \in S$, every path that starts at s has one of the following forms:

- the path has a prefix $\beta_1\beta_2 \ldots \beta_m\alpha$ where $m \geq 0$ and $\alpha \in$ *ample*(s) and each β_i is not in *ample*(s) and is independent $\underline{\text{at } s}$ of all transitions in *ample*(s), or
- the path is an infinite sequence of transitions $\beta_1\beta_2 \ldots$ where each β_i is independent $\underline{\text{at } s}$ of all transitions in *ample*(s).

Note that the [Preservation of Independence] condition on I_s ensures that transitions in *ample*(s) continue to be independent of the β_i at each of the intermediate states generated in the paths given in Condition **C1** of the preceding section. Appendix B gives a proof of this claim, along with modifications to proofs of [6] that justify that this modified version of **C1** together with the notion of state-sensitive independence give rise to correct reductions.

5 Partial-Order Reduction for Unshared Objects using Escape Analyses

In this section, we present our approach for detecting independent transitions using static and dynamic escape analysis. We first introduce some terminology related to Java-specific transition systems. Memory cells in Java can be partitioned into three categories: local (these correspond to local variables of methods, which are stack-allocated), global (these correspond to static fields of classes), and heap (these correspond to explicit and implicit instance fields of dynamically allocated created objects and array components). Note that implicit instance data includes data structures to represent locks, waiting thread sets, blocked thread sets, etc. for each object instance. A transition is said to *access* a memory cell if it performs one of the following actions associated with particular Java bytecodes [25]. Local accesses include a number of different bytecodes associated with accessing local slots in the current stack frame (e.g., iload, istore). Global (i.e., static field) accesses are reads from a static field (getstatic) and stores to a static field (putstatic). Heap (i.e., object instance) accesses are reads from an instance field (getfield), stores to an instance field (putfield), locking (entermonitor), unlocking (exitmonitor), and invocations of `wait`, `notify`, `notifyAll` methods of an object (`java.lang.Object`, note that arrays in Java are also objects). Array accesses (also included in heap accesses) are reads from an array element (bytecodes $x$aload) and stores to an array index (bytecodes $x$astore), where x is either b, c, s, i, l, f, d, or a.

To reason about whether thread transitions can fail to be independent due to accesses to heap memory cells that are visible by multiple threads, we will conservatively approximate *visibility* by *reachability along object references.* First, the *reference-type memory cells* of an object o include all static fields, instance fields, and array cells (if o is an array) of non-primitive type. An object o is *reachable from a memory cell* (i.e., a local variable, an array element, an instance field, or a static field) if and only if the cell holds the reference for o or if the cell holds the reference for an object o' and o is reachable from any of o''s reference-type memory cells. An object o is *reachable from a thread* t at a certain state s if and only if there exists a stack frame of t in s that holds a local variable v such that o is reachable from v (note that the stack frames include the `this` variable, which will allow the thread to reach the instance fields of the thread object associated with the thread), or if o is reachable from a static field of any class. Note approximating 'visibility' by reachability as defined above is conservative in several ways. For example, a lock by thread t on an object o prevents another thread t' that also synchronizes on o from accessing o's fields. In addition, certain instance fields may be reachable but never actually touched by a particular thread.

A heap-allocated object o is said to be *thread-local with respect to thread* t *at state* s if o is reachable only from thread t. A transition $\alpha \in$ *current*(s,t) is said to be *thread-local at* s if it does not access a global memory cell (static field), or if all of its accesses are either to local data or to objects that are thread-local to t at s.

Based on the notion of "thread-local", which we will detect by both static and dynamic analyses in the following sections, we can now define an independence relation that will be the key element of the correctness argument for our reduction strategy.

Definition 3 (thread-local transition independence) *Let* $s \in S$*,* t_1 *and* t_2 *be threads,* $\alpha \in$ *current*(s,t_1)*, and* β *be any transition of* t_2*. Then both* $(\alpha,\beta) \in I_s^{tl}$ *and* $(\beta,\alpha) \in I_s^{tl}$ *hold iff* $t_1 \neq t_2$ *and* α *is thread-local at* s*.*

We now argue that our independence relation satisfies the three criteria for independence relations given in Definition 2. Let $(\alpha,\beta) \in I_s^{tl}$. For the [Preservation of Enabledness] condition, note that if $\alpha,\beta \in$ *enabled*(s) the only situation where β could disable α is one in which α corresponds to an entermonitor instruction on object o (α is ready to grab o's lock), and β also corresponds to entermonitor on o (β actually acquires o's lock). However, this cannot be the case since α is thread-local and thus o is not reachable from any other thread. For all other cases of disabledness (i.e., t_1 is in a wait set, t_1's program counter does not match α, etc.), it is t_1 itself that must move to shift into a disabled state.

For the [Commutativity] condition, it is clear that if α can only access memory cells that are thread-local to t_1, then both $\alpha(\beta(s))$ and $\beta(\alpha(s))$ yield states s_1 and s_2 (respectively) that cannot be distinguished by subsequent transitions. However, it is not obvious that $s_1 = s_2$. For example, consider that if both α and β correspond to allocations of objects o_a and o_b, respectively, then $\alpha(\beta(s))$ and $\beta(\alpha(s))$ could result in two different heap shapes (in one, o_a could be positioned first in the memory address space of the heap, while o_b could be positioned first in the other). The fact that $s_1 = s_2$ in our setting relies on the fact that Bogor's heap representation is *canonical* in the sense that any two program states that have observationally equivalent heaps (i.e., heaps that cannot be distinguished by any program context) are represented by a single canonical structure in the model checker [28].

The satisfaction of the [Preservation of Independence] condition relies on the fact that if a transition α of thread t is thread-local due to the fact that it is manipulating an object o that is local to t, then α can lose its thread-local status only if t itself takes an action to cause o to escape (i.e., if o is local to t, no other threads can make o escape – only t-actions can). Appendix C.1.1 presents a more rigorous justification of this condition.

5.1 Thread-Locality by Static Analysis

Quite a few static analyses [29, 5] have been proposed to determine thread-locality of objects via *escape analysis* in languages such as Java. In general, the results of an escape analysis are used to remove unnecessary synchronization and stack allocation of objects which in turn improves runtime performance.

To statically detect thread-local objects, we use an adaptation of Erik Ruf's escape analysis [29]. Ruf's analysis detects objects that are guaranteed to be *locked* by a single thread only. Any synchronization statements that are only applied to such objects are unnecessary and can safely be removed from the program. We adapt Ruf's analysis to obtain a variant that detects objects that are guaranteed to be *accessed* by a single thread only.

The results of the analysis are available via an interface that allows one to query each variable (field or local) to determine if that variable always refers to objects that are thread-local for the lifetime of the program execution. A transition in the model is annotated as thread-local if calls to the analysis interface indicate that all variables accessed in the transition refer to only thread-local objects. We write *static-thread-local*(α) to denote the fact that transition α such an annotation.

Given this static escape information that allows us to detect thread-local objects and independent transitions, we now explain our approach for constructing the ample sets. Algorithm 1 (along with the ample set conditions **C0**–**C3**) reveal that it is always safe to include the entire set of currently enabled transitions in *ample*(s). The main idea is to improve on this by including only invisible thread-local transitions in *ample*(s).

We adopt the strategy of [6, Section 10.5.2,p. 157], which suggests that one considers the set *enabled*(s, t_i) of transitions enabled in s for some thread t_i as a candidate for *ample*(s). The rationale for this is as follows. Recall that we explained under Definition 1 that enabled transitions from the same control point of a thread cannot be independent. Also, the intuition behind ample set condition **C1** reveals that that transitions chosen for *ample*(s) should be independent of all other transitions omitted from *ample*(s). Therefore, when choosing candidates for *ample*(s), either all enabled transitions from thread t at s must be included in *ample*(s) or all must be excluded.

Algorithm 2 presents the algorithm to compute the ample set. Following the discussion in the previous paragraph, the algorithm considers as possible values of *ample*(s) the enabled transitions of each thread t. If the *ample* function finds a thread whose transitions satisfies *checkC1*, *checkC2*, and *checkC3*, then the set of enabled transitions of that thread is returned. If no such set is found, the result *ample*(s) is defined to be set of enabled transitions from all threads (i.e., the state is fully expanded).

The function *checkC1* checks to see if *all* transitions of thread t in s are classified as thread-local by static escape analysis. Note that we check *all* transitions of t for thread-locality instead of just the enabled transitions of t. This is necessary to cover the case where a thread has enabled transitions that are all thread-local but also has one or more disabled transitions β_i that are not thread-local. In such a case, it is possible that another thread t' can execute while t remains at its current control point and t' may enable t's disabled (and non-dependent) transitions. This can lead to a violation of **C1**: an execution trace exists in the unreduced system where a β_i that is not independent of

```
checkC1($s, t$)
1 for each $\alpha$ of $t$ in $s$ do
2     if $\neg$static-thread-local($\alpha$) then return FALSE
3 return TRUE
end checkC1

checkC2($X$)
4 for each $\alpha \in X$ do
5     if visible($\alpha$) then return FALSE
6 return TRUE
end checkC2

checkC3($s$, $X$)
7 for each $\alpha \in X$ do
8     if onStack($\alpha(s)$) then return FALSE
9 return TRUE
end checkC3

ample($s$)
10 for each $t$ such that enabled($s, t$) $\neq \emptyset$ do   /* enforcing C0 */
11     if checkC1($s$,$t$) $\wedge$ checkC2(enabled($s, t$)) $\wedge$ checkC3($s$,enabled($s, t$)) then
12         return enabled($s, t$)
13 return enabled($s$)
end ample
```

Algorithm 2: Ample Set Construction Algorithm Using Static Escape Analysis

```
checkC1($s, t$)
1 $G := \emptyset$   /* static fields accessed from $t$ in $s$ */
2 $O := \emptyset$   /* heap objects/arrays accessed by $t$'s enabled transitions in $s$ */
3 for each $\alpha$ of $t$ in $s$ do
4     collectAccessed($s$,$t$,$\alpha$,$G$,$O$)
5 if $G \neq \emptyset$ then return FALSE
6 for each thread $t'$ in $s$ such that $t' \neq t$ do
7     if reachable($s$,$t'$,$O$) then return FALSE
8 return TRUE
end checkC1
```

Algorithm 3: Ample Set Construction Algorithm Using Dynamic Escape Analysis

the transitions in *ample*(s) is executed before a transition from *ample*(s).[4]

The function *checkC2* determines whether all transitions in a set are invisible [6]. The function *checkC3* enforces Condition **C3**.

5.2 Thread-Locality by Dynamic Analysis

Algorithm 3 presents the function *checkC1* that uses dynamic escape analysis to determine whether all transitions of a thread are thread-local. Intuitively, the dynamic escape analysis inspects the current state of the model checker to determine whether the tran-

[4]Interestingly, for transition systems arising from Java programs as we generate them in Bandera, it is enough to check for only the enabled transitions of t since the problematic enabled/disabled transition structure described above does not arise. However, we use the definitions above for generality.

sitions of a given thread are reachable by some other threads at that particular state. The functions *checkC2*, *checkC3*, and *ample* are the same as listed in Algorithm 2.

We briefly describe the steps in function *checkC1* which are easily implemented using Bogor's state representation interface. *checkC1* first collects all the static fields, objects, or arrays that are accessed by the transitions of thread t in state s (lines 1–4). If some globals are accessed, the algorithm returns FALSE since one or more transitions are not thread-local (line 5). If some heap objects accessed by t are reachable by the other threads (note that objects reachable through static fields are reachable by all the threads), then one or more transitions are not thread-local (lines 6–7). Otherwise, all enabled transitions of t at s are thread-local (line 8).

6 Partial-Order Reduction for Shared Objects using Locking-Discipline

Having established how thread-locality information can drive partial-order reductions, we now turn to the idea of exploiting locking information in partial-order reductions as emphasized by recent work of Stoller [31, 32]. Stoller showed how it is unnecessary to explore some interleavings for objects that are accessed according to a *locking discipline* (i.e., a systematic use of locks to obtain mutually exclusive access to object fields).

Our goals are to illustrate several lock-based strategies that take a different approach than that of Stoller and to illustrate how lock-based strategies can be enhanced when they are combined with the previously introduced thread-locality strategies. After summarizing Stoller's approach, we begin with a strategy based on what we call the *weak self-locking discipline* that is conceptually simpler (and thus easier to implement) and covers many common patterns of lock usage. With this strategy providing the basic intuition, we then proceed with a more general strategy called *weak self-locking domination* that captures monitor-based locking strategies. We note that neither this strategy nor Stoller's subsumes the other, and this leads us to a strategy that combines Stoller's approach and the weak self-locking domination strategy. Our experiments in Section 8 will illustrate the relative effectiveness of each of these approaches.

6.1 Stoller's Lock-based Reductions

In well-designed Java programs, mutually exclusive access to fields of shared objects is achieved by having threads acquire the implicit locks associated with Java objects. *Race conditions* can occur when a thread accesses a shared object without first acquiring an associated lock. In such conditions, the order of thread accesses/updates to an unprotected memory cell is difficult or impossible to predict, and this often leads to erroneous functional behavior.

The Eraser algorithm [30] introduced by Savage et.al. performs a run-time analysis to detect violations of a particular locking discipline, which we refer to as LD. Such violations indicate the possibility of race conditions. Assume that there is some static annotation system that indicates the set of objects *LD-Objects* that should be accessed according to LD. For example, a reasonable annotation system might be a per-class system that indicates that all objects from a particular class should satisfy LD.

Definition 4 (Locking Discipline (LD)) *An execution of a system satisfies LD iff for all objects $o \in$ LD-Objects, one of the following conditions hold:*

- *LD-RO: o is read-only after it is initialized.*

- *LD-lock: o is lock-protected after initialization, i.e., there exists an object o' such that for all threads t and for all execution states s in which t accesses o, t owns the lock of o' in s.*

The Eraser algorithm assumes initialization to be completed when an object becomes accessed *by two different threads.*

In this paper, we are interested in reduction strategies relating to the LD-lock case (i.e., reduction strategies that are driven by *locking* patterns). The reduction strategies that we are considering are orthogonal to LD-RO, and thus, they can be easily combined. Throughout the rest of the paper, we liberally use LD to refer to LD-lock in order to simplify the presentation.

Stoller proposed reduction strategies that capitalize on independence properties that result when objects satisfy LD. The essence of his method (ignoring several technical conditions for now) as implemented in the JPF model-checker is as follows. Users are required to give annotations to classify objects into the following categories: (a) objects that are not shared between threads, (b) *communication objects* – shared objects that do not obey LD, and (c) objects that obey LD (this category can be viewed as the complement of then union of (a) and (b)). During state-space exploration, if a transition α does not access a communication object and does not perform a lock acquire on an LD object, then there is no need to consider interleavings before α.

Stoller also establishes an important point related to the correct classification LD objects: the Eraser lock-set algorithm can be modified and applied simultaneously while performing reductions during state-space exploration to detect situations where a declared LD-object actually violates LD. If undetected, such situations lead to unsafe reductions in the sense that some interleavings are omitted that should actually be explored. When LD violations are detected in Stoller's approach, the user is expected to shift the object from the LD to the communication-object classification, then restart the model-check.

The basic idea of the run-time monitoring lock-set algorithm is to associate a table with each LD object that maps each thread to the set of locks that it holds when it accesses the object. As the model checker executes transitions, when a thread t accesses an object field, the lock set associated with the object for t is intersected with the set of locks currently held by t. An empty intersection value implies that LD has been violated.

As we relate this approach to our escape-analysis-based strategies, there are several issues to bring forward. First, Stoller suggests that static analysis may be applied to avoid having the user manually identify the set of unshared objects [31, Section 8]. This is essentially what we have done with the static escape analysis of Section 5.1. Note that this follows Stoller's conditions where an object's classification (e.g., as unshared) is fixed throughout the lifetime of the system. Our dynamic escape-analysis of Section 5.2 moves beyond this to lift the restriction that objects must have a fixed classification.

Next, note that LD and the associated Eraser lock-set algorithm do not require that an object be lock-protected until after initialization has completed (where "initialization completed" is defined to be a state where the object is accessed by a second thread). This treatment is designed to cover the common situation where a single thread runs the constructor of an object (which initializes the object) without locking the object, and then the object escapes and is lock-protected after the constructor completes. With our dynamic escape analysis, it is easy to enforce different variants of this approach.

Finally, Stoller based his reductions on the persistent-set/sleep-set framework [15] – the style of partial-order reductions used in Verisoft [16]. Verisoft implements stateless search, and does not support checking of LTL properties. Thus, Stoller's proofs do not try to establish the extra conditions (e.g., as embodied in conditions **C2** and **C3** of the ample set reductions) that are required to preserve LTL$_{-X}$. However, these conditions are orthogonal to notions of independence, etc., and could be easily added to Stoller's presentation.

Appendix D recasts Stoller's LD-based reduction strategy in the ample set framework and describes our implementation of it. We use this implementation as a point of comparison with the alternate strategies that we propose.

6.2 Weak Self Locking-Based Reductions

We now consider several different reduction strategies that combine thread-locality and lock-based reduction. We begin with a simple strategy that covers common usage of locks, and then consider other approaches building on this in the following subsections.

An object o satisfies what we call the *self-locking discipline* if, for any thread t that performs a non-lock-acquiring access on o, t holds o's lock. This discipline corresponds to the simplest type of locking used in Java programs, and it is typically achieved by declaring all the methods of a particular class to be `synchronized`.

As noted above, the constructor of an object will typically access its fields without locking the object – thus, technically violating the self-locking discipline. For instance, in the bounded-buffer example of Figure 3, instances of the class `BoundedBuffer` would satisfy the self-locking discipline (since methods of the class are declared as `synchronized`) except for the fact that these instances are not locked in the constructor. Accordingly, previous work on race condition detection and other verification oriented type systems (e.g., [14]) relies on making various unchecked assumptions about the behavior of relevant object constructors – namely, that the objects being initialized do not escape the constructor (thereby disallowing the possibility that other threads could generate conflicting accesses to the object). It is easy in our framework to check this assumption using thread-locality information. Moreover, it is natural to weaken the self-locking discipline to allow arbitrary (non-lock-protected) accesses to an object as long as the object is unshared. We call this weaker version the *weak self-locking discipline*. Specifically, an object o satisfies the *weak self-locking discipline* if *in any state s in which it is not thread-local*, for any thread t that performs a non-lock-acquiring access on o, t holds o's lock (the italicized phrase distinguishes this definition from the definition of *self-locking discipline*).

The following independence relation that exploits both thread-local information and weak self-locking information.

Definition 5 (Weak Self-Locking Transition Independence (WSL)) *Let $s \in S$, t_1 and t_2 be threads, $\alpha \in current(s, t_1)$, and β be any transition of t_2. Then both $(\alpha, \beta) \in I_s^{wsl}$ and $(\beta, \alpha) \in I_s^{wsl}$ hold iff $t_1 \neq t_2$ and either:*

1. *for each object o that α accesses, o is a thread-local object, or*
2. *both of the following conditions hold:*
   - *(a) α accesses no global memory cells (static fields), and*
   - *(b) for each object o that α accesses, o satisfies the weak self-locking discipline and o is locked by t_1 in s.*

We now argue that this relation satisfies the three properties of an independence relation given in Definition 2. Since we already proved the case for thread-local transitions when considering Definition 3, we consider only the second case.

For the [Preservation of Enabledness] condition, note that if t_1 holds the lock on an object o that it is accessing, then there is no action from α on o that can disable β.

```
checkC1($s, t$)
...
5.1 if none of $t$'s transitions at state $s$ is acquiring a lock on any $o \in O$ then
5.2     $O_{WSL} := \{o \in O \mid o$'s type is in $T_{WSL}\}$
5.3     for each thread $t' \in threads(s)$ such that $t' \neq t$ do
5.4         if some objects $O'_{WSL} \subseteq O_{WSL}$ are not locked by $t \wedge reachable(s, t', O'_{WSL})$ then
5.5             signal condition on $O'_{WSL}$ is violated by $t'$ at $s$
5.6     if $O = O_{WSL}$ then return TRUE
...
end checkC1
```

Algorithm 4: Ample Set Construction Algorithm Using Weak Self-Locking Discipline

Only the acquiring by t_1 of a previously unheld lock on o can disable β (when β is also attempting to acquire o's lock), and this cannot be the case since t_1 is already holding all the locks on the objects that α is accessing.

For the [Commutativity] condition, it is clear that if both α and β are enabled and t_1 already holds the lock of o, then β does not access o (since o satisfies the weak self-locking discipline, t_2 would have to hold the lock of o, but this is not the case). Thus, α and β commute.

For the [Preservation of Independence] condition, we need to show that t_2 can never cause α to lose its independence status. For that to happen, t_2 must lock o first, but this can not happen until t_1 releases the lock on o. Appendix C.1 presents a more rigorous justification of this condition.

Algorithm 4 presents the ample-set *checkC*1 function that exploits the weak self-locking discipline. We only present the additional computations from Algorithm 3 (inserting 5.1-5.6 in between line 5 and line 6).

Similar to the LD-based implementation of Stoller, we require the user to indicate which objects are weak self-locking objects. However, there are different approaches for providing this information. We require the user to indicate the Java classes whose instances are weak self-locking objects. For example, for the bounded buffer program in Figure 3, the user can specify that the `BoundedBuffer` class instances are weak self-locking objects. Note that this pattern in software design is common, for example, many classes in the Java collection framework have the same synchronization structure such as `java.util.Vector` and `java.util.Hashtable`, and it does not put a significant burden on the user to supply such annotations.

Given a set T_{WSL} of classes, the algorithm checks if a thread t at state s only (non-lock-acquiring) accesses instances of a class in T_{WSL}. If it does, then the reduction may be applied. It proceeds by checking that none of t's transitions is lock-acquiring the objects (line 5.1). If that is the case, the algorithm collects the set O_{WSL} of objects that are accessed by thread t and whose types are in T_{WSL} (line 5.2). The objects in O_{WSL} are then checked to see if they satisfy the weak self-locking discipline (line 5.3-5.5). Specifically, if some objects from O_{WSL} are not thread-local to t and are not locked, then the user is notified that a locking-discipline violation has occurred and applied reductions may be unsafe. Note that the notification is conservative, i.e., the reductions may be safe because it may be the case that the other threads never actually access the objects in the future. Regardless of the notification, if O_{WSL} are equal to O (line 5.6), then TRUE is returned. Since the objects are assumed to satisfy O_{WSL}, they cannot be accessed by other threads even though they are reachable. Therefore, the basic "possibility of interference" condition that we realized as reachability in Section 5 (i.e., accesses to object o can possibly interfere with each other when o is reachable from two or more different threads) is now realized as accessibility.

6.3 Weak Self-Locking Domination-based Partial-Order Reduction

We now consider the case where monitors are used to guard accesses to certain objects in addition to locking the objects directly. We use the term *monitors* as described in [1, p. 203]:

> First and foremost, monitors are a data abstraction mechanism. A monitor encapsulates the representation of an abstract object and provides a set of operations that are the *only* means by which that representation is manipulated. In particular, a monitor contains variables that store the object's state and procedures that implement operations on the object. A process can access the variables in a monitor only by calling one of the monitor's procedures. Mutual exclusion is provided implicitly by ensuring that procedures in the same monitor are not executed concurrently.

For example, in the bounded buffer program in Figure 3, an instance of the `BoundedBuffer` class protects (monitors) the array object that implements the buffer.

We will automatically discover the relationship of monitors and the objects being monitored instead of requiring the user to supply it. That is, instead of requiring an annotation that explicitly specifies the monitoring relation, the relationship is inferred. We only require that the user supplies annotations indicating weak self-locking objects as described in the previous subsection. Given a state s and the set of weak self-locking objects $O_{WSL} \subseteq \mathit{objects}(s)$ (where $\mathit{objects}(s)$ returns the set of all objects in state s) that are being held by a thread t in s, one can detect the objects that are protected by the objects O_{WSL} from another thread t'. Specifically, we look for situations in which, on all reference chains from t' to the objects being accessed, there exists an object in O_{WSL} whose lock is held by t (i.e., the paths from t' to objects being accessed are *dominated* by weak self-locking objects locked by t). We revise the independence relation to exploit this observation as follows.

Definition 6 (Weak Self-Locking Dominated Transition Independence (WSL^{dom})) *Let $s \in S$, t_1 and t_2 be threads, $\alpha \in \mathit{current}(s, t_1)$, and β be any transition of t_2. Then both $(\alpha, \beta) \in I_s^{wsl,dom}$ and $(\beta, \alpha) \in I_s^{wsl,dom}$ hold iff $t_1 \neq t_2$ and either:*

1. *for each object o that α accesses, o is a thread-local object, or*
2. *both of the following conditions hold:*
   - (a) *α accesses no global memory cells (static fields), and*
   - (b) *either one of the following condition holds:*
     - i. *for each object o that α accesses, o satisfies the weak self-locking discipline and o is locked by t_1 in s, or*
     - ii. *for each object o that α accesses, o can only be reached by t_2 from some weak self-locking objects O_{WSL} and t_1 locks all objects in O_{WSL} at s.*

We now argue that the definition of the WSL^{dom} independence relation satisfies the three properties of Definition 2. We only consider the last case (2.b.ii) as the other cases are the same as for the WSL relation.

```
checkC1(s, t)
. . .
5.1  if none of t's transitions at state s is acquiring a lock on any $o \in O$ then
5.2      $O_{WSL} := \{o \in O \mid o\text{'s type is in } T_{WSL}\}$
5.3      for each thread $t' \in threads(s)$ such that $t' \neq t$ do
5.4          if some objects $O'_{WSL} \subseteq O_{WSL}$ are not locked by $t \wedge reachable(s,t',O'_{WSL})$ then
5.5              signal condition on $O'_{WSL}$ is violated by $t'$ at $s$
5.6      if $O = O_{WSL}$ then return TRUE
5.7      $O_{SL} := \{o \mid o\text{'s type is in } T_{WSL} \ \wedge \ o \text{ is locked by } t\}$
5.8      $O_{SL^{dom}} := \emptyset$
5.9      for each thread $t' \in threads(s)$ such that $t' \neq t$ do
5.10         $O_{SL^{dom}} := O_{SL^{dom}} \cup \{o \in objects(s) \mid \neg reachable_{\downarrow O_{SL}}(s, t', \{o\})\}$
5.11     if $O \subseteq O_{SL^{dom}}$ then return TRUE
. . .
end checkC1
```

Algorithm 5: Ample Set Construction Algorithm Using Weak Self-Locking Dominated Discipline

For the [Preservation of Enabledness] condition, note that if t_1 holds the lock for all objects in O_{WSL} that protect o, then it is not possible for t_2 to access, and thus, be disabled by actions on o.

For the [Commutativity] condition, since t_2 does not access o, α and β commute.

For the [Preservation of Independence] condition, relies on the fact that t_2 can never make α lose its independence because it cannot access O_{WSL} since objects O_{WSL} are weak self-locking objects and o is protected by O_{WSL}. Appendix C.1 presents the formal justification of this condition.

Algorithm 5 presents the ample-set *checkC*1 function that exploits the weak self-locking dominated discipline. As stated before, the user is only required to specify which objects are weak self-locking objects. The algorithm begins by applying the weak self-locking discipline similar to Algorithm 4 (line 5.1-5.6). It then computes the set of objects $O_{SL^{dom}}$ that are dominated by self-locking objects that are locked by t (O_{SL}) from the other threads (line 5.6-5.10). If each element $o \in O$ is in $O_{SL^{dom}}$, i.e., all accessed objects are dominated, then TRUE is returned (line 5.11).

6.4 Combining LD and WSL^{dom} Disciplines

We now describe how the two locking discipline-based strategies can be combined. To see why this combination is useful, we first show that neither discipline subsumes the other.

Figure 4 presents an example where WSL^{dom} works better than LD. There are two synchronized containers (i.e., the containers are self-locking objects) and two threads. Each thread transfers the element of its left container to its right. The transfer can happen if the container is non-empty, otherwise the thread is blocked. Before the element is actually transferred, its `x` field is incremented. Thus, the element is accessed during the transfer.

Figure 5 illustrates the state after the thread $P1$ has locked the container $C1$ (denoted $C1_{P1}$) and when the thread is incrementing $E1$'s `x` field. The dotted arrow from $P1$ to $E1$ illustrates the reference to $E1$ that resides in the JVM stack when the increment happens. Notice that $P2$ can reach $E1$ only by going through $C1$ which is locked by $P1$. Thus, the increment by $P1$ is protected by $C1$. This state satisfies the conditions

```
public class WSLdom {
  public static
      void main(String[] args) {
    Container c1 = new Container();
    Container c2 = new Container();
    c1.put(new Element());
    (new Process(c1, c2)).start();
    (new Process(c2, c1)).start();
  }
}

class Process extends Thread {
  Container left;
  Container right;

  public Process(Container left,
                 Container right) {
    this.left = left;
    this.right = right;
  }

  public void run() {
    while (true) {
      left.transfer(right);
    }
  }
}
```

```
class Container {
  private Element element;

  public void put(Element e) {
    element = e;
  }

  public synchronized void
      transfer(Container other) {
    while (element == null) {
      try {
        wait();
      } catch
        (InterruptedException ie) {}
    }
    element.x++;
    synchronized(other) {
      other.element = element;
      other.notify();
      element = null;
    }
  }
}

class Element {
  public byte x;
}
```

Figure 4: An example showing LD does not subsume WSL^{dom}

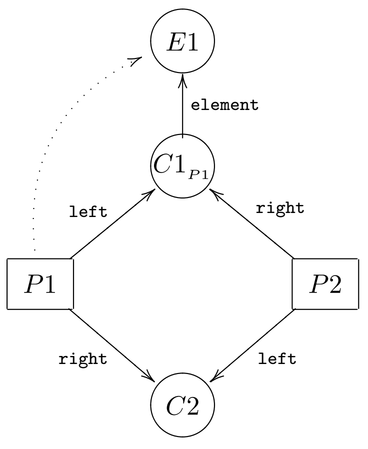

Figure 5: A state of the example in Figure 4

of WSL^{dom}. Once the element is transferred, then $P2$ can transfer the element from $C2$ to $C1$. As with $P1$, $P2$ increments the data in $E1$ first. However, this increment is now protected by $C2$ instead of $C1$. While this satisfies WSL^{dom}, it does not satisfy LD (i.e., $\{C1\} \cap \{C2\} = \emptyset$ — there does not exist a lock l such that l protects every access to $E1$).

Furthermore, because WSL^{dom} is using the notion of state-specific independence, it does not require that the conditions to be satisfied for all of the states. Analogous to the partial-order reduction using the dynamic escape analysis, the partial-order reduction

```
public class LDlock {                          class Process extends Thread {
  public boolean b;                              Object sync;
                                                 LDlock ldlock;
  public static                                  public Process(Object sync,
      void main(String[] args) {                                 LDlock ldlock) {
    Object sync = new Object();                    this.sync = sync;
    LDlock ldlock = new LDlock();                  this.ldlock = ldlock;
    new Process(sync, ldlock)                    }
        .start();                                public void run() {
    new Process(sync, ldlock)                      while (true) {
        .start();                                    synchronized(sync) {
  }                                                    ldlock.b = !ldlock.b;
}                                                    }
                                                   }
                                                 }
                                               }
```

Figure 6: An example showing LD is not subsumed by WSL^{dom}

by WSL^{dom} is applied only when the conditions hold.

Figure 6 illustrates that LD is not subsumed by WSL^{dom}. Basically, there is no reference relationship between the monitor (i.e., the object referred by `sync` in `FlipThread`) and the object being monitored (i.e., the object referred by `ldlock`). Therefore, WSL^{dom} can not be applied for this example while LD can.

We can combine the two approaches by weakening the condition of LD using WSL^{dom}. We only intersect the set of locks being held by the thread that accesses LD objects if the WSL^{dom} conditions do not hold at that state. In other words, we apply the WSL^{dom} discipline before applying LD. This combination subsumes both LD and WSL^{dom}. Furthermore, it *potentially* reduces the space requirement for the lock-set algorithm because WSL^{dom} is applied first. Thus, we potentially reduce the number of objects for which we should maintain the lock-set.

7 Optimizing by Avoiding State Storage

In the previous sections, we focused on reducing the space required for model checking by reducing the number of interleavings explored. In this section, we focus on reducing the space required by not inserting certain visited states into the *seen* set of Algorithm 1(b). We have already noted that when one has independent transitions α and β as in Figure 1, it does not matter whether one visits s_1 or s_2 on the way to r. The reduction that we now consider is based on the following heuristic: intermediate states like s_1 and s_2 are more likely to be encountered multiple times due to actions of different threads when all interleavings were included, but now that we are “thinning out” the interleavings, it is less likely that states generated from independent transitions will be arrived at from multiple paths. In addition, if there is only a single thread t enabled in a state s, it is less likely that the states generated from s by t will be arrived at from paths other than the one through s via t's action. Thus, we avoid storing a state $s' = \alpha(s)$ in *seen* when at least one of the following conditions holds: (1) α is an independent transition, or (2) α represents an action by thread t and t is the only thread enabled at s. Note that there may be several transitions that satisfy the first condition (i.e., independent). In choosing between these transitions, we prefer the one whose executing thread is the same as the last transition's executing thread. This reduces the amount of context switches, thus, it increases the sequentiality of counterexamples. Therefore, they are easier to understand.

Example		**(B)**	**(S)**	**(D)**	**(DA)**	**(SDA)**
Independent Work 1	**trans**	23302	1008	402	402	404
Threads: 3	**states**	6670	565	230	2	2
Locations: 36	**time**	6.63	.52	.32	.16	.13
Max. Objects: 22	**mem**	1.08	.41	.39	.36	.37
Independent Work 2	**trans**	18822	◁	402	402	◁
Threads: 3	**states**	5810	◁	230	2	◁
Locations: 48	**time**	5.56	◁	.47	.18	◁
Max. Objects: 22	**mem**	.96	◁	.4	.36	◁
Deadlock	**trans**	288	◁	178	253	◁
Threads: 3	**states**	142	◁	113	36	◁
Locations: 31	**time**	.26	◁	.26	.23	◁
Max. Objects: 6	**mem**	.37	◁	.36	.36	◁
Bounded Buffer	**trans**	10298	◁	7176	13488	◁
Threads: 3	**states**	4641	◁	3713	2473	◁
Locations: 64	**time**	4.11	◁	3.44	2.88	◁
Max. Objects: 10	**mem**	.89	◁	.72	.62	◁
Readers Writers	**trans**	1788086	◁	391191	700566	◁
Threads: 5	**states**	313331	◁	103638	79554	◁
Locations: 314	**time**	00:11:26	◁	00:03:42	00:03:46	◁
Max. Objects: 14	**mem**	30.5	◁	10.18	8.89	◁
Replicated Workers	**trans**	25557571	4376590	2076823	2535696	2661545
Threads: 4	**states**	4594721	1344215	827763	231219	231219
Locations: 509	**time**	09:00:54	01:12:35	00:39:48	00:21:47	00:21:10
Max. Objects: 44	**mem**	434.99	117.38	71.21	24.99	23.34

Table 1: Experiment Data

This technique is basically the *transition aggregation* technique commonly used in model checking. It is similar to using Promela's `atomic` command on independent transitions (or a single thread's transitions) as mentioned above. However, we aggregate the transitions on-the-fly instead of categorizing the transitions statically. This technique is also similar to state-caching techniques [21] that drops states as the *seen* state set starts to become too large. One difference is that our reduction technique does not *remove* states from the *seen* set. Instead, we avoid storing particular states that we believe are unlikely to be encountered in any way other than moving along the path we are currently traversing. In the case where the states are encountered again, their children states are going to be revisited. Thus, this technique does not lead to unsafe reduction, but it may introduce time overhead.

Of course, any time one omits storing visited states in *seen*, it becomes more difficult to guarantee termination of the model checking algorithm. In our setting, for example, if there is a non-terminating loop whose body consists of thread-local transitions, then model checking with our above heuristic incorporated will not terminate. Fortunately, for such models, this problem can be solved by *always* storing states that are produced from control-flow graph back-edge transitions, and this is the strategy that we implement.

A similar technique [2] is applied for model checking real-time systems using covering sets. The authors noted that their algorithm works well with a BFS state-space exploration, but perform poorly with a DFS algorithm due to large time overhead. As shown in the next section, our algorithm works really well with a DFS algorithm, and it significantly reduces the cost of model checking (both time and space).

8 Evaluating Reduction Methods

Table 1 and Table 2 show the results of evaluating our reduction strategies using six examples. For each example, we give data for each reduction strategy presented in the previous sections (whenever applicable). **(B)** is our base case that represents the state of our tools before incorporating thread-local object and lock-based reductions. It already applies a variety of sophisticated optimization techniques including collapse compression,

Example		(WSL)	(WSL^{dom})	(LD)	(LD+WSL^{dom})
Bounded Buffer	**trans**	3731	2914	4410	2914
Threads: 3	**states**	140	81	341	81
Locations: 64	**time**	.59	.54	.76	.56
Max. Objects: 10	**mem**	.46	.44	.49	.45
Readers Writers	**trans**	272680	262480	739570	262480
Threads: 5	**states**	20826	18924	86499	18924
Locations: 314	**time**	00:00:45	00:00:47	00:02:51	00:00:44
Max. Objects: 14	**mem**	3.38	3.25	9.42	3.25
Replicated Workers	**trans**	994007	994007	2908147	806863
Threads: 4	**states**	49354	49354	200947	36724
Locations: 509	**time**	00:04:08	00:05:06	00:16:39	00:03:52
Max. Objects: 44	**mem**	6.52	7.12	21.87	6.41

Table 2: Experiment Data 2

and heap and process symmetry reductions [28], and it aggregates transitions that only access local variables [7] which in effect implements partial-order reductions for local variable accesses. Each of the remaining strategies represents an *addition* of reductions to (B). **(S)** adds the static approach for discovering thread-locality (Section 5.1). The timing numbers reported in this column do not include the costs of performing the static analysis and the associated annotating transformations, since our numbers clearly indicate that the static approach cannot compete with the dynamic approach in any case. **(D)** adds the dynamic approach for discovering thread-locality (Section 5.2). **(DA)** is like (D), but uses the strategy to avoid state storage described in Section 7. **(SDA)** is a combination of (S) and (DA): for transitions that are statically annotated as thread-local (as described in Subsection 5.1) the scan of the heap required for dynamic escape analysis is omitted; for non-annotated transitions the heap scan is employed as described in Subsection 5.2. **(WSL)** adds reduction by the weak self-locking discipline described in Subsection 6.2. **(WSL^{dom})** adds reduction by the weak self-locking dominated discipline described in Subsection 6.3. **(LD)** adds reduction by Stoller's locking discipline reduction. As discussed in Section 6.4, these lock-based reductions cover different patterns of program locking so we have experimented with their combination **(LD+WSL^{dom})**.

For each example, we give metrics on the size of the example including the number of threads, control locations and the maximum number of objects allocated in a single program run across all runs of the program. For each reduction strategy, we present the number of transitions **trans**, **states**, **time** (seconds or hh:mm:ss), and memory consumption **mem** (Mb) at the end of the search; $\triangleleft$ indicates that numbers are identical to the numbers reported for the strategy to the immediate left. The reductions were implemented as extensions to Bogor [27] (which is implemented in Java), and the experiments were run on an Opteron 1.8GHz (32-bit mode) with maximum heap of 1Gb using the Java 2 Platform. In all runs, Bogor performed a full state-space search and found equivalent sets of errors across all strategies.

The *Independent Work (version 1)* example of Figure 2 represents a best-case reduction scenario where all heap-allocated data are obviously thread-local. As described in Section 3, both (S) and (D) are able to detect that the `Process` lists are thread-local, and that the entire run method may execute as a sequence of thread-local transitions. (D) improves on (S) because the dynamic escape analysis can determine that the instances of `Process` are thread-local before they are started, whereas (S) can not. Therefore, the creation and initialization of the second `Process` is not interleaved with the execution of the first `Process` in (D). In contrast, the interleavings happen in (S). When atomicity reductions are used (DA), the `run()` method bodies are executed atomically and the size of the resulting state-space contains only an initial and final state (i.e., the smallest possible state-space). Adding static escape information to (DA) consequently yields no further reduction. Since this example has no locking, the other methods have no impact and hence we did not run them.

The *Independent Work (version 2)* example of Figure 2 is identical to version 1 except that the thread local lists are allocated in the `main()` method. The static escape analysis (S) fails to identify the lists as thread local for the `Process` instances and consequently no reduction was gained. The dynamic (D) and atomicity (DA) had essentially the same performance as for version 1. As with version 1, the locking-based reductions yield no further reductions. It is interesting to note that both *Independent Work* examples could be scaled to have arbitrary numbers of threads with local lists of arbitrary length and the (DA) state-space would still remain at size 2.

In contrast, the *Deadlock* and *BoundedBuffer* examples represent nearly worst-case (thread-local) reduction scenarios where all heap-allocated data are manipulated by multiple program threads. In the *Deadlock* example, a pair of threads share objects on which locks are acquired, but there are no accesses to or through those objects. Static escape analysis (S) classifies objects that are unshared in some program states but not in others as non-thread-local; this is the case during initialization in *Deadlock*. Consequently it yields no reduction. Dynamic analysis (D) is able to detect independent transitions in the initialization phase, but that only yields minor reductions. Atomicity reductions (DA) yield a more significant reduction since many of the thread transitions are local. Despite the presence of locking, the *Deadlock* example does not contain object field reads or writes on which locking-based reductions are based. They provide no benefit over (B) without their combination with escape-based methods, and for this reason we do not report the statistics.

The *BoundedBuffer* example is more interesting. It has a pair of threads that share bounded buffer objects through which data are passed and on which the threads block waiting for available data to read or available space to write. All of the approaches presented in the paper provide reductions for this example except for the static escape analysis-based reductions. Just as for *Deadlock*, the static analysis-based technique is unable to leverage objects that are thread-local in only part of the program, and thus they yield no reduction. Dynamic (D) and atomicity (DA) approaches identify initialization states with independent actions and drive modest reductions. Locking disciplines play an important role in this system as the weak self-locking discipline is used (e.g., to update *head* and *tail* fields of the `BoundedBuffer` instances) and monitors are used (e.g., to access the array contained in the `BoundedBuffer`). All of the locking-discipline approaches provide additional reductions over (DA), and, as expected, the combined approach yields the best state space reduction, although in this case it is no better than (WSL^{dom}).

The *Readers Writers* example implements a flexible approach to implementing policies governing access to shared program regions [24]; reader or writer preference can easily be achieved. While these policies may change the conditions under which locks are acquired or released, the basic monitor relationships between objects and locks are established dynamically through heap references using the *specific notification* pattern. The dynamism in this example renders static escape analysis (S) ineffective. As for previous examples, dynamic (D) and atomicity (DA) approaches identify initialization states with independent actions and drive modest reductions and all of the locking-discipline approaches provide additional reductions over (DA). Dominated weak-self locking (WSL^{dom}) is also quite effective here yielding the best reduction.

The *Replicated Worker* example is a parallel programming framework written in Java that has been used in dozens of real scientific and engineering applications [12]. It reflects a more realistic balance between shared and local heap data; there are several shared objects, including an instance of `java.util.Vector`, and each thread in the system has several local vector instances and instances of `java.util.VectorEnumeration`. Furthermore, there are multiple locking patterns used in the implementation to govern access to shared `Vector` instances. Static escape analysis (S) is able to detect independent transitions that can be exploited for nearly a four-fold reduction in states. The rest of the reduction techniques present data that is largely consistent with that seen for the *Bounded Buffer*. In this example, the combined approach exploits the strengths of both (LD) and (WSL^{dom}) improving on each of them to yield an overall 125-fold reduction in states over (B).

We make the following observations from these results:

- Dynamic escape analysis is quite effective. We have demonstrated that the overhead is small, and that any cost is completely outweighed by the savings achieved (which in many cases are substantial). For model-checkers such as Bogor that permit an easy implementation, it seems that this approach should always be considered for implementation.
- In model-checkers where dynamic escape analysis can be implemented easily, considering static escape analysis is not worthwhile given the effectiveness of dynamic escape analysis. However, in model-checkers that do not admit an implementation of dynamic analysis, static escape analysis can be applied with good effect (as demonstrated by the numbers for the *Replicated Workers* example). Beforehand, we imagined that static analysis in combination with dynamic analysis might be useful to reduce the overhead associated with dynamic analysis (many thread-local transitions can be recognized once before model checking begins, as opposed to checking for thread-locality in every state). However, dynamic escape analysis in Bogor is so cheap, that there seems to be little point in trying to amortize some of the costs with a static analysis.
- It is easy to combine lock-based reductions with thread-local object reductions, and the resulting combination is much more effective than either approach alone.
- Applying simple heuristics to avoid storing states can be quite effective. Atomicity reductions (e.g., (DA)) can increase the number of transitions that are explored as seen in the *Replicated Workers* data. This can degrade run-time performance of the model checks, but that loss of performance is more than compensated for in our experiments by the reduction in state-space size. It seems quite possible that additional heuristics can be applied for effective reduction of state storage, and we are exploring other opportunities along these lines [11].
- It is unclear how to provide a general lock-based reduction strategy. We have taken the approach of identifying different patterns of program locking, targeting those patterns with reduction strategies, and then combining those strategies to maximize the degree of reduction. This has proven quite effective with the LD, WSL^{dom}, DA combination. We are studying strategies that target additional locking patterns that might be added to the framework.
- The empirical evaluation to date has demonstrated the potential for significant reduction. It is quite encouraging that the degree of reduction appears to be greatest for the largest, most complex example (*Replicated Workers*). We are applying our reductions to additional examples to gain a deeper understanding of the breadth of effectiveness of our techniques and the degree to which they enable scaling of software model checking.

9 Related Work

We have already discussed in detail how our work is related to existing work on partial-order reductions and escape analysis. Here we discuss several approaches that adapt partial-order reduction to target dynamic and concurrent software such as Java programs.

Iosif [22] combines heap and process symmetry reductions with partial-order reduction. This approach exploits the fact heap allocation statements are independent transitions. The heap symmetry ensures that for any execution order of enabled allocations the result be the same state. Our dynamic analysis subsumes this because Bogor uses the same heap symmetry reduction and new objects are only reachable from the thread that allocates it.

Stoller [31] exploits a common pattern in Java synchronization, i.e., locking of shared data structures. It also exploits transitions that access only thread-local objects. However, the user is required to indicate which objects are thread-local by specifying which bytecode instructions allocate thread-local objects. The tool uses this information during the search by making transitions that accesses thread-local objects invisible, and it also checks the validity of the user supplied information. In contrast, our static analysis approach automatically computes a safe approximation of the thread-local objects, and our dynamic analysis computes the most-precise objects thread-locality information automatically on-the-fly. We introduce similar locking discipline to Stoller's approach to exploit synchronization patterns in Java programs. Furthermore, we illustrate how our approach can be combined with Stoller's to achieve the best known reduction.

Brat et. al. [4] use alias analysis to establish which transitions are independent for partial-order reductions. This approach introduces an iterative algorithm where the model checker starts with an unsafe under-approximation of the alias set and refines the alias set on each iteration as the state exploration uncovers more aliasing. The iteration proceeds to a fix point in which a safe reduced state-space is constructed. In contrast, our dynamic approach does not require iteration to build a *safe* reduced state-space of the system. We can construct it on the fly because our notion of independence is state-sensitive instead of using the traditional global independence.

Flanagan and Qadeer have developed an approach for detecting when thread interleavings can be avoided using Lipton's theory of reduction [26] based on the notion of left- and right-movers. In [14], they present a type system for verifying that Java methods annotated as `atomic` are free from any interference due to interleaving, and thus clients of those methods can safely assume that the methods execute "in one step". This idea presents an interesting alternative for leveraging notions of independence: instead of using independence information to reduce the number of state explored in model checking, it is used to check that a developer has used locking in a correct way to achieve an interference-free method implementation. The type system relies on developer-supplied annotations that indicate which lock is used to protect fields. In addition, developers supply annotations to indicate which locks are held upon entry to a method and upon exit of a method. We believe that the ability to check a user-supplied annotation that a method is "atomic" is quite useful. The type system approach allows one to obtain efficient and scalable checking at the cost of a fair amount of user supplied annotations. In recent work [19], we have shown that model checking using the algorithms described in this paper can be effective for checking atomicity specifications. In particular, the approaches introduced here for detecting thread-locality and non-interference due to proper locking are applied to automatically discover properties that would otherwise need to be reported with annotations, and the increased precision of model checking yields fewer false positives when detecting atomicity violations. Of course, the tradeoff for this increased automation and precision is greater costs associated with state space exploration compared to type-checking. However, our experiments in [19] show that the state-space reductions described here enable relatively efficient checking for software units (as opposed to whole programs).

Flanagan and Qadeer have also proposed to use the theory of movers in a partial-order reduction strategy for state-space exploration [13]. They note that the partial-order reductions considered by ourselves and Stoller only incorporate reductions that move transitions in left-oriented fashion, and they argue that incorporate right-moving as well as left-moving transitions should result in better reduction techniques. Their framework relies on developer-supplied *access predicates* that indicate the conditions under which it is safe to access an object. They also show how access predicates can be automatically inferred by iterating the model checking. It is unclear how expensive this approach is since no performance numbers are given. We are currently investigating the extent to which this alternate view of partial-order reduction can be incorporated in our implementation. Note that since neither the atomic type system [14] nor the state-space reduction strategy [14] incorporates automatic detection of escape information, it is likely that incorporation of such information could provide additional benefits in the form of reducing the annotation burden or increasing the amount of reduction.

10 Conclusion

The application of model checking to reason about object-oriented programs is growing. Given the significant body of research on static analysis and transformations targeted at improving the run-time of such programs, it is natural to attempt to adapt those techniques to reduce the state-space of a system. In this paper, we have illustrated how both static and dynamic escape analysis can be applied to gather information that can be used effectively to reduce the state-space of a system. We have shown how reductions can be accomplished by adapting the previously developed partial-order reduction framework of ample sets. Further, we have shown how escape information can be combined with other lock-based reduction strategies and with heuristics for avoiding state-space storage. We believe that these techniques represent a significant step forward in the state-of-the-art of partial-order reductions for software model checking. Based on our experience thus far, the relative ease with which dynamic escape analysis results can be exploited in explicit state model checking algorithms combined with the power of its reduction suggests that it ought to become a standard component of model checkers for systems with dynamic memory allocation. Further experimental studies will be beneficial in revealing the breadth and significance of these reductions across a range of realistic concurrent Java programs.

A Overview of Static Escape Analysis

To statically detect thread-local objects, we use an adaptation of Erik Ruf's escape analysis [29]. Ruf's analysis detects objects that are guaranteed to be *locked* by a single thread only. Any synchronization statements that are only applied to such objects are unnecessary and can safely be removed from the program. We adapt Ruf's analysis to obtain a variant that detects objects that are guaranteed to be *accessed* by a single thread only. After giving a brief summary of Ruf's analysis, we describe in greater detail how we adapt it for our setting.

Ruf's analysis builds an abstraction of the heap by forming equivalence classes of expressions (i.e., reference variable names or names with field selectors) that refer to the same set of objects. Attributes recorded for each equivalence class capture information about threads which may synchronize on objects from the equivalence class. The equivalence classes are represented as *alias sets*, which are tuples of the form $\langle$*fieldmap*, *synchronized*, *syncThreads*, *global*$\rangle$ in which *fieldmap* is a function from field names to alias sets (giving alias information for member fields), *synchronized* is a boolean that indicates whether the objects represented by the current equivalence class may be synchronized, *syncThreads* is a set of threads that may synchronize on the object, and *global* is a boolean to track the escape status of the object. Unification (merging) on alias sets is defined as the join operation on boolean (*true* is top), set, and function lattices. Alias sets support creation of new instances which are isomorphic to the existing one. *Alias contexts* $\langle\langle p_0, p_1, \ldots p_n\rangle, r, e\rangle$ are used to capture the flow of information across method call/returns. They are composed of a sequence of alias sets, $p_0, p_1, \ldots p_n$ corresponding to the parameter list of the method along with an alias set r for return values r and another set e for thrown exceptions. The alias used at a method call-site is called a *site-context* and the set used for the method itself is called a *method-context*. Similar to alias sets, alias contexts support unification which is defined as the point-wise extension of alias set unification to tuples. Alias contexts support creation of new instances such that all relations between the original alias sets are preserved in the among the new instances of alias sets.

Ruf's analysis proceeds in three phases. In Phase 1, it identifies the thread allocation sites in the system and the methods executed in each thread instance. It also determines if each thread allocation site can produce multiple thread instances at run-time. In Phase 2, each global variable is associated with an alias set. It is assumed that all alias sets reachable from a global alias set (i.e., one which has its *global* field set to *true*) are global too. The analysis then does a bottom-up intra-procedural analysis on strongly-connected components(SCCs) in the system. The intra-procedural analysis involves creating alias sets for local variables and unifying them at suitable program points. The interesting cases occur at synchronization points and call-sites. At synchronization points, the *synchronized* element of the synchronized alias set is set to *true* and if it is *global* is *true* then thread allocation sites of the enclosing method is added to *syncThreads*. At call-sites, a *site-context* is created with the alias sets for receiver, arguments, return value and exception variables. For each callee at the call-sites, this site-context is then unified with the method context of the callee if the callee and the caller occur in the same SCC. If not, a new instance of the method context of the callee is created and unified with the site-context. In Phase 3, the analysis processes the SCCs in top-down topological order to process each synchronization point and call-sites. At each synchronization point, if the synchronized alias set is *contention-free*, i.e., $\mid$ *sycnThreads* $\mid < 2$, then the synchronization is marked for removal. At each call-site, a new instance, M', of the callee's method-context, M, is created. If M' is synchronized, *S.syncThreads* is injected into M'*.syncThreads*, recursively. M and M' are compared recursively to determine if the method should be specialized and called or a previous specialized instance can be called. The comparison evaluates to true when the contention-free status of corresponding alias sets in the alias contexts is same.

In our adaptation of this analysis we use a slightly altered alias set in which *synchronized* and *syncThreads* elements are absent. Instead there are two boolean elements named *accessed* and *shared* which indicate if the object was accessed and if it is shared between threads, respectively. The assumptions about globalness is unchanged.

Our analysis proceeds in three phases similar to Ruf's analysis but focuses on read-/write access points in the system rather than synchronization points. Phase 1 is similar to that of the Ruf's analysis except that we use information from Object/Value flow analysis to determine the call hierarchy (instead of the Rapid Type Analysis used by Ruf) and to identify the `java.lang.Thread.start` call-sites which may be executed multiple times during the execution of the system. In Phase 2, at each object access point, the *accessed* attribute of the corresponding alias set is set to *true*. The processing of call-sites is identical to Ruf's analysis except for when the called method is `java.lang.Thread.start`. In this case, if *accessed* element of two alias sets being unified is *true* then the *shared* element of the resulting alias set is set to *true*. The rational for this is that all objects reachable from a thread are reachable from objects created in that thread and/or from objects created by the parent thread. Hence, it suffices to determine escapism at `start` call-sites. In Phase 3, the information from the callers is propagated to the callees. When using the calculated information, if the object marked as *non-shared* is accessed in a method in a thread started at a call-site which is executed multiple time as determined in Phase 2 then that object is marked as *shared* or *escaping*.

B Modifications to Ample Set Framework Proofs

We have cast our reduction strategies in terms of the ample set partial-order reduction framework as presented in [6, Chapter 10]. That framework requires transitions marked as independent to be remain independent for all states. Our only change to ample set framework has been to relax this notion of independence to obtain the notion of *state-sensitive independence* based on *conditional independence* [23] as presented in the body of the paper.

Almost all of the proofs from [6, Chapter 10] go through unchanged. In this section, we give the "patches" to the proofs [6, Chapter 10] that are necessary to establish the correctness of the ample set reductions using state-sensitive independence.

The proofs of [6, Section 10.6] are based on a construction that aims to show that in a given transition sequence, the transitions in the sequence can be reordered by moving transitions from ample sets ahead of those transitions that are not in ample sets. We need to show that this construction is still valid when one moves to state-sensitive independence. The construction relies on the fact that in the traditional notion of independence, the independence property is not "lost" as one proceeds through the steps of the sequence (in the tradition notion of independence, a pair of transitions marked as independent are independent in all states). The following properties show that the property of state-sensitive independence is preserved in a way that allows the construction of [6, Section 10.6] to still be applied (specifically, the properties below justify the steps **B1** and **B2** in the construction [6, Section 10.6,p. 161]).

Property 1 *Let $s_0 \overset{\beta_0}{\to} s_1 \overset{\beta_1}{\to} \ldots \overset{\beta_n}{\to} s_{n+1}$ be a path such that $\alpha \in enabled(s_0)$ and $I_{s_0}(\alpha) \supseteq \{\beta_0, \beta_1, \ldots, \beta_n\}$, then $\alpha \in enabled(s_{k+1})$ and $I_{s_k}(\alpha) \supseteq \{\beta_k, \beta_{k+1}, \ldots, \beta_n\}$, for $0 \leq k \leq n$.*

Proof 1 *This is proved by induction on k:*

- *Case $k = 0$: Assume a path $s_0 \overset{\beta_0}{\to} s_1 \overset{\beta_1}{\to} \ldots \overset{\beta_n}{\to} s_{n+1}$, $\alpha \in enabled(s_0)$, and $I_{s_0}(\alpha) \supseteq \{\beta_0, \beta_1, \ldots, \beta_n\}$. Trivially, $\alpha \in enabled(s_0)$ and $I_{s_0}(\alpha) \supseteq \{\beta_0, \beta_1, \ldots, \beta_n\}$.*
- *Induction hypothesis (case $k = i$): If $\alpha \in enabled(s_0)$ and $I_{s_0}(\alpha) \supseteq \{\beta_0, \beta_1, \ldots, \beta_n\}$ for a path $s_0 \overset{\beta_0}{\to} s_1 \overset{\beta_1}{\to} \ldots \overset{\beta_n}{\to} s_{n+1}$, then $\alpha \in enabled(s_{i+1})$ and $I_{s_i}(\alpha) \supseteq \{\beta_i, \beta_{i+1}, \ldots, \beta_n\}$.*
- *Case $k = i + 1$: Assume a path $s_0 \overset{\beta_0}{\to} s_1 \overset{\beta_1}{\to} \ldots \overset{\beta_n}{\to} s_{n+1}$, $\alpha \in enabled(s_0)$, and $I_{s_0}(\alpha) \supseteq \{\beta_0, \beta_1, \ldots, \beta_n\}$. By induction hypothesis, we have $\alpha \in enabled(s_{i+1})$ and $I_{s_i}(\alpha) \supseteq \{\beta_i, \beta_{i+1}, \ldots, \beta_n\}$. By pair-wise application of the preservation of*

independence property of I_s (Definition 2), $I_{s_{i+1}}(\alpha) \supseteq \{\beta_{i+1}, \beta_{i+2}, \ldots, \beta_n\}$. By the preservation of enabledness property of I_s (Definition 2), $\alpha \in enabled(s_{i+2})$.

Property 2 *Given a path $s_0 \xrightarrow{\beta_0} s_1 \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_{k-2}} s_{k-1} \xrightarrow{\beta_{k-1}} s_k \xrightarrow{\alpha} s_{k+1}$ where $\alpha \in enabled(s_0)$ and $I_{s_0}(\alpha) \supseteq \{\beta_0, \beta_1, \ldots, \beta_{k-1}\}$, then there is also a path $s_0 \xrightarrow{\alpha} \alpha(s_0) \xrightarrow{\beta_0} \alpha(s_1) \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_{k-2}} \alpha(s_{k-1}) \xrightarrow{\beta_{k-1}} \alpha(s_k)$, where $s_{k+1} = \alpha(s_k)$.*

Proof 2 *This is proved by induction on k (the number of β transitions):*

- *Case $k = 1$: Assume $s_0 \xrightarrow{\beta_0} s_1 \xrightarrow{\alpha} s_2$, where $\alpha \in enabled(s_0)$ and $\{\beta_0\} \subseteq I_{s_0}(\alpha)$. By commutativity property of I_s (Definition 2), and because of $s_2 = \alpha(s_1)$, thus, $s_0 \xrightarrow{\alpha} \alpha(s_0) \xrightarrow{\beta_0} \alpha(s_1)$.*
- *Induction hypothesis (case $k = i$): Given a path $s_0 \xrightarrow{\beta_0} s_1 \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_{i-2}} s_{i-1} \xrightarrow{\beta_{i-1}} s_i \xrightarrow{\alpha} s_{i+1}$ where $\alpha \in enabled(s_0)$ and $I_{s_0}(\alpha) \supseteq \{\beta_0, \beta_1, \ldots, \beta_{i-1}\}$, then there is also a path $s_0 \xrightarrow{\alpha} \alpha(s_0) \xrightarrow{\beta_0} \alpha(s_1) \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_{i-2}} \alpha(s_{i-1}) \xrightarrow{\beta_{i-1}} \alpha(s_i)$, where $s_{i+1} = \alpha(s_i)$.*
- *Case $k = i + 1$: Assume a path $s_0 \xrightarrow{\beta_0} s_1 \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_{i-2}} s_{i-1} \xrightarrow{\beta_{i-1}} s_i \xrightarrow{\beta_i} s_{i+1} \xrightarrow{\alpha} s_{i+2}$ where $\alpha \in enabled(s_0)$ and $I_{s_0}(\alpha) \supseteq \{\beta_0, \beta_1, \ldots, \beta_i\}$. By Property 1, we have $\beta_i \in I_{s_i}(\alpha)$. By commutativity property of I_s (Definition 2), we have $s_i \xrightarrow{\alpha} \alpha(s_i) \xrightarrow{\beta_i} s_{i+2}$. Thus, we have a path $s_0 \xrightarrow{\beta_0} s_1 \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_{i-2}} s_{i-1} \xrightarrow{\beta_{i-1}} s_i \xrightarrow{\alpha} \alpha(s_i) \xrightarrow{\beta_i} s_{i+2}$. By induction hypothesis, there is a path $s_0 \xrightarrow{\alpha} \alpha(s_0) \xrightarrow{\beta_0} \alpha(s_1) \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_{i-2}} \alpha(s_{i-1}) \xrightarrow{\beta_{i-1}} \alpha(s_i) \xrightarrow{\beta_i} s_{i+2}$.*

C Ample Set Condition Proofs

C.1 Justification of Independence Relations (Definition 3, Definition 5, and Definition 6)

C.1.1 Justification of Thread-Local Independence

In Section 3.2, we give informal arguments to justify that the conditions for state-sensitive independence are indeed achieved by our notion of independence based on thread-local transformation. Below, we give more rigorous arguments to justify that claim.

Property 3 *For each state $s \in S$, if there exists a transition $\alpha \in current(s)$ that is thread-local at s and a transition $\beta \in enabled(s)$ such that $s' = \beta(s)$ and α is* not *thread-local at s' then it must be the case that thread(α) = thread(β) and β is not thread-local at s.*

Proof 3 *Suppose that at a state $s \in S$, there exists a transition α that is thread-local and a transition $\beta \in enabled(s)$ such that $s' = \beta(s)$ and α is* not *thread-local at s'.*

- *Assume that thread(α) $\neq$ thread(β). This cannot be the case since α is thread-local at s. That is, α does not access objects that are reachable from other threads. In other words, no other threads can cause the objects that α accesses to be reachable by a thread other than thread(α).*

- *Assume that β is thread-local. This also cannot be the case since a thread-local transition cannot make some objects to be reachable by other threads. In other words, the only way a thread-local object o can become non-thread-local is by executing a transition δ that assigns o's reference to a static field or the field of an object o' that is already non-thread-local. In either case, such a transition δ is non-thread-local by definition.*

Therefore, by contradiction, it must be the case that thread(α) = thread(β) and β is not thread-local.

Preservation of Independence:
For each state $s \in S$, and for each $(\alpha, \beta) \in I_s^{tl}$, if $\{\beta_0, \beta_1\} \subseteq I_s^{tl}(\alpha)$ and *thread*$(\beta_0) \neq$ *thread*(β_1) and $s' = \beta_0(s)$, then $\{\beta_1\} \subseteq I_{s'}^{tl}(\alpha)$.

Proof 4 *Since $(\alpha, \beta_0) \in I_s^{tl}$ either α or β_0 (or both) is current at s and a thread-local transition at s:*

- *If α is a thread-local transition or both α and β_0 are thread-local transitions, then since thread(α) $\neq$ thread(β_0) by Property 3 it must be the case that α is thread-local at s' (and note α must be current at s' since its thread did not move in the transition to s'). Since we also have thread(α) $\neq$ thread(β_1), then $(\alpha, \beta_1) \in I_{s'}^{tl}$ by definition of I_s^{tl}.*
- *If β_0 is a thread-local transition and α is not thread-local, then β_1 is also thread-local. Since we have thread(β_0) $\neq$ thread(β_1), then again by Property 3, it must be the case that β_1 is thread-local at s' (and note β_1 must be current at s' since its thread did not move in the transition to s') and thus $(\alpha, \beta_1) \in I_{s'}^{tl}$.*

The proofs for both the weak self-locking independence and weak self-locking dominated independence are similar to the thread-local independence, and thus, they are not shown.

C.2 Proofs for Condition C1

In Sections 5 and 6, we introduced several *checkC1* functions using several reduction strategies. We now prove that the functions satisfy Condition **C1**. We only prove for the *checkC1* function using the weak self-locking dominated discipline because the other functions (using the dynamic escape analysis and the weak self-locking) are subsumed by this one. We do not consider the case where the *checkC1* function notifies the user because locking discipline annotation violation, because in those cases the current model checking run is abandoned, the user corrects the annotation and the run is restarted (note that these steps can also be taken automatically).

Property 4 *If the checkC1 function using the weak self-locking dominated discipline at Figure 5 returns TRUE for a thread t at state s, then current$(s, t) \subseteq$ enabled(s, t) (i.e., current(s, t) = enabled(s, t)).*

Proof 5 *(by contradiction) Suppose that there exists $\alpha \in$ current(s, t) such that $\alpha \notin$ enabled(s, t). There are three cases that can cause the checkC1 function to return TRUE, and we show that each of these leads to a contradiction:*

- *It returns TRUE at line 5.6. This means that the objects O that are accessed by current(s, t) are self-locking objects. By definition of self-locking objects, t must* *hold all the locks of objects O at s. This leads to a contradiction, because $\alpha \in$ current(s,t) and thus, it cannot be disabled at s because α can be disabled only if α is a lock acquire operation and some other thread t' holds the lock of some of the objects in O.*

- *It returns TRUE at line 5.10. This means that the objects O that are accessed by current(s,t) are protected by self-locking objects O_{WSL}. By definition of self-locking objects, t must hold all the locks of objects O_{WSL} at s. This leads to a contradiction, because $\alpha \in$ current(s,t) and thus, it cannot be disabled at s because α can be disabled only if some other thread t' holds the lock of some object in O. However, t' cannot access O without accessing O_{WSL} that are locked by t.*
- *It returns TRUE at line 8. This means that the objects O that are accessed by current(s,t) are thread-local objects. By definition of thread-locality, only t can reach the objects O. This leads to a contradiction, because $\alpha \in$ current(s,t) and thus, it cannot be disabled at s because α can be disabled only if some other thread t' holds the lock of some object in O. However, none of the objects in O is reachable from t'.*

Therefore, by contradiction, it must be the case that if the checkC1 function using the weak self-locking dominated discipline at Figure 5 returns TRUE for a thread t at state s, then current$(s,t) \subseteq$ enabled(s,t) (i.e., current$(s,t) =$ enabled(s,t)).

Lemma 1 *The checkC1 function using the weak self-locking dominated discipline at Figure 5 satisfies Condition* **C1**.

Proof 6 *The discussion in [6, p.157-158] notes that there are two conditions that are sufficient for establishing Condition* **C1** *when selecting ample$(s) =$ enabled(s,t): (1) enabled(s,t) should be independent from transitions in all of threads, and (2) no transitions from other threads should be able to enable any transitions of t that are disabled at s. We now show that these conditions are satisfied by checkC1.*

If the checkC1 function returns TRUE for a thread t at state s, then for all threads t' where $t \neq t'$:

- *It must be the case that for all $\alpha \in$ current(s,t) and for all β of t', we have that $(\alpha,\beta) \in I_s^{wsl,dom}$. This is true by the properties established in Appendix C.1.*
- *It must be the case that current$(s,t) \setminus$ enabled$(s,t) = \emptyset$. This is true by Property 4.*

D Stoller's LD-based Strategy in the Ample Set Framework

In Section 6.1, we describe how Stoller's approach for LD can be recast in the ample set framework. The *checkC1* function that uses LD is illustrated in Figure 6. The algorithm begins with collecting accessed static fields and objects (line 1-4). If any accessed fields or objects are communication objects, then FALSE is returned (line 5). If all the fields or objects accessed are unshared objects, then TRUE is returned (line 6). For each static field g, if the intersection of the set of locks being held at s with the set of locks being held at the states where g was accessed previously is empty, then the user is signaled that the search may be unsafe (line 7-9). This is done similarly for objects (line 10-13), except that the signaling is done if the objects are not currently being initialized at s.

Given an element e and a set S, the *intersectIfNotEmpty* function (line 16-21) intersects S with the set obtained by a look-up of the element from a global table *lockSetTable* (i.e., *lockSetTable*(e)). It then maps e to the result in the table, and then the result is

```
checkC1($s, t$)
 1 $G := \emptyset$   /* static fields accessed from $t$ in $s$ */
 2 $O := \emptyset$   /* heap objects/arrays accessed by $t$'s enabled transitions in $s$ */
 3 for each $\alpha$ of $t$ in $s$ do
 4     collectAccessed($s$,$t$,$\alpha$,$G$,$O$)
 5 if $\exists o \in O.o \in O_{commun}$ or $\exists g \in G.g \in G_{commun}$ then return FALSE
 6 if $O \subseteq O_{unsh}$ then return TRUE
 7 for each $g \in G$ do
 8     if intersectIfNotEmpty($g$, collectAllLocksHeldBy($t$, $s$)) $= \emptyset$ then
 9         signal condition on $g$ is violated by $t$
10 for each $o \in O$ do
11     if $\neg$inInitialization($o$, $s$) then
12         if intersectIfNotEmpty($o$, collectAllLocksHeldBy($t$, $s$)) $= \emptyset$ then
13             signal condition on $o$ is violated by $t$
14 return TRUE
end checkC1

intersectIfNotEmpty($e$, $S$)
15 if lockSetTable maps $e$ then
16     $S'$ := lockSetTable($e$) $\cap$ $S$
17     lockSetTable := lockSetTable[$e \mapsto S'$]
18     return $S'$
19 else
20     lockSetTable := lockSetTable[$e \mapsto S$]
21     return $S$
end intersectIfNotEmpty
```

Algorithm 6: Ample Set Construction Algorithm Using LD

returned. If the element e is not in the table, then, the e is mapped to S and S is returned.

Given an object o and a state s, the *inInitialization* determines whether the object is currently being initialized (e.g., we have not yet reached a state where o is accessed by a thread other than the thread that created it). This can be implemented by keeping the state of o. When it is first created, we keep track of the thread that creates it and we also note that o is in the initialization phase. For each access to o, we check the threads that access it. If it is different than the thread that created it, then we noted that o is no longer in its initialization phase. Given a thread t and a state s, the function *collectAllLocksHeldBy* collects all the objects that are locked by t at state s. This can be done simply by heap traversal of the state. We do not show the algorithms for *inInitialization* and *collectAllLocksHeldBy* due to space constraint.

References

[1] G. R. Andrews. *Concurrent Programming: Principles and Practice.* Addison-Wesley, 1991.

[2] G. Behrmann, K. G. Larsen, and R. Pelánek. To store or not to store. In W. A. H. Jr. and F. Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 433–445. Springer, July 2003.

[3] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric SPIN. *International Journal on Software Tools for Technology Transfer*, 2002.

[4] G. Brat and W. Visser. Combining static analysis and model checking for software analysis. In *Proceedings of the 16th IEEE Conference on Automated Software Engineering*, Nov. 2001.

[5] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for object oriented languages. application to Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 1–19, Denver, CO, Oct. 1999. ACM Press.

[6] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.

[7] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[8] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. Expressing checkable properties of dynamic systems: The Bandera Specification Language. *International Journal on Software Tools for Technology Transfer*, 2002.

[9] C. Demartini, R. Iosif, and R. Sisto. dSPIN : A dynamic extension of SPIN. In *Theoretical and Applied Aspects of SPIN Model Checking (LNCS 1680)*, Sept. 1999.

[10] J. Dolby and A. A. Chien. An automatic object inlining optimization and its evaluation. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI-00)*, pages 345–357, June 2000.

[11] M. B. Dwyer, Robby, X. Deng, and J. Hatcliff. Space reductions for model checking quasi-cyclic systems. In *Proceedings of the Third International Conference on Embedded Software*, 2003.

[12] M. B. Dwyer and V. Wallentine. A framework for parallel adaptive grid simulations. *Concurrency : Practice and Experience*, 9(11):1293–1310, Nov. 1997.

[13] C. Flanagan and S. Qadeer. Transactions: A new approach to the state-explosion problem in software model checking. In *Proceedings of the 2nd Workshop on Software Model Chekcing*, 2003.

[14] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 2003.

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

[16] P. Godefroid. Model-checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 174–186, Jan. 1997.

[17] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI-99)*, pages 293–304, May 1999.

[18] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher-order and Symbolic Computation*, 13(4), 2000.

[19] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model checking. In M. Young, editor, *Proceedings of the Fifth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2004)*, number 2937 in Lecture Notes In Computer Science, Jan 2004.

[20] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.

[21] G. J. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In *Proceedings of Third International SPIN Workshop*, Apr. 1997.

[22] R. Iosif. Symmetry reduction criteria for software model checking. In *Proceedings of Ninth International SPIN Workshop*, volume 2318 of *Lecture Notes in Computer Science*, pages 22–41. Springer-Verlag, Apr. 2002.

[23] S. Katz and D. Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101:337–359, 1992.

[24] D. Lea. *Concurrent Programming in Java: Second Edition.* Addison-Wesley, 2000.

[25] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1999.

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

[27] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003.

[28] Robby, M. B. Dwyer, J. Hatcliff, and R. Iosif. Space-reduction strategies for model checking dynamic software. In *Proceedings of the 2nd Workshop on Software Model Chekcing*, 2003.

[29] E. Ruf. Effective synchronization removal for java. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI-00)*, pages 203–213, June 2000.

[30] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[31] S. Stoller. Model-checking multi-threaded distributed Java programs. In *International Journal on Software Tools for Technology Transfer*. Springer-Verlag, 2002.

[32] S. Stoller and E. Cohen. Optimistic synchronization-based state-space reduction. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (LNCS 2619)*, 2003.

[33] W. Visser, K. Havelund, G. Brat, , and S. Park. Model checking programs. In *Proceedings of the 15th IEEE Conference on Automated Software Engineering*, Sept. 2000.