

Using Shape Analysis to Reduce Finite-State Models of Concurrent Java Programs*

James C. Corbett
Department of Information and Computer Science
University of Hawai'i
Honolulu, HI 96822
USA
corbett@hawaii.edu

October 14, 1998

Abstract

Finite-state verification (e.g., model checking) provides a powerful means to detect concurrency errors, which are often subtle and difficult to reproduce. Nevertheless, widespread use of this technology by developers is unlikely until tools provide automated support for extracting the required finite-state models directly from program source. Unfortunately, the dynamic features of modern languages such as Java complicate the construction of compact finite-state models for verification. In this paper, we show how shape analysis, which has traditionally been used for computing alias information in optimizers, can be used to greatly reduce the size of finite-state models of concurrent Java programs by determining which heap-allocated variables are accessible only by a single thread, and which shared variables are protected by locks. We also provide several other state-space reductions based on the semantics of Java monitors. A prototype implementation of the reductions demonstrates their effectiveness.

Keywords

Model Extraction, Java, Concurrent Systems, Finite-state Verification, State-space Reductions, Shape Analysis.

1 Introduction

Finite-state verification tools (e.g., model checkers) have the potential to put the power of formal verification in the hands of software developers. Unlike theorem provers, these tools are largely automatic and do not require the user to understand the details of the verification process. Unlike testing tools, finite-state verifiers are exhaustive in nature and are capable of showing that a particular kind of error is absent in a model of a program. This last point is especially important for concurrent software since a given input may produce different behaviors if the threads are scheduled differently; this can produce errors that are subtle and often difficult to reproduce.

Despite the advantages finite-state verification offers software developers, the transition of this technology from research to practice has been slow. Although there are many reasons for this, two of the most formidable technical obstacles to the transfer of this technology are the *state explosion problem* and the *model construction problem*.

The state explosion problem refers to the exponential increase in the number of states in a finite-state model as the number of components grows, making analysis of all but the smallest systems computationally infeasible. Possible components include threads, processes, variables, message buffer slots, or (for hardware) registers. In the past decade, many methods have been developed to alleviate the state explosion for specific classes of systems by, for example, reducing the number of states that need to be explored [17, 33] or representing the states symbolically rather than

*This work was supported by the National Science Foundation under grant CCR-9708184. An early version of this work appeared in the Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'98) [11], March 2-4, 1998, sponsored by ACM SIGSOFT.

explicitly [5, 28]. These techniques, combined with the judicious use of abstraction, can often produce tractable models of software systems. If these methods fail, a *restricted model* of the program can usually be constructed by arbitrarily limiting the number of components and/or the ranges of variables. Restricted models do not capture all behaviors of the program, but since many design errors are manifest in small versions of a system, they can be useful for finding errors [14, 21].

A second significant obstacle to the practical application of finite-state verification to software is the model construction problem. This refers to the semantic gap between the artifacts produced by software developers and those accepted by current verification tools. Most development is done with general-purpose programming languages (e.g., C, C++, Java, Ada), but most verification tools accept specification languages designed for the simplicity of their semantics (e.g., process algebras, state machines). In order to use a verification tool on a real program, the developer must extract an abstract mathematical model of the program's salient properties and specify this model in the input language of the verification tool. This process is both error-prone and time-consuming.

In fact, these two problems are closely related since analysis of a naive model of even a trivial program is likely to be intractable. As pointed out in [20], a considerable amount of abstraction is almost always needed to construct an analyzable model. The challenge in constructing a model is to capture just enough detail of the program under analysis to check the requirements, but not so much detail as to make the analysis intractable. Generally this involves constructing an abstract and/or restricted model in which only certain control points and data values are represented.

We believe that much of what a developer does in constructing a model of a program could be automated, that tool support for this activity would greatly reduce the effort required to apply finite-state verification technology to software systems, and that this might accelerate the transfer of this technology from research to practice. The ultimate goal would be a tool that could accept a program and a property and automatically extract a finite-state model of the program accurate enough to check that property. Since constructing tractable models can require user knowledge and ingenuity, a more realistic goal is a tool that can be guided by a human analyst as to what abstractions to apply. An automated tool is better suited for sifting through the considerable amount of detail present in most software artifacts and reliably performing the abstractions and/or running static analyses on the program to insure that specific abstractions are safe.

Many problems must be solved to build such a tool for any language. In this paper, we address one specific problem in building a model extraction tool for one specific language. The language is Java. The problem we address is how to reduce the size of the finite-state model by identifying thread actions that can be made invisible and collapsed into adjacent actions, a method known as *virtual coarsening* [1].

We consider Java because, with the explosion of internet applications, Java stands to become the dominant language for writing concurrent software. A new generation of programmers is now writing concurrent applications for the first time and encountering subtle concurrency errors that have heretofore plagued mostly operating system and telephony switch developers. Java uses a monitor-like mechanism for thread synchronization that, while simple to describe, can be difficult to use correctly (a colleague teaching concurrent Java programming found that more than half of the students wrote programs with nested monitor deadlocks).

Most previous work on concurrency analysis of software has used Ada [2, 3, 15, 16, 26, 37]. Although some aspects of these methods can also be applied to Java programs, the Java language presents several new challenges/opportunities. Among them:

1. Due to the object-oriented style of typical Java programs, most of the variables that need to be represented are fields of heap allocated objects, not stack or statically allocated variables as is common in Ada. Access to data that is *owned* by a thread (i.e., accessible only by that thread) can be made invisible—this is a widely used and powerful reduction—but identifying such data on the heap is nontrivial.
2. Java uses monitors to synchronize access to shared data. Each heap object has a lock, which may protect variables in that object or in connected objects. Access to a variable that is *protected* (i.e., accessed only when a specific lock is held) can be made invisible, but again, determining which variables are protected by a lock is nontrivial.

In both case, we need information about the program's run-time heap structure in order to apply the reduction. Fortunately, there exist static analysis methods to collect such information. These so called *shape analysis* methods were developed for compiler optimization, where accurate information on aliasing can improve many standard optimizations. The methods construct a finite approximation of all possible heap structures a program could produce.

The main contribution of this paper is to show how shape analysis can be used to reduce the size of finite-state models of concurrent Java programs by determining which heap-allocated variables are actually owned by a single

thread, and which shared variables are protected by locks. In particular, we have extended the shape analysis method of Chase *et al* [6] to work in a concurrent setting and to collect information on one-to-one relationships between sets of heap objects, which is necessary for establishing that a variable is protected by a lock in a different object. We also present several other state-space reductions based on the semantics of Java monitors.

We have implemented the shape analysis and state-space reductions in a prototype tool that constructs a finite-state model of a concurrent Java program, and we have applied this tool to several small programs to demonstrate the effectiveness of the method in reducing the size of the models.

This paper is organized as follows. We first provide a brief overview of Java’s concurrency features in Section 2. Section 3 defines our formal model (transition systems) and Section 4 explains how the size of such models can be reduced with virtual coarsening given certain information on run-time heap structure is available. We then explain how to collect this information using our modified shape analysis method in Section 5. Section 6 shows how to use the heap structure information to apply the reductions. In Section 7, we describe the implementation of a prototype tool for model construction that uses the reductions and report on the application of the tool to two programs. Section 8 discusses related work and Section 9 concludes.

2 Concurrency in Java

We explain the basic concurrency constructs of Java, as well as some features of our reduction method, using two examples of common concurrent design patterns [23]. These examples are not full concurrent Java programs per se, but could represent some fragment extracted from a full program.

The first example is an instance of the common *producer/consumer pattern*, shown in Figure 1. In Java, threads are instances of the class `Thread` (or a subclass thereof) and are created using an allocator (i.e., `new`). The constructor for `Thread` takes as a parameter any object implementing the interface `Runnable`, which essentially means the object has a method `run()`. Once a thread is started by calling its `start()` method, the thread executes the `run()` method of this object.

In the example, the program begins with the execution of the static method `main()` by the main thread. This creates an instance of an `IntBuffer`, creates instances of `Producer` and `Consumer` that point to this `IntBuffer`, creates instances of `Thread` that point to the `Producer` and `Consumer`, and starts these threads, which then execute the `run()` methods of the producer and consumer. The producer and consumer threads put/get integers from the shared buffer.

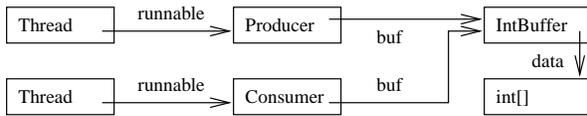
There are two types of synchronization in the producer/consumer pattern. First, access to the buffer should be mutually exclusive. Every Java object has an implicit lock. When a thread executes a `synchronized` statement, it must acquire the lock of the object named by the expression before executing the body of the statement, releasing the lock when the body is exited. If the lock is unavailable, the thread will block until the lock is released. A thread may acquire multiple locks, and may acquire the same lock more than once (without blocking). Acquiring the lock of the current object (`this`) during a method body is a common idiom and may be abbreviated by simply placing the keyword `synchronized` in the method’s signature.

The second type of synchronization involves waiting and notification: callers of `put()` must wait until there is space in the buffer, callers of `get()` must wait until the buffer is nonempty. On entry, the precondition for the operation is checked, and, if false, the thread blocks itself on the object by executing the `wait()`¹ method, which releases the lock. When a method changes the state of the object in such a way that a precondition might now be true, it notifies any waiting threads by executing the `notifyAll()` method, which wakes up all threads waiting on the object (these threads must reacquire the object’s lock before returning from `wait()`). There is also a `notify()` method that wakes up a single thread waiting on the object.

Another example of a concurrent Java program fragment is shown in Figure 2. This program fragment illustrates a second common design pattern—the *observer pattern* [23]. In this pattern, *subjects* (sometimes called observables), maintain state, while *observers* display/use this state. When the state of a subject changes, it notifies its observers, which are then responsible for querying the state of the subject to, for example, update the display. This pattern is very common in graphical user interface systems. Our example of the pattern in Figure 2 creates a linked list of subjects, and then passes this list to a `Mutator`, which walks the list and changes the states of the subjects.

¹Calls to the `wait()` method must be enclosed in a `try` block that catches the `InterruptedException` exception; we have omitted this detail from the listing for simplicity.

Heap Structure:



```

class IntBuffer {
    protected int [] data;
    protected int count = 0;
    protected int front = 0;
    public IntBuffer(int capacity) {
        data = new int[capacity]; // allocate array
        // data.length == size of array (capacity)
    }
    public void put(int x) {
        synchronized (this) {
            while (count == data.length)
                wait(); // wait until buffer not full
            data[(front + count) % data.length] = x;
            count = count + 1;
            if (count == 1) // buffer not empty
                notifyAll();
        }
    }
    public int get() {
        synchronized (this) {
            while (count == 0)
                wait(); // wait until buffer not empty
            int x = data[front];
            front = (front + 1) % data.length;
            count = count - 1;
            if (count == data.length - 1)
                notifyAll(); // buffer not full
            return x;
        }
    }
}

class Producer implements Runnable {
    protected int next = 0; // next int to produce
    protected IntBuffer buf;
    public Producer(IntBuffer b) { buf = b; }
    public void run() {
        while (true) {
            System.out.println("Put " + next);
            buf.put(next);
            // Data in buffer not represented
            // next = next + 1;
        }
    }
}

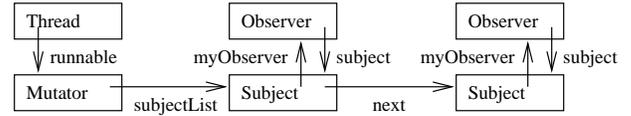
class Consumer implements Runnable {
    protected IntBuffer buf;
    public Consumer(IntBuffer b) { buf = b; }
    public void run() {
        while (true) {
            int x = buf.get();
            System.out.println("Get " + x);
        }
    }
}

public class Main {
    public static void main(String [] args) {
        IntBuffer buf = new IntBuffer(2);
        new Thread(new Producer(buf)).start();
        new Thread(new Consumer(buf)).start();
    }
}

```

Figure 1: Producer/Consumer Pattern Example

Heap Structure:



```

class Subject {
    int state;
    public Subject next; // next Subject on list
    Observer myObserver; // only one, for simplicity
    public Subject(int initialState) {
        state = initialState;
    }
    public synchronized void setObserver(Observer o) {
        myObserver = o;
    }
    public synchronized void changeState(int newState) {
        state = newState; // set and notify observer
        myObserver.changeNotification();
    }
    public synchronized int queryState() {
        return state;
    }
}

class Observer {
    Subject subject;
    int cachedState; // cache state of Subject
    public Observer(Subject s) {
        subject = s;
        cachedState = subject.queryState();
        subject.setObserver(this);
    }
    public synchronized void changeNotification() {
        int state = subject.queryState();
        if (state != cachedState) {
            cachedState = state;
            // Update display etc.
        }
    }
}

class Mutator implements Runnable {
    Subject subjectList;
    int newState;
    public Mutator(Subject subjects, int state) {
        subjectList = subjects;
        newState = state;
    }
    public void run() {
        Subject s = subjectList;
        while (s != null) {
            s.changeState(newState);
            s = s.next;
        }
    }
}

public class Main {
    public static void main(String [] args) {
        Subject first = null;
        for (int i = 0; i < 2; i++) {
            Subject s = new Subject(0);
            Observer o = new Observer(s);
            s.next = first;
            first = s;
        }
        new Thread(new Mutator(first, 1)).start();
    }
}

```

Figure 2: Observer Pattern Example

In this paper, we will focus on synchronization using locks, `wait()`, `notify()`, and `notifyAll()`, and assume threads are scheduled arbitrarily (this captures all possible behaviors). In practice, Java threads have several other methods to control their execution. In particular, they can join (block until child terminates) and yield (to the next thread); they can suspend, resume, and stop other threads (although these methods have been deprecated in recent versions of the libraries); and they can control their scheduling to some degree by setting their relative priorities. These additional features add some complexity to the transition system model described in the next section, but do not significantly affect the state space reductions that are the focus of this paper.

The models we construct are intended primarily for the detection of thread synchronization errors. Although there are no empirical studies on what kinds of synchronization errors are most commonly made by Java developers, articles like [34] and conversations with colleagues who teach concurrent programming in Java suggest that nested monitor deadlocks are the most common error. This error occurs when there is a cyclic wait among threads for locks (e.g., thread T_1 is holding lock ℓ_1 and requesting lock ℓ_2 while thread T_2 holds lock ℓ_2 requests lock ℓ_1). Designs employing waiting and notification are less common than designs that involve only mutual exclusion, but can also cause deadlock and starvation through missed signals, slipped conditions, and lockout [23]—situations in which a thread’s notification is sent too early or is never sent.

3 Formal Model

We model a concurrent Java program with a finite-state transition system. Each state of the transition system is an abstraction of the state the Java program and each transition represents the execution of code transforming this abstract state. Formally, a *transition system* is a pair (S, T) where

- $S = D_1 \times \dots \times D_n$ is a set of *states*. A state is an assignment of values to a finite set of *state variables* v_1, \dots, v_n where each v_i ranges over a finite domain D_i .
- $T \subseteq S \times S$ is a *transition relation*. T is defined by a set of guarded transformations $t_i : g_i \Rightarrow h_i$ of the state variables, where $g_i : S \rightarrow \{true, false\}$, called the *guard*, is a boolean predicate on states, and $h_i : S \rightarrow S$, called the *transformation*, is a map from states to states:

$$(s, s') \in T \text{ iff } \exists i. g_i(s) \wedge s' = h_i(s)$$

When $g_i(s)$ is true, we sometimes write $t_i(s)$ for $h_i(s)$.

A *trace* of a transition system is a sequence of transitions:

$$(s_0, s_1), (s_1, s_2), \dots, (s_{n-1}, s_n)$$

such that $(s_i, s_{i+1}) \in T$ for all $i = 0, \dots, n - 1$. Given a start state s_0 , a state s_i is *reachable* if there exists a trace starting at s_0 and ending with s_i .

The method of constructing the transition system representing a Java program is similar to the method presented in [10] for constructing the (untimed) transition system representing an Ada program. State variables are used to record the current control location of each thread, the values of key program variables, and any run-time information necessary to implement the concurrent semantics (e.g., run state of each thread, lock and wait queues of each object). Each transformation represents the execution of one or more bytecode instructions by a specific thread. A depth-first search of the state space can be used to enumerate the reachable states for analysis; at each state, a successor is generated for each ready thread, representing that thread’s execution.

We illustrate several important details of the translation in the example of Figure 3. The example program has two threads. Thread 1 acquires a lock on object \circ , sets program variable x to 1, wakes up any thread waiting on object \circ , then releases the lock on object \circ . Thread 2 acquires a lock on object \circ , blocks on object \circ , (when awakened) assigns program variable x to program variable y , then releases the lock on object \circ . The program contains a race condition—if thread 1 acquires the lock first, thread 2 will remain blocked forever; this is an example of a missed signal. The fragment of the state space at the bottom of the figure shows the state sequence when this does not occur.

Figure 3 also shows the state variables and transformations used to model the program. Note that each object has a lock state (free/taken), a queue of threads waiting for the lock, and a queue of threads waiting on the object (for notification). Threads attempting to acquire an unavailable lock (t_2, t_9) block and place themselves on the lock

Thread 1: 1: synchronized (o) { 2: x = 1; 3: o.notify(); 4: } 5: ...	Thread 2: 1: synchronized (o) { 2: o.wait(); 3: int y = x; 4: } 5: ...
---	---

State Variables:

$loc[i]$: location of thread $i = 1, 2$
 $ready[i]$: run state of thread i (boolean)
 $lock$: state of lock on object o (0 is free, 1 is taken)
 $lockQ$: threads waiting for lock on object o
 $waitQ$: threads waiting for notify on object o
 x, y : value of program variables x and y (initially 0)

Transformations:

$t_1 : (loc[1] = 1 \wedge lock = 0) \Rightarrow \{loc[1] := 2; lock := 1\}$
 $t_2 : (loc[1] = 1 \wedge lock = 1) \Rightarrow \{loc[1] := 2; ready[1] := F; append(1, lockQ)\}$
 $t_3 : (loc[1] = 2 \wedge ready[1]) \Rightarrow \{loc[1] := 3; x := 1\}$
 $t_4 : (loc[1] = 3 \wedge empty(waitQ)) \Rightarrow \{loc[1] := 4\}$
 $t_5 : (loc[1] = 3 \wedge \neg empty(waitQ)) \Rightarrow \{loc[1] := 4; append(dequeue(waitQ), lockQ)\}$
 $t_6 : (loc[1] = 4 \wedge empty(lockQ)) \Rightarrow \{loc[1] := 5; lock := 0\}$
 $t_7 : (loc[1] = 4 \wedge \neg empty(lockQ)) \Rightarrow \{loc[1] := 5; ready[dequeue(lockQ)] := T\}$

$t_8 : (loc[2] = 1 \wedge lock = 0) \Rightarrow \{loc[2] := 2; lock := 1\}$
 $t_9 : (loc[2] = 1 \wedge lock = 1) \Rightarrow \{loc[2] := 2; ready[2] := F; append(2, lockQ)\}$
 $t_{10} : (loc[2] = 2 \wedge ready[2] \wedge empty(lockQ)) \Rightarrow \{loc[2] := 3; ready[2] := F; append(2, waitQ); lock := 0\}$
 $t_{11} : (loc[2] = 2 \wedge ready[2] \wedge \neg empty(lockQ)) \Rightarrow \{loc[2] := 3; ready[2] := F; append(2, waitQ); ready[dequeue(lockQ)] := T\}$
 $t_{12} : (loc[2] = 3 \wedge ready[2]) \Rightarrow \{loc[2] := 4; y := x\}$
 $t_{13} : (loc[2] = 4 \wedge empty(lockQ)) \Rightarrow \{loc[2] := 5; lock := 0\}$
 $t_{14} : (loc[2] = 4 \wedge \neg empty(lockQ)) \Rightarrow \{loc[2] := 5; ready[dequeue(lockQ)] := T\}$

State Space (fragment): state = $(loc[1], ready[1], loc[2], ready[2], lock, lockQ, waitQ, x, y)$

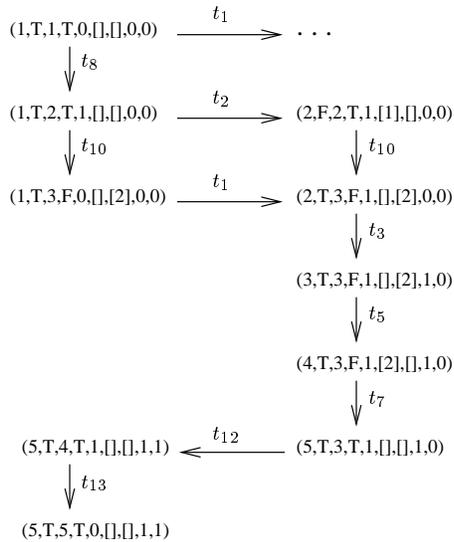


Figure 3: Example of Transition System

queue². When the lock is released, the releasing thread wakes up the first thread on the lock queue by setting its *ready* state variable (t_7, t_{14}) or releases the lock if no threads are waiting (t_6, t_{13}). Transformations that follow a potentially blocking operation (e.g., locking, `wait()`) must test the *ready* state variable in their guard; the thread must not proceed until ready. Note that *append* and *dequeue* are destructive operations that add an element to the tail and remove an element from the front of the passed queue, respectively; *dequeue* returns the element removed.

When a thread executes the `wait()` method (t_{10}, t_{11}), it blocks, places itself on the wait queue, and passes the lock to the next thread on the lock queue (or releases the lock if no threads are waiting for it). When a thread executes the `notify()` method (t_4, t_5), it simply takes the first thread from the wait queue and places that thread at the tail of the lock queue; the thread will be awakened when it gets the lock. The transformation for `notifyAll()` would move all threads from the wait queue to the lock queue. These transformations essentially mirror the implementation of the primitives in the Java virtual machine³.

Assignments to modeled program variables can be represented by transformations updating the corresponding state variables (t_3, t_{12}). The Java language specification [18] uses a weaker consistency model for memory than most languages. In particular, threads are permitted to cache local copies of variables; these copies are invalidated when any lock is acquired and written back to global memory when any lock is released. This allows compilers a great deal of flexibility in optimization, but can produce surprising results if the programmer does not use locks to synchronize access to shared data. In this paper, we do not attempt to model the behavior of Java programs that perform unsynchronized access to shared variables.

Not surprisingly, Java’s dynamic features are the most difficult to capture in a static model. In particular, the heap must be represented, but without allowing the state space to become unbounded. We bound the number of states in the model by limiting the number of objects each allocator (e.g., `new ClassName`) can create to k (we could easily allow a different bound for each allocator). For this paper, we assume k is provided by the analyst. If an allocator is executed more than k times, we create a transition to a special trap state—the model does not represent the behavior of the program beyond this point. As discussed in the introduction, such a restricted model can still be useful for finding errors.

Once the number of objects (including threads) is bounded, and assuming no (or statically bounded) recursion in the modeled part of the program, we can represent the state of the program using a fixed number of finite-state variables that record: the location of each thread (i.e., bytecode index), the run state of each thread, the values of the stack variables of each thread, the values of the operand stack slots of each thread (note that the size of the operand stack is determined by the bytecode index and hence the maximum stack size is known at compile-time), the values of the static variables, the values of the modeled instance variables of each object, the lock state of each object, and the lock and wait queues of each object. Of course, even a finite model may not be tractable to analyze. Extensive abstraction and/or restriction is almost always necessary to obtain a tractable model. Consider that a single Java `long` has 2^{64} states. Assuming we can generate one million states per second and store each state as a single bit, enumerating this state space would still require over 2 million years and more RAM than has been produced in all of history!

Model extraction requires carefully selecting which program variables to model. In general, variables directly mentioned in the property to be verified, or that could affect the values such variables, must be modeled. Modeled variables that are dynamic and/or have large ranges must usually be restricted to produce a tractable model, though sometimes they can be safety abstracted to a smaller range of abstract values using abstract interpretation [12]. In our experience, only a very small percentage of a program’s variables must be modeled to verify typical concurrency properties (e.g., absence of deadlock).

In this paper, we assume that some combination of dependency analysis, heuristics, and human assistance has determined which variables to model and any abstractions/restrictions that can be used to reduce the number and/or range of the state variables used to represent them. This is not an easy problem, but it is beyond the scope of this paper. Here, we focus on using information on run-time heap structure to reduce the number of states/transitions needed to model the program.

²The transformations shown for locking/unlocking are slightly simpler than the ones we use—in particular, these transformations do not handle multiple acquisitions of a lock by the same thread. The full transformations for lock/unlock keep track of the thread that owns the lock and maintain a count of how many acquisitions of the lock are unmatched by releases, releasing the lock only when this count goes to zero.

³Our model reflects a typical uniprocessor implementation of threads. In multiprocessor implementations, the presence of spinlocks results in more nondeterministic behavior; the model would have to be adjusted for this (e.g., by having the unlock transformation give the lock to an arbitrary waiting thread).

4 State Space Reductions

The transition system (S, T) produced by the method sketched in Section 3 is much larger than required for most analyses and may be too large to construct. Instead, we would like to construct a *reduced transition system* (S^*, T^*) where $S^* \subseteq S$ and use this for the analysis. We reduce the size of the transition system using *virtual coarsening* [1], a well-known technique for reducing the size of concurrency models by collapsing sequences of transitions that represent consecutive actions of a given component. Since we are using an interleaving model of concurrency, reducing the number of transitions in each thread greatly reduces the number of possible states by eliminating the interleavings of the collapsed transition sequences.

The reduced transition system is constructed by classifying each transformation defining T as *visible* or *invisible* and then composing each (maximal) sequence of invisible transformations in a given thread into the visible transformation following that sequence. The transitions and states generated by these composed transformations form (S^*, T^*) . For example, in Figure 3, we might replace transformations t_{12} and t_{13} with a single transformation $t_{12} \circ t_{13}$ that performs the assignment and releases the lock:

$$loc[2] = 3 \wedge ready[2] \wedge empty(lockQ) \Rightarrow \{y := x; loc[2] := 5; lock := 0\}$$

This eliminates traces in which other threads execute between these two actions, thus reducing the size of the state space.

We assume the requirement to be checked is specified as a stuttering-invariant formula f in linear temporal logic (LTL) [24], the atomic propositions of which are of the form $v_i = d_i$ where v_i is a state variable and $d_i \in D_i$. Statement $s \models_{(S, T)} f$ denotes that formula f is true in state s of transition system (S, T) . To be useful, the reduced transition system should:

1. *Be equivalent to the original transition system for the purpose of verification.* Specifically, for all $s \in S^*$ $s \models_{(S^*, T^*)} f$ if and only if $s \models_{(S, T)} f$.
2. *Be constructible directly from the program.* In particular, we should be able to build (S^*, T^*) without first constructing (S, T) , which may be too large to construct.

Although the reduced model constructed is specific to the formula f , we could construct a model suitable for a set of properties f_1, \dots, f_n by setting $f = f_1 \wedge \dots \wedge f_n$. Depending on the properties, it may be more efficient to generate a separate model for each f_i .

We classify a transformation as invisible and compose it with its successor transformation(s) only if we can show that this cannot change the truth value of f . An LTL formula is constructed by applying temporal operators to state predicates, which are boolean combinations of atomic propositions. Let p_1, \dots, p_m be the state predicates of f . An *f-observation* in a state s , denoted $P_f(s)$, is a vector of m booleans giving the value of (p_1, \dots, p_m) in s . A transformation $g \Rightarrow h$ is *f-observable* if it can change an *f-observation*:

$$\exists s \in S. g(s) \wedge (P_f(s) \neq P_f(h(s)))$$

Each trace $(s_0, s_1), \dots, (s_{n-1}, s_n)$ defines a sequence of *f-observations* $P_f(s_0), \dots, P_f(s_n)$, which we reduce by combining consecutive identical (i.e., stuttered) *f-observations*. It is easy to show that the set of these reduced *f-observation* sequences determines the truth value of f in s_0 [24]. If two traces produce the same sequence of reduced *f-observations*, we say they are *f-observation equivalent*.

Therefore, to satisfy condition 1 above, we construct the reduced transition system such that it has the same set of reduced *f-observation* sequences as the original transition system. To satisfy condition 2, we must do this without constructing (S, T) ; we must classify transformations as visible/invisible based on information obtained from the program code and/or language semantics. Note that all *f-observable* transformations must be visible regardless of whether they satisfy the reduction rules given below—we will not always repeat this caveat.

4.1 The Base Model

In this section, we review a simple virtual coarsening reduction based on local data. We then describe how this reduction applies to transition system models of Java programs. Finally, we use the reduction to define a model that will serve as the basis for evaluating the effectiveness of our program/language-specific reductions.

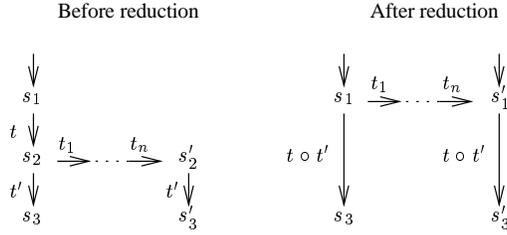


Figure 4: t, t_1, \dots, t_n, t' has the same reduced f -observations as $t_1, \dots, t_n, (t \circ t')$

One simple and widely used method for classifying transitions as invisible is based on the idea of local data. If a state variable is accessed only when a particular thread is running (i.e., only by transformations of that thread), then we say that state variable is *local* to the thread. A state variable is *accessed* by a guarded transformation if it appears in the expression defining the guard or in the imperative code defining the transformation. Transformations that access only local state variables may be made invisible provided they are not f -observable. The justification for this reduction is simple but worth reviewing.

Consider transformation t in Figure 4. Assume t is not f -observable and accesses only variables local to the thread whose code it represents. Let t' represent code in this same thread that can be executed immediately following the code represented by t . We can replace t and t' with $t \circ t'$ (if there are multiple successors to t , say t'_i for $i = 1, \dots, n$, then we replace t and t'_i for $i = 1, \dots, n$ with $t \circ t'_i$ for $i = 1, \dots, n$). To prove that this reduction does not change the truth value of f , we must show that the resulting transition system has the same set of reduced f -observation sequences as the original transition system.

For any state s_1 in which t is enabled, there may be one or more sequences of transformations t_1, \dots, t_n representing the execution of code from other threads (i.e., not the thread of t). Combining t and t' eliminates traces in which t_1, \dots, t_n occurs between t and t' . This does not eliminate any reduced f -observation sequences, however, since executing t_1, \dots, t_n before t must produce the same reduced f -observation sequence as executing t before t_1, \dots, t_n . Since t accesses only variables that t_1, \dots, t_n cannot access, t is independent of t_1, \dots, t_n and commutes: for any state s_1 in which both t and $t_1 \circ \dots \circ t_n$ are enabled, $t \circ t_1 \circ \dots \circ t_n(s_1) = t_1 \circ \dots \circ t_n \circ t(s_1)$. Since t is not f -observable, the trace obtained by executing t, t_1, \dots, t_n, t' must have the same reduced f -observation sequence as the trace obtained by executing $t_1, \dots, t_n, t \circ t'$.

When we first construct the state variables and transformations modeling the a Java program, each transformation represents a bytecode instruction (see [25] for more information on Java bytecodes). Since the state variables representing a thread's control location, operand stack slots, and stack-allocated variables are local by construction, we can immediately classify most of these transformations as invisible (provided they are not f -observable). The only transformations that access (potentially) non-local state variables are those representing:

- *get_field*, *put_field* instructions, which load/store an instance variable from/to a heap object,
- *array_load*, *array_store* instructions, which load/store an array slot⁴,
- *get_static*, *put_static* instructions, which load/store static variables,
- *monitor_enter*, *monitor_exit* instructions, which acquire/release a lock,
- *wait*, *notify*, *notifyAll* method calls, which block/signal threads waiting on an object,
- *start* method call, which starts the execution of a thread. If we were modeling other thread control calls (e.g., *suspend*, *join*), these would also be included.

Other transformations (e.g., load/store of stack allocated variable, arithmetic operations on the operand stack, control transfer) may be made invisible provided they are not f -observable. Since we expect any model construction tool for Java to apply this simple reduction, we call the resulting reduced transition system the *base model*.

⁴In Java, all arrays are heap objects.

In the sequel, we show how to reduce the size of the model further by classifying additional transformations as invisible using language semantics and/or information on run-time heap structure collected from a shape analysis of the program. In particular,

1. A *get_field*, *put_field*, *get_static*, *put_static*, *array_load*, *array_store*, *monitor_enter*, or *monitor_exit* transformation can be made invisible if the variable/lock accessed is *owned* by the thread at the instruction, meaning that only that thread can access the variable/lock at that point (e.g., the variable is in an object that is reachable from exactly one thread).
2. A *get_field*, *put_field*, *get_static*, *put_static*, *array_load*, *array_store*, *monitor_enter*, or *monitor_exit* transformation can be made invisible if the variable/lock accessed is *protected* by some lock ℓ , meaning that any thread accessing the variable/lock must be holding lock ℓ .
3. A *monitor_enter* transformation can be made invisible if the thread already holds the lock being acquired (i.e., the thread is acquiring the lock multiple times), and a *monitor_exit* transformation can be made invisible if it does not release the lock (i.e., the number of *monitor_exits* is not yet equal to the number of *monitor_ents* for that lock).
4. A *notify* or *notifyAll* transformation on an object ℓ can be made invisible and combined with succeeding transformations provided these transformations are invisible or are the *monitor_exit* transformation for object ℓ .
5. A *monitor_exit* transformation can be made invisible and combined with succeeding transformations provided these transformations are invisible or are other *monitor_exit* transformations.

In the remainder of this section, we describe and justify each of these reductions.

4.2 Owned Variable Reduction

In Section 4.1, we described how state variables that are local to a thread could be used for reduction. Here, we consider a generalization of this idea that is more useful for Java programs, in which much of the program data is heap-allocated and therefore global by the language semantics. If a variable can be accessed by only one thread at a particular program point, we say that the variable is *owned*⁵ by the thread at that point. An owned variable is different than a local variable in that a local variable is always local, whereas an owned variable may be owned only at certain statements, and may even be owned by different threads at different program points.

For example, consider the very common idiom in which objects are allocated, initialized, and then made available to other threads. This occurs in both of our example programs (see Figures 1–2). Before the reference to the allocated objects (e.g., the `IntBuffer` or the list of `Subjects`) is passed to another thread, these objects, though on the global heap, are not reachable from other threads and are therefore owned by the allocating thread. In many cases, once the objects are passed to another thread, they are never referenced again by the allocating thread. In this case, we effect a *change of ownership*—the other thread becomes the new owner of the objects. This occurs in the observer example of Figure 2, in which the `Mutator` thread becomes the owner of the `Subject` list. A change of ownership does not occur in the producer/consumer example, however, since the `IntBuffer` object is shared by the producer and consumer threads.

Determining exactly which variables are owned by a thread using static analysis is impossible. Instead, we use conservative approximations to find many (but not necessarily all) owned variables.

We classify a static variable as owned by a thread if it is accessed only by code reachable (in the call graph) from `main()`, or only by code reachable from a single `run()` method of a class that is passed to a `Thread` allocator at most once (the allocator is outside any loop or recursive procedure). Note that creating a `Thread` that then executes a `run()` method is *not* considered a method call. For example, if the variable `next` were a static member of class `Producer` in Figure 1, then since that variable is accessed only by code reachable from the `Producer`'s `run()` method, and since there is only one instance of `Producer` created, this analysis could determine `next` is owned by the producer thread.

We classify a heap allocated variable as owned by a thread if the object in which the variable is contained is reachable only from stack variables and/or static variables owned by the thread. For example, the variable `next` of class `Producer` in Figure 1 is accessible only by the producer thread.

⁵In [11], we presented a restricted version of this reduction in which we used the term “local” to describe what we here call “owned” variables.

For the purpose of our reductions, we treat the lock of an object as if it were an instance variable of the object. Thus the acquisition/release of locks on owned objects can be made invisible. Many classes are designed to be thread-safe and declare all public methods `synchronized` (this includes most of the standard library classes like `Vector` and `Hashtable`), but if these objects are never shared by multiple threads, this synchronization need not be represented in the model.

Once we have determined which variables/locks are owned, the `get_field`, `put_field`, `get_static`, `put_static`, `array_load`, `array_store`, `monitor_enter`, and `monitor_exit` transformations accessing such variables/locks can be made invisible (provided they are not f -observable). The justification for these reductions is the same as the justification for the local state variable reduction in Section 4.1—if no other thread could access the static/heap variable or lock at that point, then any sequence of transformations from other threads must commute with the transformation accessing the owned variable/object.

Applying this reduction requires knowledge about the accessibility of heap allocated variables at run-time. We describe how to collect this information using shape analysis in Section 5 and how to apply it in Section 6.

4.3 Protected Variable Reduction

We propose another technique⁶ for virtual coarsening based on Java’s monitors. A transformation that updates a variable x of an instance of class C may be made invisible provided there exists an object (lock) l_x such that any thread accessing x is holding the lock of l_x (l_x may be the instance of C containing x). We say the lock on l_x *protects* x . The intuition behind this reduction is that, even though other threads may access x , they cannot do so until the current thread releases the lock on l_x , thus any changes to x need not be visible until that lock is released.

The correctness of this reduction can be shown using the diagram in Figure 4. The reasoning is similar to that for the local variable reduction. Assume the only non-local variables t accesses are those that are protected by locks. The thread whose code t represents must hold the locks for these variables at s_1 . Therefore, although there exist transformations representing code in other threads that accesses these variables, such transformations cannot be in the sequence t_1, \dots, t_n since the other thread would block before reaching such transformations.

Assuming f does not reference the state of a shared object, this reduction allow us to represent complex updates to such objects with two transformations. For example, in the producer/consumer example, each execution of `put()` or `get()` updates several variables, yet we can represent each call with a transformation that acquires the lock and a transformation that atomically updates the state of the buffer and releases the lock.

In order to apply these reductions, we need to determine which locks protect which variables. Clearly if an instance variable of a class is only accessed within synchronized methods of that class, then the variable is protected by the lock of the object in which it is contained. Nevertheless, it is common for variables to be protected by locks in other objects. For instance, in the bounded buffer example, the array object referenced by instance variable `data` is protected by the lock on the enclosing `IntBuffer` object. This very common design pattern is known as *containment* [23]: an object A is conceptually nested in an object B by placing a reference to A in B and accessing A only within the methods of B .

Another common design pattern in which locks protect variables in other objects is *splitting locks* [23]. A class might contain independent sets of instance variables that may be updated concurrently. In this case, acquiring a lock on the entire instance would excessively limit potential parallelism. Instead, each such set of instance variables has its own lock, usually an instance of the root class `Object`. An example is given in Figure 5; two threads could concurrently update a `Programmer`’s hours and salary.

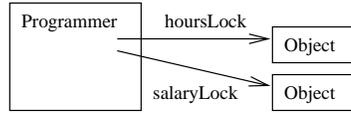
In general, determining which locks protect which variables requires information about the structure of the heap at run-time. We describe how to collect this information in Section 5 and how to apply it in Section 6.

4.4 Relock Reduction

Our last three reductions are not based on the specific Java program, but on the semantics of Java monitors. The first of these concerns transformations that acquire a lock that is already held by the current thread. We call this *relocking*. As noted in Section 2, a thread may acquire a lock more than once before releasing it; the thread continues to hold the lock until it has executed one `monitor_exit` operation for each `monitor_enter` operation on that lock. When a lock is reacquired, the `monitor_exit` transformation matching this `monitor_enter` does not release the lock.

⁶In [11], this reduction is called the “lock reduction.”

Heap Structure:



```

public class Programmer {
    protected long hours = 80;
    protected double salary = 50000.0;
    protected Object hoursLock = new Object();
    protected Object salaryLock = new Object();
    public void updateHours(long newHours) {
        synchronized (hoursLock) {
            hours = newHours;
        }
    }
    public void updateSalary(double newSalary) {
        synchronized (salaryLock) {
            salary = newSalary;
        }
    }
}
  
```

Figure 5: Example of Splitting Locks

Although relocking may be inefficient, it is not at all uncommon since public methods of a thread-safe class are usually declared `synchronized`, and these methods often call one another. A more complex example of relocking occurs in the observer example of Figure 2, in which a `Mutator` thread acquires the lock on a `Subject` to modify it, acquires the lock on its `Observer`, and then reacquires the lock on the `Subject` in `changeNotification()` to query the subject’s state.

Not surprisingly, reacquiring a lock that is already held is not an operation that can affect other threads, thus a `monitor_enter` transformation that does this may be made invisible. Similarly, the matching `monitor_exit` transformation, which does not release the lock, has no effect on other threads and can be made invisible. We can once again use Figure 4 to justify the reduction. If t is a relock transformation for the lock ℓ , then the thread whose code t represents must hold lock ℓ at s_1 . Consider any sequence of transformations t_1, \dots, t_n from other threads at s_1 . If t_1, \dots, t_n does not access lock ℓ , then clearly t_1, \dots, t_n commutes with t since they access disjoint sets of state variables. If there are one or more `monitor_enter` transformations on ℓ in t_1, \dots, t_n , then the threads executing these transformations will be blocked and queued on ℓ after t_1, \dots, t_n , regardless of when t is executed, thus the sequence still commutes with t . Therefore t, t_1, \dots, t_n, t' must have the same reduced f -observation sequence as $t_1, \dots, t_n, t \circ t'$.

This reduction may seem different than the first two in that it depends on the current state (i.e., whether the lock being acquired is already held), but since we represent a `monitor_enter` operation using several transformations covering the different cases (see Figure 3), it is not fundamentally different.

4.5 Notify Reduction

The second Java semantics-based reduction allows us to make `notify` and `notifyAll` transformations invisible under certain conditions. Specifically, we can compose a transformation representing a `notify` or `notifyAll` on an object ℓ with a succeeding transformation t' if t' is either itself invisible (by some other reduction rule) or t' is the `monitor_exit` transformation for object ℓ . The latter case is quite common since notification of a state change, when performed, is usually done at the end of the critical section. Also, in the degenerate case where ℓ ’s wait queue is empty (and thus the notification does nothing), the `notify` or `notifyAll` transformation can be made invisible and composed with any succeeding transformation.

The degenerate case is similar to the previous reductions and can be justified using the diagram in Figure 4. If the wait queue is empty, the `notify` or `notifyAll` operation has no effect and thus commutes with any sequence of transformations from other threads.

The commutivity shown in Figure 4 does not hold if the wait queue is nonempty. In this case, the notification will move some thread T_1 from the wait queue to the lock queue. If another thread T_2 executes a `monitor_enter`

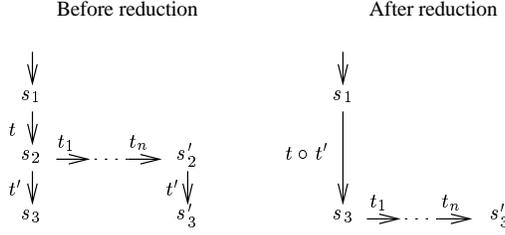


Figure 6: t, t_1, \dots, t_n, t' has the same reduced f -observations as $(t \circ t'), t_1, \dots, t_n$

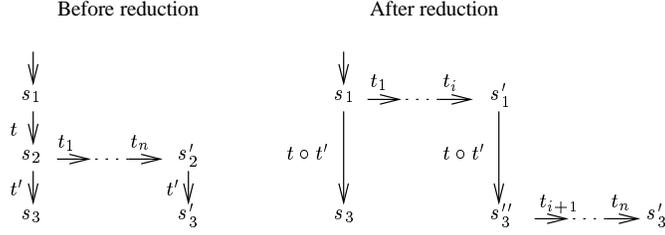


Figure 7: t, t_1, \dots, t_n, t' has the same reduced f -observations as $t_1, \dots, t_i, (t \circ t'), t_{i+1}, \dots, t_n$

transformation for ℓ in the sequence t_1, \dots, t_n , then the resulting order of threads T_1 and T_2 in the lock queue depends on whether t (the *notify*) or t_1, \dots, t_n (which contains the *monitor_enter*) occurs first. Thus $t_1, \dots, t_n, (t \circ t')$ does *not* commute with t, t_1, \dots, t_n, t' .

Under certain conditions, however, we claim that t, t_1, \dots, t_n, t' is f -observation equivalent to $(t \circ t'), t_1, \dots, t_n$, as shown in Figure 6. In particular, this is true if t is a *notify* or *notifyAll* transformation for object ℓ and t' is either invisible (by some other reduction rule) or the *monitor_exit* transformation that releases the lock on ℓ .

Clearly if t' is invisible, then t' and t_1, \dots, t_n commute by the justification of the appropriate reduction rule. They also commute if t' is the *monitor_exit* transformation that releases the lock on ℓ —any sequence of transformations t_1, \dots, t_n possible at s_2 must also be possible at s_3 since releasing a lock cannot disable any transformation; regardless of the order in which t' and t_1, \dots, t_n are executed, whatever thread is at the head of the lock queue at s_2 will end up with the lock (the lock queue cannot be empty at s_2 if t is not a degenerate *notify*). As long as both t and t' are not f -observable, t, t_1, \dots, t_n, t' is f -observation equivalent to $(t \circ t'), t_1, \dots, t_n$.

This reduction is unlike the others in that the f -observation sequence eliminated by the composition (t, t_1, \dots, t_n, t') is pushed forward to s_3 (i.e., is equivalent to $(t \circ t'), t_1, \dots, t_n$). As a result, the composed transformations produced by this reduction cannot be further composed using other reduction rules, which push the eliminated sequence backwards to s_1 . In particular, when a *notify* transformation is composed with an invisible transformation, we consider the resulting transformation to be a *notify* transformation, which could then be composed with its successor(s) using this reduction. However, two separate non-degenerate *notify* operations cannot be merged into the same transformation—this would not represent executions in which another thread attempted to acquire the lock between these operations and thus became queued between the threads awakened by the different notifies. Nor can a *notify* transformation composed with a *monitor_exit* be further composed with its successors.

4.6 Unlock Reduction

The last reduction allows us to make *monitor_exit* transformations invisible under certain conditions. Specifically, we can compose a transformation t representing a *monitor_exit* with a succeeding transformation t' if t' is either itself invisible (by some other reduction rule) or is another *monitor_exit* transformation.

Consider the diagram in Figure 7. Thread T_1 holds a lock ℓ_1 at s_1 and t represents the *monitor_exit* transformation releasing that lock. As before, t_1, \dots, t_n represents any sequence of transformations of threads other than T_1 . Making t invisible eliminates the trace t, t_1, \dots, t_n, t' , but we claim there exists an i such that $t_1 \dots, t_i, (t \circ t'), t_{i+1}, \dots, t_n$ is

enabled at s_1 and has the same sequence of reduced f -observations.

The simple case is where no t_i is a *monitor_enter* transformation for ℓ_1 (note that this is the only operation another thread could execute that affects the lock state/queue of ℓ_1 , which is what t is updating). In this case ($i = n$), the sequence t_1, \dots, t_n , is clearly independent of t and commutes.

The more complex case is where some t_i is a *monitor_enter* transformation for ℓ_1 in a thread T_2 . In this case, the sequence t_1, \dots, t_n , which is enabled at s_2 , is not necessarily enabled at s_1 . In particular, if t_{i+1} is a successor to t_i in T_2 (i.e., represents code in which T_2 is executing in the critical section protected by ℓ_1), then t_{i+1} is not enabled in the state $t_1 \circ \dots \circ t_i(s_1)$ since T_1 holds ℓ_1 at s_1 ; T_2 will block at t_i .

We can, however, split the sequence t_1, \dots, t_n into two pieces and execute *monitor_exit* (and its successor) between these pieces, as shown in Figure 7. The sequence t_1, \dots, t_i is enabled in s_1 , and the sequence t_{i+1}, \dots, t_n is enabled in s_3'' (after ℓ_1 is released by t) *provided that t' does not disable any of these transformations*. Neither *monitor_exit* nor invisible transformations can disable transformations in other threads. Thus, if t and t' are not f -observable, then the sequence t, t_1, \dots, t_n, t' is f -observation equivalent to $t_1 \dots, t_i, (t \circ t'), t_{i+1}, \dots, t_n$.

When a *monitor_exit* transformation is composed with another transformation, we consider the resulting transformation to be a *monitor_exit* transformation, which could then be composed with its successor(s) using this rule.

5 Reference Analysis

In this section, we describe a static analysis algorithm that constructs an approximation of the run-time heap structure, from which we can collect the information needed for the owned and protected variable reductions. Understanding run-time heap structure is an important problem in compiler optimization. One common approach is to construct a directed graph for each program statement that represents a finite conservative approximation of the heap structure for all control paths ending at the statement. Several different algorithms have been proposed, differing in the method of approximation.

Our algorithm is an extension of the simple algorithm given by Chase *et al* [6], which uses this basic approach. We extend Chase’s algorithm in two ways. First, we handle multi-threading; Chase’s algorithm is for sequential code. Second, we distinguish current and summary heap nodes; this allows us to collect information on one-to-one relationships between objects.

5.1 The Program

For the reference analysis, we represent a multi-threaded program as a set of control flow graphs (CFGs) whose nodes represent statements and whose arcs represent possible control steps. There is one CFG for each thread type: one CFG for the `main()` method and one CFG for each `run()` method. In this paper, we do not handle interprocedural analysis; we assume all procedure (method) calls have been inlined, thus we can only model the parts of a program with statically bounded recursion. Polymorphic calls can be inlined using a `switch` statement that branches based on the object’s type tag; since this tag is not modeled in our analysis, all methods to which the call might dispatch will be explored.

In our algorithm, we require the concept of a *loop block*. For each statement s , let $loop(s)$ be the innermost enclosing loop statement s is nested within (or `null` if s is not in any loop). The set $\{s' | loop(s') = loop(s)\}$ is called the loop block of s .

Our analysis models only reference variables and values. References are pointers to heap objects. A heap object contains a fixed number of fields, which are references to other heap objects (we do not model fields not having a reference type). For class instances, the number of fields equals the number of instance variables with a reference type (possibly zero). For arrays, the number of fields equals zero (for an array of a primitive type) or one (for an array of references); in the latter case all array elements are represented by a single field named `[]`.

In Java, references can be manipulated only in four ways: the `new` allocator returns a unique new reference, a field can be selected, a field can be updated, and references can be checked for equality (this last operation is irrelevant to the analysis).

5.2 The Storage Structure Graph

A *storage structure graph (SSG)* is a finite conservative approximation of all possible pointer paths through the heap at a particular statement s . There are two types of nodes in an SSG: *variable nodes* and *heap nodes*. There is one

variable node for each statically allocated reference variable and for each stack allocated reference variable in scope at s . There are one or two heap nodes for each allocator A (e.g., `new C()`) in the program, depending on the location of statement s in relation to A . If s is within the loop block of A or in a different thread (CFG) than A , then the SSG for s contains a *current node* for A , which represents the *current instance* of class C —the instance allocated by A in the current iteration of A 's loop block. For all statements s , the SSG for s contains a *summary node* for A , which represents the *summarized instances* of class C —all instances allocated by A in completed iterations of A 's loop block.

Each heap node has a fixed number of fields from which edges may be directed. Each edge in the SSG for a statement s represents a possible reference value at s . Edges are directed from variable nodes and fields of heap nodes towards heap nodes. In general, more than one edge may leave a variable node or heap node field since different paths to s may result in different values for that reference. Even if there is only one path to s , there may be multiple edges leaving a summary node or array field since such nodes represent multiple variables at run-time.

Several example SSGs for part of the Observer pattern of Figure 2 are shown in Figure 8. We elide parts of the code not relevant to the analysis with `...` and prepend line numbers to simple statements for identification. Variable nodes are shown as circles, heap nodes as rectangles with a slot for each field. Heap nodes are labeled with the name of the class prefixed with the statement number of the allocator. Summary nodes are suffixed with an asterisk(*). Thus `2:Subject*` represents the summary node for the allocator of class `Subject` at statement 2. Note that the linked list is represented with a self loop on node `2:Subject*` in the final SSG for statement 4.

Like Chase *et al* [6], we distinguish objects of the same class that were allocated by different allocators. This heuristic is based on the observation that objects allocated by a given allocator tend to be treated similarly. For example, both `Employee` and `Meeting` objects might contain a nested `Date` object allocated in their respective constructors (i.e., there are two `Date` allocators). By distinguishing the two kinds of `Date` objects, the analysis could determine that a `Date` inside of an `Employee` cannot be affected when the `Date` inside of a `Meeting` is updated.

A *conservative SSG* for a statement s contains the following information about the structure of the heap at run-time:

1. If there exists an edge from the node for variable X to a heap node for allocator A , then after some execution path ending at s (i.e., s has just been executed by a thread), X may point to an object allocated by A . Otherwise, X cannot point to any object allocated by A .
2. If there exists an edge from field F of the current heap node for allocator B to a heap node for allocator A , then after some execution path ending at s , the F field for the current instance allocated by B may point to an object allocated by A . Otherwise, the F field for the current instance allocated by B cannot point to any object allocated by A .
3. If there exists an edge from field F of the summary heap node for allocator B to a heap node for allocator A , then after some execution path ending at s , the F field for some summarized instance allocated by B may point to an object allocated by A . Otherwise, there is no summarized instance allocated by B whose F field points to any object allocated by A .
4. For each of the above three cases, if the heap node for allocator A is the current node, then the reference must be to the current instance allocated by A , otherwise the reference is to some summarized instance allocated by A .

Note that the useful information is the lack of an edge. One graph is more precise than another if it has a strict subset of its edges.

5.3 The Algorithm

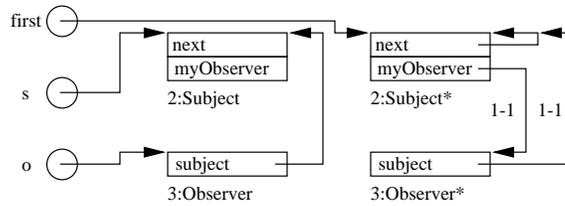
We use a modified dataflow algorithm to compute, for each statement, a conservative SSG with a small number of edges. Initially, each statement has an SSG with no edges. A worklist is initialized to contain the start statement of `main()`. On each step, a statement is removed from the head of the worklist and processed, possibly updating the SSGs for that statement and all statements in other CFGs. If any edges are added to the statement's SSG, the successors of the statement in its CFG and any dependent statements in other CFGs are added to the tail of the worklist. One statement is *dependent* on another if they may reference the same variable at run-time: they select the same static variable or instance variable. The algorithm terminates when the worklist is empty.

```

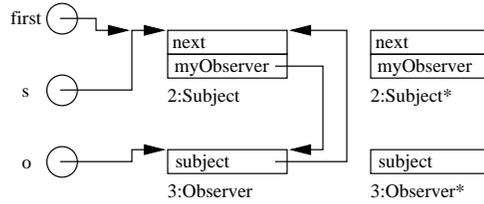
class Main {
  public static void main(String [] args) {
1:    Subject first = null;
    ...
    while (...) {
2:      Subject s = new Subject(...);
3:      Observer o = new Observer();
      { // inlined Observer()
4:        o.subject = s;
          { // inlined setObserver()
5:            synchronized (o.subject) {
              o.subject.myObserver = o;
            }
          }
        }
6:      s.next = first;
7:      first = s;
    }
    ...
  }
}

```

SSG for statement 4 (final)



SSG for statement 7 (first iteration, before summary)



SSG for statement 7 (first iteration, after summary)

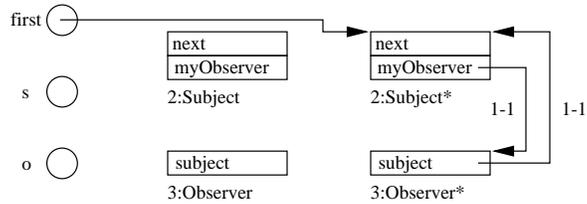


Figure 8: Reference Analysis Example

To process a statement, we employ three operations on SSGs: join, step, and summarize. First, we compute the *pre-SSG* for the statement by joining the SSGs of all immediate predecessors in its CFG. SSGs are joined by taking the union of their edge sets (this is an any-paths analysis). The pre-SSG is then updated by the step operation (discussed below) in a manner reflecting the semantics of the statement to produce the *post-SSG*. Finally, if the statement is the last statement of a loop block, the post-SSG is summarized to produce the new version of the statement’s SSG, otherwise the post-SSG is the new version. We summarize an SSG by redirecting all edges to/from the current nodes of allocators within the loop block to their corresponding summary nodes. An example of the summarization of statement 7 is given in Figure 8. The edges from local variables s and o are removed since these variables are out of scope at the end of the block.

The step operation uses abstract interpretation to update the SSG (an abstract representation of the run-time heap) according to the statement’s semantics. Only assignments to reference variables need be considered; other statements cannot add edges to the SSG (i.e., the post-SSG equals the pre-SSG). Each pointer expression has an *l-value* and an *r-value*, defined as follows. The l-value of a variable is the variable’s node. The l-value of a field selector expression $x.f$ is the set of f fields of the nodes in the r-value of x . The r-value of an expression is the set of heap nodes pointed to from the expression’s l-value, or, in the case of an allocator, the current node for that allocator.

The semantics of an assignment $E_1 = E_2$ depend on whether the left hand side is a stack variable, an owned static variable, or neither. If E_1 is either a stack variable or an owned static variable, we perform a *strong update* by removing all edges out of the node in $l\text{-value}(E_1)$ and then adding an edge from the node in $l\text{-value}(E_1)$ to each node in $r\text{-value}(E_2)$. Otherwise, we perform a *weak update* by simply adding an edge from each node/field in $l\text{-value}(E_1)$ to each node in $r\text{-value}(E_2)$. Strong updates yield more precise information, but are not conservative if E_1 is a heap variable, since then $l\text{-value}(E_1)$ represents multiple variables at run-time, nor when E_1 can be accessed concurrently by multiple threads, since we have no information on the order of these assignments.

Any edges added to a statement’s SSG (for a step or summarize operation) are also added to the SSGs for all statements in other CFGs; we assume threads may be scheduled arbitrarily, thus any statement in another thread may witness this reference value.

The execution of a thread allocator `new Thread(x)` is treated as an assignment of x to a special field `runnable` in the `Thread` object (this reflects the inlining of the constructor for `Thread`). We assume that we can statically determine the class of the object passed to a `Thread` allocator using dataflow analysis on the reference values (this is almost always the case in practice). When the allocator is processed, we add to the worklist the start statement of the CFG for the corresponding `run()` method⁷. When a statement in this `run()` method accesses an instance variable of the current object `this` (e.g., the expression `next` in the `Producer`’s `run()` method of Figure 1), the r-value of `this` is the heap node pointed to by the `runnable` field of the thread.

While constructing the SSGs, we can easily compute the set of program expressions that might reference a heap variable represented by a given heap node field; this information will be used in Section 6 to apply the owned and protected variable reductions. Let v be a slot of an SSG node A . Slot v represents one or more heap variables at run-time. We want to compute the set $Access(v)$, defined as the set of reference expressions in the program that might access (i.e., read/write) a variable represented by v . A reference expression is either a stack or static variable, or a selector of the form $x.f$ where x is a reference expression and f is a field name. A reference expression e is in $Access(v)$ if and only if $v \in l\text{-value}(e)$. We extend this definition from fields to objects in the natural way: the set of expressions that might access an object represented by SSG node A , denoted $Access(A)$, are those expressions that might access some field of A :

$$Access(A) = \bigcup_{v \in Fields(A)} Access(v)$$

where $Fields(A)$ are the slots of SSG node A .

5.4 Computing One-to-One Relationships

The summarized information gathered by the above analysis is not sufficient for the lock reduction. An SSG edge from the summary node for an allocator A to the summary node for allocator B indicates only that objects allocated by A may point to objects allocated by B . In order to show, for example, that the lock in an object allocated by A protects

⁷Technically, the thread is started when its `start()` method is called, but since we are not using any thread scheduling information, assuming the thread starts when allocated produces the same SSGs.

a variable in objects allocated by B , we need to know that for *each* object allocated by B there exists a *unique* object allocated by A ; only then would holding the lock of an A object protect a variable access in the nested B object.

We can conservatively estimate this information when SSGs are summarized and updated as follows. An edge from the summary node for A to the summary node for B is marked *one-to-one* if each A points to a different B at run-time. If A and B are in the same loop block, then an edge from some field of the summary node of A to the summary node of B , when first added to an SSG by a summarize operation, is marked one-to-one. If the field of the summary node of A is subsequently updated by a step operation in such a way that another edge to the summary node of B would have been added, then the edge is no longer marked one-to-one. For example, in the observer system of Figure 2, the arcs between the `Subject` and `Observer` summary nodes are marked one-to-one, as shown in Figure 8.

This method is based on the observation that nested objects are almost always allocated in the same loop block as their enclosing object (often in the enclosing object’s constructor). Given a constructor or loop body that allocates an object, allocates one or more nested objects, and links these objects together, the one-to-one relationships between the objects are recorded in the SSG as arcs between the current nodes of the allocators. When these nodes are summarized at the end of the loop block, this information is then preserved as annotations on the arcs between the summary nodes.

5.5 Complexity

Shape analysis is one of the more expensive computations performed by compilers. Given a program with S statements and V variables and allocators, the algorithm must construct S SSGs each containing $O(V)$ nodes and up to $O(V^2)$ edges. The running time to process a statement is (at worst) proportional to the total number of edges in all SSGs, as is the number of times a statement can be processed before a fixpoint is reached. Thus the worst case running time is $O(S^2V^4)$. Here, S is the number of statements *after* inlining all procedure calls, which could produce an exponential blowup in the number of statements.

Despite this complexity, we believe the average cost of the reference analysis will be acceptable for several reasons. First, SSGs are generally sparse; many edges in a typical SSG would violate Java’s type system and could not be generated by the analysis. Also, very few edges are added to a statement’s SSG after it has been processed once, thus each statement is typically processed only once or twice. Second, S and V refer to the number of *modeled* statements and variables—in a typical analysis, only a fraction of the program will be modeled. The reference analysis does not model variables having primitive (i.e., non-reference) types, nor need it model statements manipulating such variables. Also, a program requirement might involve only a small subset of the program’s classes; the rest of the program need not be represented. Third, since this analysis is used as a preprocessing step to reduce the size of the model generated, and since the time/space complexity of model building/checking is exponential in the size of the program, the overall complexity is likely to be dominated by the model building/checking step.

6 Applying the Reductions

In this section, we describe how to apply the owned and protected variable reductions from Section 4 using the information on heap structure collected by the method in Section 5. Although we speak of run-time entities (e.g., objects, references), we are actually using the approximate information in the SSGs to drive the reductions.

After making the transformations that reference owned and protected variables/locks invisible, we then apply the relock, notify, and unlock reductions from Section 4, which can combine locking and notify actions with these newly invisible transformations.

6.1 Applying the Owned Variable Reduction

As stated in Section 4.2, a variable/lock is owned at a particular program point if only one thread can access the variable at that point. We conservatively estimate a set of statically allocated variables that are owned using the simple analysis sketched in Section 4.2. For heap allocated variables/locks, we use the SSGs constructed in the last section to determine if and when an object is made accessible to other threads. In particular, we compute the *shared scope* of each SSG node, which is the set of program statements at which some object represented by that node might be accessible by more than one thread.

```

while (true) {
  // Receive request (e.g., connection, message)
  ...
  // Create object encapsulating request
  Request r = new Request(...);
  // Create a new thread to process request
  new Thread(r);    // Request is Runnable
}

```

Figure 9: Multi-Threaded Server Example

When an object is allocated by a thread, the object is owned by the thread until the thread places a reference to the object into a variable that is accessible by other threads. Given the set of owned static variables and the SSGs, we can compute, at each statement, the set of variables potentially accessible by other threads. This set includes any static variable that is not owned by the thread, the `runnable` field of another `Thread` object, any stack variable of another thread, and any heap variable in an object reachable (in the SSG) from any of these variables.

An assignment s in thread T_1 that may place the reference of an object represented by SSG node A into a variable that is potentially accessible by another thread is an *exposing assignment* for A and for all nodes reachable from A in the SSG for s . For example, the (implicit) assignment of the reference for the `Mutator` to the `runnable` field of the `Thread` object in the statement

```
new Thread(new Mutator(first, 1)).start();
```

of Figure 2 is an exposing assignment for the `Mutator` object as well as the linked list of `Subjects` and `Observers`. An exposing assignment s of an SSG node A in a thread T_1 causes a change of ownership if:

1. the assignment is to a `runnable` field of a `Thread` object (thus we know the object is being passed to a specific thread), and
2. thread T_1 cannot access any objects represented by A after executing s . If A is a summary node, this is true provided there does not exist a reference expression e reachable from s (in the CFG) such that $e \in \text{Access}(A)$. If A is a current node, then this is true provided there does not exist a reference expression e that follows s in the loop block of A such that $e \in \text{Access}(A)$, and there does not exist a reference expression e reachable from s such that $e \in \text{Access}(A_s)$, where A_s is the summary node for A .

This is the case for the assignment of `Mutator` mentioned above since none of the exposed objects are referenced after the exposing assignment in thread `main()`. There would also be a transfer of ownership of `Request` objects in the example of Figure 9, which is a common pattern in multi-threaded servers.

For this analysis, exposing an object is considered an access to the object, thus if A (or some nested object) were exposed to another thread subsequently, there would be no change of ownership. For example, in the producer/consumer system of Figure 1, there is no change of ownership of the `IntBuffer` object in the assignment of the `Producer` reference to the `runnable` field of the first `Thread` created since the `IntBuffer` object is subsequently exposed to another thread on the following line.

We compute the shared scope of each SSG node as follows. Initially, all shared scopes are empty. For each exposing assignment s by thread T_1 that exposes objects represented by A and that is not a change of ownership, we add the set of statements reachable from s in T_1 , as well as all statements from all other threads, to the shared scope of A . In addition, we include the shared scope of a current node in the shared scope of the corresponding summary node. In the producer/consumer system of Figure 1, the `IntBuffer` object and its nested array are shared for all statements in `main()` following the assignment of the reference to the `Producer` object to the `runnable` field of the first `Thread`, and are shared for all statements in the `run()` methods of the `Producer` and `Consumer` classes. There is a change of ownership for the `Producer` and `Consumer` objects, thus these are not shared.

A transformation representing code at statement s , may be made invisible provided that all variables/locks it might access are owned by the thread at s . A heap allocated variable/lock allocated by A is owned by the thread at s if s is not in the shared scope of A .

6.2 Applying the Protected Variable Reduction

A variable/lock is protected by a lock ℓ if any thread accessing the variable/lock must be holding ℓ . In general, the relationship between a variable and the lock that protects it may be too elaborate to determine with static analysis. Here, we propose a heuristic that we believe is widely applicable and, in particular, works for the locking design patterns given in [23]. The heuristic assumes that the relationship between the object containing the variable and the object containing the lock matches the following general pattern: either the lock object is reachable from the variable object, or vice versa, or both are reachable from a third object, or the lock and variable are in the same object.

This pattern can be expressed in terms of three roles: the *root*, the *lock*, and the *variable*. The lock object contains the lock, the variable object contains the variable, and from the root object, the other two objects are reachable. Each role must be played by exactly one object, but one object may play multiple roles. For the expression `data[i]` in the producer/consumer example of Figure 1, the `IntBuffer` object is both the root and the lock object, while the `int` array referenced by `data` is the variable object. For the expression `count`, the `IntBuffer` object plays all three roles. For the expression `salary` in the splitting locks example of Figure 5, the `Programmer` object plays the root and variable roles, while the `Object` referenced by `salaryLock` plays the role of lock.

We consider all static variables to be fields of a special environment object called `env`, which can play the roles of variable and root, but not the role of lock. This generalizes the pattern to include the case where the lock object or the variable object are reachable from static variables, and the case where the variable is static. Also, we fully qualify all expressions by prepending `this` to expressions accessing variables in the current instance, and by prepending `env` to all static variable accesses.

For each static/heap variable, we want to determine whether there exists a lock that protects the variable (i.e., any thread accessing the variable must be holding the lock). Given that the variable may be accessed at different statements by different threads, and that the heap may change shape between these statements, this is a nontrivial problem. We first give an informal description of our approach, and then describe the details of the method.

At each statement where a variable is accessed, we use the Java expression that accesses the variable and the Java expression that specifies the lock in the enclosing `synchronized` statement to identify a set of SSG nodes that could represent the root, variable, and lock objects of the pattern. If we can deduce from the SSG that there is exactly one lock object for each variable the expression could access, then we record that the access to the variable is protected by the lock. If all accesses to a given variable are protected by the same lock, then the variable is protected.

Formally, for each static/heap variable v , we want to compute $Protect(v)$: the set of locks protecting v . In our analysis, static variables are represented by variable nodes, heap variables by fields of heap nodes, and locks by heap nodes in the SSG. For each variable v , let $Access(v)$ be the set of program expressions that may access v , as defined in Section 5.3, and let $SharedAccess(v)$ be the subset of $Access(v)$ that excludes expressions at which v is owned, as determined by the analysis in Section 6.1. Owned variables that are later shared need not be protected by a lock while owned. Constructors usually initialize objects without acquiring a lock, so this refinement is important. For each expression E_v in $SharedAccess(v)$, we compute $Protect(v, E_v)$: the set of locks the thread is holding at E_v protecting v . Since a lock must protect a variable everywhere the variable is shared:

$$Protect(v) = \bigcap_{E_v \in SharedAccess(v)} Protect(v, E_v)$$

If the lock is a summary node, then the variable must be a field of a summary node; the interpretation is that each variable object is protected by a unique lock object.

Given an expression E_v accessing v , we compute $Protect(v, E_v)$ as follows. We say E_ℓ is a *lock expression* at E_v if it is the argument to some enclosing `synchronized` statement. For each E_ℓ , we define a triple (E_r, S_ℓ, S_v) where E_r is the *root expression*, which is the common prefix of E_ℓ and E_v , S_ℓ is the *lock selector*, which is the part of E_ℓ not in E_r , and S_v is the *variable selector*, which is the part of E_v not in E_r and with the final selector removed (i.e., $E_r.S_v$ is a reference to the object containing v , not v itself). For example, consider the expression `hours` in method `updateHours` in Figure 5. The fully qualified expression accessing the variable is `this.hours`, a lock expression is `this.hoursLock`, and this pair yields the triple $(this, hoursLock, \lambda)$. Note that $S_\ell = \lambda$ indicates the lock and root objects are the same, while $S_v = \lambda$ indicates that the variable and root objects are the same.

Given E_v and (E_r, S_ℓ, S_v) , we identify a candidate lock ℓ in the SSG as follows. For an SSG node n and a selector S , $n.S$ is the set of nodes reached from n by following S , while

$$S^{-1}(v) = \{n \mid v \text{ is a field of an object in } n.S\}$$

is the set of SSG nodes such that applying selector S to these nodes may reach the object containing variable v . First, in the pre-SSG for E_v , we compute the set of possible root objects for E_v 's access to v :

$$R = r\text{-value}(E_r) \cap S_v^{-1}(v)$$

If R contains exactly one node, then this node is the candidate root r and we compute the set of possible locks $L = r.S_\ell$ in the pre-SSG of E_ℓ . If L contains exactly one node ℓ , then this node is the candidate lock.

We include ℓ in $Protect(v, E_v)$ if we can deduce from the SSGs that, for each instance of v at run-time, there is a unique instance of ℓ held by the thread. Note that this does not follow immediately since r , ℓ , and the SSG nodes on the paths from r to ℓ and from r to v might represent multiple objects at run-time. Nevertheless, we can still conclude that for each variable represented by v at run-time there is a unique lock represented by ℓ if all of the following are true:

1. For each variable represented by v at run-time, there is a unique root object represented by r . This holds provided $S_v = \lambda$, r is a current node, or all arcs on the path selected by S_v are one-to-one arcs between summary nodes.
2. For each root object represented by r at run-time, there is a unique lock object represented by ℓ . This holds provided that $S_\ell = \lambda$, ℓ is a current node, or all arcs on the path selected by S_ℓ are one-to-one arcs between summary nodes.
3. No field accessed in E_r is written between E_ℓ and E_v (otherwise the root object at E_v might be different than the root object at E_ℓ).

A variable v is *protected* if $Protect(v)$ is nonempty. A transformation may be made invisible if all variables it might access are protected.

Note that inaccuracy in the reference analysis leads to a *larger* model, not an incorrect model. If we cannot determine that a variable is owned or protected then a transformation accessing that variable will be visible; the transition system will have more states, but will still be represent all behaviors of the (possibly restricted) program.

7 Empirical Results

To demonstrate the effectiveness of our reductions, we have implemented the state-space reduction method as part of a prototype tool and we have applied this tool to several program fragments, including the two examples shown in Figures 1–2. In this section, we briefly describe the implementation of the tool and present the results of applying it to these two examples.

The prototype is a model builder, not a model extractor, thus the user must extract the relevant part of the program and perform all necessary abstractions by hand before feeding the abstracted/restricted Java source to the tool (e.g., we commented out the increment of `next` in the `Producer.run()` method of Figure 1 to avoid modeling the contents of the buffer). The tool accepts a Java source file containing one or more classes (exactly one of which has a `main()` method), and constructs a finite-state model of the program. The state-space reductions described here are employed if a command-line flag is set.

The tool is written in Java and uses JavaCC, which is a parser/scanner generator for Java, and JJTree, which works with JavaCC to produce an abstract syntax tree. By traversing the abstract syntax tree, we construct a control flow graph (CFG) for the `main()` method and each `run()` method, inlining method calls and constructors as they are encountered. Each arc in the CFG is a transformation that represents a bytecode instruction; most transformations are marked as invisible in the base model (as described in Section 4.1). If no reductions are to be applied, we perform a depth first search of the program's state space. At each state, we generate a transition for each ready thread that represents the execution of that thread up to the next visible transformation (invisible transformations are composed with their successors on-the-fly while generating the state space). If the reductions are to be applied, we perform the reference analysis described in Section 5. We then apply the reductions described in Section 4 and mark additional transformations invisible before generating the state space.

For the following examples, we set f (the property to be verified) to check for deadlock. In our models, the requirement that there be no deadlock is expressed by the formula

$$\neg \diamond \bigwedge_{i \in \text{Threads}} \neg \text{ready}[i]$$

Problem Size	Base Model		Reduced Model	
	Transforms	States	Transforms	States
1	55	445	13	56
2	82	3323	19	317
3	109	25481	25	2140
4	136	out of mem	31	17043

Table 1: Results for Producer/Consumer System

Problem Size	Base Model		Reduced Model	
	Transforms	States	Transforms	States
1	49	85	3	5
2	78	1400	15	141
3	107	26367	22	1203
4	136	out of mem	29	10671
5	165	out of mem	36	98967

Table 2: Results for Observer System

(i.e., it is never eventually the case that all threads are blocked). Note that the only f -observable transformations are those that set $ready[i]$ to false—the *wait* and *monitor_enter* (lock not available) transformations.

The transition system models we construct could be translated into the input languages of existing finite-state verification tools (e.g., SPIN [19] or SMV [28]) for analysis, as in [9]. Note that our method is performed before the state-space generation/analysis and simply reduces the number of transformations defining the transition system. Although our prototype tool performs a simple state enumeration on the transition system, other analysis tools may use different more efficient techniques, such as partial order reductions or symbolic model checking. Our reductions are completely independent of whatever method is used to analyze the resulting transition system.

Table 1 shows the results of applying the tool to several sizes of the producer/consumer system of Figure 1. We scaled the problem by adding `Consumer` threads. The columns show the problem size (number of consumers), total number of visible transformations and states in the base model, and the total number of visible transformations and states in the reduced model. On this example, the reductions reduce the number of states in the model by at least an order of magnitude. In particular, the reductions allow the following transformations to be made invisible:

- All accesses to the `next` variable of the `Producer` (it is owned by the producer thread).
- Accesses to the `IntBuffer` variables that occur before the `Producer` object is passed to the `Thread` allocator (these variables are owned by the main thread until that point).
- All accesses to the `IntBuffer` variables and the nested integer array `data` (these variables are protected by the lock on the `IntBuffer` object after they become shared).
- The *notifyAll* transformations (these are merged with the *monitor_exit* transformations that follow them).

Table 2 shows the results of applying the tool to several sizes of the observer system of Figure 2. We scaled the problem by adding `Mutator` threads. The columns show the problem size (number of mutators), total number of visible transformations and states in the base model, and the total number of visible transformations and states in the reduced model. Note that, with only one `Mutator`, there is a change of ownership for the `Subjects` and `Observers`, thus we achieve greater reduction for the size 1 version. On this example also, the reductions reduce the number of states the model by at least an order of magnitude. In particular, the reductions allow the following transformations to be made invisible:

- All accesses to the variables in the `Subjects` and `Observers` by the main thread up until the `Mutator` is passed to the `Thread` constructor (these variables are owned by the main thread until this point).

- For the size 1 version of the problem only, all accesses to the variables and locks of the Subjects and Observers in the mutator thread (there is a transfer of ownership when the Mutator object is passed to the Thread constructor, so these variables and locks are owned by the mutator thread after this point).
- All accesses to the variables `state` and `myObserver` in Subjects and all accesses to variables `subject` and `cachedState` in Observers (these variables are protected by the lock on the Subject). The next variable of a Subject is not protected since it is referenced without a lock in `Mutator.run()`.
- All `monitor_enter` and `monitor_exit` transformations for the Observer object (the lock on the Observer is protected by the lock on the enclosing Subject).
- The reacquisition (and corresponding release) of the lock on the Subject when the observer calls `queryState()` in `changeNotification()` (at which point the thread already holds the lock for that Subject).

On both of these examples, applying the reductions took less than half a second of CPU time on a DEC Alpha 500a with 128 Mb of memory. Our simple state enumeration took much longer (several hours for the largest example), but this part of the prototype was intended primarily to evaluate the effectiveness of the reductions by calculating the number of reachable states. Eventually we intend to use an existing finite-state verifier to perform the analysis of the reduced model.

8 Related Work

As mentioned in Section 1, most previous work on constructing finite-state models from concurrent programs has used Ada, which before Java was probably the most widely used language supporting concurrency. With Java's rapid rise in popularity, interest in concurrency analysis for Java has begun to grow.

Demartini and Sisto [13] show how to translate most of Java's concurrency constructs into Petri nets and PROMELA⁸, which can be analyzed by other tools. Naumovich, Avrunin, and Clarke [30] show how to adapt a dataflow-based analysis method for Ada to Java's concurrency constructs. In both cases, the work is focussed on representing the semantics of Java monitors in another formalism rather than on reducing the size of the resulting model.

The state explosion problem has been attacked in many ways. Virtual coarsening is but one method for state-space reduction. Partial-order methods [17,31,33] generalize this approach by defining a *dependency relation* on the transformations; traces that differ only in the order of independent transformations are equivalent and only one representative from each equivalence class must be represented in the model [27]. Another widely used method for alleviating the state explosion, especially for hardware designs, is *symbolic model checking* [4, 5, 28], which involves representing the states symbolically, usually with Ordered Binary Decision Diagrams (OBDDs). Compositional methods [7, 8, 35] exploit modularity in a system by dividing it into smaller subsystems, verifying each subsystem, and then combining the results of these analyses to verify the full system. There are also methods based on dataflow analysis [16, 26] and integer programming [2, 29]. As we mentioned in Section 7, our reductions are applied to the transformations defining the transition system and can thus be combined with any of these analysis methods.

Shape analysis is an active area of research. Most shape analysis algorithms, including Chase *et al*'s full algorithm, are more complex than what we presented here. The goal of such algorithms is to preserve as much of the shape of common linked structures as possible. For example, the recent algorithm of Sagiv *et al* [32] can determine that, if the input to a linked list insertion routine is a linked list, the output will also be a linked list. It's not clear whether this kind of information is useful for model reduction, however. We also note that name-based alias analysis algorithms like [22] are generally faster than graph-based algorithms like the one we propose, but as noted in Section 5.5, given the context in which the analysis is to be performed, we do not anticipate the expense of the shape analysis to be a limiting factor.

9 Conclusion

We have shown how shape analysis can be used to reduce the size of finite-state models of concurrent Java programs by determining which variables are owned by threads or protected by locks. We have also given several state-space re-

⁸The input language for the SPIN verifier.

ductions based on the semantics of Java monitors. Our methods exploit common features of concurrent Java programs, including data accessed by only one thread, and encapsulated data protected by a lock.

The process of extracting models from source code must, to some degree, be depended on the source language. Although our presentation was restricted to Java, many aspects of our method are more widely applicable and could be used to reduce finite-state models of programs with heap data and/or a monitor-like synchronization primitive (e.g., Ada's protected types).

Together with colleagues, we are in the process of building a toolset to support the extraction of finite-state models from Java programs. The new toolset will allow extraction of models from bytecode files, support partial evaluation and abstract interpretation methods for reducing the model, and will output the model in a form that is directly processable by existing finite-state verifiers (e.g., SPIN, SMV). The method described here will be incorporated into this toolset.

With the arrival of Java, concurrent programming has entered the mainstream. Finite-state verification technology offers a powerful means to find concurrency errors, which are often subtle and difficult to reproduce. Unfortunately, extracting the finite-state model of a program required by existing verifiers is tedious and error-prone. As a result, widespread use of this technology is unlikely until the extraction of compact mathematical models from real software artifacts is largely automated. Methods like the one described here will be essential to support such extraction.

Acknowledgements

Thanks are due to George Avrunin for helpful comments on a draft of this paper.

References

- [1] E. Ashcroft and Z. Manna. Formalization of properties of parallel programs. *Machine Intelligence*, 6:17–41, 1971.
- [2] G. S. Avrunin, U. A. Buy, J. C. Corbett, L. K. Dillon, and J. C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Trans. Softw. Eng.*, 17(11):1204–1222, Nov. 1991.
- [3] T. Bultan, J. Fisher, and R. Gerber. Compositional verification by model checking for counter examples. In Ziel [38], pages 224–238.
- [4] T. Bultan, R. Gerber, and C. League. Verifying systems with integer constraints and linear predicates: A composite approach. In Young [36], pages 113–123.
- [5] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.
- [6] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI)*, pages 296–310, June 1990.
- [7] S. C. Cheung and J. Kramer. Enhancing compositional reachability analysis using context constraints. In *Proceedings of the first ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 115–125, Dec. 1993.
- [8] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics based tool for the verification of concurrent systems. *ACM Trans. Prog. Lang. Syst.*, 15(1):36–72, Jan. 1993.
- [9] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Trans. Softw. Eng.*, 22(3):161–180, 1996.
- [10] J. C. Corbett. Timing analysis of Ada tasking programs. *IEEE Trans. Softw. Eng.*, 22(7):461–483, 1996.
- [11] J. C. Corbett. Constructing compact models of concurrent Java programs. In Young [36].

- [12] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 238–252, New York, 1977. ACM Press.
- [13] C. Demartini and R. Sisto. Static analysis of Java multithreaded and distributed applications. In *Proceedings of the 1998 Workshop on Software Engineering for Parallel and Distributed Systems*, 1998.
- [14] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design*, October 1992.
- [15] S. Duri, U. Buy, R. Devarapalli, and S. M. Shatz. Using state space reduction methods for deadlock analysis in Ada tasking. In T. Ostrand and E. Weyuker, editors, *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, pages 51–60, New York, June 1993. ACM Press.
- [16] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In D. Wile, editor, *Proceedings of the Second Symposium on Foundations of Software Engineering*, pages 62–75, Dec. 1994.
- [17] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In K. G. Larsen and A. Skou, editors, *Computer Aided Verification, 3rd International Workshop Proceedings*, volume 575 of *Lecture Notes in Computer Science*, pages 332–242, Aalborg, Denmark, July 1991. Springer-Verlag.
- [18] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, Reading, Massachusetts, 1996.
- [19] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.
- [20] G. J. Holzmann. Designing executable abstractions. In M. Ardis, editor, *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 103–108, March 1998.
- [21] D. Jackson and C. A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. In Ziel [38], pages 239–249.
- [22] W. Landi and B. Ryder. Pointer induced aliasing: A problem classification. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 93–103, New York, 1991. ACM press.
- [23] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Reading, Massachusetts, 1997.
- [24] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specifications. In *Proceedings of the Twelfth ACM Symposium on the Principles of Programming Languages*, pages 97–105, 1985.
- [25] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, Reading, Massachusetts, 1997.
- [26] S. P. Masticola and B. G. Ryder. Static infinite wait anomaly detection in polynomial time. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume II, pages 78–87, 1990.
- [27] A. Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324, Berlin, 1987. Springer-Verlag.
- [28] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.
- [29] T. Murata, B. Shenker, and S. M. Shatz. Detection of Ada static deadlocks using Petri net invariants. *IEEE Trans. Softw. Eng.*, 15(3):314–326, 1989.
- [30] G. Naumovich, G. S. Avrunin, and L. A. Clarke. Data flow analysis for checking properties of concurrent Java programs. Technical Report 98-22, University of Massachusetts, 1998. Submitted for publication.

- [31] D. Peled. All for one, one for all: On model checking with representatives. In C. Courcoubetis, editor, *Computer Aided Verification, 5th International Conference Proceedings*, pages 409–423, Elounda, Greece, 1993.
- [32] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.
- [33] A. Valmari. A stubborn attack on state explosion. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification '90*, number 3 in DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 25–41, Providence, RI, 1991. American Mathematical Society.
- [34] A. Vermeulen. Java deadlock. *Dr. Dobb's Journal*, September 1997.
- [35] W. J. Yeh and M. Young. Compositional reachability analysis using process algebra. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*, pages 178–187, New York, Oct. 1991. ACM SIGSOFT, Association for Computing Machinery.
- [36] M. Young, editor. *Proceedings of the 1998 International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, March 1998.
- [37] M. Young, R. N. Taylor, K. Forester, and D. Brodbeck. Integrated concurrency analysis in a software development environment. In R. A. Kemmerer, editor, *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis and Verification*, pages 200–209, 1989. Appeared as *Software Engineering Notes*, 14(8).
- [38] S. Ziel, editor. *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, January 1996.