Constructing Compact Models of Concurrent Java Programs

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

Abstract

Finite-state verification technology (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. In this paper, we explore the extraction of compact concurrency models from Java code. In particular, we show how static pointer 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.

Keywords

Static Analysis, Model Extraction, Finite-state Verification

1 Introduction

Finite-state analysis tools (e.g., model checkers) can automatically detect concurrency errors, which are often subtle and difficult to reproduce. Before such tools can be applied to software, a finite-state model of the program must be constructed. This model must be accurate enough to verify the requirements and yet abstract enough to make the analysis tractable. In this paper, we consider the problem of constructing such models for concurrent Java programs.

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).

Ideally, an analysis tool could extract a model from a program and use the model to verify some property of the program (e.g., freedom from deadlock). In practice, extracting concurrency models is difficult to automate completely. In order to obtain a model small enough for a tractable analysis, an analyst must assist most existing tools by specifying what aspects of the program to model. In particular, the representation of certain variables is often necessary to make the model sufficiently accurate, but these variables must often be abstracted or restricted to make the analysis tractable. Although a model that restricts the range of a variable does not represent all possible behaviors of the program and thus cannot technically be used to verify the program has a property, the conventional wisdom is that most concurrency errors are present in small versions of a system[6, 9], thus these models can still be useful for finding errors (testing).

Most previous work on concurrency analysis of software has used Ada [7, 13, 12, 2, 3, 8]. Although some aspects of these methods can also be applied to Java programs, the Java language presents several new challenges/opportunities:

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.
2. Java threads must be created dynamically, thus is it impossible (in general) to determine how many threads a program will create. Although Ada tasks may be created dynamically, many concurrent Ada programs contain only statically allocated tasks.
3. Java has a locking mechanism to synchronize access to shared data. This can be exploited to reduce the size of the model.

The main contribution of this paper is to show how static pointer analysis can be used to reduce the size of finite-state models of concurrent Java programs. The method employs *virtual coarsening* [1], a well-known technique for reducing the size of concurrency models by collapsing invisible actions (e.g., updates to variables that are local or protected by a lock) into adjacent visible actions. The static pointer analysis is used to construct an approximation of the run-time structure of the heap at each statement. This information can be used to identify which heap objects are actually local to a thread, and which locks guard access to which variables.

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee.
ISSTA 98 Clearwater Beach Florida USA
Copyright 1998 0-89791-971-8/98/ 03..$5.00

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 static pointer analysis in Section 5. Section 6 shows how to use the heap structure information to apply the reductions. Finally, Section 7 concludes.

2 Concurrency in Java

Java's essential concurrency features are illustrated by the familiar bounded buffer system 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. Although threads may be assigned priorities to control scheduling, in this paper we assume all (modeled) threads have equal priority and are scheduled arbitrarily; this captures all possible executions.

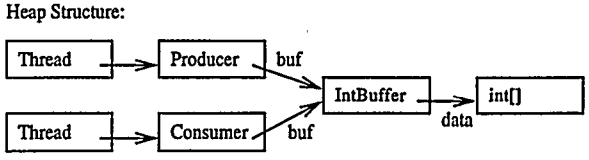
In the example, the program begins with the execution of the static method `main()` by the main thread. This creates an instance of an `IntBuffer`, creates instances of `Producer` and `Consumer` that point to this `IntBuffer`, creates instances of `Thread` that point to the `Producer` and `Consumer`, 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 bounded buffer problem. 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. 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: 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()` 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 executes the `notifyAll()` method, which wakes up all threads waiting on the object (these threads must reacquire the object's lock before returning from `wait()`).

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




```
public 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;
    }
  }
}

public 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++);
    }
  }
}

public 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: Bounded Buffer Example

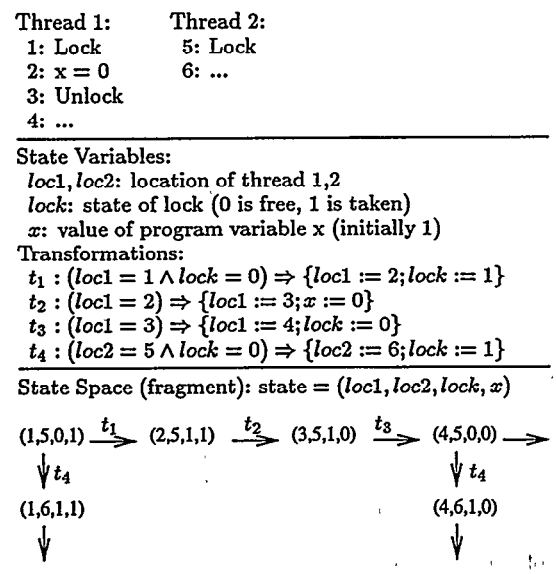


Figure 2: Example of Transition System

- $S = D_1 \times \ldots \times D_n$ is a set of *states*. A state is an assignment of values to a finite set of *state variables* $v_1, \ldots, v_n$ 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:

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

such that $(s_i, s_{i+1}) \in T$ for all $i = 0, \ldots, n-1$.

The method of constructing the transition system representing a Java program is similar to the method presented in [5] 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., whether each thread is ready, running, or blocked on some object). Each transformation represents the execution of a byte-code instruction for some 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. The small example in Figure 2 gives the flavor of the translation.

The Java heap must also be represented. We bound the number of states in the model by limiting the number of instances of each class (including `Thread`) that may exist simultaneously. For this paper, we assume these limits are provided by the analyst. If class C has instance limit k_C, when a program attempts to allocate an instance of class C at a point where k_C instances are still accessible (Java uses garbage collection), the transition system goes 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.

Consider again the bounded buffer example in Figure 1. We could generate a restricted model of this program by representing all the variables but restricting their ranges. By restricting the variables representing the contents of the buffer to $\{0, 1\}$ and the variables representing the size of the buffer to $\{0, 1, 2\}$, we would obtain a very restricted but interesting model of the program (i.e., one that would likely contain any concurrency errors).

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 is often too large to construct. Instead, we 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 amalgamating transitions. 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 2, we might replace transformations t_2 and t_3 with a single transformation $t_2 \circ t_3$ that updates x and releases the lock; we could then eliminate control location 3 from the domain of *loc1*.

We assume the requirement to be verified (tested) is specified as a stuttering-invariant formula f in linear temporal logic (LTL) [11], 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, without first constructing* (S, T).

Note that the reduced model constructed is specific to the formula f, thus the reduction must be repeated for each property verified.

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 $p_1, \ldots, p_m$ 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 $(p_1, \ldots, p_m)$ 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 $(s_0, s_1), \ldots, (s_{n-1}, s_n)$ defines a sequence of f-observations $P_f(s_0), \ldots, P_f(s_n)$, 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.

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. Below, we give two cases in which transformations representing Java code can be made invisible. In both cases, we need information about the structure of the heap at run-time to apply the reduction. We show how to collect this information in Section 5.

4.1 Local Variable Reduction

Some state variables are accessed only when a particular thread is running. For example, some program variables are locally scoped to a particular thread by the language semantics. Also, the state variable recording the control location of a thread is accessed only by that thread. Transformations that access exclusively such state variables may be made invisible provided they are not f-observable.

To understand why, consider transformation t in Figure 3. 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 $i = 1, \ldots, n$, then we replace t and t'_i for $i = 1, \ldots, n$ with $t \circ t'_i$ for $i = 1, \ldots, n$). 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 $t_1, \ldots, t_n$ representing the execution of code from other threads (i.e., not the thread of t). Combining t and t' eliminates traces in which $t_1, \ldots, t_n$ occurs between t and t'. This does not eliminate any reduced f-observation sequences, however, since executing $t_1, \ldots, t_n$ before t must produce the same reduced f-observation sequence as executing t before $t_1, \ldots, t_n$. Since t accesses only variables that $t_1, \ldots, t_n$ cannot access, t is independent of $t_1, \ldots, t_n$ and commutes: for any state s_1 in which both t and $t_{1,\ldots,n} = t_1 \circ \ldots \circ t_n$ are enabled, $t \circ t_{1,\ldots,n}(s_1) = t_{1,\ldots,n} \circ t(s_1)$. Since t is not f-observable, the trace obtained by executing $t, t_1, \ldots, t_n, t'$ must have the same reduced f-observation sequence as the trace obtained by executing $t_1, \ldots, t_n, t \circ t'$.

To use this technique, we would like to determine what variables are local to a particular Java thread (i.e., can only be referenced by that thread). A program variable is *local* to a thread if:

1. The variable is stack allocated (i.e., is declared in a method body or as a formal parameter).
2. The variable is statically allocated and referenced by at most one thread.
3. The variable is heap allocated (i.e., an instance variable of an object) and *the object is accessible only from one thread.* For example, the variable `next` of class `Producer` in Figure 1 is accessible only by the producer thread.

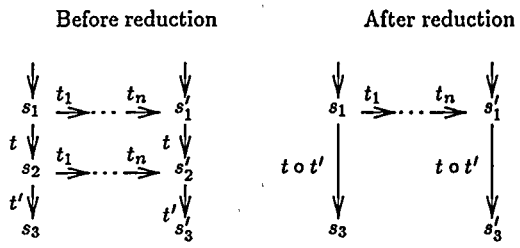


Figure 3: Reduce by combining t and t'

Case 1 is trivial to detect. Case 2 is more difficult due to the dynamic nature of thread creation, though the following conservative approximation is reasonable: a static variable may be considered local if it is accessed only by code reachable[1] 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). For example, if the variable `next` were a static member of class `Producer`, 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 local to the producer thread. Case 3 is the most difficult. Clearly if the object containing the variable is accessible only from stack or statically allocated variables that themselves are local to a specific thread (cases 1 and 2), then the heap allocated variable is also local to that thread, but determining this requires information about the accessibility of heap objects at run-time.

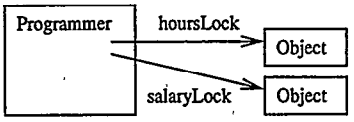
4.2 Lock Reduction

We propose another technique for virtual coarsening based on Java's locking mechanism. A transformation that updates a variable x of an instance of class C may be made invisible provided it is not f-observable and there exists an object 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 3. 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 $t_1, \ldots, t_n$ 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. In the bounded

[1] A statement s is *reachable* from a statement s' if there exists a path in the program's control flow graph from s' to s (i.e., a thread might execute s after executing s').




```
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 4: Example of Splitting Locks

buffer 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* [10]: an object X is conceptually[2] nested in an object Y by placing a reference to X in Y and accessing X only within the methods of Y.

Another common design pattern in which locks protect variables in other objects is *splitting locks* [10]. 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 4; 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. Collecting this information is the topic of the next section.

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 reductions. Understanding run-time heap structure is an important problem in compiler optimization since accurate knowledge of aliasing can improve many standard optimizations. 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* [4], which uses this basic approach. We extend Chase's algorithm in three 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. Third, we handle arrays.

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: one CFG for the `main()` method and k_C identical CFGs for each `run()` method of class C (recall k_C is the instance limit for class C). In this paper, we do not handle interprocedural analysis. We assume all procedure (method) calls have been inlined; this limits the analysis to programs 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, 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

[2] Java does not allow physical nesting of objects.

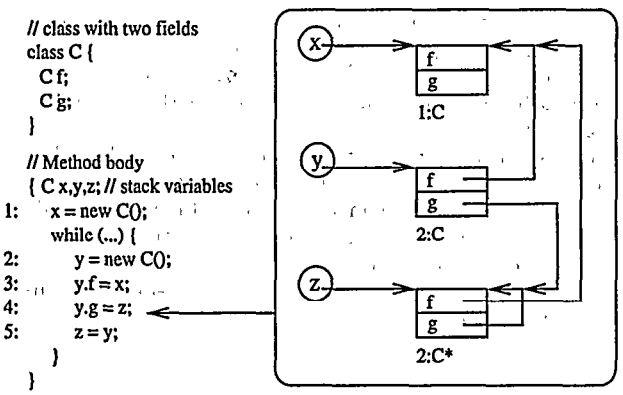


Figure 5: SSG for Statement 4

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.

An example SSG is shown in Figure 5. 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:C* represents the summary node for the allocator of class C at statement 2. We often omit disconnected nodes (e.g., the summary node for an allocator that is not in a loop). Note that the linked list is represented with a self loop on node 2:C*.

Like Chase *et al* [4], 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 the thread of its CFG), 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 as few edges as possible. 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.

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 (see the SSGs for statement 6 in Figure 6).

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 or a local static

variable. If e_1 is either a stack variable or a local static variable, we perform a *strong update* by removing all edges out of the node in *l-value*(e_1) and then adding an edge from the node in *l-value*(e_1) to each node in *r-value*(e_2). Otherwise, we perform a *weak update* by simply adding an edge from each node/field in *l-value*(e_1) to each node in *r-value*(e_2).

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`). Let X be the set of classes to which the object referenced by `x` might belong (i.e., all subclasses of the type of `x`). When the allocator is processed, we add to the worklist the start statement of every CFG for a `run()` method of a class in X (i.e., the start statement of a CFG is implicitly dependent on every thread allocator that might start[3] the thread).

When a CFG for a `run()` method of a class `C` 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 set of heap nodes for class `C` pointed to by `runnable` fields of heap nodes for class `Thread` (i.e., we do not associate a given thread/CFG with a specific thread allocator).

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 that objects allocated by A may point to objects allocated by B. We need to know if each object allocated by A points to a *different* object allocated by B; 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.

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. In fact, this is the motivation for distinguishing current from summarized instances/nodes.

[3]Technically, 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.

5.5 Example

Consider the Java source in Figure 6. The first SSG in Figure 6 is the post-SSG for statement 6 the first time it is processed (i.e., before any summary information exists). The second SSG is the result of summarizing this SSG. Note that, since nodes 3:B and 5:A are summarized together, the arc from field `a2` of 3:B* to 5:A* is labeled as one-to-one (1-1), but since 2:A is a current node, there is no one-to-one relationship between field `a1` of 3:B* and 2:A (nor would there be if a loop were added around this code and 2:A was summarized).

The last SSG is the final SSG for statement 9 (the end of the method). After statement 7, the `Thread` allocated there may have access to the A allocated by statement 2, while after statement 8, the `a1` field of some B may point to some A allocated at statement 5. Note that stack variable `b` is out of scope at statement 9 and thus can be removed from the SSG. The arc from 2:A to 0:A is added by statement 0, which is placed on the worklist when statement 7 is processed. Although we have not shown the final SSGs for statements 1-8, all these SSGs would contain this arc, even though the reference value it represents cannot appear until after statement 7; no thread scheduling information is considered.

5.6 Complexity

Given a program with S statements and V variables and allocators, our 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 do not anticipate the cost of the reference analysis to be prohibitive. First, based on the application of the algorithm to several small examples, we believe the average complexity to be much lower. 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 a few times. 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 exclusively. Also, a program requirement might involve only a small subset of the program's classes; the rest of the program need not be represented.

6 Applying the Reductions

In this section, we explain how to use the information collected by the reference analysis to apply the local variable and lock reductions.

```
  class A implements Runnable {
     A a3;
    void run() {
0:    a3 = new A();
    }
  }
  class B {
    A a1;
    A a2;
    B(A a) {
      this.a1 = a;
      this.a2 = new A();
    }
  }
  class Main {
    static B [] x;
    static void main(...) {
1:    x = new B[...];
2:    A a = new A();
      while (...) {
3:      B b = new B(a);
        { // inlined constructor
4:        b.a1 = a;
5:        b.a2 = new A();
        }
6:      x[...] = b;
      }
      if (...)
7:      (new Thread(a)).start();
      else
8:      x[...].a1 = x[...].a2;
9:  }
   }
```

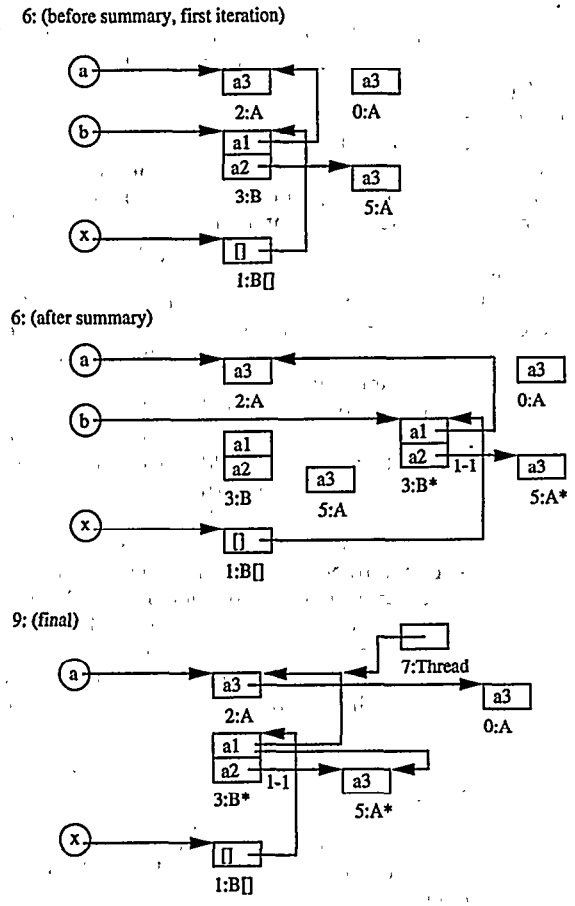


Figure 6: Reference Analysis Example

6.1 Local Variable Reduction

Applying the local variable reduction is straightforward. The set of heap nodes in an SSG that are local to a given thread are those that are accessible only from stack or static variables local to the thread. All heap variables are accessed with expressions of the form *ref.id* where *ref* is a reference expression and *id* is the name of the instance variable. The variable accessed by such an expression is local to the thread if the nodes in the r-value of *ref* are local to the thread in the pre-SSG for the statement.

Note that heap variables may be local for some statements and non-local for others. A common idiom is for an object to be allocated, initialized, and then made available to other threads (e.g., the `IntBuffer` object of the example in Figure 1). The reference analysis can determine that the instance variables of such an object are local until the object is made available to other threads.

6.2 Lock Reduction

Applying the lock reduction is more complex. We need to determine whether a variable is protected by a lock. 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 [10]. 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 accessible from the variable object, or vice versa, or both are accessible 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 accessible. Each role must be played by exactly one object, but one object may play multiple roles. For the expression `data[i]` in the bounded buffer example, 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 4, 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 accessible 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). Static variables are represented by variable nodes, heap variables by fields of heap nodes, and locks by heap nodes in the SSG. Essentially, we use the expressions accessing the variable and lock to identify the lock object; we can interpret the expressions (abstractly) using their SSGs.

Formally, for each static/heap variable v, we want to compute *Protect(v)*: the set of locks protecting v. For each such v, let *Access(v)* be the set of program expressions that

may access v; these sets can be constructed during the reference analysis. For each expression E_v in *Access(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:

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

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 4. The fully qualified expression[4] accessing the variable is `this.hours`, a lock expression is `this.hoursLock`, and this pair yields the triple (`this`, `hoursLock`, λ). 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

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

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 both 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.

A variable v is *protected* if *Protect(v)* is nonempty. A transformation may be made invisible if it is not f-observable and all variables it might access are protected or local.

Note that inaccuracy in the reference analysis leads to a *larger* model, not an incorrect model. If we cannot determine that a variable is local or protected by a lock, 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.

6.3 Example

Consider the bounded buffer example in Figure 1. The SSGs for all statements in the producer and consumer `run()` methods are isomorphic to the heap structure shown at the top of the figure (there would also be nodes for the stack variables). From these SSGs, we can deduce that variable `next` in the `Producer` object is local to the producer thread. Thus, for a formula f that does not depend on `next`, the transformation incrementing `next` may be invisible.

Also in the bounded buffer example, the variable `data[...]` in the array object and all the instance variables of the `IntBuffer` class are protected by the lock of the `IntBuffer` object. Thus, for a formula that does not depend on these variables, the sequence of transformations representing the methods `put()` and `get()` may be combined into two transformations: one to acquire the lock, the other to update the variables and release the lock.

Although a complete program is not shown for the splitting locks example of Figure 4, each allocator for `Programmer` would produce an SSG subgraph isomorphic to the heap structure shown at the top of the figure. The arcs from a summary `Programmer` node to its `Object` nodes would be one-to-one. The analysis could determine that each instance variable `hours` is protected by the `Object` accessible via field `hoursLock`.

7 Conclusion

We have proposed a method for using static pointer analysis to reduce the size of finite-state models of concurrent Java programs. Our method exploits two common design patterns in Java code: data accessible 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).

The method is currently being implemented as part of a tool intended to provide automated support for extracting finite-state models from Java source code. Although we have no empirical data on the method's performance at this time, the effectiveness of virtual coarsening for reducing concurrency models is well known, and the manual application of the method to several small examples suggests that many transitions can be made invisible for a typical formula.

With the arrival of Java, concurrent programming has entered the mainstream. Finite-state verification technology offers a powerful means to find concurrency errors, which

[4] In our analysis, the method will have been inlined and the `this` variable replaced with a new temporary holding this implicit parameter. In addition, a simple propagation analysis can be used to allow recognition of the pattern even if multiple selectors are decomposed into a series of assignments (e.g., `x.f.g` expressed as `t1 = x.f; t1.g`).

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 [14], pages 224–238.

[4] 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.

[5] J. C. Corbett. Timing analysis of Ada tasking programs. *IEEE Trans. Softw. Eng.*, 22(7):461–483, 1996.

[6] 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.

[7] 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.

[8] 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.

[9] D. Jackson and C. A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. In Ziel [14], pages 239–249.

[10] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Reading, Massachusetts, 1997.

[11] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specifications. In *Proceedings of the Twelfth ACM Symposium on the Priciples of Programming Languages*, pages 97–105, 1985.

[12] 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.

[13] 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).

[14] S. Ziel, editor. *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, January 1996.