Data Flow Analysis for Checking Properties of Concurrent Java Programs*

Gleb Naumovich
University of Massachusetts
Department of Computer Science
Amherst, MA 01003
+1 413 545 2013
naumovic@cs.umass.edu

George S. Avrunin
University of Massachusetts
Department of Mathematics
and Statistics
Amherst, MA 01003-4515
+1 413 545 4251
avrunin@math.umass.edu

Lori A. Clarke
University of Massachusetts
Department of Computer Science
Amherst, MA 01003
+1 413 545 2013
clarke@cs.umass.edu

ABSTRACT

In this paper we show how the FLAVERS data flow analysis technique, originally formulated for systems using a rendezvous concurrency model, can be applied to the various concurrency models used in Java programs. The general approach of FLAVERS is based on modeling a concurrent system as a flow graph and, using a data flow analysis algorithm over this graph, statically checking if a property holds on all (or no) executions of the program. The accuracy of this analysis can be iteratively improved, as needed, by supplying additional constraints, represented as finite state automata, to the data flow analysis algorithm.

In this paper we present an approach for analyzing Java programs that uses the constraint mechanism to model the possible communications among threads in Java programs, instead of representing them directly in the flow graph model. We also discuss a number of error-prone thread communication patterns that can arise in Java and describe how FLAVERS can be used to check for the presence of these. A preliminary evaluation of this approach is carried out by analyzing some small concurrent Java programs for these error-prone communication patterns and other, program-specific, faults.

KEYWORDS

Static analysis, data flow, concurrency, Java.

1 INTRODUCTION

With the advent of Web technology, distributed program-

ming, especially in the Java programming language, is growing rapidly in popularity. The additional complexity and inherent non-determinism of distributed systems makes understanding and reasoning about them extremely difficult. Moreover, testing such systems is problematic since, not only are there many more alternatives to explore when task interleaving is considered, but two executions of the same program with the same test data may not even produce the same results. Static analysis techniques are being developed for distributed systems to complement traditional testing approaches. These techniques statically determine if specific kinds of faults can occur on any executions of the system. In this paper, we describe how the FLAVERS static analysis approach can be modified to handle the Java concurrency constructs. In addition, we present a number of patterns of use of Java's concurrency constructs that could lead to erroneous behavior and then describe how the modified version of FLAVERS could be applied to detect these problematic or suspicious patterns.

FLAVERS (FLow Analysis for VERification of Systems) uses data flow analysis techniques to verify user-specified properties of software systems [5]. The attractiveness of this approach is in its low-order polynomial complexity bounds and its ability to improve the precision of the analysis by incrementally improving the accuracy of the program model. A prototype for FLAVERS has been implemented, called FLAVERS/Ada, that analyzes Ada programs or program models that use rendezvous communications.

In FLAVERS/Ada, programs are modeled as *trace flow graphs* that represent the potential flow of control through the program, including intertask communications and interleavings. Additional information, represented as finite state automata and called *feasibility constraints*, is used to elaborate the semantics of selected aspects of the program when needed to increase the precision of the analysis.

The emphasis of this paper is on modeling Java programs in a way that can be used by FLAVERS. The modification of the system model is not trivial, since Ada and Java use significantly different concurrency models. We describe one promising approach in which the semantics

of thread communications are represented with feasibility constraints, instead of being a part of the trace flow graph. In addition, we discuss a number of application-independent patterns of thread communications that indicate erroneous or error-prone code and discuss the use of FLAVERS for checking for the presence of such patterns. We present an initial empirical exploration that seems to support our hypothesis that the proposed approach is capable of efficiently checking both general and application-specific properties of concurrent Java programs.

The next section gives a brief description of related work. Section 3 gives a short overview of the FLAVERS approach for Ada. Section 4 first provides an introduction to the Java concurrency constructs and then presents the proposed program model for Java. Section 5 describes some suspicious patterns of thread communications. We present initial experimental results with a prototype implementation of the proposed approach in Section 6. Finally, we present a summary and describe our future directions.

2 RELATED WORK

Most work in the area of static analysis of concurrent and distributed systems has used either synchronous communication models with the rendezvous style of concurrency or asynchronous message-passing communication models. These models are different from the Java model, which supports monitors and a mixture of low-level thread synchronization primitives.

There has been some recent work concerned with modeling Java programs. Corbett [2] describes a technique for constructing compact finite state models for Java. This approach relies on a data flow algorithm for constructing an approximation of the run-time structure of the program heap that is then used to reduce the size of the concurrency model. This alias resolution approach could also be used to reduce the size of our trace flow graph program model. In this paper, however, we have not focused on the optimization of the program model.

Demartini and Sisto [4] describe two models of Java programs. The first represents Java programs with Petri nets and the second represents Java programs with Promela code. Both these models are intended to be used for reachability analysis. While several approaches have been proposed to improve the performance of reachability analysis, in general the use of reachability analysis for real software systems remains prohibitively expensive.

As an alternative to techniques with exponential worst-case bounds, such as reachability analysis (e.g. [6,8]), symbolic model checking (e.g. [12]), and integer necessary conditions [3], data flow analyses for concurrent software have been formulated with low-order polynomial execution time and storage bounds. Most of these data flow approaches check application-independent properties (e.g. [1,11,18]),

```
task body T1 is          task body T2 is
begin                    begin
   p1;                      p2;
   T2.E;                    accept E;
end T1;                     p3;
                         end T2;
```

Figure 1: Ada code example

such as deadlock. FLAVERS is one of the few data flow techniques capable of directly checking application-specific properties of concurrent software. This approach attempts to verify the property of interest for a software system using an efficient low-order polynomial algorithm while giving the user the ability to change the amount of detail modeled, and thus to improve the precision of the results, without having to rebuild the complete model of the system.

3 FLAVERS FOR ADA

With FLAVERS/Ada, programs are modeled by trace flow graphs (TFGs). The TFG for a concurrent program is based on the control flow graphs (CFGs) for the components of a system. For each CFG we identify the nodes that correspond to observable activities in the program that an analyst wants to reason about. Each node is labeled with an *event*, a user-selected name associated with such an observable activity. To reduce the size of the representation and consequently improve the efficiency of the analysis, the CFGs are refined to remove all nodes that are not labeled with an event. In addition, any node that invokes a procedure or function is replaced by the reduced CFG representation of that routine. In our experience, this inlining of routines does not cause a severe blow-up in the size of the CFGs, since the nodes annotated with events tend to be relatively sparse.

The TFG for an Ada program is obtained by connecting the reduced, inlined CFGs for all tasks. Unique *initial* and *final* nodes that represent the start and the end states of the program respectively are connected to the CFG of each task in the program. Each possible task synchronization is represented by a *communication* node, which is connected by edges to the appropriate nodes in the CFGs of both communicating tasks. In addition, *may immediately precede (MIP)* edges are added between nodes in separate CFGs to represent possible interleavings of the actions associated with these nodes. The set of such edges can be computed efficiently [13].

Figure 1 contains a trivial example of Ada code that is used for illustration purposes. In this example, task T1 first calls procedure p1, then calls entry E of task T1, and finally terminates. Task T2 first calls procedure p2, accepts an entry call for entry E, calls procedure p3, and finally terminates. We assume that procedures p1, p2, and p3 con-
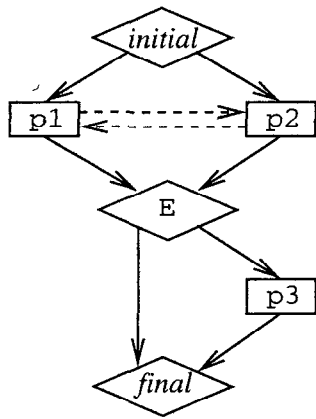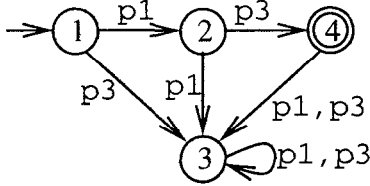
Figure 2: TFG for the example in Figure 1

Figure 3: Property for the example in Figure 1

tain no communication statements (i.e. entry calls and accepts). The corresponding TFG is shown in Figure 2. The rectangular-shaped nodes represent local computations in the tasks (procedure calls in this example). The diamond-shaped nodes labeled *initial* and *final* represent the start and the end of the program computation. Finally, the diamond-shaped node labeled E represents the rendezvous between the two tasks on entry E. Solid edges represent control flow local to the tasks, while dashed edges are MIP edges.

The set of all events associated with a model of the program is the *alphabet* of the TFG. For the TFG in Figure 2, the events of interest are the procedure calls and the rendezvous. The *language* of the TFG is the set of event sequences that occur on paths from the initial node to the final node. The resulting TFG overapproximates the set of possible sequences of these events in the sense that each real program execution must correspond to a path through the graph but some paths in the TFG may not correspond to any possible execution.

Properties can be described in a number of specification languages but are represented internally as deterministic finite state automata (FSA) over the TFG alphabet. Figure 3 gives a sample property for the example in Figure 1. This property states that on all executions of this program the call to procedure p3 must always follow the call to procedure p1 (and p1 must always be called). (Note that if the call to p3 happens first, the transition from the initial state of the property to the non-accepting sink state 3 is taken.)
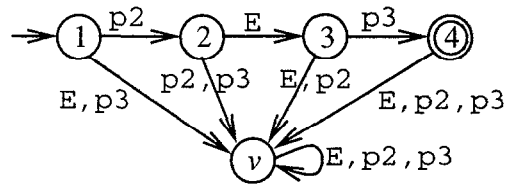

Figure 4: A feasibility constraint for the example in Figure 1

The set of all events used in the property FSA represents the alphabet of the property. The language of a property is the set of all event sequences accepted by its FSA. Conceptually, a property holds for a program if the projection of the language of the TFG on the alphabet of the property is contained in the language of this property. Data flow analysis is used to solve this containment problem by incrementally computing the set of property automata states that could be associated with the execution of each node in the TFG. This is a well-formed data flow analysis problem that can be shown to terminate, produce conservative results, and have a worst-case complexity of $O(SN^2)$, where S is the number states in the property FSA and N is the number of nodes in the TFG.

If the analysis finds that a property holds, then it is guaranteed to be valid on all possible program executions. When the analysis indicates that the property does not hold on some paths through the TFG, this may be because the program is in error or it may be because all the paths in the program model that violate this property do not correspond to feasible program executions. For example, one of the possible paths in the TFG in Figure 2 is *initial*, p1, E, *final*. But this path does not represent a legal execution of the program because, before this program can terminate, task T2 must execute procedures p2 and p3. FLAVERS provides a means for selectively removing infeasible paths from consideration by allowing the analyst to add feasibility constraints, finite state automata that model semantic restrictions on the program execution that are not reflected in the TFG. For example, CFGs, and the TFGs constructed from them, do not model the values assigned to variables during execution. Thus, paths through the TFG may not represent feasible executions because these paths do not respect the values of some variables. A feasibility constraint could be constructed to track the possible finite values or ranges of values of such a variable, thereby eliminating these infeasible paths.

An example of a feasibility constraint is shown in Figure 4. This constraint assures that the control flow within task T2 is followed. For instance, if the control path *initial*, p1, E, *final* is followed in the TFG in Figure 2, the start state of the constraint automaton is associated with node *initial* and then propagates to node p1, since p1 is not in the alphabet of this constraint (it is not an event in task T2). When this

state 1 is propagated to node E, the constraint enters the violation state v, indicating that this path represents an infeasible execution.

Each feasibility constraint has a distinct *violation* state, which signifies that the sequence of events applied to the constraint does not correspond to any legal behavior of the program. The properties to be checked for a program and the feasibility constraints are combined into a single *product automaton* with the following characteristics: (1) The product automaton accepts a sequence of events if and only if this sequence is accepted by the property automaton; (2) The product automaton goes to the violation state if and only if at least one of the constraints goes to its violation state. In practice, we use an efficient approach where the full product automaton is not actually created [16].

The containment problem on the property automaton is replaced with the containment problem on the product automaton. We say that a property holds subject to the feasibility constraints if all event sequences from the TFG language that do not send the product automaton to the violation state are accepted by this product automaton. The problem of determining if this is the case is solved by data flow analysis, which propagates the states of the product automaton through the TFG. This *state propagation* phase of the analysis involves computing, for each node in the TFG, the set of product automata states that characterize the state of the program immediately after execution of the code represented by this node. Because the data flow problem solved by state propagation is distributive, the solution of this data flow problem converges to a *join over all paths* solution [10], and so we need to look only at the final node of the TFG to determine whether the property holds. We say that a property *holds* on all terminating executions of the program if after all violation states are discarded from the final node of the TFG, only accepting states of the product automaton are present there[1].

Note that the effectiveness of FLAVERS in part depends on the user's ability to identify the aspects of the system that have to be modeled with feasibility constraints. To facilitate this task, when FLAVERS determines that the property does not hold, it produces one or more sample paths through the TFG and/or source code. The user can then decide whether these paths correspond to feasible or infeasible executions of the system. Furthermore, if a path corresponds to an infeasible execution, the reason for this infeasibility often suggests a specific constraint that models the aspect of the system that is misinterpreted in the path. FLAVERS provides automated support for many of the kinds of constraints that are usually needed.

4 ANALYSIS OF CONCURRENT JAVA PROGRAMS

[1]As described here, only terminating executions are considered.

```
class Thread1 extends Thread
{
  public Thread1() {...}
  public void run()
  {
    ...
  }
}

class Thread2 extends Thread
{
  public Thread2(Thread other,
                 Object lock) {...}
  public void run()
  {
    synchronized (lock)
    {
      t1.join();
      ...
    }
  }
}
```

```
class Example extends Thread
{
  public static void
  main(String [] args)
  {
    Object lock = new Object();
    Thread1 t1 = new Thread1();
    Thread2 t2 =
      new Thread2(t1, lock);
    synchronized (lock)
    {
      t2.start();
      t1.start();
    }
  }
}
```

Figure 5: Java code example

In this section we discuss the concurrency model employed by Java, highlight the troublesome aspects of dealing with this model in a static manner, and describe our approach to building models of Java programs in a way amenable to FLAVERS analysis. The approach that we take in this modeling is to use the feasibility constraint mechanism to represent thread interactions in Java, as opposed to incorporating these interactions in the TFG as done in FLAVERS/Ada.

Java Model of Concurrency

In Java, concurrency is modeled with *threads*. Although the term thread is used in the Java literature to refer to both thread objects and thread types, in this paper we call thread types *thread classes* and thread instances simply *threads*. Figure 5 contains an example in which thread classes Thread1 and Thread2 are defined by extending the standard Java Thread class. Threads t1 and t2 of these two respective classes are created and used in the main method of class Example.

Any Java application must contain a main() method, which serves as the "main" thread of execution. This is the only thread that is running when the program is started. Although the object containing this method does not have to extend the Thread class, it is a separate thread of control.

In Java, execution of all threads, except the main thread, is started by calling their start() methods. The run() method of a thread is never called explicitly but is invoked implicitly as a result of calling the start() method of this thread. Since only the main thread is running initially, in multi-threaded programs the main thread must instantiate and start some of the other threads. These threads may then instantiate and start other threads. For example, in Figure 5 the main thread creates (by calling the appropriate constructors) thread t1 of class Thread1 and thread t2 of class Thread2 and then starts each by invoking their

start() methods.

Java uses shared memory as the basic model for communications among threads. In addition, threads can affect the execution of other threads in a number of other ways, such as dynamically starting a thread or joining with another thread, which blocks the caller thread until the other thread finishes.

The most significant of the Java thread interaction mechanisms is based on monitors. A monitor is a portion of code (usually, but not necessarily, within a single object) in which only one thread is allowed to run at a time. Java implements this notion with synchronized statements and *locks*. Each Java object has an implicit lock, which may be used by synchronized statements[2]. To execute a synchronized statement, a thread must acquire the lock of the object indicated by this statement, and it releases this lock when it exits this synchronized statement. Since only one thread may be in possession of any given lock at any given time, this means that at most one thread at a time may be executing in one of the synchronized statements protected by that lock. In Figure 5, an object lock of Java predefined class Object is used to create the monitor in which both threads main and t2 participate. Note that the identity of object lock has to be conveyed to thread t2. In this case this is done via the constructor new Thread2(t1, lock).

Threads may interrupt their execution in monitors by calling the wait() method of the lock object of this monitor. During execution of the wait() method, the thread releases the lock and becomes inactive, thereby giving other threads an opportunity to acquire this lock. Such inactive threads may be awakened only by some other thread executing one of the notify() and notifyAll() methods of the lock object. The difference between these two methods is that notify() wakes up one arbitrary thread from all the potentially many waiting threads and notifyAll() wakes up all such threads. Similar to calls to wait(), calls to the notify() and notifyAll() methods must take place inside monitors for the corresponding locks. Both notification methods are non-blocking, which means that whether there are waiting threads or not, the notification call will return and the execution will continue.

In the rest of the paper we refer to start(), join(), wait(), notify(), and notifyAll() methods as thread *communication methods*[3].

Flow Graph Model for Java

Dynamic creation of threads is a well-known problem for static analysis. The number of instances of each thread class may be unbounded. For our analysis we make the usual assumption that there exists a known upper bound on the number of instances of each thread class. Alias resolution, including dealing with method (and thread object) polymorphism, is also an important issue. For the purposes of this paper we assume that alias resolution has been conservatively performed, using techniques such as [2, 9, 17].

The monitor-based model of communications between threads is significantly different from the communication mechanisms used by other popular concurrent languages, such as the rendezvous model of Ada 83 and CSP or the message sending model of Promela. The number of different thread communication methods in Java makes the problem of constructing the program model more difficult than the one for Ada. We solve this problem by representing only the control flow within individual threads and the interleavings of events in the TFG model of the program and use the feasibility constraint mechanism for modeling the semantics of thread interactions. Since some of the thread communication mechanisms, such as notification, require maintaining the state of many threads simultaneously, representing these mechanisms in the flow graph is cumbersome. Feasibility constraints are more readily suitable for capturing this functionality. In addition, since different ways in which threads affect each other's behavior use different thread methods, representing their functionality by separate FSAs is conceptually simpler than combining them all in one TFG[4]. One shortcoming of this approach is that, in practice, increasing the number and size of feasibility constraints frequently leads to increased time and space requirements of the FLAVERS analyses. We view the approach described here as a reasonable first step toward using FLAVERS for analysis of Java. In the future, we plan to evaluate the time and space requirements of modeling thread communications with feasibility constraints and to experiment with alternative approaches.

As with Ada, we first create a reduced, inlined control flow graph for each method in the program. Each call to a communication method is labeled with a tuple of the form (o, m, c), where o is the object owning method m, m is the method itself, and c is the calling thread. For example, for the code in Figure 5, the call t1.start in the main method will be represented with the label (t1, start, main). To make it easy to reason about groups of communications, we allow the wild-card symbol '$*$', which is used to indicate that one of the parts of the communication label can take any value. For example, (t, start, $*$) represents an event in which some thread in the program calls the start

---

[2]A related construct is a synchronized method, but the inlining performed in this approach results in code with synchronized statements.

[3]Additional thread methods stop(), suspend(), and resume() are defined in JDK 1.1 but have been deprecated in JDK 1.2 since they encourage unsafe software engineering practices. Because of this and space limitations we do not cover these methods in this paper. We discuss handling these methods in [14].

[4]Although not shown here, another advantage of this approach is that some of these communication constraints can be incorporated into certain automatically generated feasibility constraints.
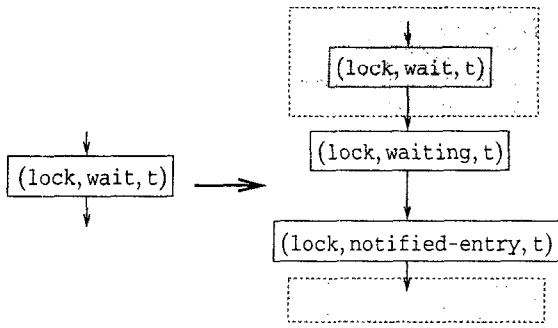
Figure 6: CFG transformation for wait() method calls

Figure 7: TFG example

method of thread t. The first node of a thread t is labeled (*, begin, t) and the last node of this thread is labeled (*, end, t). For consistency, we use this label format for arbitrary user-specified events as well. For example, the use of a variable var that occurs in thread t could be labeled (*, use_var, t).

For the purposes of our analysis, additional modeling is required for wait() method calls and synchronized blocks. Because an entrance to or exit from a synchronized block by one thread may influence executions of other threads, we represent the entrance and exit points of synchronized blocks with additional nodes labeled (lock, entry, t) and (lock, exit, t), where t is the thread modeled by the CFG and lock is the lock object of the synchronized block. We assume that the thread enters the synchronized block immediately after the entry node is executed and exits this block immediately after the exit node is executed. Thus, the entry node is outside the synchronized block and the exit node is inside this block.

The execution of a wait() method by a thread involves several activities. The thread releases the lock of the monitor containing this wait() call and then becomes inactive. After the thread receives a notification, it first has to re-acquire the lock of the monitor, before it can continue its execution. To be able to reason about all these activities of a thread, we perform a transformation that replaces each node representing a wait() method call with three different nodes, as illustrated in Figure 6. The node labeled (lock, wait, t) represents the execution of the wait() method, the node labeled (lock, waiting, t) represents the thread being idle while waiting for a notification, and the node labeled (lock, notified-entry, t) represents the thread after it received a notification and is in the process of obtaining the lock to re-enter the synchronized block. The shaded regions in the figure represent the synchronized block.

The CFGs for individual threads are combined into a TFG by using only the May Immediately Precede (MIP) edges, which, as in the approach of FLAVERS/Ada, represent all possible interleavings among pairs of nodes from different
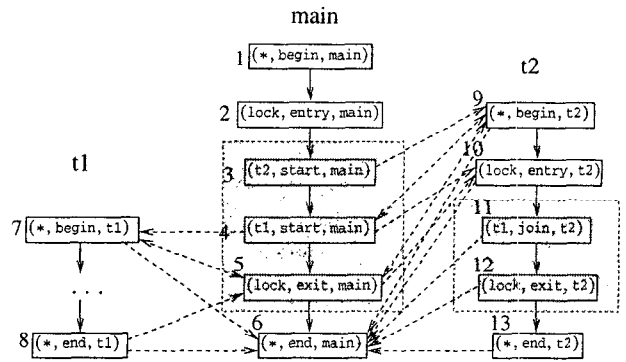
tasks. We have developed a conservative, precise, and cost-effective algorithm for generating these edges [15] that is similar to our algorithm for Ada [13]. Note that unlike the case for TFGs in FLAVERS/Ada, no additional nodes are created to represent communications among the program threads. Note also that even without representing thread communications explicitly in our Java graph model, this model conservatively overapproximates all possible executions of a program.

Figure 7 shows the TFG for the program in Figure 5. The shaded regions include nodes in the monitor of the program, solid edges represent control flow within individual threads and dashed edges are MIP edges. To simplify the figure, MIP edges between nodes from threads t1 and t2 are not shown.

Modeling Thread Communications with Feasibility Constraints

Although the TFG for a concurrent Java program represents a conservative overapproximation of all program behaviors, it does not model thread interactions. We model thread communications using feasibility constraints. Note that feasibility constraints modeling some types of thread interactions in the program may not be necessary for conclusive analysis of the property of interest. At present, we allow the user to choose the types of thread interactions in the program that are to be modeled by feasibility constraints. In future, we plan to provide guidance in the form of heuristics, suggesting construction of feasibility constraints for certain kinds of thread interactions depending on the program structure and the property being verified.

For each thread interaction mechanism present in JDK 1.2 we describe the corresponding feasibility constraint(s) and show the FSA(s). Transitions of these FSAs are defined in terms of TFG nodes. We use the label (o, m, c) to represent the set of all nodes marked with that label. We use set operations on labels to identify the set of nodes on which a transition is taken. $(*, *, *)$ stands for the set of all nodes in the graph. For example, the self-transition on state 0 in Figure 8, marked $(*, *, *) \setminus ((*, *, t) \cup (t, start, *))$
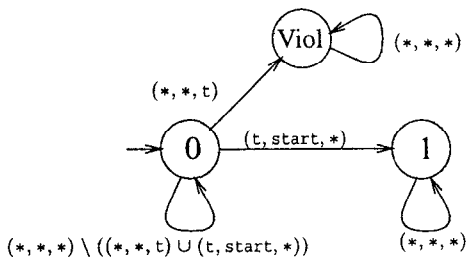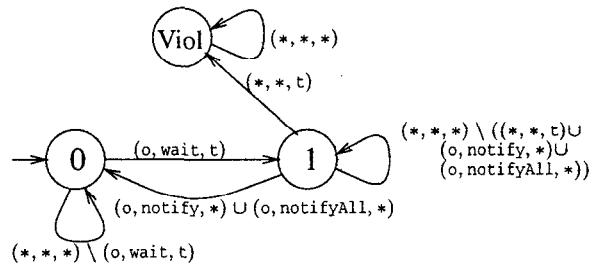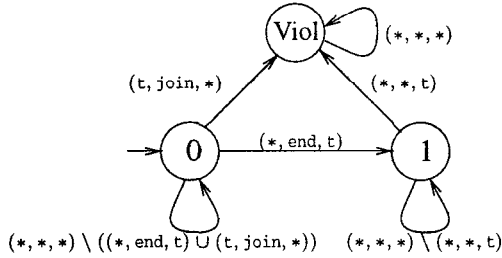
404

Figure 8: Constraint for `start`

Figure 9: Constraint for `join`

is taken upon traversal of any node that does not represent any activity performed by thread t or a call to the `start` method of thread t.

Start constraint
The start constraint enforces the requirement that a thread cannot execute until it is started by some other thread. This constraint can be constructed for each thread in the program, other than the main thread. The start constraint for a thread t is shown in Figure 8. State 0 models the situation before t is started. From this state, the transition to the violation state is taken if any node in thread t is traversed by the analysis. After a node representing a call to the method `start()` of t is traversed (this node has label (t, start, s), where thread s makes the call), the constraint makes the transition to state 1, after which no sequence of events can violate this constraint.

Using the start constraint makes it possible to model and analyze programs in which some threads may not be started at all. The CFG for each thread that may be created is constructed and included in the TFG, but the nodes of this thread's CFG will be traversed without violating this thread's start constraint only on those executions where this thread is actually started.

Join constraint
The join constraint enforces the requirement that after a thread terminates, no nodes from this thread can execute. In addition, it models the fact that a thread calling the `join()` method of another thread may proceed only after this latter thread terminates. Figure 9 shows this constraint. State 0 represents the situation where thread t has not ter-

Figure 10: Constraint for wait-notify constructs

minated. The transition to the violation state is taken from state 0 if a node representing a call to the `join()` method of thread t is traversed. Such a traversal represents an infeasible path because a call to `join()` cannot terminate until t is terminated. State 1 represents the situation after t is terminated. The transition from state 0 to state 1 is taken upon the traversal of the final node in thread t. If any node from thread t is traversed while this constraint is in state 1, the transition to the violation state is taken.

Wait-notify constraint
A wait-notify constraint models the fact that a thread can exit a state in which it is waiting for a notification only after such a notification comes from some other thread. This constraint has to be constructed for a specific thread and a specific monitor. Figure 10 shows this constraint for thread t and a monitor for object o. State 0 contains no transitions to the violation state and represents the state of the thread in which it is not waiting for a notification on object o. Once the node that represents thread t making a call to the `wait()` method of o is traversed, the constraint enters state 1. While the constraint is in this state, traversal of any node in thread t leads to the violation state, which represents the fact that, after a thread executes a `wait()` method and until it receives the corresponding notification, it stays idle. After a node corresponding to a call to either a `notify()` or a `notifyAll()` method of object o is traversed, the constraint goes back to state 0, signifying that the thread may be active now.

Because of the difference in semantics of `notify()` and `notifyAll()` methods, the state propagation has to be modified slightly to handle traversal of `notify` nodes. If there are multiple threads waiting for a notification on the same object, a `notify()` method call notifies only a single arbitrary thread. This thread may proceed, while other waiting threads must wait for another notification. Thus, if we have wait-notify constraints for multiple threads but the same lock, and a `notify` node for this lock is traversed with a state of the product automaton that represents k of these constraints being in state 1, k successor states are produced. Each successor state is characterized by exactly one wait-notify constraint changing to state 0. This change
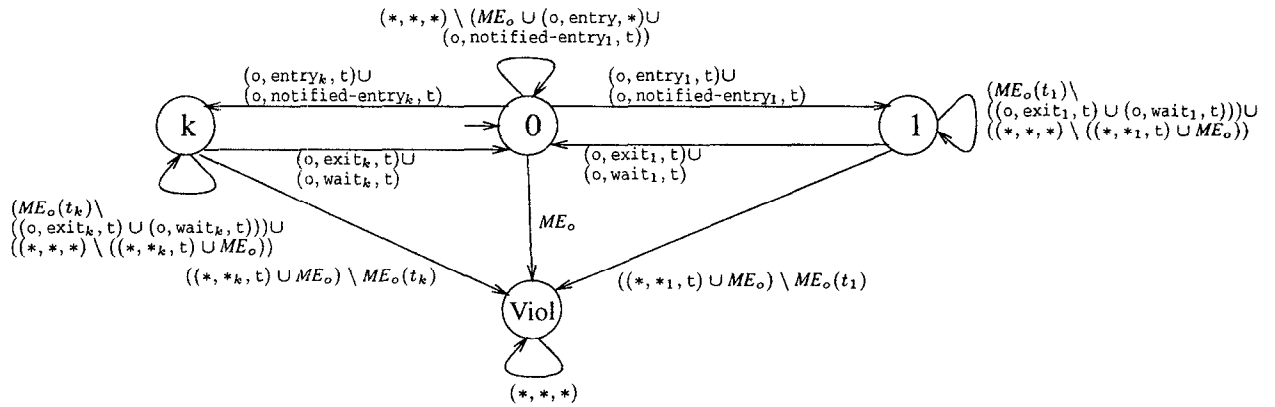
Figure 11: Monitor constraint

of the state propagation algorithm is quite straightforward and it does not introduce additional worst-case complexity. Because all threads waiting for a notification on a lock are notified by a call to the notifyAll method for this lock, traversal of a node corresponding to such a call results in a single product automaton state for each input product automaton state. In this output state, all wait-notify constraints for the corresponding lock are in state 0.

Monitor constraint

A single feasibility constraint can be created for each monitor in the program. If a program contains k threads, this constraint has $k + 2$ states: one violation state, one state that represents that no threads are executing in the monitor, and one state per thread to represent that this thread is executing in the monitor. We extend our label notation by introducing sets ME_o to represent all nodes inside the monitor for lock o and $ME_o(t)$ to represent all nodes of thread t inside the monitor for the lock o. If the threads are denoted $t_1, t_2, ..., t_k$, then $ME_o = \bigcup_{i=1}^{k} ME_o(t_i)$.

Figure 11 shows the general form of the monitor constraint, with only two states representing threads t_1 and t_k executing inside the monitor shown. State 0 represents the situation where no threads execute in this monitor. Thus, the transitions on any nodes located in this monitor will lead to the violation state. One of the threads, say t_k, may enter the monitor only after it acquired the lock, which is modeled by entry and notified-entry nodes. After one such node is executed, state k is entered. It corresponds to the situation where none of the other threads may execute inside of this monitor and thread t_k may not execute outside of this monitor. Traversals of these offending nodes will result in the constraint entering its violation state. State k may be exited only after traversing a node that represents thread t_k leaving this monitor. This happens when thread t_k either leaves the synchronized block in which it is currently executing or it executes the wait() method of object o, labeled (o, exit_k, t) and (o, wait_k, t) respectively.

Note that this constraint may be simplified in the context of a specific program. If a thread does not participate in the monitor modeled by the constraint, the state for it does not have to be created in the constraint. Similarly, if a thread, say t_i, enters the monitor but never executes the wait() method for the lock of this region, the transitions labeled (o, wait_i, t) and $(o, \text{notified-entry}_i, t)$ do not have to be included.

5   GENERAL CONCURRENCY FAULTS IN JAVA

General concurrency faults refer to situations that are considered harmful in concurrent programs, without regard to the specific application. Well-known examples are deadlocks and livelocks, when all or some of the threads in the program are stalled, and concurrent def-use faults [19]. Most of the other concurrency faults identified in the static analysis literature are application-specific. This low number of general concurrency faults is explained by the fact that most static analysis approaches deal with high-level rendezvous or message-sending concurrency models. Java provides a number of specialized, often low-level, thread communication mechanisms. One implication of this is that some combinations of these communications mechanisms may represent either erroneous or suspicious sequences of activities. Many of these sequences can be described as FSAs and detected using the approach described in this paper. Some erroneous or suspicious activities involve counts and thus cannot be represented with an FSA, but it is often possible to relax the specification to enable a representation in the FSA form.

In this section, we identify a number of general concurrency faults in Java programs. Due to space limitations, our discussion here is brief.

Premature join() Calls

A call to the join() method of a thread is *premature* if this thread has not been started at the time of the call. In Java such calls are simply ignored. The presence of program executions exhibiting such behavior is alarming because this may indicate a fault in the program logic. To detect
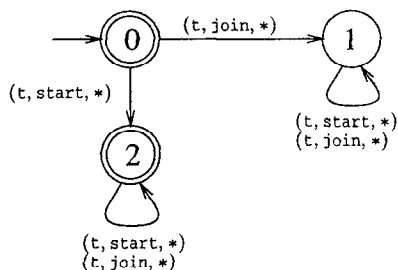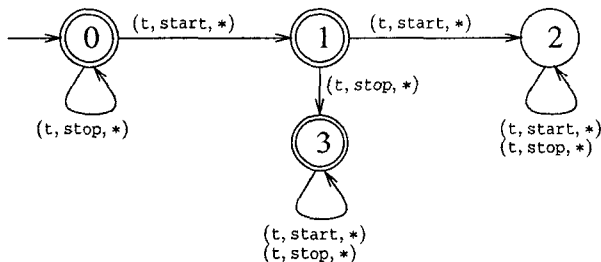
Figure 12: *Premature join() calls* property

Figure 13: Property that a thread cannot be started more than once without being stopped in between

such questionable sequences, we specify the property that the join() method of a thread may not be called before this thread is started. Figure 12 illustrates this property.

No Thread Restarted
A call to the start() method of a thread initiates execution of this thread. What happens if a thread is started twice? The answer depends on whether or not the thread is active when the start() method is called. If the thread is active, exception IllegalThreadState is thrown. If the thread has already completed its execution, the second start() call is simply ignored. Figure 13 shows a property that forbids restarting a thread while it is still active. While we believe that in most cases the possibility of two or more calls to the start() method of a thread represents a seriously erroneous situation, the exception handling mechanism of Java lets programmers catch the IllegalThreadState exception, recovering from the error. While no exceptions are thrown and the program is not interrupted in the second case, it may indicate suspicious logic, where a thread is assumed to be alive while in fact it is stopped.

Waiting Forever
One specific case of livelock that is a suspicious use of Java concurrency mechanisms is when a thread becomes inactive and never becomes active again. This happens when the thread executes the wait() method for the lock object of a monitor, but is never notified and thus never resumes its execution. The property stating that this must not happen cannot be specified as an FSA because counting is re-
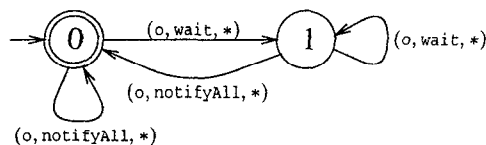

Figure 14: The property that no thread can wait forever

quired. Since the notify() method only notifies one arbitrary waiting thread, to represent this kind of livelock the number of threads having executed the wait() method of a lock object must be matched with the number of calls to the notify() method of the same lock object. Note that a specialized data flow analysis algorithm can be defined for this case, since the number of threads that can wait for a notification at the same time is bounded by the total number of threads in the program. Because of space limitations, we do not describe this approach here.

The case where only notifyAll() methods are used can be represented in the FSA form. The property that can be checked for such programs is shown in Figure 14. Note that our current approach is not capable of handling this property. Since terminating executions are defined as those where all threads terminate, the executions that violate this property will be ignored since they involve at least one thread waiting forever. At present we are working on extending the approach to handle executions that may not terminate.

No Unnecessary Notifications
Notifications issued when no threads are waiting are wasteful. In addition, they also may indicate suspicious logic (e.g. where the programmer assumes erroneously that some threads may be waiting). FLAVERS can be used to determine if certain calls to the notify() and notifyAll() methods are not necessary on some executions. Similar to the property of threads waiting forever, this property cannot be specified in general because handling calls to the notify() method involves counting. A weaker property can be checked that relies on notifyAll() methods to determine if there are any threads waiting. This property is shown in Figure 15.

Dead Interactions
We call a thread interaction, such as a call to a communication method of another thread, *dead* if by the time this interaction takes place, the target thread has already terminated. According to the Java semantics, such calls are simply ignored. While in many cases dead interactions are not harmful, in other cases they could indicate a fault in the program logic or unoptimal code. Although the general description of this fault is program-independent, it has to be checked for specific thread interaction methods. Figure 16 shows a property of dead joins, where label S represents a specific t.join() method call.
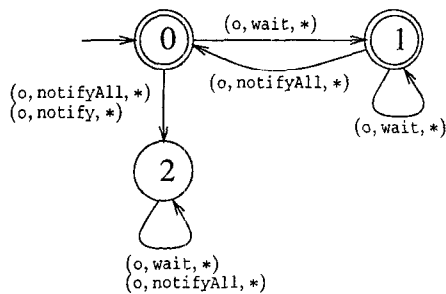
407

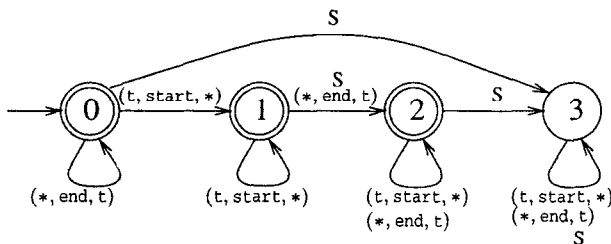Figure 15: Property that no two successive notifyAll calls on the same object can be made successively

Figure 16: Dead join property

6 PRELIMINARY EXPERIMENTAL RESULTS

Before committing to a careful implementation of FLAVERS/Java, we produced a prototype to test the feasibility of this approach. Here we present the initial data from this study, using three small concurrent Java programs. The FLAVERS/Java prototype is implemented in Java. The current version supports all control constructs other than exceptions but does not support full semantic analysis and inlining. While these shortcomings would impede analysis of large real programs, it was easy to modify our small test examples to enable the tool to parse them. The results of this study are encouraging, indicating the ability of the approach to check properties of these programs conclusively in a reasonably small amount of time comparable to that taken by FLAVERS/Ada to analyze similar examples.

Empirical Results

In this study, we consider three examples: dining philosophers, readers-writers, and gas station. The dining philosophers example is taken from [4] and has three philosophers. The readers-writers example is based on the producer-consumer example from [2] and has two readers and two writers. This example was implemented by a person not familiar with FLAVERS. The gas station example is the race-free version of [7]. It was implemented by one of the authors. This example contains two customers, one cashier, and one pump.

In each of these three examples the threads of control are synchronized primarily by monitors. In fact, no calls to

the join() methods of threads were used in any of the examples. Thus, we checked only the general concurrency properties *No thread restarted* and *No unnecessary notifications* for each example. Property *No thread restarted* was successfully verified for all examples. All examples were found to be in violation of the *No unnecessary notifications* property and there really exist feasible executions of the programs that exhibit violation of this property. No feasibility constraints were required for checking for these two general concurrency faults.

In addition, for each of the examples we checked two application-specific properties. FLAVERS/Java proved all these properties conclusively. Due to the lack of space, we do not describe each of these properties and the details of checking them with FLAVERS/Java, presenting only the summary of the results in Figure 17. For each example, we indicate the number of lines of code and the number of nodes in the corresponding TFG. The second and third columns give the number of concurrency constraints[5], as described in Section 4, and other feasibility constraints required for the analysis, respectively. In all cases these additional feasibility constraints are of two types, constraints that model control flow in a single thread, similar to the constraint in Figure 4 for the Ada example in Figure 1, and constraints that model behaviors of select program variables. The execution time in seconds is shown in the last column and includes the combined time it took to parse Java code, construct all necessary artifacts, and run the analysis. For our experiments, we used a Symantec JIT compiler for JDK 1.1 on a workstation equipped with a 266 MHz Pentium II processor and 64Mb of memory, running Windows NT.

Discussion

One interesting observation is that the only concurrency constraint required for checking properties of these three examples was the *monitor* constraint. On one hand, this is reasonable since all threads in all examples are started in a straightforward manner by the main thread, with no events of interest happening in the main thread and no calls to the join() method. On the other hand, the wait-notify mechanism is used extensively in all examples and yet no *wait-notify* constraints were required. The fact that only *monitor* constraints were required gives us hope that in general, for well-structured Java programs, the proposed approach of modeling Java concurrency with feasibility constraints will not add too much overhead to the analyses. Further experimentation will be needed to test this hypothesis.

Another observation is that whenever checking a prop-

---

[5]In all cases at first we attempted to analyze each property without using any concurrency constraints and then were adding these constraints one by one, until either a conclusive result was obtained or we were able to find a path that violated the property and corresponded to a feasible execution of the program.

	conc. const.	feas. const.	time, sec
Dining philosophers, 65 loc, 98 nodes			
no thread restarted	0	0	2.24
no unnecessary notifications	0	0	2.24
no lifting left forks together	3	6	28.95
no eating together	3	6	28.64
Readers-writers, 62 loc, 73 nodes			
no thread restarted	0	0	1.86
no unnecessary notifications	0	0	1.83
only one write at a time	1	1	2.22
no read and write together	1	1	2.22
Gas station, 63 loc, 68 nodes			
no thread restarted	0	0	2.24
no unnecessary notifications	0	0	2.16
no race	2	8	20.06
no pumping without payment	0	1	2.47

Figure 17: Analysis results for the examples

erty required at least one *monitor* constraint, *monitor* constraints had to be constructed for all lock objects in the example. This probably reflects the fact that the concept of monitors is central in Java concurrency and so each monitor used in a program imposes important restrictions on the control flow.

In the process of finding out experimentally which concurrency and feasibility constraints are necessary for conclusively proving the properties of our examples, we discovered that in most cases adding *monitor* constraints actually improved the analysis time of the tool. The reason for this is that using a *monitor* constraint significantly reduces the number of control paths explored by FLAVERS/Java.

Checking for general concurrency faults from Section 5 proved to be very straightforward, not requiring any feasibility constraints. We believe that in the case of the *No thread restarted* property this is just the result of the straightforward way of starting threads in our examples. In the case of the *No unnecessary notifications* property, we observed that all examples contain executions on which a call to the `notifyAll()` method of a lock object is executed before any calls to the `wait()` method of this object.

The timing data for this analysis are quite encouraging, given the immaturity of the analysis tool. Even with the large number of feasibility constraints needed for checking some of the properties of the dining philosophers and gas station examples, the analysis time never exceeded 30 seconds. These times are comparable to those taken by FLAVERS/Ada, a more mature tool, to check Ada versions of these programs.

7 CONCLUSION

We have presented an adaptation of the FLAVERS ap-

proach for analyzing application-specific properties of concurrent Java programs. With this approach, the semantics of each of the Java communication constructs are modeled with feasibility constraints. We view this approach as an initial proposal. In fact there is a spectrum of alternative approaches, from modeling all intertask communications as feasibility constraints, as we advocate here, to modeling all communications directly in the flow graph representation of the program. The approach described here seems to us to be a good starting point, but extensive empirical evaluation will be needed to determine the most efficient representation. We intend to undertake such studies in the future.

The proposed technique has the worst-case complexity of $O(SN^2)$, where N is the number of events of interest in the program and S is the size of the product of all finite state automata used in the analysis. Our experience with Ada programs indicates that in practice the number and size of these finite state automata are not very large. In addition, usually the combined state space of these automata is only a fraction of their full cross product. It remains to be seen if this is true for Java programs in general.

We have produced an initial implementation of the FLAVERS/Java tool and undertaken an initial case study in which we analyzed a number of properties of three small concurrent Java programs. Our prototype was able of analyzing these programs in a reasonable amount of time. These preliminary results give us hope that in general only a small fraction of all possible concurrency constraints for a program is actually needed for checking properties of this program conclusively. More extensive experimentation will be required to support this hypothesis. However, before committing to such an in-depth case study of the applicability of this approach to medium- and large-size Java programs, we will experiment with alternative modeling approaches. Subsequently, we plan on using a large set of real-world Java programs for comparing the feasibility of these modeling approaches and identifying the most promising one.

REFERENCES

[1] S. C. Cheung and J. Kramer. Tractable dataflow analysis for distributed systems. *IEEE Transactions on Software Engineering*, 20(8):579–593, Aug. 1994.

[2] J. C. Corbett. Constructing compact models of concurrent Java programs. In *ACM SIGSOFT Proceedings of the 1998 International Symposium on Software Testing and Analysis*, pages 1–10, 1998.

[3] J. C. Corbett and G. S. Avrunin. Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design*, 6:97–123, January 1995.

[4] C. Demartini and R. Sisto. Static analysis of Java multithreaded and distributed applications. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 215–222, Apr. 1998.

[5] M. Dwyer and L. Clarke. Data flow analysis for verifying properties of concurrent programs. In *ACM SIGSOFT'94, Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 62–75, December 1994.

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

[7] D. Helmbold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985.

[8] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall Software Series, 1991.

[9] W. Landi and B. Ryder. Pointer-induced aliasing: A problem taxonomy. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages (POPL '91)*, pages 93–103, Orlando, FL, USA, Jan. 1991. ACM Press.

[10] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28(2):121–163, 1990.

[11] S. P. Masticola and B. G. Ryder. Static infinite wait anomaly detection in polynomial time. In D. A. Padua, editor, *Proceedings of the 1990 International Conference on Parallel Processing. Volume 2: Software*, pages 78–87, Urbana-Champaign, IL, Aug. 1990. Pennsylvania State University Press.

[12] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.

[13] G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of the Sixth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 24–34, Nov. 1998.

[14] 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, Amherst, Apr. 1998.

[15] G. Naumovich, G. S. Avrunin, and L. A. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. Technical Report 98-44, University of Massachusetts, Amherst, Oct. 1998. http://laser.cs.umass.edu/abstracts/98-044.html.

[16] G. Naumovich, L. A. Clarke, and L. J. Osterweil. Comparing implementation strategies for composite data flow analysis problems. In *Proceedings of SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 51–58, June 1998.

[17] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *ACM SIGPLAN Proceedings of the 1994 Conference on Object-Oriented Programming*, pages 324–340, 1994.

[18] J. H. Reif and S. A. Smolka. Data flow analysis of distributed communicating processes. *International Journal of Parallel Programming*, 19(1):1–30, Feb. 1990.

[19] R. N. Taylor and L. J. Osterweil. Anomaly detection in concurrent software by static data flow analysis. *IEEE Transactions on Software Engineering*, 6(3):265–278, May 1980.