# Data Flow Analysis for Checking Properties of Concurrent Java Programs*

Gleb Naumovich, George S. Avrunin, and Lori A. Clarke
Laboratory for Advanced Software Engineering Research
Department of Computer Science
University of Massachusetts at Amherst
Amherst, MA 01003-6410
{naumovic, avrunin, clarke}@cs.umass.edu

## Abstract

In this paper we show how the FLAVERS data flow analysis technique, originally formulated for programs with the rendezvous model of concurrency, can be applied to concurrent Java programs. The general approach of FLAVERS is based on modeling a concurrent program as a flow graph and using a data flow analysis algorithm over this graph to check statically if a property holds on all executions of the program. The accuracy of this analysis can be improved by supplying additional information, represented as finite state automata, to the data flow analysis algorithm.

In this paper we present a straightforward approach for modeling Java programs that uses the accuracy improving mechanism to represent 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.

## 1 Introduction

With the advent of Web technology, distributed programming, and the Java programming language in particular, are growing 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 consider when task interleaving is considered, but executing 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 evaluate all potentially executable paths for specific kinds of faults. 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 [4]. 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 tool 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 results.

The emphasis of this paper is on modeling Java programs in a way that can be used by FLAVERS. We describe one promising approach in which the semantics of thread communications are represented with feasibility constraints, instead of being a part of the flow graph program model. 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.

The next section gives a brief description of related work.

*This research was partially supported by the Air Force Research Laboratory/IFTD and the Defense Advanced Research Projects Agency under Contract F30602-97-2-0032 and by the National Science Foundation under Grant CCR-9708184. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation, the Defense Advanced Research Projects Agency, the Air Force Research Laboratory/IFTD, or the U.S. Government.

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 introduces the modified program model for Java. Section 5 describes some suspicious patterns of thread communications and gives an example of using the modified approach to prove the absence of such patterns. Finally, we present a summary and describe directions of our future work.

## 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 flow graph program model. In this paper, however, we have not focused on the derivation of the program model.

Demartini and Sisto [3] describe two models of Java programs. The first method models Java programs with Petri nets and the second models 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 they are still prohibitively expensive to use with large software systems.

As an alternative to reachability approaches, data flow analyses for concurrent software have been formulated with low-order polynomial execution time and storage bounds. Most of these have been defined to check application-independent properties (e.g. [1, 8, 12]). FLAVERS is one of the few data flow techniques capable of directly supporting checking application-specific properties of concurrent software.

## 3 FLAVERS for Ada

In this section we give a brief general description of FLAVERS/Ada and then, in the next section, describe how this approach can be modified to deal with Java or other languages that use a similar concurrency model.

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. These nodes are labeled with *events*, user-selected names to be associated with such activities. To reduce the size of the representation, the CFGs are refined to remove all nodes that are not labeled with an event. 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 represent the start and the end states of the program respectively. 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 [9].

The set of all events associated with a model of the program is the *alphabet* of the TFG. 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 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. 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 language of the TFG is contained in the language of this property. Data flow analysis is used to solve this containment problem.

If the analysis finds that a property holds, then it is guaranteed to be valid on all possible program executions. When the results of 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 real program executions. 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 the 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 a variable and thereby enforce their semantics.

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:

- The product automaton accepts a sequence of events only if this sequence is accepted by the property automaton;
- 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 [10].

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. Once the solution of this data flow problem converges to a *join over all paths* solution [7], 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].

# 4 Analysis of Java Concurrent Programs

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.

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

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

class Thread2 extends Thread
{ public Thread2...
  public void run() {
    synchronized (o) {
      t1.join();
      ...
    }
  }
}
```

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

Figure 1: Java code example

## 4.1 Java Model of Concurrency

In Java, concurrency is modeled with *threads*. Threads are objects of classes that specify thread types. 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 1 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 is never called explicitly. 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 1 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. To execute a `synchronized` statement, a thread must acquire the lock of the object indi-

cated 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 1, an object `o` 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 `o` has to be conveyed to thread `t`. In this case this is done via the constructor `new MyThread(t1,o)`.

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.

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. Method `stop()` may be used to stop threads. When this method is called, the target thread must stop whatever it is doing and terminate. A pair of thread methods `suspend()` and `resume()` provide a means for suspending and resuming the execution of the target thread. A thread whose `suspend()` method is called halts its execution (without termination) but can continue from the same point after its `resume()` method is called. Note that a call to the `stop()` method of a thread releases all locks owned by this thread, if any. The `suspend()` method does not release the locks that the thread owns, which could lead to undesirable situations where a thread is suspended while in a monitor, thereby preventing other threads from entering this monitor. This is one of the reasons why the suspend-resume mechanism has been deprecated in JDK 1.2.

In the rest of the paper we refer to `start()`, `join()`, `wait()`, `notify()`, `notifyAll()`, `stop()`, `suspend()`, and `resume()` methods as thread *communication methods*.

## 4.2 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, 5, 11].

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 using 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[2]. 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 impact of modeling thread communications with feasibility constraints on the running time and space requirements of the analysis 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 $(\mathtt{m}, \mathtt{c}, \mathtt{o})$, where `m` is the method, `c` is the caller, and `o` is the object owning method `m`. For example, for the code in Figure 1, the call `t1.start` in the main method will be represented with the label $(\mathtt{start}, \mathtt{main}, \mathtt{t1})$. To make it easy to reason about groups of communications, we allow the wildcard symbol '$*$', which is used to indicate that one of the parts of the communication label can take any value. For example, $(\mathtt{start}, *, \mathtt{t})$ represents an event in which some thread in the program calls the `start` method of thread `t`. The first node of a thread `t` is labeled $(\mathtt{begin}, \mathtt{t}, *)$ and the last node of this thread is labeled $(\mathtt{end}, \mathtt{t}, *)$. For consistency, we use this event format for arbitrary user-specified events as well. For example, the use of a variable `var` that occurs in thread `t` could be labeled $(\mathtt{use\_var}, \mathtt{t}, *)$.

[2] Also, some of the constraints modeling thread interactions can be incorporated into the *thread automata* feasibility constraints (called task automata in [4]).

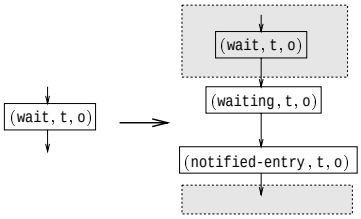


Figure 2: CFG transformation for `wait()` method calls

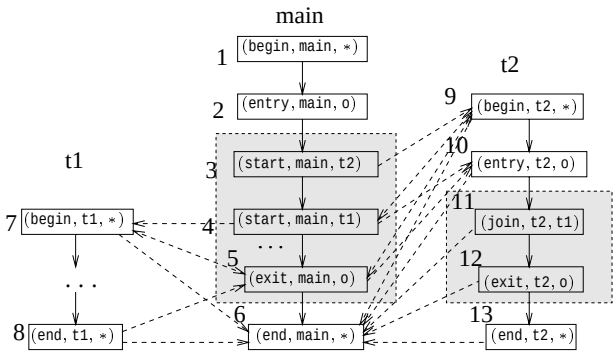


Figure 3: TFG example

For the purposes of our analysis, additional modeling is required for `wait()` method calls and `synchronized` blocks. Because a thread entering or leaving a `synchronized` block may influence executions of other threads, we represent the entrance and exit points of `synchronized` blocks with additional nodes labeled $(\texttt{entry}, \texttt{t}, \texttt{o})$ and $(\texttt{exit}, \texttt{t}, \texttt{o})$, where `t` is the thread modeled by the CFG and `o` 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 2. The node labeled $(\texttt{wait}, \texttt{t}, \texttt{o})$ represents the execution of the `wait()` method, the node labeled $(\texttt{waiting}, \texttt{t}, \texttt{o})$ represents the thread being idle while waiting for a notification, and the node labeled $(\texttt{notified-entry}, \texttt{t}, \texttt{o})$ 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 MIP edges, without communication nodes used by FLAVERS/Ada. We have developed a conservative algorithm for computing this information that is similar to our algorithm for Ada [9]. Note that even without representing thread communications explicitly in our Java graph model, this model conservatively overapproximates all possible executions of a program.

Figure 3 shows the TFG for the program in Figure 1. 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.

## 4.3 Modeling Thread Communications with Feasibility Constraints

Although the TFG for a Java concurrent program represents a conservative overapproximation of all program behaviors, it does not model thread interactions. We model thread communications using feasibility constraints.

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 $(\texttt{m}, \texttt{c}, \texttt{o})$ 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 4, marked $(*, *, *) \setminus ((*, \texttt{t}, *) \cup (\texttt{start}, *, \texttt{t}))$ 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`. We discuss the deprecated methods `stop()`, `suspend()`, and `resume()` in Section 4.4.

### 4.3.1 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 4. 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 $(\texttt{start}, \texttt{s}, \texttt{t})$, 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

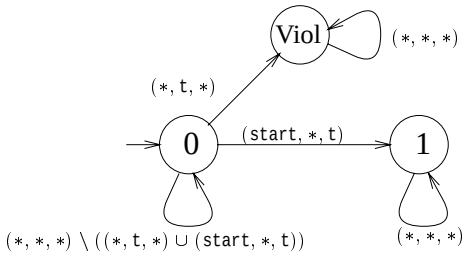


Figure 4: Constraint for `start`

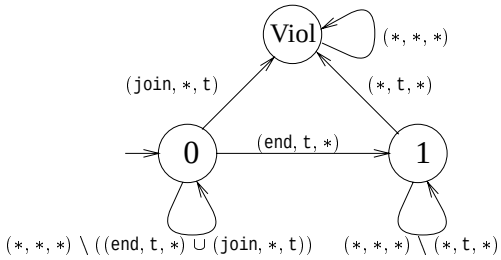


Figure 5: Constraint for `join`

start constraint only on those executions where this thread is actually started.

#### 4.3.2 Stop-join constraint

The stop-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 target thread terminates. Figure 5 shows this constraint. State 0 represents the situation where thread `t` is not terminated. 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.

#### 4.3.3 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 6 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

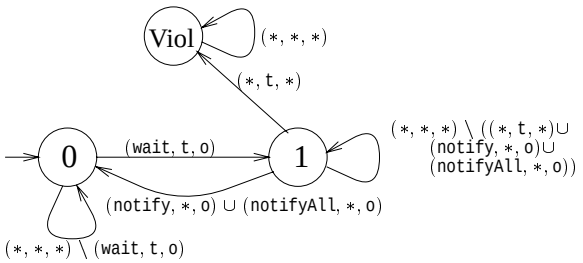


Figure 6: Constraint for wait-notify constructs

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 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.

#### 4.3.4 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

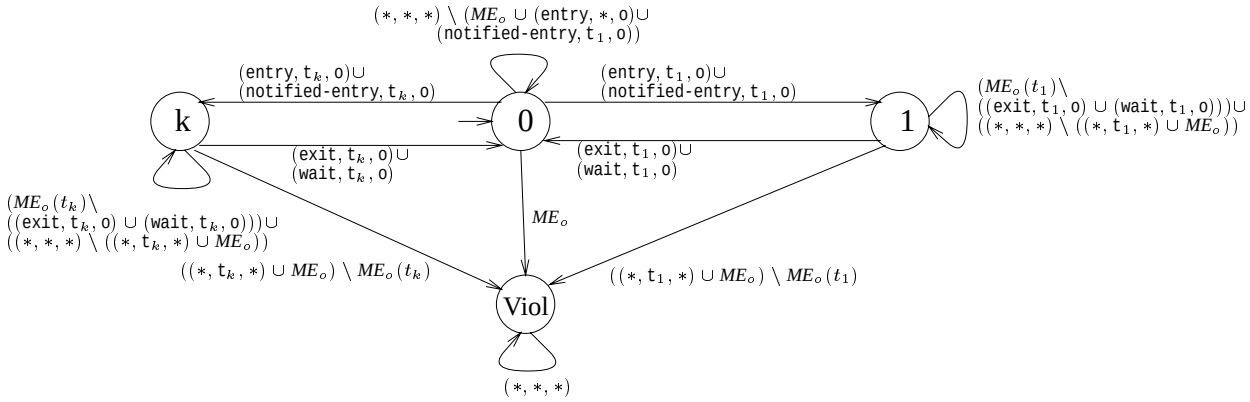


Figure 7: Monitor constraint

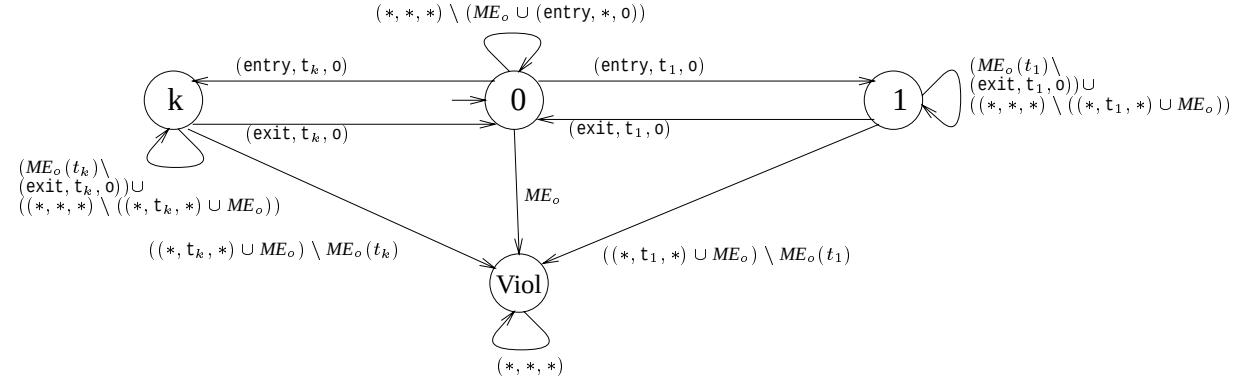


Figure 8: Monitor constraint for the example

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 7 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 of such nodes 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 $(\texttt{exit}, \texttt{t}_k, \texttt{o})$ and $(\texttt{wait}, \texttt{t}_k, \texttt{o})$ 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 $(\texttt{wait}, \texttt{t}_i, \texttt{o})$ and $(\texttt{notified-entry}, \texttt{t}_i, \texttt{o})$ do not have to be included. Figure 8 gives an example of the monitor constraint for the program in Figure 1. Note that since thread `t1` never acquires the lock of `o`, no state representing this thread executing inside the monitor of this program is present. Also, this constraint does not have transitions on `wait` and `notified-entry` nodes because there are no calls to these methods in this program.

## 4.4 Modeling the Deprecated Thread Methods

This section shows a feasibility constraint used for modeling the way that calls to the `suspend()` and `resume()` methods of a thread affect the behavior of this thread. In addition, we indicate the changes that have to be made to two of the fea-

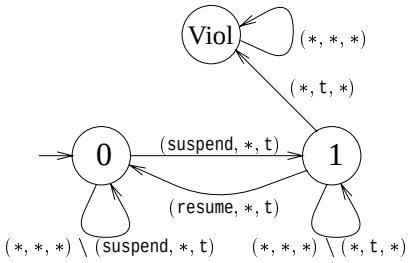


Figure 9: Constraint for suspend-resume constructs

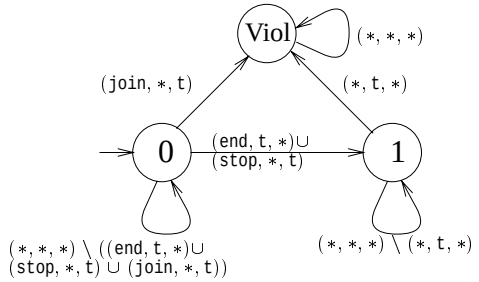


Figure 10: Constraint for `join` in the presence of calls to `stop()`

sibility constraints introduced in the last section if the deprecated `stop()` method is used.

#### 4.4.1 Suspend-resume constraint

A suspend-resume constraint models behavior of a thread that may be suspended by calls to its `suspend()` method from other threads. Figure 9 illustrates this constraint for thread `t`. State 0 represents the "active" state of `t`. No transitions to the violation state may occur from this state. When a node representing a call to the `suspend()` method of `t` is traversed, the constraint takes the transition to state 1. Since this state represents the state of the system where `t` is suspended, traversal of any node in `t` will lead to the violation state. The transition back to state 0 is taken from state 1 only after a node representing a call to the `resume()` method of `t` is traversed.

#### 4.4.2 Changes to stop-join and monitor constraints

The use of the `stop()` thread method requires changing two of the feasibility constraints introduced in Section 4.3: stop-join and monitor constraints. None of the other constraints are affected by the use of the deprecated methods.

The change to the stop-join constraint models the fact that a thread may become inactive after its `stop()` method is called. Figure 10 shows the FSA for the modified constraint. The transition from state 0, which represents the active state of thread `t`, to state 1, which represents the stopped state of this thread, is taken upon the traversal of a node that represents some thread invoking the `stop()` method of `t`.

The change to the monitor constraint reflects the fact that if the `stop()` method of a thread is called while it possesses one or more locks (i.e. this thread is executing in one or more monitors), this thread has to release all these locks before stopping. Figure 11 shows the FSA for the resulting constraint. The only difference from the original monitor constraint from Figure 7 is that the transition from state $i$, which represents thread $t_i$ executing in the monitor, to state 0, which represents the situation where no threads are executing in the monitor, is taken upon the traversal of a node that models a call to the `stop()` method of $t_i$.

# 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 [13]. Most of 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. We illustrate a number of such sequences and, where possible, show their FSA representations.

A number of erroneous or suspicious activities involve counting and thus cannot be represented with an FSA. Usually, in these cases it is possible to relax the specification to enable a representation in the FSA form, as we illustrate.

## 5.1 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 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.

We use this property to illustrate the way the concurrency constraints remove infeasible paths from consideration by the analysis. The property is checked for thread `t1` in Figure 1. The property automaton shown in Figure 12 has to be instantiated for thread `t1`, which involves replacing the `t` in

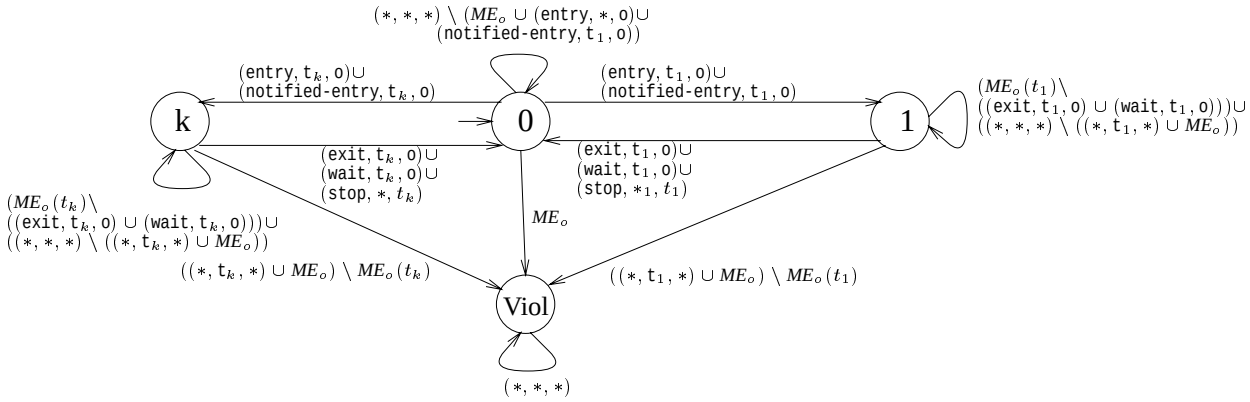


Figure 11: Monitor constraint in the presence of calls to `stop`

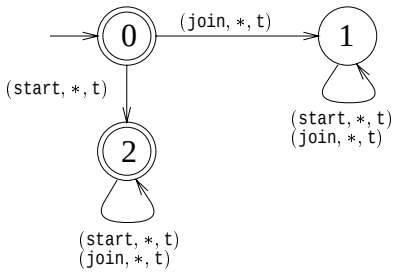


Figure 12: Property that no communication methods of a thread may be called before it is started

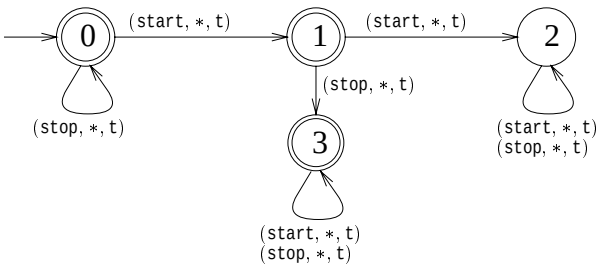


Figure 13: Property that a thread cannot be started more than once without being stopped in between

every label with a `t1`. If no constraints are used in the analysis, the approach finds, for example, the path 1, 2, 3, 9, 10, 11 through the graph that starts in the begin node of the main thread, continues through the main thread until the node representing the call to the `start` method of thread `t2`, takes the MIP edge to the begin node of thread `t2`, and continues in this thread until the node representing the call to the `join` method of thread `t2`. Thus, on this path thread `t2` calls the `join()` method of thread `t1` before `t1` is started by the main thread. However, this path is infeasible. This can be detected by using the monitor constraint in Figure 8. When the path described above is traversed, this constraint enters state 1 upon the traversal of node 3, which is the point where the main thread enters the monitor. When node 10 is traversed, the constraints takes a transition to the violation state, which signifies that the analysis must regard this path as infeasible. The monitor constraint and the thread automaton constraint, which models the control flow through the main thread, are sufficient to prove that this property holds for the example.

## 5.2 Starting a thread more than once

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. Due to this distinction, we describe two versions of this property. Figure 13 shows the first case, with the second `start()` call happening while thread `t` 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.

Figure 14 shows the second case, where the second `start()` call happens after thread `t` stopped. While no exceptions are thrown and the program is not interrupted in this case, it may indicate suspicious logic, where a thread is assumed to be alive while in fact it is stopped.

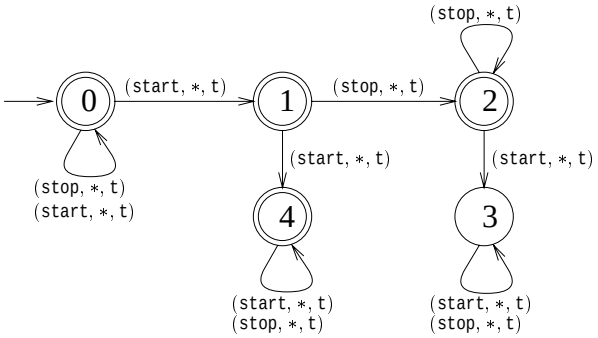


Figure 14: Property that a thread cannot be started after it was stopped

## 5.3 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 required. 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 15. It requires that each call to the `wait()` method of a lock object is eventually followed with a call to the `notifyAll()` method of this object. Note that the 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.

## 5.4 No Unnecessary Notifications

Execution of notification methods, especially `notifyAll()`, is expensive [6]. Thus, 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

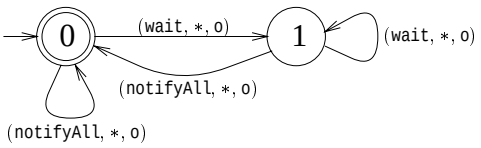


Figure 15: The property that no thread can wait forever

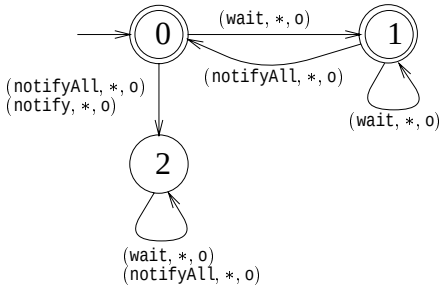


Figure 16: Property that no two successive `notifyAll` calls on the same object can be made successively

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, shown in Figure 16, requires that there are no calls to the `notify()` or `notifyAll()` methods preceded by a call to `notifyAll()` without a call to the `wait()` method in between.

## 5.5 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 breach 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 17 shows a property of dead `join`s, where label $S$ represents a specific `t.join()` method call. This property has to be run for all `t.join()` method calls in the program.

## 5.6 Suspending with Locks

Java includes two mechanisms that let threads avoid interfering with each other: monitors and suspend-resume interac-

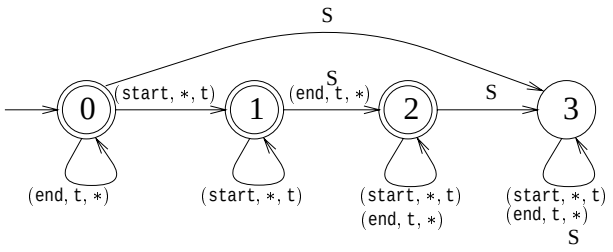


Figure 17: Dead join property

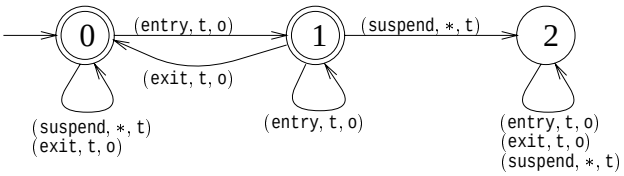


Figure 18: Property that a thread cannot be suspended while in possession of locks

tions. In general it is not a good idea to mix them, for the following reason. If a thread $t$ that is executing in a monitor is suspended, it becomes inactive but does not release the lock for this monitor. Thus, for the whole period of inactivity of thread $t$ no threads may enter this monitor. While in some cases this may be what is desired, in general having the thread execute the `wait()` statement on the object whose lock is used in this monitor is a much better choice, since in this case the thread releases the lock before becoming inactive. Figure 18 shows the property that specifies that on no executions should it be possible that a thread is suspended while in the monitor.

# 6 Conclusion

We have presented an adaptation of the FLAVERS approach 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 the finite state automata used in FLAVERS analyses 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.

We have produced an initial implementation of the FLAVERS/Java system and undertaken some small experiments, in which the program models were constructed by hand. We are in the process of automating the derivation of our program models directly from the Java code. After such tools are built, we plan on experimenting with alternatives modeling approaches and then carrying out an experimental evaluation of the tool's applicability to real-world Java programs.

# References

[1] Shing Chi Cheung and Jeff Kramer. Tractable dataflow analysis for distributed systems. *IEEE Transactions on Software Engineering*, 20(8):579–593, August 1994.

[2] James 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] Claudio Demartini and Riccardo Sisto. Static analysis of Java multithreaded and distributed applications. To appear in Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems, April 1998.

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

[5] William Landi and Barbara 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, January 1991. ACM Press.

[6] Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, Reading, MA, 1997.

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

[8] Stephen P. Masticola and Barbara G. Ryder. Static infinite wait anomaly detection in polynomial time. In

David A. Padua, editor, *Proceedings of the 1990 International Conference on Parallel Processing. Volume 2: Software*, pages 78–87, Urbana-Champaign, IL, August 1990. Pennsylvania State University Press.

[9] Gleb Naumovich and George S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. Technical Report 98-23, University of Massachusetts, Amherst, April 1998.

[10] Gleb Naumovich, Lori A. Clarke, and Leon J. Osterweil. Comparing implementation strategies for composite data flow analysis problems. To appear in proceedings of SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, June 1998.

[11] John Plevyak and Andrew 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.

[12] John H. Reif and Scott A. Smolka. Data flow analysis of distributed communicating processes. *International Journal of Parallel Programming*, 19(1):1–30, February 1990.

[13] Richard 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.