1999

An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs

Gleb Naumovich
University of Massachusetts Amherst

An Efficient Algorithm for Computing *MHP* Information for 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. Information about which statements in a concurrent program may happen in parallel (*MHP*) has a number of important applications. It can be used in program optimization, debugging, program understanding tools, improving the accuracy of data flow approaches, and detecting synchronization anomalies, such as data races. In this paper we propose a data flow algorithm for computing a conservative estimate of the *MHP* information for Java programs that has a worst-case time bound that is cubic in the size of the program. We present a preliminary experimental comparison between our algorithm and a reachability analysis algorithm that determines the ”ideal” static *MHP* information for concurrent Java programs. This initial experiment indicates that our data flow algorithm precisely computed the ideal *MHP* information in the vast majority of cases we examined. In the two out of 29 cases where the *MHP* algorithm turned out to be less than ideally precise, the number of spurious pairs was small compared to the total number of ideal *MHP* pairs.

1 Introduction

Information about which statements in a concurrent program may happen in parallel (*MHP*) has a number of important applications. It can be used for detecting synchronization anomalies, such as data races [6], for improving the accuracy of various data flow analysis and verification approaches (e.g. [9, 14, 19]), for improving program understanding tools, such as debuggers, and for detecting program optimizations. For ex-

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

Appears in Proceedings of the Seventh European Software Engineering Conference and Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 338 – 354, September 1999.

ample, in optimization, if it is known that two threads of control will never attempt to enter a critical region of code at the same time, any unnecessary locking operations can be removed.

In general, the problem of precisely computing all pairs of statements that may execute in parallel is undecidable. If we assume that all control paths in all threads of control are executable, then the problem is *NP*-complete [21]. In this paper, we call the solution with this assumption the *ideal MHP* information for a program. In practice, a trade-off must be made where, instead of the ideal information, a *conservative estimate* of all *MHP* pairs is computed. In this context a conservative estimate contains all the pairs that can actually execute in parallel but may also contain *spurious* pairs. The precision of such approaches can be measured by comparing the set of pairs computed by an approach with the ideal set, if the latter is known.

In this paper we propose a data flow algorithm for computing a conservative estimate of *MHP* information for Java programs that has a worst-case time bound that is cubic in the size of the program. In the rest of this paper we refer to this algorithm as the *MHP algorithm*. To evaluate the practical precision of our algorithm, we have carried out a preliminary experimental comparison between our algorithm and a reachability-based algorithm that determines the ideal *MHP* information for concurrent Java programs. Of course, since this reachability algorithm can only be realistically applied to small programs, our experiment was restricted to programs with a small number of statements. This initial experiment indicates that our algorithm precisely computed the ideal *MHP* information in the vast majority of cases we examined. In the two out of 29 cases where the *MHP* algorithm turned out to be less than ideally precise, the number of spurious pairs was small compared to the total number of ideal *MHP* pairs.

Several approaches for computing *MHP* information for programs using various synchronization mechanisms have been suggested. Callahan and Subhlok [4] proposed a data flow algorithm that computes, for each statement in a concurrent program with post-wait synchronization, the set of statements that must be executed before this statement can be executed (B4 analysis). Duesterwald and Soffa [6] applied this approach to the Ada rendezvous model and extended B4 analysis to be interprocedural. Masticola and Ryder [15] proposed an iterative approach that computes a conservative estimate of the set of pairs of communication statements that can never happen in parallel in a concurrent Ada program. (The complement of this set is a conservative approximation of the set of pairs that may occur in parallel.) In that work, it is assumed initially that any statement from a given process can happen in parallel with any statement in any other process. This pessimistic estimate is then improved by a series of refinements that are applied iteratively until a fixed point is reached. This approach yields more precise information than the approaches of Callahan and Subhlok and of Duesterwald and Soffa. Masticola and Ryder show that in the worst case the complexity of their approach is $\mathcal{O}(S^5)$, where S is the number of statements in a program.

Recently, Naumovich and Avrunin [17] proposed a data flow algorithm for computing *MHP* information for programs with a rendezvous model of concurrency. Although the worst-case complexity of this algorithm is $\mathcal{O}(S^6)$, their experimental results suggest that the practical complexity of this algorithm is cubic or less in the number of program statements[1]. Furthermore, the precision of this algorithm was very high for the examples they examined. For a set of 132 concurrent Ada programs, the *MHP* algorithm failed to find the ideal *MHP* information in only 5 cases. For a large majority of the examples, the *MHP* algorithm was more precise than Masticola and Ryder's approach.

The *MHP* algorithm described in this paper is similar in spirit to the algorithm proposed for the rendezvous model but has a number of significant differences prompted by the difference between the rendezvous-based synchronization in Ada and the shared variable-based synchronization in Java. First, the program model for Java is quite different from the one for Ada. Second, while the algorithm for Ada relies on distinguishing between only two node types (nodes representing internal operations in processes and nodes representing inter-process communications), the algorithm for Java has to distinguish between a number of node types corresponding to the eclectic Java synchronization statements. This also implies that the two algorithms employ very different sets of flow equations. Third, while the algorithm for Ada operates on a completely precomputed program model, for Java partial results from the algorithm can be used to modify the program model in order to obtain a more precise estimate of *MHP* information. Finally, the worst-case complexity of the *MHP* algorithm for Java is only cubic in the number of statements in the program.

The next section briefly introduces the Java model of concurrency and the graph model used in this paper. Section 3 describes the *MHP* algorithm in detail and states the major results about its termination, worst-case complexity, and conservativeness. Section 4 describes the results of an experiment in which *MHP* information was computed for a number of concurrent Java programs using both the *MHP* algorithm and the reachability approach in order to evaluate the precision and performance of the algorithm. We conclude with a summary and discussion of future work.

2 Program Model

2.1 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*. Any Java application must contain a unique `main()` method, which serves as the "main" thread of execution. This is the only thread that is running when the program is started. Other threads in the program have to be started explicitly, either by the main thread or by some other already running thread calling 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 important 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

[1] The size of the program model is $\mathcal{O}(S^2)$ in the worst case, and the worst-case complexity of the algorithm is cubic in the size of the program model. It appears that in practice the size of the program model is linear in the number of program statements S [17].

```
class Writer extends Thread
{
  public static void
   main(String [] args)
  {
    Buffer buffer = new Buffer();
    Reader r1 =
      new Reader(buffer);
    Reader r2 =
      new Reader(buffer);
    r1.start();
    r2.start();
    while (notEnough())
    {
      synchronized (buffer)
      {
        buffer.write();
        buffer.notifyAll();
      }
    }
    r1.join();
    r2.join();
  }
}
```

```
class Reader extends Thread
{
  Buffer buffer;
  public Reader(Buffer b)
  {
    buffer = b;
  }
  public void run()
  {
    while (notEnough())
    {
      synchronized (buffer)
      {
        while (buffer.isEmpty())
        {
          buffer.wait();
        }
        buffer.read();
      }
    }
  }
}
```

Fig. 1. Java code example

locks and `synchronized` blocks. Each Java object has an implicit lock, which may be used by `synchronized` blocks and methods. Before a thread can begin execution of a `synchronized` block, this thread must first acquire the lock of the object associated with this block. If this lock is unavailable, which means that another thread is executing a `synchronized` block for this lock, the thread waits until the lock becomes available. A thread releases the lock when it exits the `synchronized` block. 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` blocks protected by that lock. A `synchronized` method of an object `obj` is equivalent to a method in which the whole body is a `synchronized` block protected by the lock of the object `obj`.

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 either the `notify()` or the `notifyAll()` method 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 the notification call will return and execution will continue, whether there are waiting threads or not.

The example in Figure 1 illustrates some of the Java concurrency constructs. In this example, one thread writes into and two threads read from a shared memory buffer. (The source code for the buffer is not shown.) The main thread `Writer` instantiates a buffer object and also instantiates two threads `r1` and `r2` of thread type `Reader`. Note that `r1` and `r2` do not start their execution until the main thread calls their `start()` methods. Each of the threads repeatedly calls `notEnough()`, a function not shown in the figure that determines whether enough data has been written and read. If `notEnough()` returns true, each thread tries to enter the monitor associated with the buffer object. If the main thread enters the monitor, it writes to the buffer and then calls the `notifyAll()` method of the buffer, waking up any threads waiting to enter the monitor, if any. It then leaves the monitor and calls `notEnough()` again. If a reader enters the monitor, it checks whether the buffer contains any data. If so, it reads the data and leaves the monitor. If the buffer is empty, the reader calls the `wait()` method of the buffer object, relinquishing the lock associated with the monitor. The reader then sleeps until awakened by the main thread calling `buffer.notifyAll()`. After a reader thread wakes up, it has to reacquire the lock associated with the buffer object, before it can re-enter the monitor. After it re-enters the monitor, it reads from the buffer and then exits the monitor. After the main thread's call to function `notEnough()` evaluates to `false`, it exits its loop and calls the `join()` methods of threads `r1` and `r2`. If thread `r1` has not terminated by the time the main thread starts executing the call to `r1.join()`, the main thread will block until thread `r1` terminates, and similarly for thread `r1`. Thus, these two calls to `join()` ensure that both reader threads terminate before the main thread does. (The astute reader will notice that this program may deadlock with one of the reader threads waiting for a notification while the buffer is empty and the writer having exited its loop.)

In the rest of the paper we refer to `start()`, `join()`, `wait()`, `notify()`, and `notifyAll()` methods as thread *communication methods* [2].

2.2 Parallel Execution Graph

We use a *Parallel Execution Graph (PEG)* to represent concurrent Java programs. The PEG is built by combining, with special kinds of edges, control flow graphs (CFGs) for all threads that may be started in the program. In general, the number of instances of each thread class may be unbounded. For our analysis we make the usual static analysis assumption that there exists a known upper bound on the number of instances of each thread class. In addition, we assume that alias resolution has been done (e.g., using methods such as [5, 10]). After alias resolution is performed, we can use cloning techniques (e.g. [20]) to resolve object and method polymorphism. Under these assumptions, our program model and algorithm handle "complex" configurations of Java concurrency mechanisms, such as nested `synchronized` blocks, multiple monitors,

[2] Note that concurrency primitives `join()` and `wait()` in Java have "timed" versions. Also, 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. It is not difficult to incorporate handling of these statements in our algorithm but because of space limitations we do not do this here.

multiple instances of lock objects, etc. For example, suppose that alias resolution determines that at some point in the program a particular variable may possibly refer to two different objects. If at that point in the program this variable is used to access a monitor, cloning will produce a structure with two branches, where one branch contains the monitor access with the first object as the lock and the other branch contains the monitor access with the second object as the lock. Thus, alias resolution and cloning techniques are important for improving the precision of the *MHP* analysis.

At present we inline all called methods, except communication methods, into the control flow graphs for the threads. This results in a single CFG for each thread. Each call to a communication method is labeled with a tuple $(\texttt{object}, \texttt{name}, \texttt{caller})$, where `name` is the method name, `object` is the object owning method `name`, and `caller` is the identity of the calling thread. For example, for the code in Figure 1, the call `t1.start` in the main method will be represented with the label $(\texttt{t1}, \texttt{start}, \texttt{main})$. For convenience, we will use this notation for label nodes that do not correspond to method calls by replacing the `object` part of the label with the symbol '$*$'. For example, the first node of a thread `t` is labeled $(*, \texttt{begin}, \texttt{t})$ and the last node of this thread is labeled $(*, \texttt{end}, \texttt{t})$.

To make it easy to reason about groups of communications, we overload the symbol '$*$' to indicate that one of the parts of the communication label can take any value. For example, $(\texttt{t}, \texttt{start}, *)$ represents the set of labels in which some thread in the program calls the `start` method of thread `t`. We will write $n \in (\texttt{t}, \texttt{start}, *)$ to indicate that n is one of the nodes that represent such a call. It will be clear from the context whether a tuple $(\texttt{object}, \texttt{name}, \texttt{caller})$ denotes a label of a single node or a set of nodes with matching labels.

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 $(\texttt{lock}, \texttt{entry}, \texttt{t})$ and $(\texttt{lock}, \texttt{exit}, \texttt{t})$, 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 reason about all these implicit 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 $(\texttt{lock}, \texttt{wait}, \texttt{t})$ represents the execution of the `wait()` method, the node labeled $(\texttt{lock}, \texttt{waiting}, \texttt{t})$ represents the thread being idle while waiting for a notification, and the node labeled $(\texttt{lock}, \texttt{notified-entry}, \texttt{t})$ represents the thread after it received a notification and is in the process of trying to obtain the lock to re-enter the `synchronized` block. The shaded regions in the figure represent the `synchronized` block.

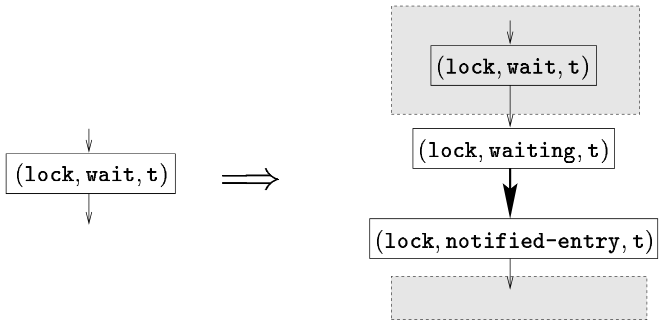

Fig. 2. CFG transformation for `wait()` method calls

The CFGs for all threads in the program are combined in a PEG by adding special kinds of edges between nodes from different CFGs. We define the following edge kinds. A *waiting* edge is created between `waiting` and `notified-entry` nodes (the bold edge in Figure 2). A *local* edge is a non-waiting edge between two nodes from the same CFG. For example, the edge between nodes labeled $(\texttt{lock}, \texttt{wait}, \texttt{t})$ and $(\texttt{lock}, \texttt{waiting}, \texttt{t})$ is a local edge. A *notify* edge is created from a node m to a node n if m is labeled $(\texttt{obj}, \texttt{notify}, \texttt{r})$ or $(\texttt{obj}, \texttt{notifyAll}, \texttt{r})$ and n is labeled $(\texttt{obj}, \texttt{notified-entry}, \texttt{s})$ for some thread object `obj`, where threads `r` and `s` are different. The set of notify edges is not precomputed but rather built during the algorithm. This improves the precision of the algorithm since information does not propagate into `notified-entry` nodes from `notify` and `notifyAll` nodes until it is determined that these statements may happen in parallel. A *start* edge is created from a node m to a node n if m is labeled $(\texttt{t}, \texttt{start}, *)$ and n is labeled $(*, \texttt{begin}, \texttt{t})$. That is, m represents a node that calls the `start()` method of the thread `t` and n is the first node in the CFG of this thread. All start edges can be computed by syntactically matching node labels.

Figure 3 shows the PEG for the program in Figure 1. The shaded regions include nodes in the monitor of the program; thin solid edges represent local control flow within individual threads; the thick solid edges are waiting edges; the dotted edges are start edges; and the dashed edges are notify edges. (Note that these notify edges are not present in the PEG originally but will be created during execution of the *MHP* algorithm.)

2.3 Additional Terminology

For convenience, we define a number of functions. *LocalPred*(n) returns the set of all immediate local predecessors of n; *NotifyPred*(n) returns the set of all notify predecessors of a `notified-entry` node n; *StartPred*(n) returns the set of all start predecessors of a `begin` node n; and *WaitingPred*(n) returns a single `waiting` predecessor of a `notified-entry` node. Sets of successors *LocalSucc*(n), *NotifySucc*(n), *StartSucc*(n), and *WaitingSucc*(n) are defined similarly. Let T denote the set of all threads that the program may create. Let $N(t)$ denote the set of all PEG nodes in thread $t \in T$. Furthermore, we define a function *thread* $: N \to T$ that maps each node in the

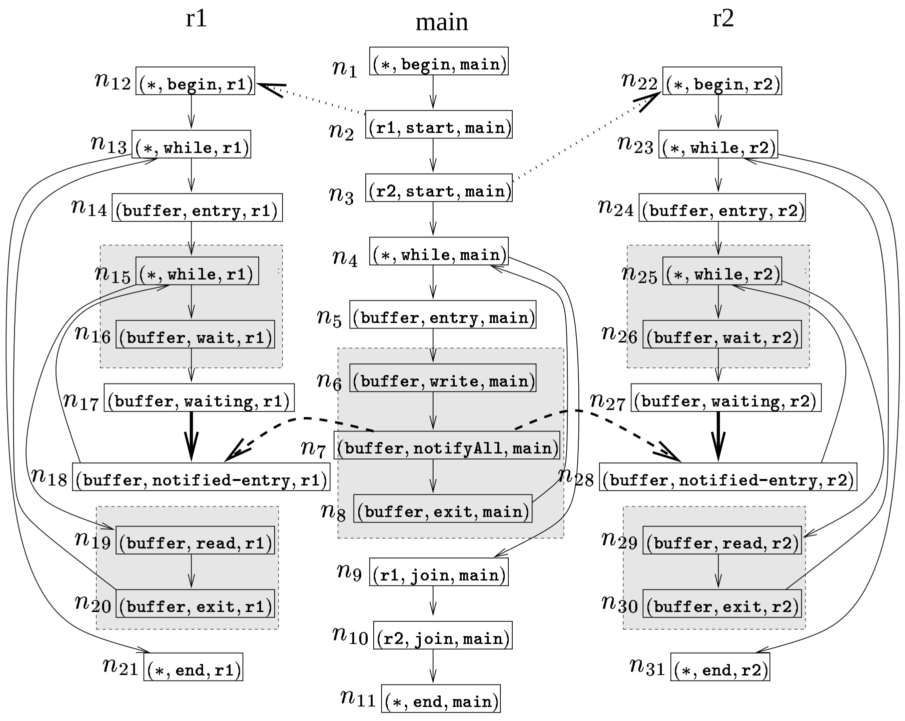
r1
main
r2
n_1 (∗, begin, main)
n_2 (r1, start, main)
n_3 (r2, start, main)
n_4 (∗, while, main)
n_5 (buffer, entry, main)
n_6 (buffer, write, main)
n_7 (buffer, notifyAll, main)
n_8 (buffer, exit, main)
n_9 (r1, join, main)
n_{10} (r2, join, main)
n_{11} (∗, end, main)
n_{12} (∗, begin, r1)
n_{13} (∗, while, r1)
n_{14} (buffer, entry, r1)
n_{15} (∗, while, r1)
n_{16} (buffer, wait, r1)
n_{17} (buffer, waiting, r1)
n_{18} (buffer, notified-entry, r1)
n_{19} (buffer, read, r1)
n_{20} (buffer, exit, r1)
n_{21} (∗, end, r1)
n_{22} (∗, begin, r2)
n_{23} (∗, while, r2)
n_{24} (buffer, entry, r2)
n_{25} (∗, while, r2)
n_{26} (buffer, wait, r2)
n_{27} (buffer, waiting, r2)
n_{28} (buffer, notified-entry, r2)
n_{29} (buffer, read, r2)
n_{30} (buffer, exit, r2)
n_{31} (∗, end, r2)

Fig. 3. PEG example

PEG to the thread to which this node belongs. For example, for the PEG in Figure 3, *thread*$(n_5) =$ `main` and *thread*$(n_{15}) =$ `r1`.

For convenience, we associate two sets with each lock object. *notifyNodes*(`obj`) is the set of all `notify` and `notifyAll` nodes for lock object `obj`: *notifyNodes*(`obj`) $=$ (`obj`, `notify`, $*$) $\cup$ (`obj`, `notifyAll`, $*$). Similarly, *waitingNodes*(`obj`) is the set of all `waiting` nodes for lock object `obj`: *waitingNodes*(`obj`) $=$ (`obj`, `waiting`, $*$). For the example in Figure 3, *notifyNodes*(`buffer`) $= \{n_7\}$ and *waitingNodes*(`buffer`) $= \{n_{17}, n_{27}\}$.

In Java, monitors are given explicitly by `synchronized` blocks and methods. Since our model captures a set of known threads, we can also statically compute the set of nodes representing code in a specific monitor. Let $Monitor_{\texttt{obj}}$ denote the set of PEG nodes in the monitor for the lock of object `obj`. For the example in Figure 3, $Monitor_{\texttt{buffer}} = \{n_6, n_7, n_8, n_{15}, n_{16}, n_{19}, n_{20}, n_{25}, n_{26}, n_{29}, n_{30}\}$.

3 The *MHP* Algorithm

In this section we present the data flow equations for the *MHP* algorithm. We do this instead of using the lattice/function space view of data flow problems [7] since it makes explanations of this algorithm more intuitive and, as will be evident, one aspect of this algorithm precludes its representation as a purely forward- or backward data flow problem or even as a bidirectional or multisource [16] data flow problem. At the end of this section we present a pseudo-code version of the worklist version of the *MHP* algorithm.

3.1 High-Level Overview

Initially we assume that each node in the PEG may not happen in parallel with any other nodes. The data flow algorithm then uses the PEG to infer that some nodes may happen in parallel with others and propagates this information from one node to another, until a fixed point is reached. At this point, for each node, the computed information represents a conservative overapproximation of all nodes that may happen in parallel with it.

To each node n of the PEG we assign a set $M(n)$ containing nodes that may happen in parallel with node n, as computed at a given point in the algorithm. In addition to the M set, we associate an *OUT* set with each node in the PEG. This set includes *MHP* information to be propagated to the successors of the node. The reason for distinguishing between the $M(n)$ and $\mathit{OUT}(n)$ sets is that, depending on the thread synchronizations associated with node n, it is possible that a certain node m may happen in parallel with node n but may never happen in parallel with n's successors or that some nodes that may not happen in parallel with n may happen in parallel with n's successors. Section 3.4 gives a detailed description of all cases where nodes are added to or removed from the M set of a node to obtain the *OUT* set for this node. Initially, M and *OUT* sets for all nodes are empty.

We propose a worklist form of the *MHP* algorithm. At the beginning, the worklist is initialized to contain all `start` nodes in the main thread of the program that are reachable from the begin node of this main thread. The reason for this is that places in the main thread of the program where new threads are started are places where new parallelism is initiated. The *MHP* algorithm then runs until the worklist becomes empty. At each step of the algorithm a node is removed from the worklist and the notify edges that come into this node, as well as the M and *OUT* sets for this node, are recomputed. If the *OUT* set changes, all successors of this node are added to the worklist. The following four subsections describe the major steps taken whenever a node is removed from the worklist. This node is referred to as the current node.

3.2 Computing Notify Edges

Notify edges connect nodes representing calls to `notify()` and `notifyAll()` methods of an object to `notified-entry` nodes for this object. The intuition behind these edges is to represent the possibility that a call to `notify()` or `notifyAll()` method wakes a waiting thread (the waiting state of this thread is represented by the corresponding `waiting` node) and this thread consequently enters the corresponding `notified-entry` node. This is possible only if the `waiting` node and the `notify` or `notifyAll` node may happen at the same time. Thus, the computation of notify successors for the current node can be captured concisely as

$$\mathit{NotifySucc}(n) = \begin{cases} \{m | m \in (\texttt{obj}, \texttt{notified-entry}, *) \\ \quad \wedge \mathit{WaitingPred}(m) \in M(n)\}, & \text{if } n \in \mathit{notifyNodes}(\texttt{obj}) \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

3.3 Computing M Sets

To compute the current value of the M set for the current node, we use the *OUT* sets of this node's predecessors, as well as information depending on the label of this node. Equation (1) gives the rule for computing the M set for nodes with all possible labels (here and in the rest of the paper, "\" stands for the set subtraction operation).

$$M(n) = M(n) \cup \begin{cases} (\bigcup_{p \in \mathit{StartPred}(n)} \mathit{OUT}(p) & \\ \quad \setminus N(\mathit{thread}(n))), & \text{if } n \in (*, \texttt{begin}, *) \\ ((\bigcup_{p \in \mathit{NotifyPred}(n)} \mathit{OUT}(p)) & \\ \quad \cap \mathit{OUT}(\mathit{WaitingPred}(n))) & \\ \quad \cup \mathit{GEN}_{\mathit{notifyAll}}(n), & \text{if } n \in (*, \texttt{notified-entry}, *) \\ \bigcup_{p \in \mathit{LocalPred}(n)} \mathit{OUT}(p), & \text{otherwise} \end{cases} \tag{1}$$

As seen in equation (1), for `begin` nodes, the M set is computed as the union of the *OUT* sets of all start predecessors of this node, with all nodes from the thread of the current node excluded[3]. The explanation is that since the `start()` method is non-blocking, the first node in the thread that is started may execute in parallel with all nodes that may execute in parallel with the node that started it.

For a `notified-entry` node n, first we compute the union of the *OUT* sets of all notify predecessors of this node. The resulting set of nodes is then intersected with the *OUT* set of the waiting predecessor of n and then the $\mathit{GEN}_{\mathit{notifyAll}}(n)$ set (defined in equation (2) below) is added to the result. The intuition behind taking the union of the *OUT* sets of all notify predecessors is that once a thread executes `wait()`, it becomes idle, and quite some time can pass before it is awakened by some other thread. Only after this happens (after a `notify()` or `notifyAll()` method call) can this thread resume its execution. This means that, in effect, these `notify` and `notifyAll` nodes are the "logical" predecessors of the node that follows the `waiting` node. The reasoning for intersecting the resulting set with the *OUT* set of n's `waiting` predecessor is that n can execute only if (1) the thread of n is waiting for a notification and (2) one of the notify predecessors of n executes.

The $\mathit{GEN}_{\mathit{notifyAll}}$ set in equation (1) handles the special case of a `notifyAll` statement awakening multiple threads. In this case the corresponding `notified-entry` nodes in these threads may all execute in parallel. We conservatively estimate the sets of such `notified-entry` nodes from threads other than that of the current node n. A node m is put in $\mathit{GEN}_{\mathit{notifyAll}}(n)$ if m refers to the same lock object `obj` as n does, the *WaitingPred* nodes of m and n may happen in parallel, and there is a node r labeled

[3] `begin` nodes are included in the *OUT* sets of the corresponding `start` nodes. See equations (3) and (4).

$(\texttt{obj}, \texttt{notifyAll}, *)$ that is a notify predecessor of both m and n. Formally,

$$GEN_{\mathit{notifyAll}}(n) = \begin{cases} \emptyset, & \text{if } n \notin (\texttt{obj}, \texttt{notified-entry}, *) \\ \{m | m \in (\texttt{obj}, \texttt{notified-entry}, *) \wedge \\ \quad \mathit{WaitingPred}(n) \in M(\mathit{WaitingPred}(m)) \wedge \\ \quad (\exists r \in N : r \in (\texttt{obj}, \texttt{notifyAll}, *) \wedge \\ \quad r \in (M(\mathit{WaitingPred}(m)) \cap M(\mathit{WaitingPred}(n))))\}, \\ & \text{if } n \in (\texttt{obj}, \texttt{notified-entry}, *) \end{cases} \tag{2}$$

3.4 Computing *OUT* Sets

The $\mathit{OUT}(n)$ set represents *MHP* information that has to be passed to the successors of n and is computed as shown in equation (3).

$$\mathit{OUT}(n) = (M(n) \cup \mathit{GEN}(n)) \setminus \mathit{KILL}(n) \tag{3}$$

$\mathit{GEN}(n)$ is the set of nodes that, although they may not be able to execute in parallel with n, may execute in parallel with n's successors. $\mathit{KILL}(n)$ is a set of nodes that must not be passed to n's successors, although n itself may happen in parallel with some of these nodes. Computation of both *GEN* and *KILL* for the current node depends on the label of this node.

The following equation gives the rule for computing the *GEN* set for the current node n.

$$\mathit{GEN}(n) = \begin{cases} (*, \texttt{begin}, \texttt{t}), & \text{if } n \in (\texttt{t}, \texttt{start}, *) \\ \mathit{NotifySucc}(n), & \text{if } \exists \texttt{obj} : n \in \mathit{notifyNodes}(\texttt{obj}) \\ \emptyset, & \text{otherwise} \end{cases} \tag{4}$$

For `start` nodes, *GEN* consists of a single node that is the begin node in the thread that is being started. Suppose, the current node is in thread `r`, starting thread `t`. Once this node is executed, thread `t` is ready to start executing in parallel with thread `r`. Thus, the begin node of thread `r` has to be passed to all successors of the current node. Note that thread `r` cannot start until `start` node completes its execution. This means this `start` node and the `begin` node of thread `r` may not happen in parallel, and so the `begin` node is not in the M set of the `start` node.

For `notify` and `notifyAll` nodes, the *GEN* set equals the set of all their notified successors. This conveys to the local successors of such `notify` and `notifyAll` nodes that they may happen in parallel with all such `notified-entry` nodes. Note that these `notified-entry` nodes may or may not be in the M sets of the `notify` and `notifyAll` nodes because a thread that is being awakened becomes notified only after the corresponding `notify` or `notifyAll` node completes its execution.

The computation of the *KILL*(n) set is shown below.

$$\mathit{KILL}(n) = \begin{cases} N(t), & \text{if } n \in (\mathtt{t}, \mathtt{join}, *) \\ \mathit{Monitor}_{\mathtt{obj}}, & \text{if } n \in (\mathtt{obj}, \mathtt{entry}, *) \cup \\ & \quad (\mathtt{obj}, \mathtt{notified\text{-}entry}, *) \\ \mathit{waitingNodes}(\mathtt{obj}), & \text{if } (n \in (\mathtt{obj}, \mathtt{notify}, *) \wedge \\ & \quad \mid \mathit{waitingNodes}(\mathtt{obj}) \mid = 1) \vee \\ & \quad (n \in (\mathtt{obj}, \mathtt{notifyAll}, *)) \\ \emptyset, & \text{otherwise} \end{cases} \tag{5}$$

If the current node n represents `join`ing another thread t, the thread containing the current node will block until thread t terminates. This means that after n completes its execution, no nodes from t may execute. Thus, all nodes from $M(n) \cap N(t)$ should be taken out of the set being passed to n's successors.

Computing the *KILL* set for `entry` and `notified-entry` nodes is quite intuitive. While a thread is executing in a `entry` or `notified-entry` node, it is not in the monitor entrance that this node represents. Once the execution of this node terminates, the thread is inside this monitor. Thus, the successors of such `entry` or `notified-entry` node may not happen in parallel with any nodes from this monitor.

Finally, if the current node is a `notifyAll` node for lock object `obj`, this means that once this node completes its execution, no threads in the program will be waiting on this object. Thus, no nodes labeled $(\mathtt{obj}, \mathtt{waiting}, *)$ must be allowed to propagate to the local successors of the current node. If the current node is a `notify` node, its execution wakes up no more than one thread. If there is exactly one `waiting` node, this `waiting` node must finish execution by the time this `notify` node finishes its execution.

3.5 Symmetry Step

Up to this point the algorithm is a standard forward data flow algorithm. After computing M and *OUT* sets for each node, however, we have to take a step that is outside this classification to ensure the symmetry $n_1 \in M(n_2) \Leftrightarrow n_2 \in M(n_1)$. We do this by adding n_2 to $M(n_1)$ if $n_1 \in M(n_2)$. The nodes whose M sets have been updated in this way are added to the worklist, since the change in their M sets may result in a change in their *OUT* sets, and so influence other nodes in the graph.

3.6 Worklist Version of the *MHP* Algorithm

The Java *MHP* algorithm, based on the equations described above, consists of two stages. The initialization stage computes *KILL* sets for all nodes, as well as the *GEN* sets for the `start` nodes. All steps of this stage correspond to computations described by equations (4) and (5). The iteration stage computes M and *OUT* sets and notify edges using a worklist containing all nodes that have to be investigated. Both stages of the algorithm are shown in Figure 4.

3.7 Termination, Conservativeness, and Complexity

For Java programs satisfying the assumptions noted in Section 2, we have proved that the *MHP* algorithm terminates, that *MHP* information computed by this algorithm is

Input: CFGs for all threads in the program and $\forall n \in N$: sets *KILL*(n) and *GEN*(n)
Output: $\forall n \in N$: a set of PEG nodes $M(n)$ such that $\forall m \notin M$: m may never happen in parallel with n.
Additional Information: W is the worklist containing nodes to be processed
$\forall n \in N$, *OUT*(n) is the set of nodes to be propagated to the successors of n
Initialization: $\forall n \in N$: *KILL*(N) = *GEN*$(N) = M(n) =$ *OUT*$(n) = \emptyset$
Initialize the worklist W to include all `start` nodes in the `main` thread that are reachable from the begin node of the main thread
THE FIRST STAGE:

```
(1)   $\forall n \in N$ :
(2)       case
(3)           $n \in (\texttt{t}, \texttt{join}, *) \Rightarrow KILL(n) = N(t)$
(4)           $n \in (\texttt{obj}, \texttt{entry}, *) \cup (\texttt{obj}, \texttt{notified-entry}, *) \Rightarrow KILL(n) = Monitor_{\texttt{obj}}$
(5)           $n \in (\texttt{obj}, \texttt{notifyAll}, *) \Rightarrow KILL(n) = waitingNodes(\texttt{obj})$
(6)           $n \in (\texttt{obj}, \texttt{notify}, *) \Rightarrow$
(7)               if $|\ waitingNodes(\texttt{obj})\ | = 1$ then
(8)                   $KILL(n) = waitingNodes(\texttt{obj})$
(9)           $n \in (\texttt{t}, \texttt{start}, *) \Rightarrow GEN(n) = (*, \texttt{begin}, \texttt{t})$
```

THE SECOND STAGE: Main loop.
We evaluate the following statements repeatedly until $W = \emptyset$

```
      // $n$ is the current node:
(1)   $n = head(W)$
      // $n$ is removed from the worklist:
(2)   $W = tail(W)$
      // $M_{old}$, $OUT_{old}$, and $NotifySucc_{old}$ are the copies of the $M$, $OUT$, and $NotifySucc$ sets for this node,
      // computed to determine new nodes inserted in these sets on this iteration
(3)   $M_{old} = M(n)$
(4)   $OUT_{old} = OUT(n)$
(5)   $NotifySucc_{old} = NotifySucc(n)$
      // computing the new set of notify successors for notify and notifyAll nodes
(6)   if $\exists o : n \in notifyNodes(\texttt{obj})$ then
(7)       $\forall m \in M(n) \cap waitingNodes(\texttt{obj})$:
              // create a new notify edge from node $n$ to the waiting successor of node $m$
(8)           $NotifySucc(n) = NotifySucc(n) \cup \{WaitingSucc(m)\}$
          // if new notify edges were added from this node, add all notify successors
          // of this node to the worklist
(9)       if $NotifySucc_{old}(n) \neq NotifySucc(n)$ then
(10)          $W = W \cup NotifySucc(n)$
(11)  Compute the set $GEN_{notifyAll}(n)$ as in equation (2)
(12)  Compute the set $M(n)$ as in equation (1)
      // the only nodes for which the GEN set has to be recomputed are notify and
      // notifyAll nodes; their GEN sets are their notify successors:
(13)  if $\exists o : n \in notifyNodes(\texttt{obj})$ then
(14)      $GEN(n) = NotifySucc(n)$
(15)  Compute the set $OUT(n)$ as in equation (3)
      // do the symmetry step for all new nodes in $M(n)$:
(16)  if $M_{old} \neq M(n)$ then
(17)      $\forall m \in (M(n) \setminus M_{old}(n))$:
(18)          $M(m) = M(m) \cup \{n\}$
              // add $m$ to the worklist because the change in $M(m)$ may lead to a
              // change in $OUT(m)$
(19)          $W = W \cup \{m\}$
      // if new nodes has been added to the OUT set of $n$, add all $n$'s successors to the worklist
(20)  if $OUT_{old} \neq OUT(n)$:
(21)      $W = W \cup (LocalSucc(n) \cup StartSucc(n))$
```

Fig. 4. *MHP* algorithm

conservative, in the sense that the sets it computes contain the ideal *MHP* sets, and that the worst-case time bound for this algorithm is $\mathcal{O}(|N|^3)$. The precise formulations of these statements and their proofs are given in [18].

4 Experimental Results

We measure the precision of both the *MHP* algorithm and the reachability analysis by the number of pairs of nodes that each approach claims can happen in parallel. The smaller this number, the more precise is the approach, since both approaches can never underestimate the set of nodes that may happen in parallel. We write P_{MHP} for the set of pairs found by the *MHP* algorithm and P_{Reach} for the set of pairs found by the reachability analysis. We say that the *MHP* algorithm is perfectly precise if $P_{MHP} = P_{Reach}$. For the cases where this equality does not hold, we are interested in the ratio between the number of spurious *MHP* pairs and the number of all pairs found by the reachability analysis $(|P_{MHP} \setminus P_{Reach}|)/|P_{Reach}|$. (Note that conservativeness of our algorithm guarantees $P_{Reach} \subseteq P_{MHP}$.)

Because there is no standardized benchmark suite of concurrent Java programs, we collected a set of Java programs from several available sources. We modified these programs so that we could use a simple parser to create the PEG without any preliminary semantical analysis. In addition, we removed the timed versions of the synchronization statements and any exception handling, since these are currently not handled in our algorithm.

The majority of our examples came from Doug Lea's book on Java concurrency [11] and its Web supplement [12]. For most of these examples Lea gives only the classes implementing various synchronization schemes, sometimes with a brief example of their use in concurrent programs. We used these synchronization schemes to construct complete multi-threaded programs. We selected sizes of the examples that could be handled by our reachability tool.

Several examples in our set came from other sources [1–3] on the Web. Finally, we wrote Java implementations for several of the Ada concurrent examples, such as dining philosophers, that are commonly used in the concurrency analysis literature. All 29 examples that we used in our experiments are described in [18].

For each example we compute three times: the time to build the PEG model, the time to run the *MHP* algorithm on this model, and the time to run the reachability analysis on this model. Both approaches are implemented in Java. For our experiments, we used a Symantec JIT compiler for JDK 1.1 on a workstation equipped with a 400 MHz Pentium II processor and 128Mb of memory, running Windows NT.

Figure 5 presents the raw data from running the *MHP* and reachability algorithms on our set of examples. In this figure, for each example Java program, the first column gives the name of the program; the second column gives the number of threads, including the main thread; the third column gives the overall number of nodes in the PEG model of the example; the fourth column gives the number of nodes that are used to model thread synchronizations, e.g., `waiting` nodes; the fifth column gives the number of node pairs found by the *MHP* algorithm; the sixth column gives the number of node pairs found by the *MHP* algorithm but not by the reachability analysis (thus, the number of node pairs found by the reachability analysis can be found by subtracting the number in the sixth

Program	**|T|**	**|N|**	**Synch. nodes**	**$\lvert P_{MHP} \rvert$**	**$\lvert P_{MHP} \setminus P_{Reach} \rvert$**	**time PEG**	**time MHP**	**time Reach**
AlternatingMessagePrinter	3	35	27	261	0	0.33	0.10	0.08
AutomatedBanking	3	280	182	11166	0	0.27	2.03	7.51
BridgeTest	5	66	46	1193	0	0.24	0.15	11.79
CHAN_OF_INT	5	62	54	974	40	0.26	0.13	5.76
Chiron	5	112	87	3382	0	0.26	0.29	1059.57
CorrectHalves	3	33	24	188	0	0.67	0.11	0.08
CorrectSquares	4	33	21	271	0	0.24	0.08	0.14
CyclicBarrier	5	54	38	887	0	0.3	0.12	10.45
GroupPictureRenderer	4	35	26	240	0	0.23	0.09	0.13
GasStation	5	93	71	2626	0	0.29	0.24	84.61
HeatingSystem	4	66	41	1140	0	0.27	0.17	2.69
HeatingSystemPutTake	4	76	53	1497	0	0.25	0.21	4.56
IncorrectHalves	3	31	20	174	0	0.23	0.08	0.06
IncorrectSquares	4	27	15	171	0	0.23	0.08	0.09
LayeredSemaphore	4	77	59	1720	0	0.24	0.21	5.90
Mutex	5	75	54	1688	0	0.69	0.23	36.25
OneCarBridgeTest	7	39	32	420	0	0.29	0.08	10.29
PCTwoLockQueue	5	34	22	358	0	0.25	0.09	1.01
PessimisticBankAccount	3	446	288	44128	0	0.35	6.28	41.91
Phil	4	68	59	1355	0	0.26	0.19	3.78
PrintService	3	23	16	108	0	0.27	0.08	0.05
Readers-writers	5	56	46	876	0	0.24	0.11	7.75
RWVSN	5	116	80	3224	0	0.30	0.27	107.81
Semaphore	5	66	46	1349	0	0.49	0.17	25.23
SemaphoreControlledChannel	5	66	46	1423	0	0.25	0.16	32.22
SplitRenderer	5	33	18	250	0	0.21	0.08	0.22
SplitRendererNested	7	51	26	727	50	0.25	0.09	4.74
ThreadedApplet	3	10	8	18	0	0.20	0.06	0.04
ThreadedAppletV2	3	14	12	36	0	0.21	0.08	0.05

Fig. 5. Raw experimental data

column from the number in the fifth column); finally, the seventh, eighth, and ninth columns show the time in seconds taken to construct the PEG for the example and to run the *MHP* and reachability algorithms respectively.

Out of the 29 example programs, the *MHP* algorithm was less precise than the reachability algorithm on only two examples, CHAN_OF_INT and SplitRendererNested. In both of these cases the number of spurious pairs was small compared to the total number of pairs of nodes that may happen in parallel (40 out of 934 and 50 out of 677 respectively).

The timing data indicate that in practice the *MHP* algorithm is very efficient. For all examples, except the AutomatedBanking and PessimBankAccount examples, running the *MHP* algorithm took under 0.3 second. For all but the simplest examples, running the *MHP* algorithm took much less time than running the reachability analysis. In fact, for most examples it took more time to construct the PEG model than it took to run the *MHP* algorithm.

5 Conclusions

Information about which pairs of statements may execute in parallel has important applications in optimization, detection of anomalies such as race conditions, and improving the accuracy of data flow analysis. Efficient and precise algorithms for computing this information are therefore of considerable value. In this paper, we have described a data flow method for computing a conservative approximation of the set of pairs of statements in a concurrent Java program that may execute in parallel. Our algorithm has a worst-case bound that is cubic in the number of statements in the program.

We carried out an initial experiment evaluating the precision of our algorithm against the precision of a technique based on exhaustive exploration of the program state space. Since this reachability technique, which is exponential in the program size, is not practical in general, we restricted the size of our example programs to those for which we could compute the "ideally" precise *MHP* information. On 27 of the 29 example programs, the *MHP* algorithm produced as precise results as the reachability analysis.

In the future, we plan to improve the applicability of the *MHP* algorithm by eliminating the use of inlining in constructing the program model. Even in its current form, the *MHP* algorithm does not require inlining methods that do not contain thread synchronizations. Such methods calls may be represented in the PEG for the program with a single node, where *MHP* information computed for this node is sufficient to determine *MHP* information for all nodes in the corresponding method. Thus, if n is a call node for method `M`, then any node in the body of `M` may happen in parallel with any node that may happen in parallel with a node representing the call to `M`. Special care must be taken when there is a possibility that a method may be called by more than one thread, in which case executions of multiple instances of this method may overlap in time. In this case, unlike thread nodes, *MHP* information for the nodes from this method will contain other nodes from the same method. To determine whether this might happen, we have to check whether any of the call nodes to `M` is in the *MHP* set of any of the other call nodes to this method (this has to be done recursively for nested method calls), in which case the *MHP* sets of all nodes in `M` must contain all nodes in `M`.

In the case of methods containing thread synchronization mechanisms, we plan to use a context-sensitive approach, extending the PEG model to include method *call* and *return* edges, similar to the approach of [8], and modifying the *MHP* algorithm accordingly.

The run time performance of the *MHP* algorithm can benefit from optimizations of the PEG model. For example, node coarsening approaches, such as described in [13,15], replace sequential regions of code that have no interaction with other threads by a single node. The resulting reduction in the number of nodes in the parallel execution graph should improve the performance of the *MHP* algorithm.

At present, the *MHP* algorithm is being used as a part of the FLAVERS/Java tool [19] for data flow-based verification of application-specific properties of concurrent Java programs. The program model used by FLAVERS represents the possibility of interleaving of events from different threads by edges of a special kind. We use the results of the *MHP* algorithm for computing such edges by creating an edge from node n to node m if m is placed in set $OUT(n)$ by the *MHP* algorithm. Using the *MHP* algorithm results in a more precise model of concurrent execution. We plan to measure the impact of the precision improvements obtained and overheads incurred by using the *MHP* algorithm on data flow algorithms used in verification of concurrent Java programs.

References

1. http://vislab-www.nps.navy.mil/ java/course/sourcecode.
2. http://www.hensa.ac.uk/parallel/groups/wotug/java/applets.
3. http://www.kai.com/assurej.
4. D. Callahan and J. Subhlok. Static analysis of low-level synchronization. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, number 1, pages 100–111, Jan. 1989.
5. J. C. Corbett. Constructing compact models of concurrent Java programs. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA-98)*, pages 1–10, Mar. 1998.
6. E. Duesterwald and M. L. Soffa. Concurrency analysis in the presence of procedures using a data flow framework. In *Proceedings of the ACM SIGSOFT Fourth Workshop on Software Testing, Analysis, and Verification*, pages 36–48, Oct. 1991.
7. M. Hecht. *Flow Analysis of Computer Programs*. North-Holland, New York, 1977.
8. S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 104–115, Oct. 1995.
9. J. Krinke. Static slicing of threaded programs. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 35–41, June 1998.
10. 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, Jan. 1991.
11. D. Lea. *Concurrent Programming in Java*. Addison-Wesley, Reading, MA, 1997.
12. D. Lea. Concurrent programming in Java. Design principles and patterns, online supplement. http://gee.cs.oswego.edu/dl/cpj/index.html, Sept. 1998.
13. D. L. Long and L. A. Clarke. Task interaction graphs for concurrency analysis. In *Proceedings of the 11th International Conference on Software Engineering*, pages 44–52, May 1989.
14. S. Masticola and B. Ryder. A model of Ada programs for static deadlock detection in polynomial time. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 97–107, May 1991.
15. S. Masticola and B. Ryder. Non-concurrency analysis. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 129–138, May 1993.
16. S. P. Masticola, T. J. Marlowe, and B. G. Ryder. Lattice frameworks for multisource and bidirectional data flow problems. *ACM Transactions on Programming Languages and Systems*, 17(5):777–803, September 1995.
17. 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.
18. 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.

19. G. Naumovich, G. S. Avrunin, and L. A. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *Proceedings of the 21st International Conference on Software Engineering*, pages 399–410, May 1999.
20. 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, Oct. 1994.
21. R. N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.