An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs

Gleb Naumovich, George S. Avrunin and Lori A. Clarke

CMPSCI Technical Report 98-44
October 5, 1998

Laboratory for Advanced Software Engineering Research
Computer Science Department
University of Massachusetts

Effort partially supported by the Air Force Materiel Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under Contract number F30602-97-2-0032 and by the National Science Foundation under Grants CCR9407182 and CCR-9708184.

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

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 in program optimization, debugging, program understanding tools, improving the accuracy of data flow approaches (e.g. [11, 8, 14]), and detecting synchronization anomalies, such as data races. For example, 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 [15]. In this paper, we call the solution with this assumption the *ideal MHP* information for a program. To compute the *MHP* information efficiently, 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 set 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 the *MHP* information for Java programs with 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 of our algorithm and a reachability analysis that determines the ideal *MHP* information for concurrent Java programs. 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 the *MHP* information for programs using various synchronization mechanisms have been proposed. 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 [5] applied this approach to the Ada rendezvous model and extended B4 analysis to be interprocedural. Masticola and Ryder proposed an iterative approach of non-concurrency analysis [12] that computes a conservative estimate of the set of pairs of communication statements that can never happen in parallel in a concurrent Ada program.

*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.

(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. Masticola and Ryder show that their algorithm yields more precise information than the approaches of Callahan and Subhlok and of Duesterwald and Soffa.

Recently, Naumovich and Avrunin [13] proposed a data flow algorithm for computing the *MHP* information for programs with a rendezvous model of concurrency. Although the worst-case complexity of this algorithm is $\mathcal{O}(S^6)$, where S is the number of statements in a program, their experimental results suggest that the practical complexity of this algorithm is cubic or less in the number of program statements. Furthermore, the practical precision of this algorithm was very high. 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 non-concurrency analysis.

The *MHP* algorithm described in this paper is similar in spirit to the algorithm in [13] 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 tasks and nodes representing intertask communications), the algorithm for Java has to distinguish between a number of nodes 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, the program model used by the algorithm for Java can be improved during the algorithm run, in the interests of computing a more precise estimate of the *MHP* information for the program. 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 describes the Java concurrency model and the terminology used in this paper and describes a graph model for concurrent Java programs. Section 3 describes the *MHP* algorithm in detail. The complexity of the *MHP* algorithm is discussed in Section 4, where a more efficient form of the algorithm is proposed. Section 5 introduces a reachability approach to computing the "ideal" *MHP* information and proves conservativeness of our algorithm with respect to this information. Section 6 describes the results of an experiment in which the *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 of the algorithm. We conclude with a summary and discussion of future work.

2 Overview

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*. There are two ways in which new thread classes can be defined. One is to extend the standard class `Thread` and the other is to implement the standard `Runnable` interface. The distinction between these ways is subtle but both involve specifying executable behaviors for threads of this class in the method with the predefined name `run()`. For the purposes of this paper we can assume without loss of generality that all thread classes are defined by extending the `Thread` class. Figure 1 contains an example in which thread class `MyThread` is defined by extending the standard Java `Thread` class. Thread `t1` of this class is created and used in the main method of class `Example`.

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. 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

```
class MyThread extends Thread
{
  Object lock;
  public MyThread(Object obj)
  {
    lock = obj;
  }
  public void run()
  {
    synchronized (lock)
    {
      lock.notify();
    }
    System.out.print(''Thread t1 done'');
  }
}
```

```
class Example extends Thread
{
  public static void
   main(String [] args)
  {
    Object lock = new Object();
    MyThread t1 = new MyThread(lock);
    synchronized (lock)
    {
      t1.start();
      lock.wait();
    }
    t1.join();
  }
}
```

Figure 1: Java code example

the appropriate constructor) thread `t1` of class `MyThread` and later starts it by invoking its `start()` method.

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. In Figure 1, the main thread is blocked until `t1` terminates at the point where it calls the `join()` method of thread `t1`.

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 `synchronized` blocks and *locks*. 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 blocks until the lock becomes available. 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 regular method, the whole body of which is a `synchronized` block protected by the lock of the object `obj`. In Figure 1, an object `lock` of Java predefined class `Object` is used to create the monitor in which both threads `main` and `t1` participate. Note that the identity of object `lock` has to be conveyed to thread `t1`. In this case this is done via the constructor `new MyThread(lock)`.

Threads may interrupt their execution in monitors by calling the `wait()` method of the lock object of this monitor. During execution of the `wait()` method, the thread releases the lock and becomes inactive, thereby giving other threads an opportunity to acquire this lock. The main thread Figure 1 performs such a call to the `wait()` method of object `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 whether there are waiting threads or not, the notification call will return and execution will continue. In Figure 1, thread `t1` calls the `notify()` method of object `lock`.

The example in Figure 1 will be used throughout this paper for illustrations. Despite the presence of two threads (the main thread and thread `t1`) in this example, there is very little non-determinism during execution of this program due to synchronization mechanisms used. Because the main thread starts threads `t1` while this main thread is executing the `synchronized` block for object `lock`, thread `t1` cannot start execution of its `synchronized` block until the main thread releases the lock of object `lock`. The main thread does that by calling the `wait()` method of `lock`. After that thread `t1` can enter its `synchronized` block and execute the `notify()` method of object `lock`. At this point the main thread is awakened but before

it can continue, it has to ackquire the lock of object `lock` again. The main thread cannot acquire this lock immediately because first thread `t1` has to release this lock. This happens when thread `t1` exits its synchronized block. The only non-determinism in the program execution can happen only now. Execution of the print statement by thread `t1` and reacquiring the lock of object `lock` and subsequent exiting the `synchronized` block by the main thread may be interleaved. If by the time the main thread starts executing the `join()` call thread `t1` have not terminated, the main thread will block until thread `t1` terminates. Otherwise, the main thread executes this call without blocking and then terminates.

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

Note that concurrency primitives `join()` and `wait()` in Java have "timed" versions. For example, in addition to method `join()`, threads have method `join(long msec)`. If a thread `t1` calls this method for thread `t2`, thread `t1` waits for thread `t2` to terminate, but not for more than `msec` milliseconds. If thread `t2` is still active after this time expires, thread `t1` stops waiting and continues its execution. The "timed" version of method `wait()` operates according to similar semantics. In this work we do not describe the support necessary for these "timed" versions of `join()` and `wait()`. Since in our model there is no explicit notion of time, the difference between "timed" and "untimed" versions of these synchronization primitives is that the threads executing the timed versions of `join()` and `wait()` have an option of not waiting for thread termination or a notification call respectively. Most aspects of the algorithm presented in this work can be easily modified to take the possibility of the "timed" synchronizations into account. In the interest of simplicity, we do not do this here.

2.2 Program Model

In this section we present and discuss construction of the program model, *Parallel Execution Graph (PEG)*. The PEG for a concurrent Java program is built by combining control flow graphs (CFGs) for all threads that may be started in this program with special kinds of edges.

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.

At present we inline all called methods into 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 of the form $(\texttt{object}, \texttt{name}, \texttt{caller})$, 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 to label nodes that do not correspond to method calls by replacing the `object` part of the label with 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 $(*, \texttt{end}, \texttt{t})$. A node representing an `if` branch that occurs in thread `t` is labeled $(*, \texttt{if}, \texttt{t})$.

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. In general, 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 be able to reason about

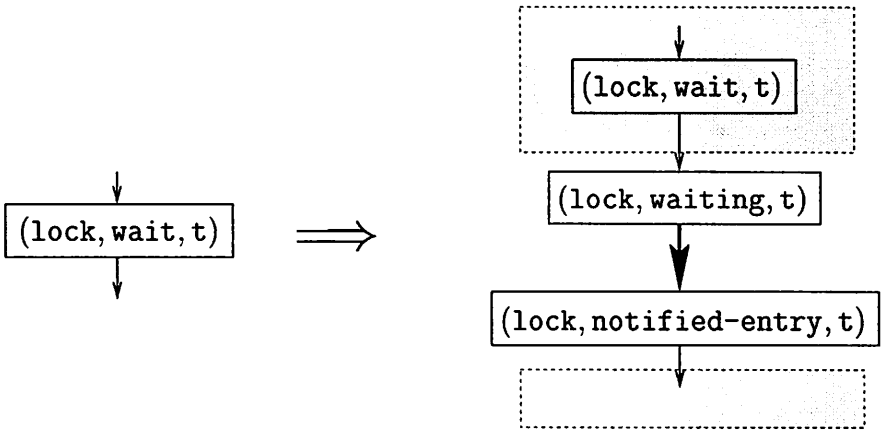

Figure 2: CFG transformation for `wait()` method calls

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 (`lock`, `wait`, `t`) represents the execution of the `wait()` method, the node labeled (`lock`, `waiting`, `t`) represents the thread being idle while waiting for a notification, and the node labeled (`lock`, `notified-entry`, `t`) represents the thread after it received a notification and is in the process of trying to obtain the lock to re-enter the `synchronized` block. The shaded regions in the figure represent the `synchronized` block. Note that the thick edge between the `waiting` and `notified-entry` nodes in Figure 2 does not represent the normal transfer of control between two nodes. The reason for this is that when a thread executes the `wait()` method of a lock object, the moment when this thread is awakened does not depend on this thread, but rather on some other thread that executes the `notify()` or `notifyAll()` method of this lock object. Thus, execution of a `notified-entry` node depends not only on its `waiting` predecessor by also on the node representing such a call to `notify()` or `notifyAll()`.

The CFGs for all threads in the program are combined in a PEG by adding special kinds of edges between nodes from different CFGs. These edges are used by our algorithm for capturing certain dependencies between thread synchronization statements in different threads. We define the following edge kinds.

local A local edge is non-waiting edge between two nodes from the same CFG. For example, the edge between nodes labeled (`lock`, `wait`, `t`) and (`lock`, `waiting`, `t`) is a local edge.

waiting A waiting edge is created between `waiting` and `notified-entry` nodes as shown in Figure 2.

notify A notify edge is created from a node m to a node n if m is labeled (`o`, `notify`, `r`) or (`o`, `notifyAll`, `r`) and n is labeled (`o`, `notified-entry`, `s`) for some thread object `o`, 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 the information about which nodes may happen in parallel may suppress addition of such edges in some cases, which restricts the amount of information that may propagate into `notified-entry` nodes from `notify` and `notifyAll` nodes.

start A start edge is created from a node m to a node n if m is labeled with (`t`, `start`, $*$) and n is labeled with ($*$, `begin`, `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 such edges can be computed by syntactically matching node labels.

Note that, unlike for `notify-all` nodes, no edges from other threads can enter `join` nodes. Despite the similarity between a thread waiting for a notification from another thread and a thread waiting for another thread to terminate, there is a significant difference. When a thread executes the `wait()` method of a lock

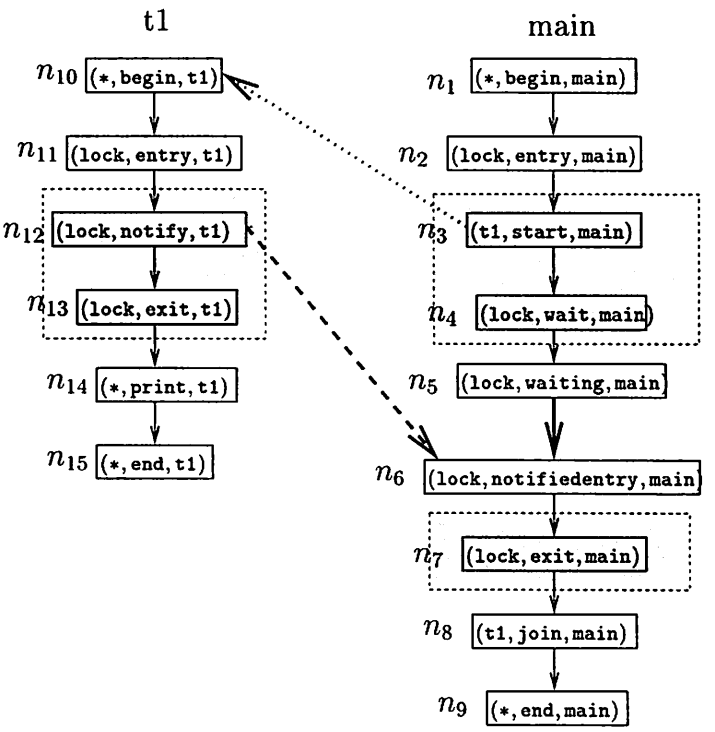

Figure 3: PEG example

object, it always has to wait until some other thread executes the `notify()` or `notifyAll()` method for this lock object. But when a thread executes the `join()` method of another thread `t`, it only has to wait if `t` is in fact executing. If thread `t` either has not been started yet or has already terminated, no waiting is necessary. Thus, our proposed algorithm handles these two thread communication mechanisms differently, not requiring additional edges for `join` nodes.

Figure 3 shows the PEG for the program in Figure 1. The shaded regions include nodes in the monitor of the program, where thin solid edges represent control flow within individual threads, the thick solid edge is a waiting edge, the dotted edge is a start edge, and the dashed edge is a notify edge. Note that this notify edge is not present in the PEG originally but will be created during the *MHP* algorithm.

For convenience, we associate a number of functions that map each node to the set of this node's predecessors of a specific edge kind:

- *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.
- *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 PEG to the thread to which this node belongs. For example, for the PEG in Figure 3, $thread(n_5) =$ `main` and $thread(n_9) =$ `t1`.

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*(lock) = $\{n_4\}$ and *waitingNodes*(lock) = $\{n_{11}\}$.

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 the object obj and let $Monitor_{\texttt{obj}}(t)$ denote the part of the monitor executed by tread t: $Monitor_{\texttt{obj}}(t) = Monitor_{\texttt{obj}} \cap N(t)$. For the example in Figure 3, $Monitor_{\texttt{lock}} = \{n_4, n_5, n_{10}, n_{13}\}$ and $Monitor_{\texttt{lock}}(main) = \{n_4, n_5\}$.

3 The *MHP* algorithm

3.1 High Level Overview

In [13] we introduced an *MHP* algorithm for computing pairs of statements that may happen in parallel in programs with the rendezvous model of concurrency. The general structure of the algorithm presented here is similar. Initially we assume for each node in the PEG that this node may happen in parallel with no 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, the computed information represents, for each node, a conservative overapproximation of all nodes that may happen in parallel with it.

An interesting feature of the proposed *MHP* algorithm for Java is that in addition to the information about which nodes may happen in parallel, it also dynamically computes some control information. The *MHP* information computed by the algorithm is used to compute notify edges. Subsequently, these notify edges can be used for propagating the *MHP* information. Using this approach allows us to compute the *MHP* information more precisely.

To each node n of the PEG we assign a set $M(n)$ which contains 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 the *MHP* information to be propagated to the successors of the node. The *OUT* set for a node is computed by adding some nodes to and/or removing some nodes from the M set for this node. The reason for distinguishing between the $M(n)$ and $OUT(n)$ 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.

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, checking for each such node if it is reachable from the begin node of this main thread. The reasoning for this is that places in the main thread of the program where new threads are started are places where new parallelism is initiated. Our algorithm generates nodes to be put in *OUT* sets of such start nodes. 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

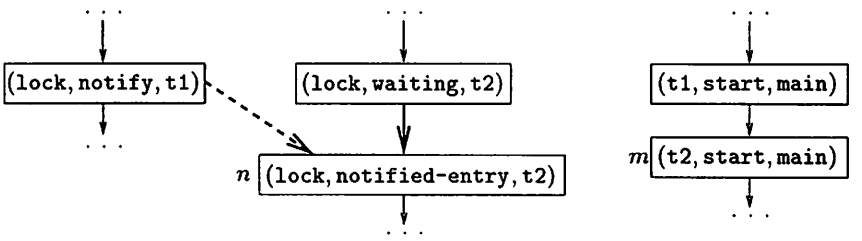

Figure 4: Example of a situation when a node m is contained in a notify predecessor but not in the waiting predecessor of a `notified-entry` node

concisely as

$$NotifySucc(n) = \begin{cases} \{m | m \in (\mathtt{o}, \mathtt{notified\text{-}entry}, *) \wedge WaitingPred(m) \in M(n)\}, & \text{if } n \in notifyNodes(\mathtt{o}) \\ \text{undefined}, & \text{otherwise} \end{cases} \quad (1)$$

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 some additional information for nodes with certain labels. Equation (3) gives the rule for computing the M set for nodes with all possible labels.

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[1]. 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 $GEN_{notifyAll}(n)$ set 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 once this happens (after a `notify()` or `notifyAll()` method call), this thread may 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 this `waiting` predecessor represents the state of this thread immediately before it becomes notified. Thus, if one of the notify predecessor of n may happen in parallel with some node m but the `waiting` predecessor of n may not happen in parallel with m, n may not happen in parallel with m. An example is illustrated by the PEG fragment in Figure 4. one of notify predecessors of n may happen in parallel with a node that starts the thread containing n. Such a node will never be contained in the OUT of n's waiting predecessor. Another case where intersecting the union of OUT sets of the notify predecessors of a node with the OUT set of this node's waiting predecessor is if the `waiting` predecessor of n may happen in parallel with some node r but none of n's notify predecessor may happen in parallel with r. This case is illustrated in Figure 5. In this example the waiting predecessor of n may happen in parallel with a `waiting` node r for the same lock object in another thread. The `notifyAll` predecessor p of n wakes up all threads for this lock object and thus node r is not included in $OUT(p)$ (see equations (4) and (6)). Thus, node n correctly does not obtain node r for its M set because the threads containing n and r become notified simultaneously, and so nodes n and r do not execute in parallel.

The $GEN_{notifyAll}$ set in equation (3) handles a special case of a single `notifyAll` statement awaking 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, labeled $(\mathtt{o}, \mathtt{notified\text{-}entry}, *)$ from other threads than that of the current node n, also labeled $(\mathtt{o}, \mathtt{notified\text{-}entry}, *)$. A node m has to

[1] `begin` nodes are included in the OUT sets of the corresponding `start` nodes. See equations (4) and (5).

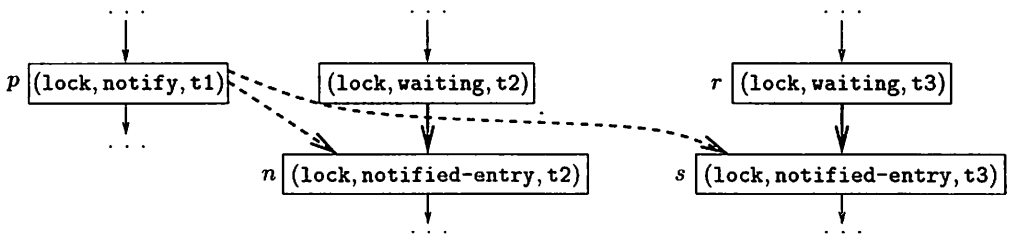

Figure 5: Example of a situation when a node m is contained in in the waiting predecessor but not in a notify predecessor of a `notified-entry` node

be put in $GEN_{notifyAll}(n)$ if m is also labeled $(\mathtt{o}, \mathtt{notified\text{-}entry}, *)$, the *WaitingPred* nodes of m and n may happen in parallel, and there is a node r labeled $(\mathtt{o}, \mathtt{notifyAll}, *)$ that is a notify predecessor of both m and n. For example, in the PEG example in Figure 5 nodes n and s are inserted into $GEN_{notifyAll}$ of each other. Equation (2) formally states this:

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

$$M(n) = M(n) \cup \begin{cases} (\bigcup_{p \in StartPred(n)} OUT(p) \setminus N(thread(n))), & \text{if } n \in (*, \mathtt{begin}, *) \\ ((\bigcup_{p \in NotifyPred(n)} OUT(p)) \cap OUT(WaitingPred(n))) & \\ \quad \cup GEN_{notifyAll}(n), & \text{if } n \in (*, \mathtt{notified\text{-}entry}, *) \\ \bigcup_{p \in LocalPred(n)} OUT(p), & \text{otherwise} \end{cases} \tag{3}$$

3.4 Computing *OUT* Sets

The $OUT(n)$ set represents the *MHP* information that has to be passed to the successors of n. In general, $OUT(n)$ is not equal $M(n)$ because of the semantics of some Java communication methods. Equation (4) gives the general way of computing the *OUT* set for the current node n.

$$OUT(n) = (M(n) \cup GEN(n)) \setminus KILL(n) \tag{4}$$

$GEN(n)$ 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. $KILL(n)$ 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.

$$GEN(n) = \begin{cases} (*, \mathtt{begin}, \mathtt{t}) & , \text{ if } n \in (\mathtt{t}, \mathtt{start}, *) \\ NotifySucc(n), & \text{if } n \in notifyNodes(\mathtt{o}) \\ \emptyset, & \text{otherwise} \end{cases} \tag{5}$$

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

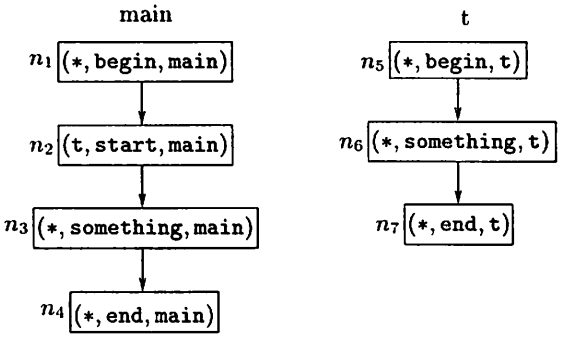

Figure 6: Illustration of the necessity of the symmetry step

of the current node. Note that *while* the current `start` node is executing, thread `r` has not been started yet, and so its `begin` node is not in the M set of the current 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 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 $KILL(n)$ set represents the nodes that must not be passed to the successors of n. The computation of this set depends on the label of n. The following equation gives a rigorous description of this computation.

$$KILL(n) = \begin{cases} N(t), & \text{if } n \in (\texttt{t}, \texttt{join}, *) \\ Monitor_{\texttt{o}}, & \text{if } n \in (\texttt{o}, \texttt{entry}, *) \cup (\texttt{o}, \texttt{notified-entry}, *) \\ waitingNodes(\texttt{o}), & \text{if } (n \in (\texttt{o}, \texttt{notifyAll}, *)) \vee \\ & \quad (n \in (\texttt{o}, \texttt{notify}, *) \wedge \mid waitingNodes(\texttt{o}) \mid = 1) \\ \emptyset & \text{otherwise} \end{cases} \tag{6}$$

If the current node n represents `joining` 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. The semantics of such nodes are such that while the thread is executing in a `entry` or `notified-entry` node, it is not in the monitor entrance to which 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 `o`, this means that once this node completes its execution, no threads in the program will be waiting on this object. Thus, no nodes labeled (`o`, `waiting`, $*$) must be allowed to propagate in 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 that represents a state from which the thread may be woken up by this `notify` 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-flow data flow algorithm. However, after computing M and OUT sets for each node, we have to take an additional step to ensure the symmetry $n_1 \in M(n_2) \Leftrightarrow n_2 \in M(n_1)$ by adding n_2 to $M(n_1)$ if $n_1 \in M(n_2)$. The necessity of this step is illustrated in Figure 6, which shows the PEG for a very simple program consisting of two threads. In this example, nodes n_3 and n_6 represent some non-synchronization statements. Node n_5 is inserted in the $OUT(n_2)$ set according to

equation (4) and later, according to equation (3), n_5 is added to the M set of node n_3. However, unless the symmetry step is performed, according to equation (3), $M(n_5)$ stays empty. As a result, $n_5 \in M(n_3)$ but $n_3 \notin M(n_5)$.

The symmetry step is done after the M and OUT sets for a node are computed. 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: initialization stage that computes *KILL* sets for all nodes, as well as the *GEN* sets for `start` nodes, and iteration stage that computes M and OUT sets and notify edges using a worklist containing all nodes that have to be investigated.

The following algorithm implements the first stage of computing the *MHP* information. All steps of this algorithm correspond to computations described by equations (5) and (6).

Algorithm 1 (Stage I, Initialization). Input:

CFGs for all threads in the program

Output:

$\forall n \in N$: *sets* $KILL(n)$,*and* $GEN(n)$

Initialization:

$$\forall n \in N : KILL(n) = GEN(n) = \emptyset$$

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

The following algorithm implements the second stage of computing the *MHP* information.

Algorithm 2 (Stage II, Base *MHP* Algorithm). 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:

$$\forall n \in N : M(n) = OUT(n) = \emptyset$$

Initialize the worklist W *to include all* `start` *nodes in the* `main` *thread*

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 ∃o : n ∈ notifyNodes(o) then
(7)       ∀m ∈ M(n) ∩ waitingNodes(o):
              // create a new notify edge from node n to the waiting successor of node m
(8)           NotifySucc(n) = NotifySucc(n) ∪ { 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) ≠ NotifySucc(n) then
(10)          W = W ∪ NotifySucc(n)
(11)  Compute the set GEN_notifyAll(n) as in equation (2)
(12)  Compute the set M(n) as in equation (3)
      // 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 ∃o : n ∈ notifyNodes(o) then
(14)      GEN(n) = NotifySucc(n)
(15)  Compute the set OUT(n) as in equation (4)
      // do the symmetry step for all new nodes in M(n):
(16)  if M_old ≠ M(n) then
(17)      ∀m ∈ (M(n) \ M_old(n)):
(18)          M(m) = M(m) ∪ {n}
              // add m to the worklist because the change in M(m) may lead to a change in OUT(m)
(19)          W = W ∪ {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 ≠ OUT(n):
(21)      W = W ∪ (LocalSsucc(n) ∪ StartSucc(n))
```

The following lemma proves that the order in which nodes are added to the worklist by executing lines (19) and (21) and then taken from the worklist does not affect the final values of the M sets computed by the algorithm for each node in the PEG.

Lemma 1. *The result of running Algorithm 2 on a PEG does not depend on the order in which nodes are placed on and removed from the worklist.*

Proof. Fix two arbitrary orders in which Algorithm 2 puts nodes on and takes nodes off the worklist. We will superscript the sets computed by the run of this algorithm with the first order with 1 and with the second order with 2 in this proof. In addition, we will subscript all sets by the iteration on which they are computed. Thus, $M(n)^1_i$ refers to the M set of node n computed on the i'th iteration of the algorithm with the first order of putting nodes on the worklist.

To prove this lemma, we use the induction by the number of iterations of the main loop of the algorithm with the first order of putting nodes on the worklist to show that any node m found to be in $M(n)^1$ on some iteration is also found to be in $M(n)^2$ on some iteration and, similarly, any node m found to be in $OUT(n)^1$ is also found to be in $OUT(n)^2$. For the inductive step of the proof, we consider all possible ways in which

nodes may be inserted in $M(n)^1$ and $OUT(n)^1$ and for each of these ways we prove, relying on the inductive hypothesis, that the same way is used for inserting the same nodes in $M(n)^2$ and $OUT(n)^2$ respectively.

Since before any iterations of the algorithm $\forall n \in N, M(n) = OUT(n) = \emptyset$, trivially each node in $M(n)_0^1$ is also in $M(n)_0^2$ and each node node in OUT_0^1 is also in $OUT(n)_0^2$.

Suppose that $\forall i : 1 \leq i \leq s, m \in M(n)_i^1 \Rightarrow \exists j, m \in M(n)_j^2$, and, similarly, $\forall i : 1 \leq i \leq s, m \in OUT(n)_i^1 \Rightarrow \exists j, m \in OUT(n)_j^2$.

Suppose that n is the current node on $s+1$'st iteration. First we investigate computing $M(n)_{s+1}^1$. Node m can be inserted in $M(n)_{s+1}^1$ either by symmetry or as computed by equation (3). If m is inserted in $M(n)_{s+1}^1$ by symmetry, this means that $n \in M(m)_s^1$, and so by the induction hypothesis $\exists j, n \in M(m)_j^2$. Hence, the symmetry step will be performed by the algorithm with the second order of the worklist, and so $m \in M(n)_j^2$.

According to equation (3), computation of M sets depends on the node label. Suppose first that $n \in (*, \texttt{begin}, *)$. Then

$$M(n)_{s+1}^1 = M(n)_s^1 \cup \bigcup_{p \in StartPred(n)} OUT(p)_s^1 \setminus N(thread(n))$$

and

$$M(n)_{j+1}^2 = M(n)_j^2 \cup \bigcup_{p \in StartPred(n)} OUT(p)_j^2 \setminus N(thread(n))$$

There are two cases which lead to a node m being inserted in $M(n)_{s+1}^1$. The first case is when $m \in M(n)_s^1$. Then by the induction hypothesis $\exists j, m \in M(n)_j^2$. The second case is when $\exists p \in StartPred(n) : m \in OUT(p)_s^1 \wedge m \notin N(thread(n))$. Again, by the induction hypothesis, $\exists j, m \in OUT(p)_j^2$, and so $m \in M(p)_{j+1}^2$.

If $n \in (*, \texttt{notified-entry}, *)$, then, according to equation (3),

$$M(n)_{s+1}^1 = M(n)_s^1 \cup ((\bigcup_{p \in NotifyPred(n)_{s+1}^1} OUT(p)_s^1) \cap OUT(WaitingPred(n))_s^1) \cup GEN_{notifyAll}(n)_{s+1}^1$$

and

$$M(n)_{j+1}^2 = M(n)_j^2 \cup ((\bigcup_{p \in NotifyPred(n)_{j+1}^2} OUT(p)_j^2) \cap OUT(WaitingPred(n))_j^2) \cup GEN_{notifyAll}(n)_{j+1}^2.$$

First we prove that if $m \in NotifyPred(n)_{s+1}^1$, then $\exists k, m \in NotifyPred(n)_{k+1}^2$. According to equation (1),

$$NotifySucc(n)_{s+1}^1 = \{m | m \in (\texttt{o}, \texttt{notified-entry}, *) \wedge WaitingPred(m) \in M(n)_s^1\}$$

and

$$NotifySucc(n)_{j+1}^2 = \{m | m \in (\texttt{o}, \texttt{notified-entry}, *) \wedge WaitingPred(m) \in M(n)_j^2\}.$$

By the induction hypothesis, since $m \in M(n)_s^1 \Rightarrow \exists j, m \in M(n)_j^2$, it follows that $m \in NotifySucc(n)_{j+1}^2$.

Now we prove that if $m \in GEN_{notifyAll}(n)_{s+1}^1$, $\exists k', m \in GEN_{notifyAll}(n)_{k'}^2$. The computation of $GEN_{notifyAll}$ sets depends only on values of M_s^1 sets (see equation (2)), and so by the induction hypothesis, and because once the M set of a `notify` or `notifyAll` node changes, its `notified-entry` successors are put on the worklist, $\exists k', m \in GEN_{notifyAll}(n)_{k'}^2$.

For all other kinds of nodes $m \in M(n)_{s+1}^1 \Rightarrow \exists j, m \in M(n)_j^2$ easily follows from the induction hypothesis.

Now consider the computation of OUT_{s+1}^1 sets. According to equation (4), m is inserted in $OUT(n)_{s+1}^1$ if m is either in $M(n)_{s+1}^1$ or $GEN(n)_{s+1}^1$ but not in $KILL(n)$. Since $KILL$ sets for all nodes are precomputed and do not change during the algorithm, the effect of these sets on the algorithm runs with both orders of putting nodes on the worklist is the same.

If $m \in M(n)_{s+1}^1 \wedge m \notin KILL(n)$, then, as we have just proved, $\exists j, m \in M(n)_j^2$, and so by equation (4), $m \in OUT(n)_j^2$.

Suppose now that $m \in GEN(n)^1_{s+1}$. Since *GEN* sets for all nodes except `notify` and `notifyAll` nodes are precomputed, we only have to consider `notify` and `notifyAll` nodes. According to equation (5), for each such node the *GEN* set equals the set of its notify successors. In the proof of the part for M sets we have already proved that $m \in NotifySucc(n)^1_{s+1} \Rightarrow \exists j, m \in NotifySucc(n)^2_j$. Thus, for any node $m \in GEN(n)^1_{s+1}$, $\exists j, m \in GEN(n)^2_j$. □

3.7 Example

We illustrate the *MHP* algorithm on the example Java program in Figure 1. The PEG for this example appears in Figure 3. Initially, the notify edge between nodes n_{12} and n_6 is absent from the graph. The first stage of the algorithm computes initial values of the *GEN* and *KILL* sets for all nodes. These sets are empty for all nodes, except for the following cases:

- $GEN(n_3) = \{n_{10}\}$
- $KILL(n_2) = \{n_{12}, n_{13}\}$
- $KILL(n_6) = \{n_{12}, n_{13}\}$
- $KILL(n_{11}) = \{n_3, n_4, n_7\}$
- $KILL(n_{12}) = \{n_5\}$

The table in Figure 7 shows the information computed on each iteration of the *MHP* algorithm for this example. The first column in this table shows the iteration number; the second column gives the ID for the current node (just a number k instead of n_k); the third column lists nodes that were added to the M set of the current node as a result of this iteration; the fourth column lists nodes that were added to the *OUT* set of the current node as the result of this iteration; the fifth column shows the nodes into whose M sets the current node is added by the symmetry step; and the final column shows the worklist after the iteration[2]. Figure 8 shows the PEG for this example with all nodes annotated with their M sets at the termination of the *MHP* algorithm.

4 Termination and Complexity

The following theorem states that the *MHP* algorithm presented in the previous section always terminates after a finite number of iterations.

Theorem 2. *For any Java program, Algorithms 1 and 2 always terminate.*

Proof. The proof for Algorithm 1 is trivial. Since every node in the PEG is examined only once, this algorithm terminates in $\mathcal{O}(|N|)$ steps.

For all $m \in N$, let $M(m)_i$ be the value of $M(m)$ at the end of the i-th iteration of the main loop, (1)-(21) of Algorithm 2. Similarly, let $OUT(m)_i$ be the value of $OUT(m)$ at the end of the i-th iteration. To show that Algorithm 2 always terminates in a finite number of steps, it suffices to show that, for $m \in N$ and all $i \geq 0$, $M(m)_i \subseteq M(m)_{i+1}$ and $OUT(m)_i \subseteq OUT(m)_i$. Since $M(m)_i, OUT(m)_i \subseteq N$ for all i, this shows that the there is some finite k such that the sets $M(m)$ and $OUT(m)$ stabilize in k iterations, i.e., that $M(m)_i = M(m)_{i+1}$ and $OUT(m)_i = OUT(m)_{i+1}$ for all $i \geq k$. Once this happens, the worklist will eventually become empty and the algorithm will terminate.

To show that $M(m)$ and $OUT(m)$ stabilize in a finite number of iterations, we argue by induction. Since the initial values of M and *OUT* sets are empty ($M(m)_0 = OUT(m)_0 = \emptyset$, the claim is clear for $i = 0$. Suppose it holds for all $i \leq s$ and consider iteration $s + 1$.

[2]Note that in this example the position in the worklist to which nodes are added is chosen to minimize the number of iterations. According to Lemma 1, the order in which nodes are put on and taken from the worklist does not affect the outcome of the algorithm.

Iter	Current n	New nodes in $M(n)$	New nodes in $OUT(n)$	Symmetry	Worklist
1	3		10		4, 10
2	4	10	10	10	5, 10
3	5	10	10	10	10
4	10		4, 5		11
5	11	4, 5	4, 5	4, 5	12, 4, 5
6	12	5	6	5	13, 4, 5, 6
7	13	6	6	6	14, 4, 5, 6
8	14	6	6	6	15, 4, 5, 6
9	15	6	6	6	4, 5, 6
10	4		11		5, 6
11	5		11, 12		6
12	6		14, 15		7
13	7	14, 15	14, 15	14, 15	8, 14, 15
14	8	14, 15		14, 15	9, 14, 15
15	9				14, 15
16	14		7, 8		15
17	15		7, 8		

Figure 7: Information computed on the iterations of the *MHP* algorithm for the example in Figure 3

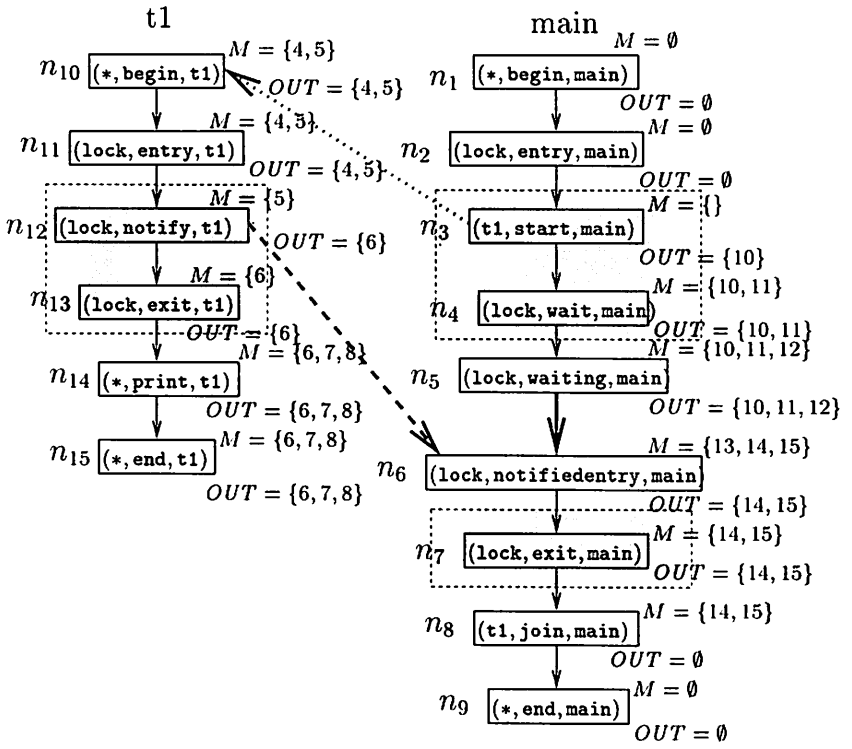

Figure 8: PEG for the example with the *MHP* information added

Let $n = head(W)$ be the current node for this iteration. We distinguish three cases, according to the label of n.

Suppose $n \in (*, \texttt{begin}, *)$. Then line (12) of the algorithm computes $M(n)$ according to equation (3) and we have

$$M(n)_s = (\bigcup_{p \in StartPred(n)} OUT(p)_{s-1}) \setminus N(thread(n))$$

and

$$M(n)_{s+1} = (\bigcup_{p \in StartPred(n)} OUT(p)_s) \setminus N(thread(n))$$

By the induction hypothesis, we know that, for all p, $OUT(p)_{s-1} \subseteq OUT(p)_s$, so clearly $M(n)_s \subseteq M(n)_{s+1}$.

The only way in which this iteration of the algorithm can change the value of any $M(m)$ is by adding n to $M(m)_s$ in lines (16)-(18). It follows that $M(m)_s \subseteq M(m)_{s+1}$ for all $s \in N$.

The set $OUT(n)$ is computed in line (15) of the algorithm, using equation (4). Since $n \in (*, \texttt{begin}, *)$, we know that the sets $GEN(n)$ and $KILL(n)$ do not change, and $OUT(n)_s \subseteq OUT(n)_{s+1}$. This iteration of the algorithm does not change $OUT(m)$ for any $m \neq n$, so we have $OUT(m) \subseteq OUT(m)_{s+1}$ for all $m \in N$.

Now suppose $n \in (*, \texttt{notified-entry}, *)$. According to equation (3),

$$M(n)_s = M(n)_{s-1} \cup ((\bigcup_{p \in NotifyPred(n)_s} OUT(p)_{s-1}) \cap OUT(WaitingPred(n))_{s-1}) \cup GEN_{notifyAll}(n)_s$$

and

$$M(n)_{s+1} = M(n)_s \cup ((\bigcup_{p \in NotifyPred(n)_{s+1}} OUT(p)_s) \cap OUT(WaitingPred(n))_s) \cup GEN_{notifyAll}(n)_{s+1}$$

By the induction hypothesis, we know that, for all p, $OUT(p)_{s-1} \subseteq OUT(p)_s$ and $M(p)_{s-1} \subseteq M(p)_s$. Thus, to prove that $M(n)_s \subseteq M(n)_{s+1}$, we have to prove that (1) $GEN_{notifyAll}(n)_s \subseteq GEN_{notifyAll}(n)_{s+1}$ and (2) $NotifyPred(n)_s \subseteq NotifyPred(n)_{s+1}$. Consider (1). According to equation (2),

$$\begin{aligned} GEN_{notifyAll}(n)_s = \{m | \exists o : m \in (o, \texttt{notified-entry}, *) \wedge WaitingPred(n) \in M(WaitingPred(m))_{s-1} \wedge \\ (\exists r \in N : r \in (o, \texttt{notifyAll}, *) \wedge r \in (M(WaitingPred(m))_{s-1} \cap M(WaitingPred(n))_{s-1}))\} \end{aligned}$$

and

$$\begin{aligned} GEN_{notifyAll}(n)_{s+1} = \{m | \exists o : m \in (o, \texttt{notified-entry}, *) \wedge WaitingPred(n) \in M(WaitingPred(m))_s \wedge \\ (\exists r \in N : r \in (o, \texttt{notifyAll}, *) \wedge r \in (M(WaitingPred(m))_s \cap M(WaitingPred(n))_s))\} \end{aligned}$$

Since by the induction hypothesis, for any node p, $M(p)_{s-1} \subseteq M(p)_s$, (1) follows.

Now consider (2). According to equation (1),

$$NotifySucc(n)_s = \{m | m \in (o, \texttt{notified-entry}, *) \wedge WaitingPred(m) \in M(n)_{s-1}\}$$

and

$$NotifySucc(n)_{s+1} = \{m | m \in (o, \texttt{notified-entry}, *) \wedge WaitingPred(m) \in M(n)_s\}$$

Again, by the induction hypothesis, (2) follows. Thus, we proved that $M(n)_s \subseteq M(n)_{s+1}$ for `notified-entry` nodes. The proof of $OUT(n)_s \subseteq OUT(n)_{s+1}$ is identical to the proof of this statement for `begin` nodes.

Finally, consider the case where n is any node except a `begin` or `notified-entry` node. Then, according to equation (3),

$$M(n)_s = \bigcup_{p \in LocalPred(n)} OUT(p)_{s-1}$$

and

$$M(n)_{s+1} = \bigcup_{p \in LocalPred(n)} OUT(p)_s$$

$M(n)_s \subseteq M(n)_{s+1}$ follows from the induction hypothesis of $OUT(n)_{s-1} \subseteq OUT(n)_s$.

According to equation (4),

$$OUT(n)_s = (M(n)_s \cup GEN(n)_s) \setminus KILL(n)_s$$

and

$$OUT(n)_{s+1} = (M(n)_{s+1} \cup GEN(n)_{s+1}) \setminus KILL(n)_{s+1}$$

The *KILL* sets are constant for all nodes in the TFG, so $KILL(n)_s = KILL(n)_{s+1}$. For all nodes except `notify` and `notifyAll` nodes, $GEN(n)_s = GEN(n)_{s+1}$. For `notify` and `notifyAll` nodes, according to equation (5), $GEN(n)_s = NotifySucc(n)_s$ and $GEN(n)_{s+1} = NotifySucc(n)_{s+1}$, and we already proved in the previous case that $NotifySucc(n)_s \subseteq NotifySucc(n)_{s+1}$. Thus, $OUT(n)_s \subseteq OUT(n)_{s+1}$. □

Before we make a statement about the complexity of our *MHP* algorithm, we introduce an efficient form of the worklist version of this algorithm. This optimized algorithm limits the amount of information computed and passed among the nodes in the PEG by sending each node from the *OUT* set of a given node to each of its successors only once. In addition, instead of completely recomputing the $GEN_{notifyAll}$ set for `notified-entry` nodes, a `notified-entry` node may be added to the $GEN_{notifyAll}$ node of another `notified-entry` node only once. This is done by associating an additional set *notifiedPartners* with each `notified-entry` node n. Another `notified-entry` node m is inserted in this set if these two nodes are in different threads, m has never been inserted in $notifiedPartners(n)$ before, and m and n have a common notify predecessor. After node m appears in $notifiedPartners(n)$, it is copied in $GEN_{notifyAll}(n)$ if it is detected that the waiting predecessors of m and n appear in the M sets of each other. This two-step computation of $GEN_{notifyAll}$ sets implements equation (2). In interests of efficiency, when a node m is copied from $notifiedPartners(n)$ to $GEN_{notifyAll}(n)$, it is removed from $notifiedPartners(n)$. Combined with the fact that a node is added to $notifiedPartners(n)$ at most once over the run of the algorithm, this ensures that a node is added to $GEN_{notifyAll}(n)$ at most once over the run of the algorithm.

The *GEN* sets for `notify` and `notifyAll` nodes are managed in a way similar to the way the *OUT* sets for all nodes are managed: a node m can be added to $GEN(n)$ at most once. This is done by inserting a `notified-entry` node m in the *GEN* set of a `notify` or `notifyAll` node n only at the point where it is first determined that n is a notify predecessor of m. After $GEN(n)$ is updated in such a manner, it is used for computing $OUT(n)$ and then is set to be an empty set at the end of the iteration of the main loop. This ensures that a node that appears in $GEN(n)$ is inserted in $OUT(n)$ exactly once.

Another new set used in this algorithm is set $M_{new}(n)$, computed for each node n in the PEG. This set is used to store nodes that were determined to be able to happen in parallel with n since the last time n was the current node. Nodes are added to $M_{new}(n)$ both by symmetry and while n is the current node of the main loop of the algorithm. Using the set $M_{new}(n)$ instead of $M(n)$ in many cases allows us to reduce the number of nodes used in several of the set operations computed by the algorithm. At the end of each iteration of the main loop of the algorithm, nodes from the M_{new} set of the current node are copied in the M set of this node and then this M_{new} set becomes empty.

Finally, because a node m is added to the *OUT* set of another node n at most once and the next time n is the current node m is removed from $OUT(n)$, when a new notify successor m is computed for a `notify` or `notifyAll` node n, an additional step should be taken to propagate all nodes that have appeared in $OUT(n)$

from the beginning of the run of the algorithm in $M_{new}(m)$, and not only the nodes that are currently in $OUT(n)$.

After presenting a rigorous pseudocode description of the efficient algorithm, we show that this algorithm computes exactly the same information as Algorithm 2 and prove that the complexity of the efficient algorithm is $\mathcal{O}(|N|^3)$.

Algorithm 3 (Stage II, Efficient *MHP* Algorithm). 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* $\forall m \in M$: m *may happen in parallel with* n, *and* $\forall \overline{m} \notin M(n) : \overline{m}$ *may never happen in parallel with* n.

Additional Information:

W *is the worklist containing nodes to be processed. It is implemented as a FIFO buffer*
$\forall n \in N, OUT(n)$ *is the set of nodes to be propagated to the successors of* n
$\forall n \in (*, \texttt{notified-entry}, *)$, $notifiedPartners(n)$ *is the set of candidates for inclusion in set* $GEN_{notifyAll}(n)$
$\forall n \in (*, \texttt{notified-entry}, *)$, $M_{Wpred}(n)$ *is the set of nodes in the* M *set of the corresponding* `waiting` *node during the previous iteration with* n *as the current node*
$\forall n \in N, M_{new}(n)$ *is the set of nodes that may happen in parallel with* n, *discovered since the previous iteration with* n *as the current node*

Initialization:

$$\forall n \in N : M(n) = M_{new}(n) = OUT(n) = OUT_{full}(n) = \emptyset$$

$$\forall n \in (*, \texttt{notified-entry}, *), M_{Wpred}(n) = notifiedPartners(n) = \emptyset$$

Initialize the worklist W *to include all* `start` *nodes in the* `main` *thread*

Main loop:

We evaluate the following statements repeatedly until $W = \emptyset$

(1) $n = head(W)$
(2) $W = tail(W)$
(3) $GEN_{notifyAll}(n) = \emptyset$
// If n is a `notified-entry` node, check if the M set of its `waiting` predecessor changed
// since the last time n was the current node:
(4) if $n \in (*, \texttt{notified-entry}, *)$ and $M_{Wpred}(n) \neq M(WaitingPred(n))$ then
// For all nodes in the *notifiedPartners* set of n, check if their `waiting` predecessors
// are in the M set of n's `waiting` predecessor; add each such node to $GEN_{notifyAll}(n)$
// and remove it from the *notifiedPartners* set of n:
(5) $\forall m \in notifiedPartners(n)$:
(6) if $WaitingPred(m) \in M(WaitingPred(n))$ then
(7) $GEN_{notifyAll}(n) = GEN_{notifyAll}(n) \cup \{m\}$
(8) $notifiedPartners(n) = notifiedPartners(n) \setminus \{m\}$
// Remember what the M set of the `waiting` predecessor of n contained on this iteration:
(9) $M_{Wpred}(n) = M(WaitingPred(n))$
(10) if $n \in (*, \texttt{notified-entry}, *)$ then
(11) Compute the set $M_{new}(n)$ as in equation (3)
else

(12) Compute the set $M_{new}(n)$ as in equation (3), except use $OUT_{full}(WaitingPred(n))$ instead of $OUT(WaitingPred(n))$

// Remove from $M_{new}(n)$ all nodes that previously appeared there:

(13) $M_{new}(n) = M_{new}(n) \setminus M(n)$

(14) if $\exists o : n \in notifyNodes(o)$ then

Examine all `waiting` nodes in $M_{new}(n)$:

(15) $\forall m \in (M_{new}(n) \cap waitingNodes(o))$:

// If n is a `notifyAll` node, insert the `notified-entry` successor of node m in the // *notifiedPartners* sets of those notify successors of n that are not in the same // thread with m only if it was not inserted there before:

(16) if $n \in (o, \texttt{notifyAll}, *)$ then

(17) $\forall v \in NotifySucc(n)$:

(18) if $WaitingSucc(m)$ was never inserted in $notifiedPartners(v)$ before and $thread(m) \neq thread(v)$, then

(19) $notifiedPartners(v) = notifiedPartners(v) \cup \{WaitingSucc(m)\}$

// Make the `waiting` successor of node m a notify successor of the current node // and also insert it in the *GEN* set of the current node:

(20) $NotifySucc(n) = NotifySucc(n) \cup \{WaitingSucc(m)\}$

(21) $GEN(n) = GEN(n) \cup \{WaitingSucc(m)\}$

// Also, recompute the *KILL* set of the current node and insert it in the M_{new} set for the new notify successor

(22) $M_{new}(WaitingSucc(m)) = M_{new}(WaitingSucc(m)) \cup OUT_{full}$

(23) $OUT(n) = (M_{new}(n) \cup GEN(n)) \setminus KILL(n)$

// Do the symmetry step for all nodes in $M_{new}(n)$ and also put all these nodes on the worklist:

(24) $\forall m \in M_{new}(n)$:

(25) $M_{new}(m) = M_{new}(m) \cup \{n\}$

(26) $W = W \cup \{m\}$

// If new nodes has been added to the OUT set of n, add all n's successors to the worklist

(27) if $OUT(n) \neq \emptyset$ then

(28) $W = W \cup (LocalSucc(n) \cup NotifySucc(n) \cup StartSucc(n))$

// Copy all nodes from $M_{new}(n)$ to $M(n)$ and make the *GEN* and M_{new} sets of n empty:

(29) $GEN(n) = \emptyset$

(30) $M(n) = M(n) \cup M_{new}(n)$

(31) $M_{new}(n) = \emptyset$

(32) $OUT_{full}(n) = OUT_{full}(n) \cup OUT(n)$

The following theorem proves that this efficient algorithm computes precisely the same *MHP* information for all nodes in the PEG as that computed by the base algorithm 2. Since both algorithms accumulate the nodes that may happen in parallel with any given node n in a set $M(n)$, this theorem proves that for every node n in the PEG, after both algorithms terminate, the M sets for n computed by the two algorithms are the same.

Theorem 3. *Algorithms 2 and 3 compute identical M information for all nodes of the given PEG graph.*

Proof. There are a number of differences between the two algorithms, the most significant being that the efficient algorithm never puts a node in the *OUT* set of any node more than once. This restricts the amount of information that is propagated among nodes. This change necessitates the implementation of the worklist as a FIFO in the efficient algorithm, so that all successors of the current node are taken from the worklist before the current node is taken from the worklist again.

Since, according to Lemma 1, the order in which nodes are put on the worklist and taken from the worklist does not matter, the second difference alone cannot induce different results being produced by the two algorithms.

In this proof, we superscript the sets computed by algorithm 2 with ^{base} and superscript the sets computed by algorithm 3 with ^{eff}. We subscript these sets with the number of the iteration of the main loop of the

corresponding algorithm on which they were computed. For example, $OUT(n)_j^{eff}$ refers to the OUT set of *node* n, computed on the j's iteration of the efficient algorithm 3.

First we prove that for each node in the TFG the efficient algorithm computes a superset of the M set that the base algorithm computes for this node. For this, we prove the following two statements that, when combined, result in a stronger statement.

$$\forall i, \forall m, n \in N, m \in M(n)_i^{base} \Rightarrow \exists j, m \in M(n)_j^{eff} \tag{7}$$

$$\forall i, \forall m, n \in N, m \in OUT(n)_i^{base} \Rightarrow \exists j, m \in OUT(n)_j^{eff} \tag{8}$$

We prove both statements at the same time using induction on the number of iterations of the base algorithm. For the inductive step of the proof, we reason about all possible ways in which a node m may propagate in the M set of node n by the base algorithm. For each of these ways we show that a mechanism exists in the efficient algorithm that puts m in $M(n)$. Simultaneously, the same is proved for the OUT sets for the nodes in the PEG.

For the base case of induction, trivially $\forall n \in N, M(n)_0^{base} = M(n)_0^{eff} = OUT(n)_0^{base} = OUT(n)_0^{eff} = \emptyset$. By the way of the induction hypothesis, we suppose that both equations (7) and (8) hold for all $i \leq s$. Consider iteration $s+1$.

Let $n = head(W)$ be the current node for this iteration. Suppose that in the base algorithm m is inserted in $M(n)_{s+1}^{base}$ for the first time on this iteration. Suppose first that m is inserted in $M(n)_{s+1}^{base}$ by symmetry. Then it must be the case that $\exists i \leq s : n \in M(m)_i^{base}$. By the induction hypothesis, $\exists j : n \in M(m)_j^{eff}$, and so m is inserted in $M(n)_{new_j}^{eff}$ by symmetry, from which it gets in $M(n)_j^{eff}$.

If m is not inserted in $M(n)_{s+1}^{base}$ by symmetry, we consider several cases based on the label of n. Suppose first that $n \in (*, \texttt{begin}, *)$. According to equation (3),

$$M(n)_{s+1}^{base} = M(n)_s^{base} \cup (\bigcup_{p \in StartPred(n)} OUT(p)_s^{base} \setminus N(thread(n)))$$

By the induction hypothesis, $\exists j, m \in OUT(p)_j^{eff}$. At this iteration j of the efficient algorithm, since $OUT(p)^{eff}$ changes, all successors of p, including n, are put on the worklist and, since the worklist is processed in FIFO order, n will be taken from the worklist before p is taken from the worklist the next time. This means that $\exists j' > j, m \in M_{new}(n)_{j'}^{eff}$, and so $m \in M(n)_{j'}^{eff}$.

Suppose now that $n \in (*, \texttt{notified-entry}, *)$. According to equation (3),

$$M(n)_{s+1}^{base} = M(n)_s^{base} \cup ((\bigcup_{p \in NotifyPred(n)_{s+1}^{base}} OUT(p)_s^{base}) \cap OUT(WaitingPred(n))_s^{base}) \cup GEN_{notifyAll}(n)_{s+1}^{base}$$

and according to the efficient algorithm,

$$M_{new}(n)_{s+1}^{eff} = M_{new}(n)_s^{eff} \cup ((\bigcup_{p \in NotifyPred(n)_{s+1}^{eff}} OUT(p)_s^{eff}) \cap OUT_{full}(WaitingPred(n))_s^{eff}) \cup GEN_{notifyAll}(n)_{s+1}^{eff}$$

Consider some node $m \in OUT(p)_s^{base}$, where $p \in NotifyPred(n)_{s+1}^{base}$. Since the base algorithm determines that n is a notify successor of p on some iteration prior to $s+1$. In computing notify successors of a node, both the base and the efficient algorithms use the same procedure, utilizing M sets computed so far for certain nodes in the graph. By the induction hypothesis, we can conclude that there exists an iteration j of the main loop of the efficient algorithm, on which n is made a notify successor of p. Suppose first that $\exists j' \leq j, m \in OUT(p)_{j'}^{eff}$. Then m will be inserted in $M_{new}(n)_j^{eff}$ in line (20) of the efficient algorithm. If, on the other hand, $\forall j' \leq j, m \notin OUT(p)_{j'}^{eff}$, then, by induction hypothesis, $\exists j'' > j, m \in OUT(p)_{j''}^{eff}$, and hence

all successors of p, including n, are put on the worklist. Since the worklist has a FIFO order, n will be taken *from* the worklist before p is taken from the worklist the next time, and thus m will propagate in $M_{new}(n)^{eff}$, since by the induction hypothesis, if $m \in OUT(WaitingPred(n))_s^{base}$, then $\exists k, m \in OUT(WaitingPred(n))_k^{eff}$ and hence $m \in OUT_{full}(WaitingPred(n))_k^{eff}$. Then, for $k' = \max(j'', k)$, $OUT_{full}(p)_{k'}^{eff}$ contains m.

Suppose now that $m \in GEN_{notifyAll}(n)_{s+1}^{base}$ According to equation (2),

$$GEN_{notifyAll}(n)_{s+1}^{base} = \{m | \exists o : m \in (o, \texttt{notified-entry}, *) \wedge WaitingPred(n) \in M(WaitingPred(m))_s^{base} \wedge (\exists r \in N : r \in (o, \texttt{notifyAll}, *) \wedge r \in (M(WaitingPred(m))_s^{base} \cap M(WaitingPred(n))_s^{base}))\}$$

By the induction hypothesis,

$$\exists j, WaitingPred(n) \in M(WaitingPred(m))_j^{eff} \wedge r \in M(WaitingPred(m))_j^{eff} \wedge r \in M(WaitingPred(n))_j^{eff}. \quad (9)$$

Thus, we only have to prove that the efficient algorithm performs all these checks. Note that the efficient algorithm associates a *notifiedPartners* set with each `notified-entry` node n. A node v is added to this set if it is determined that both v and n are notify successors of the same `notifyAll` node, which is equivalent to checking equation (9).

Now consider the computation of OUT sets. According to equation (4),

$$OUT(n)_{s+1}^{base} = (M(n)_{s+1}^{base} \cup GEN(n)_{s+1}^{base}) \setminus KILL(n)_{s+1}^{base}.$$

According to the efficient algorithm,

$$OUT(n)_j^{eff} = (M_{new}(n)_j^{eff} \cup GEN(n)_j^{eff}) \setminus KILL(n)_j^{eff}.$$

Since the $KILL$ sets are precomputed and never changed in the main loop of the algorithm, they are the same for both algorithms. Suppose now that $m \in M(n)_{s+1}^{base}$. As we have just proved, $\exists j, m \in M_{new}(n)_j^{eff}$, and so $m \in OUT(n)_j^{eff}$. Suppose that $m \in GEN(n)_{s+1}^{base}$. If n is not a `notify` or `notifyAll` node, then its GEN set is constant and the same in both the base and the efficient algorithm. If n is a `notify` or `notifyAll` node, then $GEN(n)_{s+1}^{base} = NotifySucc(n)_{s+1}^{base}$. In the part of the proof for M sets we already proved that the efficient algorithm computes the same notify edges as the base algorithm. Thus, they compute the same GEN sets.

Showing that the MHP information computed by the efficient algorithm is also computed by the base algorithm 2 is similar. □

The following theorem states and proves the worst-case time bounds on the efficient version of the MHP algorithm.

Theorem 4 (Polynomial-Time Boundedness). *The worst-case time bound for computing MHP sets for all nodes in the PEG is* $\mathcal{O}(|N|^3)$.

Proof. In this proof we assume that all sets are implemented as lookup tables and the worklist is a linked list. Instead of multiplying the maximal possible number of iterations of the main loop by the complexity of the operations performed on each iteration, this proof reasons about each of the operations that may be performed on an iteration of the main loop, computing the complexity of this operation over all iterations of the main loop.

The efficient algorithm puts a node on the worklist if either its M_{new} set changes by symmetry or the OUT set of one of its predecessors changes. Since the M set of a node can have $\mathcal{O}(|N|)$ nodes, the maximal number of times a node can be put on the worklist because its M set is updated by symmetry is $\mathcal{O}(|N|)$. Because each node has $\mathcal{O}(|N|)$ predecessors and the OUT set of each of its predecessors can change $\mathcal{O}(|N|)$ times, the total number of iterations of the main loop is $\mathcal{O}(|N|^3)$. This bound equals the upper bound on the running time of this algorithm and so we have to consider all steps in the main loop except those taking

constant time (lines 1–3, 26, 28). For the rest of the computations in the main loop we have to use amortized analysis, estimating their complexity over all iterations of the algorithm.

Consider computations in lines 4–9. Computing the condition in line 4 takes constant time, assuming that checking equality of two M sets does not require examining these sets element by element[3]. Since lines 5–9 are computed only if the M set of the unique `waiting` node corresponding to the current `notified-entry` node changes, these lines have to be computed $\mathcal{O}(|N|)$ times for each of the $\mathcal{O}(|N|)$ `notified-entry` nodes in the graph. Computing the loop in lines 5–8 takes $\mathcal{O}(|N|)$, and thus over the complete run of the efficient algorithm computation of code in lines 4–9 takes $\mathcal{O}(|N|^3)$.

The M_{new} set for the current node is computed according to equation (3), where $M(n)$ is replaced with $M_{new}(n)$. The common step for each of the three cases in this equation is to add nodes from the OUT sets of all predecessors of n. Since any node $m \in N$ can appear in the OUT set of a node only once and each node has $\mathcal{O}(|N|)$ predecessors, adding a single node from one of n's predecessors to $M_{new}(n)$ happens $\mathcal{O}(|N|^2)$ times for each of N nodes in the graph. For `begin` nodes there is an additional step of checking that the node being added is not in $N(thread(n))$. This takes constant time for each added node and thus does not change the overall $\mathcal{O}(|N|^3)$ bound. Similarly, for a `notified-entry` node n there is the step of finding the intersection of the union of the OUT sets of n's notify predecessors with the OUT_{full} set of the n's waiting predecessor. For each node in this union checking whether it should be placed is in the intersection takes constant time. Thus, the overall $\mathcal{O}(|N|^3)$ bound is not changed.

Line 13 takes nodes that are already in $M(n)$ from $M_{new}(n)$. Since each of the N nodes in the PEG can appear in $M_{new}(n)$ $\mathcal{O}(|N|)$ times, over the course of the algorithm this check takes $\mathcal{O}(|N|^2)$ for each of the N nodes.

For each of the $\mathcal{O}(|N|)$ `notify` and `notifyAll` nodes, the condition in line 15 is checked $\mathcal{O}(|N|)$ times, since this is the number of times a new node can appear in $M_{new}(n)$. The fact that at this point in the algorithm a node can appear in $M_{new}(n)$ only on a single iteration of the main loop ensures that the loop in lines 15–22 is executed $\mathcal{O}(|N|)$ times for each of the $\mathcal{O}(|N|)$ `notify` and `notifyAll` nodes. This means that over the course of the algorithm, for each such node, loop in lines 17–19 is executed $\mathcal{O}(|N|)$ times for each pair (n, m), where n is a `notify` or `notifyAll` node and m is a `waiting` node for the same lock object. Each iteration of this loop (lines 18–19) takes constant time. Computation of lines 20 and 21 also takes constant time. Line 22 adds the OUT_{full} set of the current node to the M_{new} set of its new notify successor. Since this is done only once for each notify edge created during the algorithm, over the course of the algorithm it can be done $\mathcal{O}(|N|^2)$ times. Each time the operation takes $\mathcal{O}(|N|)$. Thus, over the course of the algorithm, for each `notify` and `notifyAll` node the computation of lines 14–22 is $\mathcal{O}(|N|^2)$.

For each node n in the PEG, the computation of $OUT(n)$ in line 23 is necessary only if either $M_{new}(n)$ or $GEN(n)$ changes, which happens $\mathcal{O}(|N|)$. Similarly, for each node n, the computations in lines 24–26 take $\mathcal{O}(|N|)$ over the whole run of the algorithm.

Since the set $OUT(n)$ can become non-empty $\mathcal{O}(|N|)$ times over the course of the algorithm, computations in lines 27–28 take $\mathcal{O}(|N|^2)$ for each node in the graph.

Finally, over the course of the algorithm, computing lines 30 and 32 takes $\mathcal{O}(|N|)$ for each node in the graph.

Combining the complexities for all parts of the main loop, we obtain the complexity claimed in the statement of this theorem. □

5 Conservativeness

To prove that the proposed *MHP* algorithm is conservative, we define a reachability analysis for computing the *MHP* information and then prove that the information computed by the *MHP* algorithm is not "better" than that computed by the reachability approach. Formally, we prove that whenever the reachability approach determines that two nodes may happen in parallel, the *MHP* algorithm determines this as well.

[3]For example, a unique integer ID can be associated with each set. This ID should change when new nodes are added to the set but should be copied when the set is copied. Then after the assignment in line 9, sets $M_{Wpred}(n)$ and $M(WaitingPred(n))$ have the same ID. If the comparison in line 4 indicates that these two nodes have different IDs, this means that new nodes have been added to $M(WaitingPred(n))$ since the last time n was the current node.

For the reachability algorithm, we will characterize states of the program with tuples in which each of the components represents the state of a single thread. A state of the thread is uniquely given by the node in this thread that is currently executing and the following two symbols:

B — indicates that this thread is in a "before execution" state, that is, this thread has not been started yet, and

D — indicates that this thread is in a "dead" state, that is, it has terminated its execution.

We call these tuples *markings* and refer to them using lower-case Greek letters.

We use Δ to denote the transition function for markings. This function operates on a marking is associated with a PEG node present in this marking. The transition function describes the set of markings that can be obtained as a result of executing this node in the given marking. One or more threads may change their state as a result of each transition. Let $\mu = (n_1, n_2, ..., n_{|T|})$ be the current marking and let n denote the identity of one of the nodes n_k, where $1 \leq k \leq |T|$. The transition $\Delta_n(\mu)$ depends on the label of node n as is defined as follows:

$n \in (\mathtt{t}_i, \mathtt{start}, \mathtt{t}_k) \Rightarrow$

$$\Delta_n(\mu) = \begin{cases} \{(n_1, ..., n'_k, ..., n'_i, ..., n_{|T|}) \mid n'_k \in \mathit{LocalSucc}(n_k), \mathit{label}(n'_i) = (*, \mathtt{begin}, \mathtt{t}_i)\}, & \text{if } n_i = B_i \\ \{\mu\}, & \text{if } n_i = D_i \\ \emptyset, & \text{otherwise} \end{cases} \tag{10}$$

$n \in (\mathtt{t}_i, \mathtt{join}, \mathtt{t}_k) \Rightarrow$

$$\Delta_n(\mu) = \begin{cases} \{(n_1, ..., n'_k, ..., D_i, ..., n_{|T|}) \mid n'_k \in \mathit{LocalSucc}(n_k)\}, & \text{if } n_i = D_i \\ \emptyset, & \text{otherwise} \end{cases} \tag{11}$$

$n \in (\mathtt{o}, \mathtt{entry}, \mathtt{t}_k) \cup (\mathtt{o}, \mathtt{notified\text{-}entry}, \mathtt{t}_k) \Rightarrow$

$$\Delta_n(\mu) = \begin{cases} \{(n_1, ..., n'_k, ..., n_{|T|}) \mid n'_k \in \mathit{LocalSucc}(n_k)\}, & \text{if } \forall i, 1 \leq i \leq |T|, i \neq k, n_i \neq \mathit{Monitor}_\mathtt{o} \\ \emptyset, & \text{otherwise} \end{cases} \tag{12}$$

$n \in (\mathtt{o}, \mathtt{notify}, \mathtt{t}_k) \Rightarrow$

$$\Delta_n(\mu) = \begin{cases} \{(n_1, ..., n'_k, ..., n_{|T|}) \mid n'_k \in \mathit{LocalSucc}(n_k)\}, & \text{if } \forall i, 1 \leq i \leq |T|, n_i \notin (\mathtt{o}, \mathtt{waiting}, *) \\ \bigcup_{1 \leq i \leq |T| : n_i \in (\mathtt{o}, \mathtt{waiting}, *)} \{(n_1, ..., n'_k, ..., n'_i, ..., n_{|T|}) \mid & \\ n'_k \in \mathit{LocalSucc}(n_k)\}, \text{ where }, n_i = \mathit{WaitingPred}(n'_i), & \text{otherwise} \end{cases} \tag{13}$$

$n \in (\mathtt{o}, \mathtt{notifyAll}, \mathtt{t}_k) \Rightarrow$

$$\Delta_n(\mu) = \begin{cases} \{(n_1, ..., n'_k, ..., n_{|T|}) \mid n'_k \in \mathit{LocalSucc}(n_k)\}, & \text{if } \forall i, 1 \leq i \leq |T|, n_i \notin (\mathtt{o}, \mathtt{waiting}, *) \\ \{(n_1, ..., n'_{i_1}, ..., n'_k, ..., n'_{i_r}, ..., n_{|T|}) \mid n'_k \in \mathit{LocalSucc}(n_k)\}, & \\ \text{ where }, \forall j, 1 \leq j \leq r, n_{i_j} \in (\mathtt{o}, \mathtt{waiting}, *), & \\ n_{i_j} = \mathit{WaitingPred}(n'_{i_j}) & \\ \wedge \forall s : \forall j, 1 \leq j \leq r, s \neq r, n_s \notin (\mathtt{o}, \mathtt{waiting}, *), & \text{otherwise} \end{cases} \tag{14}$$

otherwise

$$\Delta_n(\mu) = \{(n_1, ..., n'_k, ..., n_{|T|}) \mid n'_k \in \mathit{LocalSucc}(n_k)\} \tag{15}$$

If node n is not one of the nodes in the given marking μ, we define $\Delta_n(\mu)$ to return the empty set:

$$\Delta_n(\mu) = \emptyset \text{ if } n \in N(t_k) \text{ and } \mu = (n_1, ..., n_k, ..., n_{|T|}), \text{ where } n_k \neq n \tag{16}$$

The reachability graph is constructed in the usual way, picking nodes in the current marking one by one and executing their transition functions, connecting each resulting marking with the current one by an edge. This process starts with the initial marking $\mu_0 = (n_0, B_2, ..., B_{|T|})$, where $label(n_0) = (*, \texttt{begin}, \texttt{main})$. We denote the set of all reachable markings for a program *ReachMarkings*.

The following theorem proves that the presented reachability algorithm produces a conservative estimate of all reachable states of a concurrent Java program.

Theorem 5. *For any program state reached during the program execution there exists a marking μ produced by the reachability algorithm.*

Proof. We prove this by induction on the length of the program execution, measuring it by the length of the corresponding sequence of PEG nodes. This proof relies on our belief that the transitions presented in equations (10) – (16) completely and correctly describe the set of all possible actions in a concurrent Java programs under restrictions described in Section 2.1.

Before any code was executed, the program is in its initial state, where only the main thread is initialized and ready to start execution. This is represented by marking μ_0.

Suppose the statement of the theorem holds for the first s steps of the program execution

Consider the $s+1$'st step of the execution. Since a step of the execution is represented by an instruction in one of the threads, and these instructions are mapped to the PEG nodes, this execution can be represented by the execution of a PEG node. The transitions of the reachability function accurately describe all possible results of an execution of a PEG node. Therefore, the state of the program reached as the result of this $s+1$'st step is represented by one of the markings obtained as the result of the reachability algorithm transition based on the executed node. □

Before we prove conservativeness of the *MHP* algorithm, we prove the following lemma stating that the reachability algorithm never produces a marking containing two nodes from the same monitor.

Lemma 6. $\forall \mu \in ReachMarkings, \forall m, n \in \mu : \neg\exists Monitor_o, m, n \in Monitor_o$.

Proof. We prove the statement of this lemma by induction on the path from μ_0 to μ. To prove the inductive step, we investigate all possible ways in which nodes from monitors may be inserted in the markings. For each of these ways we show that a node from a monitor is never inserted in a marking if this marking already contains a node from this monitor.

If the length of this path is 0 (in other words, $\mu = \mu_0$), the statement of this lemma trivially holds, since the initial marking contains only one node (the first node of the main thread).

Suppose that the statement of this lemma holds for all paths from μ_0 to μ of length s.

Consider any μ such that the path from μ_0 to μ has length $s+1$. Let μ' be the previous marking on this path, i.e., the μ' is reachable from μ_0 in s transitions and $\exists r \in N : \mu \in \Delta_r(\mu')$. By the induction hypothesis, the statement of this lemma holds for μ'. Consider the following cases based on the label of r.

Suppose first that $r \in (o, \texttt{entry}, *) \cup (o, \texttt{notified-entry}, *)$ for some lock object o. By the transition rule (12), in order for transition $\Delta_r(\mu')$ to produce a non-empty set of markings, it should be the case that no nodes in μ' except r may be in $Monitor_o$. Since only r is changed in marking μ' to obtain marking μ, the statement of the lemma holds for μ.

Now consider all other labels, i.e., $r \notin (*, \texttt{entry}, *) \cup (*, \texttt{notified-entry}, *)$. By the construction of the PEG, a thread enters a monitor only after its current node changes from an `entry` or `notified-entry` node to any of its successors. According to the transition rules, this is possible only as the result of transition (12). Thus, since all other transitions cannot change the number of nodes from any given monitor in the marking, the statement of this lemma holds for μ. □

The following theorem proves conservativeness of the *MHP* algorithm by showing that for any marking obtained by the reachability algorithm, all pairs of nodes from this marking are identified by the *MHP* algorithm as able to happen in parallel with each other.

Theorem 7 (Conservativeness). $\forall \mu \in \mathit{ReachMarkings}, \forall n, m \in \mu : n \in M(m) \wedge m \in M(n)$.

Proof. We prove this theorem by induction on the length of the path from μ_0 to μ. For the inductive step we consider the last transition on the path from μ_0 to μ. We consider all possible labels for the node on which this transition is based. For each label, we show that, under the inductive hypothesis, any two nodes that appear in marking μ are placed in the M sets of each other by the base algorithm 2. The statement of this theorem trivially holds for a path of length 0 ($\mu = \mu_0$), since μ_0 contains only one PEG node.

Suppose that the statement of this theorem holds for all markings reachable from the initial marking via a path of length s.

Now consider any marking μ reachable from μ_0 via a path of length $s+1$. Let μ' be the previous marking on this path, i.e., the μ' is reachable from μ_0 in s transitions and $\exists r \in N : \mu \in \Delta_r(\mu')$. By the induction hypothesis, the statement of this lemma holds for μ'. We consider all possible cases based on the label of r. Note that in all cases it is implied that if neither n nor m were changed by transition Δ_r, by the induction hypothesis $n \in M(m)$. Also, in all cases the symmetry step of the *MHP* algorithm ensures that whenever it is proved that $n \in M(m)$, it automatically follows that $m \in M(n)$, and vice versa.

$r \in (\mathtt{t}_k, \mathtt{start}, \mathtt{t}_i)$ There are several possibilities that depend on the place of n and m in μ. Suppose first that $n = n_i$ (and thus $r \in \mathit{LocalPred}(n)$) and $m \neq n_k$. Then $m \in \mu'$. By the inductive hypothesis ($r \in \mu'$ as well), $m \in M(r)$. According to the *MHP* algorithm, m is placed in $\mathit{OUT}(r)$ (KILL set of a `start` node is always empty). Thus, m propagates into $\mathit{IN}_{\mathit{LocalPred}}(n)$ and hence into $M(n)$.

Suppose now that $n = n_i$ and $m = n_k$. Then, according to the reachability transition (10), $m \in (*, \mathtt{begin}, \mathtt{t}_k)$. Then the *MHP* algorithm puts m in the $\mathit{GEN}_{\mathit{start}}(r)$ set. Hence, $m \in \mathit{OUT}(r) \Rightarrow M(n)$.

Suppose that $n \neq n_i$ and $m = n_k$. Then $n \in \mu'$ and so by the inductive hypothesis, $n \in M(r) \Rightarrow n \in M(r) \Rightarrow n \in \mathit{OUT}(r) \Rightarrow n \in \mathit{IN}_{\mathit{StartPred}}(m) \Rightarrow n \in M(m)$.

$r \in (\mathtt{t}_k, \mathtt{join}, \mathtt{t}_i)$ Suppose first that $n = n_i$ and $m \neq n_k$. Then $m \in \mu'$ and by the inductive hypothesis $m \in M(r)$. Because $m \notin N(t_k)$, it follows that $m \notin \mathit{KILL}_{\mathit{join}}(r)$. Hence, m propagates into $\mathit{OUT}(r)$ and from there into $M(n)$.

Note that it is impossible that $m = n_k$ because for the transition (11) to be possible, it must be the case that $n_k = D_k$ before the transition. Since this transition does not change n_k, it remains equal D_k after the transition.

$r \in (\mathtt{o}, \mathtt{entry}, \mathtt{t}_i) \cup (\mathtt{o}, \mathtt{notified\text{-}entry}, \mathtt{t}_i)$ Suppose $n = n_i$. Because the transition (12) changes only one node in the marking, $m \in \mu'$. By the induction hypothesis, $m \in M(r)$. Then, by construction of monitors (any `entry` node is outside the monitor but all its successors are inside the monitor), $n \in \mathit{Monitor}_{\mathtt{o}}$. By Lemma 6, $m \notin \mathit{Monitor}_{\mathtt{o}}$, and so $m \notin \mathit{KILL}_{\mathit{Monitor}}(r)$. Thus, $m \in \mathit{OUT}(r)$, and so m propagates into $M(n)$.

$r \in (\mathtt{o}, \mathtt{notify}, \mathtt{t}_i)$ We consider several possibilities based on the position of m and n in marking μ and their labels. Suppose first that $n = n_i$, $m \in (\mathtt{o}, \mathtt{notified\text{-}entry}, \mathtt{t}_i)$, and m is a new node in the marking (i.e., $m \notin \mu'$). Let node u be the `waiting` predecessor of m: $u = \mathit{WaitingPred}(m)$. Since $u \in \mu'$, by the inductive hypothesis $u \in M(r)$. According to the *MHP* algorithm, this is the necessary condition for $r \in \mathit{NotifyPred}(m)$. Using this connection between r and m, the *MHP* algorithm puts m in $\mathit{GEN}_{\mathit{notify}}(r)$. Because m is not a `waiting` node, $m \notin \mathit{KILL}(r)$, and so $m \in \mathit{OUT}(r) \Rightarrow m \in M(n)$.

Suppose now that $n = n_i$ and m is not a new node in the marking. Then $m \in \mu'$ and so $m \in M(r)$. Suppose that $m \in (\mathtt{o}, \mathtt{waiting}, *)$. According to the transition rule (13), there must have been another `waiting` node for the monitor of the lock object o in marking μ' that was used for this transition. Thus, $\exists m, q \in \mu' : m, q \in (\mathtt{o}, \mathtt{waiting}, *)$. This means that $m \notin \mathit{KILL}_{\mathit{notifyAll}}(r)$. If m is not a `waiting` node for the monitor of the lock object o, then $m \notin \mathit{KILL}_{\mathit{notifyAll}}(r)$. Hence, $m \in \mathit{OUT}(r) \Rightarrow m \in M(n)$.

Suppose that $n \neq n_i$ and m is a new `notified-entry` node in the marking. Since $n \in \mu'$, by the induction hypothesis $n \in M(r)$. Suppose first that $n \notin (\mathtt{o}, \mathtt{waiting}, *)$. This means that $n \notin \mathit{KILL}(r)$ and so $n \in \mathit{OUT}(r) \Rightarrow n \in \mathit{IN}_{\mathit{NotifyPred}}(m) \Rightarrow n \in M(n)$. Suppose now that $n \in (\mathtt{o}, \mathtt{waiting}, *)$. Then there are at least two `waiting` nodes for the monitor of the lock object o in markings μ'. It follows that $n \notin \mathit{KILL}_{\mathit{notifyAll}}(r) \Rightarrow n \in \mathit{OUT}(r) \Rightarrow n \in M(m)$.

```
class MyThread1
{
  Object lock;
  public MyThread1(Object obj)
  {
    lock = obj;
  }
  public void run()
  {
    synchronized (lock)
    {
      System.out.print("stmt1");
    }
  }
}

class Main
{
  public static void
   main(String [] args)
  {
    MyThread2 t2 = new Mythread2();
    t2.start();
    System.out.print("stmt2");
  }
}
```

```
class MyThread2 extends Thread
{
  MyThread2()
  {
    super();
  }
  public void run()
  {
    Object lock = new Object();
    MyThread1 t1 = new MyThread1(lock);
    synchronized (lock)
    {
      t1.start();
      System.out.print("stmt3");
      t1.join();
    }
  }
}
```

Figure 9: Code for a counter-example showing that the *MHP* algorithm is less precise than the reachability algorithm

$r \in (\mathtt{o}, \mathtt{notifyAll}, \mathtt{t}_i)$ Transitions on `notifyAll` nodes may produce multiple new `notified-entry` nodes in marking resulting from the transition rule (14). Also, according to the same transition rule, marking μ contains no `waiting` nodes for the monitor of the lock object used in this transition.

Suppose first that $n = n_i$ and m is a new `notified-entry` node in marking μ. Reasoning that proves that m propagates into $M(n)$ is the same as in the similar case for transitions based on `notify` nodes.

Suppose now that $n \neq n_i$ and n is not a new `notified-entry` node, while m is a new `notified-entry` node. Then $n \in \mu'$ and by the induction hypothesis $n \in M(r)$. Since, according to the rules of this transition, it is impossible that $n \in (\mathtt{o}, \mathtt{waiting}, *)$, $n \notin KILL(r) \Rightarrow n \in OUT(r) \Rightarrow n \in IN_{NotifyPred}(m) \Rightarrow n \in M(m)$.

Suppose that both m and n are new `notified-entry` nodes in marking μ. Let $p_1 = WaitingPred(n)$ and $p_2 = WaitingPred(m)$. Since $p_1, p_2 \in \mu'$, by the induction hypothesis $p_1 \in M(p_2) \wedge r \in M(p_1) \wedge r \in M(p_2)$. Hence, $n \in GEN_{notifyAll}(m)$ and thus $n \in M(m)$.

Finally, suppose that $n = n_i$ and m is not a new `notified-entry` node. Then $m \in \mu'$ and by the induction hypothesis $m \in M(r)$. According to the transition rule (14), it is impossible that $m \in (\mathtt{o}, \mathtt{waiting}, *)$, which means that $m \notin KILL(r) \Rightarrow m \in OUT(r) \Rightarrow m \in M(n)$.

All other cases Whenever a transition happens based on a local event, only one node r in marking μ' changes to one of its successors. Suppose that n is such a successor in marking μ. Since $m \in \mu'$, by the induction hypothesis $m \in M(r)$. Since $KILL(r) = \emptyset$, m propagates into $OUT(r)$ and from there to $M(n)$.

Note that the list of all possible labels of r is missing the possibility of r being a `waiting` node. The reason for this is that a transition (15) based on a `waiting` node produces no new markings because `waiting` nodes have no local successors. □

The following lemma shows that in general the *MHP* algorithm is not as precise as the reachability algorithm. This is done by showing a counterexample to the hypothesis that the *MHP* algorithm is not less precise than the reachability algorithm.

Lemma 8. *$\exists$ a program such that $\exists m, n \in N : m \in M(n) \wedge \forall \mu \in ReachMarkings : m \in \mu \wedge n \notin \mu$.*

Proof. We prove the statement of this lemma by demonstrating an example, the code for which is shown in Figure 9 and the corresponding PEG shown in Figure 10. For this example the *MHP* algorithm finds a pair of nodes that never happen in the same marking found by the reachability algorithm.

In this program, the main thread starts thread `t2` and then executes a `print` statement. Thread `t2` enters a monitor, starts thread `t1`, executes a `print` statement, waits for thread `t1` to terminate and then exits the

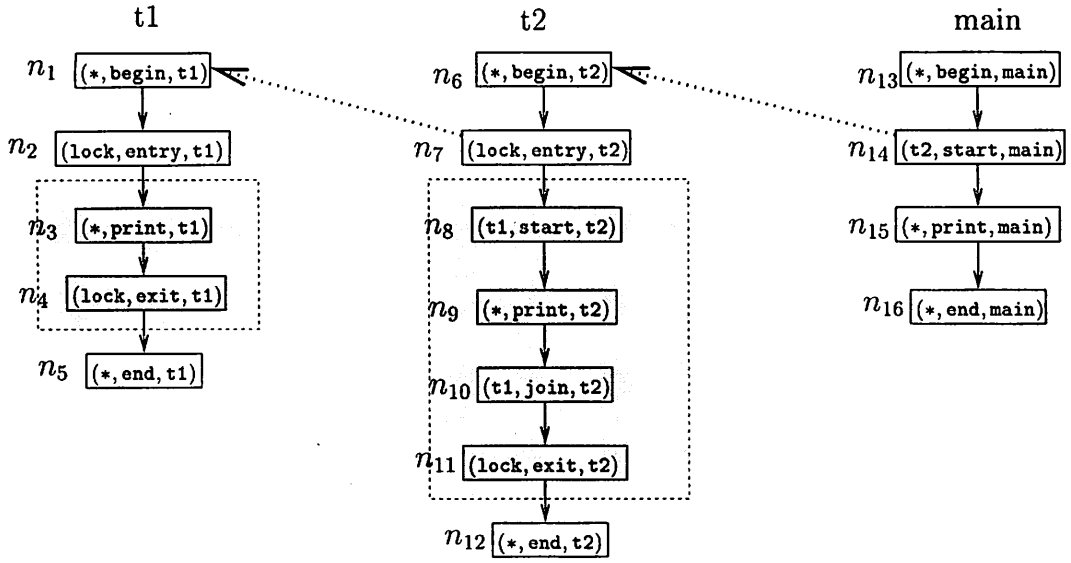

Figure 10: PEG for the counter-example in Figure 9

monitor. Thread `t1` enters the same monitor, executes a `print` statement and then exits the monitor. Note that the example program always deadlocks, because by the time thread `t2` starts thread `t1` (at node n_8), thread `t1` is already in the monitor. Further, thread `t1` does not exit the monitor before calling the `join` method of thread `t2` (at node n_{10}). This means that thread `t1` can never enter the monitor, in other words, go past node n_2. Thus, the program enters a state where thread `t2` is waiting for thread `t1` to terminate and thread `t1` is waiting for thread `t2` to exit the monitor. This lets the reachability algorithm detects that node n_3 is not reachable. In the *MHP* algorithm, however, node n_{15} can propagate into $M(n_3)$ (since node n_{15} is not in the monitor, it can propagate from $OUT(n_2)$ in $M(n_3)$. □

6 Experimental Results

In this section we present the results of an initial experimental evaluation of the *MHP* algorithm. We compare the precision and analysis time of our algorithm with that of a reachability-based analysis algorithm introduced in Section 5. This reachability algorithm represents a theoretically more precise but less efficient way to compute the *MHP* information. We show that for the sample of concurrent Java programs we used in this study, our algorithm has significantly lower timing requirements than the reachability algorithm, while being almost as precise. In the next subsection we describe the design of our experiment and the programs we used. In Section 6.2 we give the numerical results of the experiment. Finally, in Section 6.3 we discuss and summarize the results of this experiment.

6.1 Design of the Experiment

We use the reachability analysis approach from Section 5 to measure the precision of the *MHP* analysis. According to Theorem 7, the reachability analysis always computes at least as precise *MHP* information as the *MHP* algorithm. (The reachability algorithm exhaustively examines all possible paths through the PEG and thus finds not only pairs of nodes that may happen in parallel, put also triples, quadruples etc. of such nodes. This adds precision because more information about node interaction is captured.) Thus, the *MHP* information computed by the reachability analysis represents an "ideal" *MHP* information for the program. Note that this does not imply that if the reachability algorithm finds that two nodes may happen in parallel, then there exists a real execution of the program on which these two nodes happen in parallel. It is possible

that in reality no such execution exists because manipulations of program variables preclude some of the executions of the program that were examined by the reachability algorithm.

We measure the precision of both the *MHP* algorithm and the reachability analysis by the number of pairs of nodes that this approach claims can happen in parallel. The smaller this number, the more precise is the approach, since both approaches can never underestimate the possibility for two nodes to 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 $(|P_{MHP} \setminus P_{Reach}|)/|P_{Reach}|$.

In comparing time requirements for the two approaches, we do not not include the time taken to derive the PEG model, because this has to be done by both approaches. Thus, for each example we compute three timing characteristics: the time to build the PEG model, the time to run the *MHP* algorithm, and the time to run the reachability analysis for this example. For our experiments, we used a Symantec JIT compiler for JDK 1.1 on a workstation equipped with a 266 MHz Pentium II processor and 128Mb of memory, running Windows NT.

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 to enable the parser currently used with the *MHP* algorithm to handle them, so that we could use all the example without a preliminary semantical analysis. In addition, we removed the timed versions of the synchronization statements and any exception handling, since thse are currently not handled in our algorithm.

The majority of our examples came from Doug Lea's book on Java concurrency [9] and its Web supplement [10]. For the majority 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. For example, given the class `Semaphore`, we wrote a program that consists of four threads, each of which repeatedly asks for permission to enter a single common semaphore. This semaphore permits no more than a pre-set number of threads to execute a critical region of code. After having received permission to enter the semaphore, each thread executes this critical region and then exits the semaphore. In this and most other examples we removed some statements, such as assignment statements, that do not affect the precision of the *MHP* information computed by either the *MHP* algorithm or the reachability algorithm. This allowed us to reduce the size of the reachable state space for the examples, which enabled the reachability algorithm to complete in more cases. Similarly, for examples involving a number of identical threads, such as producer-consumer type of examples where a number of producer and consumer threads utilize a shared resource, we sometimes had to decrease the number of threads to enable the reachability algorithm to handle these cases.

Several more examples in our set came from other sources [1, 2, 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. The following table lists all 29 examples and for each gives its brief description and an indication of where it comes from.

Name	Description	From
AlternatingMessagePrinter	A simple example with two threads alternatingly entering a monitor to print a line of text.	p. 19, [9]
AutomatedBanking	Extends the PessimBankAccount example by adding a thread that automatically transfers money from a checking account to the linked savings account if the amount in the checking account exceeds a certain threshold.	p. 290, [9]
GroupPictureRenderer	Simulates rendering a picture. The main thread divides the picture into three parts and starts a thread to render each of the parts. The main thread then waits for these three threads to terminate before proceeding to assemble display the completed picture.	p. 206, [9]

HeatingSystem	A temperature sensor measures outside temperature and passes the measurement on to a temperature comparator, which decides whether the temperature is above or below optimal. The comparator then passes a command onto a heater to either turn on or turn off. The three threads implementing these three entities communicate via bounded synchronized buffers.	based on p. 227, [9]
HeatingSystemPutTake	Same as the HeatingSystem example, except that threads communicate via zero-capacity put-take connectors [10].	based on p. 227, [9]
PCTwoLockQueue	Two producers and two consumers operate on a synchronized queue that uses different locks for putting elements at the end of the queue and taking elements from the front of the queue.	based on p. 76 and p. 144, [9]
PessimBankAccount	Two users have access to two accounts. The available operations are a deposit to, withdrawal from an account, and transfer between the two accounts.	p. 271, [9]
PrintService	Two print service threads compete for a single printer resource. Instead of using a synchronized resource, the threads use their own locks to gain exclusive use of the printer. This example contains a deadlock.	p. 52, [9]
RWVSN	An example with two readers and two writers accessing a shared resource. A complicated system of locks is used to implement an efficient queueing mechanism.	p. 301, [9]
SplitRenderer	Similar to the GroupPictureRenderer example, except here the main thread uses the `join` mechanism instead of the `wait()` mechanism to wait for the four threads that it starts to terminate.	p. 204, [9]
SplitRendererNested	Similar to the SplitRenderer example, except that the main thread starts two helper threads, each of which may or may not start two more helper threads. The `join` mechanism is used to ensure that the helper threads complete.	based on p. 204, [9]
ThreadedApplet	A very simple applet involving three threads without synchronization.	p. 15, [9]
ThreadedAppletV2	The ThreadedApplet example with a single monitor added.	p. 15, [9]
CyclicBarrier	Implements the synchronization scheme where all threads (four in this instantiation) repeatedly synchronize at a single point.	[10]
LayeredSemaphore	Protects a region of critical code with two semaphores instead of one, as in the Semaphore example. Four threads repeatedly access this region of critical code.	based on LayredSync, [10]
Mutex	Implements a non-reentrant locking mechanism. This example contains three threads that repeatedly acquire and release this lock and one thread that attempts to acquire this lock twice without releasing it between the first and second acquisitions.	[10]

Semaphore	Contains four threads that repeatedly enter a semaphore that permits a limited number of threads to execute at the same time.	[10]
SemaphoreControlledChannel	Two sending and two receiving threads use a message channel that is protected with two semaphores.	[10]
BridgeTest	Simulates a one-lane bridge that can hold at most 3 cars at a time. This example represents an instantiation with 6 cars.	[1]
OneCarBridgeTest	Simulates a one-lane bridge that can hold at most 1 car at a time. This example represents an instantiation with 6 cars.	[1]
CHAN_OF_INT	An example of two readers and two writers using the channel with different queues for waiting readers and writers	[2]
CorrectHalves	An applet that starts two threads, of which one scans down the top of the window, drawing it alternately red then white, and one which scans down the bottom of the window, drawing it alternately green then black.	[3]
CorrectSquares	An applet that starts two threads which attempt to simultaneously cause squares of random sizes to be drawn.	[3]
IncorrectHalves	Same as the CorrectHalves example except that insufficient synchronization results in colors appearing in incorrect halves of the screen.	[3]
IncorrectSquares	Same as the CorrectSquares example except that insufficient synchronization results in the possibility of drawing non-square rectangles.	[3]
Chiron	Our implementation of an instantiation of a Chiron user interface architecture [7].	Written by the authors based on the Ada example
Phil	The dining philosophers example with three philosophers sitting in a circle. Each of the philosophers alternates between eating and thinking. To eat, a philosopher must acquire two forks, each of which it shares with one of its neighbors. To avoid deadlock, all philosophers except one pick the fork to their left first, then the fork to their right. All philosophers are implemented with threads and the forks are represented as resources shared by the philosopher threads.	Written by the authors based on the Ada example
RW	The readers-writers example with two readers and two writers repeatedly accessing a single shared object protected by a single lock.	Written by the authors based on the Ada example
GasStation	A simulation of a gas station with three customers, one cashier, and one pump. To get gas, each customer must pay the cashier first. Having obtained a payment, the cashier orders the pump to dispatch a certain amount of gas to the correct customer. All customers and the cashier are implemented as threads and the pump and the cash register are implemented as shared synchronized resources.	Written by the authors based on the Ada example

Program	Threads	Nodes	Synch. nodes	$\|P_{MHP}\|$	$\|P_{MHP} \setminus P_{Reach}\|$	R	time PEG	time MHP	time Reach
AlternatingMessagePrinter	3	35	27	261	0	0	0.33	0.10	0.08
AutomatedBanking	3	280	182	11166	0	0	0.27	2.03	7.51
BridgeTest	5	66	46	1193	0	0	0.24	0.15	11.79
CHAN_OF_INT	5	62	54	974	40	0.043	0.26	0.13	5.76
Chiron	5	112	87	3382	0	0	0.26	0.29	1059.57
CorrectHalves	3	33	24	188	0	0	0.67	0.11	0.08
CorrectSquares	4	33	21	271	0	0	0.24	0.08	0.14
CyclicBarrier	5	54	38	887	0	0	0.3	0.12	10.45
GroupPictureRenderer	4	35	26	240	0	0	0.23	0.09	0.13
GasStation	5	93	71	2626	0	0	0.29	0.24	84.61
HeatingSystem	4	66	41	1140	0	0	0.27	0.17	2.69
HeatingSystemPutTake	4	76	53	1497	0	0	0.25	0.21	4.56
IncorrectHalves	3	31	20	174	0	0	0.23	0.08	0.06
IncorrectSquares	4	27	15	171	0	0	0.23	0.08	0.09
LayeredSemaphore	4	77	59	1720	0	0	0.24	0.21	5.90
Mutex	5	75	54	1688	0	0	0.69	0.23	36.25
OneCarBridgeTest	7	39	32	420	0	0	0.29	0.08	10.29
PCTwoLockQueue	5	34	22	358	0	0	0.25	0.09	1.01
PessimBankAccount	3	446	288	44128	0	0	0.35	6.28	41.91
Phil	4	68	59	1355	0	0	0.26	0.19	3.78
PrintService	3	23	16	108	0	0	0.27	0.08	0.05
Readers-writers	5	56	46	876	0	0	0.24	0.11	7.75
RWVSN	5	116	80	3224	0	0	0.30	0.27	107.81
Semaphore	5	66	46	1349	0	0	0.49	0.17	25.23
SemaphoreControlledChan	5	66	46	1423	0	0	0.25	0.16	32.22
SplitRenderer	5	33	18	250	0	0	0.21	0.08	0.22
SplitRendererNested	7	51	26	727	50	0.074	0.25	0.09	4.74
ThreadedApplet	3	10	8	18	0	0	0.20	0.06	0.04
ThreadedAppletV2	3	14	12	36	0	0	0.21	0.08	0.05

Figure 11: Raw experimental data

6.2 Results

Figure 11 presents the raw data of 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 and sixth column give the number of node pairs found by the *MHP* and reachability algorithm respectively; the seventh column gives the number of node pairs found by the *MHP* algorithm but not by the reachability analysis; finally, the eighth, ninth, and tenth columns show the time in seconds taken to construct the PEG for the example and to run the *MHP* and reachability algorithms respectively.

Note that 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 the latter, the main thread

starts two helper threads (*external* helper threads) and then executes `join()` for each of them. Each of these helper threads can in turn spin off two helper threads of their own (*internal* helper threads) and then execute `join()` for each of them. The information available to the *MHP* algorithm is not sufficient to deduce that when a call to the `join()` method of one of the external helper threads in the main thread succeeds, it means that both internal helper threads for this external helper thread, if started, must have terminated. Note that this example was constructed by us specifically to demonstrate a case where imprecision is introduced in this way. While this example represent a reasonable architecture, it would be interesting to see if the `join()` mechanism of Java is used in similar examples in practice.

6.3 Discussion

The results of this initial experiments are very encouraging, as they show that our algorithm is both very precise and efficient. Of the 29 examples we ran the *MHP* algorithm and the reachability analysis on, only in two cases did the *MHP* algorithm find some node pairs that were not found by the reachability analysis. In both of these cases the number of pairs found by the *MHP* algorithm that were not found by the reachability analysis was small compared to the total number of pairs of nodes that may happen in parallel (40 out of 934 for the CHAN_OF_INT example and 50 out 677 for the SplitRendererNested example).

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 1 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.

7 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 the 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 extend the *MHP* algorithm to apply to programs containing method calls without using inlining. 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 the *MHP* information computed for this node is sufficient to determine the *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, the *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 [6], and modifying the *MHP* algorithm accordingly.

At present, the *MHP* algorithm is being used as a part of the FLAVERS/Java tool [14] for data flow-based verification of application-specific properties of concurrent Java programs. 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 SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 100–111, 1988.

[5] 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, Victoria, B.C., October 1991.

[6] 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.

[7] R. K. Keller, M. Cameron, R. N. Taylor, and D. B. Troup. User interface development and software environments: The chiron-1 system. In *Proceedings of the 13th International Conference on Software Engineering*, pages 208–218, October 1991.

[8] 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. ACM Press, June 1998.

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

[10] D. Lea. Concurrent programming in Java. Design principles and patterns, online supplement. `http://gee.cs.oswego.edu/dl/cpj/index.html`, Sept. 1998.

[11] 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. ACM, May 1991.

[12] S. Masticola and B. Ryder. Non-concurrency analysis. In *Proceedings of the Twelfth of Symposium on Principles and Practices of Parallel Programming*, San Diego, CA, May 1993.

[13] G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel, Nov. 1998. To appear in the proceedings of the Sixth ACM SIGSOFT Symposium on the Foundations of Software Engineering. `http://laser.cs.umass.edu/abstracts/98-023.html`.

[14] G. Naumovich, G. S. Avrunin, and L. A. Clarke. Data flow analysis for checking properties of concurrent Java programs. Technical Report 98-22, University of Massachusetts, Amherst, Apr. 1998. `http://laser.cs.umass.edu/abstracts/98-022.html`.

[15] R. N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.