# A Class of Synchronization Operations that Permit Efficient Race Detection

D. P. Helmbold, C. E. McDowell

UCSC-CRL-93-29
August 2, 1993

Board of Studies in Computer and Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064

## ABSTRACT

We present an efficient algorithm for "precisely" determining the possible alternate orderings of events from a program trace for a class of synchronization operations. The class includes, post and wait without clear, a restricted form of semaphores, and message passing where the sender names the receiver. The ordering information is precise in that a race detection system could be built using the ordering information such that if the program does not contain any data races for a given input, no races will be reported, and if the the program does contain data races, at least one of those races will be reported.

**Keywords:** trace analysis, race detection, debugging, parallel programming, event ordering

# 1 Introduction

A common task when debugging parallel programs containing explicit synchronization is identifying the sources of non-determinism generally referred to as races. One approach to detect races is to analyze a trace of a program execution. The simplest technique is to look for unsynchronized blocks of code that were actually executed concurrently and contain conflicting data accesses (such as a read and a write to the same variable). This simple technique identifies only those races that actually occurred and may significantly under report the races in the program as even slight variations in the speed of the tasks can result in different pairs of blocks executing concurrently, and different races being detected. In order to improve the accuracy of race detection it appears necessary to consider alternative executions when analyzing a trace.

The approach of our work and others [NM91, NG92, HMW93] is to compute a partial order representing the order in which the events from a trace did or might have occurred. This ordering information can be combined with memory access information to perform either on-the-fly or post-mortem race detection. The goal is a partial order $R$ such that if $e_1 R e_2$ then $e_1$ is guaranteed to happen before $e_2$ in every execution in which they both occur. The results have always fallen short of this goal, at least for polynomial time algorithms. The resulting partial orders have either ordered events that should not have been ordered (missing potential races) or failed to order events that are in fact always ordered (reporting infeasible races).

Recently, Netzer and Ghosh showed how to compute a "precise guaranteed ordering graph" in polynomial time for programs using post and wait synchronization (without clears) [NG92]. Their ordering graph is precise in that a race detection system that relies on this ordering information can be constructed such that:

1. If the program contains races when executed with the given input then the detection system will report a non-empty subset of the races that could occur.
2. If the program contains no races when executed with the given input then the detection system will report no races.

The race detection system may still miss some races and report infeasible races that are artifacts (i.e. controlled by outcomes) of earlier races.

Two events $e_1$ and $e_2$ are ordered by Netzer and Ghosh's guaranteed ordering graph if and only if $e_1$ is guaranteed to happen before $e_2$ in every execution of the straight line program resulting from changing all conditional branches in the original program into unconditional branches in the direction taken by the execution producing the trace being analyzed. This straight line program is *inferred* from the original program and the trace. We introduced the inferred program to characterize the guaranteed orderings in a previous paper [HMW93], and we will use it again in this paper.

**Definition 1.1:** *Given a trace $T$ of some program $P$, the program* **inferred** *by $P$ and $T$ (written $P_T$) is the same as program $P$ except that all conditional branches are replaced with unconditional branches that branch to the destination taken during the execution that created trace $T$.*

Note that program $P_T$ is never created, it is simply a vehicle for reasoning about event orderings.

Another justification of the "precise" claim for Netzer and Ghosh's result is the *if and only if* nature of their guaranteed ordering graph (i.e. they find precisely the orderings that hold for the inferred program). In this paper we extend Netzer and Ghosh's result for post/wait synchronization without clears to other kinds of synchronization constructs. We define a class of synchronization constructs for which there exists an efficient algorithm to determine this "precise guaranteed ordering graph."

Informally, a synchronization construct is in the class if all straight line programs that use this construct either always terminate normally (all tasks complete) or always deadlock in the same state (i.e. each task is blocked on a particular synchronization operation). This class includes:

- post/wait synchronization without clears [NG92],
- message passing where the receiver is named by the sender, and
- semaphores when all $P(X)$ operations for a particular semaphore $X$ occur in the same task.

Netzer and Ghosh define a *maximal valid execution* [NG92] for an event $e$ which is a maximal sequence of post and wait events from an inferred program that satisfies the post/wait semantics and could precede the event $e$. Although an event could have several maximal valid execution sequences, they prove that, for any event $e$, all maximal valid execution sequences for that event contain exactly the same set of events (although possibly in different orders). Note that each maximal valid execution is a prefix of a serialized execution of program $P_T$, but not every prefix of a serialization of $P_T$ is a maximal valid execution.

The main result of this paper is that if the synchronization constructs used yield a similar notion of maximal sequence where each event's maximal valid execution sequences contains the same set of events, then there is an efficient algorithm to compute the precise guaranteed ordering graph for the inferred program in polynomial time. This uniqueness of maximal valid execution sequences is characterized by Lemma 3.4 and Definition 3.2, i.e., any program that uses only *monotonic* synchronization constructs as defined below will exhibit the desired behavior.

## 2 Events and Orderings

The guaranteed ordering graph contains nodes that represent events from a particular execution of a program. Since we wish to generalize the ordering information from a single trace to a program as a whole, we must have a way of identifying the "same" event in another execution.

Although any program activity of interest could be represented as an event, we make the simplifying assumptions that all events represent the execution of a synchronization construct in the program and each execution of a synchronization construct is represented

by an event. In addition, we assume that the events in any execution can be legally serialized. Note that some high level synchronization constructs, such as Ada entry call and accept, cannot be represented as single events since neither a call nor the corresponding accept can finish before the other starts. This problem is easily resolved by treating an entry call as two events.

**Definition 2.1:** *Event $e_1$ in trace $T_1$* **corresponds** *to event $e_2$ in trace $T_2$ if $e_1$ and $e_2$ represent the execution of the same synchronization statement(s) by the same*[1] *task and $pred(e_1)$*[2] *in $T_1$ corresponds to $pred(e_2)$ in $T_2$.*

In the remainder of this paper, if event $e_1$ in trace $T_1$ corresponds to event $e_2$ in trace $T_2$ we will use $e_1$ (or equivalently $e_2$) to refer to both events (e.g. $e_1$ in $T_2$ corresponds to $e_1$ in $T_1$). Note that given a trace $T_1$ of program $P$ on some input, there may be another trace $T_2$ of $P$ on the same input where not all events in $T_1$ correspond to events in $T_2$.

We use "$\xrightarrow{T}$" to represent the causal ordering of the events in a particular execution or trace $T$. This causal ordering will generally be a partial order and is the irreflexive transitive closure of edges from each event to the next event (if any) executed by the same task and edges from unblocking events to the blocked events that they unblocked. Note that we use the $\xrightarrow{T}$ causal ordering for analysis purposes and need not compute it. Our algorithms require only local logs indicating the events executed by each task (in the order they were executed by that task).

The following formally defines the *must-have-happened-before* relation which is the relation represented by the guaranteed ordering graph. We use "$\prec$" to represent the must-have-happened-before relation for an inferred program.

**Definition 2.2:** *Let $T$ be trace of parallel program $P$ running with some input $I$. For any two events $e_1$ and $e_2$ in $T$, $e_1 \prec e_2$ iff $e_1 \xrightarrow{T'} e_2$ in every trace $T'$ of the inferred program $P_T$.*

Different executions of the same program on the same input can have different $\xrightarrow{T}$ relations. The must-have-happened-before relation, $\prec$, is the intersection of all these $\xrightarrow{T}$ relations.

## 3 Monotone Synchronization and Ordering Relationships

In this section we define the class of programs that our algorithm is designed to operate on (MSSL programs). We then prove a close relationship between traces of MSSL programs and the must-have-happened-before relation (Theorem 3.5). We end the section with a proof that our algorithm can be used to satisfy the two requirements of precise guaranteed ordering graphs mentioned in the introduction.

---

[1] Tasks can be identified across executions by a simple lexicographic method such as the English half of an English-Hebrew label [NR88].

[2] $pred(e)$ is the event preceding event $e$ in the same task. If $e$ is the first event in the task then $pred(e) = \emptyset$ and $\emptyset$ for one trace corresponds to $\emptyset$ for all other traces.

**Definition 3.1:** *A* **straight line parallel program** *is one in which there are no synchronization operations inside of loops or conditional statements and all loops terminate on all inputs.*

**Definition 3.2:** *A set of synchronization constructs is* **monotonic** *if every straight line parallel program using only synchronization constructs in the set either always terminates normally (all tasks complete) or always deadlocks in the same state.*

A consequence of the above definition is that once an event becomes unblocked, it remains unblocked throughout the program's execution. This is the motivation behind calling the set of synchronization constructs monotonic.

**Definition 3.3:** *If the synchronization constructs used by a program are monotonic then the program is a* **monotonically synchronized (MS)** *program. If, in addition, the program is a straight line program, then it is a* **monotonically synchronized straight line (MSSL)** *program.*

If program $P$ is a monotonically synchronized program then the program $P_T$ inferred by trace $T$ of $P$ is a MSSL program. Theorem 3.5 (below) gives us an easy way to determine the must-have-happened-before relation for a MSSL program. For each event, form the modified program and compare their traces. This idea was used by Netzer and Ghosh [NG92] and is exploited by our algorithms in the next section.

**Lemma 3.4:** *Every trace of a MSSL program contains the same events.*

**Proof**: Follows directly from the definition.

□

**Theorem 3.5:** *Let program $P$ be a MSSL program where some task $t$ executes event $e$, program $P'$ be the MSSL program $P$ modified so that task $t$ stops just before executing event $e$, $T$ be a trace of $P$, and $T'$ be a trace of $P'$.*
*In $P$, event $e \prec e'$ iff $e' \in T$ and $e' \notin T'$.*

**Proof**: Since $P$ (and $P'$) are MSSL programs, each trace of $P$ (resp. $P'$) contains the same events. Since $e$ is executed by task $t$ in $P$, event $e$ appears in every trace of $P$.

For the forward direction assume that $e \prec e'$ (with respect to program $P$). By definition, for every trace $T$ of $P$ we have $e \xrightarrow{T} e'$. Since every trace of $P'$ is also a partial trace of $P$ and no trace of $P'$ contains event $e$, no trace of $P'$ can contain event $e'$.

For the other direction assume to the contrary that event $e' \in T$, event $e' \notin T'$, and not $e \prec e'$. Thus there is a serial trace $T$ of an execution of $P$ having the form $T = T_a e' T_b e T_c$. Then $T_a e' T_b$ does not contain any events from task $t$ (the task that executes event $e$ in program $P$) following $e$ and is therefore a prefix of a trace of $P'$. By Lemma 3.4 all traces of $P'$ contain the same events, contradicting the assumption that $e' \notin T'$.

□

In [HMW93] we proved that if $e_1 \prec e_2$ in the inferred program $P_T$ then whenever $e_2 \rightarrow e_1$ in some trace $T'$ of $P$, there is a race in $P$ that happens before either $e_1$ or $e_2$. This implies that a race detection system that uses the guaranteed ordering graph for $P_T$ will report a non-empty subset of the races in $P$ when run on the input used to produce $T$. This is point one from our informal notion of precise guaranteed ordering graph in the introduction. Theorem 3.6 (below) proves that if program $P$ contains no races and the ordering graph is exact for $P_T$, then no races will be reported for program $P$. This satisfies point two from our informal notion of precise guaranteed ordering graph in the introduction.

**Theorem 3.6:** *If program $P$ contains no data races when executed on input $I$, then $e \prec f$ in $P_T$ iff $e \prec f$ in $P$ on input $I$.*

**Proof**:
($\Leftarrow$) Assume to the contrary that $e \prec f$ in $P$ but $e \not\prec f$ in $P_T$. This means that there is a serial trace $T'$ of $P_T$ having the form $T' = T_a f T_b e T_c$. Because there are no data races in $P$ on input $I$, every execution of $P$ on input $I$ takes the same branches, in particular the same branches taken by $P_T$. Thus trace $T'$ is also a trace of $P$ which contradicts $e \prec f$ in $P$.

($\Rightarrow$) Assume to the contrary that $e \prec f$ in $P_T$ but $e \not\prec f$ in $P$. This means that there is a serial trace $T'$ of $P$ having the form $T' = T_a f T_b e T_c$. Again, since every execution of $P$ on input $I$ takes the same branches as taken by $P_T$, trace $T'$ must also be a trace of $P_T$ which contradicts $e \prec f$ in $P_T$.

□

# 4 Our Algorithm

The basic algorithm is quite similar to the one presented by Netzer and Ghosh [NG92]. It is a direct consequence of the observation captured in Theorem 3.5. It essentially finds what Netzer and Ghosh called the maximal valid execution sequence for each event and then adds the appropriate edges to the Guaranteed Ordering Graph.

The algorithm considers each task in turn. When considering task $i$ it adds all (non-transitive) edges leaving the events executed by task $i$. The algorithm artificially blocks each event in task $i$ and lets the remainder of program run as far as possible (the repeat loop). When no more events can be legally executed by other tasks, the algorithm adds an edge from the artificially blocked event in task $i$ to the first unexecuted event in each other task, as per Theorem 3.5. The algorithm then adds the edge from the artificially blocked event to the next event in task $i$, unblocks and executes the event that was artificially blocked, and then repeats the process with the next event executed by task $i$. A simple example showing a partial execution of the algorithm is shown in Figure 4.1.

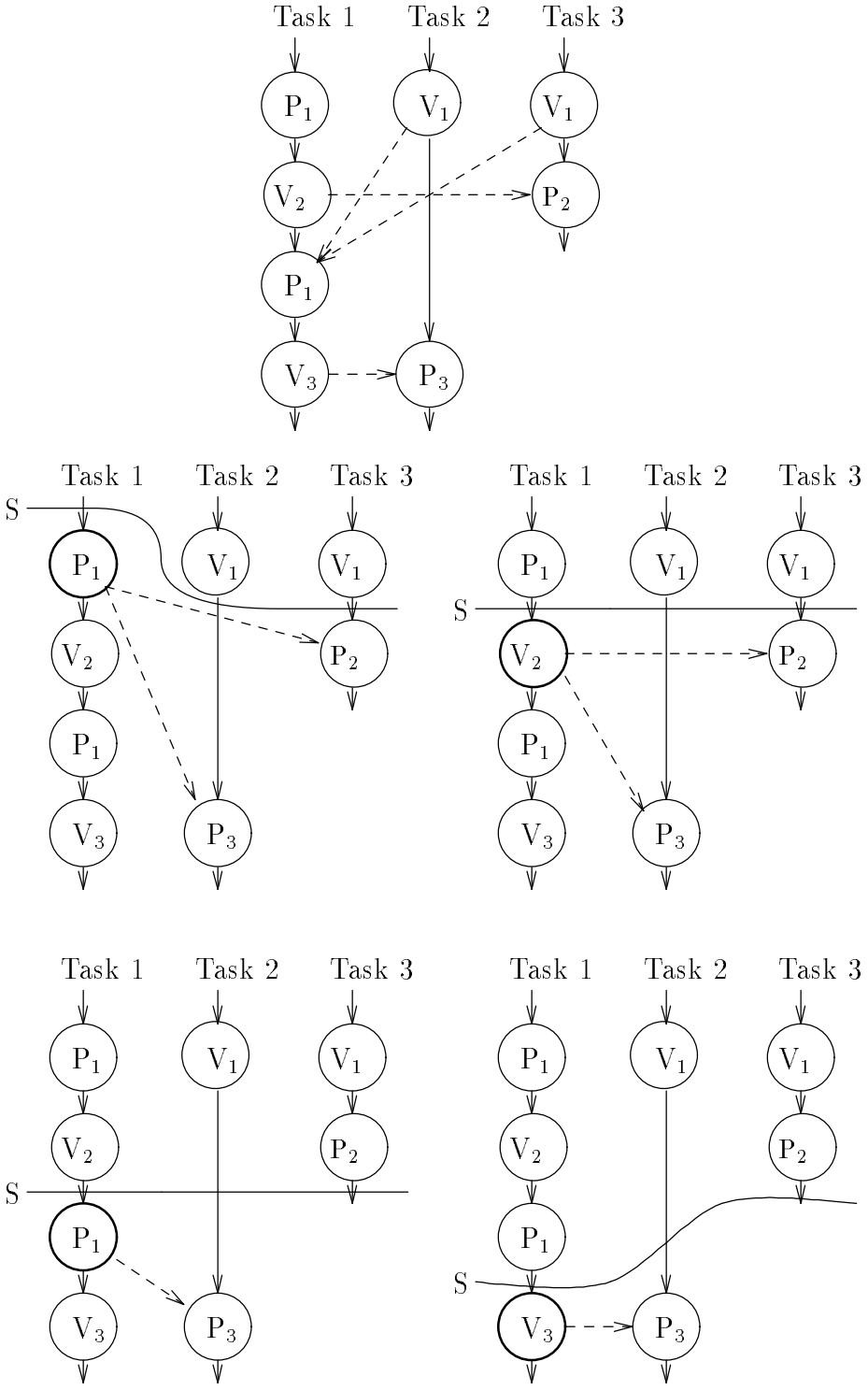


Figure 4.1: The top graph shows the (non-transitive) guaranteed orderings for a simple program. These are also the edges found by Algorithm 4.1. In the the remaining four graphs the line labeled S indicates the unique state reached when the event in the heavy circle is blocked. Each represents a different value for S in Algorithm 4.1 at the exit of the repeat...until loop with $i = 1$.

**Algorithm 4.1:** *Guaranteed Ordering Graph Algorithm*

Input: An MSSL program $P$ and a trace $T$ of program $P$.
Output: A graph representing a (reduced) partial order over the events in $T$.
A *state* is a vector indicating for each task the next event to be executed by that task.
We assume that the tasks are numbered from 1 to $t$.

```
Create an initial state S_0 with each task ready to execute it's first event.
for i = 1 to t {for each task}
    let S = S_0
    while S[i] ≠ terminated
        repeat
            for j = 1 to t {for each task}
                while j ≠ i and S[j] is unblocked
                    unblock any events waiting for S[j]
                    set S[j] to succ(S[j])
        until no more changes are possible
        for j = 1 to t
            if j ≠ i add an edge from S[i] to S[j]
        unblock any events waiting for S[i]
        add an edge from S[i] to succ(S[i])
        set S[i] = succ(S[i])
    end while
end for
```

If testing whether an event is blocked or unblocked can be done in constant time then Algorithm 4.1 has a worst case running time of $O(nt^2)$ where $n$ is the number of events in $T$ and $t$ is the number of tasks in $P$. The worst running time can be reduced to $O(nt)$ if a clever data structure can be used to identify the unblocked events which are about to be executed without iterating through the tasks. One way to do this for post/wait synchronization was presented in [NG92]. Appendix A contains an algorithm with worst case running time $O(nt)$ for the other sets of monotonic synchronization constructs described in the following section.

## 5 Programs in the class

We currently can identify three sets of synchronization constructs that can be used in MS programs:

- programs that use Post and Wait without clear (PW-programs),
- programs that use Message Passing where the Sender Names the Receiver and the receiver blocks but does not name the sender (MPSNR-programs), and

- programs that use counting semaphores with the restriction that each semaphore's P-operations are all executed by the same task (SRP-programs).

**Lemma 5.1:** *SRP-programs are MS programs.*

**Proof**: Assume to the contrary that some straight line SRP-program $P$ has two serial traces, $T_1$ and $T_2$ ending in different states. Because $P$ is a straight line program there must be some event in one of the traces that is not in the other. Assume without loss of generality that some event appearing in trace $T_1$ does not appear in trace $T_2$, and let event $e$ be the first event appearing in $T_1$ that does not appear in $T_2$. Then, trace $T_1$ can be written $T_a e T_b$ where all events in $T_a$ appear in trace $T_2$. In particular, the event $pred(e)$ (if it exists) appears in $T_2$, so event $e$ must be blocked at the end of trace $T_2$. Since V-operations are never blocked, $e$ must be a P-operation, say the $k$th P-operation on some signal. Since all P-operations on a given signal are executed by the same task, $k$ is a property of the program rather than the execution, and trace $T_2$ contains exactly $k-1$ P-operations on the semaphore. Sub-trace $T_a$, and thus trace $T_2$, contains at least $k$ V-operations on the semaphore unblocking $e$ in trace $T_1$. Therefore, event $e$ is not blocked at the end of trace $T_2$ and we have our contradiction.
□

**Lemma 5.2:** *PW-programs are MS programs.*

**Proof**: Assume to the contrary that some straight line PW-program $P$ has two serial traces, $T_1$ and $T_2$ that end in different states. Because $P$ is a straight line program there must be some event in one of the traces that is not in the other. Assume without loss of generality that some event appearing in trace $T_1$ does not appear in trace $T_2$, and let event $e$ be the first event appearing in $T_1$ that does not appear in $T_2$. Then, trace $T_1$ can be written $T_a e T_b$ where all events in $T_a$ appear in trace $T_2$. If $e$ is a Post then $pred(e)$ is in $T_2$ and $e$ could be appended to $T_2$. If $e$ is a Wait then $T_a$ contains a Post to satisfy the Wait and therefore so does $T_2$ so $e$ could be appended to $T_2$.
□

**Lemma 5.3:** *MPSNR-programs are MS programs.*

**Proof**: An MPSNR-program can be translated into an equivalent (with respect to synchronization) program by the following. Each receive in task R is replaced by V(R). Each Send-to(R) is replaced by V(R). The resulting program will be an SRP-program and the result then follows from Lemma 5.1.
□

## 6 Conclusions and Further Work

Because so many questions regarding races and event orderings are intractable [CS88, NM90], it is important to precisely identify those questions that can be answered in polynomial time. We have shown that if every straight line program using a set of synchronization operations always ends in the same state (monotonic synchronization), then the must-have-happened-before relation for any such program can be efficiently computed. This extends

the results of Netzer and Ghosh [NG92], who show how to efficiently compute the must-have-happened-before relation for straight line programs using post/wait synchronization (without clear).

Although few real programs are straight line, the execution trace of any program allows one to infer a straight line program. Rather than analyzing a parallel program directly, we analyze the straight line program inferred by a trace of the program and then generalize the results to the original parallel program. We have previously shown that an approximation to the must-have-happened-before relation can produce a race detection algorithm that reports a non-empty subset of the races in the original program. In this paper we have shown that when the must-have-happened-before relation (for the inferred program) is exact, then no races will be reported for the original program if it contains no races (on the same input).

There are also non-monotonic sets of synchronization constructs which allow the exact and efficient computation of the must-have-happened-before relation for straight line programs. One trivial example is properly nested lock/unlock operations. These operations are non-monotonic since if one task executes

lock 1; lock 2; unlock 2; unlock 1;

while another does

lock 2; lock 1; unlock 1; unlock 2;

then both tasks can either complete or deadlock. On the other hand, any of the tasks can run to completion before any other events are executed. Thus the only must-have-happened-before relationships are between events executed by the same task. Another example is the use of a single semaphore. Programs that use only one semaphore are not monotonically synchronized but a polynomial algorithm for computing precise ordering information has been found[LKN93]. An interesting open question is if there is a more interesting non-monotonic set of synchronization operations where the must-have-happened-before relation can be exactly and efficiently computed (for straight line programs).

Our characterization of monotonic sets of synchronization constructs may be difficult to apply in all cases. It is tempting to claim that monotonicity is equivalent to (or at least implied by):

whenever an event becomes unblocked then it remains unblocked.

Unfortunately there are some unusual sets of synchronization operations that meet this unblocking criteria but are not monotonic. One concrete example is the familiar P and V operations restricted as in SRP-programs (defined in Section 5) combined with a "double" operation which doubles the current count of unmatched V's on the semaphore. We are working towards simple requirements on the synchronization operations that are equivalent to (or at least imply) monotonicity.

**Acknowledgement**

This work was partially supported by a grant from the National Science Foundation (grant # CCR-9102635).

## References

[CS88] D. Callahan and J. Subhlok. Static analysis of low-level synchronization. In *Proc. Workshop on Parallel and Distributed Debugging*, pages 100–111, May 1988.

[HMW93] D. P. Helmbold, C. E. McDowell, and J. Z. Wang. Determining possible event orders by analyzing sequential traces. *IEEE Transactions on Parallel and Distributed Systems*, 1993. Also UCSC Tech. Rep. UCSC-CRL-91-36.

[LKN93] H-I. Lu, P. N. Klein, and R. H. B. Netzer. Detecting race conditions in parallel programs that use one semaphore. Technical report, Brown Univ., 1993.

[NG92] R. H. B. Netzer and S. Ghosh. Efficient race condition detection for shared-memory programs with Post/Wait synchronization. In *Proc. International Conf. on Parallel Processing*, 1992.

[NM90] R. H. B. Netzer and B. P. Miller. On the complexity of event ordering for shared-memory parallel program executions. In *Proc. International Conf. on Parallel Processing*, volume II, pages 93–97, 1990.

[NM91] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. *SIGPLAN Notices (Proc. PPOPP)*, 26(7):133–144, 1991.

[NR88] I. Nudler and L. Rudolph. Tools for efficient development of efficient parallel programs. In *First Israeli Conference on Computer Systems Engineering*, 1988.

## A Worst case $O(nt)$ algorithm for SRP-programs

The following algorithm has a worst case running time $O(nt)$ and computes the guaranteed ordering graph for programs using semaphore style synchronization with the restriction that all P-operations on the same semaphore occur in the same task. As noted in Section 5, the synchronization pattern of message passing with one way naming is similar to that of SRP-programs, and only cosmetic changes to the algorithm are required for these message passing programs.

The algorithm assumes that the tasks are numbered from 1 to $t$ and that succ($e$) is the next event executed by the same task that executed event $e$. The constant arrays first[$i$] and task[$x$] are initialized to contain the first event executed by each task $i$ and the task executing the P-operations on semaphore $x$ respectively. The array count[$x$] stores the number of excess V-operations on each semaphore $x$ and the array waiting[$j$] stores the blocked P-operation that task $j$ is waiting on (or $\emptyset$ if task $i$ is not blocked). The algorithm also uses a queue $Q$ containing the events which have been simulated, but whose successors have not yet been seen. Finally variable $e$-block stores the event in task $i$ which is artificially held up.

**Algorithm A.1:** *Optimized Guaranteed Ordering Graph Algorithm*

Input: A straight line SRP-program $P$ and a trace $T$ of program $P$.
Output: A graph representing a (reduced) partial order over the events in $T$.

```
main:
    for i = 1 to t { for each task }
        initialize count[x] to all zeros and waiting[j] to all ∅
        initialize Q to empty and e-block to first[i]
        for j = 1 to t { for each task }
            if (j ≠ i) then simulate (first[j])
        while (e-block ≠ terminated) do
            while Q not empty do
                dequeue (e)
                simulate (succ(e))
            for j = 1 to t
                if (j ≠ i) then add edge from e-block to waiting[j]
            simulate (e-block)
end main

Procedure simulate (e):
    if e is a P-operation on semaphore x then
        if (count[x] = 0) then { event e blocked }
            waiting[task[x]] := e
        else { event e is not blocked as count[x] ≥ 1 }
            enqueue(e)
            count[x] := count[x] - 1
    if e is a V-operation on semaphore x then
        enqueue(e)
        if (waiting[task[x]] = e') and (e' a P-operation on x) then
            {e' now unblocked }
            waiting[task[x]] := ∅
            enqueue(e')
        else { no P-operations on x currently blocked }
            count[x] := count[x] + 1
end simulate
```