A Conservative Data Flow Algorithm for Detecting All Pairs of Statements that May Happen in Parallel*

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

ABSTRACT

Information about which pairs of statements in a concurrent program can execute in parallel is important for optimizing and debugging programs, for detecting anomalies, and for improving the accuracy of data flow analysis. In this paper, we describe a new data flow algorithm that finds a conservative approximation of the set of all such pairs. We have carried out an initial comparison of the precision of our algorithm and that of the most precise of the earlier approaches, Masticola and Ryder's non-concurrency analysis [8], using a sample of 159 concurrent Ada programs that includes the collection assembled by Masticola and Ryder. For these examples, our algorithm was almost always more precise than non-concurrency analysis, in the sense that the set of pairs identified by our algorithm as possibly happening in parallel is a proper subset of the set identified by non-concurrency analysis. In 132 cases, we were able to use reachability analysis to determine exactly the set of pairs of statements that may happen in parallel. For these cases, there were a total of only 10 pairs identified by our algorithm that cannot actually happen in parallel.

*This research was partially supported by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory/IFTD under agreements F30602-94-C-0137 and F30602-97-2-0032, and by the National Science Foundation under Grants CCR-9407182 and 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.

1. INTRODUCTION

As the number and significance of parallel and concurrent programs continue to increase, so does the need for methods to provide developers with information about the possible behavior of those programs. In this paper, we address the problem of determining which pairs of statements in a concurrent program can possibly execute in parallel. Information about this aspect of the behavior of a concurrent program has applications in debugging, optimization (both manual and automatic), detection of synchronization anomalies such as data races, and improving the accuracy of data flow analysis [8].

The problem of precisely determining the pairs of statements that can execute in parallel is known to be *NP*-complete [12]. Most work in the area has therefore focused on finding methods for computing a *conservative approximation* to the set of pairs that can execute in parallel, that is, computing a set of pairs of statements that contains all the pairs that can actually execute in parallel but may also contain additional pairs. The goal is to find a useful tradeoff between precision and cost.

Several approaches have been proposed. Callahan and Subhlok [1] proposed a data flow algorithm that computes for each statement in a concurrent program the set of statements that must be executed before this statement can be executed (B4 analysis). Duesterwald and Soffa [2] applied this approach to the Ada rendezvous model and extended B4 analysis to be interprocedural. Masticola and Ryder proposed an iterative approach they called *non-concurrency analysis* [8] that computes a conservative estimate of the set of pairs of communication statements that can never happen in parallel in a concurrent Ada program. (The complement of this set is a conservative approximation of the set of pairs that may occur in parallel.) In that work, it is assumed initially that any statement from a given process can happen in parallel with any statement in any other process. This pessimistic estimate is then improved by a series of *refinements* that are applied iteratively until a fixed point is reached. Masticola and Ryder show that their algorithm yields more precise information than the approaches of Callahan and Subhlok and of Duesterwald and Soffa.

In this paper, we propose a new data flow algorithm for computing a conservative approximation of the set of pairs of statements that can execute in parallel in a concurrent Ada program. We have conducted a preliminary empirical comparison of our algorithm and non-concurrency analysis, using a set of 159 Ada programs that includes the programs used by Masticola and Ryder to evaluate non-concurrency analysis. For the purposes of this comparison, we took the complement of the set of pairs of statements identified by our algorithm as possibly occurring in parallel to get a conservative approximation of the set of pairs of statements that cannot occur together, as computed by non-concurrency analysis. On these programs, our algorithm finds all of the pairs identified by non-concurrency analysis in 150 cases; in 118 cases, our algorithm finds pairs that are not found by non-concurrency analysis. In 9 cases, non-concurrency analysis identifies pairs that are not found by our algorithm but, in all of these cases, our algorithm finds many more pairs that are not identified by non-concurrency analysis. For 132 cases, we were able to run a reachability analysis to determine exactly the pairs of statements that cannot occur in parallel. (In the remaining cases, the reachability analysis ran out of memory.) For these 132 programs, there were 5 cases in which our algorithm failed to find all the pairs of statements that cannot happen together, missing a total of 10 pairs.

The next section introduces the program model that we use and describes our algorithm. Section 3 briefly describes non-concurrency analysis and the relation between its program model and the model used for our algorithm. Section 4 presents the results of the comparison of our algorithm and non-concurrency analysis, and Section 5 discusses some conclusions and describes future work.

2. THE *MHP* ALGORITHM

2.1. Program representation

The program representation used in this work is the *trace flow graph* (TFG) introduced by Dwyer and Clarke [3,4]. This representation is conservative in the sense that it models a superset of all feasible program executions. Informally, TFGs are forests of *control flow graphs* (CFGs), one for each concurrent process, or *task*, in the program, with nodes and edges added to represent intertask communications. (If the code region represented by node n in one task contains a synchronization statement that can correspond to one represented by node m in another task, a new node is added with incoming edges from n and m and outgoing edges to all successors of n and m. This is illustrated in the figures.)

The TFG model deliberately does not specify exactly what kind of region in the task each CFG node represents, imposing only the weak restrictions that a region cannot contain more than one synchronization statement and that, if a region contains a synchronization statement, it must be the last statement in this region. This underspecification provides for greater flexibility of the model. For example, a CFG node can represent a single machine level instruction, a basic block, or even a set of paths from one synchronization point to another. We also add a unique *initial* node that has no incoming edges and has outgoing edges to the start nodes of all CFGs and a unique *final* node that has no outgoing edges and has incoming edges from the end nodes of all CFGs.

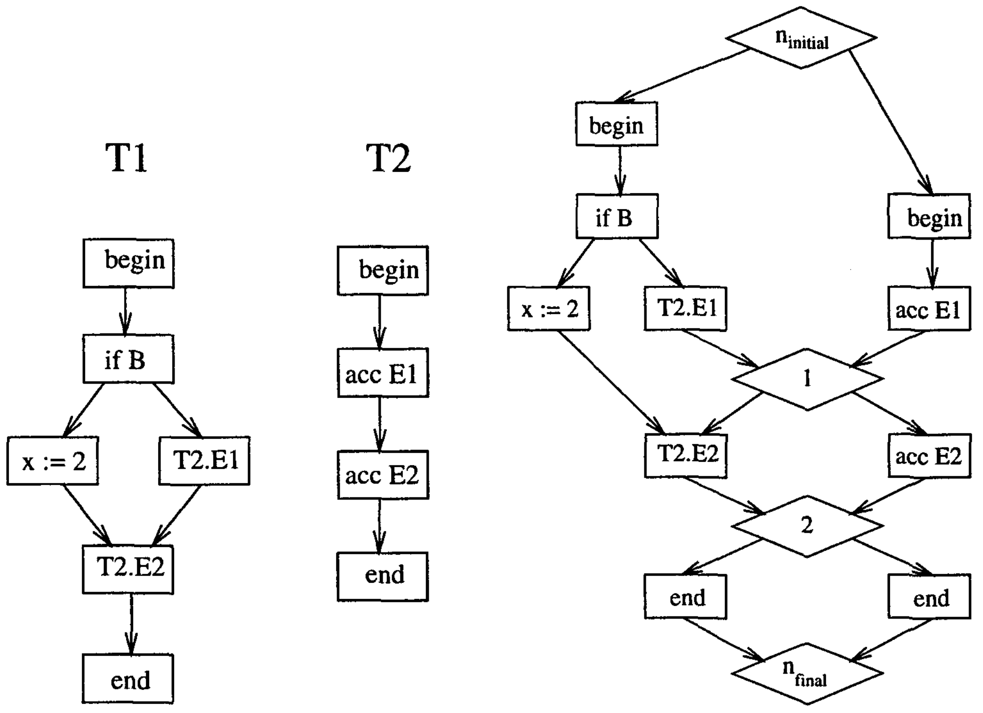
Formally, a TFG is a labeled directed graph $(N, E, n_{initial}, n_{final}, \mu)$, where N is the set of nodes, $E \subseteq N \times N$ is the set of edges, $n_{initial}, n_{final} \in N$ are unique initial and final nodes, and μ is a mapping from nodes to regions of code within tasks. The set of all nodes from the CFGs for all tasks forms the set of *local* TFG nodes, $LOCAL$. The elements of the set of non-local nodes, $COM = N \setminus (LOCAL \cup \{n_{initial}, n_{final}\})$, are *communication* nodes, which represent task rendezvous. In building a TFG from a collection of CFGs, the communication nodes are obtained by syntactic matching of synchronization statements. As a result, some nodes in COM may be unreachable, but our algorithm is capable of detecting some of these. For each node n, we let $Pred(n)$ and $Succ(n)$ be the sets of (immediate) predecessors and successors of n, respectively.

Figure 1(a) shows a program that consists of two communicating Ada tasks, Figure 1(b) shows the corresponding CFGs with nodes labeled with the corresponding Ada program statements, and Figure 1(c) gives the corresponding TFG. The local nodes in this TFG have the same labels as the corresponding nodes in the CFGs. Nodes 1 and 2 are communication nodes; node 1 represents the communication between the tasks at entry call `T2.E1`, and node 2 represents the communication at entry call `T2.E2`.

The TFG model offers a compact representation of the program's behavior. The number of local nodes in the TFG is linear in the number of program statements. A communication node is created for every syntactically matching pair of call and accept statements, so, for example, the local node corresponding to a single `call(T2.E)` statement in task `T1` would share a separate communication node successor with each `accept E` statement in task `T2`. In the worst case,

```
task body T1 is
begin
  if B then
    x:= 2;
    T2.E1;
  end if;
  T2.E2;
end T2

task body T2 is
begin
  accept E1;
  accept E2;
end T2;
```



(a) Code (b) CFGs (c) TFG

Figure 1: A TFG example

this results in the number of communication nodes being quadratic in the number of program statements. However, this blow-up in the number of communication nodes does not seem common in practice. Our experimental results support this hypothesis.

Given a pair (m, n) of nodes in a TFG, we are interested in determining whether, on some computer system, the program represented by the TFG has an execution in which code corresponding to a statement in the task region represented by m executes at the same time as code corresponding to a statement in the task region represented by n. (For the sake of brevity, in the rest of the paper we will use the phrase "node n executes" to mean "an instruction from the task region represented by node n executes".) If there is such an execution, we say that m and n ***may happen in parallel***, and define $MHP_{perf}(m, n)$ to be true. This definition of the MHP_{perf} relation identifies the "ideal" set of pairs of statements that may execute in parallel. The algorithm presented in this paper computes a conservative approximation *MHP* to MHP_{perf}.

2.2. The *MHP* algorithm

In this section we give the detailed description of the *MHP* algorithm and state the major results about its termination, conservativeness, and worst-case time bound. Rather than using the lattice/function space view of data flow problems [5], we give data flow equations for TFG nodes. This is done for two reasons. First, it makes explanations and especially proving properties of this algorithm more intuitive. Second, one aspect of the algorithm precludes its representation as a purely forward- or backward-flow data flow problem or even as a bidirectional [10] data flow problem. We conclude the description of the algorithm by giving pseudo-code for its worklist version.

Our algorithm associates three sets with each node n of the TFG: $GEN(n)$, $IN(n)$, and $M(n)$. The set $M(n)$ is the current approximation to the set of nodes that may happen in parallel with n, while $GEN(n)$ represents the nodes we can place in the approximation based on information local to n and $IN(n)$ represents the nodes we can place in the approximation using information propagated from the predecessors of n. Initially, all three sets for all nodes are empty. These sets are repeatedly recomputed until the algorithm reaches a fixed point and the sets do not change. At this point set $M(n)$ represents a conservative overestimate of nodes with which node n may execute in parallel.

In addition to these three sets, we assign a ***Reach*** bit to each communication node. This bit is initially set to `false`. Its value is set to `true` if, on some iteration, each of its two local predecessors belongs to the

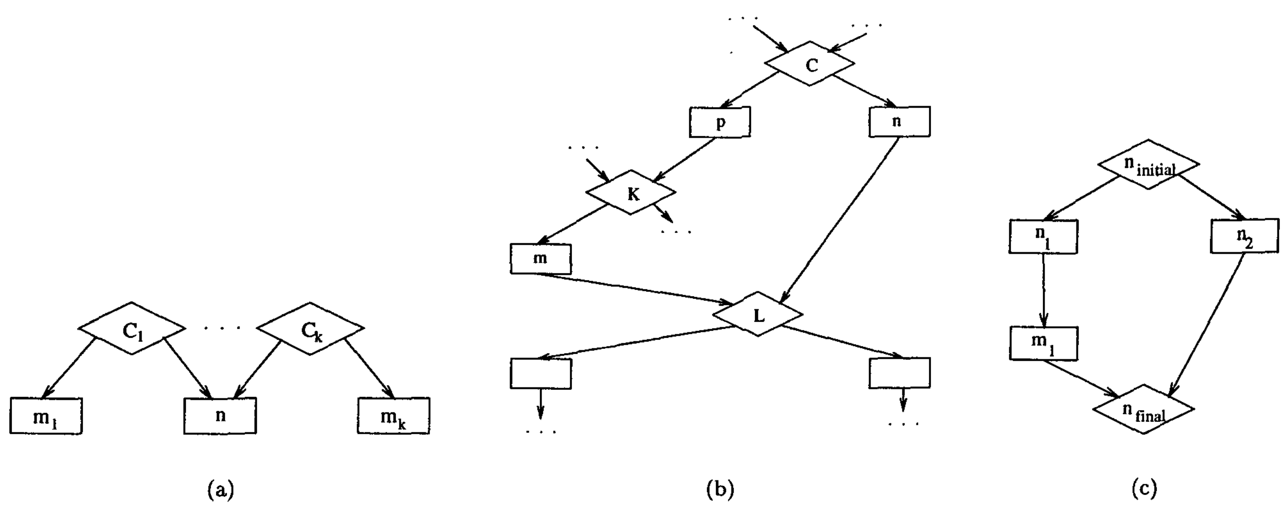


Figure 2: Illustrations for the *MHP* equations

M set of the other. Intuitively, a task rendezvous represented by a communication node can take place only if both tasks are ready to participate in it. Until the *Reach* bit of a communication node is set to `true`, the algorithm assumes that the task synchronization represented by this node is not possible. If the *Reach* bit of a communication node is still `false` after the algorithm reaches a fixed point, this means that the task communication represented by it is not possible on any execution of the program.

The sets *GEN* and *IN* are computed on each iteration of the algorithm as follows. If n is a local node, let P be the set consisting of $n_{initial}$, if n is a successor of $n_{initial}$, and all communication nodes C that have n as a successor and have $Reach(C)$ set to `true`. Then $GEN(n) = \left(\bigcup_{p\in P} Succ(p)\right) \setminus \{m \mid m \text{ is in the same task as } n\}$. Informally, $GEN(n)$ is the set of local nodes m such that m and n are both successors of a reachable communication node, or of the initial node. The idea is that, if a local node is a successor of a reachable communication node, it may happen in parallel with other successors of this communication node since both tasks participating in the communication can execute immediately after the communication. For example, in Figure 2(a), after rendezvous C_1 is executed, nodes m_1 and n may happen in parallel. If n is a communication node, $GEN(n) = \emptyset$.

For a local node n, we put $IN(n) = \bigcup_{p\in Pred(n)} M(p)$, while if n is a communication node, we put

$$IN(n) = \begin{cases} \bigcap_{p\in Pred(n)} M(p) & \text{if } Reach(n) \\ \emptyset & \text{otherwise.} \end{cases}$$

Here the idea is that, since tasks can execute at varying rates, a local node that may execute in parallel with another node may also execute in parallel with all local successors of that node. A communication node, however, can execute only when both of its predecessors have executed, and so may not execute in parallel with a node that cannot execute in parallel with *both* of its predecessors. Figure 2(b) provides an illustration. Suppose that nodes n and p may happen in parallel (i.e., that node C is reachable), and nodes m and p may not happen in parallel. Since node L can happen only after both m and n happened, it may not happen in parallel with node p. Note that by construction a communication node can never have nodes in its *IN* set from the two tasks whose rendezvous it represents.

On each iteration, we set $M(n) = IN(n) \cup GEN(n)$. Up to this point the algorithm is a standard forward-flow data flow algorithm [5]. However, after computing *GEN*, *IN*, and *M* 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 $n_1 \in M(n_2)$. Figure 2(c) illustrates this necessity: without this additional step the *M* sets of nodes n_1 and m_1 are $\{n_2\}$ ($GEN(m_1) = \{n_2\}$ and $IN(m_2) = \{n_2\}$), but the *M* set of n_2 is $\{n_1\}$ ($GEN(n_2) = \{n_1\}$). Thus, $n_2 \in M(m_1)$ holds but $m_1 \in M(n_2)$ does not and so the symmetry step in necessary to put m_1 in $M(n_2)$.

In Figure 3, we give a worklist version of the *MHP* algorithm. Although steps (12)-(14) do not allow casting the algorithm in the general data flow algorithm form and using the standard complexity results [7] directly, we can show that the algorithm has polynomial worst case bound, as stated below in Theorem 4.

To conclude the discussion of the *MHP* algorithm, we state some results about its termination, conservative-

Input: A TFG $(N, E, n_{initial}, n_{final}, \mu)$

Output: $\forall n \in N$: a set $MHP(n)$ of TFG nodes such that $\forall m \notin MHP(n)$, m may not happen in parallel with n.

Initialization: *The M sets for all nodes are initially empty, and the worklist W initially contains start nodes for all tasks in the program.*

For each $n \in N$, set $M(n) = \emptyset$.

Set $W = (n_1, ..., n_k)$, where $\bigcup_{i=1}^{k} n_i = \{n \mid (n_{initial}, n) \in E\}$

Main Loop: *We evaluate the following statements repeatedly until $W = \emptyset$*

```
(1)   $n :=$ the first element from $W$
(2)   $W := W \setminus \{n\}$
(3)   $M_{\text{old}} := M(n)$
(4)   if $n \in COM$ then
(5)       $\{p_1, p_2\} := Pred(n)$
(6)       $Reach(n) := (p_1 \in M(p_2))$
(7)       if $Reach(n)$ then
(8)           $M(n) := M(p_1) \cap M(p_2)$
          end if;
      else
(9)       Compute $GEN(n)$
(10)      $M(n) := \bigcup_{p \in Pred(n)} M(p) \cup GEN(n)$
      end if;
(11)  if $M_{\text{old}} \neq M(n)$ then
(12)      For each $m \in (M(n) \setminus M_{\text{old}})$
(13)          $M(m) := M(m) \cup \{n\}$
(14)          $W := W \cup Succ(m)$
(15)      $W := W \cup Succ(n)$
      end if;
Finalization:
      For each $n \in N$
(16)      $MHP(n) := M(n)$
```

Figure 3: *MHP* algorithm

ness, and polynomial-time boundedness.

Theorem 1 (Termination). *Given a TFG for a concurrent program, the worklist version of the* MHP *algorithm will eventually terminate.*

Termination follows easily from the finiteness of the information that can appear in the M sets of all nodes in the TFG and from the fact that the M sets of all nodes increase monotonically.

Theorem 2 (Correctness). *After the* MHP *algorithm terminates,* $M(n) = GEN(n) \cup \bigcup_{q \in Pred(n)} M(q)$ *for every reachable local node n, i.e., the algorithm finds a fixed point of the data flow equations.*

The fact that the algorithm computes a fixed point follows from the observation that, whenever an $M(n)$ is changed, all nodes directly affected by the change are placed on the worklist.

Theorem 3 (Conservativeness). *For all* $n_1, n_2 \in N$, $MHP_{perf}(n_1, n_2) \Rightarrow n_1 \in MHP(n_2)$.

The proof of this result is based on a case-by-case examination of all configurations of nodes n_1 and n_2 in the TFG.

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

To prove this, we construct an optimized version of the worklist algorithm which limits the amount of information passed among the nodes in the TFG by sending each node from the M set of a given node to each of its successors only once. Then we prove that this efficient algorithm computes exactly the same information as the *MHP* algorithm in Figure 3 and show that the complexity of the efficient algorithm is $\mathcal{O}(|N|^3)$.

The idea of the efficient algorithm is that, if node m is one of control predecessors of node n, each node in $M(m)$ should only be inserted in $M(n)$ once. This is achieved by defining an additional set OUT for each node. A node is placed in $OUT(m)$ if and only if it is in $M(m)$ and has never been placed in $OUT(m)$ before. The IN set for a local node is set equal to the union of the OUT sets of all its predecessors. Since each local node has $\mathcal{O}(|N|)$ predecessors, and the number of nodes that can be put in the OUT set of each of the predecessors in the course of the algorithm is $\mathcal{O}(|N|)$, the number of times a node is added to the IN set of each local node is $\mathcal{O}(|N|^2)$. Combined over all local nodes, there are at most $\mathcal{O}(|N|^3)$ insertions in IN (and thus M) sets. Because each of the communication nodes in the graph has exactly two predecessors, computation of IN sets for communication nodes can be based on the M sets of their predecessors, exactly as in the algorithm in Figure 3, and thus takes $\mathcal{O}(|N|)$ operations. Each node in the graph may be placed on the worklist at most $\mathcal{O}(|N|^2)$ times, and so the overall complexity of computing M sets for communication nodes is $\mathcal{O}(|N|^3)$.

The GEN sets for all nodes in the graph can be largely precomputed and only modified in the course of the algorithm as communication predecessors of local nodes become reachable. This precomputation takes $\mathcal{O}(|N|^3)$ and the modification is $\mathcal{O}(|N|)$ for each local node for the course of the algorithm.

Combining the complexities for the parts of the algorithm, we obtain the cumulative complexity of $\mathcal{O}(|N|^3)$.

We note that, in the worst case $|N|$ is itself quadratic in the number of statements in the program (due to the introduction of communication nodes). In practice, however, $|N|$ is usually a small multiple of the number of program statements. In Section 4, we report on the relation between the number of TFG nodes and the size of the program for our sample of 159 programs.

3. COMPARING NON-CONCURRENCY ANALYSIS WITH THE *MHP* ALGORITHM

This section introduces the most precise of the previous approaches for computing the *MHP* information, Masticola and Ryder's non-concurrency analysis. Since the program model used by this approach is different from TFG, we describe the technique for creating TFGs automatically from the non-concurrency graphs. Finally, since the *MHP* algorithm computes pairs of nodes that may happen in parallel and non-concurrency analysis computes pairs of nodes that cannot happen in parallel, we present a mapping between these two sorts of data. This mapping allows us to compare the information computed by the two approaches.

3.1. Non-concurrency Analysis

Non-concurrency analysis computes *can't happen together (CHT)* information, which is the opposite of what the *MHP* algorithm computes. The model of the program used in this approach is the *sync graph*, where each node represents a number of control paths in a task that end in a single synchronization point. Possible rendezvous are represented as *hyperedges*, connecting the synchronizing paths. Initially it is assumed that a given node can happen together with any of the nodes in the other tasks. Four *CHT refinements* are then applied, in arbitrary order, until a fixed point is reached. The four refinements are *pinning analysis*, *B4 analysis*, *RPC analysis*, and *critical section analysis*. The complexity of each of the four refinements is $\mathcal{O}(|N_{sync}|^3)$, and the complexity of the overall approach is $\mathcal{O}(|N_{sync}|^5)$, where N_{sync} is the set of sync graph nodes [9]. The number of sync graph nodes is proportional to the number of statements in the program.

3.2. Deriving TFGs from Sync Graphs

In order to compare information computed on sync graphs and TFGs, we construct a special *restricted trace flow graph* (RTFG) from a sync graph. Here we give only a sketch of this construction. We create a single local RTFG node for each sync graph node, except for those nodes in the sync graph that represent entry calls to accept statements with bodies, for which two local RTFG nodes are constructed. One RTFG node represents the execution of the caller task before the callee task accepts the call. The second RTFG node represents the state of the calling task while the accept body executes. Similarly, a single communication RTFG node is created for each hyperedge that models an entry call to an accept statement without a body and two communication nodes are created for each hyperedge that models an entry call to an accept statement with a body.

Figure 4 gives an example. The sync graph in Figure 4(b) models the communication structure of the simple program in Figure 4(a). The hyperedge representing the call to entry E, made by task T1, is shown as a dashed line, and the wavy line represents the subgraph corresponding to the body of the accept statement in T2. The RTFG derived from this sync graph is shown in Figure 4(c). The matching sync graph and RTFG nodes are labeled with the same numbers. The node labeled $1'$ in the RTFG represents the second local node created for the sync graph node 1.

3.3. Mappings between the information computed by the two approaches

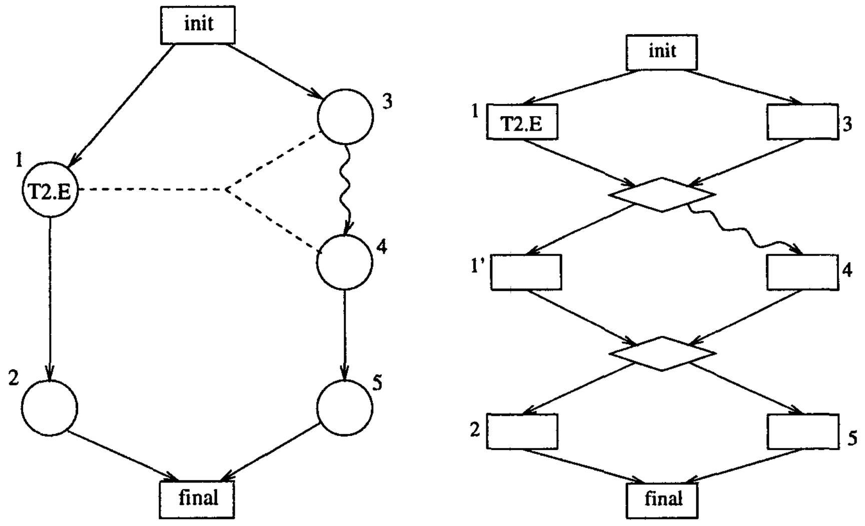
The algorithm for constructing RTFGs from sync graphs provides us with a mapping $\mu : N_{sync} \to 2^N$, where N_{sync} is the set of nodes in the sync graph and N is the set of nodes in the corresponding RTFG. We define a function $\mu^{-1} : 2^N \to 2^{N_{sync}}$ by setting $\mu^{-1}(S) = \{\tilde{n} \mid \mu(\tilde{n}) \cap S \neq \emptyset\}$.

Using these mappings, we can "translate" the *MHP* information from the RTFG to the corresponding sync graph by mapping the *MHP* set computed for a node n in the RTFG to the corresponding node in the sync graph. In cases where a sync graph node $\tilde{n}$ has two corresponding RTFG nodes n_1 and n_2, $MHP(\tilde{n})$ is defined as the union of the two translated sets $MHP(n_1)$ and $MHP(n_2)$. In general, $MHP(\tilde{n}) = \bigcup_{n \in \mu(\tilde{n})} \mu^{-1}(MHP(n))$.

The result is that each node in the sync graph has a *CHT* set and an *MHP* set associated with it. To compare these sets, we must take the facts that, for each node $\tilde{n}$ in the sync graph, $\tilde{n} \notin MHP(\tilde{n})$ and $\tilde{n} \notin CHT(\tilde{n})$ into account. For any function $A\colon N_{sync} \to 2^{N_{sync}}$, let $A^+(\tilde{n}) = A(\tilde{n}) \cup \{\tilde{n}\}$. Then $CHT(\tilde{n})$ computed by non-concurrency analysis corresponds to $\overline{MHP^+(\tilde{n})}$ computed by the *MHP* algorithm, where the bar indicates the complement. Thus, to compare the precision of the two techniques, we compare the sets $CHT(\tilde{n})$ and $\overline{MHP^+(\tilde{n})}$. One tech-

```
task body T1 is
begin
  T2.E;
end T1;

task body T2 is
begin
  accept E do
    ...
  end E;
end T2;
```



(a) Code (b) Sync graph (c) RTFG

Figure 4: Example of RTFG construction

nique is more precise than the other if, for each node $\tilde{n}$, the set computed by the former contains the set computed by the latter.

3.4. Theoretical comparison

We compared the theoretical precision of information computed by the *MHP* algorithm and non-concurrency analysis. Specifically, we compared the *MHP* algorithm to each of the four refinements used by non-concurrency analysis, attempting to prove or disprove that our algorithm is more precise than this refinement. We say that the *MHP* algorithm ***subsumes*** a refinement if, given that the *MHP* information was at least as precise as the *CHT* information before the refinement (i.e., that $CHT(\tilde{n}) \subseteq \overline{MHP^{+}(\tilde{n})}$ for all $\tilde{n}$), that is still the case after the refinement. Due to space limitations, we briefly state the results of this comparison without proof.

We were able to prove that the *MHP* algorithm subsumes the pinning and B4 refinements of the non-concurrency approach. On the other hand, we found counterexamples showing that the *MHP* algorithm does not subsume the critical section and RPC refinements. The *MHP* algorithm can be improved to take advantage of critical section regions[1]. However, the resulting algorithm is more complicated than the one presented in this paper and its worst-case complexity is $\mathcal{O}(|N|^5)$. Our initial evaluation of the performance of the *MHP* algorithm, discussed in the next section, seems to indicate that the gain in precision may not warrant this added complexity. We plan to investigate these trade-offs in our future work.

[1]In the TFG model the subgraph corresponding to the RPC structure is just a special case of the critical section structure. Therefore, this extension of the *MHP* algorithm also takes advantage of the information about remote procedure calls.

4. EXPERIMENTAL RESULTS

We measure the precision of the information computed by a technique in terms of the set of pairs of nodes in the sync graph that this technique determined cannot happen in parallel. We write P_{NCA} for the set of *CHT* pairs found by non-concurrency analysis and P_{MHP} for the set of *CHT* pairs found by the *MHP* algorithm.

Stephen Masticola graciously provided us with his implementation of non-concurrency analysis, written in C. We used this for our experiments, together with our own implementation of the *MHP* algorithm, written in Java. In addition, we wrote a reachability tool to find all reachable program states of the RTFG model, also in Java. Although the reachability tool runs out of memory for some of our test programs, in the cases where it ran successfully, it determined the "ideal" set of pairs of nodes that can happen in parallel. Assuming that no data sharing and no unreachable code exists in the program, this set is equal MHP_{perf}. Given this set, we computed CHT_{perf}, the "ideal" set of pairs of nodes that cannot happen together. We ran the non-concurrency tool on a Sun Sparc 10 with 32 MB of memory, and the *MHP* tool and the reachability tool on an AlphaStation 200 with 128 MB of memory. (The non-concurrency tool would not compile on the AlphaStation, which is our primary platform.)

We used a sample of 159 Ada programs, including the suite of 138 programs Masticola and Ryder used

$\lvert N_{sync}\rvert$	$\lvert P_{NCA}\rvert$	$\lvert P_{MHP}\rvert$	$\lvert CHT_{perf}\rvert$	$\lvert P_{MHP} \setminus P_{NCA}\rvert$	$\lvert P_{NCA} \setminus P_{MHP}\rvert$	$\lvert CHT_{perf} \setminus P_{NCA}\rvert$	$\lvert CHT_{perf} \setminus P_{MHP}\rvert$	*NCA* time	*MHP* time	reach. time
699	72373	98103		25990	260			277.54	1860.41	
55	334	361	362	28	1	28	1	2.81	0.32	3.14
88	1039	1155	1157	118	2	118	2	10.67	1.04	140.90
194	668	815		177	30			57.62	26.30	
232	800	1025		261	36			90.15	48.04	
97	953	1282		337	8			35.16	1.37	
44	345	355	356	11	1	11	1	0.89	0.30	0.35
268	15395	17310	17312	1917	2	1917	2	26.58	45.14	19.54
56	373	423	427	54	4	54	4	2.12	0.45	1.72

Table 1: Data for the 9 cases where non-concurrency analysis found some pairs that the *MHP* algorithm did not

in their experiments with non-concurrency analysis. Most of the remaining programs are examples drawn from the concurrency literature, such as the dining philosophers and the gas station. Of the 159 programs, 25 did not have loops. The sizes of the programs range from only a few lines of code to several thousand lines. This program sample contains several groups of programs representing different sizes and variations of the same basic example and actually contains approximately 90 significantly different examples. It is, of course, unlikely that this sample of relatively small programs is representative of concurrent Ada programs in general, but our results provide some initial data indicating that the *MHP* algorithm is very often more precise than non-concurrency analysis.

In the following discussion of the results, we separate the program sample into three subsets, which we discuss separately. First, we consider the 25 programs without loops. For all of these programs, which come from the Masticola-Ryder collection, the *MHP* algorithm found all the *CHT* pairs found by non-concurrency analysis. Second, we describe our results for the 9 programs in which non-concurrency analysis detected some *CHT* pairs not found by our *MHP* algorithm. Finally, we describe the results for the remaining 125 programs, those with loops for which the *MHP* algorithm found all the *CHT* pairs found by non-concurrency analysis. The focus of our discussion is on the detection of *CHT* pairs by the two approaches. We do comment briefly on the execution times for non-concurrency analysis and our *MHP* approach, but these times do not have much significance. Neither we nor Masticola and Ryder aimed to maximize the speed of the implementations. In addition, non-concurrency analysis was implemented in C, a compiled language, and the *MHP* algorithm was implemented in Java, an interpreted language. Finally, as mentioned above, the tools were run on different machines with different operating systems, clock speeds, and memory sizes. Thus, we view the comparison of the precision of the two approaches as the primary goal of this experiment.

4.1. Programs without loops

We realize that the programs without loops are not likely to be realistic examples, and so we consider them separately from other programs. The reachability analysis was completed successfully for all but one of the 25 programs without loops, and in all such cases the *MHP* algorithm found all pairs found by reachability (so $P_{MHP} = CHT_{perf}$ for all RTFG nodes). In 8 cases, the *MHP* algorithm found a small number of pairs that non-concurrency analysis did not, with the average ratio $|P_{MHP}|/|P_{NCA}|$ of 1.01. The average timing ratio (NCA time)/(*MHP* time) was 1.82, with most running times for both tools well under a second.

4.2. Programs where non-concurrency analysis found pairs that the *MHP* algorithm did not

Non-concurrency analysis found some *CHT* pairs not found by the *MHP* algorithm in 9 of the 159 cases we ran. The complete data for these cases are presented in Table 1. The first column of this table shows the program size in terms of the number of nodes in the sync graph. The next three columns give the number of pairs of nodes that cannot happen together, as found by the three different methods. The fifth column gives the number of pairs found by the *MHP* algorithm that were not found by non-concurrency analysis, while the sixth column gives the number of pairs found by non-concurrency analysis but not by the *MHP* algorithm. The seventh and eighth columns give, for the 5 cases that our reachability tool could handle, the number of nodes in CHT_{perf} that were not found by non-concurrency analysis and by the *MHP* algorithm, respectively. Finally, the last three columns show the time used by each of the analysis methods; times are in seconds and include both user and system time. An interesting observation is that, for the 5 cases in which our reachability tool could determine CHT_{perf}, although neither the *MHP* algorithm nor non-concurrency analysis found all possible pairs, the combination of the two approaches was as precise as reachability.

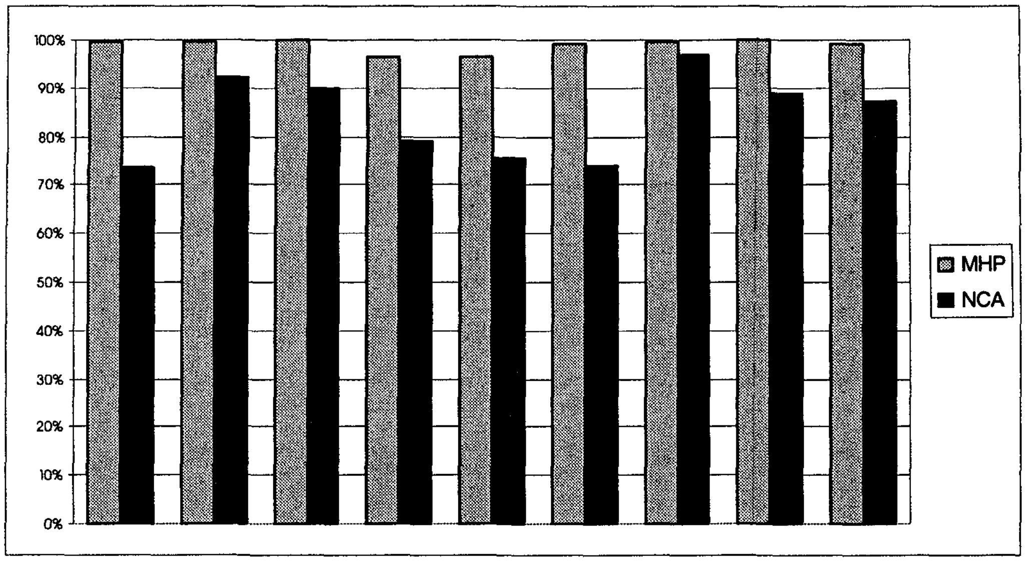


Figure 5: Precision comparison for the 9 cases where non-concurrency analysis found some pairs that the *MHP* algorithm did not

Figure 5 compares the precision of the two approaches by comparing the total number of *CHT* pairs found by each of them to the number of *CHT* pairs in the union $P_{NCA} \cup P_{MHP}$. As just noted, this union is equal to CHT_{perf} in the 5 cases that our reachability tool could handle. Note that in all cases the *MHP* algorithm outperformed non-concurrency analysis in terms of the total number of *CHT* pairs found.

4.3. The other 125 programs

The remaining 125 programs are those that have loops and where the *MHP* algorithm found all *CHT* pairs that non-concurrency analysis did. Of these, the reachability tool ran in 102 cases. For all of these 102 cases in which we were able to determine CHT_{perf}, the *MHP* algorithm found all the pairs in CHT_{perf}. Non-concurrency analysis found all the pairs in CHT_{perf} in only 22 cases.

Of these 125 programs, there were 101 cases in which the *MHP* algorithm found some pairs that were not found by non-concurrency analysis (in the remaining 24 cases, the *MHP* algorithm and non-concurrency analysis found exactly the same pairs). Figure 6 plots the ratio $|P_{MHP}|/|P_{NCA}|$ against the program size, measured as the number of nodes in the sync graph. The average precision ratio $|P_{MHP}|/|P_{NCA}|$ was 1.41 and the average timing ratio (NCA time)/(*MHP* time) was 2.94. The running times of both tools were under 4 minutes for all programs.

4.4. The number of RTFG nodes

In addition to comparing the performance of the two approaches, we examined the question of potential quadratic blow-up in the number of RTFG nodes. We plot the number of sync graph nodes against the number of RTFG nodes in Figure 7. The figure also shows the least-squares regression line, which has a slope of 1.84. The correlation coefficient is .984. This sample of programs thus offers strong support for the hypothesis that, in practice, the number of RTFG nodes depends linearly on the number of sync graph nodes. Since the size of the sync graph is linear in the number of program statements, the same appears to be true for RTFGs.

5. CONCLUSION

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 program that may execute in parallel. Theoretically, neither non-concurrency analysis nor our *MHP* algorithm has a clear advantage in precision. However, based on our experimental data, the *MHP* algorithm often is able to determine the pairs of statements that may execute in parallel more precisely than non-concurrency analysis.

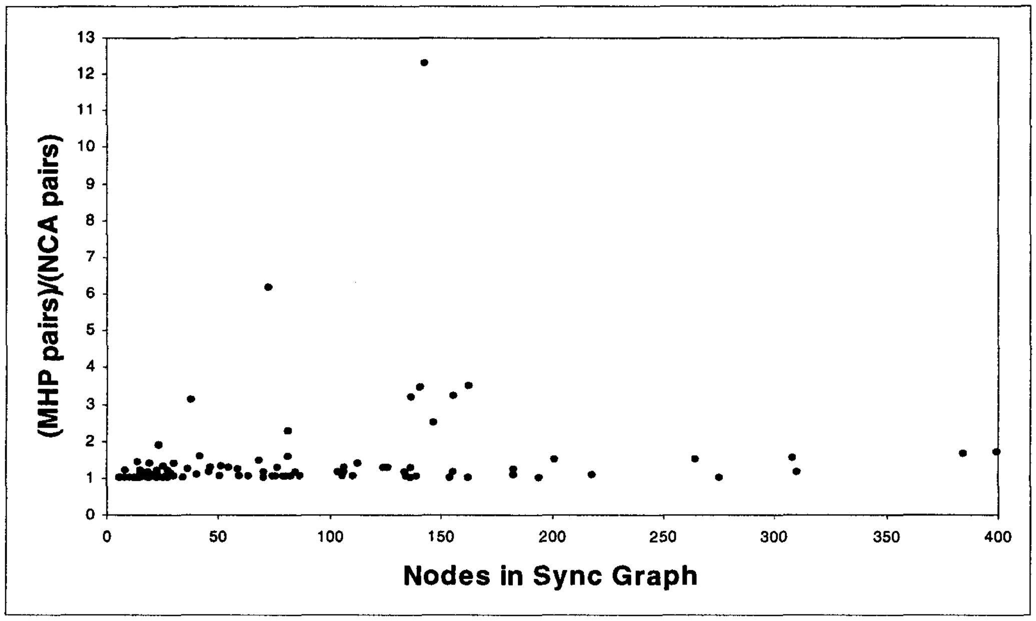


Figure 6: The precision ratio $|P_{MHP}|/|P_{NCA}|$ for the 125 programs with loops where the *MHP* algorithm found all *CHT* pairs found by non-concurrency analysis

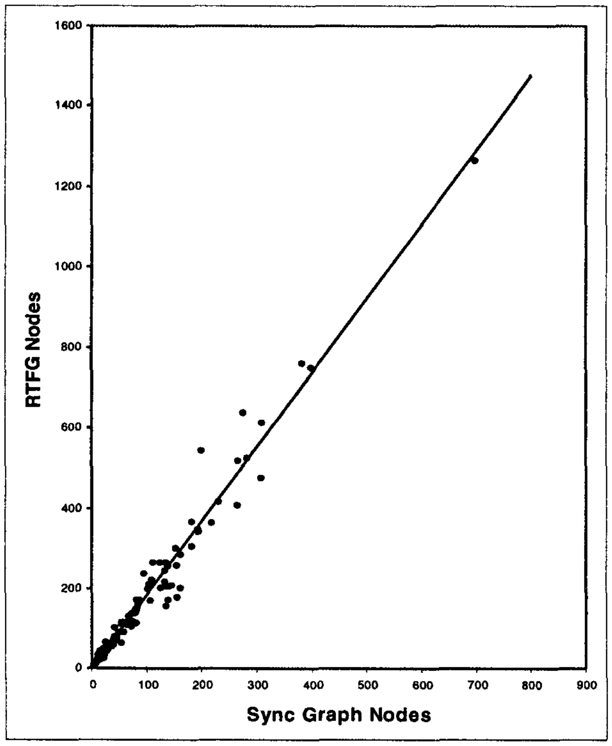


Figure 7: Least-squares fit of the number of RTFG nodes to the number of sync graph nodes

As a part of our experiments, we compared the precision of the *MHP* algorithm with the precision of a technique based on the exhaustive exploration of the program state space. While this reachability technique, being exponential in the program size, is not practical in general, with its help we were able to compute "perfectly" precise information for many examples. For these examples, the information computed by the *MHP* algorithm was remarkably close to that of the reachability technique.

At present, the *MHP* algorithm is being used as part of the FLAVERS tool [3, 11] for data flow analysis of concurrent programs.

In the future, we plan to extend the *MHP* algorithm to apply to programs containing procedure and function calls without using inlining. Even in its current form, the *MHP* algorithm can be easily used to support a limited form of interprocedural *MHP* analysis, with the restriction that procedures may not contain task entry calls. Under this restriction, the *MHP* sets computed for procedure call nodes are sufficient to determine the *MHP* sets for all nodes in this procedure. Thus, if n is a call node for procedure P, then any node in the body of P may happen in parallel with any node in *MHP*(n), computed the *MHP* algorithm. Special care must be taken when there is a possibility that a procedure may be called by more than one task, in which case executions of multiple instances of this procedure may overlap in time. In this case, unlike task nodes, the *MHP* sets of nodes from the procedure will contain other nodes from the same procedure. To determine whether this might happen, we have to check whether any of the call nodes to P is in the *MHP* set of any of the other call nodes to this procedure (this has to be done recursively for nested procedure calls), in which case the *MHP* sets of all nodes in P must contain all nodes in P.

In the case of procedures containing entry calls, we plan to use a context-sensitive approach, extending the TFG model to include procedure *call* and *return* edges, similar to the approach of [6], and modifying the *MHP* algorithm accordingly.

In addition, we plan to implement an algorithm that improves the precision of the *MHP* algorithm by taking advantage of information about regions in the program that can only execute in a mutually exclusive fashion, in a way similar to the critical section analysis refinement of non-concurrency analysis. Then we plan to carry out a careful comparison of the performance of this improved algorithm with that of the algorithm presented in this paper and of the non-concurrency approach. The initial hypothesis, which seems to be supported by this work, is that in practice the improved algorithm will be only marginally more precise than the current algorithm. We hope to perform these experiments for a larger program sample with more realistic programs and to evaluate the trade-offs of precision and cost added by the improved algorithm.

Finally, we are working on an *MHP* algorithm for concurrent Java programs. The differences in the way communications between threads of control are realized in Ada and Java imply different program models. While we are able to use the same general principle for Java as the one we introduce in this paper for Ada, there are a number of significant changes in the data flow equations used by the algorithm for Java. It will be interesting to see if the practical precision of the *MHP* algorithm depends on the differences in communication mechanisms of the different concurrent languages.

Acknowledgments

We thank Stephen Masticola and Barbara Ryder for graciously providing us with their non-concurrency analysis tool and a set of sample Ada programs that we used in our experiments. We are also grateful to Lori Clarke for helpful suggestions on this work.

References

[1] 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.

[2] 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.

[3] M. Dwyer. *Data Flow Analysis for Verifying Correctness Properties of Concurrent Programs.* PhD thesis, University of Massachussetts, Amherst, 1995.

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

[5] M. Hecht. *Flow Analysis of Computer Programs.* North-Holland, New York, 1977.

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

[8] 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.

[9] S. P. Masticola. *Static detection of deadlocks in polynomial time.* PhD thesis, Rutgers University, 1993.

[10] S. P. Masticola, T. J. Marlowe, and B. G. Ryder. Lattice frameworks for multisource and bidirectional data flow problems. *ACM Transactions on Programming Languages and Systems*, 17(5):777–803, September 1995.

[11] G. N. Naumovich, L. A. Clarke, L. J. Osterweil, and M. B. Dwyer. Verification of concurrent software with FLAVERS. In *Proceedings of the 19th International Conference on Software Engineering*, pages 594–595, May 1997.

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