DATA FLOW ANALYSIS OF COMMUNICATING PROCESSES --Extended Abstract

John H. Reif
Computer Science Department
University of Rochester
Rochester, New York 14627

0. Abstract

Data flow analysis is a technique essential to the compile-time optimization of computer programs, wherein facts relevant to program optimizations are discovered by the global propagation of facts obvious locally.

This paper extends flow analysis techniques developed for sequential programs to the analysis of communicating, concurrent processes.

1. Introduction

We consider a set of "distant" concurrent processes, each sequentially executing a distinct program and communicating by the transmission and reception of messages. By "distant" it is meant that there is no interference between processes by shared variables, interrupts, or any other synchronization primitives beyond the message primitives. (The processes might in fact reside in the same machine.)

Various channels are available for communication between processes, and each channel has a unique process which is the destination of messages transmitted through this channel. The communication between processes is static if the channel arguments to message primitives are constants, and otherwise is dynamic. We consider both these cases. In the usual semantics for message communication, [Feldman, 1976], the process transmitting a message need not wait until reception of the message, and thus a queue of yet-to-be-received messages is associated with each channel. (In a more restrictive semantics proposed in [Hoare, 1977], the transmitter of a message M is required to wait until acknowledgement (a "handshake") of reception of M.)

We are interested in data flow analysis of communicating processes: the discovery of facts about distant processes and propagation of these facts across process boundaries. An analysis problem of particular interest here is reachability: can a given program statement ever be reached in some execution? Reachability is perhaps the simplest of data flow analysis problems.

Section 2 describes our flow model for a system of communicating processes, in which the control flow through each process's program is represented by a flow graph. This model allows for all executions valid in the usual semantics of communicating processes, but also may allow for additional, spurious executions. Of course, a statement unreachable in our model is also unreachable in the usual, more powerful semantics.

Unfortunately, we show in Section 3 that testing reachability in our flow model is recursively undecidable, using a reduction from state reachability of two-counter machines. In the case of static communication, we show that testing reachability in our flow model is log-space reducible to and from coverability in Petri nets, which [Lipton, 1975] has shown to require at least exponential space, infinitely often. Nevertheless, we have developed polynomial-time algorithms for approximate solutions to analysis problems.

***Preprint of paper appearing in 6th Annual ACM Symposium on Principals of Programming Languages*, San Antonio, TX, January 1979, pp. 257-268.**

As an aid to our flow analysis, we define in Section 4 a special acyclic directed graph called an event spanning graph containing a spanning tree of each process's flow graph, as well as certain edges (called message links), connecting pairs of TRANSMIT and RECEIVE statements between which a message may be sent. If no event spanning graph exists, then some program statement is unreachable. message may be sent. If no event spanning graph exists, then some program statement is unreachable.

Section 4 presents a linear time algorithm for constructing, when possible, an event spanning graph for the case of static communication.

In Section 5, we describe an iterative technique for data flow analysis of communicating processes with statis communication, which generalizes a technique for data flow analysis of sequentially executed programs due to [Hecht and Ullman, 1975]. Their algorithm repeated (until convergence) a pass through the flow graph of a single program, in topological order of its DAG (directed acyclic graph); our proposed algorithm repeats a pass through all the flow graphs of a set of communicating processes, in topological order of their event spanning graph.

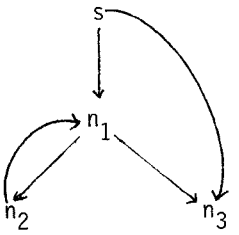
Section 6 extends our data flow analysis to the case of dynamic communication. We present an algorithm which builds an event spanning graph while simultaneously performing data flow analysis.

Section 7 concludes the paper.

2. The Flow Model for Communicating Processes

We describe here a flow model for a system of communicating processes. Let C be a set of symbols denoting communication channels. Each process P sequentially executes a distinct program, represented by a process flow graph G = (N,E,s). Each node $n \in N$ corresponds to a single (non-control) program statement. The edge set $E \subseteq N \times N$ consists of pairs of nodes between which control may transfer. An execution path is a path of G beginning at the start node s. Occasionally, we will distinguish a final or exit node $n_f \in N$ from which control may be considered to exit.

Example 2.1



The above flow graph has execution path (s, n_1, n_2, n_1, n_3), among others.

The state local to the process P is specified by a program counter (pointing to the currently executed program statement), the value bindings of the program variables local to P, and the message queues for channels with destination P (to be described shortly). Program statements include:

(1) Assignment statements of the form $X \leftarrow \alpha$ where X is a program variable local to P and α an expression, with the usual effect of setting X to the result of evaluating α.

(2) A transmit statement of the form TRANSMIT(α_1,α_2). The first argument α_1 must evaluate to a message channel $c \in C$, and α_2 evaluates to the message to be transmitted, say M. The message M cannot be a pointer value, but is otherwise unrestricted (in particular, M may be a communication channel). The second argument, α_2, may be absent, in which case some fixed default message is transmitted.

(3) A receive statement of the form $X \leftarrow$ RECEIVE(α) where the argument α must evaluate to a communication channel $c \in C$, and X is a program variable local to P assigned the value of the message received. We assume no start node of a flow graph is a RECEIVE statement.

(4) No-op (empty) statements will also be allowed.

Control statements are not found in N since the control flow is specified in our model by the edges of flow graph G. The sets of program variables local to distinct processes are disjoint, and are assumed to have no shared values. Thus, there is no interference between processes except that induced from our message primitives.

An event is the execution of a statement by a process and will be assumed to occur instantaneously; it might be preceded and succeeded by other events and might also occur simultaneously with others. A sequential execution of a single process flow graph G is a total ordering of events resulting from the sequential execution of the statements occurring in an execution path of G.

Given process flow graphs $G_1,\ldots,G_r$, an execution in our flow model is a partial ordering of events $(\mathcal{E},\rightarrow)$ in which those events associated with any given process flow graph G_i form a sequential execution of G_i, and such that $(\mathcal{E},\rightarrow)$ is consistent with the semantics of the message primitives TRANSMIT and RECEIVE as defined below. Intuitively, $e\rightarrow e'$ if event $e \in \mathcal{E}$ preceeds event $e' \in \mathcal{E}$.

If $e \in \mathcal{E}$ is an event resulting from the execution of statement

RECEIVE (α)

and α evaluates to channel c, then e must be preceeded by a unique event $e' \in \mathcal{E}$ resulting from the execution of a TRANSMIT statement whose first argument evaluates to channel c and whose second argument evaluates to the message received. In addition, we assume that if a process sends two successive messages, M_1 and M_2, over a channel c, they arrive in the order that they were transmitted.

Hence, if $e_1,e_2 \in \mathcal{E}$ are events resulting from the reception of M_1,M_2 and $e'_1, e'_2 \in \mathcal{E}$ are the corresponding message transmitting events, $e_1 \rightsquigarrow e_2$ implies $e'_1 \rightsquigarrow e'_2$.

No further assumptions about message behavior are made. Note that the resulting semantics are nondeterministic -- two simultaneous message transmissions by different processes over the same channel must arrive in sequential order, but we make no assumptions about this order.

An operational semantics for communicating processes is specified by designating for each channel $c \in C$ a message queue Q(c), listing the messages transmitted but yet-to-be-received over channel c. A RECEIVE statement has the effect of deleting the current message M at the front of the appropriate message queue, and evaluates to M. The order of messages appearing in the queue need only be consistent with our assumption that successive transmissions over a given channel from a given process be received in order of transmission.

3. Complexity of Reachability in the Flow Model

A program statement n is reachable if it appears in some execution in our flow model. In this section we consider the complexity of testing reachability.

In the general case of dynamic communication, it will be shown that the reachability problem is undecidable. Even in the case of a static communication (where the channel arguments to all TRANSMIT and RECEIVE statements are constants), testing reachability requires exponential space, infinitely often.

A two-counter machine is a finite-state automata augmented with a pair of counters which may b e incremented, decremented, and tested for zero. The halting problem for two-counter machines is known to be recursively unsolvable [Minsky, 1961]. Our proof that reachability of program statements is undecidable utilizes a simple recursive reduction from the halting problem for two-counter machines. This proof exploits the ability of a single TRANSMIT statement to communicate over various channels, depending on the evaluation of its channel arguments.

Theorem 3.1: Reachability is undecidable in the flow model of communicating processes.

Proof:

Consider a two-counter machine $(S, q_0, q_f, \mathcal{I}, C_1, C_2)$ with state set S, initial state $q_0 \in S$, final state $q_f \in S$, set of instructions $\mathcal{I}$, and counters C_1, C_2. Associated with each state $q \in S - \{q_f\}$ is an instruction $I_q \in \mathcal{I}$ of one of the following forms:

(1) (increment, i,q') where in state q the counter C_i is incremented by 1 and state q' is entered.

(2) (decrement, i, q') where in state q the counter C_i is decremented by 1 and state q' is entered.

(3) (test, i, q_1,q_2) where in state q if counter C_i=0 then state q_1 is entered and otherwise state q_2 is entered.

A computation begins in the initial state q_0 with both counters set to 0. Computations resulting in transitions to undefined states or negative counters are undefined. The computation halts at state q_f.

To simulate this two-counter machine, we build a process P which communicates to and from itself on channels $\{C_1, C_2, \#, \$\}$, (that is, P is the origin and destination of all messages). The flow graph of P is $G = (N,E,n_{q_0})$.

Initially, the message queues $Q(C_1)$, $Q(C_2)$, Q(#), and Q($) are empty. Associated with each instruction $I_q \in \mathcal{I}$ is a subgraph G_q of G which simulates this instruction. We shall claim that if C_i contains the integer $k \geq 0$, then $Q(C_i) = \#^k$.

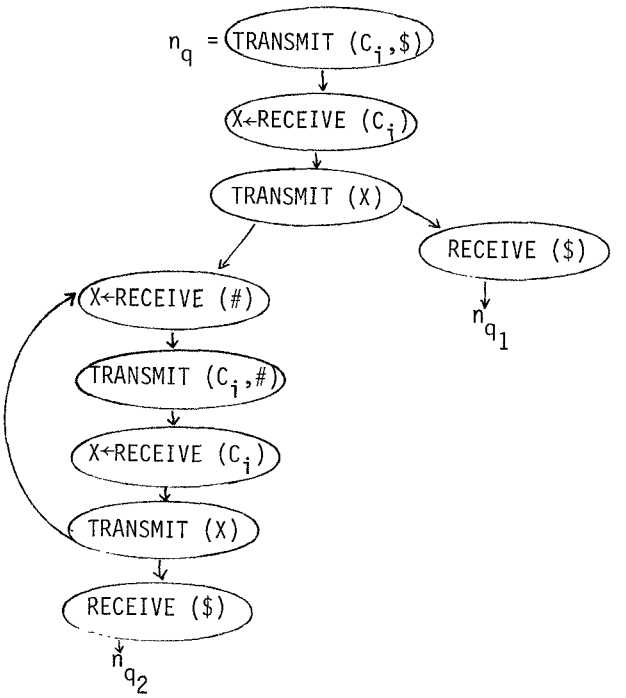
(1) If I_q = (increment, i, q') then G_q is of the form:

Hence, an execution of q results in the addition of # to $Q(C_i)$.

(2) If I_q = (decrement, i, q') then G_q is of the form:

and so an execution of q results in the deletion of some $\#\in Q(C_i)$ if this message queue is not empty, and otherwise if $Q(C_i)$ is empty, then the process P hangs.

(3) If I_q = (test, i, q_1, q_2) then G_q is of the form:



It may easily be shown that if $Q(C_i)$ is empty on execution of q, then n_{q_1} is reached and otherwise n_{q_2} is reached. In either case the queues are restored to their state just before execution of n_q.

Hence, the node n_{q_f} is reachable in some execution of P just in the case the given two-counter machine halts. □

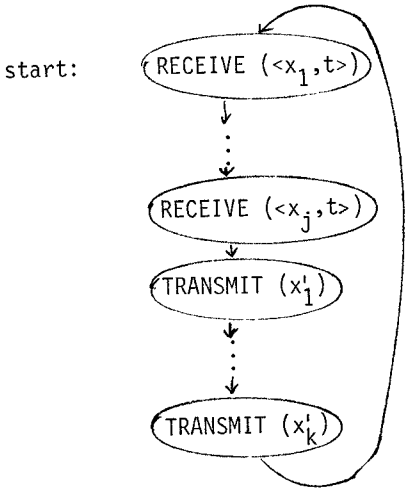
Next, let us assume the channel arguments to all TRANSMIT and RECEIVE statements are constants--so that the communication is static. With this restriction, there is a strong resemblance to a synchronization structure called a Petri net. In fact, we can show that reachability of statements in this case is polynomial-time reducible to and from coverability of Petri net markings.

A Petri net is a bipartite directed graph $PN = (\pi \cup T, E_{PN})$ with a set of <u>places</u> π, a set of <u>transitions</u> T, and a set of directed edges E_{PN}, each edge containing exactly one place and one transition. A <u>marking</u> of PN is a mapping μ from the places T to the non-negative integers. Given a marking μ and a transition $t \in T$ with no predecessor marked by μ with 0, t is <u>fired</u> by decrementing the markings of predecessors of t by 1 and incrementing the markings of successors of t by 1. The resulting marking μ' is said to be <u>derived</u> from μ. A marking μ' is <u>derivable</u> from initial marking μ_0 if there exists a sequence of markings $(\mu_0, \mu_1, \ldots, \mu_k = \mu)$ such that μ_i is derived from μ_{i-1} for i=1, ..., k. A marking μ is <u>coverable</u> if there exists a derivable marking μ' such that $\mu'(x) \geq \mu(x)$ for all places $x \in \pi$. A survey of Petri nets appears in [Peterson, 1977].

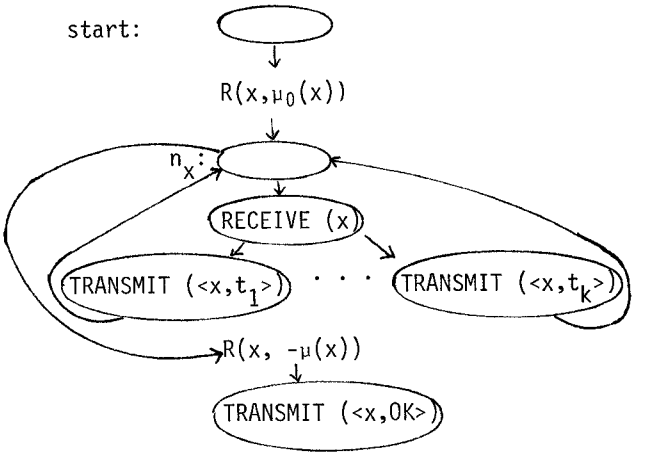
<u>Theorem 3.2</u>: Coverability in Petri nets is log-space reducible to reachability in our flow model with static communication.

<u>Proof</u>:

Let $PN = (\pi \cup T, E_{PN})$ be a Petri net with initial marking μ_0 and suppose we wish to test if marking μ is reachable in PN. For each transition $t \in T$ with predecessors $x_1, \ldots, x_j$ and successors $x'_1, \ldots, x'_k$, we have a process flow graph G_t:



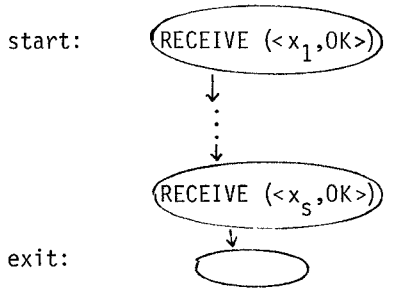
Observe that on receiving messages from channels $\langle x_1,t\rangle, \ldots, \langle x_j,t\rangle$, G_t transmits messages over channels $x'_1, \ldots, x'_k$. For each place $x \in \pi$ with successors $t_1, \ldots, t_k$ there is a process flow graph G_x:



Observe that for each message received over channel x, G_x transmits over one of the channels $\langle x,t_1\rangle, \ldots,$ or $\langle x,t_k\rangle$.

For any place $x \in \pi$ and integer k, R(x,k) is a subgraph which adds k messages to Q(x) if $k \geq 0$, and otherwise attempts to delete $|k|$ messages from Q(x) if k<0. If k<0 and $|k| > |Q(c)|$ then the process hangs and no successors of R(x,k) are reached. A simple but somewhat tedious construction produces an R(x,k) of size $O(\log |k|)$. The first time the node n_x is reached, Q(x) contains $\mu_0(x)$ messages. The exit node of process flow graph G_x is reached just in the case Q(x) contains $\geq \mu(x)$ messages on some visit to n_x.

Finally, we have a process flow graph G_0:



where $\pi = \{x_1, \ldots, x_s\}$ is the set of places. It follows that the exit node of G_0 is reachable if and only if the marking μ is coverable in Petri net PN with initial marking μ_0. □

By using Petri nets to simulate Turing Machine computations with exponential space bounds, [Lipton, 1975] has shown coverability of Petri net markings is EXP-SPACE hard. By Theorem 3.2, we have that

Corollary 3.1: Reachability in the flow model with static communication is EXP-SPACE hard. It is also straightforward to show:

Theorem 3.3: Reachability with static communication is log-space reducible to coverability in Petri nets.

4. Event Spanning Graphs

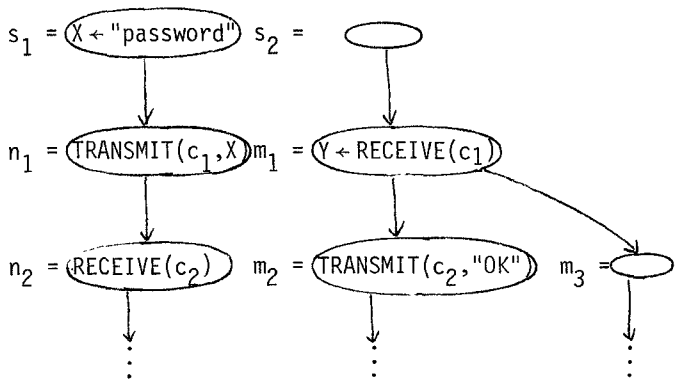
A message link is a pair of TRANSMIT, RECEIVE statements which communicate through the same channel.

Throughout this section we assume static communication; the channel arguments to TRANSMIT and RECEIVE statements are assumed constant. Thus, we may statically determine the set ML of all message links. (Since the number of message links may be quadratic in size of the process flow graphs, for efficiency the set ML is never explicitly constructed by our algorithm). Fix $G_1 = (N_1, E_1, s_1), \ldots, G_r = (N_r, E_r, s_r)$ as the process flow graphs. Let $N = \cup N_i$ be the set of program statements. Let an event spanning graph be an acyclic directed graph ESG such that

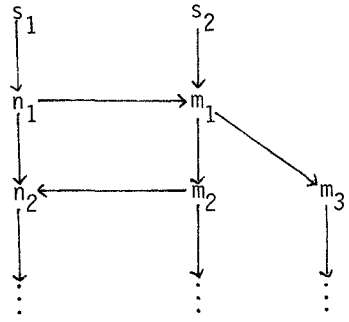
(1) For each i=1, ..., r the subgraph ESG_i of ESG, induced by dropping all nodes but those of N_i and deleting all edges except those between nodes of N_i, is a spanning tree of G_i.

(2) For each RECEIVE statement $n \in N$ there is a path in ESG from a start node to n, containing a TRANSMIT statement communicating over the same channel as n.

Example 4.1: The following are portions of flow graphs G_1 and G_2:

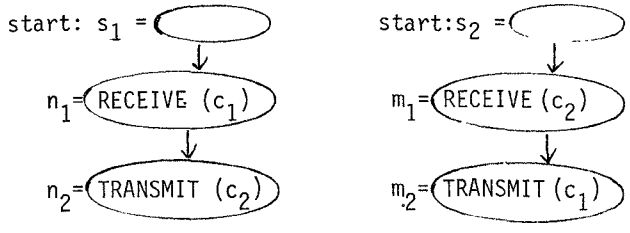


The corresponding portion of an event spanning graph ESG is:



First, we characterize the case where there exists no event spanning graph. A set $N_B \subseteq N$ of RECEIVE statements is a blocking set if for each TRANSMIT statement $m \in N$ with the same channel argument as an element of N_B, all execution paths from a start node to m contain some element of N_B.

Example 4.2: The set $B = \{n_1, m_1\}$ is a blocking set for the program flow graphs illustrated below:



It is easy to show

Lemma 4.1: There is a nonempty blocking set N_B, just in the case there is no event spanning graph.

Theorem 4.1: If there is no event spanning graph, then some state $n \in N$ is unreachable.

Proof:

By Lemma 4.1, there must be a nonempty blocking set N_B.

Suppose some $n \in N_B$ is reached in some execution $(\mathfrak{E},\rightarrow)$, and no other element of N_B is reached strictly before n. Since n is a RECEIVE statement, by definition of ESG there must be an execution of a TRANSMIT statement m over the same channel as n, previous to the first execution of n. This implies that in the flow graph G_i containing m, there is a path from the start node of G_i to m which contains no element of N_B, contradicting the assumption that N_B is a blocking set. □

We now present an algorithm for constructing either an event spanning graph or a blocking set.

Algorithm A

INPUT Process flow graphs $G_1 = (N_1,E_1,s_1)$, ..., $G_r=(N_r,E_r,s_r)$ with program statements $N = \cup N_i$ and message channel set C.

OUTPUT An event spanning graph ESG, if it exists, and otherwise a nonempty blocking set N_B.

```
begin
  VISIT-Q ← ∅;
  for each n ε N do
     if n is a start node then
      begin
         add n to VISIT-Q;
         VISIT(n) ← true;
      end;
     else VISIT(n) ← false;
     for each channel c ε C do
         begin
           SILENCE(c) ← true;
           WAITING(c) ← ∅;
         end;
  Initialize the digraph ESG to node set
        {s1,...,sr} and edge set ∅;
  until VISIT-Q = ∅ do
      begin
         choose and delete some statement
              n ε VISIT-Q;
         VISIT-NEXT ← ∅;
         if n is a TRANSMIT statement
              over a channel c and SILENCE(c)
            then begin
               SILENCE(c) ← false;
               VISIT-NEXT ← WAITING(c);
               WAITING(c) ← ∅;
            end;
         if n is a RECEIVE statement over a
              channel c and SILENCE(c)
            then add n to WAITING(c);
         else for each successor m of n such
              that (not VISIT(m))
            do add m to VISIT-NEXT;
         for each m ε VISIT-NEXT do
            begin
               if not VISIT(m) then
                  begin
                     add m to the node set of ESG;
                     VISIT(m) ← true;
                  end;
               add m to VISIT-Q;
               add (n,m) to the edge set of ESG;
            end;
      end;
  N_B ← ∅;
  for each channel c ε C do N_B ← N_B ∪ WAITING(c);
  if N_B = ∅
     then return event spanning graph ESG;
     else return blocking set N_B;
end;
```

In the above algorithm, a node $n \in N$ has been visited if VISIT(n) has been set to true. It is easy to verify that for each channel $c \in C$, if no transmit statement over c has been visited, then

(1) SILENCE(c) = true

(2) WAITING(c) contains all RECEIVE statements over channel c so far visited.

Otherwise, when a TRANSMIT statement over c is visited then

(1') SILENCE(c) is set to false

(2') All RECEIVE statements in WAITING(c) are added to ACTIVE, and WAITING(c) is set to ∅.

Theorem 5.1: If an event spanning graph exists then the above algorithm returns one, else it returns a nonempty blocking set.

Proof:

It is easy to verify that if the algorithm returns a digraph, then it is an event spanning graph. On the other hand, suppose the algorithm returns the set $N_B = \cup_{c \in C} WAITING(c)$. We claim this is a blocking set. Suppose not. Then there exists a RECEIVE statement $n \in N_B$, and a TRANSMIT statement m communicating over the same chann c as n, and a path p in a flow graph G_i from a start node s_i to m and avoiding all elements of N_B. In the above algorithm s_i is initially added to VISIT-Q. It is easy to show that all other elements of p are also eventually added to VISIT-Q

including m. Thus WAITING(c) is eventually set to $\emptyset$, a contradiction with the assumption that $n \in N_B$. □

5. Data Flow Analysis of Communicating Processes with Static Communication

Data flow analysis yields information about a program. This information is too weak to suffice for program correctness proofs; however, it is sufficient for the usual compile time program optimizations. For example, it may be discovered that certain program variables or expressions always evaluate to constants; hence the compiler may substitute single load instructions for more complex sequences of instructions which compute the same constant value. We wish to extend data flow analysis techniques to the analysis of concurrent programs. ([Hecht, 1977] is an informative text on data flow analysis of sequential programs.)

Let D be a set of predicates. We assume a semi-lattice $(D,\vee)$, where $\vee: D^2 \to D$ is the usual lattice meet operation, which is binary, associative, commutative, and idem potent on D. We require $\vee$ to be the following weakening of logical disconjunction:

for each $p, q \in D$, if p or q hold, then $p \vee q$ holds.

The lattice partial order => is defined:

$q \Rightarrow p$ if and only if $p \vee q = p$ for all $p, q \in D$.

Note that => is the restriction of logical implication to domain D;

$q \Rightarrow p$ just in the case q implies p, for all $p, q \in D$.

We assume D contains TRUE, FALSE as minimum and maximum values, respectively, so (p => TRUE) and (FALSE => p) for all $p \in D$. We also assume that $(D,\vee)$ is well founded, containing no infinite strictly decreasing chains $p_1 \Rightarrow p_2 \Rightarrow \ldots$. Similar data flow analysis frameworks appear in [Graham and Wegman, 1977; Hecht, 1977; Wegbreit, 1975].

Example 5.1: We consider a domain D_I containing finite sets of inclusion relationships of the form:

$$X \in S$$

where X is a variable and S is a nonempty set of constants. We assume that the sets of inclusion relationships contained in D_I are normalized, containing no more than one inclusion relationship for any variable, and no more than a fixed number k_0 of constants in any inclusion relationship. The domain also contains FALSE (representing inconsistent inclusion relationships of the form $X \in \{\ \}$) and TRUE (considered the empty set $\emptyset$).

Given $p,q \in D_I$, the meet operation $\vee_I$ combines sets p and q, and normalizes the results. The relation $p \Rightarrow_I q$ holds for $p,q \in D_I$ if for each inclusion relationship $X \in S$ of q there is a corresponding inclusion relationship $X \in S'$ of p where $S \subseteq S'$. Since each element of D_I is a finite set and each inclusion relationship is of bounded size, the semilattice $(D_I,\vee_I)$ is well-founded. (end of Example 5.1)

A function $f: D \to D$ is isotone if $f(p) \Rightarrow f(p')$ for all $p,p' \in D$, such that $p \Rightarrow p'$. Similarly $g: D^2 \to D$ is isotone if $g(p,q) \Rightarrow g(p',q')$ for all $p,p', q,q' \in D$ such that $p \Rightarrow p'$ and $q \Rightarrow q'$.

If a program statement is neither a TRANSMIT or RECEIVE statement, then we assume a isotone function (the transfer function of n) $\Delta_n: D \to D$ weakly describing the change of state on execution of program statement n. More precisely if $p \in D$ holds just before execution of program statement n, then $\Delta_n(p)$ holds on exit from n. ($\Delta_n(p)$ need not be the strongest such predicate, as would be required in a Hoare Logic.)

Example 5.2:

$$\Delta_{X \leftarrow Y+X}(\{X \in \{1,2\},\ Y \in \{3\}\}) = \{X \in \{4,5\},\ Y \in \{3\}\}.$$

That is, if X is known to be either 1 or 2 and Y is known to be 3, then after execution of the statement $X \leftarrow Y+X$, the variable X may be either 4 or 5.

We associate with each TRANSMIT statement n an isotone function $\tau_n: D \to D$ such that is $p \in D$ holds on input to n, then $\tau_n(p)$ is a predicate describing the value of the message transmitted by n. A symbol M_c is used to represent the collection of all messages transmitted over channel c.

Example 5.3:

If n = TRANSMIT(c,X*Y), then

$$\tau_n(\{X \in \{2,3\},\ Y \in \{4\}\}) = \{M_c \in \{8,12\}\}.$$

For each RECEIVE statement n, we assume a an isotone function $p_n: D^2 \to D$ such that if $p \in D$ holds just on input to n and $q \in D$ holds for each message received by n, then $\rho_n(p,q)$ holds on output to statement n.

Example 5.4: Given program statement $n = (X \leftarrow RECEIVE(c))$, we have $\rho_n(\{X \in \{1,2\}, Y \in \{5\}\}, \{M_c \in \{3,4\}\}) = \{X \in \{3,4\}, Y \in \{5\}\}$.

The objective of our data flow analysis is to compute for each program statement $n \in N$ the predicates IP_n, $OP_n \in D$ where:

(1) IP_n holds an input to n on all executions.
(2) OP_n holds an output from n on all executions.

Let C be the set of channels occurring as argument to TRANSMIT and RECEIVE statements. For each channel $c \in C$ we wish also to compute $MP_c \in D$, a predicate holding for all messages transmitted over channel c.

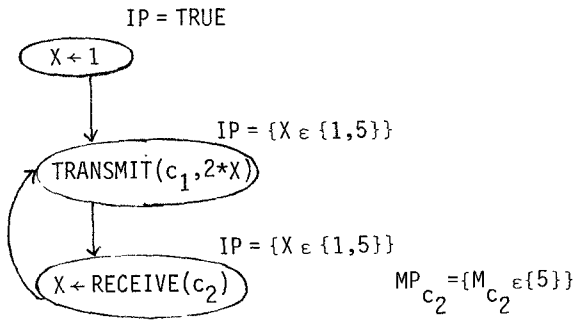
Specifically, we wish minimal $IP_n, OP_n \in D$ for each program statement $n \in N$, and minimal $MP_c \in D$ for each channel $c \in C$ satisfying:

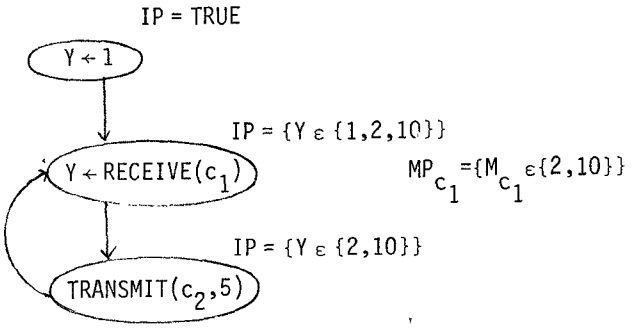
(1) IP_n = TRUE if n is a start node
$= \bigvee OP_m$ otherwise
all predecessors m of n.

(2) $OP_n = IP_n$ if n is a TRANSMIT statement
$= \rho_n(IP_n, MP_c)$ if n is a RECEIVE statement over channel c
$= \Delta_n(IP_n)$ otherwise.

(3) $MP_c = \bigvee \tau_n(IP_n)$.
all TRANSMIT statements $n \in N$ over channel c.

Example 5.5: Illustrated below is an example containing maximal solutions to the data flow equations in the case of the domain of inclusion relations:





Recall that an event spanning graph of the last section is acyclic and has node set N, the set of program statements. A topological ordering of an acyclic directed graph is a total ordering of its node set ordering predecessors before successors. A topological ordering may be easily computed in linear time as described in [Knuth, 1968].

We now present an algorithm which repeatedly computes approximations to the above data flow analysis equations. In each pass, the algorithm visits each node in N in topological order of an event spanning graph.

Algorithm B

INPUT Program statements N, channel set C, process flow graphs $G_1, \ldots, G_r$, an event spanning graph ESG, and the functions Δ_n, τ_n, ρ_n defined for each appropriate $n \in N$.

OUTPUT Minimal IP_n, OP_n, MP_c, for $n \in N$, $c \in C$ satisfying the data flow analysis equations (1), (2), and (3).

```
begin
   for each n ∈ N do
      begin
         if n is a start node then IP_n ← TRUE
                              else IP_n ← FALSE;
         OP_n ← FALSE;
      end;
   for each c ∈ C do NP_c ← FALSE;
   compute a topological ordering of ESG;
   until no change do
      for each n ∈ N in the topological ordering
                    of ESG do
         begin
            for each predecessor m of n
               do IP_n ← IP_n ∨ OP_m;
            if n is a TRANSMIT statement then
               begin
                  let c be the channel over
                     which n transmits;
                  MP_c ← MP_c ∨ τ_n(IP_n);
                  OP_n ← IP_n;
```

```
            end;
        else if n is a RECEIVE statement then
            begin
                let c be the channel through
                    which n communicates;
                OP_n ← ρ_n(IP_n,MP_c);
            end
        else OP_n ← Δ_n(IP_n);
    end;
end;
```

Let ML be the set of all message links. Let E be the set of all the edges of the process flow graphs $G_1, \ldots, G_r$. Clearly, each iteration of the algorithm requires $O(|N| + |E| + |ML|)$ operations. In the worst case, this algorithm converges after $|N|\lambda$ iterations, where λ is the longest strictly decreasing chain of (D,V). Thus the total time cost is $O(|N|(|N| + |E| + |ML|)\lambda)$. (There is evidence, however, that the algorithm converges much faster in practice.) On convergence, the data flow analysis equations (1), (2), and (3) are clearly satisfied, and we can show

Theorem 5.1: Algorithm B yields a minimal solution to the data flow analysis equations (1), (2), and (3).

6. Data Flow Analysis in the Case of Dynamic Communication

In the previous section we assumed that all message arguments of TRANSMIT and RECEIVE statements are constants. We are able to further refine our data flow analysis techniques to the case of a dynamic communication: where the channel arguments of the communication primitives are expressions which must evaluate to channels but not necessarily the same channel on all executions. Our analysis is made more difficult by the fact that the messages communicated between processes may be channel names. For example, a given process may inform other processes of new channels over which they may communicate. Let C be the set of all channels over which processes might communicate.

Let (D,V) be a semilattice with ordering => and let τ_n, ρ_n, and Δ_n be the isotone functions describing transformations of predicates in D through TRANSMIT, RECEIVE, and non-communication statements, respectively as defined in the last section. We also assume for each TRANSMIT or RECEIVE statement $n \in N$ the isotone function $CHANNELS_n$ mapping from D to the power set of C.

Approximation is an essential technique in global flow analysis. Here we use $CHANNELS_n$ to approximate the possible channels over which n may communicate. More formally, given a predicate $p \in D$ weakly describing the state just before execution of statement n, $CHANNELS_n(p)$ must contain at least all channels over which n may communicate. By isotone, we mean here that if $p \Rightarrow q$ then $CHANNELS_n(p) \subseteq CHANNELS_n(q)$ for all $p,q \in D$.

Example 6.1: In the domain D_I of inclusion relationships defined in the last section,

$$CHANNELS_{RECEIVE(Y)}(Y \in \{c_1,c_2\}) = \{c_1,c_2\}$$

We seek minimal IP_n, OP_n, $MP_c \in D$ for all program statements $n \in N$ and channels $c \in C$ satisfying:

(1') IP_n = TRUE if n is a start node
$= \bigvee OP_m$ otherwise.
all predecessors m of n.

(2') $OP_n = IP_n$ if n is a TRANSMIT statement
$= \bigvee \rho_n(IP_n,MP_c)$ if n is a RECEIVE statement
all $c \in CHANNELS_n(IP_n)$.
$= \Delta_n(IP_n)$ otherwise.

(3') $MP_c = \bigvee \tau_n(IP_n)$
all TRANSMIT statements with
$c \in CHANNELS_n(IP_n)$.

For any solution of the above data flow equations, IP_n and OP_n are predicates in D holding on input and output, respectively, to program statement $n \in N$, and MP_c holds for all messages transmitted over channel $c \in C$. Note that the above equations differ from those given in the last section for static communication, only in that we use $CHANNELS_n(IP_n)$ to estimate the channels over which each TRANSMIT or RECEIVE statement n may communicate.

The following algorithm yields a minimal solution to the equations (1'), (2'), and (3') as long as the data flow domain (D,V) has no infinite strictly decreasing chains. Our algorithm combines Algorithms A and B of the

previous sections, building an event spanning graph ESG while simultaneously carrying out data flow analysis. As in Algorithm B, the event spanning graph ESG is used to order the nodes through which we propagate data flow information.

Algorithm C

INPUT The channel set C, process flow graphs $G_1,=(N_1,E_1,s_1),\ldots,G_r=(N_r,E_r,s_r)$ with the set of program statements $N=\bigcup N_i$, and the functions τ_n, ρ_n, Δ_n and $CHANNELS_n$ defined for each appropriate $n \in N$.

OUTPUT Blocking set N_B, or event spanning graph ESG with minimal IP_n, OP_n, MP_c for $n \in N$ and $c \in C$ satisfying the data flow analysis equations (1'), (2'), and (3').

```
begin
   procedure PROPAGATE(n);
      begin
       VISIT-NEXT ← ∅;
       for each predecessor m of n
          do IP_n ← IP_n ∨ OP_m;
       if n is a TRANSMIT statement then
            begin
               OP_n ← IP_n;
               for each c ε CHANNELS_n(IP_n) do
                  begin
                     MP_c ← MP_c ∨ τ_n(IP_n);
                     if WAITING(c) ≠ ∅ then
                        begin
                           for each m ε WAITING(c)
                             such that WAIT(m) do
                              begin
                                 WAIT(m) ← false;
                                 add m to
                                   VISIT-NEXT;
                              end;
                           WAITING(c) ← ∅;
                        end;
                  end;
            end;
       else if n is a RECEIVE statement then
          (if WAIT(n) then
             for each c ε CHANNELS_n(IP_n) do
                      add n to WAITING(c);
          else for each c ε CHANNELS_n(IP_n) do
                      OP_n ← OP_n ∨ ρ_n(ip_n,MP_c));
          else OP_n ← Δ_n(IP_n);
          if not WAIT(n) do
            for each successor m of n such that
                  (not VISIT(M)) do
                   add m to VISIT-NEXT;
          for each m ε VISIT-NEXT do
                     begin
                        if not VISIT(m) then
                           begin
                              add m to the node
                                set of ESG;
                             VISIT(m) ← true;
                           end;
                        add m to VISIT-Q;
                        add  (n,m) to the edge
                           set of ESG;
                     end;
      end;
INITIALIZE:
     VISIT-Q ← ∅;
     for n ε N do
     if n is a start node then
        begin
           VISIT(n) ← true;
           IP_n ← TRUE;
           OP_n ← FALSE;
           add n to VISIT-Q;
        end;
     else
        begin
           VISIT(n) ← false;
           IP_n ← OP_n ← FALSE;
           if n is a RECEIVE statement then
                    WAIT(n) ← true;
           else WAIT(n) ← false;
        end;
     for each channel c ε C do
        begin
           WAITING(c) ← ∅;
           MP_c ← FALSE;
        end;
    Initialize digraph ESG to node set {s_1,...,s_r} and
       edge set { };
      MAIN:
        until no change do
           begin
              until VISIT-Q = ∅ do
                 begin
                    choose and delete some n ε VISIT-Q;
                    PROPAGATE(n);
                 end;
              for each node n of ESG in topological
                    order do PROPAGATE(n);
           end;
        N_B ← ∅;
        for each c ε C do N_B ← N_B ∪ WAITING(c);
        if N_B = ∅ then return blocking set N_B;
                  else return ESG and IP_n,OP_n, MP_c
                       for each n ε N and c ε C;

end;
```

A proof of algorithm C will appear in the full version of this paper.

7. Conclusion

The data flow analysis algorithms presented in this paper are currently being implemented in a program analysis system for a language Zeno [Ball et al. 1978] with concurrent communicating processes. We are developing direct (non-iterative) data flow analysis algorithms which run very efficiently in the case the process's programs and their communications are well structured.

Bibliography

[1] Ball, J.E., Williams, G.J. and Low, J.R., "Preliminary ZENO language description", TR41, Computer Science Department, University of Rochester, October 1978.

[2] Graham, S. and Wegman, M., "A fast and usually linear algorithm for global flow analysis", JACM, 23, No. 1, (January, 1977), pp. 172-202.

[3] Feldman, J.A., "A programming methodology for distributed computing (among other things)", TR9, Computer Science Dept., University of Rochester, Rochester, New York, (1976).

[4] Hanson, P.B., "Concurrent programming concepts", Computing Surveys, Vol. 5, No. 4 (Dec. 1973), pp. 223-245.

[5] Hecht, M.S., Data Flow Analysis of Computer Programs, American Elsevier, New York (1977).

[6] Hecht, M.S. and Ullman, J.D., "Analysis of a simple algorithm for global flow problems", SIAM J. of Computing, Vol. 4, No. 4 (Dec. 1975), pp. 519-532.

[7] Hoare, C.A.R., "Communicating sequential processes", Computer Science Dept., Queen's University, Belfast (March, 1977).

[8] Jones, N.D., Landweber, C.H., and Lien, Y.E., "Compexity of some problems in Petri nets" Theoretical Computer Science, No. 4 (1977), pp. 277-309.

[9] Kam, J.B. and Ullman, J.D., "Monotone data flow analysis frameworks", TR167, Computer Science Dept., Princeton University (Jan. 1976).

[10] Killdall, G.A., "A unified approach to program optimization", Proc. ACM, Symp. on Principles of Programming Languages (Jan. 1975), pp. 10-21.

[11] Knuth, D.E., The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass. (1968).

[11] Knuth, D. E., The Art of Computer Programming, Vol 1:Fundamental Algorithms, Addison-Wesley, Reading, Mass. (1968).

[12] Lipton, R., "The reachability problem and boundedness problem for Petri nets and exponential-space hard", Conf. on Petri nets and Related Methods, M.I.T. (July, 1975); also Yale Research Report #62, (1976).

[13] Minsky, M., "Recursive unsolvability of Post's Problem", Ann. of Math., 74 (1961), pp. 437-454.

[14] Peterson, M.L., "Petri nets," Computing Surveys, Vol. 9, No. 3 (Sept. 1977), pp. 223-252

[15] Tarjan, R.E., "Depth-first search and linear graph algorithms", SIAM J. of Computing, Vol. 1, No. 2 (June 1972), pp. 46-100.

[16] Wegbreit, B., "Property extraction in well-founded property sets", IEEE Trans. on Software Engg. 1, 3, 1975, pp. 270-285.

APPENDIX

Graph-Theoretic Definitions

A directed graph (N,E) consists of a finite set N of nodes and a set E of ordered pairs (n,m) of distinct nodes, called edges. If (n,m) is an edge, m is a successor of n and n is a predecessor of m. A graph (N',E') is a subgraph of (N,E) if $N' \subseteq N$ and $E' \subseteq E$. A path of length k from n to m is a sequence of nodes $(n = n_0, n_1, \ldots, n_k = m)$ such that $(n_i, n_{i+1}) \in E$ for $0 \leq i < k$. The path is simple if $n_0, \ldots, n_k$ are distinct (except possibly $n_0 = n_k$) and the path is a cycle if $n_0 = n_k$. A graph is acyclic if it contains no cycles. A directed graph $(N_1 \cup N_2, E)$ is bipartite if N_1, N_2 are disjoint and $E \subseteq (N_1 \times N_2) \cup (N_2 \times N_1)$.

A flow graph G = (N,E,s) is a directed graph (N,E) with a distinguished start node s such that for any node $n \in N - \{s\}$ there is a path from s to n. A (directed, rooted) tree T = (N,E,s) is a flow graph such that |E| = |N|-1. The start node s is the root of the tree. Any tree is acyclic, and if n is any node in a tree T, there is a unique path from s to n. If G = (N,E,s) is a flow graph and T = (N',E',s') is a tree such that (N',E') is a subgraph of G and N=N' and s=s', then T is a spanning tree of G.