Data Flow Analysis of Distributed Communicating Processes[1]

John H. Reif[2] and Scott A. Smolka[3]

Received June 1989; revised June 1990

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 several known techniques for data flow analysis of sequential programs to the static analysis of distributed communicating processes. In particular, we present iterative algorithms for detecting unreachable program statements, and for determining the values of program expressions. The latter information can be used to place bounds on the size of variables and messages. Our main innovation is the *event spanning graph*, which serves as a heuristic for ordering the nodes through which data flow information is propagated. We consider both *static communication*, where all channel arguments are constants, and the more difficult *dynamic communication*, where channel arguments may be variables and channels may be passed as messages.

KEY WORDS: Communicating processes; data flow analysis; message passing; reachability.

[1] A preliminary version of this paper appeared in the proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages, pp. 257–268, June 1979.

[2] Supported by National Science Foundation Grant NSF MCS82-00269 and the Office of Naval Research Contract N00014-80-C-0647. Author's address: Department of Computer Science, Duke University, Durham, North Carolina 27706.

[3] Supported by National Science Foundation Grants NSF DCR-8505873 and NSF CCR-8704309. Author's address: Department of Computer Science, SUNY at Stony Brook, Stony Brook, New York 11794-4400.

Preprint of paper appearing** in **International Journal of Parallel Programming, **Vol. 19, No. 1, February 1990.**

1. INTRODUCTION

1.1. The Problem

We consider a collection of processes, each sequentially executing a distinct program and communicating by the transmission and reception of messages. We assume that there is no interference between processes by shared variables, interrupts, or any other synchronization primitives beyond the message primitives.

Various channels are availabe for communication among processes, and each channel has a unique process which is the destination of messages transmitted via this channel. Communication among processes is *static* if the channel arguments to message primitives are constants, and otherwise is *dynamic*. Hoare's CSP(1) is a language proposal incorporating static communication, while NIL,(2) a high-level systems programming language developed at IBM Research, Yorktown Heights, uses dynamic communication. Dynamic communication can also be found in the process calculi of Refs. 3 and 4. We consider both static and dynamic communication.

We assume semantics for message passing where the process transmitting a message need not wait until reception of the message, and thus an unbounded queue of yet-to-be-received messages is associated with each channel. This type of *asynchronous* message passing is used in NIL and in the distributed programming language PLITS.(5)

Message passing in CSP is *synchronous*: the transmitter of a message M is required to wait until acknowledgement of reception of M. This kind of communication is difficult to implement due to synchronization problems which arise. Reif and Spirakis(6) give real-time randomized algorithms for achieving the required synchronization.

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. To our knowledge, Reif(7) was the first paper on this topic; of which, this paper is an expanded version. An analysis problem of particular interest here is *reachability*: Can a given program statement ever be reached in some execution? Reachability is perhaps the most fundamental of data flow analysis problems. If a program statement n in a process P is unreachable, then an attempt by process P to reach n will result in a deadlock. We are also concerned with the problem of detecting the values of program expressions. This information can be used to place bounds on the size of variables and messages.

Our solution method does not in general yield an optimal approximation to the data flow analysis problem, particularly for models in which processes communicate over ordered message queues. However, our technique is sound and efficient (in fact our algorithm for reachability analysis in the case of static communication runs in time linear in the program size). We view our solution method as a means to a useful first approximation to the data flow analysis problem, a technique one would use before submitting a program of distributed communicating processes to more sophisticated and expensive analysis methods.

1.2. Our Solution

Section 2 describes our *flow graph model* for systems of communicating processes. The potential flow of control of each process's program is represented by a flow graph, as is usual in data flow analysis.[8] In a flow graph, conditional branches from the original program are replaced by purely nondeterministic branches. 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 in which conditionals are interpreted. Thus our resulting data flow analysis techniques are *conservative.*

Note that we are not necessarily interested here in models strong enough for correctness proofs. Instead we desire *reasonable models* for which analysis algorithms exist and are powerful enough to be useful for practical situations of program analysis.

As an aid to our flow analysis, we define in Section 3 a special directed acyclic graph called an *event spanning graph.* It contains 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 can be sent. Importantly, if no event spanning graph exists, then some program statement is unreachable. The converse, however, is not necessarily true. Section 3 presents a linear-time algorithm for constructing, when possible, an event spanning graph for the case of static communication.

In Section 4, we describe an iterative technique for communicating processes with static communication that determines the values of program expressions. This generalizes a technique for data flow analysis of sequentially executed programs due to Hecht and Ullman.[9] Their algorithm repeats (until convergence) a pass through the flow graph of a single program, in topological order of its depth-first spanning tree. 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 5 extends our data flow analysis to the case of dynamic communication. We present an algorithm that builds an event spanning graph while simultaneously determining the values of program expressions. Section 6 concludes.

1.3. Related Work

Since the publication of Reif,(7) Cousot and Cousot(10) have developed a semantic analysis technique for CSP programs that can be used as an invariance proof technique or for data flow analysis.

More recently, Peng and Purushothaman(11) have proposed a data flow approach to analyzing networks of two communicating finite state machines for non-progress properties (deadlock and unspecified reception). They consider FIFO message buffers as opposed to the unordered message buffers considered here. However, their approach involves the construction of the network's product machine, while the complexity of our analysis algorithm is linear in the size of the network description.

Schlichting and Schneider(12) have developed a deductive proof system for a model of distributed communicating processes similar to our own. Furthermore, the predicate transformers they use for communication statements are similar to the ones we use for data flow analysis. Their technique is intended for correctness proofs rather than data flow analysis, and is not mechanizable. Other related work includes Ref. 13.

A companion paper by Reif and Smolka,(14) investigates the complexity of reachability for several models of distributed communicating processes.

2. THE FLOW GRAPH MODEL FOR COMMUNICATING PROCESSES

We describe here a flow graph model for a system of processes $\{P_1,..., P_r\}$. These processes intercommunicate over channels having names taken from the set C. Each process P_i sequentially executes a distinct program represented by flow graph $G_i = \langle N_i, E_i, s_i \rangle$. Each node $n \in N_i$ corresponds to a single (non-control) program statement, and is labeled by an assignment, transmit, receive, or no-op statement. The edge set $E_i \subseteq N_i \times N_i$ consists of pairs of nodes between which control may transfer. An *execution path* of G_i is a path of G_i beginning at the start node s_i (see Fig. 1 for an example).

To simplify the definition of reachability here we assume, without loss of generality, that each start node s_i is labeled by a no-op statement.

2.1. Informal Syntax and Semantics of Communicating Processes

Process P_i may execute program statements of the form:

(1) *Assignment statements* "$X \leftarrow E$" where X is a program variable local to P_i and E is an expression. This statement has the usual effect of setting X to the result of evaluating E.

(2) *Transmit statements* "TRANSMIT (E_1, E_2)" where expression E_1 must evaluate to a message channel $c \in C$, and E_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 can be a communication channel (in the case of dynamic communication). E_2 may be absent, in which case some fixed default message is sent. The transmit statement is assumed to be executed without delay, regardless of the number of messages previously transmitted over channel c.

(3) *Receive statements* "$X \leftarrow$ RECEIVE (E)" where E must evaluate to a communication channel $c \in C$, and X is an optional program variable local to P_i assigned the value of the message received. If no message is in the message queue for channel c, then the receive statement's execution is blocked until a message is transmitted over channel c.

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

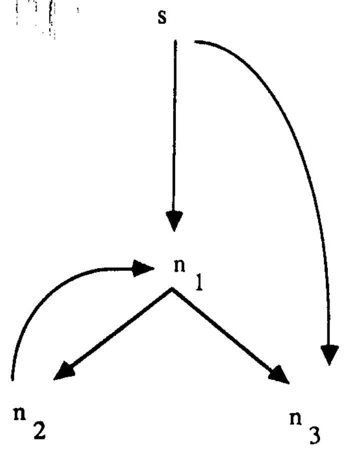

Fig. 1. The above flow graph has execution path (s, n_1, n_2, n_1, n_3), among others.

Control statements are not found in N_i since the control flow is specified in our model by the edges of flow graph G_i. 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.

2.2. Formal Semantics of the Flow Graph Model

We present a description of the operational behavior for our flow graph model of communicating processes. We refer to this model as M_0 and it is distinguished by the fact that messages are received in the order they are transmitted. A weaker model, M_1, will be presented next.

We assume a common value domain $V = Num \cup C$ for each process in the system. Here $Num = \{0, 1, 2, \ldots\}$ and C is the set of channel names. Any other domain can be substituted for *Num*, if so desired. Thus variables in our language will be of type *Num* or *channel*.

We assume that programs to which we apply our data flow analysis techniques are *well-typed*. For a program to be well-typed, the first argument of a transmit statement and the lone argument of a receive statement must evaluate to channels in C. Such "channel expressions" are restricted to be channel variables or channel names, in the case of dynamic communication, and to channel names only in the case of static communication. For a program to be well-typed it must also be the case that:

(1) The type of the expression on the right-hand side of an assignment statement matches the type of the variable on the left-hand side.

(2) "Semantically matching" transmit and receive statements agree on type. A transmit statement T and receive statement R semantically match if T transmits a message M along channel c and R receives M from c. T and R agree on type if the type of M matches the type of the variable on the left-hand side of R.

Let $\{P_1, \ldots, P_r\}$ be our system of processes, and let G_i be the flow graph of the program executed by P_i, $1 \leqslant i \leqslant r$. Also, let Var_i be the set of variable names local to process P_i. We can formally describe the state of the system at any point in its execution in terms of a *global state*

$$S = \langle [n_1, \ldots, n_r], [m_1, \ldots, m_r], \{b_c \mid c \in C\} \rangle$$

Here:

$n_i \in N_i$ is process P_i's *current node*.

m_i: $Var_i \rightarrow V \cup \{unbound\}$ is P_i's *memory function*, which performs the usual mapping of identifiers to values. We extend m_i to expressions by letting $m_i(E)$ denote the value of expression E based on the bindings in m_i.

$b_c \in V^*$ is the current *buffer contents* associated with channel $c \in C$. Intuitively, b_c describes the sequence of yet-to-be-received messages transmitted over channel c.

The concurrent execution of a system of communicating processes proceeds as the "evolution" of one system global state into another. Such evolution involves the execution of *exactly one* program statement labeling a flow graph node. The execution of a statement is assumed to happen instantaneously and is referred to as an *event*. We use $e_1, e_2, \ldots,$ to denote events. System evolution is nondeterministic: any *enabled* statement may be executed next. Assignment, transmit, and no-op statements are always enabled, i.e. in any global state. A receive statement is enabled in global state S only if $b_c \neq \varepsilon$ (the empty channel), where the channel argument to the receive statement evaluates to c.

Notation: Let f: $D_1 \rightarrow D_2$ be a function. Then, $f[d_2/d_1]$, $d_1 \in D_1$ and $d_2 \in D_2$, denotes the function that is everywhere the same as f, except possibly on d_1 where its value is d_2.

Definition 1. Let $S = \langle Q = [n_1, \ldots, n_r],\ M = [m_1, \ldots, m_r],\ B = \{b_c \mid c \in C\} \rangle$ be a global state. Then *S can evolve through event $e = n_i'$ into* global state $S' = \langle Q' = [n_1, \ldots, n_{i-1}, n_i', n_{i+1}, \ldots, n_r], M', B' \rangle$ iff n_i' is enabled in S, (n_i, n_i') is an edge in flow graph G_i, and
if $e =$ "$X \leftarrow E$" then
M' equals M with m_i replaced by $m_i[m_i(E)/X]$, i.e. X is now bound to the value of expression E, and B' equals B.
if $e =$ "TRANSMIT (E_1, E_2)" then
M' equals M and, assuming E_1 evaluates to $c \in C$, B' equals B with b_c replaced by *append* $(m_i(E_2), b_c)$; i.e., the value of expression E_2 is appended to the rear of buffer b_c.
if $e =$ "$X \leftarrow$ RECEIVE (E)" then
assuming E evaluates to c, M' equals M with m_i replaced by $m_i[head(b_c)/X]$; i.e. X is now bound to the value at the head of buffer b_c. Also, B' equals B with b_c replaced by $rest(b_c)$, i.e. the head element of b_c is removed. ▮

An *execution* is a possibly infinite sequence of global states $S_0, S_1, S_2, \ldots,$ such that S_0 is the *initial global state*

$$S_{init} = \langle [s_1, \ldots, s_r], [m_1 \Leftarrow \underline{unbound}, \ldots, m_r \Leftarrow \underline{unbound}], \{b_c = \varepsilon \mid c \in C\} \rangle$$

and S_i can evolve into S_{i+1}, $i \geqslant 0$. Regarding S_{init}, recall that s_i is the start node of G_i and by convention is labeled with "no-op"; $\underline{unbound}$ is the everywhere *unbound* function (thus, all of memory is initially undefined); and ε represents the empty sequence (thus, all buffers are initially empty).

In general, a system of communicating processes may have many possible executions. We say that *global state S is reachable* if it is contained in some execution. *Flow graph node q in G_i is reachable* if there exists a reachable global state $S = \langle Q, M, B \rangle$ such that the ith element of Q is q.

Note that any execution of events is consistent with the following partial order:

(1) Events associated with process flow graph G_i form a sequential execution of G_i.

(2) If e_{rec} is an event resulting from the execution of statement "RECEIVE (E)," and E evaluates to channel c, then e_{rec} must be preceded by a unique event e_{trans} 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 have that if e_1, e_2 are events resulting from the reception of messages M_1, M_2, and e'_1, e'_2 are the corresponding transmit events, then e_1 precedes e_2 implies e'_1 precedes e'_2.

The resulting semantics are nondeterministic in the sense that two simultaneous message transmissions by two processes over the same channel must arrive in sequential order, but we make no assumptions about this order. We note that the above semantics could also have been presented using a Plotkin-style operational semantics[15] or Petri nets (see Ref. 16), where analogous notions of reachability have been defined.

In flow graph model M_1 no assumptions are made about the order in which messages are received, and a receive statement does not delete a message from the specified buffer; it simply copies the message. Such semantics correspond to "bulletin board" style message delivery. Thus in M_1, *sets* of values from V are used to represent the message buffers, i.e., $b_c \in 2^V$. Here 2^V is the power set of V. Also, the definition of system evolution changes as follows:

(1) For $e =$ "TRANSMIT (E_1, E_2)," the new buffer state B' becomes B with b_c replaced by $b_c \cup m_i(E_2)$; i.e. the transmitted value is added to the previous set of values.

(2) For $e =$ "$X \leftarrow$ RECEIVE (E)," M' becomes M with m_i replaced by $m_i[v/X]$, for some $v \in b_c$; i.e. X gets bound to any one of the values in b_c. Since no message is deleted, B remains the same.

In Ref. 14, we consider the complexity of testing unreachability of flow graph nodes. In the general case of dynamic communication, we show that the reachability problem is undecidable. Even in the case of static communication, testing reachability is decidable but requires exponential space, infinitely often, in model M_0; and is NP-complete in M_1. In the next section we give an algorithm that provides a sufficient test for unreachability in models M_0 and M_1 under the assumption of static communication. This approximation has the advantage of being polynomial (linear) time.

3. EVENT SPANNING GRAPHS

In this section we present an algorithm that attempts to construct the *event spanning graph* of a system of communicating processes (not every system has one). When successful, the resultant event spanning graph will be used to guide the data flow analysis (Section 4). When unsuccessful, we can conclude that some flow graph node is unreachable.

A *message link* is an ordered pair of transmit and receive statements that specify the same channel. Static communication is assumed throughout this section: the channel argument to any transmit or receive statement labeling a flow graph node n is a constant $c(n)$. Thus we may statically determine the set ML of all message links. (Since the number of message links may be quadratic in the size of the process flow graphs, for efficiency the set ML is never explicitly constructed by our algorithm.)

Fix $G_i = \langle N_i, E_i, s_i \rangle$, $1 \leqslant i \leqslant r$, as the process flow graphs. Let $N = \bigcup_{1 \leqslant i \leqslant r} N_i$ be the set of all flow graph nodes. Then the *event spanning graph* is the directed acyclic graph $ESG\langle N, E \cup ML \rangle$ where $E = \bigcup_{1 \leqslant i \leqslant r} E_i$, such that:

(1) Let ESG_i be the subgraph of ESG induced by dropping all nodes but those of N_i and deleting all edges except those between nodes of N_i, $1 \leqslant i \leqslant r$. Then, ESG_i is a spanning tree of G_i.

(2) For each node $n \in N$ labeled by a receive statement, there is a path in ESG from some flow graph's start node to n containing a transmit statement communicating over the same channel as n.

Intuitively, the event spanning graph may describe, for each $n \in N$, an execution in which n may be reached. Restriction (1) insures that n is reachable within the flow graph containing n. Restriction (2) attempts to insure that at least one message is transmitted for each receive statement. In model M_1, this is sufficient to guarantee the reachability of all statements; in M_0, this is not necessarily the case. The event spanning graph is exemplified in Fig. 2.

Not every system of communicating processes has an event spanning graph. A minimal 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. An example blocking set is given in Fig. 3. It is easy to show:

Proposition 1. There is a nonempty blocking set N_B if there is no event spanning graph.

Lemma 1. For models M_0 and M_1, if there is no event spanning graph, then some flow graph node $n \in N$ is unreachable.

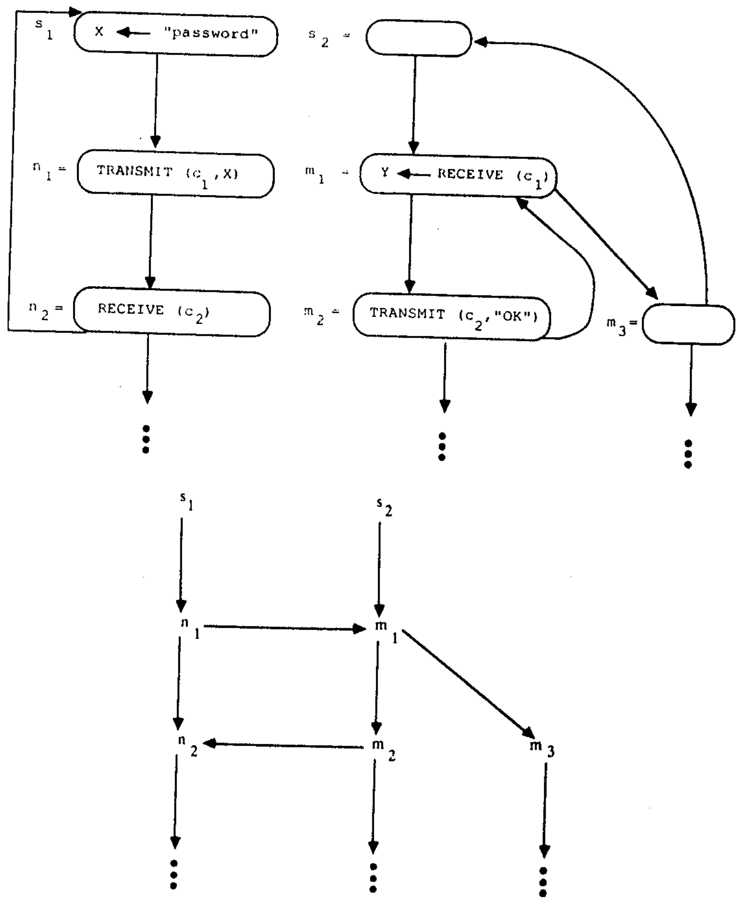

Fig. 2. Portions of flow graphs G_1 and G_2, with the corresponding portion of an event spanning graph *ESG*.

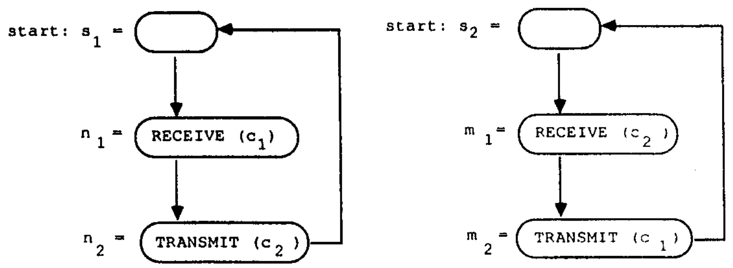

Fig. 3. The set $B = \{n_1, m_1\}$ is a blocking set of the program flow graphs illustrated.

Proof. By the Proposition, there must be a nonempty blocking set N_B. For the sake of contradiction, suppose some $n \in N_B$ is reached in some execution and no other element of N_B is reached 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. This contradicts the assumption that N_B is a blocking set. ▮

Simple counter examples shows that the converse of Lemma 1 does not hold. What follows is an algorithm for constructing either an event spanning graph or a blocking set.

Algorithm A

INPUT: Process flow graphs $G_i = <N_i, E_i, s_i>$, $1 \leq i \leq r$, with the set of program statements $N = \cup N_i$; and message channel set C.

OUTPUT: If an event spanning graph exists then its edge set ES, and otherwise a nonempty blocking set N_B (N_B may not be unique).

```
INITIALIZATION:
  $Q \leftarrow \emptyset$
  for each $n \in N$ do
    $\Gamma(n) \leftarrow \{m \in N \mid m$ immediately succeeds $n\}$
    if $n$ is a start node then
      add $n$ to $Q$
      $VISIT(n) \leftarrow$ true
    else
      $VISIT(n) \leftarrow$ false
  end for
```

```
    for each channel $c \in C$ do
        $WAIT(c) \leftarrow \underline{true}$
        $WAITING(c) \leftarrow \emptyset$
        $TALKER(c) \leftarrow \underline{null}$
    end for
    $ES \leftarrow \emptyset$

Procedure $VISIT\text{-}NEXT(n, S)$
    for each $m \in S$ do
        if $VISIT(m) = \underline{false}$ then
            $VISIT(m) \leftarrow \underline{true}$
            add $(n, m)$ to $ES$
            add $m$ to $Q$
        end if
end procedure

$MAIN\ LOOP$:
    until $Q = \emptyset$ do    -- $Q$ is the set of states still to be visited.
        choose and delete some $n \in Q$
        if $n$ is a transmit statement then
            $VISIT\text{-}NEXT(n, \Gamma(n))$
            if $WAIT(c(n))$ then
                $TALKER(c(n)) \leftarrow n$
                $Q \leftarrow Q \cup WAITING(c(n))$
                $WAITING(c(n)) \leftarrow \emptyset$
                $WAIT(c(n)) \leftarrow \underline{false}$
            end if
        end if
        if $n$ is a receive statement then
            if $WAIT(c(n))$ then
                add $n$ to $WAITING(c(n))$
            else
                add edge $(TALKER(c(n)), n)$ to $ES$
                $VISIT\text{-}NEXT(n, \Gamma(n))$
            end if
        if $n$ is neither a receive nor transmit statement then
            $VISIT\text{-}NEXT(n, \Gamma(n))$
    end until

$BLOCKING\ SET\ TEST$:
    for each channel $c \in C$ do
        $N_B \leftarrow N_B \cup WAITING(c)$
    if $N_B = \emptyset$ then
        return event spanning graph $ESG = <N, ES>$
    else
        return blocking set $N_B$
```

In Algorithm A, a node $n \in N$ has been "visited" if $VISIT(n)$ has been set to *true*. Initially, only the start nodes have been visited. It is easy to verify that for each channel $c \in C$, if *no* transmit statement over c has been visited, then

(1) $TALKER(c) = null$

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

Otherwise, when n is the first transmit statement visited with $c(n) = c$, then

(1′) $TALKER(c)$ is set to n, and

(2′) Each receive statement m in $WAITING(c)$ is put back in Q, and $WAITING(c)$ is set to $\varnothing$. When each such m is revisited, edge (n, m) will be added to ES. Similarly, edge (n, m') is added to ES for any other receive statements m' over channel c visited subsequently.

Theorem 1. If an event spanning graph exists, then Algorithm A returns one; else it returns a nonempty blocking set.

Proof. It is easy to verify that if $N_B = \varnothing$, Algorithm A returns an event spanning graph. On the other hand, suppose Algorithm A returns a nonempty blocking set $N_B = \bigcup_{c \in C} WAITING(c)$. We claim this is a blocking set. Suppose not. Then there exists a receive statement $n \in N_B$, a transmit statement m communicating over the same channel c as n, and a path p in a flow graph G_i from start node s_i to m avoiding all elements of N_B. In Algorithm A, s_i is initially visited and added to Q. Furthermore, all other elements of p are also eventually visited and added to Q, including m. Thus $WAITING(c)$ is eventually set to $\varnothing$ and remains empty till termination. This contradicts the assumption that $n \in N_B$.

Termination of Algorithm A is guaranteed since an element $n \in N$ is added to Q at most twice, and an element of Q is deleted at each iteration of the until loop. In particular, a non-receive statement is added to Q at most once, and a receive statement may be added twice, the second time occurring after an appropriate talker has been found. ▌

As alluded to above, the computational complexity of Algorithm A is $O(|N| + |E|)$ time. This can be seen by noticing that the main loop is executed $O(|N|)$ times, and that $O(|N|)$ message links are added to ESG.

4. DATA FLOW ANALYSIS OF COMMUNICATING PROCESSES WITH STATIC COMMUNICATION

Data flow analysis yields information about a program. This information is in general too weak for program correctness proofs, but it is sufficient for the usual compile-time 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. (Hecht[8] is an informative text on data flow analysis of sequential programs.) Here, we extend such data flow analysis techniques to the analysis of concurrent programs with static communication. Hence our value domain for program variables is simply $V = Num$; i.e. variables cannot take on channel names as values.

4.1. A Data Flow Analysis Framework

We present a general data flow analysis framework similar to the ones in Refs. 8, 17 and 18. We then customize this framework for the purpose of determining the values of program expressions in a system of communicating processes. The results we obtain are as strong as possible for model M_1, where message delivery is unordered, in that one cannot do better with symbolic execution. For model M_0, where messages are delivered in order, our results are "conservative" (see Section 4.3).

Let D be a set of predicates. We assume a semi-lattice $(D, \bigsqcup)$, where $\bigsqcup: D^2 \rightarrow D$, the usual lattice join operation, is associative, commutative, and idempotent with respect to D. We require $\bigsqcup$ to be the following weakening of logical disjunction:

$$\forall p, q \in D\text{: if } p \text{ or } q \text{ holds, then } p \sqcup q \text{ holds.}$$

The lattice partial order $\sqsubseteq$ is defined as:

$$\forall p, q \in D\text{: } p \sqsubseteq q \text{ iff } p \sqcup q = q.$$

Note that $\sqsubseteq$ is the restriction of logical implication to D:

$$\forall p, q \in D\text{: } p \sqsubseteq q \text{ iff } p \text{ implies } q.$$

We assume that D contains $FALSE$ and $TRUE$ as minimum and maximum values, respectively. So $FALSE \sqsubseteq p$ and $p \sqsubseteq TRUE, \forall p \in D$. Also, we define the strict ordering $\sqsubset$ such that:

$$p \sqsubset q \text{ holds in } D \text{ iff } p \sqsubseteq q \text{ and } p \neq q.$$

We assume that $(D, \bigsqcup)$ is *of finite length*, containing no infinite strictly increasing chains $p_1 \sqsubset p_2 \sqsubset \cdots$. Finally, we say that a function $f: D \to D$ is *monotonic* if $f(p) \sqsubseteq f(p')$ for all p, p' such that $p \sqsubseteq p'$. Similarly, $g: D^2 \to D$ is monotonic if $g(p, q) \sqsubseteq g(p', q')$, for all p, p' and $q, q' \in D$ such that $p \sqsubseteq p'$ and $q \sqsubseteq q'$.

For the *flow analysis of communicating processes*, we consider a family of domains D_M, each member of which is a domain of *membership functions*. Let *Var* be a set of program variables, V the set of values over which these variables may range, and S a subset of V. Then $D_M(Var)$ is the domain of functions $p: Var \to 2^V$ such that $p(X) = S$ means that during execution, the values assumed by program variable X are members of the set S. We often represent p, in the obvious way, as the set of pairs $\{(X, S)\}$, and refer to S as the "value set" of X.

Note that D_M is parameterized by the set of variable names over which a membership function may range. For systems of communicating processes $\{P_1, \ldots, P_r\}$, we will consider domains $D_M(Var_1), \ldots, D_M(Var_r)$.

We assume that all membership functions p in $D_M(Var)$ are *total*, and that for each variable $X \in Var$, $|p(X)|$ is bounded by a constant k_0. The latter assumption is necessary to ensure termination of our data flow analysis algorithm. We believe that this assumption is justifiable since many distributed programs are finite-state, synchronization and coordination protocols in particular. If not, then various approximation techniques can be used. For example, by modifying our approach slightly, it can be used to determine whether the value of a program variable is *undefined*, equal to a constant value c, or *nonconstant*.

Given $p, q \in D_M(Var)$, the join operator $\bigsqcup_M$ produces the function:

$$p \sqcup_M q(X) = p(X) \cup q(X), \forall X \in Var.$$

The relation $p \sqsubseteq_M q$ holds for $p, q \in D_M(Var)$ if for all X in *Var*, $p(X) \subseteq q(X)$. Since the domain and range of each function in $D_M(Var)$ is of bounded size, the semilattice $(D_M(Var), \bigsqcup_M)$ is of finite length. Its least element is the constant function $\varnothing$, and its greatest element is the constant function V.

4.2. Predicate Transformers for Flow Analysis of Communicating Processes

With each node n of a process flow graph we need to associate a function that produces an output membership function from an input membership function. We can think of this function as a "predicate transformer," since it transforms predicates of the form $value(X) \in S$ into predicates of the form $value(X) \in S'$. The nature of the transformer depends, of course, on the type of statement labeling n. As we will show (Proposition 2), all transformers are monotonic functions.

Notation. Let E denote an expression having free variables $X_1, \ldots, X_k$. Then, $E[v_1/X_1, \ldots, v_k/X_k]$ denotes the value of E when X_i is bound to v_i, $1 \leqslant i \leqslant k$.

Let n be labeled by the assignment statement "$X \leftarrow E$," where E is an expression involving program variables $X_1, \ldots, X_k$. We associate with n the function $\Delta_n: D_M(Var) \to D_M(Var)$, weakly describing the change of state on execution of program statement n. (Δ_n is often referred to as the "transfer function" of n.) More precisely, if $p \in D_M(Var)$ holds just before the execution of 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 Hoare logic.) Δ_n is defined as:

$$\Delta_n(p) = p[S/X] \text{ (see Section 2.2 for an explanation of this notation)}$$

where $S = \{E[v_1/X_1, \ldots, v_k/X_k] \mid v_i \in p(X_i), 1 \leqslant i \leqslant k\}$.

Intuitively, the new value set of X is obtained by taking all possible evaluations of E based on the incoming value sets of $X_1, \ldots, X_k$. For example, let n be labeled by "$X \leftarrow Y + X$." Then

$$\Delta_n(\{(X, \{1, 2\}), (Y, \{3\})\}) = \{(X, \{4, 5\}), (Y, \{3\})\}.$$

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.

In order to describe the functions associated with nodes labeled by transmit or receive statements, we introduce an *auxiliary variable* $M_c \in V$, for each channel $c \in C$. Intuitively, $M_c = S$ means that channel c may contain any of the values in S, $S \subseteq V$. Furthermore, just as we had a separate domain of membership functions $D_M(Var_i)$ for each process P_i, we will have a separate domain of membership functions for each channel c. Let AV_c denote the singleton set containing auxiliary variable M_c. Then $D_M(AV_c)$ is the domain of membership functions $q: AV_c \to 2^V$. Our objective will be to compute a q such that $q(M_c)$ records the "history" of all values transmitted over channel c during the lifetime of the program. Membership over auxiliary variables directly mimic the message passing semantics of model M_1, where a set rather than a queue is used as the channel data structure.

Let n be labeled by the transmit statement "TRANSMIT (c, E)," where $c \in C$ and E is an expression as before. We associate with n the function $\tau_n: D_M(Var) \rightarrow D_M(AV_c)$, describing the value of the message transmitted by n. Let p belong to $D_M(Var)$. Then

$$\tau_n(p) = q \text{ such that } q(M_c) = \{E[v_1/X_1, \ldots, v_k/X_k] \mid v_i \in p(X_i), 1 \leqslant i \leqslant k\}.$$

Intuitively, the value of the message transmitted may be any of the possible evaluations of E. For example, let n be labeled by "TRANSMIT $(c, X * Y)$." Then

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

Finally, let n be labeled by the receive statement "$X \leftarrow$ RECEIVE (c)," where $c \in C$. We associate with n the function $\rho_n: D_M(Var) \times D_M(AV_c) \rightarrow D_M(Var)$, such that if $p \in D_M(Var)$ and $q \in D_M(AV_c)$ hold on input to n, then $\rho_n(p, q)$ holds on output from node n:

$$\rho_n(p, q) = p[q(M_c)/X].$$

Intuitively, X can assume any of the values ever transmitted over channel c. For example, let n be labeled by "$X \leftarrow$ RECEIVE (c)." Then

$$\rho_n(\{(X, \{1, 2\}), (Y, \{5\})\}, \{(M_c, \{3, 4\})\}) = \{(X, \{3, 4\}), (Y, \{5\})\}.$$

Propositon 2. Functions Δ_n, τ_n, and ρ_n are monotonic, for all flow graph nodes n.

Proof. Consider first $\Delta_n: D_M(Var) \rightarrow D_M(Var)$ for node n labeled by "$X \leftarrow E$." Assume that variables $X_1, \ldots, X_k$ appear free in E. Let $p, q \in D_M(Var)$ such that $p \sqsubseteq_M q$. Since $p \sqsubseteq_M q$ implies $p(X_1) \subseteq q(X_1), \ldots, p(X_k) \subseteq q(X_k)$, we have that $\Delta_n(p)(X) \subseteq \Delta_n(q)(X)$. Given that the function returned by $\Delta_n(p)$ is the same as p except possibly at X, we have shown that $\Delta_n(p) \sqsubseteq_M \Delta_n(q)$.

The proof for τ_n is similar for that of Δ_n. Consider next $\rho_n: D_M(Var) \times D_M(AV_c) \rightarrow D_M(Var)$ for node n labeled by "$X \leftarrow$ RECEIVE (c)." Let $p, p' \in D_M(Var)$ and $q, q' \in D_M(AV_c)$, such that $p \sqsubseteq_M p'$ and $q \sqsubseteq_M q'$. Since $q \sqsubseteq_M q'$ implies $q(M_c) \subseteq q'(M_c)$, we have that $\rho_n(p, q)(X) \subseteq \rho_n(p', q')(X)$. Given that the function returned by $\rho_n(p, q)$ is the same as p except possibly at X, we have shown that $\rho_n(p, q) \sqsubseteq_M \rho_n(p', q')$. ▮

4.3. Data Flow Analysis Algorithm

The objective of our data flow analysis is to compute for each program statement $n \in N$, the membership functions IP_n (Input Predicates) and OP_n

(Output Predicates) belonging to $D_M(Var_i)$, assuming n is a node of flow graph G_i, where

(i) IP_n holds on input to n for all executions.

(ii) OP_n holds on output from n for all executions.

Let C be the set of channels occurring as arguments to transmit and receive statements. For each channel $c \in C$, we wish also to compute $MP_c \in D_M(AV_c)$ (Message Predicate), recording all messages sent over channel c.

Specifically, we wish to compute IP_n and OP_n for each flow graph node $n \in N$, and MP_c for each channel $c \in C$, satisfying the following recursive equations:

(1) $IP_n = \bigsqcup_M OP_m$, where the join is taken over the set $\Gamma^{-1}(n)$ of all immediate predecessors m of n.

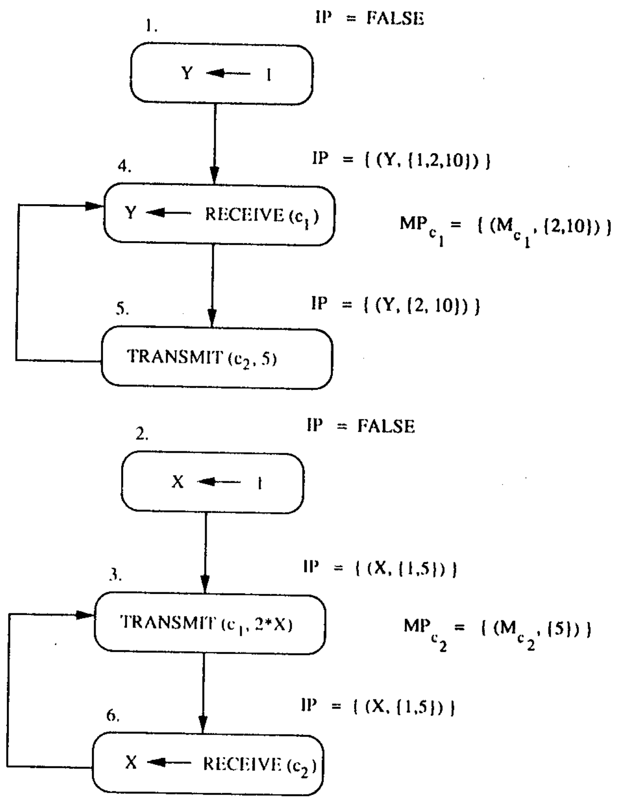

Fig. 4. An example program annotated with minimal solutions to data flow equations (1)–(3).

(2) $OP_n = IP_n$ if n is a node labeled by a transmit statement;
$= \rho_n(IP_n, MP_{c(n)})$ if n is labeled by a receive statement;
$= \Delta_n(IP_n)$ otherwise.

(3) $MP_c = \bigsqcup_M \tau_n(IP_n)$, where the join is taken over all transmit statements $n \in N$ with channel argument $c = c(n)$.

It is easy to show that this system of equations does not have a unique solution. We desire the *least* solution, which gives us the most specific information about the values of program variables. We obtain this iteratively, starting with an approximation that is too small and work up. An example of the data flow analysis problem is given in Fig. 4.

Recall that an event spanning graph of the last section is acyclic and has node set N, the set of all flow graph nodes. A *topological ordering* of an acyclic directed graph is a total ordering of its node set that is consistent with its original partial order. A topological ordering can be easily computed in linear time as described in Ref. 19.

We now present an algorithm which repeatedly computes approximations to the previous data flow analysis equations. In each pass, the algorithm visits each node in topological order of an event spanning graph. Any topological ordering may be used; in fact, any ordering will suffice. However, very often convergence is obtained more quickly if a topological ordering of the event spanning graph is used. This point is discussed further below.

Algorithm B

INPUT: Process flow graphs $G_i = <N_i, E_i, s_i>$, $1 \leq i \leq r$, with the set of program statements $N = \cup N_i$; message channel set C; an event spanning graph ESG; and the functions Δ_n, τ_n, ρ_n defined appropriately for each $n \in N$.

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

```
INITIALIZATION:
  for each $n \in N$ do
    $IP_n \leftarrow FALSE$
    $OP_n \leftarrow FALSE$
  end for

  for each channel $c \in C$ do
    $MP_c \leftarrow FALSE$

MAIN LOOP:
  until no change in any $IP_n$, $OP_n$, $MP_c$ do
```

```
    for each $n \in N$ in topological order of $ESG$ do
      for each immediate predecessor $m$ of $n$ do
        $IP_n \leftarrow IP_n \sqcup_M OP_m$
      if $n$ is a transmit statement then
        $OP_n \leftarrow IP_n$
        $MP_{c(n)} \leftarrow MP_{c(n)} \sqcup_M \tau_n(IP_n)$
      end if
      if $n$ is a receive statement then
        $OP_n \leftarrow \rho_n(IP_n, MP_{c(n)})$
      if $n$ is neither a receive nor transmit statement then
        $OP_n \leftarrow \Delta_n(IP_n)$
    end for
  end until
```

The behavior of Algorithm B is illustrated in Table I on the data flow analysis problem of Fig. 4. In Fig. 4, the numbering of the nodes indicates the topological sort of the network's event spanning graph that we use to carry out the analysis. We use this node ordering to associate a node with each row of the table. The columns correspond to the various membership predicates. Table I shows how the values of the membership predicates change as the nodes are visited in topological order, iteration by iteration. Within an iteration, each table entry contains the value of the membership predicate immediately after the calculation of the node's input membership

Table I. Simulation of Algorithm B on the data flow analysis problem of Fig. 4.

	IP_3	IP_4	IP_5	OP_5	IP_6	OP_6	MP_{c_1}	MP_{c_2}
3	{1}	∅	∅	∅	∅	∅	∅	∅
4		{1}					{2}	
5			{2}					
6				{2}	{1}			{5}
3	{1, 5}					{5}		
4		{1, 2}					{2, 10}	
5			{2, 10}					
6				{2, 10}	{1, 5}			
3								
4		{1, 2, 10}						
5								
6								

predicate and just before the application of the node's predicate transformer. Only nodes 3 through 6 are considered (the behavior of nodes 1 and 2 is not very interesting), and only a single value set is given for each membership predicate. This should cause no confusion as the first process has only the program variable Y, and the second process has only the program variable X. For readability, a membership predicate's value set is given only when it assumes a new value.

Table I contains the execution of the first three iterations of the algorithm; there is no change to any of the membership predicates in iteration 4, after which the algorithm terminates.

Theorem 2. Algorithm B is correct: it always terminates and, upon termination, data flow analysis equations (1)–(3) are satisfied.

Proof. Data flow equations (1)–(3), directly encoded by the if statements of the main loop, define a system of mutually recursive equations over the semilattice $(D_M, \bigsqcup_M)$. Furthermore, all operators in this system are monotonic (Proposition 2). Since all variables (i.e. IP_n, OP_n, and MP_c, $n \in N$, $c \in C$) are initialized to $FALSE$ (the least element of the domain), fixed-point iteration must yield their least fixed points. The finite length of $(D_M, \bigsqcup_M)$ ensures that this iteration is finite. ▮

To determine the computational complexity of Algorithm B, let ML be the set of all message links, let E be the set of all edges in the process flow graphs $G_1,\ldots, G_r$, and let K be the total number of program and channel variables, i.e. $K = \sum_{1 \leqslant i \leqslant r} |Var_i| + |C|$. Clearly, each iteration of the until loop requires $O(|N| + |E|)$ operations. In the worst case, Algorithm B converges after $|N|\,\lambda$ iterations, where $\lambda = K\,|V|$. Intuitively, it takes at most $|N|$ iterations to propagate a newly generated value from one flow graph node to any other, and in the worst case this value increases the size of exactly one variable's value set. Thus, the total time cost is at most $O(|N|(|N| + |E|)\,\lambda)$.

We expect that the number of iterations actually performed by Algorithm B to be considerably less than $|N|\,\lambda$. Consider a value v generated (by an assignment statement) at flow graph node n. By using an event spanning graph ESG, the number of iterations required to propagate v locally, i.e. to a node m where n and m are in the same flow graph G_i, is minimized. This is because the subgraph of ESG on nodes and edges only in G_i forms a $DFST$ (depth-first-search spanning tree) of G_i. From Ref. 20, we know that v can be propagated from n to m in at most d_i iterations, where d_i is the depth of G_i. (Let D be a DFST of a flow graph G. Then the depth of G is the largest number of retreating edges along any cycle-free path in G.)

Regarding the propagation of value v globally, i.e. to a node m in a flow graph different from the one in which n resides, each message link in *ESG* reduces by one the requisite number of iterations.

As mentioned earlier, the results obtained using Algorithm B are as strong as possible for model M_1. That is, they cannot be improved upon by any other symbolic execution technique. This is not necessarily true for model M_0 where it is assumed that messages are received in transmitted order: variables MP_c are sets and thus do not maintain the ordering information. Our results are, however, *conservative*: $IP_n(X) = S$ guarantees that, upon entry to flow graph node n, the values assumed by X during execution form a subset of S. In terms of our semantics of Section 2.2, we require that, for any reachable global state $S = \langle [n_1,..., n_r], [m_1,..., m_r], B \rangle$ such that n is an immediate successor of n_i, $m_i(X) \in IP_n(X)$, where $X \in Var_i$ and $n \in N_i$.

To see why Algorithm B is conservative, refer to the data flow equations given at the beginning of the section (Algorithm B computes the least fixed point of these equations). Equations (1) and (2), part 3, encode the local propagation of data within a process due to local flow of control (flow graph edges) and assignment statements. Equations (2), parts 1 and 2, and (3) encode the global propagation due to message passing. The accounting of this global propagation is certainly conservative as $M_{c(n)}$ will contain all values ever transmitted over $c(n)$.

5. DATA FLOW ANALYSIS IN THE CASE OF DYNAMIC COMMUNICATION

In the previous section we assumed that all message channel arguments of transmit and receive statements are constants. We are able to further refine our data flow analysis techniques to the case of *dynamic communication*: where the channel arguments of the communication primitives are expressions that must evaluate to channels but not necessarily the same channel on all executions. Our analysis is made more difficult by the fact that 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. In NIL,[2] a language for programming "secure" systems, this viewpoint is taken to the extreme: a process P *cannot* communicate with a process Q unless Q explicitly passes P a port over which communication can take place.

Since we are dealing with dynamic communication, the value set over which program variables may range is $V = Num \cup C$. As before, C is the set of all channels over which processes might communicate. Let τ_n, ρ_n, and

Δ_n be our predicate transformers as defined in Section 4. We assume for each transmit or receive statement $n \in N$ a function $\gamma_n: D_M(Var) \to 2^C$ such that if $p \in D_M(Var)$ holds on input to n, then $\gamma_n(p)$ hold on output from n. Let Y be the variable occurring as the channel argument to n, e.g. n is labeled by "$X \leftarrow$ RECEIVE (Y)." Then, simply,

$$\gamma_n(p) = p(Y).$$

Approximation is an essential technique in gobal flow analysis. Here we use γ_n to approximate the possible channels over which n may communicate. More formally, given a membership function $p \in D_M(Var)$ weakly describing the state just before execution of n, $\gamma_n(p)$ must contain at least all channels over which n may communicate. For example, let n be labeled by "TRANSMIT $(Y, X+Z)$." Then

$$\gamma_n(\{(Y, \{c_1, c_2\}),...\}) = \{c_1, c_2\}.$$

As expected, γ_n is monotonic for all flow graph nodes n. The proof is simple and follows the proof of Proposition 2.

For dynamic communication, we seek minimal $IP_n, OP_n \in D_M(Var)$ for all flow graph nodes $n \in N$, and $MP_c \in D_M(AV_c)$ for all channels $c \in C$, satisfying:

(1′) $IP_n = \bigsqcup_M OP_m$, where the join is taken over the set $\Gamma^{-1}(n)$ of all immediate predecessors m of n.

(2′) $OP_n = IP_n$ if n is a node labeled by a transmit statement;
$= \bigsqcup_M \rho_n(IP_n, MP_c)$ if n is labeled by a receive statement, where the join is taken over all $c \in \gamma_n(IP_n)$;
$= \Delta_n(IP_n)$ otherwise.

(3′) $MP_c = \bigsqcup_M \tau_n(IP_n)$, where the join is taken over all transmit statements $n \in N$ with $c \in \gamma_n(IP_n)$.

For any solution to the above data flow equations, IP_n and OP_n are predicates in $D_M(Var)$ holding on input and output, respectively to flow graph node $n \in N$; and MP_c holds for all messages transmitted over channel $c \in C$. Note that the previous equations differ from those given in Section 4 for static communication, only in that we use $\gamma_n(IP_n)$ to estimate the channels over which each transmit and receive statement n may communicate. An example of the data flow analysis problem for dynamic communication is given in Fig. 5.

The following algorithm yields a minimal solution to Eqs. (1′)–(3′) assuming as before that the data flow domain $(D_M, \bigsqcup_M)$ has no infinitely strictly increasing chains. Our algorithm combines Algorithms A and B

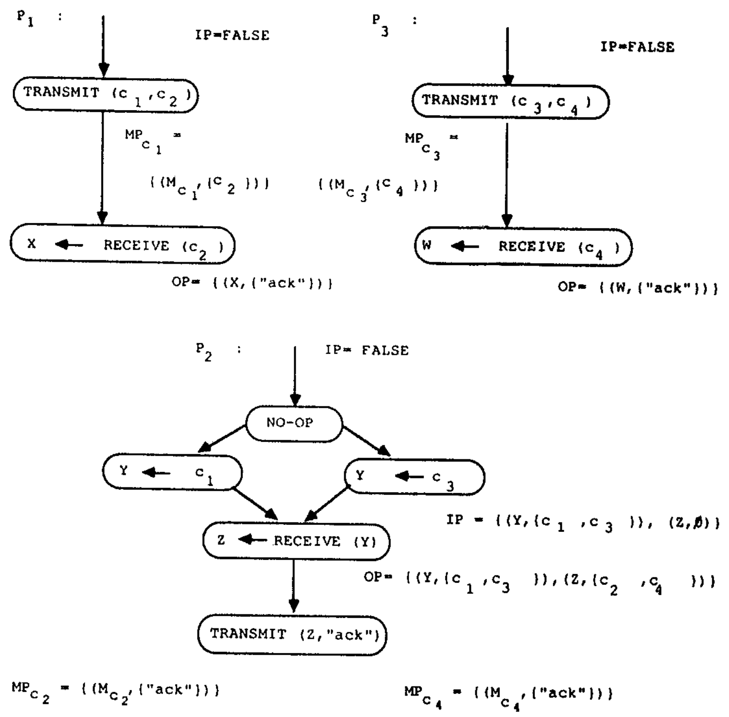

Fig. 5. An example program annotated with minimal solutions to data flow equations (1′)–(3′).

of the previous sections, building an event spanning graph *ESG* while simultaneously carrying out the data flow analysis. As in Algorithm B, the event spanning graph is used to order the nodes through which we propagate data flow information. The procedure *VISIT-NEXT* is the same as in Algorithm A.

Algorithm C

INPUT: Process flow graphs $G_i = <N_i, E_i, s_i>$, $1 \leq i \leq r$, with the set of program statements $N = \cup N_i$; message channel set C; and the functions Δ_n, τ_n, ρ_n, and γ_n defined appropriately for each $n \in N$.

OUTPUT: Event spanning graph $ESG = <N, ES>$ if it exists, with minimal IP_n, OP_n, MP_c, for all $n \in N$, $c \in C$, satisfying data flow equations (1′), (2′), and (3′); otherwise a nonempty blocking set N_B.

```
INITIALIZATION:
  $Q \leftarrow \emptyset$
  for each $n \in N$ do
    if $n$ is a start node then
      add $n$ to $Q$
      $VISIT(n) \leftarrow \underline{true}$
    else
      $VISIT(n) \leftarrow \underline{false}$
    $IP_n \leftarrow FALSE$
    $OP_n \leftarrow FALSE$
  end for
  for each channel $c \in C$ do
    $WAIT(c) \leftarrow \underline{true}$
    $WAITING(c) \leftarrow \emptyset$
    $TALKER(c) \leftarrow \underline{null}$
    $MP_c \leftarrow FALSE$
  end for
  $ES \leftarrow \emptyset$

function wait(n)
  $wait \leftarrow \bigwedge_c WAIT(c)$, for each $c \in \gamma_n(IP_n)$
end function

Procedure PROPAGATE(n)
  for each immediate predecessor $m$ of $n$ do
    $IP_n \leftarrow IP_n \sqcup_M OP_n$
  if $n$ is a transmit statement then
    $OP_n \leftarrow IP_n$
    $VISIT\text{-}NEXT(n, \Gamma(n))$
    for each $c \in \gamma_n(IP_n)$ do
      $MP_c \leftarrow MP_c \sqcup_M \tau_n(IP_n)$
      if $WAIT(c)$ then
        $TALKER(c) \leftarrow n$
        $Q \leftarrow Q \cup WAITING(c)$
        $WAITING(c) \leftarrow \emptyset$
        $WAIT(c) \leftarrow \underline{false}$
      end if
    end for
  end if
  if $n$ is a receive statement then
    if $wait(n)$ then
      for each $c \in \gamma_n(IP_n)$ do
        add $n$ to $WAITING(c)$
    else
      for each $c \in \gamma_n(IP_n)$ such that $\neg WAIT(c)$ do
        $OP_n \leftarrow OP_n \sqcup_M \rho_n(IP_n, MP_c)$
        add edge $(TALKER(c), n)$ to $ES$
      end for
      $VISIT\text{-}NEXT(n, \Gamma(n))$
    end if
  end if

  if $n$ is neither a receive nor transmit statement then
    $OP_n \leftarrow \Delta_n(IP_n)$
    $VISIT\text{-}NEXT(n, \Gamma(n))$
  end if
end procedure


MAIN LOOP:
  until no change in any $IP_n$, $OP_n$, $MP_c$ do
    until $Q = \emptyset$ do
      choose and delete some $n \in Q$
      $PROPAGATE(n)$
    end until

    for each node $n$ of $(N, ES)$ in topological order do
      $PROPAGATE(n)$
  end until


BLOCKING SET TEST:
  for each channel $c \in C$ do
    $N_B \leftarrow N_B \cup \{n \mid n \in WAITING(c) \wedge wait(n) = \underline{true}\}$
  if $N_B \neq \emptyset$ then
    return blocking set $N_B$
  else
    return event spanning graph $ESG = <N, ES>$
      and $IP_n$, $OP_n$, $MP_c$ for each $n \in N$ and $c \in C$.
```

Theorem 3. On convergence, Algorithm C either outputs a non-empty blocking set, or an event spanning graph is output and data flow Eqs. (1′)–(3′) are satisfied. Moreover, Algorithm C always terminates.

Proof. Algorithm C is a generalization of both Algorithms A and B. We examine its correctness first in terms of event-spanning-graph construction, and then in terms of data flow analysis. In the case of dynamic communication, a receive statement n is reachable if a talker (a reachable transmit statement) can be found for any c in $\gamma_n(IP_n)$. Accordingly, if no such talker has been encountered before visiting n, then $wait(n)$ (initially true) is still true and n is added to each $WAITING\ (c)$. Otherwise, at least one talker has been found, and each such talker is used to construct a message link with n.

Similarly, a transmit statement n can serve as a talker for each c in $\gamma_n(IP_n)$. Thus, upon encountering n, the appropriate actions (see also the discussion of Algorithm A) are taken to make n the talker for all c in $\gamma_n(IP_n)$ such that $WAIT\,(c)$ is still true.

Finally, when checking for a nonempty blocking set, we must disregard receive statements in $WAITING\,(c)$ with $wait(n)=false$. These statements have found a talker for a channel c' in $\gamma_n(IP_n)$, $c \neq c'$. We can now argue, as in the proof of Theorem 1, that Algorithm C returns an event spanning graph if one exists; else it returns a nonempty blocking set.

Regarding data flow analysis, procedure *PROPAGATE* directly encodes data flow Eqs. (1′)–(3′). *PROPAGATE* is called from both the until loop and for loop of the main loop of Algorithm C. In the until loop, data flow analysis is performed in conjunction with event-spanning-graph construction. Upon termination of the until loop, those receive statements that blocked the construction of the event spanning graph will be contained in the various *WAITING* sets. In the for loop, data flow analysis is performed in topological order of the so-far-constructed event spanning graph. During this loop it is possible that blocked receive statements will become unblocked due to the discovery of qualified talkers. Thus, upon completion of the for loop, these unblocked receive statements will have been added to Q. The until loop will then continue its event-spanning-graph construction/data flow analysis starting from these unblocked nodes.

Note that once an event spanning graph has been constructed, the subsequent iterations of the main loop will involve the for loop only. Termination of Algorithm C is guaranteed since nodes are added to Q at most twice in the until $Q=\varnothing$ loop, and each iteration of this loop deletes a node from Q. Concerning the termination of the main until loop, data flow Eqs. (1′)–(3′) still (as a result of the monotonicity of γ_n) define a system of monotonic recursive equations in the IP_n, OP_n, and MP_c variables. Fixed-point iteration must yield their least fixed points, and the finite length of $(D_M, \bigsqcup_M)$ again ensures the finiteness of this iteration. ∎

Regarding the computational complexity of Algorithm C, each iteration of the main until loop requires $O(|N|^2+|E|)$ operations (versus the $O(|N|+|E|)$ operations needed per iteration of Algorithm B) since $\sum_{n\in N}|\gamma_n(IP_n)|=O(|N|^2)$. That is, there may be $O(|N|)$ transmit/receive nodes n each with $|\gamma_n(IP_n)|=O(|N|)$. In the worst case, Algorithm C, like Algorithm B, converges in $O(|N|\,\lambda)$ iterations of the main until loop, since the "dynamic" calculation of the event spanning graph delays convergence only by an additive factor of $O(|N|)$ iterations.

6. CONCLUSIONS

We have shown that data flow analysis techniques for sequential programs can be extended to concurrent programs in which processes communicate through message passing. To achieve this, we introduced the event spanning graph, a generalization of the (depth-first) spanning tree used in flow analysis of sequential programs. The event spanning graph serves to order the nodes through which data flow information is propagated, both within a process and across process boundaries. Importantly, if no event spanning graph exists, then some program statement is unreachable.

We presented two data flow analysis algorithms: one for static communication (all channel arguments are constants) and one for the harder, dynamic communication (channel arguments may be variables and channels may be passed as messages). Our algorithms detects the values of program expressions within a system of distributed processes and can be used to place bounds on the size of variables and messages. We intend to investigate the possibility of computing a more accurate approximation for models in which message queues are ordered and messages are deleted, without greatly sacrificing efficiency.

It would be interesting to develop direct (non-iterative) data flow analysis algorithms which run very efficiently when the process' programs and their communications are well-structured; or when message transmissions can be simply modeled as procedure calls. In the latter case, known interprocedural data flow analysis methods such as Refs. 21 and 22 would be of use. Note that in general it would be quite misleading to identify processes and procedures, and messages and procedure calls, since the semantics of procedure calls is entirely sequential and a message transmission does not evoke a process execution.

APPENDIX A: GRAPH-THEORETIC DEFINITIONS

We consider *directed graphs* (N, E) consisting 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. Let $\Gamma(n)$ be the set of successors of node n. A graph (N', E') is a subgraph of (N, E) if $N' \subseteq N$ and $E' \subseteq E$. A *path* p of *length* k from n to m is a sequence of nodes $p = (n = n_0, n_1, \ldots, n_k = m)$ such that $(n_i, n_{i+1}) \in E$ for $0 \leqslant i \leqslant k$. The path is

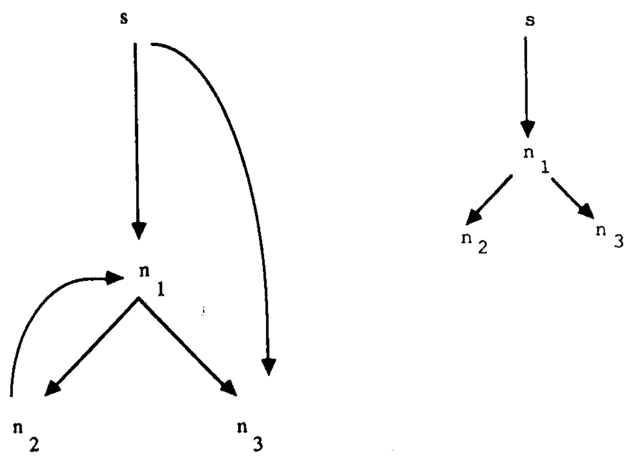

Fig. 6. Example: a flow graph G and spanning tree of G.

simple if $n_0, \ldots, n_k$ are distinct (except possibly $n_0 = n_k$) and the path is *cyclic* if $n_0 = n_k$. A graph is *acyclic* if it contains no cycles.

A *flow graph* $G = (N, E, s)$ is a directed graph (N, E) with a distinguished *start node* s such that, for any node $n \in N$, there is a path from s to n. A (*directed, rooted*) *tree* $T = (N, E, s)$ is a flow graph with $|E| = |N| - 1$. The start node s is the *root* of the tree. Any tree is acyclic, and if n is a node in tree T, then there is a unique path from s to n. If n and m are nodes in a tree T and there is a path from n to m, then n is an *ancestor* of m and m is a *descendant* of n. In a tree, each node has a unique predecessor called its *parent* (except the root which has no predecessor). The successors of a node in a tree are its *children*. 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, $N = N'$, and $s = s'$, then T is a *spanning tree* of G. See Fig. 6 for an example.

ACKNOWLEDGMENTS

The authors are grateful to the anonymous referees for their careful reading of the manuscript and the resulting constructive comments. They would also like to thank to Gene Stark and S. Purushothaman for valuable discussions on data flow analysis of communicating processes.

REFERENCES

1. C.A.R. Hoare, Communicating sequential processes, *Comm. ACM* **21**(8):666–677 (1978).
2. R. E. Strom and N. Halim, A new programming methodology for long-lived software systems, *IBM J. Research and Development* **28**(1):52–59 (1984).

3. U. Engberg and M. Nielsen, *A Calculus of Communicating Systems with Label-Passing*, Report DAIMI PB-208, Computer Science Department, University of Aarhus (1986).
4. R. Milner, J. Parrow, and D. Walker, *A Calculus of Mobile Processes, Part I*, Technical Report ECS-LFCS-89-85, Department of Computer Science, University of Edinburgh (June 1989).
5. J. A. Feldman, A programming methodology for distributed computing (among other things), *Comm. ACM* **22**(6):353–368 (1979).
6. J. H. Reif and P. Spirakis, Distributed algorithms for synchronizing interprocess communication within real time, *Proc. 13th ACM Symp. Theory of Computation*, Madison, Wisconsin 133–145 (1981). Also rewritten as Real-time synchronization of interprocess communications, TR-25-82, Aiken Computation Lab, Harvard Univ., Cambridge, Massachusetts (1982).
7. J. H. Reif, Dataflow analysis of communicating processes, *Proc. 6th ACM Symp. Principles of Programming Languages*, pp. 257–268 (June 1979).
8. M. S. Hecht, *Data Flow Analysis of Computer Programs*, American Elsevier, New York (1977).
9. M. S. Hecht and J. D. Ullman, A simple algorithm for global data flow analysis problems, *SIAM J. Comput.* **4**(4):519–532 (1975).
10. P. Cousot and R. Cousot, Semantic analysis of communicating sequential processes, in *Proc. 7th Int'l Colloq. on Automata, Languages and Programming*, Lecture Notes in Computer Science **85**:119–133, Springer-Verlag (1980).
11. W. Peng and S. Purushothaman, Towards data flow analysis of communicating finite state machines, *Proc. 8th ACM Symp. Principles of Distributed Computing* (August 1989).
12. R. D. Schlichting and F. B. Schneider, Understanding and using asynchronous message passing, *Proc. 1st ACM Symp. Principles of Distributed Computing*, Ottawa, Canada, pp. 141–147 (August 1982).
13. P. F. Kearney, *Reasoning about Nondeterministic Data Flow*, Ph.D. Thesis, Department of Computer Science, University of Queensland, Australia (July 1988).
14. J. H. Reif and S. A. Smolka, The complexity of reachability in distributed communicating processes, *Acta Informatica* **25**:333–354 (1988).
15. G. D. Plotkin, *A Structural Approach to Operational Semantics*, Report DAIMI FN-19, Computer Science Department, Aarhus University (1981).
16. U. Goltz and W. Reisig, CSP programs as nets with individual tokens, *Advances in Petri Nets 1984* (G. Rozenberg, ed.), Lecture Notes in Computer Science **188**:169–196, Springer-Verlag (1985).
17. S. L. Graham and M. Wegman, A fast and usually linear algorithm for global flow analysis, *J. ACM* **23**(1):172–202 (1976).
18. B. Wegbreit, Property extraction in well-founded property sets, *IEEE Trans. on Software Engineering* **1**(3):270–285 (1975).
19. D. E. Knuth, *The Art of Computer Programming, Vol. I: Fundamental Algorithms*, Addison-Wesley, Reading, Massachusetts (1968).
20. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Massachusetts (1986).
21. B. K. Rosen, Data flow analysis for procedural languages, RC-5211, IBM T. J. Watson Research Center, Yorktown Heights, New York (1975).
22. T. C. Spillman, Exposing side effects in a PL/I optimizing compiler, *Proc. IFIP Congress 71*, North Holland, Amsterdam, pp. 376–381 (1971).