(To be published in the Proc. of the 4th European Software Engineering Conference ESEC'93.)

Tractable Flow Analysis for Anomaly Detection in Distributed Programs

S.C. Cheung J. Kramer

Department of Computing,
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ, UK.

Abstract. Each process in a distributed program or design can be modelled as a process flow graph, where nodes represent program statements and directed edges represent control flows. This paper describes a flow analysis method to detect unreachable statements by examining the control flows and communication patterns in a collection of process flow graphs. The method can analyse programs with loops, non-deterministic structures and synchronous communication using an algorithm with a quadratic complexity in terms of program size. The method follows an approach described by Reif and Smolka [9] but delivers a more accurate result in assessing the reachability of statements. The higher accuracy is achieved using three techniques: statement dependency, history sets and statement re-reachability. The method is illustrated by a pump control application for a mining environment. A prototype has been implemented and its performance is presented.

1. Introduction

There is a need for automatic analysis techniques to detect anomalous behaviour in distributed systems. Exhaustive analysis of the behaviour in terms of state space graphs is difficult due to the combinatorial state explostion problem, where the state space of a system can increase geometrically in terms of the system size [12]. To avoid this problem, more tractable but non-exhaustive techniques using flow analysis have been proposed [6, 9, 13].

In our analysis, a distributed application is made up of a system of sequential processes. Processes in the system *communicate synchronously* with each other by transmitting and receiving messages. A process sending a message needs to wait for the process receiving the message; and vice versa. We assume a pure message passing system where synchronous communications are the only way through which processes can interact with one another.

Flow analysis is a method to detect two classes of errors in distributed applications: *synchronisation errors* and *data-usage errors* [5]. Synchronisation errors, such as deadlock and starvation, are errors usually caused by communication anomalies. Data-usage errors are errors caused by misuse in data variables. They include errors of reading uninitialised variables or simultaneously updating a shared variable by parallel processes. In this paper, we are only concerned with the synchronisation errors.

Process *P*	Process *Q*	Process *R*
x, y: variables	x, y: variables	x, y: variables
p1 loop	*q1* loop	*r1* y = 0
p2 read x from file1	*q2* read x from file2	*r2* loop
p3 send x to channel a	*q3* send x to channel b	*r3* receive x from channel a
p4 receive y from channel d	*q4* receive y from channel e	*r4* y = func(x, y)
p5 if y > 0 then	*q5* send x - y to channel d	*r5* if y > 0 then
p6 send x*y to channel c	*q6* end loop	*r6* receive y to channel b
p7 end loop		*r7* else
		r8 send x to channel e
		r9 receive y from channel c
		r10 end loop

Fig. 1. A System *S* of Three Processes *P*, *Q* and *R*

For instance, consider a system *S* with three sequential processes *P*, *Q* and *R* whose behaviours are shown in Fig. 1. A flow analysis of system *S* using the method described in this paper, reveals that statements *p4*, *p5*, *p6* and *p7* of process *P*, statements *q4*, *q5* and *q6* of process *Q*, and statements *r8*, *r9* and *r10* of process *R* are never executed.

2. Related Work

Flow analysis of distributed systems has been studied by several researchers.

Kanellakis and Smolka proposed an algorithm to detect non-progress properties [3]. However, the algorithm is only polynomial under various assumptions, such as the interconnection of processes is a tree and that the processes are free of loops.

Peng and Purushothaman proposed a flow analysis technique for networks of two communicating finite state machines to detect non-progress properties [7]. The approach involves the construction of the network's product machine restricted to a network with two communicating processes. The work is extended in [8] to analyse a network of multiple processes but the complexity of the algorithm may grow exponentially with the number of processes.

Yang and Chung proposed a method to check the feasibility of a given linear execution path, based on several synchronisation rules [13]. However, the number of execution paths to be examined, increases exponentially with the number of non-deterministic structures.

Reif and Smolka refined their work [10] in [9]. In their paper [9], a linear algorithm is presented to detect unreachable statements and construct an event spanning graph for asynchronous communication systems. The algorithm does not give a sufficiently accurate result in the communication models where messages are removed from the channels after reception. Some statements identified as reachable by the algorithm may, in fact, be unreachable in the actual systems. A higher accuracy can be achieved with the method described in this paper with a small increase in computational cost.

Mercouroff proposed an approximate method to identify matching communication pairs based on their number of occurrences [6]. Lattices of arithmetic congruences are required to approximate the sets of event counters. However, there is a lack of detail in selecting the factor adjusting the sensitivity of analysis, constructing the lattices of arithmetic congruences, and analysis of the complexity.

Avrunin et. al. adopted an approach that approximates a concurrent system by a set of event inequalities [1]. The algorithm may produce "spurious" solutions that do not correspond to the behaviour of the concurrent system. Depending on individual cases, the computation of inequalities can suffer from exponential complexity. In addition, the approach provides little information about the anomaly when a set of inequalities is found to be inconsistent.

Generally, each method has its pros and cons. The compromises are usually made along three lines: how general is the method, how accurate is the result, and how tractable is the method. It is unlikely that there is a single method which can achieve all three goals simultaneously. In this paper, we describe a tractable method that is general enough for practical problems, and useful enough for a preliminary understanding of processes' behaviour. An approach similar to [9] is taken because of its generality and tractability. Although our method does not reveal all possible synchronisation errors in a system, it manages to identify a number of errors which cannot be easily spotted manually. It is a technique that one would employ to acquire an initial analysis of a design before submitting it to more sophisticated but computationally expensive analysis methods, such as state space graph construction or proposition provers.

Our method is general enough to analyse systems with arbitrarily loops and nested non-deterministic structures. The method adopts an approach similar to [9], but delivers a higher accuracy with a complexity quadratic in terms of program statements. The higher accuracy is achieved by cutting down spurious execution sequences using three techniques: statement dependency, history sets and statement re-reachability.

These techniques will be described respectively in sections 6, 7 and 8 along with three polynomial algorithms: A, B and C. Algorithm A, described in section 5, is an adaptation of Reif's algorithm in [9] for synchronous communication systems. The algorithm is used to compute the dependency among statements in section 6. Algorithm B is an algorithm achieving a more accurate result than A with the first two techniques: statement dependency and history sets. Algorithm C improves B with the third technique: statement re-reachability.

These algorithms are illustrated by a pump controller example in section 9. A prototype based on algorithm C has been implemented and the performance is given in section 10.

3. The Flow Graph Model

The flow graph model presented below is adapted from [9] for synchronous communication. A formal description of a similar flow graph model can be found in [9].

The behaviour of a process can be described by a *process flow graph (PFG)*, where nodes represent statements and directed edges represent control flows. For example, process *P* in Fig. 1 can be described by a PFG in Fig. 2a. Flow graphs (sometimes known as flow charts) are commonly used in the design and analysis of sequential systems. They are usually extended to accommodate transmit and receive statements for distributed systems.

3.1. Descriptions

Nodes in a PFG can be classified into three types: internal, transmitting and receiving. Internal nodes are executed locally without a need for synchronisation while both transmitting and receiving nodes have to be executed synchronously with their complements in other processes. A node *n'* is said to be a *complement* of *n* if (i) they both share the same channel; and (ii) one of them is a transmitting node and the other is a receiving node. For example, in Fig. 2, node *a*! is a complement of *a*?, and vice versa.

In a PFG, an internal node is symbolised by a local name (e.g. *p* in Fig. 2a); a transmitting node is symbolised by a *channel name* followed by '!' (e.g. *a*! in Fig. 2a); and a receiving node is symbolised by a channel name followed by '?' (e.g. d? in Fig. 2a). A node is said to be a *child* (*parent*) of another node *n*, if it is a succeeding (preceding) neighbour of *n* in a PFG. For example, node *d*? is a child and *p* is a parent of *a*! in Fig. 2a.

A node in a PFG has to be *visited* before it is executed. Initially, only *start nodes*, marked by two concentric circles in PFGs, are visited. When a node is visited, the type of the node is examined. If the type is internal, the node is executed immediately. Otherwise, the node has to wait for a visit of its complement before it can be executed. When a node has been executed, one of its children in the PFG is chosen arbitrarily for the next visit. The process terminates when no child can be chosen. The process blocks when it is visiting a communicating node that can never be executed synchronously with a complement.

3.2. Definitions

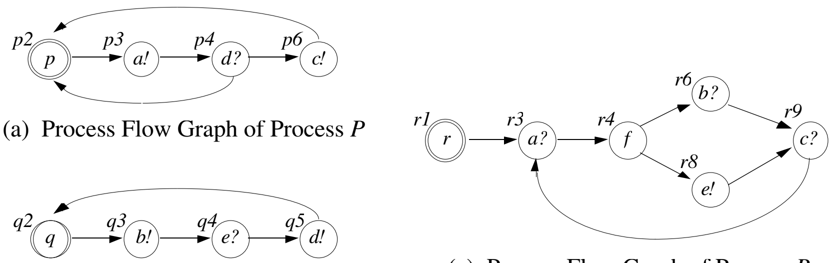
A PFG G_i of a process P_i in a system is a tuple $< N_i, E_i, s_i >$, where N_i denotes the set of nodes, E_i denotes the set of directed edges, and s_i denotes the *start node*. There is a single thread of control in a PFG. An *execution sequence* is a sequence of nodes that have been executed by a process. The execution sequence must be a path in the associated PFG beginning at the *start node*. For example, "*p*.*a*!" is a sequence taken by process *P* to execute nodes *p* and *a*! in Fig. 2.

A node in a PFG is visited by means of an execution sequence. A node can only be visited by the current execution sequence if it is a child of the last element in the sequence. A node is said to be *reachable* by the current execution sequence if it can be visited and executed by the sequence; otherwise it is said to be *unreachable*. For example, in Fig. 2a, node *d*? cannot be visited by an execution sequence "*p*"; therefore it is unreachable by the sequence. However, *d*? can be visited by an sequence "*p*.*a*!". If it can then be executed upon the visit, we say that *d*? is reachable by "*p*.*a*!"; otherwise we say that *d*? is unreachable by "*p*.*a*!". If *d*? is unreachable by any execution sequence, it is said to be unreachable.

Conditional "*if-then-else*" statements from the original program are modelled as non-deterministic choices in the PFG. The data value held by the conditional or guarded variables are discarded in the model. This simplification is common to many synchronisation analysis techniques for distributed programs but has the disadvantage of allowing additional and spurious execution sequences [11]. Symbolic execution techniques can be used if data values are considered [14].

A node identified as unreachable in our method implies the corresponding statement can never be executed in the original program where the usual semantics of conditionals are considered. But a node identified as reachable in our method does not necessarily mean the corresponding statement can be executed in the original program. Hence, the results delivered are *conservative* in *unreachability* but *not conservative* in *reachability*.

3.3. Example



(a) Process Flow Graph of Process *P*

(b) Process Flow Graph of Process *Q*

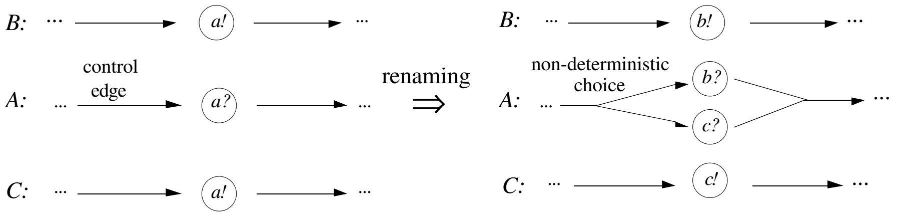
(c) Process Flow Graph of Process *R*

Fig. 2. PFGs of Process P, Q and R

Fig. 2 shows the PFGs of process *P*, *Q* and *R* in Fig. 1. Corresponding statement labels in Fig. 1 are shown next to nodes in the PFGs. Some internal statements in the original programs are omitted to make the graph simpler. The *if* statement *p5* is modelled as a non-deterministic choice of two edges ($d?$,p) and ($d?$,$c!$). Similarly the *if-then-else* statement *r5* is modelled as a non-deterministic choice of two edges (f,$b?$) and (f,$e!$).

4. Renaming of Statements

For simplicity, the flow analysis algorithms presented in the paper assume uniqueness of synchronisation pairs, such that there is a unique complement for each communicating node in a collection of PFGs. For example, there is a unique complement $a!$ in the system, which may synchronise with the node $a?$ in Fig. 2. Under this assumption, a node is therefore unreachable if it can never be executed synchronously with its unique complement. This assumption imposes no restriction to the generality of the algorithms. This can be achieved by a simple renaming mechanism and introduction of non-deterministic structures.



(a) PFGs before Renaming

(b) PFGs after Renaming

Fig. 3. Achieving Unique Synchronisation Pairs by Renaming

For example, a set *Chl* (say $\{b, c\}$) is generated when node $a?$ is identified to have two complements in Fig. 3a. The statement $a?$ in the original graph is then replaced by a non-deterministic structure of branches containing nodes $b?$ and $c?$ in Fig. 3b. The channel names of the complements are exclusively renamed to $b!$ and $c!$.

while there is a node *x* in the system with more than one complement do
    let *n* be the number of complements
    generate a set *Chl* containing *n* unique channel names
    replace the node *x* by a non-deterministic structure of *n* branches, each of them containing a

node, and the channel of each node is exclusively named by a name in *Chl*
rename the channel name of each complement exclusively by a name in *Chl*
end while

5. Adaptation of Reif's Algorithm

A linear algorithm to detect unreachable nodes for asynchronous systems is given in [9]. The algorithm is adapted slightly in this section for synchronous systems. In the adapted algorithm, *RS* denotes a set of reachable nodes. A set of unreachable nodes *US* is given by subtracting *RS* from the set of nodes in the system.

5.1. Algorithm A

The algorithm works according to the following rules:

(i) All start nodes can be visited.
(ii) An internal node is reachable if it can be visited.
(iii) A communicating node is reachable if both the node and its complement can be visited.
(iv) Children of a reachable node can be visited.

5.2. An Implementation of Algorithm A

Let
G (=<N,E>) be an union of all PFGs in the system, where N is the set of nodes and E is the set of edges.
Q denote a queue of unprocessed reachable nodes
RS denote the set of reachable nodes
BS denote the set of visited communicating nodes waiting for the visit of their complements
n' denote the complement of node n

```
procedure initialise( G )
      empty sets Q, RS and BS
      for each n ∈ N do
            if n is a start node then
                  block( n )
end initialise

procedure block( n )
     if n is an internal node then
          add n to RS and Q
      else if n' ∈ BS then
            remove n' from BS
            add both n and n' to RS and Q
      else
            add n to BS
end block
```

```
procedure alga( G )          /*  main procedure  */
      initialise( G )
      while Q ≠ Ø do
            choose and delete some n in Q
            for each child c of node n do
              if ( c ∉ BS ) and ( c ∉ RS ) then
                    block( c )
      end while
      return RS
end alga
```

The *block* procedure checks to see if a node being visited is reachable[1]. The node is assumed reachable if (i) it is an internal node (e.g. node p in Fig. 2a) or (ii) its complement has been waiting for synchronous execution in set *BS*. The set *BS* contains transmitting and receiving nodes which are waiting for their complements. Newly reachable nodes are inserted to the processing queue Q for further processing. When a node in Q is processed, its children not having been visited are examined by the *block* procedure.

The algorithm terminates when the processing queue Q is empty. The complexity of the algorithm is $O(|N|+|E|)$, where N is the set of nodes and E is the set of control edges in a system of PFGs. Upon termination, the algorithm delivers a reachable set *RS*. Nodes in set *RS* are potentially reachable in the sense that they may not be actually reachable in the

---

1 The implementation assumes a closed system where transmitting or receiving nodes are blocked forever if they have no complement. A modification of the if-condition in procedure *block* can change the assumption to an open system.

original program. The unreachable set US of unreachable nodes is given by

$$US = N \setminus RS .$$

Nodes in set US are definitely unreachable in the sense that they must be unreachable in the original program. A correctness proof of Reif's original algorithm and analysis of the complexity can be found in [9].

The problem with this algorithm is the inaccuracy in detecting reachable nodes. For example in Fig. 2, although some nodes in the PFGs are actually unreachable, all of them are identified as reachable by the algorithm. This is because the rules on which the algorithm based, only provide necessary conditions but not sufficient conditions for reachability. This inaccuracy can be improved by our method with the use of the techniques described in sections 6, 7 and 8. Nevertheless, the algorithm is a useful tool to determine the node dependency in section 6 because of its simplicity and tractability.

6. Dependency

Algorithm A does not perform an exhaustive elaboration of execution sequences in a given system. It therefore invariably contains some spurious execution sequences which may not be exhibited by the actual system. These spurious execution sequences account for the inaccurate conclusions on the node reachability. A reduction in these spurious execution sequences results in a higher accuracy. The reduction can be achieved by working out the depend relations among nodes in the same PFG. Execution sequences with order of nodes violating the relations are spurious. Since each execution sequence is local to a PFG, only depend relations among nodes in the same PFG need to be considered.

6.1. Depend Relations

A node x is said to *depend on y*, written as xDy, if x is only reachable by execution sequences containing y. For example in Fig. 2c, node a? cannot be executed unless r has been executed; and node f cannot be executed unless a? has been executed. Suppose x and y are two nodes belonging to the same PFG. Therefore, a? depends on r and f depends on a?.

For convenience, let us denote ξ_i as the set of all possible execution sequences taken by a process P_i, whose behaviour can be modelled as a PFG $\langle N_i, E_i, s_i \rangle$; and let $\|\varepsilon\|$ be the set of nodes that has been traversed by an execution sequence $\varepsilon \in \xi_i$. The definition of depend relation can be given as:

$$\forall\, x, y \in N_i,\ xDy \stackrel{\text{def}}{=\!=} (\ \forall\, \varepsilon \in \xi_i, x \text{ is reachable by } \varepsilon \Rightarrow y \in \|\varepsilon\|\ )$$

For instance, node a? is only reachable by execution sequences containing r; and node f is only reachable by execution sequences containing a?. Therefore, execution sequences described by a node order, which does not have any r before an a?, or does not have any a? before an f, must be spurious. There are two properties of the depend relation:

The depend relation is *transitive* so that
for any three nodes a, b and c, $aDb \wedge bDc \Rightarrow aDc$.

A *symmetrical* depend relation between two nodes implies a synchronisation anomaly; i.e.
for any two nodes a and b, $aDb \wedge bDa \Rightarrow a$ is unreachable and b is unreachable.

6.2. Derivation of Dependency Set

Depend relations can be efficiently determined based on the unreachable sets provided by algorithm A. Suppose x and y are two nodes in a PFG and x is found to be unreachable when y is unreachable. Then x depends on y.

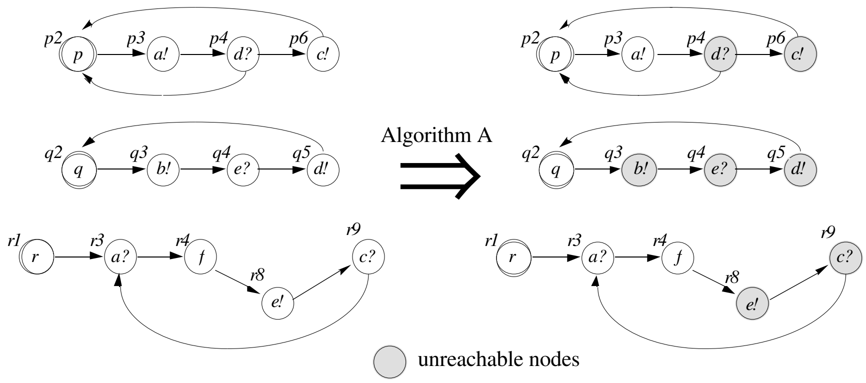


Fig. 4. Deriving Dependent Nodes by Algorithm A
(the bottom process started with statement *r1* is different from that in Fig. 2)

For instance, algorithm A reports an empty unreachable set for a system in Fig. 2. But it reports an unreachable set { *e*!, *c*? } when node *b*? is removed from the PFG of process *R* in Fig. 4. So, both nodes *e*! and *c*? depend on *b*?.

Suppose *n* is a node in a PFG $<N_i, E_i, s_i>$, and US_n (US_n') is the unreachable set given by algorithm A with (without) the presence of node *n* in the PFG. Let us denote a *dependency set* D_n as a set of nodes depending on *n* such that

$$D_n = (US_n' \setminus US_n) \cap N_i$$

Equating *n* to node *b*? in Fig. 2, we get

$$D_{b?} = (\{ d?, c!, b!, e?, d!, e!, c? \} \setminus \{\}) \cap \{ r, a?, f, b?, e!, c? \} = \{ e!, c? \} .$$

The dependency set D_n for a given node *n* can be determined by algorithm A in a complexity $O(|N|+|E|)$, where *N* is the set of nodes and *E* is the set of control edges. To obtain the dependency sets for each node in the system, we need to run the algorithm $|N|$ times. Hence, the dependency between every pairs of nodes in a system can be computed in a complexity $O(|N|(|N|+|E|))$. Section 10.2 shows the time taken by a prototype to work out the dependency for the example systems given in the paper.

6.3. Depend Relations and Spurious Execution sequences

The number of spurious execution sequences can be reduced by the depend relations. Execution sequences violating the depend relations must be spurious. For example, an execution sequence "*r*.*a*?.*f*.*e*!" (in Fig. 2c), which is considered legitimate by algorithm A, can be eliminated because it violates the relation that *e*! depends on *b*?. The execution sequence does not contain an occurrence of *b*? before an occurrence of *e*! in the sequence.

7. History Sets

To detect spurious execution sequences, all possible sequences are examined. However, an explicit elaboration of all possible sequences is computationally too expensive. Some kind of tradeoff is needed to keep the method tractable for non-trival systems. A compromise is to approximate the event histories by a set of nodes, and omit the ordering information (e.g. node *x* occurs before node *y*). For example, an execution sequence "*r*.*a*?.*f*.*e*!" can be approximated by {*r*, *a*?, *f*, *e*!}. So, a single set of nodes can represent a number of execution sequences. The approximation essentially trades off the execution details for tractability.

7.1. Definition of History Sets

For every node n in a PFG $<N_i, E_i, s_i>$, it associates with a single history set, denoted as H_n. H_n is a minimal set with all nodes appearing in those execution sequences that may reach n.

$$H_n = \bigcup_j \{ s \mid s \in \| \varepsilon_j \| \bullet \varepsilon_j \in \xi_i \text{ and } n \text{ is reachable by } \varepsilon_j \}$$

An existence of node x in the history set H_n indicates n is reachable by some execution sequences containing x.

$$\forall\, x \in N_i, x \in H_n \Rightarrow \exists\, \varepsilon \in \xi \bullet (n \text{ is reachable by } \varepsilon \wedge x \in \|\varepsilon\|)$$

Suppose H_f is empty and node f is reachable by an execution sequence "$r.a?$" in Fig. 2c. The execution sequence is recorded by adding nodes r and $a?$ to set H_f and becomes $\{r, a?\}$. Now, from node f's point of view, existence of nodes r and $a?$ in the history set H_f indicates that it is reachable by execution sequences containing some elements in $\{r, a?\}$. The ordering information that r occurs before $a?$ is ignored. This approximation results in a big reduction in storage and computational efforts to keep the method tractable for practical problems.

7.2. Storage and Computation of History Sets

For all history set associated with the nodes in a PFG $<N_i, E_i, s_i>$, each of them contain at most $|N_i|$ elements. If the existence of a node in a history set is represented by a storage bit, the maximum requirements of memory (in bits) to store all history sets for nodes in the PFG $= |N_i|^2$. Therefore, a system of processes $\{ P_1, ..., P_r \}$ would require a maximum of M bits to store all history sets, where M equals

$$\sum_{i=1}^{r} |N_i|^2 \quad .$$

For example, a medium sized system of 64 PFGs each with 64 nodes, would require a maximum 32K bytes of memory to hold all history sets.

In addition, set computations concerning history sets can be efficiently performed by bitwise arithmetic supported in many imperative languages (e.g. C). For instance, merging two sets can be performed by a bitwise-OR operation between two integer variables representing the sets.

7.3. Reduction of Spurious Execution sequences

The number of spurious execution sequences can be reduced by using the techniques of dependency and history sets. Suppose n is a node and D_n^{-1} is the set of nodes on which n depends. The following propositions can be derived using the techniques of dependency and history sets [2].

Proposition 1: Suppose m is a reachable parent of a node n, and m occurs at the end of an execution sequence ε. *Node n is reachable by* ε *only if the set* $H_m \cup \{m\}$ *contains all the nodes on which n depends.*

The necessary condition for reachability in proposition 1 is not enforced by Algorithm A. A check of the necessary condition prevents a node reachable by *dependency violating* execution sequences. As a result, the number of spurious execution sequences is reduced and the accuracy is improved. These conditions, however, do not eliminate all spurious execution sequences because a history set is merely an approximation of the actual execution sequences.

Proposition 2: For any node n in a process flow graph,

$$H_n \subseteq \bigcup_m \{\, s \mid s \in H_m \cup \{m\} \bullet m \textit{ is a reachable parent of } n \wedge D_n^{-1} \subseteq H_m \cup \{m\} \,\}$$

This proposition establishes an upper bound of the history set of a node based on those of its reachable parents. An upper bound of a history set is assumed in algorithm B (see rule (iv) in section 7.4). So, if the upper bound of a history set is empty then the actual history set must also be empty, which means n has not been reachable by any execution sequences.

7.4. Algorithm B

Algorithm B achieves a higher accuracy in reachability by extending the rules in algorithm A based on propositions 1 and 2.

- (i) All start nodes can be visited.
- (ii) An internal node is reachable if it can be visited.
- (iii) A communicating node is reachable if both the node and its complement can be visited.
- (iv) $H_n = \bigcup_m (H_m \cup \{m\})$, where node m is a reachable parent of node n and the set $H_m \cup \{m\}$ contains all the nodes on which n depends.
- (v) A node n can be visited if H_n is non-empty.
- (vi) All history sets are initially empty.

This set of rules can be reduced to that of algorithm A if

- (a) we ignore the dependency information so that nodes do not depend on one another in (iv); and
- (b) we also ignore the history set by replacing the set $\{m\}$ in (iv) with a set $\{\bullet\}$ which contains a token carrying no value.

An application of algorithm B on a system modelled in Fig. 2 reveals that nodes $d?$ and $c!$ in Fig. 2a, nodes $e?$ and $d!$ in Fig. 2b, and nodes $e!$ and $c?$ in Fig. 2c are unreachable(see Fig. 5). These nodes are reported reachable by algorithm A.

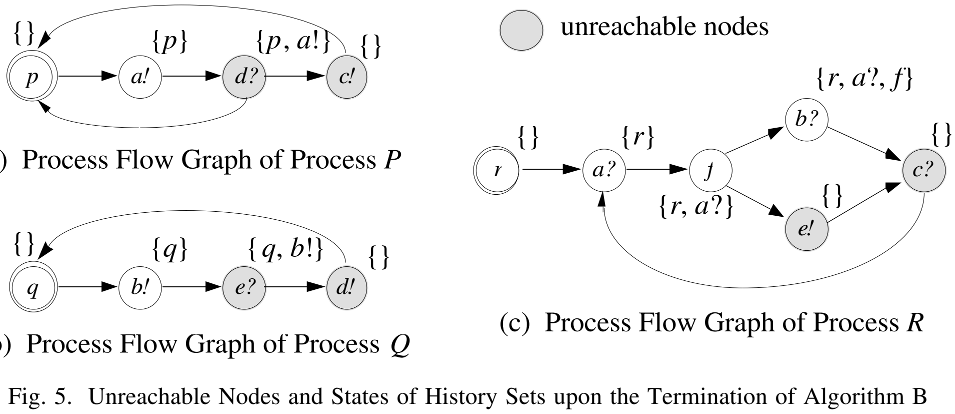


Fig. 5. Unreachable Nodes and States of History Sets upon the Termination of Algorithm B

7.5. An Implementation of Algorithm B

Denote

the complement of a node n as n';
the history set held by a node n as H_n;
the set of nodes depending on n as D_n and the set of nodes on which n depends as D_n^{-1};

Q as a queue holding unprocessed reachable nodes;
V as a set holding nodes which can be visited but not yet reachable;
R as a set holding reachable nodes;

```
procedure initialise( )
      empty Q, V, R and all history sets
      for each node n in the system do
            work out D_n and D_n^-1
      for each start node s in the system do
            visit( s, Ø )
end initialise

procedure execute( n )
      K = H_n ∪ {n}
      for each immediate child c of n do
            if ¬(K ⊆ H_c) and (D_c^-1 ⊆ K) then
                  visit( c, K )
end execute

procedure visit( n, K )
      H_n = H_n ∪ K
      if (n ∉ R) and (n ∉ V) then
            block( n )
      else if (n ∈ R) then
            execute( n )
end visit

procedure block( n )
      if n is an internal node then
            add n to both R and Q
      else if n' ∉ V then
            add n to V
      else
            remove n' fromV
            add n and n' to both R and Q
      end if
end block

procedure algc( )        /*  main procedure    */
      initialise( )
      while Q ≠ Ø do
            remove a node n from Q
            execute( n )
      end while
      report sets R
end algc
```

The *initialise* procedure initialises all history sets to empty (rule (vi)) and asserts that all start nodes can be visited (rule (i)). The *block* procedure examines the reachability of visited nodes according to rule (ii) and (iii). Nodes that have been newly found reachable are inserted to Q, and wait for further processing. The *execute* procedure visits the children of a node according to rule (iv). The procedure *visit* detects if there are any additional nodes that can be visited according to rule (v). The procedure also keeps an upper bound of the history set of a node and propagates the changes to children of the node. The algorithm terminates when no more nodes can become reachable, i.e. when Q is empty.

Suppose a process flow graph of a process P_i is given by a tuple $<N_i, E_i, s_i>$. Upon the termination of the algorithm, the following result is delivered:

$N_i \cap R$	gives the set of reachable nodes in process P_i
$N_i \setminus R$	gives the set of unreachable nodes in process P_i

The results delivered are conservative in unreachability but not conservative in reachability. A discussion of the limitations and comparisons with algorithms A and C will be given in section 9.

7.6. Complexity of Algorithm B

Suppose n is a node in a PFG $<N_i, E_i, s_i>$. Let us denote $\#visit_n$, $\#execute_n$ and $\#block_n$ as the number of times the procedures *visit*, *execute* and *block* are called with the parameter equals n. The *if-clause* in the procedure *execute* asserts

$$\#visit_n \le |N_i| \quad .$$

The first *if-clause* in procedure *visit* enforces that node n is at most blocked for once. This asserts

$$\#block_n \le 1 \quad .$$

The *execute* procedure can be called by the *algc* procedure and the *visit* procedure. Since n is put into the queue Q via the calling of the *block* procedure in the *visit* procedure, From this, it can be asserted

$$\#execute_n \leq \#visit_n \leq |N_i| \quad .$$

Please note that set operations in the implementation can be achieved in constant time by a bitwise arithmetic operation supported in imperative languages. The computational effort for a node n is equal to the sum of the efforts taken by procedure *visit*, *block* and *execute* for a node n together with the efforts to evaluate the sets D_n and D_n^{-1}. Let us denote $depend_n$ as the computational effort to compute D_n and D_n^{-1}. The computational effort for a node n is given by

$$\lambda|N_i| + depend_n \qquad \text{(where } \lambda \text{ is a small positive integer constant).}$$

Therefore the computational effort of algorithm B is given by

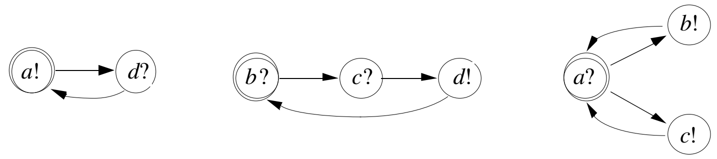
$$\sum_{P_i} \lambda|N_i|^2 + \sum_n depend_n$$

$$\leq \quad O(|N|^2) + O(|N|(|N|+|E|)) \quad \text{(where } N \text{ and } E \text{ stand for set of nodes and set of control flows in the system respectively)}$$

Hence, the complexity of algorithm B is $O(|N|(|N|+|E|))$ which is dominated by the efforts to compute the dependency set for each node in the system.

8. Re-reachability

The accuracy of algorithm B can be further improved using a concept of re-reachability by checking whether nodes can be executed more than once. Consider a system S' consisting of three processes: U, V and W modelled as follows:



(a) PFG of process U (b) PFG of process V (c) PFG of process W

Fig. 6. PFGs of Processes U, V and W

In system S', nodes $c!$, $c?$, $d!$ and $d?$ are unreachable. This anomaly cannot be detected by an algorithm based on the rules in section 5.1 or 7.4. The accuracy of the algorithm can be improved with the knowledge that $a?$ can never be executed more than once. Using this additional information, the anomaly in system S' can be detected.

8.1. Definition of Re-reachability

A node n is *re-reachable* by an execution sequence ε if and only if it is reachable by ε and $\|\varepsilon\|$ contains n. A node n is *first reachable* by an execution sequence ε if and only if it is reachable by ε and $\|\varepsilon\|$ does not contain n.

$\forall\, n \in N_i, \varepsilon \in \xi_i,$

$\quad n$ is re-reachable by $\varepsilon \Leftrightarrow n$ is reachable by $\varepsilon \wedge n \in \|\varepsilon\|$

and

$\quad n$ is first reachable by $\varepsilon \Leftrightarrow n$ is reachable by $\varepsilon \wedge n \notin \|\varepsilon\|$

A reachable node is re-reachable if it is re-reachable by some ε; otherwise it is first reachable. A node has been *executed once* by an execution sequence ε if it appears once in ε. A node has been *executed more than once* by an execution sequence ε if it appears more than once in ε.

8.2. Further Reduction of Spurious Execution sequences

Spurious execution sequences can be further reduced using the technique of re-reachability. Assuming the notations in section 7.3, the following two propositions can be derived [2].

Proposition 3: Suppose m is a reachable parent of a node n, and m ONLY occurs at the end of an execution sequence ε. *Node n is reachable by* ε *only if the set* $(H_m \setminus D_m) \cup \{m\}$ *contains all the nodes on which n depends.*

Proposition 3 enforces a stricter necessary condition for reachability than that in proposition 1. Spurious execution sequences can be further eliminated based on a stricter necessary condition for those nodes whose parents only occur at the end of the sequences. This additional information is used in algorithm C to improve the accuracy of algorithm B.

Proposition 4: For any node n in a process flow graph,

$$H_n \subseteq \bigcup_m \{ s \mid s \in (H_m \setminus D_m) \cup \{m\} \bullet m \text{ is a first reachable parent of } n \wedge D_n^{-1} \subseteq (H_m \setminus D_m) \cup \{m\} \} \cup$$
$$\bigcup_m \{ s \mid s \in H_m \cup \{m\} \bullet m \text{ is a re-reachable parent of } n \wedge D_n^{-1} \subseteq H_m \cup \{m\} \}.$$

Like proposition 2, proposition 4 allows us to establish an upper bound of a node's history set from its reachable parents. It enforces a tighter upper bound than that in proposition 2 with the additional information of re-reachability. An upper bound of a history set is assumed in algorithm C (see rule (iv) in section 8.3). Therefore, if the upper bound of a history set is empty then the actual history set must also be empty, which means n has never been visited by any execution sequences. By enforcing a tighter upper bound, it can disclose some empty history set which cannot be disclosed by proposition 2.

8.3. Algorithm C

The accuracy of algorithm B can be further improved using the following rules. The rules only allow a node to be visited by execution sequences which satisfy proposition 3.

(i) All start nodes can be visited.
(ii) An internal node is reachable (re-reachable) if it can be visited (re-visited).
(iii) A communicating node is reachable (re-reachable) if both the node and its complement can be visited (re-visited).
(iv) $H_n = \bigcup_m (K_m \cup \{m\})$, where node m is either
   (a) a first reachable parent of node n and the set $(H_m \setminus D_m) \cup \{m\}$ contains all the nodes on which n depends. In this case, $K_m = H_m \setminus D_m$; or
   (b) a re-reachable parent of node n and the set $H_m \cup \{m\}$ contains all the nodes on which n depends. In this case, $K_m = H_m$.
(v) A node n can be visited if H_n is non-empty.
(vi) A node n can be re-visited if $n \in H_n$.
(vii) All history sets are initially empty.

If the re-reachability information is ignored by setting D_m in the rules (iv) to Ø, Proposition 3 is reduced to Proposition 1, Proposition 4 is reduced to Proposition 2, and the set of rules in algorithm C is reduced to that of algorithm B.

An application of algorithm C on a system modelled in Fig. 6 reveals that node $d?$ in Fig. 6a, nodes $c?$ and $d!$ in Fig. 6b, and node $c!$ in Fig. 6c are unreachable (see Fig. 7). It further identifies that node $a!$ in Fig. 6a, node $b?$ in Fig. 6b, and nodes $a?$ and $b!$ in Fig. 6c are reachable but not re-reachable.

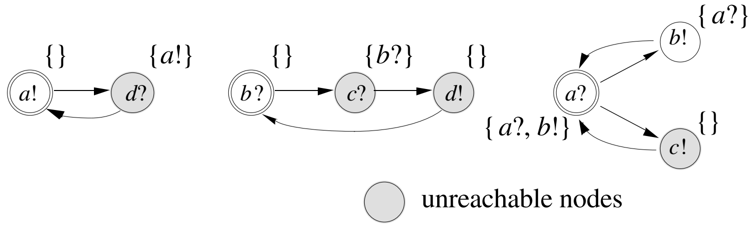


Fig. 7. Unreachable Nodes and States of History Sets upon the Termination of Algorithm C

Since node $a?$ is not re-reachable, the element $b!$ in the history set $H_{a?}$ is not passed onward to node $c!$ which depends on node $b!$.

8.4. An Implementation of Algorithm C

Denote
- the complement of a node n as n';
- the history set held by a node n as H_n;
- the set of nodes depending on n as D_n;
- the set of nodes on which n depends as D_n^{-1};
- Q as a queue holding unprocessed reachable and re-reachable nodes;
- V as a set holding nodes which can be visited but not yet reachable;
- VV as a set holding nodes which can be re-visited but not yet re-reachable;
- R as a set holding reachable nodes;
- RR as a set holding re-reachable nodes.

```
procedure initialise( )
    empty Q, V, VV, R, RR and all history sets
    for each node n in the system do
        work out D_n and D_n^-1
    for each start node s in the system do
        visit( s, Ø )
end initialise

procedure execute( n )
    if n ∈ RR then
        K = H_n ∪ {n}
    else
        K = ( H_n \ D_n ) ∪ {n}
    for each immediate child c of n do
        if ¬(K ⊆ H_c) and (D_c^-1 ⊆ K) then
            visit( c, K )
end execute

procedure algc( )        /*   main procedure   */
    initialise( )
    while Q ≠ Ø do
        remove a node n from Q
        execute( n )
    end while
    report sets R and RR
end algc

procedure block( n, VS, RS )
    if n is an internal node then
        add n to both RS and Q
    else if n' ∉ VS then
        add n to VS
    else
        remove n' fromVS
        add n and n' to both RS and Q
    end if
end block

procedure visit( n , K )
    H_n = H_n ∪ K
    if (n ∈ H_n) and (n ∉ RR) and (n ∉ VV) then
        block( n, VV, RR )
    else if (n ∉ R) and (n ∉ V) then
        block( n, V, R )
    else if (n ∈ R) then
        execute( n )
end visit
```

The *initialise* procedure initialises all history sets to empty (rule (vii)) and asserts that all start nodes can be visited (rule (i)). The *block* procedure examines the reachability of visited nodes according to rule (ii) and (iii). Nodes that have been newly asserted reachable or re-reachable are inserted to Q, waiting to be executed. The *execute* procedure visits the children of a node according to rule (iv). The procedure *visit* detects if there are any additional nodes that can be visited or re-visited according to rule (v) and (vi). The procedure also keeps an upper bound of the history set of a node and propagates the changes to children of the node. The algorithm terminates when there no more nodes can become reachable or re-reachable; i.e. when Q is empty.

Suppose the PFG of a process P_i is given by a tuple $<N_i, E_i, s_i>$. Upon the termination of the algorithm, the following information is delivered:

$N_i \cap R$	gives the set of reachable nodes in process P_i
$N_i \setminus R$	gives the set of unreachable nodes in process P_i
$N_i \cap RR$	gives the set of re-reachable nodes in process P_i
$N_i \setminus RR$	gives the set of not re-reachable nodes in process P_i

8.5. Complexity

Algorithm C gives a more accurate result than that of algorithm B with the same complexity in $O(|N|(|N|+|E|))$ where N is the set of nodes and E is the set of edges in a system modelled as a collection of process flow graphs. The complexity of algorithm C can be analysed using similar steps of algorithm B as given in section 7.6. Like algorithm B, the computational effort of the algorithm C is dominated by the effort to evaluate the dependency sets for each node in the system. This domination is confirmed by the statistics of the prototypes given in section 10.2.

9. A Pump Control System - An Example

9.1. Introduction of the Pump System

As an example for flow analysis method, we present a simplified pump control application for mining environment [4].

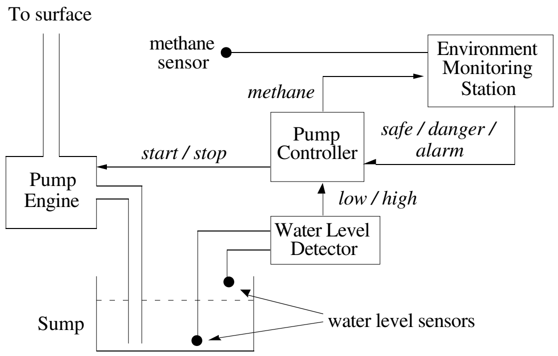


Fig. 8. Control of a Main Pump for Mine Drainage

Fig. 8 shows the schematic of a simplified pump installation. It is used to pump mine-water collected in a sump at shaft bottom to the surface. The pump runs automatically, controlled by water level as sensed by the high- and low-level detectors. Detection of high level causes the pump to run until low level is indicated. The pump is situated underground in a coal-mine, and so for safety reasons it must not be started or continue running when the level of methane in the atmosphere exceeds a preset safety limit. The pump controller gets information on methane level by communicating with a nearby environment monitoring station.

9.2. Flow Graphs of the Pump System

The system is composed of four sequential processes, namely pump controller, environment monitoring station, water level detector and pump engine.

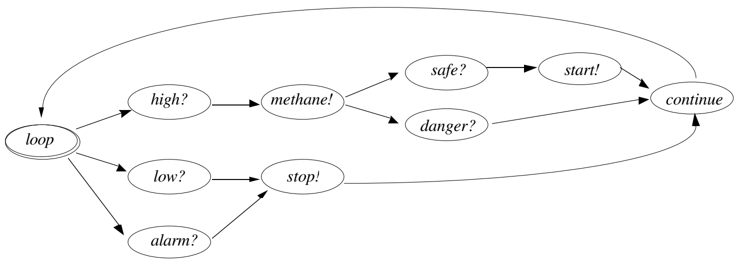


Fig. 9(a). Pump Controller

Fig. 9(a) describes the behaviour of a pump controller. When the pump controller is informed a high water level, it checks the methane level with the environment monitoring station. If the methane remains at a safe level, it starts the pump engine; otherwise it does nothing. If it is informed of a low water level or is warned of the alarming methane level, it stops the pump engine immediately.

The environment monitoring station in Fig. 9(b) upon request to check the methane level, it replies whether the methane level is safe or danger. The station also periodically checks the methane level and gives warning if the methane level is at an alarming level.

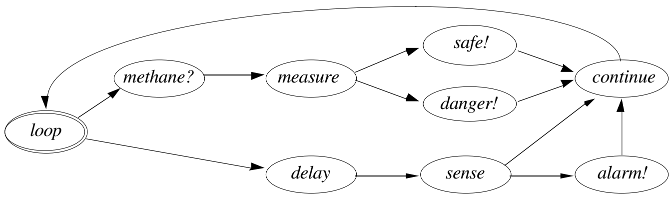


Fig. 9(b). Environment Monitoring Station

The water level detector in Fig. 9(c) notifies the pump controller if the water level is low or high. The pump engine in Fig. 9(d) accepts the command to start or stop the engine.

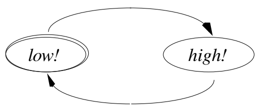


Fig. 9(c). Water Level Detector

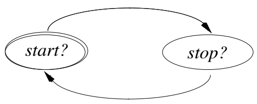


Fig. 9(d). Pump Engine

9.3. Flow Analysis with Algorithm A

An application of the algorithm A fails to detect any unreachable nodes in the system. The unreachable set *US* given by the algorithm is empty.

9.4. Flow Analysis with Algorithm B and C

In this example, algorithm B gives the same results as algorithm C. The information of re-reachability, in this example, gives no further information to isolate more unreachable nodes. The dependency analysis derives a set of depend relations which includes (i) the *high*? node depends on *low*? node and (ii) *stop*! node depends on *high*? node in the PFG of Fig. 9(a). To differentiate internal nodes with the same name, let us denote $node_{PC}$ and $node_{EMS}$ for nodes belonging to pump controller and environment monitoring station respectively for convenience. An application of the algorithm B or C on the example gives

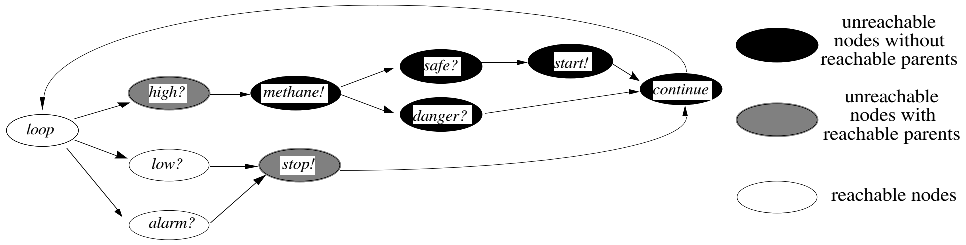


Fig. 10 PFG of the Pump Controller showing the Results of the Algorithm B & C

A set of unreachable nodes = { *high*?, *high*!, *methane*?, *methane*!, *safe*?, *safe*!, *danger*?, *danger*!, *start*?, *start*!, *stop*?, *stop*!, $continue_{PC}$, *measure* }

A set of reachable nodes = { *low*?, *low*!, *alarm*?, *alarm*!, $loop_{PC}$, $loop_{EMS}$, *delay*, *sense*, $continue_{EMS}$ }

A set of not re-reachable nodes = { *high*?, *high*!, *methane*?, *methane*!, *safe*?, *safe*!, *danger*?, *danger*!, *start*?, *start*!, *stop*?, *stop*!, $continue_{PC}$, *measure*, *low*?, *low*!, *alarm*?, *alarm*!, $loop_{PC}$ }

A set of re-reachable nodes = { $loop_{EMS}$, *delay*, *sense*, $continue_{EMS}$ }

Nodes in the set of unreachable nodes can never be executed by the actual system. Nodes in the set of reachable nodes may be executed by the actual system. Nodes in the set of not re-reachable nodes can never be executed more than once by the actual system. Nodes in the set of re-reachable nodes may be executed more than once by the actual system.

A designer can focus on those unreachable nodes which have reachable parents in the PFGs. These nodes identify the potential places at which the processes are likely to be block. Examples of these nodes in the PFG of the pump controller are slightly shaded in Fig. 10.

The results are confirmed by a prototype implementation of the algorithms on SUN IPX workstations. The performance of the prototype is given in section 10.1.

10. Analysis

10.1. Limitations of Algorithms A, B and C

Let RS_a, RS_b, RS_c, RS_r be the set of reachable nodes given by algorithm A, algorithm B, algorithm C and the real system respectively. Similarly, let US_a, US_b, US_c, US_r be the set of unreachable nodes given by algorithm A, algorithm B, algorithm C and the real system respectively. The sets can be related by the following equations:

$$RS_a \supseteq RS_b \supseteq RS_c \supseteq RS_r$$
$$US_a \subseteq US_b \subseteq US_c \subseteq US_r$$
$$RS_a \cup US_a = RS_b \cup US_b = RS_c \cup US_c = RS_r \cup US_r = \text{the set of nodes in the system}$$

The results in unreachability delivered by algorithm A, B, and C are conservative. A node identified as unreachable by our method implies the corresponding statement can never be executed in the original program where the usual semantics of conditionals are considered. The results in unreachability delivered by the algorithms are not conservative. A node identified as reachable by our method does not necessarily mean the corresponding statement can be executed in the original program. Algorithm B achieves a more accurate result than algorithm A by reducing spurious execution sequences using Proposition 1. With the use of Proposition 3, algorithm C cuts down the spurious execution sequences further based on the additional information on re-reachability.

Algorithm C also detects the set of nodes which can never be executed more than once by the system. Let RR_c and RR_r be the set of re-reachable nodes given by algorithm C and the real system respectively; and UR_c and UR_r be the set of not re-reachable nodes given by algorithm C and the real system respectively. They can be related by:

$RR_c \supseteq RR_r$
$UR_c \subseteq UR_r$
$RR_c \cup UR_c = RR_r \cup UR_r$ = the set of nodes in the system

10.2. Performance of the Prototypes

Algorithm A and algorithm C have been implemented on a SUN IPX workstation according to section 5.2 and 8.4 respectively. The following table gives the performance of the prototype for the examples given in this paper.

Examples	#nodes	Computational Time in milliseconds			Ratio = T_c / T_a
		Dependency Evaluation	Algorithm A T_a	Algorithm C T_c	
Fig. 2	14	2.6	0.23	2.8	12.2
Fig. 6	8	0.8	0.13	0.8	6.15
Fig. 9	23	8.5	0.42	8.6	20.5

Table 11 Prototype Performance

The columns give, respectively, the experiments performed, the number of nodes in each example, the time taken for algorithm A, the time taken for algorithm C, the time taken to evaluate the dependency based on algorithm A and the ratio between the time taken by algorithm C and algorithm A. All the experiments were run on a SUN IPX workstation with 16MB of memory; times given are in CPU milliseconds on that machine and include both user and system time. The CPU time is given by looping each experiment 10000 times and then divide the time by 10000.

The results show the time taken to derive the dependency among nodes in the system occupies approximately 99% of the total time taken by algorithm C. This confirms the complexity analysis that the complexity of algorithm C is dominated by the complexity of dependency derivation. The ratio between the time taken by algorithm C and that taken by algorithm A is roughly equal to the number of nodes in the example. This agrees with the difference in $O(|N|)$ between complexity of algorithm C and that of algorithm A, where N is the set of nodes in the PFGs modelling the program. The experiments confirm the computational tractability of algorithm C. It takes 8 seconds for algorithm C to analyse a dining philosopher system of 64 diners.

11. Conclusions and Future Work

In this paper, we have presented three flow analysis algorithms to detect synchronisation anomalies in a distributed program. All the three algorithms support distributed programs with arbitrarily number of loops and non-deterministic structures. Algorithm A is an adaptation of the algorithm proposed by Reif [9]. The accuracy of algorithm A is improved in algorithm B with the techniques of dependency and history sets. Algorithm C further improves the accuracy of algorithm B by considering re-reachability . The complexity of algorithm C is quadratic in terms of the statements in a program and its tractability is confirmed by the performance of a prototype given in section 10.2. Given the tractability of algorithm C, it can be readily applied to designs of practical applications for preliminary checks which help to increase the designers' confidence in the designs.

We intend to further improve the prototypes and perform more analysis using case studies. We are also extending the technique to handle dynamic communication, where communication structure is dynamic and channels may be passed as arguments. Data flow analysis techniques can be employed to derive all the potential communication structures of such systems.

Acknowledgement

The authors would like to acknowledge discussions with our colleagues in the Parallel and Distributed Systems Group during the formulation of these ideas. We gratefully acknowledge the Swiss Bank and the DTI (Grant Ref: IED 410/36/2) for their financial support. We would also like to express our thanks to the anonymous reviewers for their invaluable suggestions to improve the quality of the paper.

References

1. G.S. Avrunin, U.A. Buy, J.C. Corbett, et al.: Automated Analysis of Concurrent Systems with the Constrained Expression Toolset. *IEEE TOSE 17*, 11, 1204-1222.
2. S.C. Cheung and J. Kramer: Tractable Flow Analysis for Anomaly Detection in Distributed Programs, Technical Report, DOC 1993/3, March 1993.
3. P.C. Kanellakis and S.A. Smolka: On the Analysis of Cooperation and Antagonism in Networks of Communicating Processes. In Proc. *4th ACM Symposium on Principles of Distributed Computing*, Ontario, Canada, August 1985, pp. 23-38.
4. J. Kramer, J. Magee, M. Sloman, et al.: CONIC: an Integrated Approach to Distributed Computer Control Systems. *IEE Proceedings, Part E 130*, 1 (January 1983), 1-10.
5. C.E. McDowell and D.P. Helmbold: Debugging Concurrent Programs. *ACM Computing Surveys 21*, 4, 593-623.
6. N. Mercouroff: An Algorithm for Analyzing Communicting Processes. In Proc. *Mathematical Foundation of Programming Semantics '91*, Pittsburgh, PA, March 1991, published in LNCS 598.
7. W. Peng and S. Purushothaman: Towards data flow analysis of communicating finite state machines. In Proc. *8th ACM Symposium on Principles of Distributed Computing*, August 1989.
8. W. Peng and S. Purushothaman: Data Flow Analysis of Communicating Finite State Machines. *ACM Transactions on Programming Languages and Systems 13*, 3, 399-432.
9. J. Reif and S. Smolka: Data Flow Analysis of Distributed Communicating Processes. *Internation Journal of Parallel Programming 19*, 1 (1990), 1-30.
10. J.H. Reif: Dataflow Analysis of Communicating Processes. In Proc. *6th ACM Symposium on Principles of Programming Languages*, June 1979, pp. 257-268.
11. K.C. Tai and C.Y. Din: Validation of Concurrency in Software Specification and Design. In Proc. *3rd International Workshop on Software Specification and Design*, London, UK, August 1985, pp. 223-227.
12. R.N. Taylor: Complexity of Analyzing the Synchonization Structure of Concurrent Programs. *Acta Informatica 19*, 57-84.
13. R.D. Yang and C.G. Chung: The Analysis of Infeasible Concurrent Paths of Concurrent Ada Programs. In Proc. *14th Annual International Computer Software and Applications Conference (COMPSAC 90)*, Chicago, Illinois, October 1990, pp. 424-429.
14. M. Young and R.N. Taylor: Combining Static Concurrency Analysis with Symbolic Execution. *IEEE Transactions on Software Engineering 14*, 10, 1499-1511.