View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Crossref

Scientific Programming 14 (2006) 151–170
IOS Press

Regular Paper

Communication analysis of distributed programs

Sharon Simmons[a,*], Dennis Edwards[a] and Phil Kearns[b]

[a]*Department of Computer Science, University of West Florida, 11000 University Parkway, Pensacola, FL 32503, USA*

[b]*Department of Computer Science, The College of William & Mary, Post Office Box 8795, Williamsburg, VA 23185, USA*

Abstract. Capturing and examining the causal and concurrent relationships of a distributed system is essential to a wide range of distributed systems applications. Many approaches to gathering this information rely on trace files of executions. The information obtained through tracing is limited to those executions observed.

We present a methodology that analyzes the source code of the distributed system. Our analysis considers each process's source code and produces a single comprehensive graph of the system's possible behaviors. The graph, termed the partial order graph (*POG*), uniquely represents each possible partial order of the system. Causal and concurrent relationships can be extracted relative either to a particular partial order, which is synonymous to a single execution, or to a collection of partial orders. The graph provides a means of reasoning about the system in terms of relationships that will definitely occur, may possible occur, and will never occur.

Distributed assert statements provide a means to monitor distributed system executions. By constructing the *POG* prior to system execution, the causality information provided by the *POG* enables run-time evaluation of the assert statement without relying on traces or addition messages.

1. Introduction

Core to a wide range of distributed system challenges is examining the causal and concurrent relationships among events of an execution. Some examples of these challenges are deadlock detection [2], debugging [6,11,12], rollback and recovery [5,9], termination detection [8,20], mutual exclusion violation [1] and global predicate evaluation [3,7, 13,18]. Techniques designed to meet these challenges is an ongoing area of research. Most current techniques make use of trace files generated during execution. Trace files of an execution provide the information needed to deduce the causal and concurrent relationships for that particular execution but are limited to the execution from which they were gathered. Other executions of the systems can define different relationships not present in the trace.

We present a methodology that provides a comprehensive examination of a distributed system. Each possible partial order is discovered and combined into a single, representative graph. This partial order graph, or *POG*, provides a complete view of the system without being tethered to any particular execution. From the *POG*, causal

*Corresponding author: Sharon Simmons, Tel.: +1 850/473 7349; Fax: +1 850/857 6056; E-mail: ssimmons@cs.uwf.edu.

ISSN 1058-9244/06/$17.00 © 2006 – IOS Press and the authors. All rights reserved

and concurrent relationships can be extracted. The extracted relationships can convey a view of the system ranging from a single execution to all possible executions. We have developed a prototype tool based on static analysis of a system's source code to produce the *POG* for asynchronous communication systems [14,15].

Taylor [19] has developed an algorithm for statically analyzing the synchronous communication of a distributed program. Synchronous communication occurs when the transmitting process blocks until the message is received by the destination process. Our research into the static analysis of source code was partially motivated by Taylor's results, but differs in both objectives and outcomes.

Taylor's work determines all possible total orders of execution where a single partial order may be embedded in many total orders. In enumerating the total orders, the concurrency defined by a partial order is not preserved. Our work uniquely identifies each partial order of the system. From each identified partial order, we preserve the inherent concurrency.

In Section 2, our system model is presented. Sections 3, 4 and 5 present each step in generating the *POG* from source code. Although we intitially present the method in a simplified environment, Section 6 explains how looping constructs are incorporated. Event relationships are extracted from the *POG* in Section 7. In Section 8 an example of the utility of the technique is provided. Conclusions are found in Section 9.

2. Model

A distributed system consists of a fixed number of processes $\Pi = \{P_0, \ldots, P_{N-1}\}$. The execution of a distributed program is viewed as a set of events $E = E_0 \cup \ldots \cup E_{N-1}$ where E_i represents the events of P_i, and an irreflexive *partial* order is defined on these events [10]: $\rightarrow \subseteq E \times E$. The $\rightarrow$ relation is commonly referred to as *happens before* or *causally precedes*. For $e, f \in E$, $e \rightarrow f$ if and only if e has potential causal impact upon f.

Interprocess communication defines the happens before relationship among events on different processes. Asynchronous communication [17] occurs when a process places a message "on the network," and continues execution. This type of communication, for example, is offered by the transport level datagram service UDP. The destination process blocks until it receives the message, then continues execution. We assume FIFO point-to-point communication.

In an asynchronous communication regime, the happens before relationship is the smallest relation satisfying the following three conditions: (1) if e and f are events in the same process, and e precedes f, then $e \rightarrow f$; (2) if e is the transmission of a message and f is the receipt of the same message, then $e \rightarrow f$; and (3) if $e \rightarrow f$ and $f \rightarrow g$, then $e \rightarrow g$.

If $e \rightarrow f$, e causally precedes f and f causally succeeds e. If $e \not\rightarrow f$ and $f \not\rightarrow e$, then e and f are causally unrelated or concurrent, denoted $e \| f$, and neither can causally affect the other.

One of possibly many partial orders is defined when a distributed system executes. The potential for many partial orders exists from branches in control of execution, from communication delays, and from unpredictable process execution speeds. Consider the source code of a three process distributed system shown in Fig. 1. The function *async_send*(i, msg) transmits msg to P_i, and *async_recv*(j, msg) receives msg from P_j.

One of two possible partial orders is defined when this program executes. The time-space diagrams representing the two possible partial orders, PO_1 and PO_2, are shown in Fig. 2. Set $\mathcal{P}$ is the set of possible partial orders of a distributed system's execution. For the example system, $\mathcal{P} = \{PO_1, PO_2\}$. For any given execution, one partial order, $\rho \in \mathcal{P}$, is produced.

The first step in generating the *POG* is to examine the source code and create a flow graph for each process. Each flow graph enumerates the possible execution paths for that process. Communication analysis of the flow graphs is used to construct an intermediate graph that represents the causal and concurrent relationships possible in all executions of the system. An optimized graph, the *POG*, is derived from the intermediate graph.

A compiler for ANSI start C has been implemented to create a *POG* from source code. First, the steps to generate a *POG* are presented for source code without looping constructs using the small example pseudo code shown in Fig. 1 as an example. Looping constructs are then incorporated into the *POG* generation methodologies and two more complex program examples are given.

```
P0 ::
begin
    x = 1
    async_send(1, x)
    x = 2
    async_send(1, x)
end

P1 ::
begin
    if
        async_recv(0, x)
        async_recv(2, y)
    else
        async_recv(2, y)
        async_recv(0, x)
    endif
    async_recv(0, x)
end

P2 ::
begin
    y = 3
    z = 4
    async_send(1, y)
end
```

Fig. 1. A simple 3 process system.

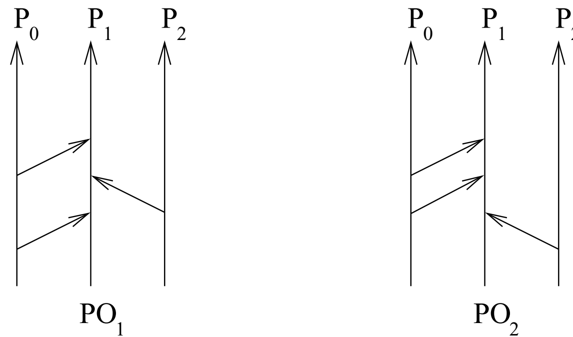


Fig. 2. Two partial orders.

3. Control flow graphs

The possible flows of control through each process, P_i, in the system are represented by a control flow graph FG_i. The graph $FG_i = \{V_i, A_i, \alpha_i\}$ where V_i is the set of nodes, A_i is the set of arcs, and $\alpha_i \in V_i$ is the root node of the graph. Nodes represent *computation*, *communication*, or *control* constructs in the source code. The root node, α_i, is the start of execution of process P_i. Arcs represent the flow of execution of a process created from the source code. If an arc exists from node v to node v', then v' can be executed immediately following the execution of v. Multiple paths may exist through FG_i from α_i, but all paths will terminate in a single leaf node, ω_i if the program terminates.

Returning to the example source code of Fig. 1, we see that processes P_0 and P_2 have a single execution path. Process P_1, containing a selection construct, has two possible execution paths.

A slight reduction in the complexity of each FG_i is realized through the collapse of consecutive *computation* statements into a single representative node, c. We represent communication constructs as t^j, the transmission of a message to P_j, and r^j, the receipt of a message from P_j. Nodes representing statements in the body of a selection construct are bounded by an IF node and an ENDIF node. Figure 3 shows the control flow graphs derived from the example source code.

The nodes of FG_i represent syntactic constructs in the source code of P_i. An execution of P_i may be viewed as a traversal of FG_i, from α_i to ω_i. An event created from the execution of a statement in P_i corresponds to the locus of control passing through the node of FG_i representing that source code statement. In the remaining discussion of the flow graphs, the symbol representing a node of FG_i is also used to represent the event corresponding to the execution of the source code associated with that node. Nodes and events will be distinguished by context.

To summarize the discussion of FG_i, the graph FG_i represents all possible execution paths of P_i. A path from α_i to ω_i represents a single, possible execution of P_i. The order of events on the α_i to ω_i path indicates the execution order of those events if the selected path is executed. If both events v and v' are executed in P_i such that $v \rightarrow v'$, then a path in FG_i will exist from v to v'.

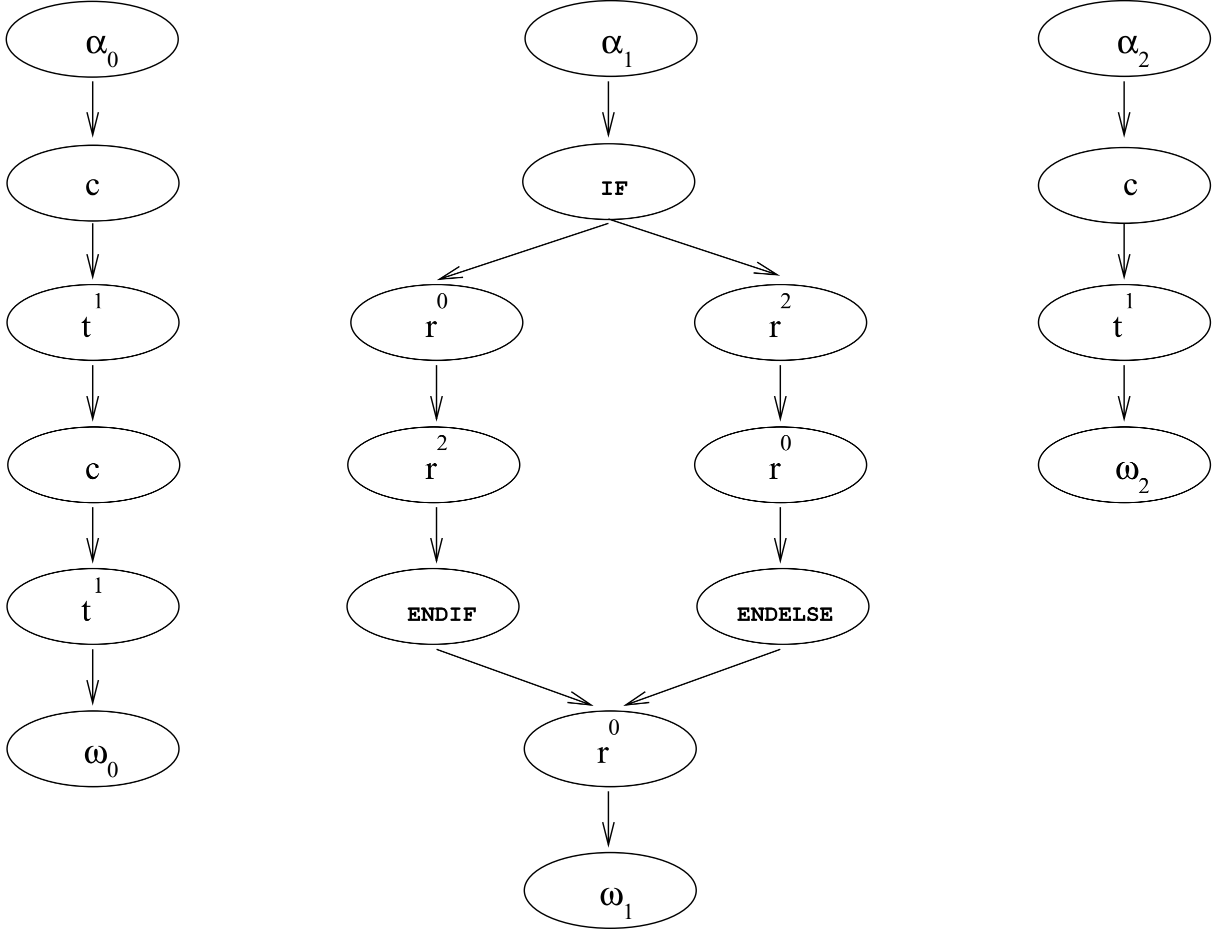


Fig. 3. Flow graphs for a simple 3 process system.

We define an *immediate communication successor* set, *ICS*(v), for each communication node, $v \in V_i$. Node v' is an immediate communication successor of node v if (1) there exists a path from v to v', (2) $v' \in \{t^j, r^j, \omega_i\}$, and (3) there does not exist a communication node $v'' \neq v'$ on the path from v to v'. Immediate communication successor sets are used in the following construction of the intermediate graph.

4. Intermediate graph

From the flow graphs of the distributed system's constituent processes, an intermediate graph $\mathcal{S}$ is constructed. This graph represents all possible executions by representing all possible partial orders of the system. Concurrency as well as causality among the events is preserved. A partial order graph, *POG*, will be constructed from $\mathcal{S}$ by combining nodes that are both causally equivalent and derived from equivalent partial orders.

We begin by generating *concurrent communication states*, *CCS*s, from the flow graphs. Each *CCS* is an ordered N-tuple, $(v_0, v_1, \ldots, v_{N-1})$, where $v_i \in V_i$ is either α_i, t^j, r^j, or ω_i. If $v_i \in \{t^j, r^j\}$, it denotes the next communication command to be executed in P_i. Each *CCS* is represented by a node in $\mathcal{S}$.

The root node of $\mathcal{S}$, CCS_0, contains the root nodes of each flow graph.

$$CCS_0 = \{\alpha_0, \alpha_1, \ldots, \alpha_{N-1}\}$$

Successor *CCS*s are generated using the set of immediate communication successors, *ICS*(v), for each entry, v, of the *CCS*. The $\mathcal{S}$ graph represents the hierarchy of successor *CCS*s.

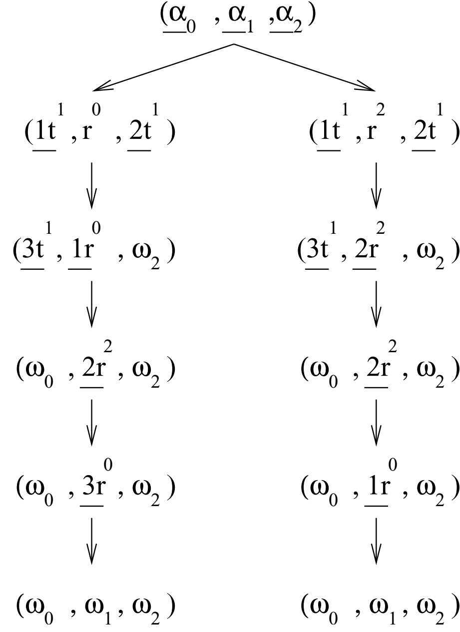


Fig. 4. Tree $\mathcal{S}$ for simple 3 process system.

Causal communication constraints force message receipts to be delayed until their corresponding transmissions have been processed. Those receives that can not be immediately processed are not "*ready*." A "*ready*" receive in a *CCS* is one whose corresponding transmission command occurred in an ancestor *CCS*. Root nodes of each flow graph, α_i, and message transmission elements, t^j, are intrinsically "*ready*".

All the communication commands of a *CCS* that are ready to execute are concurrent. Other concurrent relationships may also be present in $\mathcal{S}$ as will be examined in a later section.

A *CCS* with no ready elements has no successor states. The *immediate successor states*, *ISS*(*CCS*), are determined from the immediate communication successor sets of the *CCS*'s ready commands. The states in *ISS*(*CCS*) will be children of *CCS* in $\mathcal{S}$. The following steps determine *ISS*(*CCS*):

1. Let $\mathcal{R} = \{v_i : v_i \in$ *CCS* and v_i is ready$\}$.
2. Generate *CCS'* $\in$ *ISS*(*CCS*) by replacing each $v_i \in \mathcal{R}$ with an element of *ICS*(v_i).
3. Repeating step 2 until all unique *CCS'*s are generated from the ready commands' successor sets. The number of successor states of *CCS* is

$$|ISS(CCS)| = \Pi_{i=0}^{N-1} |ICS(v_i)| \text{ if } v_i \in \mathcal{R}$$

A *CCS* containing no ready commands and one or more receive commands (that are not ready) has no successors and is an *invalid terminal state* of the distributed system. A *CCS* comprised of all ω_i elements is a *valid terminal state*.

Ready communication events in each *CCS* will be annotated with an integer counter which will be used in later steps of *POG* construction. The annotation on each message transmission to P_i indicates the number of transmissions to P_i encountered in predecessors of the current *CCS*. For example, the entry $3t^1$ indicates that this message transmitted to P_1 has been preceded by two other messages sent to P_1. The number prepended to the transmit events does not imply the receipt order of messages. Message delivery order is determined by the execution order of receive events in the current partial order.

A ready receive is prepended with the number taken from the matching transmit. For example, the $r^j \in P_1$ matching the transmit, $3t^1$, would be prepended with the number 3. While transmissions are numbered sequentially, numbers associated with receipts may be unordered. Values attached to receipts will be used to distinguish textually identical receipts that occur in different orders, thus defined by different partial orders. Not only does the numeric

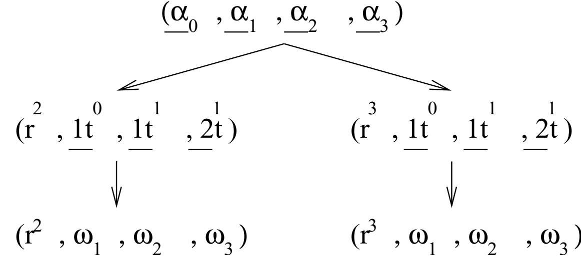


Fig. 5. Same partial orders.

attachment distinguish between syntactically identical communication commands, it also provides a method of matching transmissions with corresponding receipts.

Figure 4 shows the $\mathcal{S}$ graph derived from the flow graphs of Fig. 3. Appended numbers are shown, and "*ready*" events are underlined. Consider the left partial order in Fig. 4. The first node following the root consists of a receive in P_1 that is not ready and two ready transmissions that are sequentially numbered. In the successor node, the receive from P_0 is now ready since the the transmit from P_0 to P_1 occurred in an ancestor node. In the right partial order, the first ready receive of P_1 is a receipt from P_2. Referring back to the source code of Fig. 1, the order of receives in P_1 differs within the `if-else` construct.

Receives that are not ready in a node of $\mathcal{S}$ are indicators of what can happen next in a process. The ready events are the only events that occur, and they forge the causal relationships among events on different processes. If only the ready events of each node in $\mathcal{S}$ are considered, we observe that nodes can contain identical events. In the following section, the partial order graph of the system is constructed based on these observations.

5. Partial order graph

The construction of the partial order graph, *POG*, from the $\mathcal{S}$ graph is based on the observations outlined in the previous section. Only ready commands will be represented in the *POG*, where in the $\mathcal{S}$ graph, all commands were represented to facilitate the matching of transmits and receives. In some cases, two or more branches of $\mathcal{S}$ represent the same partial order. From the tree $\mathcal{S}$, the *POG* is constructed by combining branches that represent the same partial order into a single representative branch containing only ready commands.

Consider the simple $\mathcal{S}$ graph shown in Fig. 5 constructed from a four process system. The receives of P_0 are not ready. The transmits of each *CCS* are replaced in the child *CCS*s with a terminal node of FG_i. Both leaf node branches indicate that P_0 does not complete execution. The two branches shown represent the same partial order. Receives that are not ready are not part of the partial order.

A *POG* is a directed graph (N, A, η) where N is the set of nodes, A is the set of arcs, and $\eta \in N$ is the root node of *POG*. The nodes of the *POG* are generated from nodes of $\mathcal{S}$ such that the *POG* nodes represent the transmit and ready receive commands of the $\mathcal{S}$ nodes.

Since only the ready events of an $\mathcal{S}$ node need representation in a *POG* node, a node of the *POG* may not contain an element from each P_i. The executing process of an event in a node will not be positionally identified as in $\mathcal{S}$. Instead, subscripts will identify the process executing the command. A transmission entry has the format ct_i^j where c is the counter, i is the process executing the transmit and j is the destination process. A ready receive entry has the format cr_i^j where c is the counter, i is the process executing the receive and j is the sender.

In constructing the *POG*, it will be necessary to determine if a set of *CCS*'s has *equivalent* transmit and ready receive communication entries. Equivalent communications are found in *CCS* and *CCS'* if each ready communication command in one *CCS* exactly matches (in both prepended value and communicating partner identification) the positionally corresponding command in the other *CCS*. Specifically, *CCS* and *CCS'* have equivalent communications if the following conditions are met.

– At least one transmit or ready receive command is in either *CCS* or *CCS'*.
– $\forall_i$, if $v_i = c\chi^j \in$ *CCS* where χ is either t or r and $c\chi^j$ is ready, then $v_i = c\chi^j \in$ *CCS'* and $c\chi^j$ is ready.

```
VisitNodes ⇐ {}
CurrentPNode ⇐ Create_PNode( "root" )
VisitNodes ⇐ Enqueue <CurrentPNode, ISS(CCS_0)>
while ( VisitNodes not empty)
    item ⇐ Dequeue VisitNodes
    CurrentPNode ⇐ item.POGnode
    NodeSet ⇐ item.SnodeSet
    while (( equivNodes ⇐ Equiv_Set( NodeSet)) ≠ ∅)
        N ⇐ Create_PNode ( Create_Label( equivNodes))
        add arc from CurrentPNode to N
        SuccSet ⇐ {}
        for each Snode ∈ equivNodes
            SuccSet ⇐ SuccSet ∪ ISS(Snode)
        VisitNodes ⇐ Enqueue < N, SuccSet>
        NodeSet ⇐ NodeSet - equivNodes
    for each Snode ∈ NodeSet
        if ((Label ⇐ Create_Label(Snode)) ≠ NULL)
            N ⇐ Create_PNode( Label)
        else
            N ⇐ Create_PNode("end")
        add arc from CurrentPNode to N
```

Fig. 6. Algorithm to create *POG* from $\mathcal{S}$.

If *CCS* and *CCS'* have equivalent communication commands, the equivalent communication commands are all the transmit and ready receive commands that occur in *CCS* and *CCS'*.

We begin by traversing $\mathcal{S}$ breath-first and identifying the equivalent nodes. From these nodes, a representative node is created in the *POG* that is labeled with the equivalent communication commands of the identified nodes. An algorithm is presented for the *POG* construction. The following data structures and functions are essential to that algorithm.

VisitNodes :: a queue with entries of the format <POGnode, $\mathcal{S}$nodeSet>. POGnode is a reference to a node of the *POG* and $\mathcal{S}$nodeSet is a set of $\mathcal{S}$ nodes.

SuccSet :: a set of $\mathcal{S}$ nodes built from the successors of equivalent nodes.

Create_PNode(*CommoLabel*) :: creates a node of the *POG* and labels the newly created node with the string *CommoLabel*.

ISS($\mathcal{S}$*node*) :: constructs a set of nodes from $\mathcal{S}$ that are immediate successors of node $\mathcal{S}$*node*.

Equiv_Set($\mathcal{S}$*nodeSet*) :: extracts from $\mathcal{S}$*nodeSet* those nodes that have equivalent communication commands and returns these nodes as a set.

CreateLabel($\mathcal{S}$*nodeSet*) :: creates a label from the transmits and ready receives of the equivalent nodes in $\mathcal{S}$*nodeSet*. A string label is returned. Note that $\mathcal{S}$*nodeSet* may only have one entry and the label returned is for the ready commands of this one node.

The algorithm for creating the *POG* from $\mathcal{S}$ is given in Fig. 6. The construction prunes the tree $\mathcal{S}$ so that one branch of the *POG* from root to leaf node represents an unique partial order $\rho \in \mathcal{P}$. The resulting *POG* represents all partial orders of the system.

The communication behavior of the system directly influences the size of the *POG*. The width of the graph is determined by the number of possible partial orders. For each control branch that results in two communication choices, a new partial order is created thereby producing a branch in the *POG*. Admittedly, the *POG*'s width can grow large, but the graph accurately represents the complexity of the communication.

The worst case performance of the static analysis is exponential in the number of possible concurrent states. For the worst case, an unlikely system must exist. Assume every node of a flow graph can occur in the same concurrent state with every node from the other processes' flow graphs. If we let T be the number of nodes of all the processes' flow graphs, then an upper bound on the number of nodes of one flow graph is $O(T)$. The worst case bound on the number of concurrent states is $O(T^N)$, where N is the number of processes in the distributed application.

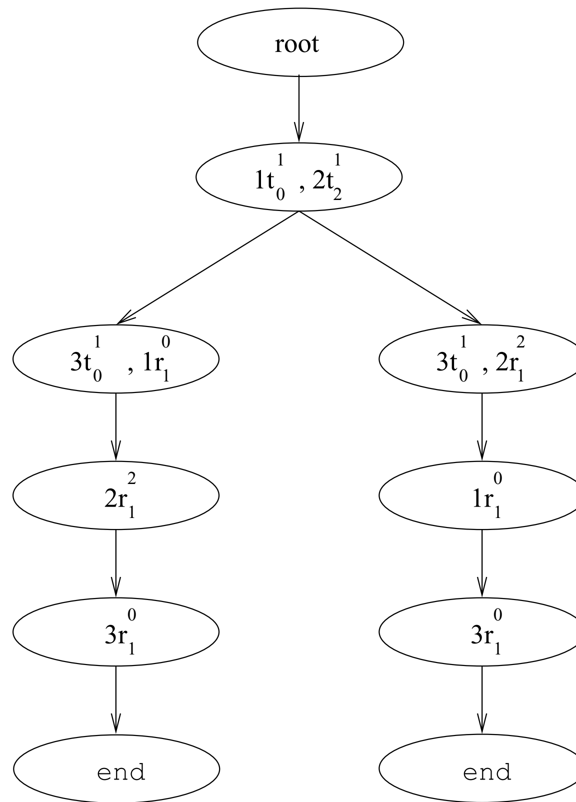


Fig. 7. *POG* derived from $\mathcal{S}$ of Fig. 4.

Although static analysis can have exponential performance, the time spent analyzing does not affect the execution of the distributed system. The analysis is done prior to execution, and provides insight into the application's behavior that can be leveraged at run-time.

Figure 7 shows the *POG* representing the example distributed program of Fig. 1, and generated from $\mathcal{S}$ of Fig. 4. Notice that the two partial orders of Fig. 2 are each represented as a path from root to an `end` node in the *POG*. In particular, the left path of the *POG* represents PO_1, and the right path of the *POG* represents PO_2.

6. Looping constructs

Three looping constructs are considered: `do-while`, `while` and `for`. Each type of loop has one unique entry point and one unique exit point. When loops are nested, each loop has it's own entry and exit point. Each loop in a process's source code is represented as a cycle in the process's corresponding flow graph. The cycle is accomplished with a *back edge* from the exit point of the loop to the entry point of the loop.

To demonstrate the incorporation of loops into the steps for generating the *POG*, two distributed programs have been selected. The first is set partition [4], SETPART, which was chosen for its concise applicability. SETPART consists of two distributed processes, P_0 and P_1, that partition disjoint integer sets S and T. P_0 maintains S and P_1 maintains T. An element of S is exchanged with an element of T until the elements of S are less than the elements of T. The source code of SETPART is shown in Fig. 8 and the corresponding flow graphs are shown in Fig. 9.

The second program is a distributed sort, DS, which was chosen for its increased size and complexity. A total of q integers are sorted in ascending order by six processes. The process are connected in a logical ring so that P_i's neighbors are P_{i-1} and P_{i+1}. Initially each process is assigned $q/6$ elements. Each process follows the general source code as shown with P_i in Fig. 10. The corresponding flow graphs are shown in Fig. 11.

```
P0 ::                              P1 ::
mx = max(S)                        while(true)
async_send(1,mx)                       async_recv(0,y)
S = S - mx                             T = T ∪ y
async_recv(1,x)                        mn = min(T)
S = S ∪ mx                             async_send(0,mn)
mx = max(S)                            T = T - mn
while(mx > x)                      endwhile
    async_send(1,mx)
    S = S - mx
    async_recv(1,x)
    S = S ∪ mx
    mx = max(S)
endwhile
```

Fig. 8. Set partition.

With the possibility of loops in the source code of each process, loops are also possible in $\mathcal{S}$. Additions are required for detecting the repeated execution of communication commands and representing these repetitions as cycles in $\mathcal{S}$. Cycles occur in $\mathcal{S}$ if

1. a send command is in the body of a loop,
2. a send command and its matching ready receive are each in a loop body, or
3. a combination of (1) and (2).

Following are the steps taken to incorporate loops into $\mathcal{S}$. When a node n is added to $\mathcal{S}$, a check is made to determine if the state represented by n has already been represented by another node in n's execution path. This is done by comparing n with its ancestors. First n is compared with its immediate predecessor, i.e., parent node. If the parent does not represent the same state, then the grandparent is compared against n. This continues until either a node that represents the same state of n is found or the root of $\mathcal{S}$ is reached.

Two comparisons are required two determine if node n and its ancestor node n' represent the same state. Nodes n and n' represent the same state if

- for each entry $v_i \in n$, there exists $v_i' \in n'$ that is identical with the exception of the counter value, **and**
- for each v_i and v_i' pair, v_i and v_i' correspond to the same node of FG_i.

If nodes n and n' represent the same state, then node n' is possibly the entry point of a loop, and the parent of n is possibly the exit point of this loop. The next decision is whether to add a back edge from the parent of n to n' to form the loop.

If n' is the parent of n, then a loop has been detected. A back edge is added from n' to itself and node n is removed from $\mathcal{S}$. If n' is not the parent of n, then n' must be an ancestor of the parent of n. That is, other nodes are found between n' and n in $\mathcal{S}$. To identify a loop, the body must be repeated as nodes in $\mathcal{S}$. A loop is detected only if the nodes from n' to the parent of n, the loop body, are repeated immediately following node n. A loop detection results in a back edge inserted from the parent of n to n' and the removal of n and its descendants from $\mathcal{S}$.

The $\mathcal{S}$ graph for SETPART is shown in Fig. 12 and the $\mathcal{S}$ graph for DS is shown in Fig. 13. Note that for SETPART only one cyclic behavior exists where as with DS four loops exist. The details of the algorithms and the implementation for generating $\mathcal{S}$ can be found in [14].

Loops incorporated into $\mathcal{S}$ propagate to the *POG*. Only the function Equiv_Sets() of the algorithm to create the *POG* needs modification to handle the back edges of $\mathcal{S}$. Additional tests are needed to determine the equivalency of $\mathcal{S}$ nodes. Suppose *CSS* and *CSS'* are found to have equivalent communication commands, and the nodes that represent *CSS* and *CSS'* are n and n'. Function Equiv_Sets() must check whether both n and n' have an incoming back edge or both n and n' have an outgoing back edge.

If no back edges are found, then n and n' are equivalent. If only one of the nodes is referenced by a back edge, then n and n' are not equivalent. When nodes n and n' are each pointed to by a back edge, both nodes are entry points of

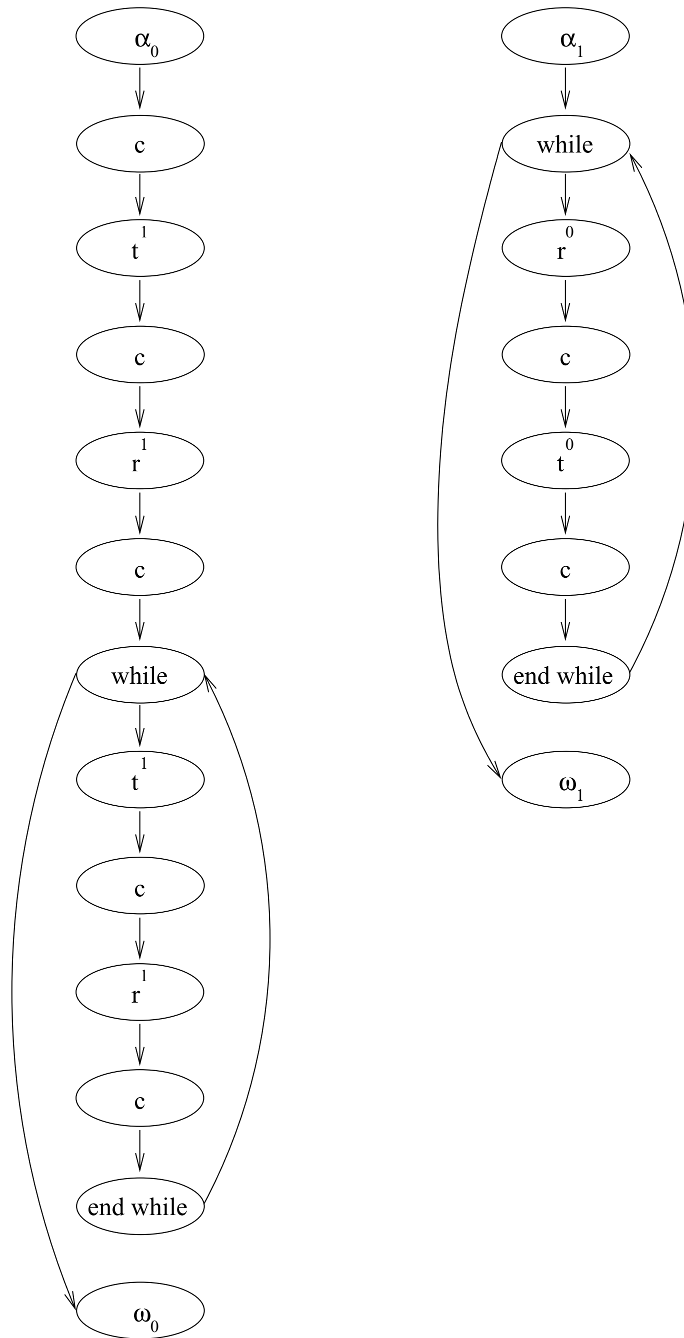


Fig. 9. Flow Graphs for SETPART.

loops in $\mathcal{S}$. Next is to determine whether the loop associated with node n is equivalent to the loop associated with node n'.

Suppose node b is the node that has a back edge to node n. Nodes n and b form a subtree rooted at n, where the nodes that are descendants of n but not descendants of b comprise the nodes of the subtree. Also suppose node b' is the node that has a back edge to node n', then nodes n' and b' also form a subtree. Nodes are compared as the subtrees are traversed in lock step, starting at the root nodes, in depth first order to determine if the loops are

```
Pi ::
    integer pid, phase
    arrays list, recv_list
    pid = i
    read q/6 elements into list
    sort list
    for phase = 0 to 5
        if phase is even
            async_send(i + 1 mod N, list)
            async_recv(i + 1 mod N, recv_list)
            list = merge_sort(list, recv_list, first)
        endif
        if phase is odd && pid != 0 && pid != N-1
            async_send(i − 1 mod N, list)
            async_recv(i − 1 mod N, recv_list)
            list = merge_sort(list, recv_list, last)
        endif
    endfor


merge_sort(list, recv_list, half)
    array merge_list
    merge_list = merging of recv_list and list
    sort merge_list
    if half = first
        return first half of elements in merge_list
    else
        return last half of elements in merge_list
    endif
```

Fig. 10. Distributed sort – Generic for all processes.

equivalent.

The current node c of one subtree is compared against current node c' of the other subtree. If the CCS of node c is equivalent to the CCS of node c', and the number of children of c is equal to the number of children c' then the traversal continues. If either condition is false, the loops are not equivalent and the traversal stops. If the subtrees are traversed without falsifying either condition, then the loops are equivalent and nodes n and n' are represented by a single node in the *POG*.

If both n and n' both have an outgoing back edge, then an addition test is required to determine the equivalence of the nodes. Let d be the node pointed to by the back edge of n and let d' be the node pointed to by the back edge of n'. If d and d' are equivalent, then so are n and n'. Both n and n' will be represented by a single node in the *POG*. A single back edge will be added from the *POG* node representing n and n' to the *POG* node representing the equivalent $\mathcal{S}$ nodes pointed to by the back edge of n and n'. Figures 14 and 15 show the *POG*s generated for SETPART and DS.

7. Event relationships

The *POG* conveys concurrent and causal relationships for each possible partial order. Some of the concurrent relationships are explicitly represented in each node of the *POG* where other concurrent and causal relationships can easily be derived. Consider the node in level 2 of Fig. 7. The two transmissions, $1t_0^1$ and $2t_2^1$, are explicitly shown

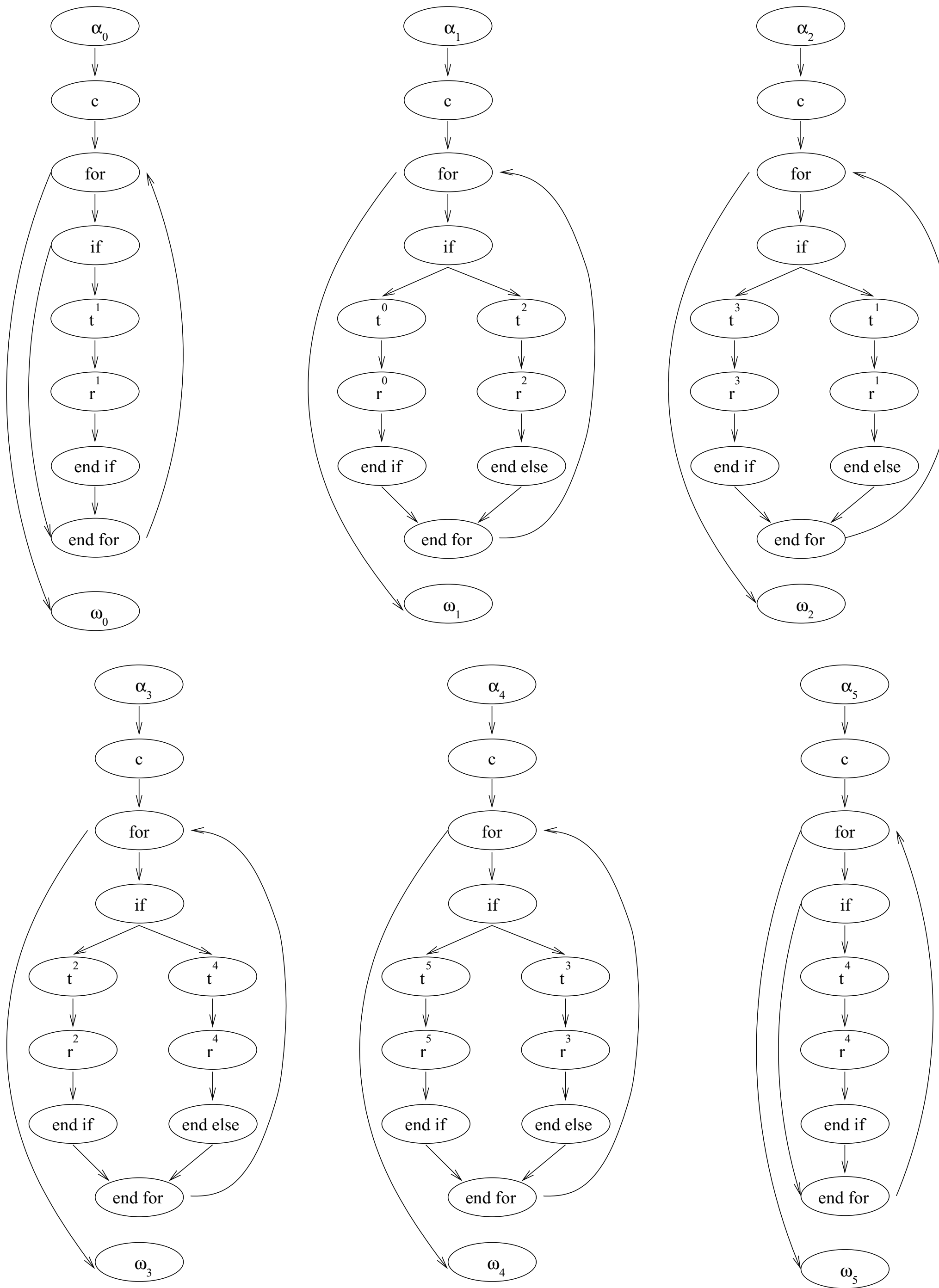


Fig. 11. Flow graphs for DS.

to be concurrent. Specifically, the communication events represented by a node of the *POG* are concurrent in that partial order. Other concurrency relationships are implicit and must be derived from absence of causality.

If a transmission event, ct_i^j, is an entry of node n of the *POG*, then the matching receive, cr_j^i, is found in a descendant node of n. For example, consider again $2t_2^1$ of level 2. The matching receive, $2r_1^2$, is found in the

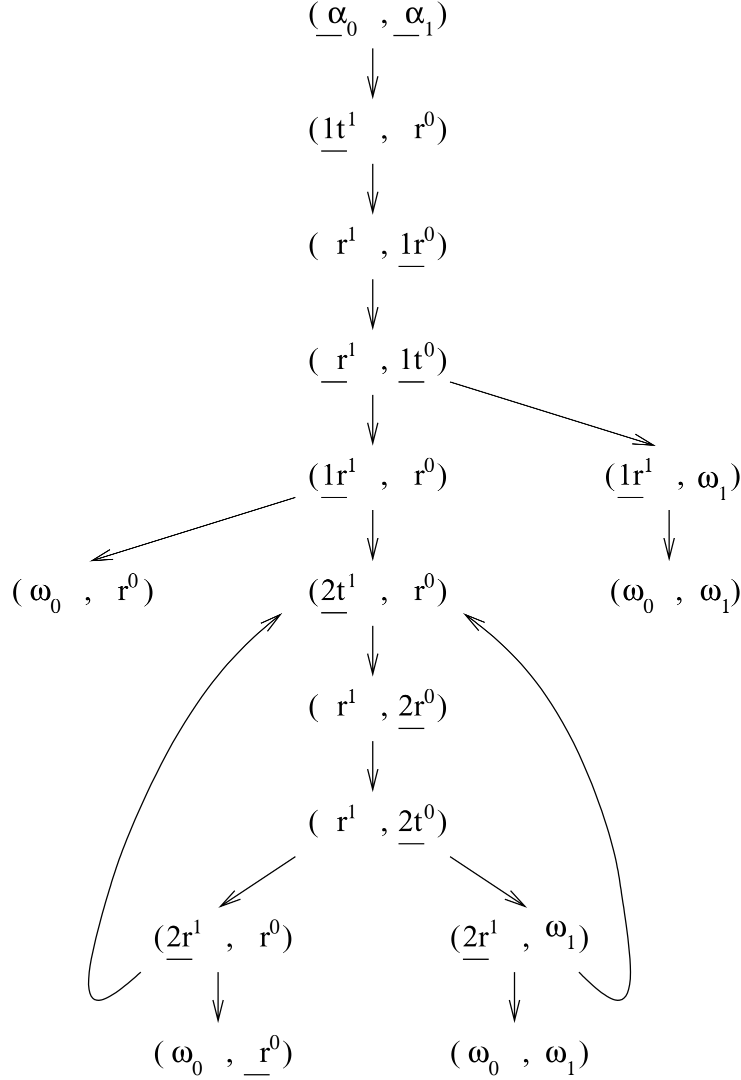


Fig. 12. $\mathcal{S}$ for SETPART.

left partial order in a descendant node at level 4. The matching receive is also found in the right partial order in a descendant node at level 3.

The positional relationships between nodes, as well as the event content of nodes, is used to derive relationships among events. Consider, for example, event $1r_1^0$ in level 3 and event $3r_1^0$ in level 5 of the left partial order. The common subscript indicates that both events are executed on P_1, thereby ensuring that a causal relationship exists between them. Since $1r_1^0$ occurs in an ancestor node of $3r_1^0$, we conclude that $1r_1^0 \rightarrow 3r_1^0$. Deriving the remaining interprocess relationships requires additional reasoning.

A communication transitive path is relevant to a particular partial order and follows the definition of *happens before*. A communication transitive path of $t + 1$ events, from e_j^0 to $e_{i \neq j}^t$, is a series of communication events $e_j^0, \ldots, e_i^t$ such that

- $e_k^v \rightarrow e_l^{v+1}$,where there does not exist an event e' that is an event of the path such that $e_k^v \rightarrow e' \rightarrow e_l^{v+1}$,
- for e_k^v and e_l^{v+1}, where $k \neq l$, e_k^v and e_l^{v+1} are a transmit/receive pair (e_k^v being the transmission and e_l^{v+1} being the receipt), and
- for e_k^v, e_l^{v+1}, where $k \neq l$, the next event of the path (if it exists), e_l^{v+2}, must occur on P_l.

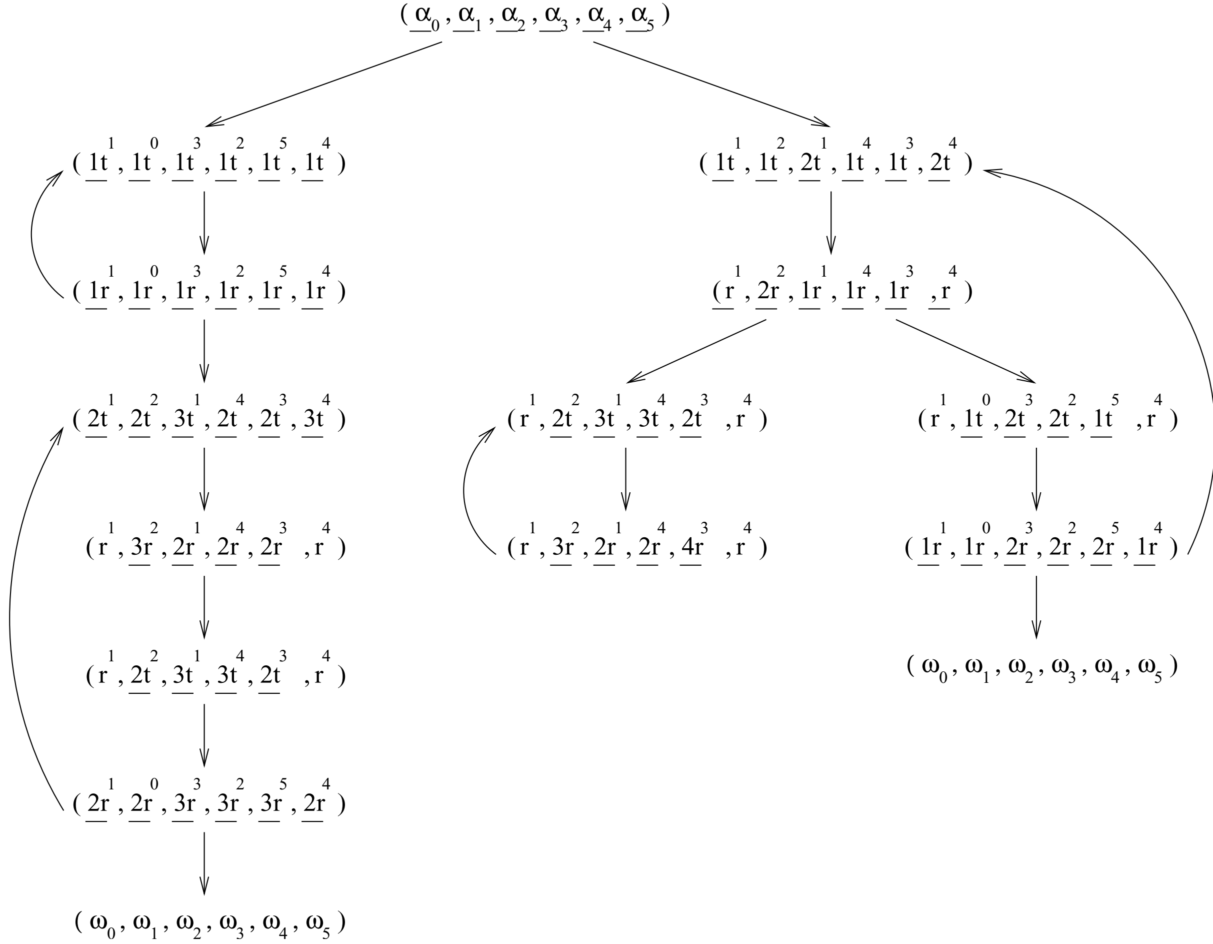


Fig. 13. $\mathcal{S}$ for DS.

If $e_j^0, e_k^1, e_k^2, e_l^3, e_l^4, e_i^5$ is a valid communication transitive path of length 6, e_j^0 is a transmission to P_k, and e_k^1 is the corresponding receipt, e_k^2 is a transmission to P_l, and e_l^3 is the corresponding receipt, and e_l^4 is a transmission to P_i and e_i^5 is the corresponding receipt.

For any two communication events, e and e', represented in the *POG* , exactly one of the following relationships will hold:

- e and e' are in different partial orders;
- e and e' are concurrently related, $e \| e'$; or
- e and e' are causally related, either $e \rightarrow e'$ or $e' \rightarrow e$.

If node n, representing event e, and node n', representing event e', are not on a single path from the root to the END, then the nodes are contained in different partial orders. The two events can not occur in the same execution since they are in different partial orders. Causal and concurrent relationships do not exist between events in different executions. Therefore, e and e' are not related: neither through causality nor concurrency.

If, on the other hand, a single path from the root to the END contains both n and n', then the nodes are contained in the same partial order. Events e and e', now from the same execution of the system, are related. If a communicative transitive path exists from e to e', then $e \rightarrow e'$. If no communicative transitive path exists between the events, then $e \| e'$; the events are concurrent.

The information derived from the *POG* is not limited to a single execution of the system. Since all partial orders are represented, questions regarding the possibilities of execution can be answered. For example,

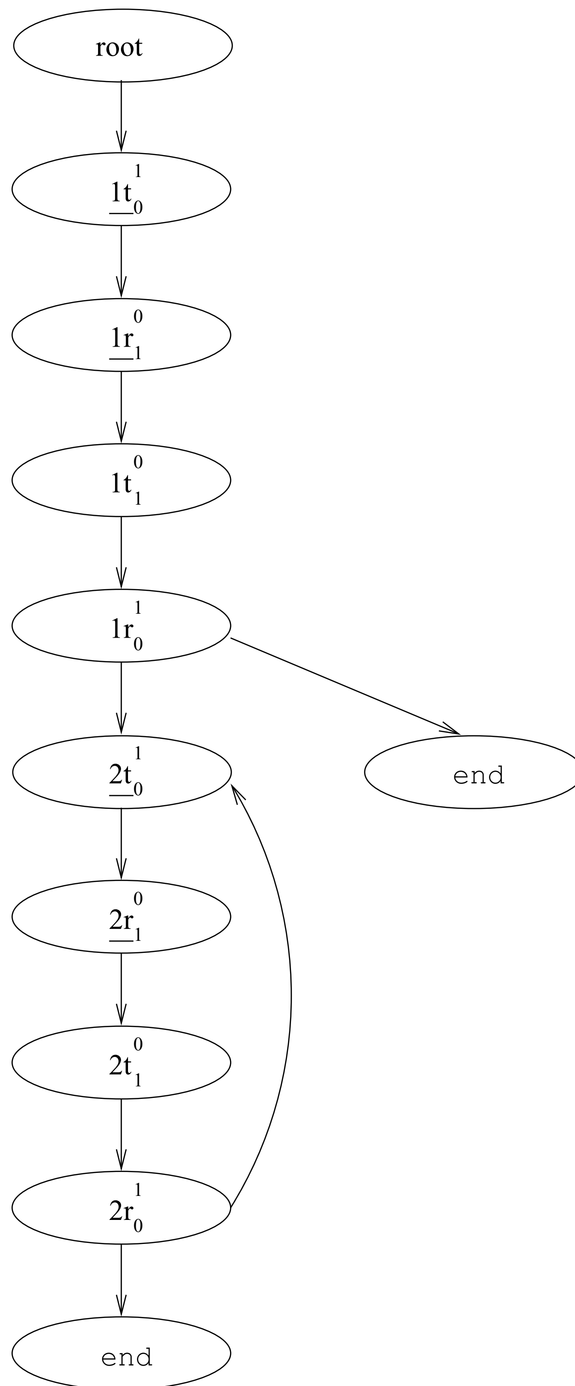


Fig. 14. *POG* for SETPART.

– Is it possible that e and e' can execute concurrently?
– Is there a single event that will always precede event e?
– Which events can have an immediate causal effect on e?
– Can event e have a causal effect on event e'?

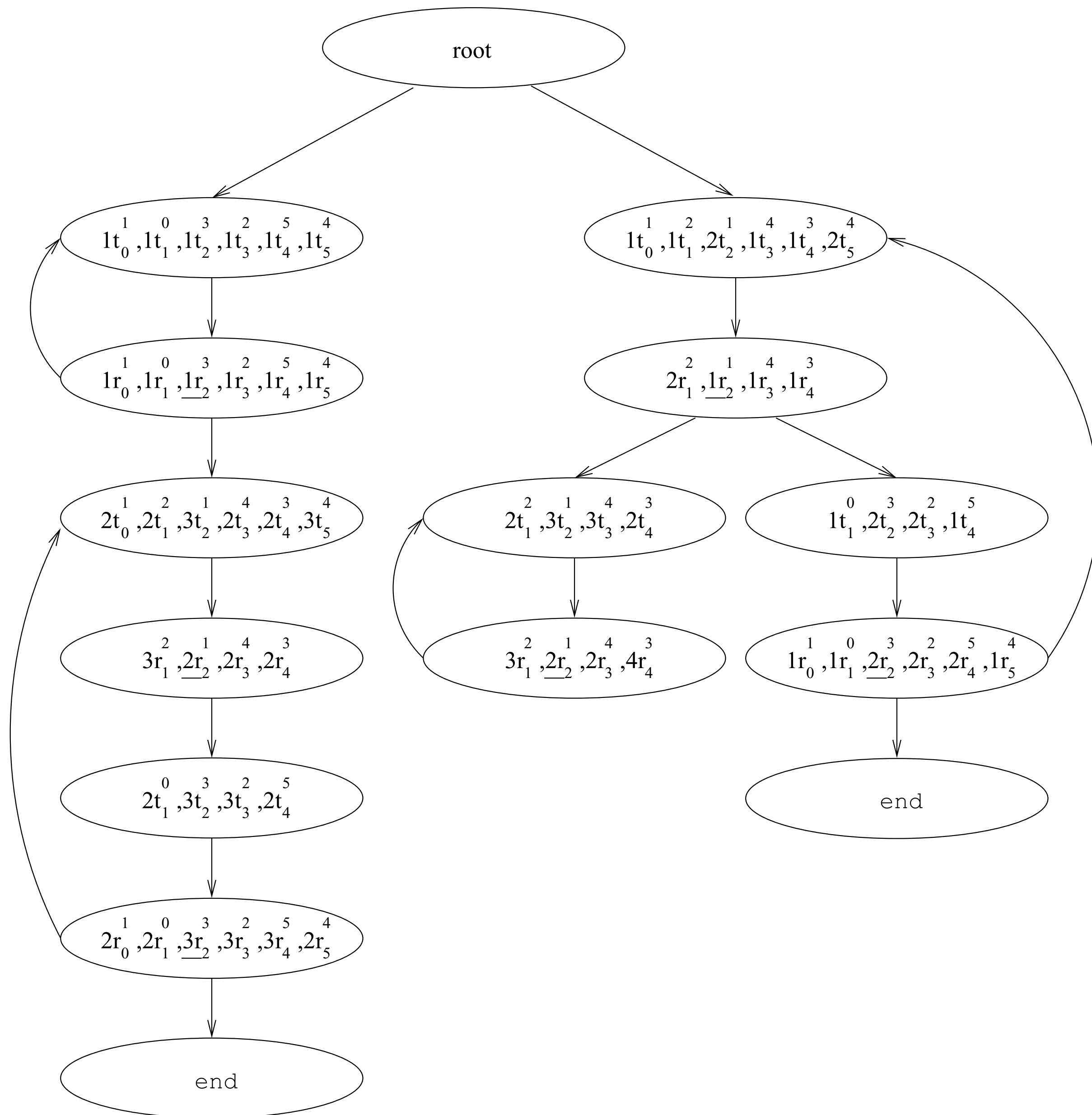


Fig. 15. *POG* for DS.

These questions indicate the significance of a unified representation of a distributed system. Challenges of distributed systems that require reasoning about event relationships can be answered with respect to one, any, or all possible executions.

8. Monitoring execution

Reasoning about causal and concurrent relationships is fundamental to monitoring and debugging distributed systems. Monitoring the execution of a distributed program requires reasoning about constituent processes' execution as a single collective entity. We have extrapolated the semantics of the assert statement for sequential programs into

```
P0 ::                              P1 ::
mx = max(S)                        while(true)
async_send(1,mx)                      async_recv(0,y)
S = S - mx                            T = T ∪ y
async_recv(1,x)                       mn = min(T)
S = S ∪ mx                            async_send(0,mn)
mx = max(S)                           assert(y = max(S) ≥ mn > x ∧
while(mx > x)                         |S| = |S0|∧ S ∩ T = y)
   async_send(1,mx)                   T = T - mn
   S = S - mx                      endwhile
   async_recv(1,x)
   S = S ∪ mx
   mx = max(S)
endwhile
```

Fig. 16. Set partition with assert.

the distributed context and developed a run-time methodology for monitoring and debugging using distributed assert statements. The *POG* is the basis for the efficient evaluation methodology of the distributed assert statement.

The distributed assert has the format

$$\mathbf{assert}(P)$$

where P is a global predicate that is anchored at a control point of one processes and evaluated when the process executes the assert. If P is true then the program continues its execution. If P is false, however, the program is aborted, and a diagnostic message is produced.

Our global predicate is not restricted by the format of the logical comparisons, and variables from different processes can be compared directly. Both stable and unstable properties can be monitored. A distributed assert statement monitors a distributed system's execution, but only a subset of the system execution states are relevant for evaluation. In particular, the distributed assert presented in this paper monitors the execution having the most recent causal impact on the assert statement.

The property that is monitored is determined by the predicate. Examples of deadlock detection, mutual exclusion violation and specific behaviors of a program can be found in [14]. Distributed assert statements have been developed for the two example programs SETPART and DS as shown in Figs 16 and 17, respectively.

The assert statement in P_1 of SETPART,

$$\mathbf{assert}(y = \max(S) \geqslant mn > x \wedge |S| = |S_0| \wedge S \cap T = y)$$

is evaluated on each exchange of data between the processes. The clauses of the assert predicate compare variables from both processes. A false evaluation indicates erroneous execution of the program. SETPART's error is identified by the assert's falsifying clause. If y is not equal to max(S); P_0 did not send the correct value. If $\max(S) \ngeqslant mn$; processing should have stopped on the last exchange, and a likely error is P_0's exchange loop condition. If $mn \ngtr x$; either a value other than the minimum of T was chosen, or P_0 has erroneously altered the variable x since the last exchange. If the new size of S has changed, P_0 has not correctly added or removed a value from S since the last exchange. If the intersection of S and T is not equal to y; either S or T has not been correctly updated since the last exchange, and the results of the other clauses help in identifying the incorrect set. For example, suppose the programmer mistypes `async_send(1, mx)` of P_0 as `async_send(1, x)`. The clause y =max(S) evaluates to false and identifies an incorrect value sent by P_0.

For DS, process P_2's source code is shown below with the two assert statements A_{2a} and A_{2b}. Two assert statement could be in any one of the six processes and provide the same meaningful information. Process P_2 was arbitrarily selected. The clause $P_i.list \leqslant P_i.recv_list$ tests whether every element in $P_i.list$ is less than or equal to all elements of $P_i.recv_list$, and the clause $P_i.list \geqslant P_i.recv_list$ tests whether every element in $P_i.list$ is greater than or equal to all elements $P_i.recv_list$. The clause $P_i.recv_list = P_{i+1}.list$, for $i = 2 \ldots 4$, of assert A_{2a} determines whether process P_i received the correct list from its right neighbor P_{i+1}. The clause $P_i.recv_list = P_{i-1}.list$, for

```
P2 ::
    integer pid, phase
    arrays list, recv_list
    pid = 2
    read q/6 elements into list
    sort list
    for phase = 0 to 5
        if phase is even
            async_send(3,list)
            async_recv(3,recv_list)
            list = merge_sort(list, recv_list, first)
A2a         assert( P2.list ≤ P2.recv_list ∧ P2.recv_list = P3.list ∧
                P3.list ≤ P3.recv_list ∧ P3.recv_list = P4.list ∧
                P4.list ≤ P4.recv_list ∧ P4.recv_list = P5.list ∧
                P5.list ≥ P5.recv_list )
        endif
        if phase is odd && pid != 0 && pid != N-1
            async_send(1,list)
            async_recv(1,recv_list)
            list = merge_sort(list, recv_list, last)
A2b         assert( P2.list ≥ P2.recv_list ∧ P2.recv_list = P1.list ∧
                P1.list ≥ P0.recv_list ∧ P1.recv_list = P0.list ∧
                P0.list ≤ P0.recv_list )
        endif
    endfor


merge_sort(list, recv_list, half)
    array merge_list
    merge_list = merging of recv_list and list
    sort merge_list
    if half = first
        return first half of elements in merge_list
    else
        return last half of elements in merge_list
    endif
```

Fig. 17. P_2 of Distributed Sort with asserts.

$i = 1 \dots 2$, of assert A_{2b} determines whether process P_i received the correct list from its left neighbor P_{i-1}. The clauses $P_i.list \leqslant P_i.recv_list$ and $P_i.list \geqslant P_i.recv_list$ ensure that `merge_sort( )` correctly sorted and halved the merged list.

For a distributed assert to be evaluated during run-time, the local state information of the constituent processes must be shipped to the process executing the assert. This is necessary since the assert predicate can compare variables from the different processes but the value of these non-local variables are not available in the process executing the assert.

A brute-force implementation of evaluating distributed assert statements would have each process piggyback its state information on each of its outgoing messages. This approach would alter every message in the distributed execution and porportionally increase the execution time of the system [14]. By using the *POG*, we can reduce the number of messages piggybacking state information. The amount of reduction varies according to the distribruted program, but, in general, the reduction is significant. For each partial order, exactly one message in each process is identified for piggybacking the correct state information.

If event e in P_i is the evaluation of the assert statement, $LCP(e, j)$ where $j \neq i$, denotes event e's *latest causally preceding* in P_j. We define $LCP(e, j) = f$ if and only if f is an event in P_j, such that f happens before e, and

there does not exist an event f' in P_j such that f happens before f' and f' happens before e. Given a partial order and an event e, there exist exactly one LCP event in each process.

The LCP events are the means by which we reduce the number of messages piggybacking state information, which can be identified from the *POG*. By using the *POG*, a distributed assert is evaluated but the methodology neither generates and analyzes traces, nor adds messages to the original distributed application. Only the LCP messages are increased in size to propagate state information to the assert control point for assertion evaluation [16].

For event e, a *causal cut* through e is the set of events consisting of e and the LCP event of e of each process for a partial order α. The causal cut through event e, denoted $CC(e)$, is defined as

$$CC(e) = \{e\} \bigcup_{\substack{0 \leqslant k < N \\ k \neq i}} \{\mathit{LCP}(e, k)\}\,.$$

Intuitively, $CC(e)$ is the "latest" set of events of a partial order which can have a causal impact upon e. For each partial order, there is one causal cut for a given event. The LCP events that comprise the causal cut for one partial order may differ from the LCP events that comprise the causal cut for a different partial order.

From each causal cut *CC*, a global state exists for evaluating a distributed assert statement. If e is the execution of a causal distributed assert statement in P_i, then the *causal global state*, anchored on e, is

$$\mathit{CGState}(e) = \{\mathit{pre}(f) : f \in \mathit{CC}(e)\}.$$

The $\mathit{pre}(f)$ denotes the local state of P_i in which the execution of f is begun. When the system executes, one of the possible partial orders is identified. The global state corresponding to this partial order can then be used for assertion evaluation.

In [16], it is proved that *LCP* events are communication events. In particular, for event e of P_i, each *LCP*(*e,j*) is a send event. Corresponding to each *LCP* send event is a receive event, denoted *LCP'*. A causal cut consists of *LCP* send events. The *LCP* and *LCP'* events of a partial order comprise the communication events that are sufficient for delivering the *CGState* data to the process evaluating the assert. In Figs 14 and 15 the underlined entries indicate the *LCP* and *LCP'* events.

In [14,15], a two-pass compiler prototype system for enabling the evaluation of a causal distributed assert statement is presented. This system ensures that when an assert is executed, the relevant components of the causal global state are immediately available at the process executing the assert. From the *POG*, the causal cuts can be identified prior to program execution. Our prototype statically analyzes the distributed source code to

– generate the *POG*,
– identify the *LCP* and *LCP'* events of the distributed assert statements from the *POG*, and
– append the necessary causal global state information to the already existing *LCP* send commands.

9. Conclusions

This paper has presented a methodology for analyzing the communication of a distributed system. Source code is analyzed to produce a partial order graph, *POG*, that conveys all possible execution scenarios. In particular, the *POG* represents all possible causal relationships that can be forged by executing the system's processes. Concurrent relationships are upheld in the *POG* by not imposing an order on the events where causality does not exists.

Our methodology first produces a graph representing the flow of control through each program of the distributed system. These flow graphs represent the possible orders of execution of statements in the source code. Each path through the graph from root to leaf provides the execution order of events within a single execution of the program.

The flow graphs of the individual programs are combined to create a unified representation of the system, resulting in an intermediate graph $\mathcal{S}$. The graph $\mathcal{S}$ represents all partial orders of execution while preserving concurrency. However, the graph may contain duplicate nodes that can be combined to produce a more compressed, yet still accurate, representation of the system.

The final resulting graph, the *POG*, is constructed from the intermediate graph by combining duplicate representations of the same partial order. The *POG* is derived from $\mathcal{S}$ so that each path from the root node to a leaf node

represents one partial order and each partial order is presented in one path of the *POG*. The generation of the *POG* from C source code has been accomplished with a prototype system.

Causal and concurrent relationships can be derived from the *POG* and this information can aid in solving difficult distributed system challenges. Distributed assert statements provide a means to monitor the execution of a distributed system. By using the causal information contained in the *POG*, messages can be identified prior to execution for delivering the state information to the assert's control point for run-time evaluation. The *POG* enables the identification of the *latest causally preceding* messages relative to the assert's control point. A prototype two-pass compiler exists for identifying the *latest causally preceding* messages and automating the piggybacking of relevant statement information on these messages. The *POG* enables this ability.

References

[1] J.H. Anderson, *Lamport on Mutual Exclusion: 27 Years of Planting Seeds*, In Proceedings of the twentieth annual ACM symposium on Principles of distributed computing, ACM Press, 2001, 3–12.

[2] K.M. Chandy and L. Lamport, Distributed snapshots: Determining global states of distributed systems, *ACM Transactions on Computer Systems* **3**(1) (1985), 63–75.

[3] R. Cooper and K. Marzulo, *Consistent Detection of Global Predicates*, In Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, 1991, 163–173.

[4] E.W. Dijkstra, A correctness proof for networks of communicating process – A small exercise. Technical Report EWD-607, Burroughs, 1977.

[5] E.N. (Mootaz) Elnozahy, L. Alvisi, Y.-M. Wang and D.B. Johnson, A survey of rollback-recovery protocols in message-passing systems, *ACM Comput Surv* **34**(3) (2002), 375–408.

[6] C.J. Fidge, *Partial Orders for Parallel Debugging*, In Proceedings Workshop Parallel and Distributed Debugging, University of Queensland, May 1988, 183–194.

[7] V. Garg and N. Mittal, *On Slicing a Distributed Computation*, In In Proceedings of the 21st IEEE International Conference on Distributed Computing Systems, 2001, 322–332.

[8] F.C. Gartner, Fundamentals of fault-tolerant distributed computing in asynchronous environments, *ACM Comput Surv* **31**(1) (1999), 1–26.

[9] B. Gupta and S.K. Banerjee, A roll-forward recovery scheme for solving the problem of coasting forward for distributed systems, *SIGOPS Oper Syst Rev* **35**(3) (2001), 55–66.

[10] L. Lamport, Time, clocks, and the ordering of events in a distributed system, *Communications of the ACM* **21**(7) (1978), 558–565.

[11] M.S. Meier, K.L. Miller, D.P. Pazel, J. syula R. Rao and J.R. Russell, *Experiences with Building Distributed Debuggers*, In Proceedings of the SIGMETRICS symposium on Parallel and distributed tools, ACM Press, 1996, 70–79.

[12] N. Mittal and V.K. Garg, *Debugging Distributed Programs using Controlled Re-Execution*, In Proceedings of the nineteenth annual ACM symposium on Princi ples of distributed computing, ACM Press, 2000, 239–248.

[13] R. Schwarz and F. Mattern, Detecting causal relationships in distributed computations: In search of the holy grail, *Distributed Computing* **7**(3) (1994), 149–174.

[14] S. Simmons, *Causal Distributed Assert Statement*, PhD thesis, The College of William and Mary, 1999.

[15] S. Simmons and P. Kearns, *A Causal Assert Statement for Distributed Systems*, Proceedings of the Seventh IASTED/ISMM International Conference on Parallel and Distributed Computing and Systems, IASTED-ACTA Press, 1995, 495–498.

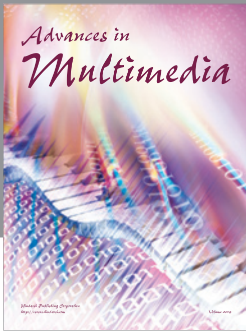
[16] S. Simmons and P. Kearns, *Runtime Evaluations of a Distributed Assert*, Proceedings of the ISCA Fifteenth International Conference on Parallel and Distributed Systems, 2002, 179–186.

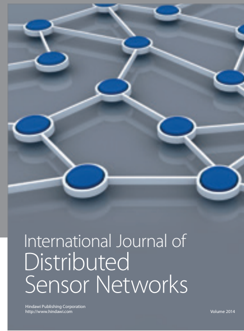
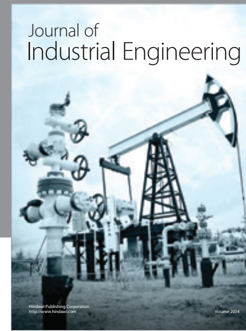
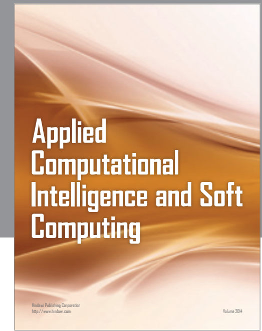
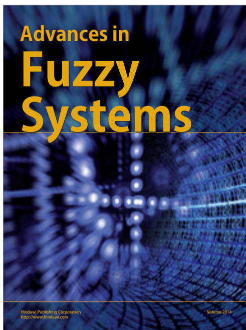
[17] A.S. Tannenbaum and M. van Steen, *Distributed Systems*, Prentice Hall. 2002.

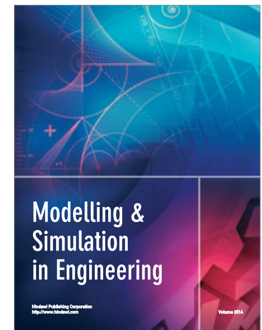
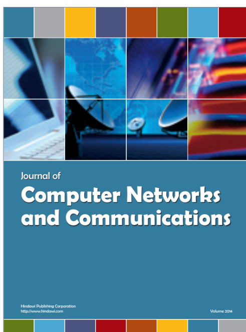
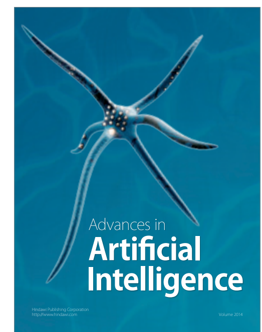
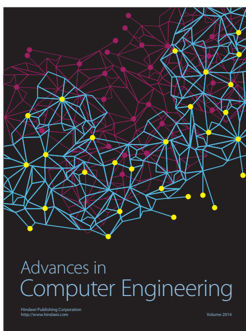
[18] A. Tarafdar and V.K. Garg, *Addressing False Causality while Detecting Predicates in Distributed Programs*, In Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98), Amsterdam, The Netherlands, 1998, 94–101.

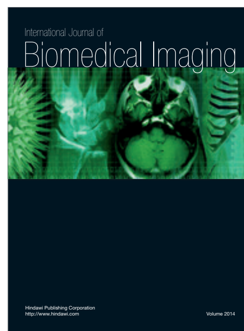
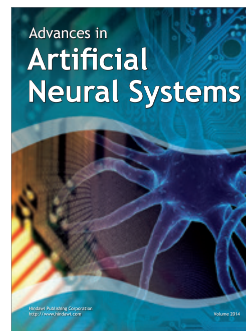
[19] R.N. Taylor, A general-purpose algorithm for analyzing concurrent programs, *Communications of the ACM* **26**(5) (May 1983), 362–376.

[20] G. Tel and F. Mattern, The derivation of distributed termination detection algorithms from garbage collection schemes, *ACM Trans Program Lang Syst* **15**(1) (1993), 1–35.

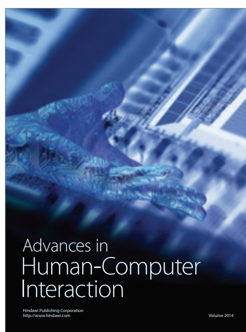
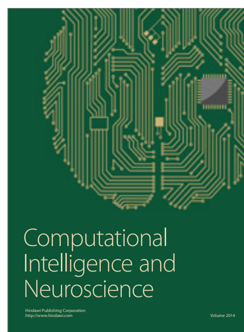

Advances in
Multimedia

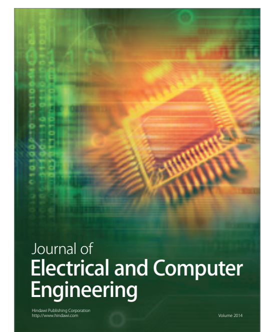
The Scientific
World Journal

International Journal of
Distributed
Sensor Networks

Journal of
Industrial Engineering

Applied
Computational
Intelligence and Soft
Computing

Advances in
Fuzzy
Systems

Hindawi
Submit your manuscripts at
http://www.hindawi.com

Modelling &
Simulation
in Engineering

Journal of
Computer Networks
and Communications

Advances in
Artificial
Intelligence

Advances in
Computer Engineering

International Journal of
Computer Games
Technology

International Journal of
Biomedical Imaging

Advances in
Artificial
Neural Systems

Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer
Interaction

Computational
Intelligence and
Neuroscience

International Journal of
Reconfigurable
Computing

Journal of
Electrical and Computer
Engineering