DYNAMIC SLICING OF DISTRIBUTED PROGRAMS

BOGDAN KOREL*, ROGER FERGUSON*

A static slice S of a sequential program Q is an executable part of Q that computes the same function as Q does in a subset of variables at some selected point of interest. As originally introduced, it involves all potential terminating program executions. In debugging, however, we typically deal with a particular incorrect execution and are interested in locating the cause of incorrectness of that execution. Therefore, we are interested in a slice, referred to as a dynamic slice, that preserves the program's behavior for a specific input. The concept of dynamic slicing has been initially introduced for sequential programs. In this paper we extend the concept of dynamic slicing to distributed programs. A distributed program introduces new challenges (e.g., nonreproducible behaviors, nondeterministic selection of communication events, etc.) that are not captured by the original definition of dynamic slicing. Consequently, the concept of dynamic slicing needs to be extended in order to make it a useful tool for debugging distributed programs.

1. Introduction

A slice S of a sequential program Q is an executable part of Q that computes the same function as Q does in a subset V of variables at some selected point of interest p (Weiser, 1984; Horowitz *et al.,* 1990). The program slice S consists of all statements in Q that potentially affect variables in V at p. Slicing has been shown useful in program debugging by narrowing the size of the suspected part of incorrect code. As originally introduced, it involves all potential terminating program executions, including those which are infeasible, that is, those for which there is no input data that causes their actual execution. In debugging, however, we typically deal with a particular incorrect execution and are interested in locating the cause of incorrectness (programming fault) of that execution. Therefore, we are interested in a slice that preserves the program's behavior for a specific input, rather than that for a set of all inputs for which the program terminates. To emphasize its dependence on a particular program execution, it is referred to as dynamic; in contrast, the original slice, defined on the program flow graph, will be referred to as static (Weiser, 1984; Horowitz *et al.,* 1990).

The concept of dynamic slicing was originally introduced for sequential programs in (Korel and Laski, 1988). The algorithms to find dynamic slices have been

* Department of Computer Science, Wayne State University, Detroit, MI 48202, USA

presented in (Korel and Laski, 1990b; Agrawal and Horgan, 1990). Our experience with dynamic slicing for sequential programs has shown that it can significantly reduce the size of the program as compared to static slicing. This can lead to more efficient fault localization because the searching space for a fault can be reduced. We believe, therefore, that dynamic slicing is better suited for the purpose of debugging (in particular, fault localization) than static slicing.

In this paper, we extend the concept of dynamic slicing to distributed programs; in particular, for Ada–like programs. Distributed programs introduce several problems which do not exist in sequential programs. Consequently, the concept of dynamic slicing must be extended in order to make it a useful tool for debugging of distributed programs. One of the major problems in distributed software is the reproducible program execution. Distributed programs often make nondeterministic selections of communications events, e.g., the select statement in Ada. Thus repeated executions of a distributed program with the same input data may result in the execution of different program paths. The results of the program may not be reproducible by simply reexecuting the program. In order to reproduce the execution of a distributed program, not only program input data but also appropriate choices for the nondeterministic selections must be provided. Tai et al. have investigated the reproducible–testing problem for Ada tasking programs (Tai, 1985; Tai *et al.*, 1991). They have suggested a method to explicitly control the execution sequence by introducing a run–time scheduler (much like the monitor tasks used in distributed debugging) that ensures that the execution order of the communication statements matches that of a predefined or recorded event sequences, i.e., a distributed program is initially executed on some input, and event sequences are recorded during execution of this program. Before any communication event is allowed to occur by the run–time scheduler, the event must be the next one specified in the event sequence. If the next event in the event sequence is not called for, then the program (task) is suspended by the scheduler to allow the *proper* event to occur. In this manner, the sequence of events is enforced during distributed program execution.

In order to use dynamic slicing to debug distributed programs, it is required that event traces be recorded during distributed program execution by, for example, a distributed debugger. Suppose during program execution an incorrect value of variable v has been observed at a certain point of execution of task t of a distributed program. In order to derive the dynamic slice for this incorrect variable, we use dynamic influence analysis which determines those events in the program's execution that affected the incorrect value of variable v. From this analysis, a partial program text, i.e., dynamic slice, is derived. In this paper we will show that taking into account a particular execution of a distributed program might significantly reduce the size of the slice. Moreover, the slice can be further reduced by the run time handling of arrays, pointers (Korel and Laski, 1988; Korel, 1990a; Chan and Chen, 1987), and tasks. The main advantage of dynamic slicing is that, when used for debugging, it provides finer localization information by reducing the search space for the fault in the program; this can make it easier for programmers to locate the causes of incorrectness.

Dynamic slicing is described for distributed programs written in Ada, although the results have applicability to any programs that use rendezvous-like synchronization. Because of the complexity of the synchronization constructs in Ada, we will not consider the entire Ada language. In this paper we consider only a subset of Ada which does not include selective wait's with else parts, conditional and timed entry calls, delay statements, dynamic creation of tasks, and abort statements.

The organization of this paper is as follows. In the next section, basic concepts and notations are introduced. Section 3 introduces the basic concept of dynamic slicing for distributed programs. Dynamic influence concepts are presented in Section 4. The sequential and distributed algorithms for finding dynamic slices are presented in Section 5. Finally, in the Conclusions, further research is outlined.

2. Basic Concepts

In this paper a graph model of distributed programs is used. In particular, each Ada program task defines a flow graph; each statement in the task is represented by a node in the flow graph; each transfer of control is represented by a direct edge. We distinguish two types of nodes in Ada programs: synchronization nodes and non-synchronization nodes. ***Synchronization nodes*** represent entry call statements and accept statements. All other nodes that do not constitute synchronization nodes are called ***non-synchronization nodes***, i.e., assignment statements, input/output statements, the predicate part of if-then-else, or loop statements (referred to as a *test node*), etc. An ***internal edge*** corresponds to a possible transfer of control between non-synchronization nodes inside a task; an internal edge associated with a test node will be called a ***branch***. A ***synchronization edge*** between synchronization nodes of different tasks represents the Ada rendezvous. In particular, there exists a synchronization edge between a node (on the caller task) for the event of calling the rendezvous, and a node (on the callee task) for the event of accepting the call. In addition, there is a synchronization edge between a node (on the callee) for the event of exiting from the rendezvous block, and a node (on the caller) for the event of returning from the rendezvous call.

Let $T = \{t_1, t_2, ..., t_m\}$ denote a set of all tasks in Ada program Q. A *flow graph* of task $t \in T$ is a directed graph $G_t = (N_t, A_t, s_t)$ where:

i) N_t is a set of nodes,

ii) $A_t \subseteq N_t \times N_t$ is a binary relation on N_t, referred to as a set of edges, and

iii) s_t is a unique entry node, $s_t \in N_t$.

An *edge* $(n_k, n_j) \in A_t$ corresponds to a possible transfer of control from node n_k to node n_j. For instance, (2,3), (2,4), and (5,6) are edges in the task $R1$ of the program of Figure 1. An edge (n_k, n_j) is called a branch if n_k is a test node.

A *path* L_t in a flow graph of task t is a sequence $L_t = < n_{k_1}, n_{k_2}, ..., n_{k_q} >$ of nodes, such that $n_{k_1} = s_t$, and for all j, $1 \leq j < q$, $(n_{k_j}, n_{k_{j+1}}) \in A_t$. For example, $L_S =< 1,2,3,4,17,18,19,20,9,10,11,12,5,6,7,8,9,10,11,12,13,14,15,16 >$ is a path of task S in program of Figure 1.

Notationally, a path L_t will be represented as an abstract list (Jones, 1980) whose elements can be accessed by position (sometimes referred to as an execution position) in it, that is, $L_t[j]$ will denote the j–th element (node) of L_t, e.g., for L_S given above $L_S[4] = 4$, $L_S[6] = 18$, $L_S[8] = 20$, etc. To distinguish between multiple occurrences of the same node in the path we will sometimes refer to a node together with its position in a path, rather than to the node itself, e.g., node 4 at position 4, node 20 at position 8 in L_S.

By a distributed program path of program Q we mean $P = ((L_{t_1}, R_{t_1}), (L_{t_2}, R_{t_2}), ..., (L_{t_m}, R_{t_m}))$, where L_{t_i} is a path in the flow graph of task t_i, and $R_{t_i} =< (A_1, C_1, B_1), ..., (A_k, C_k, B_k) >$ is a sequence of rendezvouses (similar to R–sequence introduced by Tai in (Tai, 1985) associated with task t_i, where each A_j indicates the start of a rendezvous (accept statement identifier) in task t_i, and C_j and B_j denote the caller of this rendezvous, i.e., entry call identifier and its task identifier, respectively. The sequences of rendezvouses are used by the run–time scheduler to ensure the reproducible program execution. Before any rendezvous is allowed to occur by the run–time scheduler, the rendezvous must be the next one specified in the rendezvous sequence. If the next rendezvous in the rendezvous sequence is not called for, then the program (task) is suspended by the scheduler to allow the *proper* rendezvous to occur. In this manner, the sequence of rendezvous is enforced during distributed program execution.

```
Task W1 is
n, p: natural;
begin
1   input (n, p);
2   if (p = 1) then
3       S.put_hi(n);
    else
4       S.put_low(n);
  end if;
end W1
```

```
Task W2 is
n, p: natural;
begin
1   input (n, p);
2   if (p = 1) then
3       S.put_hi(n);
    else
4       S.put_low(n);
  end if;
end W2
```

```
Task W3 is
n, p: natural;
begin
1   input (n, p);
2   if (p = 1) then
3       S.put_hi(n);
    else
4       S.put_low(n);
  end if;
end W3
```

```
Task R1 is
y, p: natural;
begin
1   input (p);
2   if (p = 1) then
3       S.get_hi(y);
    else
4       S.get_low(y);
  end if;
5   compute(y);
6   output(y)
end R1
```

```
Task R2 is
y, p: natural;
begin
1   input (p);
2   if (p = 1) then
3       S.get_hi(y);
    else
4       S.get_low(y);
  end if;
5   compute(y);
6   output(y)
end R2
```

```
Task R3 is
y, p: natural;
begin
1   input (p);
2   if (p = 1) then
3       S.get_hi(y);
    else
4       S.get_low(y);
  end if;
5   compute(y);
6   output(y)
end R3
```

```
Task S is
buf_low, buf_hi: array [1 .. 200] of natural;
head_low, head_hi, tail_low, tail_hi: natural;
begin
1    head_low := 1;
2    tail_low := 1;
3    tail_hi := 1;
4    head_hi := 1;
     loop
        select
5          accept get_low (x: out natural);
6           x := buf_low[tail_low];
7           tail_low := tail_low +1;
8          end get_low;
        or
9          accept put_low(x : in natural);
10          buf_low[head_low] := x;
11          head_low := head_low + 1;
12         end put_low;
        or
13         accept get_hi (x: out natural);
14          x :=buf_hi[tail_hi];
15          tail_hi := tail_hi +1;
16         end get_hi;
        or
17         accept put_hi(x : in natural);
18          buf_hi[head_hi]:= x;
19          head_hi := head_hi + 1;
20         end put_hi;
        end select
     endloop;
     end S
```

Fig. 1. A sample distributed Ada-like program.

Example 1. Suppose that the program of Figure 1 is executed on the following program input:

Task $W1$: $n = 4$, $p = 2$	Task $R2$: $p = 2$
Task $W2$: $n = 2$, $p = 1$	Task $R3$: $p = 1$
Task $W3$: $n = 8$, $p = 2$	

and the following distributed program path P_1 is traversed:

$P_1 = ((L_{W1}, E), (L_{W2}, E), (L_{W3}, E), (L_{R2}, E), (L_{R3}, E), (L_S, R_S))$
where,
$L_{W1} =< 1, 2, 4 >$
$L_{W2} =< 1, 2, 3 >$
$L_{W3} =< 1, 2, 4 >$
$L_{R2} =< 1, 2, 4, 5, 6 >$

$L_{R3} = < 1, 2, 3, 5, 6 >$
$L_S = < 1, 2, 3, 4, 17, 18, 19, 20, 9, 10, 11, 12, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 >$
$R_S = < (17, 3, W2), (9, 4, W3), (5, 4, R2), (9, 4, W1), (13, 3, R3) >$
E is an empty rendezvous sequence.

The above distributed program path is presented in graphical form in Figure 2. In this path, task S accepts, in the first step, entry call 3 from Task $W2$, then entry call 4 from task $W3$, entry call 4 from $R2$, entry call 4 from $W1$, and finally entry call 3 from task $R3$.

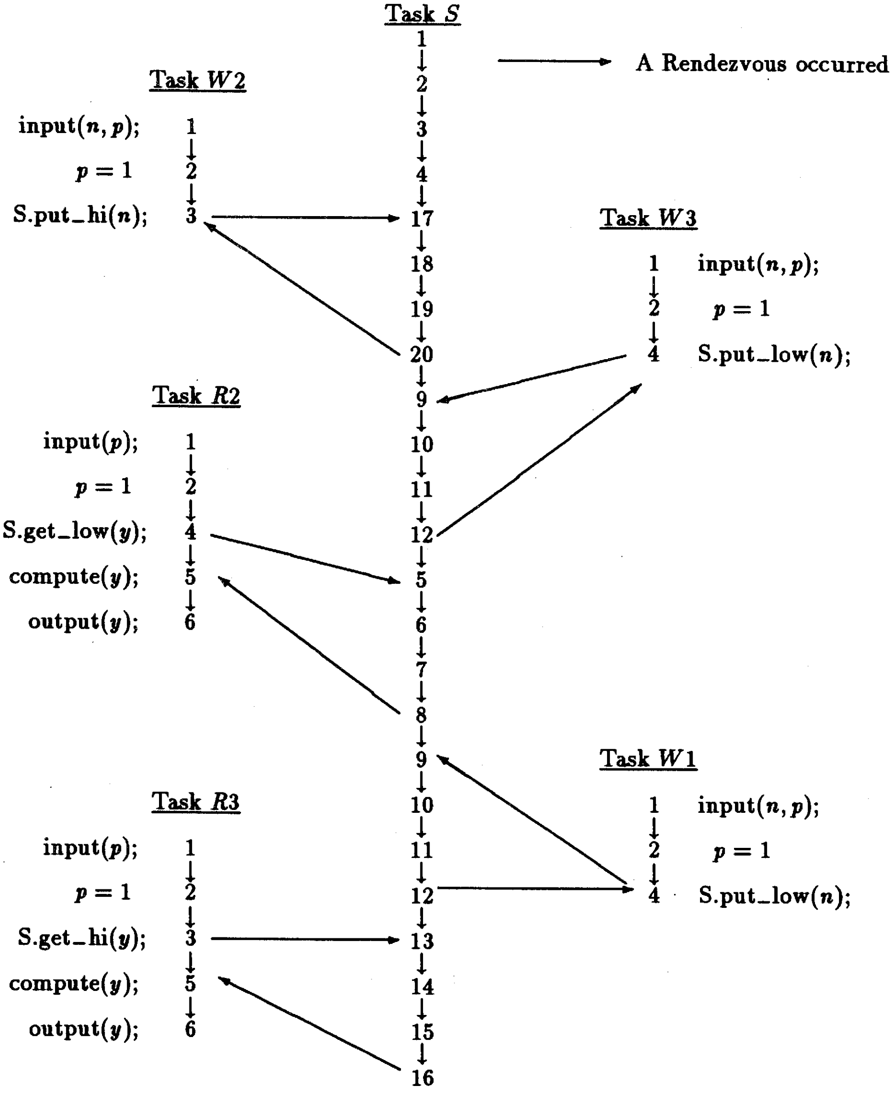

Fig. 2. A sample distributed program path for program of Figure 1.

3. Dynamic Slice for Distributed Programs

Intuitively, a dynamic slice is an executable part of the program whose behavior is identical to that of the original program with respect to a variable (or a set of variables) at some execution position q in task w. Such a *view of interest* is captured by the following:

Definition 1. A *dynamic slicing criterion* of distributed program Q is a quintuple $C = (x, P, w, q, v)$, where P is a distributed program path which has been traversed during execution of program Q on input x, q is a position in path L_w of task w, and v is a variable in task w.

The following is a sample dynamic slicing criterion for the program of Figure 1 which has been executed on the program input x from Example 1: $C_1 = (x, P_1, R3, 5, y)$, where P_1 is a distributed program path shown in Example 1, y is a variable in task $R3$, and $q = 5$ is a position of node 6 in $L_{R3} =< 1, 2, 3, 5, 6 >$. From this slicing criterion we are interested in that part of the distributed program which has affected the value of variable y at position 5 (before execution of node 6) in task $R3$.

Observe that our slicing criterion is defined with respect to a given distributed path traversed on a specific input x, rather than with respect to the set of all possible distributed paths. In addition, for the sake of presentation, we define the dynamic slicing criterion for one variable as opposed to a set of variables (Weiser, 1984).

To formally define the dynamic slice for a distributed program we need some definitions of list operations. Let $L =< n_{k_1}, n_{k_2}, ..., n_{k_p} >$ be a list and q be a position in L. By **Len**(L) we denote the length of L, i.e., the number of elements of L. By **Front**(L, q) we denote a sublist $< n_{k_1}, n_{k_2}, ..., n_{k_q} >$, containing the first q elements of L. By **Del**(L, r), where r is a predicate, we mean a sublist (subpath) obtained from L by deleting from it all elements $L[i]$ that satisfy r. In other words, **Del**(L, r) is the result of an exhaustive application of the delete operation to elements $L[i]$ that satisfy $r(L[i])$.

Definition 2. Let $C = (x, P, w, q, v)$ be a slicing criterion on distributed program Q and P a distributed path of Q which has been traversed on input x. A dynamic slice of distributed program Q on C is any executable program Q' that is obtained from Q by deleting zero or more statements from it and when executed on input x with rendezvous sequences $R'_{t_1}, R'_{t_2}, ..., R'_{t_m}$, which are enforced by the run-time scheduler during execution of Q', produces a distributed path P' for which there exists an execution position q' in path L'_w of task w such that:

1) The value of v before the execution of instruction $L_w[q]$ in Q equals the value of v before the execution of instruction $L'_w[q']$ in Q',

2) $L_w[q] = L'_w[q']$,

3) for each task $t \in T$:
if $t = w$,
then $\mathbf{Front}(L'_w, q') = \mathbf{Del}(\mathbf{Front}(L_w, q), L_w[k] \notin N'_w$ and $1 \leq k \leq q)$
else $L'_t = \mathbf{Del}(L_t, L_t[k] \notin N'_t$ and $1 \leq k \leq \mathbf{Len}(L_t))$,

where, each R'_t is derived from R_t by removing all rendezvous for which the call node and corresponding accept node do not exist in the dynamic slice Q'. More formally:

$$R'_t = \mathbf{Del}(R_t, r(t, R_t[k]) \text{ and } 1 \leq k \leq \mathbf{Len}(R_t))$$

where,

```
function $r(t, (A, B, C))$: boolean;
    return($A \notin N_t$ and $B \notin N_C$)
 end;
```

The following clarifies the conditions in the above definition of dynamic slice. Condition 1 states that the value of variable v at position q in L_w should be equal to the value of v at *equivalent* position q' in L'_w. This *equivalent* position is determined by condition 3. In condition 2, it is required that a node at position q in L_w will appear in the dynamic slice. This is important for programs with loops. Our experience with program slices shows that programmers can be easily confused if that node (statement) is not included in the slice, especially if this node is inside a loop. Condition 3 requires that the original execution of Q be partially preserved during execution of a dynamic slice, i.e., distributed path P' of dynamic slice Q' be equivalent to the original distributed path P from which all nodes not included in Q' are removed. This, for example, will guarantee that if a loop in Q (in task t) iterates k–times, then the same loop, if included in Q' (in task t), also iterates k–times. Observe that the rendezvous sequences, recorded during execution of Q on x, are needed to partially reproduce the behavior of the original execution of Q during execution of dynamic slice Q'. Since some statements (nodes) are removed from the original program, to form a dynamic slice, it is obvious that the rendezvouses which relate to the removed synchronization nodes should also be removed from the rendezvous sequences. These updated sequences are used, by the run–time scheduler, during execution of the dynamic slice Q' to partially reproduce behavior of the original execution of Q on program input x.

In the next section we introduce the basic concepts of dynamic influence that are concerned with control and data flow along the executed distributed path. These concepts are used in the method for finding dynamic slices.

4. Dynamic Influence Concepts

Let $L_t =< n_{k_1}, n_{k_2}, ..., n_{k_q} >$ be a path in task t that is traversed on a program input x. A definition of variable v in L_t is a node n_{k_p} which assigns a value to that variable. In Ada–like programs a **definition** of variable v can be: (1) an assignment statement, (2) an input statement, (3) accept statement in which v is declared as an input parameter, e.g., **accept put_hi** (x: in natural) in task S of Figure 1 is a definition of x, and (4) an entry call statement in which v appears as an actual parameter and corresponding formal parameter of v is declared as an output parameter, e.g., **S.get_hi**(y) in task $R3$ of Figure 1 is a definition of y.

A **use** of variable v in L_t is a node n_{k_p} in which this variable is referenced. In particular, a use can be: (1) an assignment statement, (2) an output statement, (3) an entry call statement in which v appears as an actual parameter and corresponding formal parameter of v is declared as an input parameter, e.g., **S.put_hi**(n) in Task $W2$ is a use of n, and (4) a return statement from the rendezvous block in which v is declared as an output parameter (in the corresponding accept statement), e.g., **end get_hi** in Task S is a use of x.

Let $U(n_{k_p})$ be the set of variables whose values are used in n_{k_p} and $D(n_{k_p})$ be the set of variables whose values are defined in n_{k_p}. In what follows we introduce the concept of influence between two nodes n_{k_p} and n_{k_i} in L_t.

> We say that n_{k_p} has *data influence* on n_{k_i} by variable v, $p < i$, iff (1) $v \in U(n_{k_i})$, (2) $v \in D(n_{k_p})$, and (3) for all j, $p < j < i$, $v \notin D(n_{k_j})$.

This influence describes a situation where one node assigns a value to an item of data and the other node uses that value. For instance, in the traversed subpath P_1 of Example 1 (see Figure 3) in Task S node 18 has data influence on node 14 by variable buf_hi[1], assignment statement 14 has data influence on end–get–statement 16 by x, and accept–put–statement 17 has data influence on assignment statement 18 by x.

Notice that in the static approach (Ferrante *et al.*, 1987; Horowitz *et al.*, 1990) an entire array is usually treated as a single variable. This is due to the difficulty of determining the values of the array subscripts. In contrast, dynamic approach is oriented towards only those array elements that have actually been manipulated during the execution, e.g., buf_hi[1] at node 14 and 18 in Task S.

The next type of influence is referred to as a control influence. This influence captures the dependence between test nodes and nodes which can be chosen to execute or not execute by these test nodes. To define the control influence, we first introduce the notion of the scope of influence for the if– and while–statements:

a) *if* Z *then* B_1 *else* B_2 *end if*

X is in the scope of control influence of Z iff X appears in B_1 or B_2.

b) *while* Z *do* B *end loop*

X is in the scope of control influence of Z iff X appears in B.

The scope of control influence captures the influence between test nodes and nodes which can be chosen to execute or not execute by these test nodes. Let $L_t = < n_{k_1}, n_{k_2}, ..., n_{k_q} >$ be a path in task t that is traversed on a program input x.

> We say that n_{k_p} has control influence on n_{k_r} in L_t, $p < r$, iff for all j, $p < j \leq r, n_{k_j}$ is in the scope of control influence of n_{k_p}.

For example, node 2 in L_{R3} of Example 1 has control influence on node 3 but does not have influence on nodes 5 and 6. Note, the concept for control influence can easily be extended for the Ada **select** and **loop** statements in the similar way.

The influences described above exist only between nodes in the path of one task. Now we define an influence which can exist only between synchronization nodes. This influence captures the situation where one synchronization node in task t sends a data item and another synchronization node in task b receives this data item. Let $L_t = < n_{k_1}, n_{k_2}, ..., n_{k_q} >$ be a path in task t and $L_b = < m_{j_1}, m_{j_2}, ..., m_{j_l} >$ be a path in task b.

> We say that node n_{k_p} in L_t has *communication influence* on node m_{j_i} in L_b iff
> (1) n_{k_p} is an entry call statement, (2) m_{j_i} is an accept statement, (3) the rendezvous has occurred between n_{k_p} and m_{j_i}, and (4) there exists an input parameter in m_{j_i},
> or
> (1) n_{k_p} is a return statement from the rendezvous block, (2) m_{j_i} is an entry call statement, (3) the return from the rendezvous has occurred between n_{k_p} and m_{j_i}, and (4) there exists an output parameter in m_{j_i}.

For example, in the traversed subpath P_1 of Example 1, node 3 (entry call statement) in Task $W2$ has communication influence on node 17 (accept statement) in Task S (the value of variable n is sent from Task $W2$ to accept statement of Task S), and node 16 (return statement from get_hi) in Task S has communication influence on node 3 (entry call statement) in Task $R3$; in this case, the value of parameter x in Task S is sent to the entry call statement in Task $R3$.

The influences between nodes in the distributed program path can be represented graphically as an *influence network*, where each link between nodes represents data, control, or communication influence between them. The example of the influence subnetwork is presented in Figure 3. The influences described above capture only direct influences between nodes in the distributed program path. The influence network can be used to determine indirect influences between different nodes by *chains* of directed influences. This is captured by the following:

> We say that node n_{k_p} in L_t has influence on node m_{j_i} in L_b iff there exists a path in the influence network from node n_{k_p} in L_t to node m_{j_i} in L_b.

For example, it is easy to see from the dependence subnetwork of Figure 3 that node 2 in task $W2$ has influence on node 6 in task $R3$.

5. Finding Dynamic Slices

There can be many different dynamic slices for a given program and a slicing criterion; there is always at least one such slice, the entire program itself. Naturally, we are interested in a statement–minimal slice, i.e., one that has the minimum

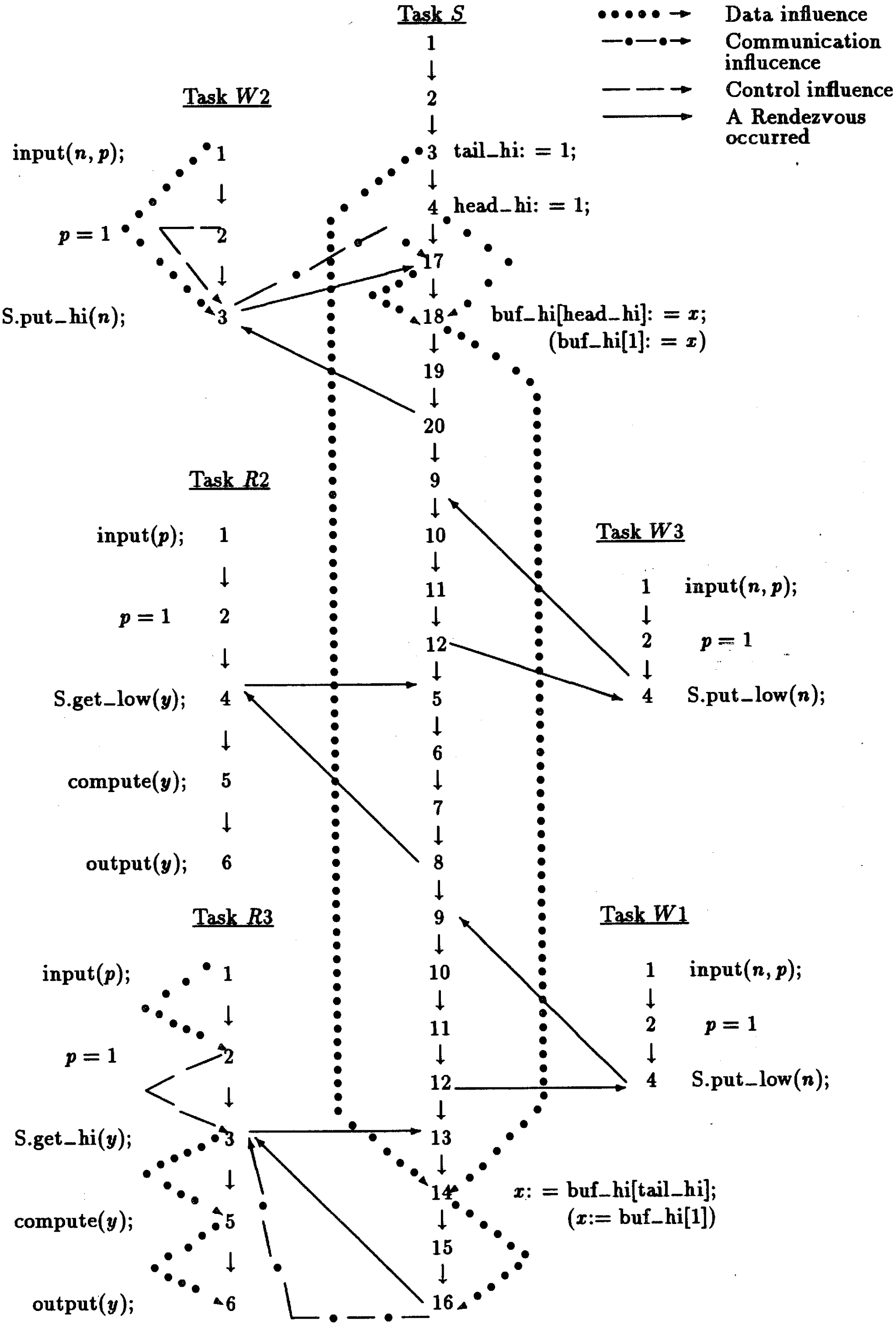

Fig. 3. Influence subnetwork for the distributed path P_1 from Example 1.

number of statements. The method of dynamic slice derivation presented in this paper is based on the dynamic influence concepts and leads to conservative slices, guaranteed to have the slice properties but with potentially too many statements.

Given a slicing criterion $C = (\boldsymbol{x}, P, w, q, v)$, a dynamic slice Q' contains only those statements (nodes) of program Q that influence the variable v at q in task w. The derivation of the slice is done in two steps: first all nodes in path P that influence variable v at q in task w are marked and then a dynamic slice is reconstructed from the marked nodes in P. In addition, the updated rendezvous sequences for the dynamic slice are derived from the original rendezvous sequences and the marked nodes in P.

The following is a sequential algorithm for marking nodes in path P:

```
1.  Set all nodes in $P$ as unmarked and not visited
2.  Mark the last definition of $v$ at position $q$ in task $w$
3.  Mark nodes which have control influence on $L_W[q]$
4.  WHILE there exists a marked but not visited node in path $P$ DO
5.      Select not visited but marked node $x$
6.      Set node $x$ as visited
7.      Mark all unmarked nodes which have influence on node $x$
8.      Mark all unmarked multiple occurrences of node $x$
9.  ENDWHILE
```

In this algorithm, after setting all nodes as unmarked and not visited, the last definition of variable v at position q is found and marked, where the last definition is defined as follows:

> We say that n_{k_p} is a *last definition* of variable v at position q in L_W, $p < q$, iff (1) $v \in D(n_{k_p})$, and (2) for all j, $p < j < q$, $v \notin D(n_{k_j})$.

The last definition of variable v at q in task w is a unique node which has last assigned a value to v when position q is reached in L_W.

In step 3, all nodes which have control influence on node $L_W[q]$ are marked. This is because of Condition 2 of Definition 2, in which it is required that a node at position q in L_W appear in the dynamic slice. In the next steps of the algorithm (loop 4–9), not visited but marked node x in P is selected, and then all nodes which have influence on x are marked (based on the influence network). Step 8 requires special explanation: When influence concepts defined above are used to derive a dynamic slice, they fail to guarantee that Condition 3 in Definition 2 is satisfied (Korel and Laski, 1988). Recall that according to the latter, it is required that the number of loop iterations be preserved if the loop is present in a slice. Towards that goal, we mark all occurrences of a node which has been executed more than once in program Q. The process of marking nodes in loop 4–9 is repeated until all marked nodes are visited.

Given marked path P, the dynamic slice Q' is found in a straightforward way. First, remove from the original program Q all statements (nodes) which are not marked in P. Note that because of Condition 2 of Definition 2, the statement corresponding to node $L_W[q]$ in task w is not removed even if this node is not marked. Second, to ensure syntactical correctness all necessary declarations are to be included.

In order to guarantee the partially reproducible execution of the dynamic slice with respect to the execution of the original program, the updated rendezvous sequences $R'_{t_1}, R'_{t_2}, ..., R'_{t_m}$ must be derived from the original rendezvous sequences $R_{t_1}, R_{t_2}, ..., R_{t_m}$. These updated sequences are used by the run–time scheduler during execution of the dynamic slice. The updated rendezvous sequences are derived in the following way: For each rendezvous sequence $R_{t_i} =< (A_1, C_1, B_1), ..., (A_k, C_k, B_k) >$ remove all elements (A_j, C_j, B_j) from R_{t_i} for which A_j is not marked in L_{t_i}.

Example 2. Consider the distributed program of Figure 1 and its path P_1 from Example 1. For the slicing criterion: $C_1 = (x, P_1, R3, 6, y)$, where x is a program input presented in Example 1, the following are the marked nodes in path P_1:
L_{W2} : 1, 2, 3
L_{R3} : 1, 2, 3, 5, 6
L_S : 3, 4, 17, 18, 20, 13, 14, 16.
Nodes in L_{W1}, L_{W3}, and L_{R2} are not marked.

The corresponding dynamic slice is derived from those marked nodes in P_1 and is presented in Figure 4. The updated rendezvous sequence $R_S = < (17, 3, W2), (13, 3, R3) >$ is also easily derived from the original rendezvous sequence R_S based on the marked nodes in P_1; this rendezvous sequence must be used by the run–time scheduler to partially reproduce behavior of the original execution of the program of Figure 1 on the same input, leading to exactly the same value of variable y in task $R3$ at node 6 (the distributed program path for the dynamic slice is presented in Figure 5). From this dynamic slice it is easy to see a significant reduction of the original program (around 30% of the original program).

```
Task W2 is
n,p: natural;
begin
1   input (n,p);
2   if (p = 1) then
3       S.put_hi(n);
    else
    end if;
end W2
```

```
Task R3 is
y,p: natural;
begin
1   input (p);
2   if (p = 1) then
3       S.get_hi(y);
    else
    end if;
5   compute(y);
6   output(y)
end R3
```

```
Task S is
buf_hi: array [1 .. 200] of natural;
head_hi, tail_hi: natural;
begin
 3    tail_hi := 1;
 4    head_hi := 1;
      loop
          select
13            accept get_hi (x: out natural);
14             x :=buf_hi[tail_hi];
16            end get_hi;
          or
17            accept put_hi(x : in natural);
18             buf_hi[head_hi]:= x;
20            end put_hi;
          end select
      endloop;
      end S
```

Fig. 4. A dynamic program slice of Example 2.

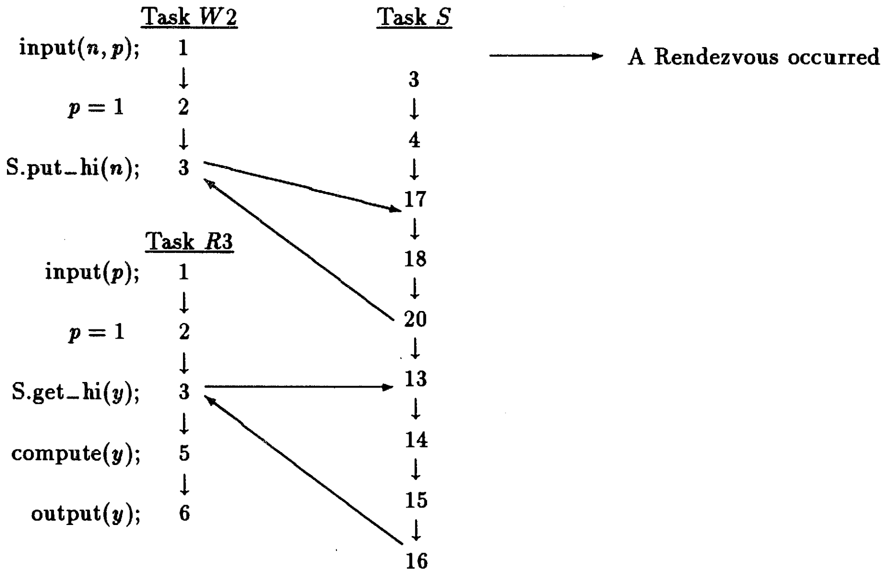

Fig. 5. The distributed program path for the dynamic program slice of Figure 4.

In the sequential algorithm, it is assumed that all recorded traces are sent to one computer site and the dynamic slice derivation is done on this site. This can work fine for relatively short traces. However, for very long traces this approach can be prohibitive. For this reason, we have developed a distributed version of the algorithm for finding dynamic slices. Now we present a distributed version of the algorithm.

The distributed algorithm consists of a set of identical algorithms located on each computer site in the network. The algorithm, which is presented in Figure 6, is responsible for the influence analysis and node marking for the task(s) residing on this site (for the sake of simplicity, we assume that only one task resides on each site). The distributed analysis starts at the site of task w where the incorrect value of variable v has been observed (notice that lines 2 and 3 of the algorithm of Figure 6 are executed at this site). In the distributed algorithm the control and data influences are handled locally on each computer site in the exactly the same manner as in the sequential algorithm. The major difference is with communication

```
1.   Set all nodes in $L_t$ as unmarked and not visited
/*  The following two statements are executed only on the site of task $w$ */
2.       Mark the last definition of $v$ at position $q$ in task $w$
3.       Mark nodes which have control influence on $L_W[q]$
4.       WHILE not global termination DO
5.           WHILE (message queue is not empty) or
6.                     (there exists a marked but not visited node in path $L_t$) DO
7.               Remove a message from the queue
8.               Mark a node in $L_t$ specified in the message
9.               WHILE there exists a marked but not visited node in path $L_t$ DO
10.                  Select not visited but marked node $x$
11.                  Set node $x$ as visited
12.                  Mark all unmarked multiple occurrences of node $x$ in $L_t$
13.                  IF $x$ is a communication node
14.                  THEN
15.                      IF $x$ is an accept node
16.                      THEN
17.                          Determine task_id $t_i$ and call_entry_id from R-sequence
                             of task $t$ for $x$
18.                          Send a message to task $t_i$ to mark the corresponding entry
                             call node in $L_{t_i}$
19.                      ELSE /* $x$ is a call_entry to task $t_j$ */
20.                          Send a message to task $t_j$ to mark the corresponding
                             end_accept node in $L_{t_j}$
21.                      ENDIF
22.                  ELSE
23.                      Mark all unmarked nodes in $L_t$ which have data or control
                         influence on node $x$
24.                  ENDIF
25.              ENDWHILE
26.          ENDWHILE
27.          Exchange messages to determine global termination
28.  ENDWHILE
```

Fig. 6. A distributed algorithm for finding dynamic program slices.

influences. If the current node x is a communication node then (1) a task is identified with whom the rendezvous has occurred and then (2) a request is sent to the slicing algorithm of that task in order to mark the corresponding communication node in this task path. The incoming messages are stored on the message queue by the special process which is only responsible for accepting messages from other sites. The algorithm removes in line 7 the incoming messages from the message queue, and marks the nodes specified in those messages. For example, consider node 3 of Figure 3 in task $R3$. The algorithm responsible for task $R3$ will send a message to the corresponding algorithm of task S to mark the end node of accept get_hi statement in path of task S. Similarly, for node 17 in task S, the algorithm of task S will determine, from the rendezvous sequence, the corresponding task (in our case task $W2$ and node 3) and will send a message to the algorithm of task $W2$ to mark node 3 in the path of $W2$. After the marking process is completed and the agreement about the global termination is reached, the corresponding slices for each task are derived from the marked nodes in task paths.

6. Conclusions

In this paper, we have presented the concepts of dynamic slicing for distributed programs. The idea of dynamic slicing has been first described in (Korel and Laski, 1988) for sequential programs. This paper extends this idea for distributed programs which introduce several problems which do not exist in sequential programs. The main advantage of dynamic slicing is that the size of a slice can be significantly reduced by taking into account a particular program execution, rather than all possible executions of a distributed program. The motivation for dynamic slicing is its use in program debugging because it provides finer localization information by reducing the searching space for faults in distributed programs. Consequently, it is easier for programmers to localize the causes of incorrectness. Ideally, the dynamic slicing tool should be a part of distributed debugger, e.g., (Bates, 1988; Cooper, 1987; Elshoff, 1988; Choi *et al.*, 1991), in order to make debugging more efficient. Information which is recorded by distributed debuggers could be used for the purpose of slicing.

We now highlight some directions for the future research. One direction is to incorporate static dependence analysis (e.g., Ferrante *et al.*, 1987; Horowitz *et al.*, 1990; Korel, 1987) into dynamic slicing in order to reduce the amount of recorded information during execution of distributed programs (this can be of a great importance for very long program executions). Static analysis could significantly enhance the derivation of dynamic slices. The second direction of the research is to use dynamic slicing in the process of test data generation for distributed programs. In (Korel, 1990a; Korel *et al.*, 1991) a novel approach of automated test data generation has been proposed. This approach is based on actual program execution in order to derive required test data. Dynamic slicing should significantly improve the efficiency of this method by requiring that only a part of a distributed program, i.e., a dynamic slice, should be executed rather than the whole program.

References

Agrawal H. and Horgan J. (1990): *Dynamic program slicing.*– SIGPLAN Notices, v.25, No.6, pp.246–256.

Bates P. (1988): *Distributed debugging tools for heterogeneous distributed systems.*– 8th Int. Conf. *Distributed Computing Systems*, San Jose, CA, USA, pp.308–315.

Cooper R. (1987): *Pilgrim: A debugger for distributed systems.*– 7th Int. Conf. *Distributed Computing Systems*, Berlin, West Germany, pp.458–465.

Elshoff I. (1988): *A distributed debugger for Amoeba.*– Proc. ACM SIGPLAN and SIGOPS Workshop *Parallel and Distributed Debugging*, Madison, WI, USA, pp.1–10.

Chan F. and Chen T. (1987): *AIDA – A dynamic data flow anomaly detection system for Pascal programs.*– Software–Practice and Experience, v.17, No.3, pp.227–239.

Choi J., Miller B. and Netzer R. (1991): *Techniques for debugging parallel programs with flowback analysis.*– Trans. Programming Languages and Systems, v.13, No.4, pp.491–530.

Ferrante J., Ottenstein K. and Warren J. (1987): *The program dependence graph and its use in optimization.*– ACM Trans. Program. Lang. & Systems, v.9, No.3, pp.319–349.

Horowitz S., Reps T. and Binkley D. (1990): *Interprocedural slicing using dependence graphs.*– Trans. Programming Languages and Systems, v.12, No.1, pp.26–60.

Jones C. (1980): *Software Development: A Rigorous Approach.*– New York: Prentice–Hall.

Korel B. (1987): *The program dependence graph in static program testing.*– Information Processing Letters, v.24, No.1, pp.103–108

Korel B. and Laski J. (1988): *Dynamic program slicing.*– Information Processing Letters, v.29, No.3, pp.155–163.

Korel B. (1990a): *Automated software test data generation.*– IEEE Trans. Software Engineering, v.16, No.8, pp.870–879.

Korel B. and Laski J. (1990b): *Dynamic slicing in computer programs.*– Journal of Systems and Software, v.13, No.3, pp.187–195.

Korel B., Wedde H. and Ferguson R. (1991): *Automated Test Data Generation for Distributed Software.*– *COMPSAC-91* Conf., Tokyo, Japan, pp.680–685.

Tai K.C. (1985): *Reproducible Testing of Concurrent Ada Programs.*– SoftFair: the 2nd Conf. *Software Development Tools, Techniques, and Aiternatives*, San Francisco, USA, pp.114–121.

Tai K.C., Carver R. and Obaid E. (1991): *Debugging concurrent ada programs by deterministic execution.*– IEEE Trans. Software Enginering, v.17, No.1, pp.45–63.

Weiser M (1984): *Program slicing.*– IEEE Trans. Software Engineering, v.10, No.4, pp.352–357.

Received June 8, 1992
Revised September 20, 1992