A Heuristic Approach to Path Selection Problem in Concurrent Program Testing

Su-Yu Hsu and Chyan-Goei Chung

Dept. of Computer Science and Information Engineering, National Chiao Tung University
Hsinchu, Taiwan 30050, R. O. C.

Abstract

Repeated execution of a concurrent program with the same input may produce different sequence of rendezvous, thus making different output. The output of a concurrent program executed is determined by both the statements traversed by each task and the sequence of rendezvous. The traversed statements of each task constitute a task path, all task paths in an execution instance constitute a concurrent path.

This paper proposes a new approach to generate all feasible concurrent paths from task path sets of a concurrent program. All possible rendezvous sequences of each feasible concurrent path are also generated. A heuristic approach is also proposed to select the suitable concurrent path set which satisfies node or branch coverage criterion.

1: Introduction

Structural testing is a widely used program validation strategy. Path testing [1] which concerns control structures is an approach to structural testing. The control structure of a program is represented by a graph. From the program graph, all feasible paths can be automatically generated. According to a certain coverage criterion, an optimal path set can then be selected and its test data can be determined. Repeated execution of a sequential program with the same test data will traverse the same path and produce the same outcome.

A concurrent program consists of a fixed set of tasks, where the tasks are simultaneously executed. Repeated execution of a concurrent program with the same data may produce different outcomes due to the variant rendezvous sequences. Actually, the relative progress of task execution and ready alternatives of "select" statements [2] used in the concurrent program affect the rendezvous sequence. This is referred to as a "reproducible testing problem" [3]. Consequently, the methods in path testing for a sequential program can not be applied to a concurrent program directly.

If the test data include both the input data and a rendezvous sequence, the tested concurrent program will be reproducible. According to the new definition of test data, we need a new path testing methodology for concurrent programs. Most previous research on concurrent program testing has focused on how to control or replay test execution [4] [5] and how to perform static analysis [6]. However, Yang [7] first proposed a path testing methodology for concurrent program testing, which is extended from that for sequential program testing. Based on his model, an execution of a concurrent program is viewed as involving a concurrent path (which is comprised of one path for each task), and the tasks' synchronization is modeled as a concurrent route to traverse the concurrent path involved in the execution. Hence, the reproducible test of a concurrent program can be achieved by an input and a concurrent route. The methodology includes five steps: (1) select a set of concurrent paths, (2) generate the concurrent routes along each selected concurrent path, (3) generate data for each concurrent route along the selected concurrent paths, and (4) execute the program with the data generated.

Yang also proposed that all feasible concurrent paths can be generated by reachability analysis technique [8], and that all concurrent routes of a feasible concurrent path can be determined by all possible synchronization sequences of rendezvous statements of all tasks. His approach is very time- and space-consuming because of the existence of many states resulting from infeasible concurrent paths and infeasible synchronizations.

A concurrent path is a member of the cartesian product of task path sets; the paths of each task path set can be easily generated by conventional methods [1]. A feasible concurrent path guarantees that all tasks will be terminated normally. From the number of entry call statements for each entry name, number of accept statements for the corresponding entry names, and the generated rendezvous sequences, we can determine whether a concurrent path is feasible or not. Based on this idea, a *task-path-based approach* to generating all feasible concurrent paths and their corresponding rendezvous sequences is proposed in this paper. Since generating the input data is dependent on the program semantics, it is not a concern here.

For structural testing, the test elements must be identified first from the program graph. Node coverage and branch coverage are the two well known kinds of test criteria. To minimize test effort, the optimum path set

0-8186-2755-7/92 $03.00 © 1992 IEEE

should be selected to meet coverage criteria. In this paper, we also propose a *greatest coverage increment first approach* to feasible concurrent path selection for node and branch coverage.

Our approach to generating feasible concurrent paths has the merit of the generated feasible concurrent paths associated with rendezvous sequences serving as the basis for both path selection and reproducible test execution [3]. The approach can be applied to concurrent programs in a language by remote–procedure–call–like synchronization, typically Ada. The proposed feasible concurrent path selection method may not result in an optimal solution with respect to testing cost, but the approach is simple enough to get the approximate solution with the benefit of efficiency. In this paper, we make the following assumptions. The concurrent program is written in Ada and run in a distributed environment. All of the tasks are initiated on entry to the main procedure. There are no shared variables among tasks. No entry name is related to interrupt. The entry names are statically known.

The remainder of this paper is organized as follows. In Section 2, the proposed task–path–based approach, the concept of how to construct the set of concurrent paths and their associated rendezvous paths are described. Section 3 addresses the method of determining the feasibility of a concurrent path. We apply our feasibility determination method to an example in Section 4. The heuristic approach to feasible concurrent path selection is described in Section 5. A conclusion and future research directions are given in Section 6.

2: A task–path–based approach to generating feasible concurrent paths

2.1: Basics

Before we describe the approach to generating all feasible concurrent paths of a concurrent program, some fundamental terminologies are defined as follows. A *concurrent program* consists of a fixed set of tasks. It is denoted by $P = \{T_i \mid i = 1, 2, ..., n\}$, where n is the number of tasks of the concurrent program. When a concurrent program is executed and normally terminated, each task will traverse a sequence of statements. The sequence of statements is called a *task path*. A *concurrent path* is a collection of task paths such that there is exactly one task path per task in the collection. It is represented by an ordered n–tuple $(C_{1(j_{[1]})}, C_{2(j_{[2]})}, ..., C_{i(j_{[i]})}, ..., C_{n(j_{[n]})})$, where $C_{i(j_{[i]})}$ is the $j_{[i]}$th task path of task T_i, $j_{[i]} \in \{1, 2, \ldots, r_i\}$ and r_i is the number of task paths of task T_i. The execution of a concurrent program will traverse a concurrent path. The execution terminates normally only when the tasks are all finished; the traversed concurrent path is called a *feasible concurrent path*.

In case that a concurrent path has no rendezvous statements, it must be feasible. Concurrent paths of this kind are generated without further checking. For a concurrent path with rendezvous statements, which is traversed by a concurrent program with normal termination, the rendezvouses among tasks will follow a sequential order. A *rendezvous pair* is a pair of an entry call and an accept statement with the same entry name. It is represented by (entry call statement number, accept statement number). A *rendezvous sequence* is a sequence of rendezvous pairs such that the sequence preserves the original rendezvous statement order in a concurrent path. It is represented by [rendezvous pair *1*, rendezvous pair *2*, ...].

A feasible concurrent path may have more than one rendezvous sequence. Hence, the testing tool should have the capability of forcing the concurrent program to follow a desired concurrent path and a desired rendezvous sequence. To enumerate all rendezvous sequences of a concurrent path, we need to examine the rendezvous statements only. The following terms are thereafter defined. The task paths with nonrendezvous statements being discarded are called *task rendezvous paths*. A *rendezvous path* is a collection of task rendezvous paths such that there is exactly one task rendezvous path per task in the collection. It can be represented by an ordered n–tuple $(R_{1(j_{[1]})}, R_{2(j_{[2]})}, ..., R_{i(j_{[i]})}, ..., R_{n(j_{[n]})})$, where $R_{i(j_{[i]})}$ is the rendezvous–related portion of the task path $C_{i(j_{[i]})}$.

2.2: The approach

A concurrent program contains several tasks. Every task can be treated as a sequential program. The technique of flowgraph construction for a sequential program is well established [1]. Here we extend the flowgraph model to concurrent programs. Note that a select statement is a kind of selection (or called branch) structure. A concurrent program is modeled by a set of *task flowgraphs*, each of which exposes the control flow of one task.

When a concurrent program is executed, there must be exactly one task path for each task being traversed. Therefore, we have to generate all possible task paths for every task flowgraph first. To overcome the problem of repetition resolution in expanding paths, we limit the number of repetitions up to a constant number as most researchers do [9]. There are several techniques to expand paths from a flowgraph [9]. We adopt the *node reduction method*. This method selects any node except the task start node and the task termination node for removal, then replaces it by a set of edges which correspond to all of the ways you can form the cartesian product of the set of incoming edges of the node with the set of outgoing edges of the node. At last, the selected node is recorded associated with its substituting edges. The above steps are repeated until all nodes except the task start node and the task termination node are removed. The list of recorded nodes in every resulting path, excluding the selection nodes and the iteration nodes comprises a task path. Consequently, we obtain a *task path set*, { C_{i1} , C_{i2} , ..., C_{ir_i} } where r_i is the number of task paths of task T_i. We use a table *TASK_PATH*$_i$ to represent the task paths of task T_i. The task path sets of a concurrent program form the *task path graph* and their cartesian product forms the *concurrent path set*.

When the nonrendezvous statements except the task start node and the task termination node in each task path of task T_i are discarded, it results in a *task rendezvous path set*, { R_{i1} , R_{i2} , ..., R_{ir_i} }. A task rendezvous path contains a list of zero or more rendezvous statements. The task rendezvous path sets of all tasks form the *task rendezvous path graph* and their cartesian product forms the *rendezvous path set* for the concurrent program.

Since a concurrent program can be decomposed into a set of rendezvous paths from task path sets, the set of feasible concurrent paths for a program can be obtained by repeatedly determining the feasibility of each rendezvous path. Their associated rendezvous sequences, if exist, are also generated.

3: Feasibility determination

Since the feasibility of a concurrent path is up to the normal termination of each task, the following two conditions must be satisfied such that the rendezvous statements in each task path can be successfully fulfilled. (1) For any entry name, the numbers of entry call statements and of accept statements must be equal, which is called the *basic rule*. (2) The rendezvous sequence involved in the execution of a feasible concurrent path must follow the original rendezvous statement order in the task paths of the concurrent path, which is called the *task-path-constraints*. Hence, the feasibility of each concurrent path can be determined by the following steps. First, we check whether the concurrent path satisfies condition (1). Then, we check whether the rendezvous statements in the concurrent path can lead to a rendezvous sequence satisfying condition (2). If both conditions are satisfied, the concurrent path is feasible.

3.1: Static and dynamic entries

Before we describe the method of arranging rendezvous sequences for rendezvous statements of a concurrent path, some terms need to be defined. An entry name called by only one task, regardless of the number of entry calls, is called a *static entry*. A sequence of rendezvous pairs that satisfies the task-path-constraints for all of the rendezvous statements related to a static entry is called a *static valid sequence*. A sequence of rendezvous pairs that satisfies the task-path-constraints for all of the rendezvous statements related to all of the static entries is called a *static rendezvous sequence*. The pairs in the static rendezvous sequences are called *static rendezvous pairs*. An entry name, called by more than one task, is called a *dynamic entry*. A sequence of rendezvous pairs that satisfies the task-path-constraints for all of the rendezvous statements related to a dynamic entry is called a *dynamic valid sequence*. A sequence of rendezvous pairs that satisfies the task-path-constraints for all of the rendezvous statements related to all of the dynamic entries is called a *dynamic rendezvous sequence*. The pairs in the dynamic rendezvous sequences are called *dynamic rendezvous pairs*.

Thus, the procedure to arrange rendezvous sequences for a concurrent path is divided into three steps. First, we arrange static rendezvous sequences. Second, we arrange dynamic rendezvous sequences. Third, we merge static rendezvous sequences with dynamic rendezvous sequences. The methods are described in Sections 3.2, 3.3, and 3.4 respectively.

3.2: Static rendezvous sequences

To arrange static rendezvous sequences, we first obtain a unique *static valid sequence* for each static entry. Supposing that there are s static entries, we have s static valid sequences. The static rendezvous sequences can be obtained by merging all of the static valid sequences. Each of the obtained static rendezvous sequences must satisfy two conditions: (1) it preserves the rendezvous sequence of the involving static valid sequences, and (2) it preserves the statement sequence of the original task paths. Therefore, we construct a *dependency graph* for obtaining the static rendezvous sequences. In the graph, a node is a rendezvous pair in the static valid sequences. We construct the outlinks for each node as follows. We find the smallest statement number which is greater than the entry call statement number, if one exists. The found statement, of course, must occur in the same task with the entry call statement. Thus, we obtain an outlink to the rendezvous pair containing the found statement. The other outlink can also be similarly obtained for the accept statement, if one exists. Thus, each node has two successors at most. After constructing the dependency graph, we apply topological sorting to it. In the method, we select a rendezvous pair without parents first. Then, we delete the selected pair and its outlinks. The process is repeated until one of the following termination conditions occurs. (*1*) No such pair can be selected. (*2*) The graph is completely traversed.

When the first termination condition occurs, the graph has a cycle. It means that the static valid sequences can not lead to a static rendezvous sequence. Hence, deadlock may occur. The candidate concurrent path is thus infeasible. When the second termination condition occurs, only one sequence is generated. However, we are required to generate all possible sequences. It is quite different from the conventional topological sorting method which generates one possible sequence. Therefore, we use a tree to represent the process of generating sequences. Initially, a dummy node is used as the root (at level *0*). We have *m* branches in a node, when there are *m* nodes found to be without parents. The number of levels is the number of rendezvous pairs in the static valid sequences. A path from the root to a leaf defines a static rendezvous sequence. By traversing all of the paths in the tree, the static rendezvous sequences are obtained.

3.3: Dynamic rendezvous sequences

To arrange dynamic rendezvous sequences, we first obtain a *dynamic valid sequence set* for each dynamic entry. Second, we merge the dynamic valid sequences in a member of the cartesian product of the dynamic valid sequence sets.

Dynamic valid sequences: In a concurrent path, if two entry call statements that occur in different task paths involve the same entry name, the accept statement of the entry name must occur twice in another task path. Otherwise, deadlock occurs. Thus, there are two possible ways to arrange their rendezvous sequences.

For a dynamic entry, suppose that there are m accept statements in a task path, and k calling tasks each of which has n_j ($1 \le j \le k$) entry call statements. Hence, $\sum_{j=1}^{k} n_j$ is equal to m. Since the accept statements must occur in a certain task path according to Ada syntax rules, we are required to assign these accept statements to all entry call statements–some of which occur in different task paths. This is fulfilled by assigning proper accept statements to each of the calling tasks. For the first calling task, there are m accept statements to be matched with its n_1 entry call statements. Thus, there are $C(m, n_1)$ possible arrangements. For the second calling task, there remain $(m-n_1)$ accept statements to be matched with its n_2 entry call statements. Thus, there are $C(m-n_1, n_2)$ arrangements. In general, for the jth calling task, there are $C(m - \sum_{i=1}^{j} n_{i-1},\ n_j)$ arrangements. Therefore, the number of dynamic valid sequences for a dynamic entry is

$$\prod_{j=1}^{k} C(m - \sum_{i=1}^{j} n_{i-1},\ n_j) = m!/(n_1!\ n_2!\ \dots\ n_k!).$$

To represent all dynamic valid sequences of a dynamic entry, we construct an *assignment tree* of k levels, where k is as defined before. The root is at level *0*. We represent the possible assignments for the jth calling task at level j. Thus, each node at level $(j-1)$ has $C(m - \sum_{i=1}^{j} n_{i-1},\ n_j)$ successors. Any of the successors, say node s, represents an assignment of n_j accept statements chosen from a candidate set with $(m - \sum_{i=1}^{j} n_{i-1})$ accept statements to the jth calling task. The candidate set consists of all of the accept statements of the dynamic entry excluding those assigned in the ancestor nodes of the node s. The root is a dummy node. Any assignment is stored in an array with size n_j; and the array is sorted in the ascending order of the accept statement numbers. That is, the accept statement with number l ($1 \le l \le m$), in the ith element of the array is assigned to the ith call statement of the jth calling task. Each of the paths from level *0* to level k defines a dynamic valid sequence. Therefore, we identify the dynamic valid sequences by traversing all of the paths in the assignment tree from the root to the leaves.

Dynamic rendezvous sequences: Supposing that there are d dynamic entries, we have d sets of dynamic valid sequences, V_1, V_2, ..., and V_d. The dynamic rendezvous sequences can be obtained by merging the dynamic valid sequences in each member of the cartesian product of the dynamic valid sequence sets. Each of the obtained dynamic rendezvous sequences must satisfy two conditions: (1) it preserves the rendezvous sequence of the involving dynamic valid sequences, and (2) it preserves the statement sequence of the original task paths. Therefore, the merging process for obtaining static rendezvous sequences can also be applied to obtain the dynamic rendezvous sequences similarly. When the first termination condition occurs, the dependency graph has a cycle. It means that the dynamic valid sequences in the candidate member can not lead to a dynamic rendezvous sequence. Hence, it is discarded. Otherwise, the result is represented in a tree with x levels, where x is the number of rendezvous pairs in the candidate member. A path from the root to a leaf defines a dynamic rendezvous sequence. By traversing all of the paths in the tree, the dynamic rendezvous sequences are obtained.

3.4: Rendezvous sequences

After both the static rendezvous sequence set and the dynamic rendezvous sequence set have been obtained, we can form the resultant rendezvous sequences by merging rendezvous pairs in each member of the cartesian product of the two sets. If no any candidate member can lead to a rendezvous sequence, the candidate concurrent path is infeasible.

An array with size n (i.e., the number of tasks) is used to keep the statement numbers for the statements to be considered for possible rendezvous. The ith element is used for task i. All of the elements of the array are initialized to be one. If a statement number in a rendezvous pair is equal to the statement number kept in an array element corresponding to the task in which the statement occurs, the statement is called *activatable*. When both statements in a rendezvous pair are activatable, the rendezvous pair is *arrangeable*.

The first rendezvous pair of both rendezvous sequences in the candidate member is checked to determine whether it is arrangeable. If either pair is arrangeable, it is generated. If both pairs are arrangeable, two possible sequences for them are generated. After the arrangeable pairs are generated, we delete them in their corresponding rendezvous sequence, add one for the corresponding array element, and repeat the checking process until (*1*) no arrangeable pair is found, or (*2*) all of the rendezvous pairs are examined. When the first termination condition occurs, deadlock happens. Namely, the candidate member can not lead to a rendezvous sequence.

4: Application of the feasibility determination method

This section exhibits the process to determine the feasibility of a rendezvous path. Assuming that the task rendezvous paths for tasks 1, 2, 3, and 4 are $R_{11} = (!C, !A, !D)$, $R_{21} = (!B, ?A, !B)$, $R_{31} = (?B, ?D, ?B, ?D)$, and $R_{41} = (?C, !D)$ where *!* and *?* denote entry call and accept statements and are followed by an entry name. It is easy to verify that the rendezvous path complies with the basic rule. Entries *A*, *B* and *C* are static entries because they have only one calling task. However, the entry call statements corresponding to entry *D* belong to different tasks, so entry *D* is a dynamic entry. For ease of description, the statement number of the *k*th rendezvous statement in task rendezvous path R_{ij}, is abbreviated as S_{ijk}.

The static valid sequences for static entries *A*, *B*, and *C* are: $[(S_{112}, S_{212})]$, $[(S_{211}, S_{311}), (S_{213}, S_{313})]$, and $[(S_{111}, S_{411})]$ respectively. Two static rendezvous sequences: (*1*) $[(S_{211}, S_{311}), (S_{111}, S_{411}), (S_{112}, S_{212}), (S_{213}, S_{313})]$ and (*2*) $[(S_{111}, S_{411}), (S_{211}, S_{311}), (S_{112}, S_{212}), (S_{213}, S_{313})]$ are obtained after topological sorting. Since the unique dynamic entry *D* has two accept statements and two entry call statements, it has only two dynamic valid sequences: (*1*) $[(S_{113}, S_{312}), (S_{412}, S_{314})]$ and (*2*) $[(S_{412}, S_{312}), (S_{113}, S_{314})]$. They are the dynamic rendezvous sequences. Thus, there are four members in the cartesian product of the static and dynamic rendezvous sequence sets. Figure 1 shows the process to merge the first static rendezvous sequence and the second dynamic rendezvous sequence.

5: A heuristic approach to feasible concurrent path selection

In this section, we propose a heuristic approach to selecting feasible concurrent paths for the two most widely used coverage criteria, node and branch coverage. First, we construct a coverage table under a specified coverage criterion. A *coverage table* is a two-dimensional matrix. The row of the table is indexed by the test elements, i.e., nodes or branches; the column is indexed by the feasible concurrent paths. The element in the *i*th row and the *j*th column represents the occurrence (*0* or *1*) of the *i*th test element (a node or a branch) in the *j*th feasible concurrent path.

5.1: Reduction of the coverage table

Since the coverage table exposes that some feasible concurrent paths must be selected and some others must not be selected in any case, a procedure is needed to designate these kinds of feasible concurrent paths. Deleting them and their covering test elements results in a reduced coverage table.

Some terms need to be defined and used in the procedure. Row *i dominates* row *j* if the set of feasible concurrent paths covering test element *i* is a superset of the set of feasible concurrent paths covering test element *j*. Since a feasible concurrent path covering test element *j* should cover test element *i*, row *i* can be deleted. Row *i* is *insensitive* if every feasible concurrent path passes through test element *i*. Row *i* can be deleted because the corresponding test element is always covered. Row *i* is *essential* if only one feasible concurrent path passes through test element *i*. The unique passing feasible concurrent path should be definitely selected. Column *j* is *overlapped* by column *i* if the set of test elements passed by feasible concurrent path *j* is a subset of the set of test elements passed by feasible concurrent path *i*. Hence, column *j* can be deleted and the feasible concurrent path *j* should not be selected definitely.

PROCEDURE REDUCE (in *E*, out *E'*, out P_s, out P_n)

{*E* and *E'* are the coverage table and the reduced one. $E = (M, P)$ where *M* and *P* are the original sets of test elements and feasible concurrent paths in *E* respectively. $E' = (M', P')$ where *M'* and *P'* are the reduced sets of test elements and feasible concurrent paths in *E'* respectively. P_s and P_n are the sets of definitely selected and definitely not selected feasible concurrent paths. P_s and P_n are initialized to be empty.}

```
Step 1. $M' = M, P' = P$
Step 2. Delete the dominating rows, $M_d$.
            $M' = M' - M_d$.
Step 3. Delete the insensitive rows, $M_v$.
            $M' = M' - M_v$.
Step 4. Delete the zero columns, $P_z$; the corresponding
            feasible concurrent paths are not selected.
            $P' = P' - P_z$, $P_n = P_n + P_z$.
Step 5. Select the feasible concurrent paths involving the
            essential rows; the corresponding columns, $P_e$, and their affected rows, $M_e$, are
```

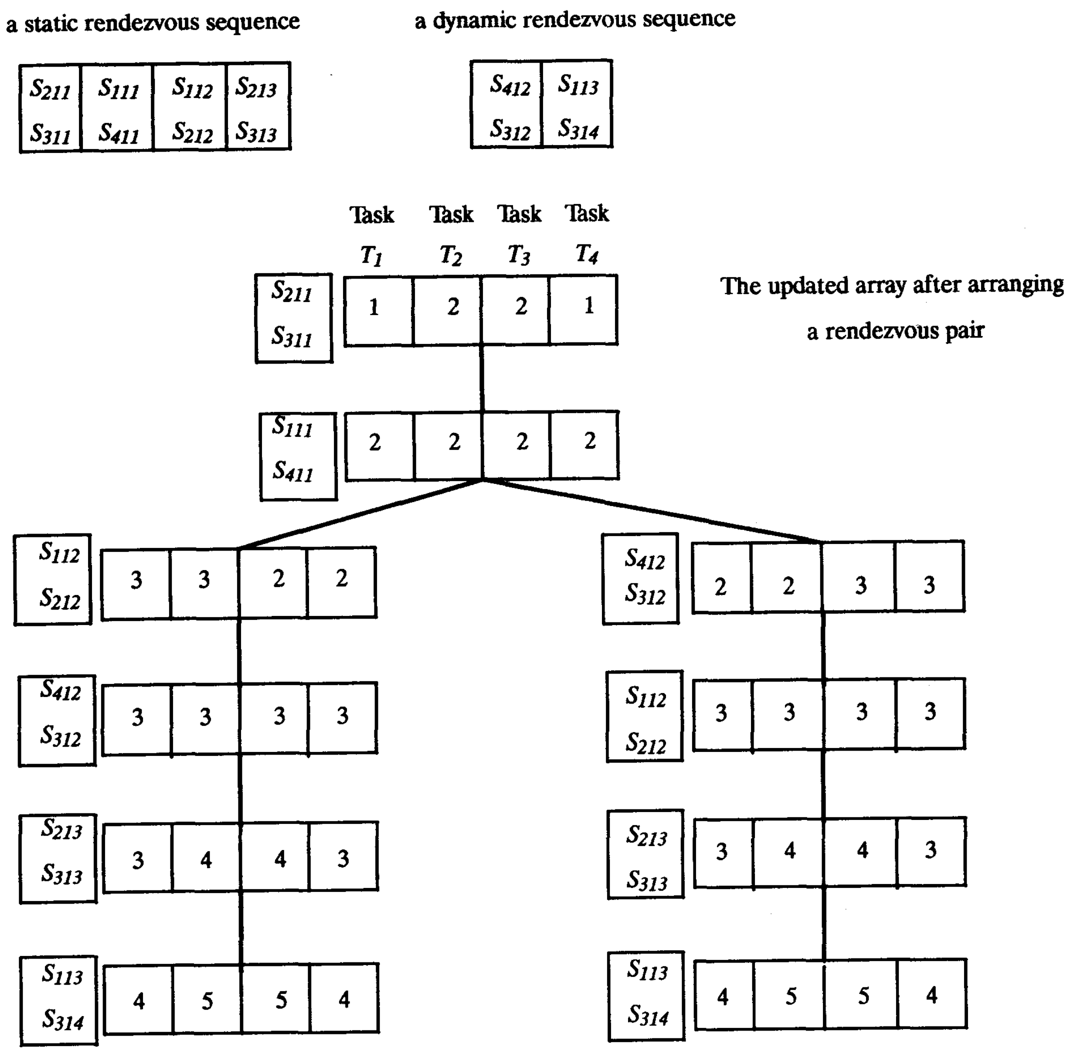
a static rendezvous sequence
S_{211} S_{311}
S_{111} S_{411}
S_{112} S_{212}
S_{213} S_{313}
a dynamic rendezvous sequence
S_{412} S_{312}
S_{113} S_{314}
Task T_1
Task T_2
Task T_3
Task T_4
The updated array after arranging a rendezvous pair
S_{211} S_{311} 1 2 2 1
S_{111} S_{411} 2 2 2 2
S_{112} S_{212} 3 3 2 2
S_{412} S_{312} 3 3 3 3
S_{213} S_{313} 3 4 4 3
S_{113} S_{314} 4 5 5 4
S_{412} S_{312} 2 2 3 3
S_{112} S_{212} 3 3 3 3
S_{213} S_{313} 3 4 4 3
S_{113} S_{314} 4 5 5 4

Figure 1. Merging a static rendezvous sequence and a dynamic rendezvous sequence.

Note that the rendezvous pair beside the updated array is an arrangeable one.

deleted.
$M' = M' - M_e$, $P' = P' - P_e$, $P_s = P_s + P_e$.

Step 6. Delete the overlapped columns, P_l; the corresponding feasible concurrent paths are not selected.
$P' = P' - P_l$, $P_n = P_n + P_l$.

Step 7. Delete the zero rows, M_z.
$M' = M' - M_z$.

Step 8. $E' = (M', P')$.

END PROCEDURE.

5.2: The greatest coverage increment first approach

After the feasible concurrent paths which have to be selected and not to be selected are designated by the procedure REDUCE, the problem size of the concurrent path selection is reduced because the coverage table is diminished. We compute the coverage increment for each feasible concurrent path in the reduced coverage table. The coverage increment refers to the number of test elements covered by a feasible concurrent path in the reduced coverage table. Then, we select the one with the greatest coverage increment. After deleting the selected feasible concurrent path and its covering test elements, we repeat the process of the coverage increment computation and path selection until there is no increment any more or all of the feasible concurrent paths are classified to be selected or not selected. The steps are detailed in the following procedure.

```
PROCEDURE SELECT (in $E$, out $P_s$)

{$E$ is the coverage table of the concurrent program. $E = (M, P)$ where $M$ and $P$ are the original sets of test elements and feasible concurrent paths in $E$ respectively. $P_s$ is the set of selected feasible concurrent paths.}

Step 1.
  Call REDUCE ($E$, $E'$, $P_s$, $P_n$).
Step 2.
  While $P'$ is not empty,
    2.1 Compute the coverage increment for each
        member in $P'$ according to $E' = (M', P')$.
    2.2 If there exist one or more feasible concurrent
        paths with the greatest coverage increment,
        2.2.1 If there is a tie,
                  Select the feasible concurrent path, $y$,
                    with the smallest number of
                    rendezvous sequences.
              Else
                  Select the unique feasible concurrent
                    path, $y$.
              End If.
        2.2.2 $P_s = P_s + \{y\}$, $P' = P' - \{y\}$,
              $M' = M' - \{$the test elements covered by $y\}$.
    Else
              Exit the while loop.
    End If.
  End While.
END PROCEDURE.
```

6: Conclusion

Reproducible testing is a necessary requirement for concurrent program testing. For path testing, the first task is to identify the set of feasible concurrent paths and their associated rendezvous sequences. Then, the test data set can be defined and the testing of a concurrent program can be reproduced. This paper proposes a systematic approach to generating feasible concurrent paths from the cartesian product of task path sets. It avoids computing all possible concurrency states which are widely used in static analysis techniques, such as reachability analysis. Another advantage of the approach is that all potential rendezvous sequences are also generated with each feasible concurrent path. They can serve as the basis for path selection, data generation and reproducible test execution. Based on this result, the path selection under node and branch coverage is accomplished by a greatest coverage increment first approach. The path selection method is simple and effective.

References

1. B. Beizer, *Software Testing Techniques*, 2nd ed., VAN NOSTRAND REINHOLD, New York, NY, 1990.
2. *Reference Manual for the Ada Programming Language*, United States Department of Defense, ANSI standard Ada, January 1983.
3. K. C. Tai, "Reproducible Testing of Concurrent Ada Programs," *Proc. of 2nd Conf. on Software Development Tools*, pp. 114–121, 1985.
4. J. J. P. Tsai, *et al.*, "A noninterference monitoring and replay mechanism for real-time software testing and debugging," *IEEE Trans. Soft. Eng.*, Vol. 16, No. 8, pp.897–916, 1990.
5. K. C. Tai, *et al.*, "Debugging concurrent Ada programs by deterministic execution," *IEEE Trans. SE*, to appear, 1990.
6. S. M. Shatz, *et al.*, "Design and implementation of a Petri net based toolkit for Ada Tasking Analysis," *IEEE Trans. Parallel and Distributed Systems*, Vol. 1, No. 4, pp. 424–441, 1990.
7. R. D. Yang, *A Path Analysis Approach to Concurrent Program Testing*, Ph.D Dissertation, National Chiao Tung Univ., Taiwan, R. O. C., 1990.
8. R. N. Taylor, "A general purpose algorithm for analyzing concurrent programs," *Communication of ACM*, Vol. 26, No. 5, pp. 362-376, 1983.
9. J. C. Lin, *The Zero–One Integer Programming Model for Path Selection Problem of Structural Testing*, Ph.D Dissertation, National Chiao Tung Univ., Taiwan, R. O. C., 1989.