Path analysis testing of concurrent programs

R-D Yang and C-G Chung*

Path analysis testing is a widely used approach to program testing. However, the conventional path analysis testing method is designed specifically for sequential program testing; it is inapplicable to concurrent program testing because of the existence of multi-loci of control and task synchronizations.

A path analysis approach to concurrent program testing is proposed. A concurrent path model is presented to model the execution behaviour of concurrent programs. In the model, an execution of a concurrent program is seen as involving a concurrent path (which is comprised of the paths of all concurrent tasks), and the tasks' synchronizations are modelled as a concurrent route to traverse the concurrent path involved in the execution. Accordingly, testing is a process to examine the correctness of each concurrent route along all concurrent paths of concurrent programs. Examples are given to demonstrate the effectiveness of path analysis testing for concurrent programs and some practical issues of path analysis testing, namely, test path selection, test generation, and test execution, are discussed. Moreover, the errors of concurrent programs are classified into three classes: domain errors, computation errors, and missing path errors, similar to the error classification for sequential programs. Based on the error classification, the potential reliability of path analysis testing for concurrent programs is assessed.

software testing, concurrent program testing, path analysis testing, validation

Testing is one of the most practical, effective approaches to software validation. Several methods, techniques, and tools have been proposed to support users in testing[1,2]. However, most previous approaches are specifically designed for sequential programs that exhibit deterministic execution behaviour, and thus for sequential program testing; directly applying these approaches to concurrent program testing may raise several new problems[3-7].

A concurrent program is a program containing components, called tasks, which can run in parallel. In the execution of a concurrent program, tasks may communicate and synchronize with one another. The result of an execution of a concurrent program **P** is determined by **P**, input, and the synchronization relationship among concurrent tasks in **P**. There is no fixed execution order among statements of concurrent tasks and the task synchronization relationships vary with the relative progress of the execution of concurrent tasks. A task may synchronize with one task in one execution, but with another task in another execution. As a result, the execution behaviour of a concurrent program is indeterminate; repeated execution of a concurrent program with the same input may produce different behaviours.

To test a program, tests should be prepared according to some coverage criteria. (In the authors' view, a test for program testing is the actual data to be tested, rather than the acts to test the program with the data.) Then the program is executed with the test. If there is error in the execution, the erroneous condition needs to be replayed to locate the error. However, because of the indeterminacy of concurrent programs, some new testing problems are raised. For example, it is difficult to know the possible execution behaviours of a concurrent program, and as a result there is no whole picture about the program to be tested. It is difficult to identify exactly an execution behaviour to test, and as a result there is no idea about which part of a concurrent program is tested in a test execution and how to assess the test result. It is difficult to control the program execution for testing a specific execution behaviour, and therefore there is no easy way to replay the error for debugging. In other words, it is generally accepted that testing concurrent programs is more difficult than testing sequential ones.

The problems of concurrent program testing were first addressed by Brinch Hansen in his early work on testing multiprogramming systems and Monitor[8,9]. However, to the authors' knowledge, few other researchers have paid attention to them since then. Recently, as concurrent programming has become more and more popular, so the problems are being noticed again. Tai generalized Brinch Hansen's concept and developed techniques for reproducible testing of concurrent programs[5,10,11]. He characterized an execution of a concurrent program by the synchronization sequence, which is the sequence of synchronization statements traversed in the execution. Execution control mechanisms for enforcing the synchronization sequence are designed. Hence if inputs and synchronization sequences for execution control, referred to as IN_SYN tests by Tai, are systematically selected then a reproducible testing strategy for concurrent programs is obtained. However, Tai did not go into detail about systematic testing.

Taylor used concurrency graphs obtained from static analysis[12] to represent the possible synchronization behaviours of a concurrent program. Based on the concurrency graph, he defined several coverage criteria[7], which are similar to the coverage criteria of sequential programs based on the flowgraph, for measuring the test coverage of synchronization behaviour. Unfortunately,

Department of Computer Science and Information Engineering, National Chiao Tung University, HsinChu, Taiwan, ROC.
*Mitac International Corp., Taipei, Taiwan, ROC

Taylor's criteria have two major defects, namely, it is complicated to generate concurrency graphs for a concurrent program and it is difficult to monitor concurrency states involved in the execution. On the other hand, Weiss's work is about the theoretical foundation of concurrent program testing[13]. He simulated the execution behaviour of a concurrent program by a set of serializations, which are sequential programs generated from merging the bodies of the tasks in the program. A formal framework for studying concurrent program testing is built. However, to generate serializations for a concurrent program is difficult and tedious, especially when the number of tasks is large and the execution process lengthy.

In summary, past research has its own emphases; some results have been achieved on different problems of concurrent program testing. But, as discussed above, it is not easy to use them in practice. There is a lack of rigorous, systematic methods for concurrent program testing.

As concurrent programming becomes more and more popular, the need for validation techniques for concurrent programs emerges. Without other breakthroughs in validation techniques, testing is still the best choice for concurrent program validation. Therefore, it is now apparent that the problems of concurrent program testing need to be reconsidered.

This paper proposes a path analysis approach to the development of an integrated testing method suitable for concurrent programs, motivated from the path analysis testing of sequential programs. As the development of an integrated testing method is a complicated task, it is impossible to complete the development in one step. The results presented in this paper constitute the first step of this development. The execution behaviour of a concurrent program is modelled by flowgraph and rendezvous graph, the test for path analysis testing defined, and a path analysis methodology for concurrent program testing presented. To assess the reliability of the methodology, a path-based approach to error classification of concurrent programs is described, which serves as a basis for the assessment.

The concurrent programming language used in the research is Ada. But, to emphasize the discussion on the issues caused by task rendezvous, concurrent Ada programs are considered that have only static naming of tasks and do not include:

- select statements with an else part, delay alternatives, or a terminate alternative
- conditional or timed entry call statements
- select statements with two or more alternatives for the same entry
- shared variables

The organization of this paper is as follows. The next section assesses the execution behaviour of concurrent programs and presents a concurrent path model using two kinds of graphs to model the static and dynamic structure of a concurrent program. Then, based on the model, a path analysis testing methodology for concurrent programs is proposed and three important practical issues discussed. The potential realiability of the proposed path analysis testing methodology is analysed, and, finally, some conclusions given.

MODELLING CONCURRENT PROGRAMS

Execution behaviour of concurrent programs

In test execution of a program, it should be continually checked if the execution behaviour of the program is correct. What is the execution behaviour of a program? It is difficult to give an exact answer as it is dependent on the requirements of the underlying tasks. The execution behaviour of a program may be the output values, signals, or the statements traversed in the execution. For example, the execution behaviour of a computation-intensive program may be the output values produced in the execution. And the execution result of a real-time control program may be the control signal generated in the execution. It is unnecessary to argue what the execution behaviour of a program really is. What is important is whether the definition of execution behaviour can meet the requirements of the underlying tasks.

From the testing point of view, one major concern is the correctness of an execution. Thus execution behaviour should be the information that is significant for determining the correctness of the execution. Based on this viewpoint for a concurrent Ada program, the correctness of the program is dependent on not only its computation result, but also its rendezvous conditions. Thus the execution behaviour of a concurrent program, in the authors' view, should consist of four parts:

- the output produced by each task
- the execution path of each task
- the rendezvous sequence of each task
- the data exchange in each rendezvous

The first two parts represent the computation behaviour and the second two parts the rendezvous behaviour. The other execution details, the execution order of statements of different tasks, the sequence of output values produced by different tasks, and the like, may also be regarded as the execution behaviour of a concurrent program. But the authors do not consider these execution details in their research, because they believe all order-sensitive intertask actions should be directly or indirectly controlled by some task rendezvous and so the correctness of those actions should be determined from the rendezvous behaviour. Furthermore, it is difficult to identify these execution details produced in the execution, especially when the program is run on a multiprocessor environment.

Concurrent path model

A concurrent program consists of concurrent rendezvous tasks. The possible execution behaviours of the concurrent program are composed of the execution behaviours of the concurrent tasks within it. Therefore, to model the execution behaviour of a concurrent program means modelling the possible execution behaviour of a single task first.

A task has its own execution flow and may rendezvous with other tasks in execution, so the execution behaviour of the task is determined by the input, the sequence of statements, and the sequence of rendezvous involved in the execution. Therefore, the possible execution behaviours of the task are bounded by the possible inputs, the possible sequence of statements, and the sequences of rendezvous of the task. Consequently, two kinds of graphs, task flowgraph and rendezvous graph, are proposed to model the execution behaviour of a task. The task flowgraph, corresponding to the syntactical view of execution behaviour of the task, models the possible execution flow of the statements of the task. The task rendezvous graph, corresponding to the run-time view, models the possible rendezvous sequences of the task.

In the execution of a concurrent program, each task will traverse its own path of its flowgraph as well as rendezvous graphs. The execution can be regarded as involving two sets of concurrent paths. One is the concurrent paths of flowgraph, called a C-path, representing the execution flow of all tasks, and the other is the concurrent paths of rendezvous graph, called a C-route, representing the rendezvous sequence of concurrent tasks. For the same input, the same execution behaviour will be produced in successive executions provided that the same C-path and C-route are involved in each execution. An execution behaviour of a concurrent program can be characterized by the input, C-path, and C-route involved in the execution. Consequently, the execution behaviour of a concurrent program can be modelled by its C-paths and C-routes. The authors' concurrent path model is founded on such an idea. The detail of the concurrent path model will be described in this section.

Program flowgraph

For the sake of easy discussion, let **P** be a concurrent program consisting of T_1, T_2 . . ., T_n, n tasks. A task T_i can be represented as a rooted, directed graph, called a task flowgraph, $G_i = (N_i,E_i)$, where N_i is a set of nodes and E_i is a set of edges. There is a one-to-one correspondence between the statements of T_i and the nodes of N_i. Edge $e = (n_1,n_2) \in E_i$ iff there is a possible flow of control from the statement that n_1 represents to the statement represented by n_2. For each G_i, there are two special nodes. One is r_i, which is the rooted node of G_i, representing task T_i's first executable statement. The other is e_i, which is the ended node of G_i, representing task T_i's last executable statement.

A task path is a sequence of nodes, $n_0n_1n_2 \ldots n_m$, where n_0 is the rooted node, n_m is the ended node, and for each $0 \leqslant j < m$, $(n_j,n_{j+1}) \in E_i$. A path represents one possible execution sequence of task T_i.

The program flowgraph of **P**, PF=(FN,FE), is a directed graph, where $FN = \bigcup_{i=1}^{n} N_i$ and $FE = \bigcup_{i=1}^{n} E_i$. For

```
task body T1 is                 task body T2 is
  Y : INTEGER;                    Z : INTEGER;
begin                           begin
  T3.E1(Y);                       T3.E1(Z);
  write(Y);                       write(Z);
end T1;                         end T2;

  task body T3 is
    X:INTEGER;
  begin
    read X;
    for i in 1..2 loop
      accept E1(T: out INTEGER) do
        X: = X + 1;
        T: = X;
      end E1;
    end loop;
  end T3;
```

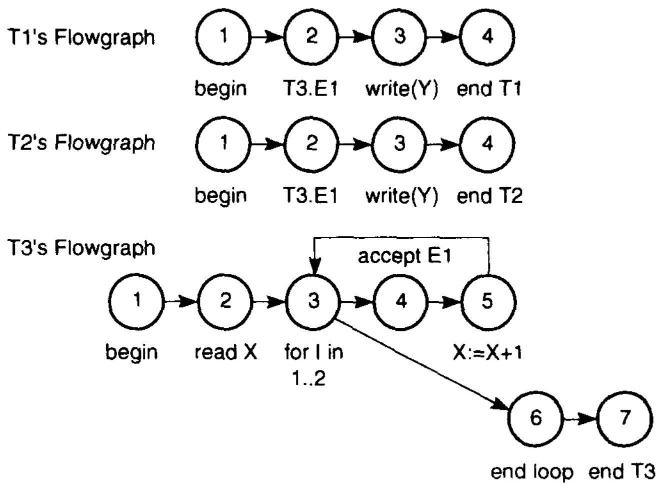


Figure 1. Program Ex1 and program flowgraph of program Ex1

example, the program flowgraph of program Ex1 is given in Figure 1.

In the execution of **P**, each task will traverse a path of its task flowgraph. Therefore, the execution can be regarded as involving a set of concurrent paths (called a C-path). A C-path is an n-tuple $(P_1,P_2, \ldots, P_n)$ where, for each i, P_i is a task path of task T_i.

A feasible C-path is a C-path $(P_1,P_2, \ldots, P_n)$ in which there is at least one input α which causes each task T_i to traverse P_i in the execution of **P** with α.

A C-path may be infeasible due to data dependency (see Figure 2) and rendezvous dependency (see Figure 3). As the infeasible C-path resulting from rendezvous dependency can be observed from syntactic structure without execution, so it is also called a static infeasible C-path. A static infeasible C-path may result in deadlock during real execution or may not be executable, but just an inherent inaccuracy of static analysis. It is assumed, unless otherwise specified, that the C-paths referred to in the following discussion are not statically infeasible.

Although a C-path has identified the statement sequence of each task, repeated execution of **P** with the same input traversing the same C-path may still produce different execution behaviours. As shown in Figure 1, program Ex1 has only one feasible C-path, but may

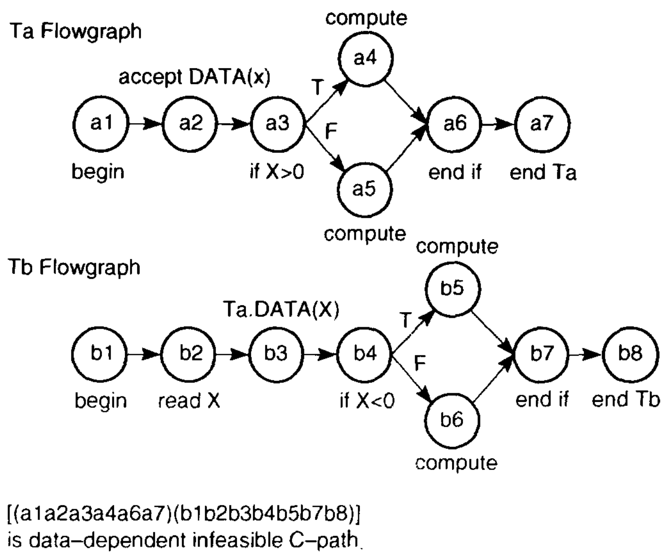


Figure 2. Infeasible C-path due to data dependency

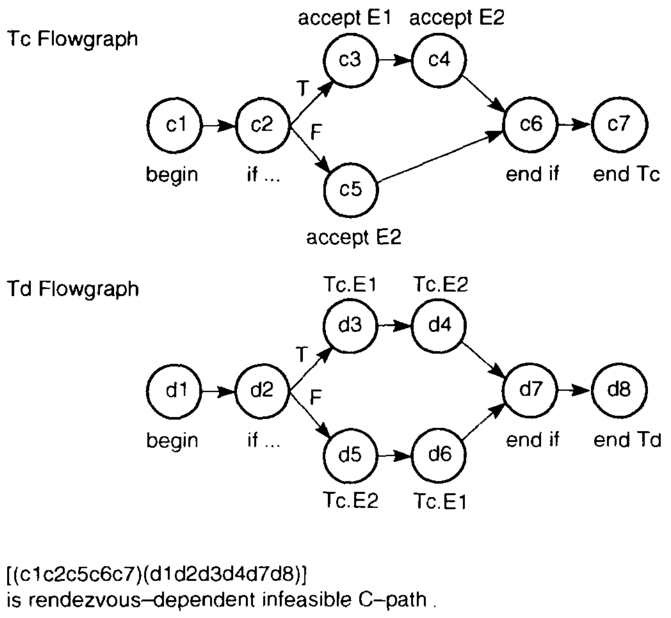


Figure 3. Infeasible C-path due to rendezvous dependency

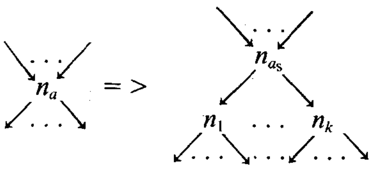
produce different outputs, (Y = X + 1, Z = X + 2) and (Y = X + 2, Z = X + 1), in successive execution with the same X. The reason is that the execution behaviour may be influenced by the rendezvous conditions among concurrent tasks. Therefore, to characterize each distinct execution behaviour along a C-path, a rendezvous graph is needed to model the rendezvous conditions among concurrent tasks.

Program rendezvous graph

A rendezvous statement is one of the following statements: select, entry call, and accept.

To model the possible rendezvous conditions of a task as a graph, an intuitive idea is to represent a rendezvous statement as a node and a possible transition from one rendezvous statement to another as an edge. However, for an accept statement accepting entry called by multiple tasks such a representation cannot reflect the nondeterministic nature of different tasks calling the same entry. In the authors' approach, the execution of an accept statement accepting a multiple-task-call entry is considered as a two-level action. At the first level, the accepting task makes a nondeterministic selection to choose one task to rendezvous, which reflects the nondeterministic arrival order of calling tasks. Then, at the second level is the task rendezvous with the chosen task. Consequently, in the model the possible rendezvous conditions of a task T_i are modelled as a task rendezvous graph, $RG_i = (RN_i, RE_i)$, where RN_i is a set of nodes and RE_i is a set of edges. The sets RN_i and RE_i are obtained from task flowgraph G_i of T_i by applying the following two rules:

(1) All nodes, except rooted and ended nodes, corresponding to nonrendezvous statements, are deleted. An edge between nodes n_1 and n_2 exists if there is a path from n_1 to n_2 where n_1 is the first node in the path and n_2 is the last node in the path, and no rendezvous statement exists between n_1 and n_2.
(2) Each node n_a corresponding to accept statement accepting entry called by k ($k > 1$) tasks is replaced by the following nodes $(n_{a_s}, n_1, n_2, \ldots, n_k)$, where n_{a_s} is a node for representing the nondeterministic selection to choose one of k tasks and each node n_j, $1 \leqslant j \leqslant k$, represents that T_i rendezvous with the jth task on the entry accepted by the statement represented by n_a. The edge $(n_{a_s}, n_j) \in RN_i$, for each $1 \leqslant j \leqslant k$. For each edge $(n_x, n_a) \in E_i$, there is an edge $(n_x, n_{a_s}) \in RN_i$ and, for each edge $(n_a, n_y) \in E_i$, there are k edges $(n_j, n_y) \in RN_i$ $1 \leqslant j \leqslant k$. This rule can be represented by the following graphical form:



These two rules only use syntactic information of a concurrent program, so it is easy to construct the task rendezvous graph of a task from its task flowgraph. Different rules can be defined for different concurrent programming languages. So the approach to concurrent program modelling can also be applied to other languages. This is a benefit of the model.

A task route (to distinguish the name of the path of rendezvous graph from that of flowgraph, the term 'route' is used instead of 'path') is a sequence of nodes, $n_0 n_1 n_2 \ldots n_m$, where n_0 is the rooted node, n_m is the ended node, and for each $0 \leqslant j < m$, edge $(n_j, n_{j+1}) \in RE_i$. A route represents one possible rendezvous sequence of task T_i.

The program rendezvous graph of **P**, RG = (RN,RE), is a directed graph, where $RN = \bigcup_{i=1}^{n} RN_i$ and $RE = \bigcup_{i=1}^{n} RE_i$. For example, the program rendezvous graph of Ex1 is shown in Figure 4. Note that the black node in Figure 4 is introduced for the convenience of illustration. In

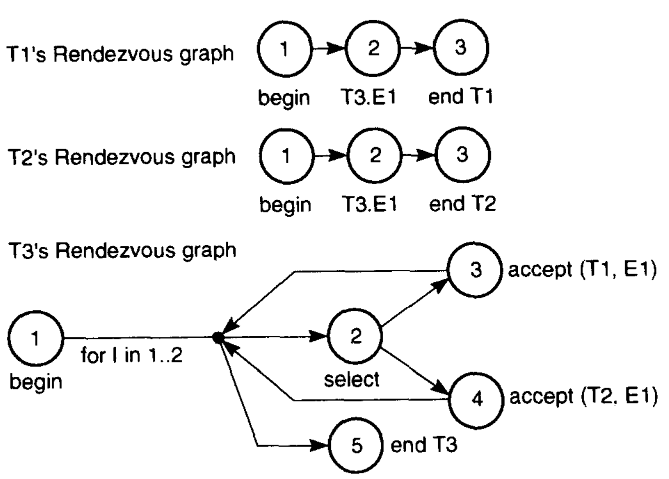
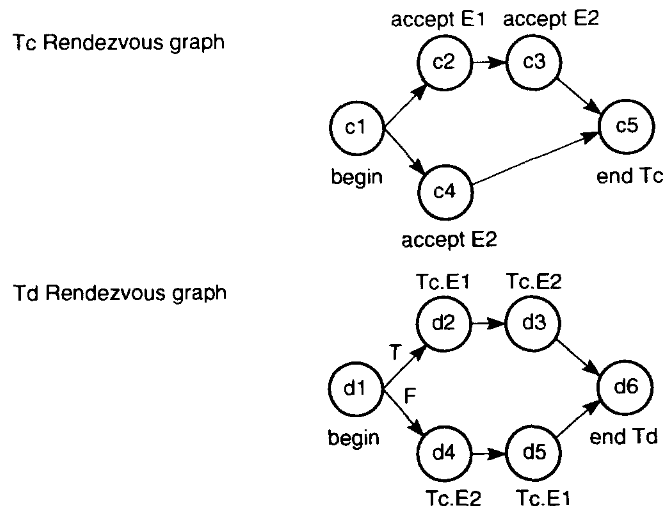


Figure 4. Program rendezvous graph of program Ex1



[(c1c2c3c5)(d1d4d5d6)]
is an infeasible C–route.

Figure 5. Example of infeasible C-route

fact, such a node does not exist in the real program rendezvous graph.

In the execution of a concurrent program, each task will traverse a task route of its rendezvous graph. Therefore, the execution can be regarded as involving a set of concurrent task routes (called a C-route). A C-route is an n-tuple $(R_1,R_2, \ldots, R_n)$ where, for each i, R_i is a task route of task T_i.

A feasible C-route is a C-route $(R_1,R_2, \ldots, R_n)$ in which there is at least one input α that causes each task T_i to traverse R_i in the execution of **P** with α. The existence of infeasible C-routes is because the rendezvous sequences of concurrent tasks are inconsistent (see Figure 5).

In the execution of a concurrent program, it will traverse a C-path as well as a C-route. The C-routes that are possibly traversed in an execution along a C-path are referred to as the C-routes along this C-path. Let $\mathbf{P}_i$ be a C-path. The set of C-routes along $\mathbf{P}_i$ is denoted as C-route $(\mathbf{P}_i)$.

Based on the definition of C-route, the following theorem can be put forward.

Theorem 1
If two executions of a concurrent program with the same input traverse the same C-route, then these two executions must produce the same execution behaviour.

Proof
It has been shown[14] that the execution behaviour of a concurrent program **P** with an input x can be characterized by the partial order rendezvous sequences of each task in **P**. According to the definition of rendezvous graph, represent the entry calls on the same entry as different nodes, so a C-route defines a unique rendezvous sequence for each task, which means the execution behaviour can be characterized by the C-route traversed in the execution.

Corollary 1
In two executions of a concurrent program with the same input, if these two executions traverse the same C-route then they must traverse the same C-path.

Proof
This is an immediate result of theorem 1.

Theorem 1 has shown that an execution behaviour of a concurrent program can be characterized by the input, C-path, and C-route involved in the execution. Thus the model of concurrent programs is obtained.

Model

In the authors' view, **P** is composed of a set of C-paths and each C-path has a set of C-routes along with it. In the execution of **P** with an input each task will traverse its own task path as well as a task route. The C-path (C-route) involved in this execution is composed of the task paths (task routes) traversed in the execution. In different executions, different C-paths and C-routes may be traversed when some tasks traverse different task paths (task routes). **P** will produce the same execution behaviour in successive executions of **P** with the same input provided that **P** traverses the same C-path as well as the same C-route. Therefore, a three tuple (α,σ,δ) is used to represent one possible execution behaviour of **P**, called a computation of **P**, where α is an input, σ is a C-path, and δ is a C-route. The three tuple (α,σ,δ) corresponds to the computation realised by the execution of **P** with input α, traversing C-path σ and C-route δ. Although, according to corollary 1, the execution behaviour of **P** with an input can be characterized by the C-route traversed in the execution, a C-path is included into the model of computation. This is because the C-path can be served as a medium to connect the inter-relationship between input and C-route, and as a basis to partition the possible computations of **P** into subgroups.

Let **D** be the set of inputs of **P**, **R** the set of C-routes of the rendezvous graph of **P**, and **F** the set of C-paths of the flowgraph of **P**. The possible computations of **P** are bounded by the set $\psi = \{(\alpha,\sigma,\delta) | \alpha\epsilon\mathbf{D},\sigma\epsilon\mathbf{F},\delta\epsilon\mathbf{R}\}$.

C-path σ of Ex1:
[(1,2,3,4),(1,2,3,4),(1,2,3,4,5,3,4,5,3,6,7)]
The following four C-routes are syntactically possible C-routes of σ:
C-route δ1: [(1,2,3),(1,2,3),(1,2,3,2,4,5)]
C-route δ2: [(1,2,3),(1,2,3),(1,2,4,2,3,5)]
C-route δ3: [(1,2,3),(1,2,3),(1,2,3,2,3,5)]
C-route δ4: [(1,2,3),(1,2,3),(1,2,3,2,4,5)]
δ1 and δ2 are feasible C-routes and so the computations, C1 = (x,σ,δ1) and C2 = (x,σ,δ2), are feasible.
δ and δ4 are infeasible C-routes and so the computations, C3 = (x,σ,δ3) and C4 = (x,σ,δ4), are infeasible.

Figure 6. Computations of Ex1

Definition 1
The concurrent path model of a concurrent program **P** is a four tuple (**D**,**R**,**F**,ψ) where:

- **D** is the set of inputs of **P**
- **R** is the set of C-routes of the program rendezvous graph of **P**
- **F** is the set of C-paths of the program flowgraph of **P**
- ψ is the set of possible computations of **P**.ψ = {(α,σ,δ)| where α∈**D**, σ∈**F**, and δ∈**R**}

The set ψ is possibly larger than the set of real computations of **P**. There are some computations (α,σ,δ) that cannot be realised by any execution of **P**. This is because σ(δ) may be an infeasible C-path (C-route) or α may not be an input traversing σ(δ). For example, some feasible and infeasible computations of Ex1 are shown in Figure 6.

A computation, λ, of **P** that cannot be realised in any execution of **P**, is called infeasible, denoted by infeasible (**P**,λ); otherwise, it is called a feasible computation, denoted by feasible (**P**,λ)

To determine the feasibility of a computation (α,σ,δ) a mechanism to check the possible execution conditions of **P** is needed. In practice, such a mechanism can be obtained by providing an execution control mechanism to force the execution of **P** to traverse σ and δ with input α. (Note that as σ is determined by α and δ so an execution control mechanism for enforcing δ is enough.) The run-time monitor of the execution control mechanism takes the responsibility for identifying the feasibility of the execution. The monitor signals that an execution is infeasible when it detects that the execution of some tasks are blocked by the execution control mechanism because the next nodes (rendezvous statements) to be executed are inconsistent with that defined in σ (δ) and no other task is ready to run. The execution control mechanism then rejects this computation for further investigation. Note that if the tasks are blocked because of mutually waiting, not because of the execution control mechanism, then the execution is dead. For example, forcing the execution to traverse the infeasible C-path shown in Figure 3 will result in dead computations. The execution of task Tc and Td will be blocked because of unsatisfied rendezvous requests. A computation resulting in dead execution in the controlled execution corresponds to the existence of errors in **P**. To separate dead computations from infeasible computations is a major function of the run-time monitor of the execution control mechanism.

The C-route of the approach is similar to the synchronization sequence of Tai. But, based on the graph model, it is easier to analyse. Besides, the authors' approach includes a static C-path, which connects the inter-relationship among inputs and C-routes. Thus a structural testing approach is obtained for generating input for a specific rendezvous condition and vice versa. In comparison with Weiss's approach, the authors' approach is a partial order one. It is not necessary to arrange the order of irrelevant statements nor to merge the body of concurrent tasks to obtain a feasible interleaving. It is left to the running environment. The running environment always chooses a feasible interleaving. If no such interleaving exists, this test must be infeasible. The running environment will reject it. But, with Weiss's approach, it is necessary to merge the body of concurrent tasks in advance to generate a feasible interleaving. For example, to test the program Ex1 with Weiss's approach, it is necessary to arrange the execution of all statements into a totally ordered sequence, including rendezvous statements and sequential statements. Thus if the statement sequence is not in a proper order, for example, write(Y) in T1 is placed in front of accept E1(T. . .) in T3, that means the entry call completes before it is accepted. This is impossible in Ada so the statement sequence must be infeasible. Care must be taken not to generate such statement sequences in Weiss's approach. But, with the authors' approach, it is necessary only to select the static C-path and the order of tasks calling the same entry. The resolution of irrelevant orders is left to the run-time environment, such as the order of the sequential statements and accept statement mentioned above. So the model of concurrent programs is more suitable for concurrent program testing as the generation and realisation of computations of a concurrent program is simpler than other approaches.

PATH ANALYSIS TESTING FOR CONCURRENT PROGRAMS

Test for path analysis testing

In the authors' model, a computation of a concurrent program is characterized by the three tuple (x,σ,δ), where x is an input, σ a C-path, and δ a C-route. According to corollary 1, σ can be uniquely determined by (x,δ). Thus the two tuple (x,δ) is used as a test for concurrent program testing where x is an input and δ is a C-route. A (x,δ) can definitely identify the exact computation to be tested. Such a test is called a definite test.

Theorem 2
The definite tests of a concurrent program are partitioned into a disjointed subset by C-paths.

To select a definite test, a strategy to select C-routes as well as inputs is needed. As the inputs and C-routes of **P** are inter-related, without considering the execution path of each task it is difficult to generate inputs for a specific

C-route and vice versa, so, in practice, a strategy is also needed to select C-paths to connect the relationship among them. To execute a definite test, an execution control mechanism should enforce the C-route specified in the test. As controlled execution introduces overhead during execution, so a single input value may also be used as a test, which is called an indefinite test. To execute an indefinite test is simple, it is necessary only to execute **P** with the test. The execution must produce a feasible computation without introducing execution overhead, but with the loss of reproducibility and, possibly, reliability. Therefore, a compromised trade-off between execution overhead and effectiveness is an important issue of concurrent program testing.

Test execution strategy

To execute a concurrent program with definite tests, two approaches can be employed:

- nondeterministic execution approach
- controlled execution approach

In the former approach, the program is executed without control, that is, only the input part of the definite test is used. The rendezvous sequence part is seen as output of the program. For a definite test (x,δ), if, in an execution with x, the concurrent tasks traverse the same C-route δ then the test (x,δ) is considered as having been tested. This approach is simple. But, for a set of definite tests that have the same input but different C-routes, several executions do not guarantee that each test will be covered. In the latter approach, the program is executed under the control of an execution control mechanism. For a definite test (x,δ), the mechanism controls the interaction of concurrent tasks in accordance with δ. In this way, the C-route part of a definite test is seen as another kind of input. A program is executed just as it can accept a definite test. This approach can guarantee that all definite tests will be covered. However, it introduces some overheads, such as extra rendezvous and unnecessary serialization, during the controlled execution. Thus it is an expensive approach.

The authors' approach is a compromise. They partition the test execution of a concurrent program into two stages. In the first stage, the nondeterministic execution approach is used. The purpose of this stage is to minimize the need of execution control as much as possible. After a certain number of definite tests are tested or when errors are detected, the controlled execution approach is then used to address those that are possibly erroneous but not covered in the first stage.

Path analysis testing methodology

The basic spirit of conventional path analysis testing can be interpreted from two viewpoints. One is the function partition view. A program is seen as containing a set of functions, each implemented by a path of the program. The correctness of the program is founded on that of each function. Based on this viewpoint, testing is a process to examine the correctness of each path. The other viewpoint is the input space partition view. A program is considered as having an input space. A path corresponds to a subset of inputs to traverse it in the execution and so the input space is partitioned into subspaces by the path. Each subspace corresponds to a group of inputs that have some error detection capabilities in common. Based on this viewpoint, the idea of path analysis testing is to partition the program input space into path domains. The program is then executed on tests that are constructed by choosing test data from these domains.

Based on the same concept, a C-path can be seen as implementing a function of a concurrent program. If each C-path is correct then the program is correct. From this viewpoint, testing a concurrent program is a process to examine the correctness of each C-path of the concurrent program. In the partitioning view, as described in the previous section, a C-path partitions the definite tests of a concurrent program into disjointed subsets. But one difference is that the possible definite tests of a C-path are further partitioned into subsets by the C-routes along the C-path. The definite tests with different C-routes correspond to different rendezvous conditions. Thus it is more reasonable to select definite tests from each subset partitioned by the C-routes.

In summary, the authors' approach to path analysis testing of concurrent programs is to view a concurrent program as consisting of a set of C-paths that partition the definite tests of the program into disjoint subsets. Testing is a process to examine the correctness of each C-path. To test a C-path, a set of definite tests is constructed by choosing one definite test from each subset partitioned by the C-routes along the C-path.

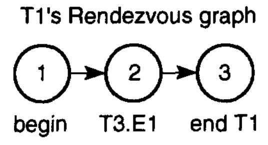
In practice, the possible number of C-paths is huge. It is impossible to test them all. Thus the issue in path analysis is to select and exercise a small but sufficient set of C-paths. To do so, the strategy for path analysis testing of concurrent programs consists of the following six basic processes:

- Select a set of task paths for each individual task.
- Select a set of C-paths composed of the selected task paths.
- Generate the C-routes along each selected C-path.
- Generate test inputs for each C-route along the concurrent selected C-paths.
- Execute the program with the input part of definite tests.
- Execute the program with controlled execution.

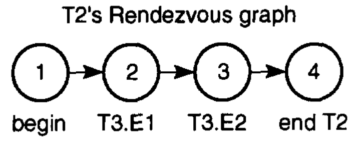
Examples

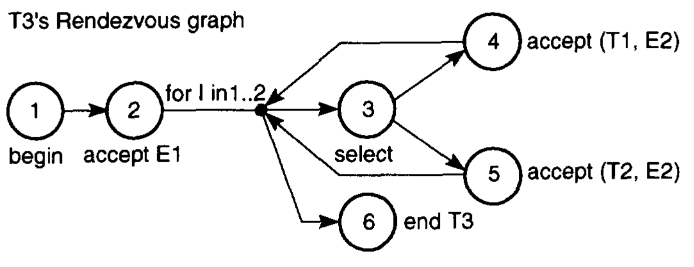
To illustrate the effectiveness of the above path analysis testing methodology, the methodology is applied to test some concurrent programs. It can be observed that several buried errors can be effectively detected with the methodology.

```
     task body T1 is
       N:INTEGER;
1    begin
2      read N;
3      T3.E2(N,1);
4    end T1;
```



```
     task body T2 is
       M,L:INTEGER;
1    begin
2      read M,L;
3      T3.E1(L);
4      T3.E2(M,2);
5    end T2;
```





```
     task body T3 is
       V1,V2,V3,V4,ID,RESULT : INTEGER;
       SUM: array(1..2) of INTEGER;
1    begin
2      accept E1(V1 : in INTEGER) do
3        V3:=V1;
4      end E1;
5      for I in 1..2 loop
6        accept E2(V2,TID : in INTEGER) do
7          V4 := V2;
8          ID := TID;
9        end E2;
10       SUM(ID) := V3 + V4;
     <error : SUM(1) := V3 + V4;>
11     end loop;
12     write(SUM(1),SUM(2));
13   end T3;
```

Figure 7. Program Ex2, and its program rendezvous graph

C-path
[(1,2,3,4),(1,2,3,4,5),
(1,2,3,4,5,6,7,8,9,10,5,6,7,8,9,10,11,12,13)]

C-route R1:
[(1,2,3),(1,2,3,4),(1,2,3,4,3,5,6)]

C-route R2:
[(1,2,3),(1,2,3,4),(1,2,3,5,3,4,6)]

Figure 8. C-path and two C-routes of Ex2

Example 1

To test the program Ex2 in Figure 7 with the path analysis testing methodology, first, the task path for each task is selected. As can be seen, each task in Ex2 has exactly one task path. Therefore, with these task paths, second, a unique C-path can be composed for testing Ex2. Third, based on the analysis of possible rendezvous conditions, two possible C-routes, R1 and R2, can be generated for the C-path of Ex2 (see Figure 8). Fourth, test inputs are generated to compose definite test cases for each C-route. Finally, two stages of test execution are applied to each generated definite test case.

Assume that, in task T2 of Ex2, the index ID is misused as I. This error can be detected by the execution with definite test case ((M,N),R2), where $M \neq N$.

Example 2

The program shown in Figure 9 implements the bounded buffer problem with buffer size two. This program, called BOUNDED_BUFFER, consists of three tasks BUFFER, PRODUCER, and CONSUMER. It is forbidden to deposit data when the buffer is full and to withdraw data when the buffer is empty. To test BOUNDED_BUFFER, first, select task paths for BUFFER, PRODUCER, and CONSUMER. The task paths of PRODUCER and CONSUMER correspond to the amount of data that are deposited to and withdrawn from the buffer, respectively, while the task path of BUFFER corresponds to the arrival sequence of tasks calling BUFFER. Second, to test the boundary interaction conditions among BUFFER, PRODUCER, and CONSUMER, compose the three C-paths CP1, CP2, and CP3 shown in Figure 9. CP1 corresponds to the conditions that PRODUCER deposits data to an empty buffer then CONSUMER withdraws the data, CP2 corresponds to the condition that CONSUMER withdraws data from an empty buffer before PRODUCER deposits the data, and CP3 corresponds to the condition that PRODUCER deposits data to a full buffer. Third, generate C-routes for CP1, CP2, and CP3, which are C1, C2, and C3, as shown in Figure 10, respectively. Fourth, generate D1, D2, and D3, which are, respectively, the definite test cases generated for testing CP1, CP2, and CP3. Finally, two stages of test execution are applied to D1, D2, and D3.

For a correct BOUNDED_BUFFER, D1 must correspond to feasible computation, and D2 and D3 must result in infeasible computation. Based on such characteristics, these three definite test cases can detect several kinds of errors that may exist in BOUNDED_BUFFER. For example, D2 is effectively for detecting the error that the data count of BUFFER is illegally initialized with 1. It is because, with the error, D2 becomes feasible. D1 is effectively for testing the error that '$COUNT > 0$' is written as '$COUNT <= 0$' as, with such an error, test execution with D1 will result in infeasible computation. And D3 can detect the error that '$COUNT < 2$' is written as '$COUNT <= 2$' because with this error D3 will become feasible.

Practical issues

To use the proposed methodology in practice, the following three major issues should be solved.

```
   procedure BOUNDED_BUFFER is
    task PRODUCER;
    task CONSUMER is
     entry COUNT(in INTEGER);
    end CONSUMER;
    task BUFFER is
     entry DEPOSIT(V : in CHAR);
     entry WITHDRAW(V : out CHAR);
    end BUFFER;

   task body PRODUCER is;
    ...
1   begin
2    read N;
    -- DATA Count
3    CONSUMER.COUNT(N)
4    for I in 1..N loop
5     read DATA
6     BUFFER.DEPOSIT(DATA);
7    end loop;
8   end PRODUCER;
   task body CONSUMER is
    N : INTEGER;
1   begin
2    accept COUNT(N) ...;
3    for I in 1..N loop
4     BUFFER:WITHDRAW(DATA);
5      compute RESULT;
6      output RESULT;
7     end loop;
8   end CONSUMER;

    task body BUFFER is
     COUNT : INTEGER;
     DATA : CHAR;
1    begin
2     COUNT := 0; {error: COUNT := 1}
     -- initial DATA count
3     loop
4      select
5       when COUNT<2 => {error: COUNT <=2}
6        accept DEPOSIT(DATA) ...;
7        COUNT := COUNT+1;
8      or
9       when COUNT>0 => {error: COUNT<=0}
10       accept WITHDRAW(DATA)...;
11       COUNT := COUNT-1;
12     end select;
13    end loop;
14   end BUFFER;

CP1 = [(1,2,3,4,5,6,7,8),(1,2,3,4,5,6,7,8)
       (1,2,3,4,5,6,7,3,4,8,9,10,11,12,13,14)]

CP2 = [(1,2,3,4,5,6,7,8),(1,2,3,4,5,6,7,8)
       (1,2,3,4,8,9,10,11,3,4,5,6,7,12,13,14)]

CP3 = [(1,2,3,4,5,6,4,5,6,4,5,6,7,8)
       (1,2,3,4,5,6,4,5,6,4,5,6,7,8)
       (1,2,3,4,5,6,7,3,4,5,6,7,3,4,5,6,7,3,4,8,9,
        10,11,3,4,8,9,10,11,3,4,8,9,10,11,12,13,14)]
```

Figure 9. Program BOUNDED_BUFFER, and its three C-paths, CP1, CP2, CP3

C-path selection

Coverage criteria should be defined for C-path selection. Defining coverage criteria for concurrent programs should consider two different viewpoints. One is from the task local view. Criteria should be defined for the test coverage of the execution behaviour of an individual task. The coverage criteria defined for sequential programs, such as statement and branch coverage, can also be used for this purpose. Another viewpoint is from the program integration view. Criteria should be defined for the test coverage of program integrated execution behaviour, especially for the rendezvous conditions among concurrent tasks. Taylor and Weiss have addressed this issue but, as discussed above, there are some problems in their approaches. There is still a lack of practical coverage criteria for program integrated behaviour.

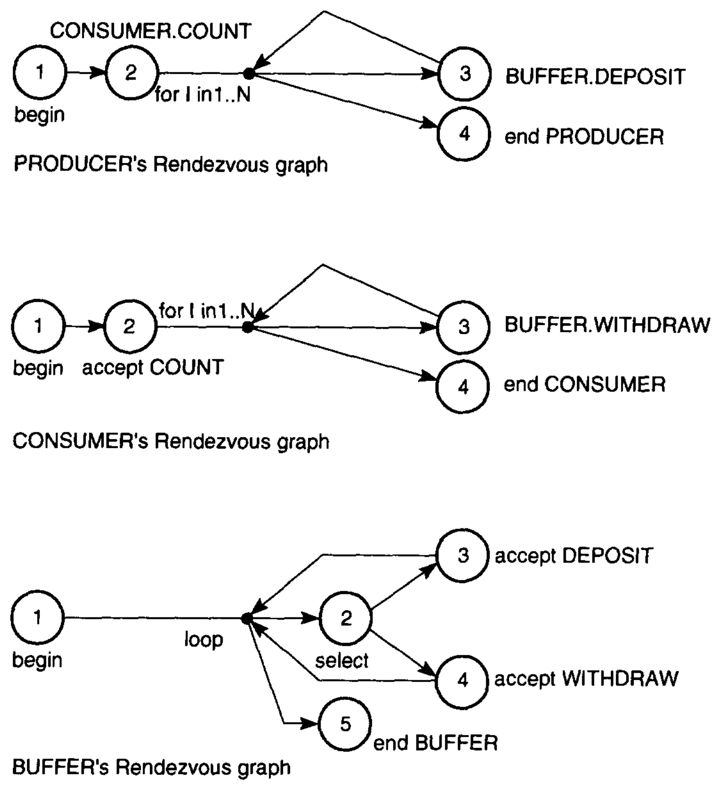
In addition, techniques are also needed to generate C-paths that satisfy the desired coverage criteria. There have been several path generation techniques for sequential programs. They can be used for path generation of individual tasks according to task local coverage criteria. However, C-path generation is much more difficult because simultaneous consideration needs to be given not only to the path of concurrent tasks but also to the inter-relationship among the paths of synchronized tasks. No doubt, to consider multiple related tasks' paths at the same time is more difficult than to consider one at a time. However, previous research on path generation problems emphasized that for sequential programs; no attention, to the authors' knowledge, has been paid to that for concurrent ones.

Definite test generation

This process can be further separated into two subprocesses, namely, C-route generation and input generation. The former subprocess requires a static analysis technique to analyse the statically possible C-routes for each C-path to identify the static errors and possible distinct computations along each C-path. The latter subprocess requires a symbolic evaluator to compute the domain of inputs for each C-path. Combining together the generated C-routes and inputs, the domain of definite tests of each C-path can be obtained. Then, some definite test can be selected from the path domain for test execution. The selection might be guided by the characteristic of each type of errors. For example, each C-route along each C-path should be tested at least once to detect rendezvous-dominated errors. The domain testing method developed for sequential programs can also be used for input-dominated error. But for racing errors, further study is needed.

Test execution

As discussed before, the authors' text execution strategy consists of two stages, namely, monitoring and controlling stages. The issue of test execution, therefore, is to design a test execution mechanism for tracing and enforcing a C-route. Some execution control mechanisms have been proposed for debugging concurrent programs[4,15]. Users can examine execution details of concurrent programs with these mechanisms. However, since testing usually involves a large volume of test cases, the more execution details corresponds to the more efforts



C-route C1:
[(1,2,3,4),(1,2,3,4),(1,2,3,2,4,5)]

C-route C2:
[(1,2,3,4),(1,2,3,4),((1,2,4,2,3,5)]

C-route C3:
[(1,2,3,3,3,4),(1,2,3,3,3,4),
(1,2,3,2,3,2,3,2,4,2,4,2,4,5)]

D1=("A",C1)

D2=("A",C2)

D3=("AAA",C3)

Figure 10. Program rendezvous graph, three C-routes, and definite test cases of BOUNDED_BUFFER

required in test execution. These execution control mechanisms are not suitable for testing purposes. From the testing viewpoint, the major concerns of a test execution mechanism are reproducibility and simplicity. Reproducibility means the test result could be reproduced for further investigations, specifically, for analysing test coverage and locating errors. Simplicity means it should minimize the tester's efforts and the interference to the program's original execution behaviour, in test execution. There is still a lack of such a test execution mechanism.

RELIABILITY ANALYSIS

This section discusses the reliability of the path analysis testing methodology for concurrent programs. First, the errors of concurrent programs are classified into several classes and then it is proved that some classes of errors can be detected by testing each C-path and C-route at least once, but some cannot. Thus the study shows a good result of the potential reliability of path analysis testing for concurrent programs. This study is similar to that of Howden[2]. Howden's result, however, is not applicable to path analysis testing of concurrent programs since, as shown later, concurrent programs exhibit quite different error characteristics from those of sequential ones.

This section is organized as follows. First, the error characteristic of concurrent programs from a path-based perspective is discussed. Second, the potential reliability of path analysis testing to detect these errors is assessed.

Errors classification

It is useful to distinguish different types of errors of a program, thereby selecting the most suitable tests for detecting different types of errors. There are several methods for classifying errors in a sequential program. Conventional classifications, however, are not applicable for concurrent programs. Because a concurrent program has concurrent synchronized tasks, new types of errors may be contained in a concurrent program, for example, the synchronization error[5,15]. But there is no general agreement about the classification of errors in a concurrent program. This is because not only are the characteristics of errors in a concurrent program not well understood, but also there is no suitable basis for error classification.

This section presents a path-based approach to error classification of concurrent programs. The approach is derived for the path-based error classification for sequential programs[2].

In path-based classification for sequential programs, program errors are characterized by the difference between the path of the program and that of a hypothesis correct program. If each path of a program is associated with a path domain, which is the subset of inputs that may traverse the path, and a path computation, which is the sequence of computations carried out by the path, the effects of program errors can be described in terms of their effects on the path domain and path computation of the paths. Accordingly, errors of a sequential program P can be classified into three broad classes:

- P is said to have a domain error if the path domain of a path in P is different from that of the corresponding path in the hypothesis correct program.

- P is said to have a computation error if the path computation of a path in P is different from that of the corresponding path in the hypothesis correct program.
- P is said to have a missing path error if some paths exist in the hypothesis correct program but do not exist in P.

Based on the same concept, a path-based error classification for concurrent programs can be obtained if the definition of the path domain and path computation for a C-path is extended.

The set of definite tests is partitioned into subsets by the C-path. Therefore, associated with each C-path is the subset of definite tests that causes the C-path to be followed and a sequence of computations that is carried out in traversing the C-path. The sequence of computation is defined by the sequence of statements of the C-path and the C-route traversed in the execution.

Definition 2

Suppose $\mathbf{P}_i$ is a C-path of a program **P**. Then the path domain $\mathbf{Z}_i = \mathbf{Z}(\mathbf{P}_i)$ for $\mathbf{P}_i$ is the subset of the definite tests that causes $\mathbf{P}_i$ to be traversed. The path computation $\mathbf{Q}_i = \mathbf{Q}(\mathbf{P}_i)$ for $\mathbf{P}_i$ is the function computed by the sequence of computation in traversing $\mathbf{P}_i$.

Let $\mathbf{R}_{i1}, \mathbf{R}_{i2}, \ldots, \mathbf{R}_{ir}$ be the C-routes along a C-path $\mathbf{P}_i$. One C-route, say $\mathbf{R}_{ij}$, defines one kind of intertask data dependency and identifies one subset of inputs, namely, $\mathbf{D}_{ij}$, to traverse $\mathbf{P}_i$ with $\mathbf{R}_j$. Symbolic execution[16,17] can be used to construct a system of predicates in terms of input variables that describes the domain of $\mathbf{D}_{ij}$ and thus a subdomain $\mathbf{Z}_{ij} = \{(x,\mathbf{R}_{ij}) | \, x \in \mathbf{D}_{ij}\}$ of definite tests. Different C-routes define different intertask dependencies and thus correspond to different subdomains. Accordingly, r subdomains of definite tests for $\mathbf{P}_i$ can be obtained. Composing all subdomains together, the path domain of $\mathbf{P}_i$ can be obtained. Similarly, symbolic execution can also be used to construct a set of expressions describing the path computation in traversing $\mathbf{P}_i$ for each $\mathbf{R}_{ij}$.

The error of concurrent programs can be classified into the three classes, namely, domain error, computation error, and missing path error, according to their effects on the path domain and path computation. Let **P*** be a hypothesis correct program corresponding to **P**. If there is an isomorphism between the C-path of **P** and **P*** then **P** must be correct. If **P** is not correct, then there must be no isomorphism between **P** and **P***. Either the domains or the computations, or both, of **P** and **P*** will be different.

Definition 3

A concurrent program **P** is said to have a domain error if there is an isomorphism between the C-path $\mathbf{P}_i$ of **P** and the C-path $\mathbf{P}_i^*$ of **P*** such that for all pairs of paths $(\mathbf{P}_i, \mathbf{P}_i^*)$, $\mathbf{Q}(\mathbf{P}_i) = \mathbf{Q}(\mathbf{P}_i^*)$, but that for some pairs $(\mathbf{P}_k, \mathbf{P}_k^*)$, $\mathbf{Z}(\mathbf{P}_k) \neq \mathbf{Z}(\mathbf{P}_k^*)$. For example, errors 1 and 2 in Figure 11 are domain errors.

This type of error exists because some statements that

```
procedure Ex3 is
 task ADD;
 task SUB;
 task DATA is
  entry GET_Y(V1 : in INTEGER);
  entry GET_Z(V2 : in INTEGER);
  entry READ_X(V3 : out INTEGER);
  entry WRITE_X(V4 : in INTEGER);
 end DATA;
task body ADD is
 X,Y:INTEGER;
 begin
  read Y;
  DATA.GET_Y(Y);
  DATA.READ_X(X);
  X := X + Y;
  DATA.WRITE_X(X);
 end ADD;
task body SUB is
 X,Z: INTEGER;
 begin
  read Z;
  DATA.GET_Z(Z);
  DATA_READ_X(X);
  X := X - Z;
  DATA_WRITE_X(X);
 end SUB;

begin
 null;
end EX2;

task body DATA is
  V1,V2,V3,X,Y,Z,TOTAL,RESULT:INTEGER;
 1  begin
 2   read X;
 3   accept GET_Y(V1 : in INTEGER) do
 4     Y := V1;
 5   end GET_Y;
 6   accept GET_Z(V2 : in INTEGER) do
 7     Z := V2;
 8   end GET_Z;
 9   if X > 0 and Y > 0 and Z > 0 then
10      RESULT := X + Y - Z;
<error2 : RESULT := X + 2*Y-Z>
11      for I in 1..4 loop
12        select
13          accept READ_X(V3: out INTEGER) do
14            V3 := X;
15          end READ_X;
16        or
17          accept WRITE_X(V4 : in INTEGER) do
18            X := V4;
19          end WRITE_X;
20        end select;
21      end loop;
22      if X /= RESULT then
<error 1 : if X < RESULT then>
<error 2 : if X < RESULT then>
23        write("race condition");
24      else
25        write("no race condition");
26      end if;
27   else
28     write("data error");
29   end if;
30  end DATA;
```

Figure 11. Program Ex3 and some errors

may affect the flow of control are expressed incorrectly or some rendezvous statements are placed in incorrect sequence. As a result, it may traverse a wrong C-path or may result in infeasible computation in the execution with a definite test. Note that the task sequencing error is clasified[15] as domain because C-routes are treated as input. An incorrect sequence corresponds to a shift of 'input' domain.

Let $\mathbf{P}_i$ be a C-path in $\mathbf{P}$ containing a domain error. The domain error of $\mathbf{P}_i$ can be further classified into three subclasses according to the difference between the hypothesis correct domain $\mathbf{D}_i^*$ and the implementation domain $\mathbf{D}_i$.

Definition 4

$\mathbf{P}_i$ is said to have a static domain error if $\mathbf{Z}_i^* \neq \mathbf{Z}_i$ and $\exists$ a definite test $(x,\delta) \in D_i^*$ but $\delta \notin \text{route}(\mathbf{P}_i)$.

This error represents that a C-route is feasible in $\mathbf{P}_i^*$ but is not a static feasible C-route of $\mathbf{P}$ and so called a static domain error.

Definition 5

$\mathbf{P}_i$ is said to have an input-dominated domain error if $\exists$ $(x,\delta) \in \mathbf{Z}_i^* - \mathbf{Z}_i$ then $\forall$ $(x,\delta') \in \mathbf{Z}^* => (x,\delta) \in \mathbf{Z}_i^* - \mathbf{Z}_i$ and if $\exists$ $(x,\delta) \in \mathbf{Z}_i - \mathbf{Z}_i^*$ then $\forall$ $(x,\delta') \in \mathbf{Z}_i => (x,\delta') \in \mathbf{Z}_i - \mathbf{Z}_i^*$. (Note, $\mathbf{Z}_i^* - \mathbf{Z}_i$ corresponds to set difference.)

Definition 6

$\mathbf{P}_i$ is said to have a rendezvous-dominated domain error if $\exists$ $(x,\delta) \in \mathbf{Z}_i^* - \mathbf{Z}_i$ then $\forall$ $(x',\delta) \in D^* => (x',\delta) \in \mathbf{Z}_i^* - \mathbf{Z}_i$ and if $\exists$ $(x,\delta) \in \mathbf{Z}_i - \mathbf{Z}_i^*$ then $\forall$ $(x',\delta) \in \mathbf{Z}_i => (x',\delta) \in \mathbf{Z}_i - \mathbf{Z}_i^*$.

Definition 7

$\mathbf{P}_i$ is said to have a race-domain error if, for some $(x,\delta) \in \mathbf{Z}_i^* - \mathbf{Z}_i$, $\exists$ both $(x,\delta') \in \mathbf{Z}_i - \mathbf{Z}_i^*$ and $(x',\delta) \in \mathbf{Z}_i - \mathbf{Z}_i^*$, and/or, for some $(y,\sigma) \in \mathbf{Z}_i - \mathbf{Z}_i^*$, $\exists$ both $(y,\sigma') \in \mathbf{Z}_i^* - \mathbf{Z}_i$ and $(y',\sigma) \in \mathbf{Z}_i^* - \mathbf{Z}_i$.

For example, error 3 in Figure 12 is an input-dominated domain error. Errors 1 and 2 in Figure 11 are rendezvous-dominated and race-domain errors, respectively.

Definition 8

A concurrent program $\mathbf{P}$ is said to have a computation error if there is an isomorphism between the C-path $\mathbf{P}_i$ of $\mathbf{P}$ and the C-path $\mathbf{P}_i^*$ of $\mathbf{P}^*$ such that for all pairs of paths $(\mathbf{P}_i,\mathbf{P}_i^*)$,$\mathbf{Z}(\mathbf{P}_i) = \mathbf{Z}(\mathbf{P}_i^*)$, but that for some pairs $(\mathbf{P}_k,\mathbf{P}_k^*)$,$\mathbf{Q}(\mathbf{P}_k) \neq \mathbf{Q}(\mathbf{P}_k^*)$.

A computation error exists because of the incorrect or mission actions that do not affect flow of control or the C-routes along $\mathbf{P}_i$. Let $\mathbf{P}_i$ be a C-path in $\mathbf{P}$ containing a domain error. As with domain error, the computation error of $\mathbf{P}_i$ can be further classified into three subclasses according to the difference between the hypothesis correct computation $\mathbf{Q}_i^*$ and the implementation computation $\mathbf{Q}_i$.

Definition 9

$\mathbf{P}_i$ is said to have an input-dominated computation error if $(x,\delta) \in \mathbf{Z}_i^*$, but $\mathbf{Q}_i(x,\delta) \neq \mathbf{Q}_i^*(x,\delta)$ then $\forall$ δ' $(x,\delta') \in \mathbf{Z}_i^*$ $=> \mathbf{Q}_i(x,\delta') \neq \mathbf{Q}_i(x,\delta')$.

```
procedure Ex4 is
 task T1;
 task T2;
 task T3 is
  entry E1(V1 : in INTEGER);
  entry E2(V2 : in INTEGER);
 end T3;

task body T1 is
 N : INTEGER;
 begin
  read N;
  T3.E2(N,1);
 end T1;
task body T2 is
 M,L : INTEGER;
 begin
  read M,L;
  T3.E1(L);
  T3.E2(M,2);
 end T2;
begin
 null;
end EX3;

task body T3 is
 V1,V2,V3,V4,ID,TOTAL,RESULT : INTEGER;
 SUM,EXP : array(1..2) of INTEGER;
 begin
  accept E1(V1 : in INTEGER) do
   V3 := V1;
  end E1;
  for I in 1..2 loop
   accept E2(V2,TID : in INTEGER) do
    V4 := V2;
    ID := TID;
   end E2;
   SUM(ID) := V3 + V4;
<error 1 : SUM(I) := V3 + V4;>
   EXP(ID) := V4**ID;
<error 2 : EXP(I) := EXP(I)**I>
  end loop;
  if SUM(1) /= SUM(2) then
<error 3 : if SUM(1) > SUM(2)>
   TOTAL := SUM(1) + SUM(2) - V3;
<error 4 : TOTAL := SUM(1) + SUM(2)+V3>
   RESULT := 2 * EXP(1) + EXP(2);
   write(SUM(1),SUM2(2),TOTAL,RESULT);
  else
   write("input error");
  end if;
 end T3;
```

Figure 12. Program Ex4 and some errors

Definition 10

$\mathbf{P}$ is said to have a rendezvous-dominated computation error if $(x,\delta) \in \mathbf{Z}_i^*$, but $\mathbf{Q}_i(x,\delta) \neq \mathbf{Q}_i^*(x,\delta)$ then $\forall$ x', $(x',\delta) \in \mathbf{Z}_i^* => \mathbf{Q}_i(x',\delta) \neq \mathbf{Q}_i^*(x,\delta)$

Definition 11

$\mathbf{P}_i$ is said to have a race computation error if $\exists$ $(x,\delta),(x',\delta),(x,\sigma) \in \mathbf{Z}_i^*$ and $\mathbf{Q}_i(x,\delta) \neq \mathbf{Q}_i^*(x,\delta)$, but $\mathbf{Q}_i(x,\sigma) = \mathbf{Q}_i^*(x,\sigma)$ and $\mathbf{Q}_i(x',\delta) = \mathbf{Q}_i^*(x',\delta)$.

For example, errors 1, 2, and 4 in Figure 12 are rendezvous-dominated, race, and input-dominated computation errors, respectively.

Definition 12

A concurrent program **P** is said to have a missing path error if there is an isomorphism between the C-path $\mathbf{P}_i$ of **P** and a subset of the C-path $\mathbf{P}_i^*$ of $\mathbf{P}^*$ such that for all pairs of paths $(\mathbf{P}_i,\mathbf{P}_i^*)$,$\mathbf{Q}(\mathbf{P}_i)=\mathbf{Q}(\mathbf{P}_i^*)$ and $\mathbf{Z}(\mathbf{P}_i^*) \subset \mathbf{Z}(\mathbf{P}_i)$.

This type of error arises from failure to test a particular condition and hence results in the execution (or no execution) of inappropriate actions. Some C-routes may become infeasible and some input may traverse a wrong C-path, but the correct one does not exist.

Reliability analysis of path analysis testing

The reliability analysis is based on an idealized situation that it is possible to test every C-path in a concurrent program and every C-route along each C-path. In such a situation, generate the complete set of path domain $\{\mathbf{Z}\}^n_{i=1}$ for a concurrent program and then construct a (possibly infinite) set of tests by choosing a set of elements, which is a minimal set of definite tests to cover each distinct C-route, at random from each nonempty $\mathbf{Z}_i$. This strategy will be referred to as concurrent P-testing.

Definition 13

Suppose **P** is a concurrent program for computing a function F whose definite test domain is the set **Z**. Let $\boldsymbol{T} \subset \mathbf{Z}$. $\boldsymbol{T}$ is a reliable test set for **P** if $\mathbf{P}(x,\delta)=F(x,\delta)$ for all $(x,\delta) \in \boldsymbol{T} => P(x,\delta) = F(x,\delta)$ for all $(x,\delta) \in \mathbf{Z}$.

This definition is an extension of the definition of a reliable test set[2]. Based on this definition of a reliable test set, some theorems of Howden can also be extended for concurrent P-testing. Therefore, in the following presentation, only the theorems that are unique in concurrent P-testing are described.

Theorem 3

The rendezvous-dominated domain errors of a C-path $\mathbf{P}_i$ can be detected by testing each distinct C-route along $\mathbf{P}_i$ at least once.

Proof

Let $\mathbf{R}_{i1}, \mathbf{R}_{i2}, \ldots, \mathbf{R}_{ir}$ be the C-routes along $\mathbf{P}_i$. Let $(x_1,\mathbf{R}_{i1})$, $(x_2,\mathbf{R}_{i2}), \ldots, (x_r,\mathbf{R}_{ir})$ be r definite tests for each C-route along $\mathbf{P}_i$. According to the definition of a rendezvous-dominated domain error, $\mathbf{P}_i$ contains a rendezvous-dominated domain error if and only if some $1 \leqslant j \leqslant r$ such that $(x_j,\mathbf{R}_{ij}) \in \mathbf{Z}_i$, but $(x_j,\mathbf{R}_{ij}) \notin \mathbf{Z}_i^*$ as $(x_j, \mathbf{R}_{ij}) \notin \mathbf{Z}_i$, but $(x_j, \mathbf{R}_{ij}) \in \mathbf{Z}_j^*$. Thus if $(x_j, \mathbf{R}_{ij})$ is tested once, the rendezvous-dominated domain error will be detected.

Theorem 4

Concurrent P-testing is reliable for testing concurrent programs that contain only rendezvous-dominated domain errors.

Theorem 5

Concurrent P-testing is reliable for testing input-dominated domain errors in **P** if and only if there is a feasible C-path $\mathbf{P}_i$ such that $\mathbf{Z}_i \cap \mathbf{Z}_i^* = \phi$.

Proof

(1) $\mathbf{Z}_i \cap \mathbf{Z}_i^* = \phi$. On this condition, any definite test in $\mathbf{Z}_i$ can detect that $\mathbf{P}_i$ has a domain error. Thus concurrent P-testing is reliable for **P**.

(2) $\mathbf{Z}_i \cap \mathbf{Z}_i^* \neq \phi$. Let $\mathbf{R}_{i1}, \mathbf{R}_{i2}, \ldots, \mathbf{R}_{ir}$ be the C-routes along $\mathbf{P}_i$ and, $1 \leqslant j \leqslant r$, $(x_j,\mathbf{R}_{ij}) \in \mathbf{Z}_i \cap \mathbf{Z}_i^*$ be r definite tests selected by concurrent P-testing. Then test execution with these r tests will not detect the domain error of $\mathbf{P}_i$ and, as the domain error in $\mathbf{P}_i$ is an input-dominated domain error, these r tests can only determine the correctness of other tests, say (x,δ), where $x = x_j$, $1 \leqslant j \leqslant r$. But the correctness of the other test (x',δ), $x' \neq x_j$ $1 \leqslant j \leqslant r$, is still unknown. Thus concurrent P-testing is not reliable.

Theorem 6

The rendezvous-dominated computation error of a C-path **P** can be detected by testing each C-route along **P** at least once.

Proof

Let $\mathbf{R}_{i1}, \mathbf{R}_{i2}, \ldots, \mathbf{R}_{ir}$ be the C-routes along $\mathbf{P}_i$. Let $(x_1,\mathbf{R}_{i1})$, $(x_2)\mathbf{R}_{i2}), \ldots, (x_r,\mathbf{R}_{ir})$ be r definite tests for each C-route along $\mathbf{P}_i$. According to the definition of a rendezvous-dominated computation error, $\mathbf{P}_i$ contains a rendezvous-dominated computation error if and only if some $1 \leqslant j \leqslant r$ such that $(x_j,\mathbf{R}_{ij}) \in \mathbf{Z}_i$ but $\mathbf{Q}_i(x_j,\mathbf{R}_{ij}) \neq \mathbf{Q}_i^*(x_j,\mathbf{R}_{ij})$. Thus if each definite test $(x_j,\mathbf{R}_{ij})$ is tested once, the rendezvous-dominated computation error can be detected.

Theorem 7

Concurrent P-testing is reliable for programs that contain only rendezvous-dominated computation errors.

Theorem 8

Concurrent P-testing is reliable for testing input-dominated computation errors in **P** if and only if there is a feasible C-path $\mathbf{P}_i$ such that $\mathbf{Q}_i(t) \neq \mathbf{Q}_i^*(t)$ for all $t \in \mathbf{Z}_i$.

The above analysis has shown that concurrent P-testing is especially useful for testing concurrent programs that contain rendezvous-dominated domain (computation) errors. But, for other classes of errors, the reliability of concurrent P-testing is similar to P-testing for sequential programs.

CONCLUSIONS

This paper proposes a path analysis approach to concurrent program testing. A concurrent path model is presented to model the execution behaviour of concurrent programs. Flowgraph is used to model the static structure, and rendezvous graph to model the dynamic structure, of a concurrent program. Based on the model, the definite test for path analysis testing is defined, a path analysis testing methodology proposed, and examples given to detect buried errors with the methodology. Also, several practical issues of the approach are discussed.

Moreover, to investigate further the effectiveness of the proposed methodology, the possible errors of a concurrent program are classified into three broad classes, namely, domain error, computation error, and missing path error, and the potential reliability to detect each kind of errors discussed. It is proved that some rendezvous-sensitive errors can be reliably detected by the proposed methodology in an idealized situation.

Path testing is the most well known and practical approach to the testing of sequential programs. The related issues of path testing, the theoretical basis, coverage criteria, testing tools, etc. have been studied for many years[1,2]. The result presented in this paper has shown a bright start to use experience for the testing of concurrent programs.

With the increasing complexity of computer software, any testing methodology without tool support must tend to be horrible. To this end, the authors are implementing a testing environment to support testers in practising the ideas described here. The environment includes a tool to generate C-paths in accordance with coverage criteria such as branch coverage and rendezvous coverage[18], a static analyser[19] for analysing the C-routes along the generated C-paths, and an execution control mechanism[20] for controlling and monitoring C-route traversed in the execution.

REFERENCES

1 **Goodenough, J B and Gerhart, S L** 'Toward a theory of test data selection' *IEEE Trans. Soft. Eng.* Vol 1 No 2 (June 1975) pp 156–173

2 **Howden, W E** 'Reliability of the path analysis testing strategy' *IEEE Trans. Soft. Eng.* Vol 2 No 2 (September 1976) pp 208–215

3 **Gait, J** 'A probe effect in concurrent programs" *Soft. Pract. Exper.* (March 1986) pp 225–233

4 **German, S M** 'Monitoring for deadlock and blocking in Ada tasking' *IEEE Trans. Soft. Eng.* Vol 10 No 6 (November 1984) pp 764–777

5 **Tai, K C** 'On testing concurrent programs' in *Proc. Compsac 85* (October 1985) pp 310–317

6 **Tai, K C and Obaid, E E** 'Reproducible testing of Ada tasking programs' in *Proc. IEEE 2nd Int. Conf. Ada Applications and Environment* (April 1986) pp 69–79

7 **Taylor, R N and Kelly, C D** 'Structural testing of concurrent programs' in *Proc. Workshop on Software Testing* Banff, Canada (July 1986) pp 164–169

8 **Brinch Hansen, P** 'Testing a multiprogramming system' *Soft. Pract. Exper.* Vol 3 (1973) pp 145–150

9 **Brinch Hansen, P** 'Reproducible testing of monitors' *Soft. Pract. Exper.* Vol 8 (1978) pp 721–729

10 **Tai, K C** 'Reproducible testing of concurrent Ada programs' in *Proc. SOFTFAIR II, Second Conf. Software Development Tools, Techniques, and Alternatives* (December 1985) pp 114–120

11 **Tai, K C and Obaid, E E** 'Reproducible testing of Ada tasking programs' in *Proc. IEEE 2nd Int. Conf. Ada Applications and Environments* (April 1986)

12 **Taylor, R N** 'A general-purpose algorithm for analyzing concurrent programs' *Commun. ACM* Vol 26 No 5 (May 1983) pp 362–376

13 **Weiss, S N** 'A formal framework for the study of concurrent program testing' in *Proc. 2nd Workshop on Software Testing* Banff, Canada (July 1988) pp 106–113

14 **Yang, R D and Chung, C G** 'Testing concurrent Ada programs with reproducible test cases' in *Proc. Int. Conf. Software Engineering and Knowledge Engineering* Skokie, IL, USA (15 June 1989)

15 **Helmbold, D and Luckham, D** 'Debugging Ada tasking programs' *IEEE Software* (March 1985) pp 47–57

16 **Young, M and Taylor, R N** 'Combining static concurrency analysis with symbolic execution' in *Proc. Workshop on Software Testing* Banff, Canada (July 1986) pp 170–178

17 **Dillon, L K** 'An experience with two symbolic execution-based approaches to formal verification of Ada tasking programs' in *Proc. 2nd Workshop on Software Testing* Banff, Canada (July 1988) pp 114–122

18 **Yang, R D** 'A path analysis approach to concurrent program testing' *PhD dissertation* Computer Engineering Department, National Chiao Tung University, Taiwan, ROC (February 1990)

19 **Hu, H C** 'A static analyzer for validating concurrent Ada programs' *Masters thesis* NCTU, Taiwan, ROC (June 1989)

20 **Chia, C Y** 'An execution controller for concurrent Ada program with tracing and forcing functions' *Masters thesis* NCTU, Taiwan, ROC (June 1989)