1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

TESTING AND METRICS FOR CONCURRENT COMPUTATION BASED ON RENDEZVOUS MECHANISM

Ying-Hong Wang, Chi-Ming Chung, Huan-Chao Keh, and Chia-Yu Chien

Graduate Institute of Information Engineering, TamKang University
Tamsui, Taipei Hsien, Taiwan, R.O.C.
e-mail: inhon@mail.tku.edu.tw

Abstract

Testing and metrics are two important approaches to assure the reliability and quality of software industries. Testing and metrics of sequential programs have been fairly sophisticated processes, with various methodologies and tools available for use in building and demonstrating the correctness of a program. The emergence of concurrent programming in the recent years, however, introduces new testing problems and difficulties that cannot be solved by testing techniques of traditional sequential programs. One of the difficult tasks is that concurrent programs can have many instances of execution for the same set of input data. Many testing methodologies of concurrent program are proposed to solve controlled execution and determinism. However, there are few discussions of concurrent software testing from an inter-task perspective. Yet, the common characteristics of concurrent programming are explicit identification of the large grain parallel computation units (tasks), and the explicit inter-task communication via a rendezvous-style mechanism. This paper focuses testing on the concurrent programming through a task decomposition mechanism. Four testing criteria are proposed to test a concurrent program. Programmers can choose an appropriate testing strategy depending on the properties of the concurrent programs. Associated with the strategies, four equations are provided to measure the complexity of the concurrent programs.

Index Terms: Concurrent programs, software testing criterion, software complexity, Ada language, rendezvous.

1. Introduction

Software testing and metrics are very important techniques in a software development life cycle (SDLC). The purposes of software testing and metrics are in the assurance of software quality and software correctness. Testing and metrics techniques of sequential programs are fairly mature and have various methodologies and tools available for usage. In the past decade, testing issues of concurrent programming are discussed widely which result in many new problems that cannot be solved by traditional debugging techniques of sequential programming. In this paper, we discuss the testing problems of concurrent programs and propose new testing strategies from an inter-tasks perspective.

Concurrent programs consist of components that can be executed in parallel. The ability of concurrent programs has many advantages such as reducing program time, leading to notational convenience and conceptual elegance in writing operating systems, real-time systems, database systems, and simulation programs, ...etc. [5]. However, testing concurrent programs is difficult. Due to the nondeterminism, concurrent programs can result in many instances of an execution for the same set of input data. Although repeated execution of a nondeterministic concurrent program is possible, it is still not sufficient to investigate all such instances of execution. A worse case scenario shows that a fault occurs in only one instance of an execution; but that instance of execution is never tested. Thus, any realistic parallel program testing method must be able to investigate more than one instance of executions corresponding to the input data set of a possible fault.

Testing of sequential programs has been established as a fairly sophisticated process, with various methodologies and tools available for being used in building and demonstrating confidence in the program being tested. The emergence of concurrent programming in recent years [7; 14], however, has presented new testing problems and difficulties which cannot be solved by regular sequential testing techniques [12; 13]. Many testing strategies are proposed based on different techniques; but shortcomings exist. We describe the related research in the next section. However, there are few investigations discuss concurrent software testing from an inter-task perspective.

The common characteristics of concurrent programming are the explicit identification of large grain parallel computation units (tasks), and the explicit inter-task communication via a rendezvous-style mechanism. Existing concurrent programming languages supply these capacities (e.g., HAL/S [12], CSP [6], Ada, and PCF FORTRAN [10], etc.). Excluding the talent of inter-task communication, each parallel computation unit of a concurrent program has the same structure as a sequential program. To provide a specific basis for the further discussions, we choose Ada for our example. Although, the results are applicable to any programs that use

rendezvous-like synchronization. Ada allows the specification and simultaneous execution of a number of tasks. The means for task synchronization and primary method of inter-task communication is a ***rendezvous***. The rendezvous concept combines process synchronization and communication [1, 4]. Two processes interact by first synchronizing, then exchanging information, before continuing to perform their individual activities. This synchronization or communication to exchange information is called the rendezvous [5]. Thus we focus software testing in rendezvous of concurrent programs. We propose some testing strategies and metrics based on rendezvous. To provide a focus, the discussion in the rest of this paper will be with respect to Ada. Moreover, we assume that variables are not shared by different tasks of concurrent units.

The rest of this paper is organized as follows. Section 2 introduces a survey of concurrent programming testing. In section 3, we propose four testing criteria based on rendezvous. Section 4 presents the coverage criterion hierarchy and some related proofs. Considering rendezvous, we further propose four equations to measure the complexity of concurrent programs. The equations are presented in section 5. Section 6 concludes the paper and describes our future works.

2. The Background of Concurrent Program Testing

The testing methodologies of concurrent programs are widely proposed within the last few years. The testing strategies of concurrent programs can be divided into four categories [3], as specified below.

The first is static analysis. Taylor et al. propose a structural (or white-box) testing method [13]. This technique applies the traditional structural testing strategies to concurrent programs. The authors focus the discussion on Ada programs. Each program unit (subprogram, task, package, or generic) defines a flow graph. Each statement in the unit is represented by a node; and each transfer of control is represented by a directed edge in the graph. Weiss obtains another approach by considering a concurrent program as a set of sequential programs [15].

The second technique is based on deterministic execution. Tai, Carver and Obaid propose a deterministic execution technique to debug concurrent Ada programs [11]. The proposed strategy is primarily to solve the following problem. When debugging an erroneous execution of **P** with input **X**, there is no guarantee that this execution will be repeated by executing **P** with input **X**.

Another technique is based on execution traces. A mechanism for *noninterference monitoring* and reproduction of a program behavior of real-time software systems is proposed by Tsai et al. [14]. This mechanism uses the recorded execution history of a program to control the replay of the program and guarantees the reproduction of its errors. The principal objective is to develop a "noninterference" software testing and a debugging system. It ensures a minimum intervention with the execution of a target system, while providing users with a comprehensive testing and debugging environment.

Yet another technique based on *Petri nets* is proposed by Morasca and Pezze [8]. However, its shortcoming is practically infeasible for large programs.

The last technique is based on controlled execution. Damodaran-Kamal and Francioni proposed a theory for testing *nondeterminacy in message passing programs* that is based on *controlled execution* with permuted delivery of messages [2]. Generally, any nondeterminacy detection strategy is intrusive when it requires instrumentation of the code. In a controlled execution, the delivery of messages sent to a process and sent by a process is regulated to control the execution of the process. A controlled execution permits experimentation with different race scenarios via permuting the order of delivery of messages at a receiver.

3. A Rendezvous Oriented Testing for Concurrent Programs

3.1 The Principles of Rendezvous in Ada

Rendezvous in the Ada programming language is implemented by "*entry*" call and "*accept*" the entry call. Tasks contain entries which are called by other tasks for synchronization and communication. Two tasks synchronize when the calling task makes an entry call and the called task accepts the entry call rendezvous. The synchronization rules in Ada are following. Suppose two tasks, A and B, need to synchronize or exchange information. Task A, which calls an entry of task B, will wait if B is not ready to accept the entry call. This entry call will be queued. If A does not want to wait, then A can use a facility that allows the entry call to be withdrawn if it cannot be accepted immediately. Alternatively, A can elect to wait for a specified time period for B to accept the entry call before withdrawing the entry call. When A is waiting for a result of making an entry call and B becomes ready, then a rendezvous occurs between A and B at the called entry. During the rendezvous, task A (the calling task) is suspended while B (the called task) continues its execution. B presumably records the information sent to it by A or returns information to A. Both A and B resume execution in parallel at the end of the rendezvous. Tasks can communicate during a rendezvous. This communication may be bi-directional and take place using entry arguments and the corresponding parameters of the accept statement corresponding to the entry call. If several tasks call the same entry of a task, then the calling tasks will rendezvous with the called task in the order in which the calls are received by the called task, i.e., in the **FIFO** order. Example 1 is illustrated the rendezvous mechanism of Ada [5]. There are two tasks, *PRODUCER* and *CONSUMER*. The PRODUCER will continue reading input from keyboard and sending each character to the CONSUMER. The CONSUMER will translate all lower-case characters to upper-case characters, and then prints

the characters. This is shown in below.

Example 1:
The specifications of the two tasks are:

```
task PRODUCER;

task CONSUMER is
    entry RECEIVE(C: character);
end CONSUMER;
```

The bodies of the two tasks are shown as follows:

```
task body PRODUCER is
    C: character;
begin
  while not END_OF_FILE(STANDARD_INPUT) loop
        if END_OF_LINE(STANDARD_INPUT) then
            CONSUMER.RECEIVE(ASCII.LF);
        end if;
        GET(C); CONSUMER.RECEIVE(C);
    end loop;
    CONSUMER.RECEIVE(ASCII.LF);
end PRODUCER;


task body CONSUMER is
    X: character;
begin
    loop
        accept RECEIVE(C: character) do
          -- names of calling tasks are not specified
              X := C;    -- value of C stored in X
        end RECEIVE;
        if X = ASCII.LF then NEW_LINE;
        else PUT(UPPER(X));
        end if;
    end loop;
end CONSUMER;
```

In the body of CONSUMER, the statements from "accept RECEIVE" to "end RECEIVE" is called a block of accept statement.

3.2 Rendezvous Testing Criteria

In this section, we will discuss the basic type of rendezvous in Ada and how to test it completely. Generally, a space-time diagram, shown in figure 3-1, is a convenient form to represent a parallel execution. In the space-time diagram, time flows from top to bottom. The vertical lines represent different processes. And the diagonal arrows represent message passing, i.e., rendezvous. However, it cannot represent multiple entry acceptance statements of rendezvous. This is one of the nondeterministic types.

In this paper, we will use a modified space-time diagram [16] to show the types of rendezvous. We introduce a circle on the time flow to represent an entry call or entry acceptance statement and label the entry name on the diagonal arrow to describe the occurring entry. The circles are divided into two classes: the entry call nodes and the entry acceptance nodes, marked as **EC** and **EA** respectively. The modified space-time diagram of example 1 is shown in figure 3-2.

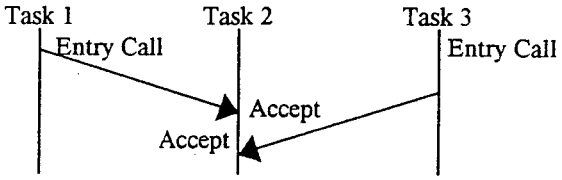


Fig. 3-1. A example of the space-time diagram

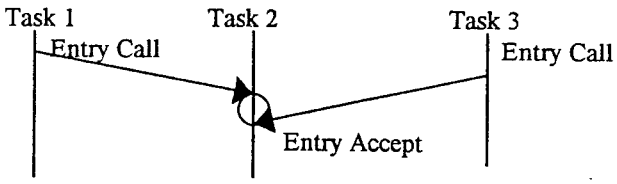


Fig. 3-2. A example of the modified space-time diagram

According to the principles of Ada rendezvous described in subsection 3.1, the basic type of rendezvous is a calling task invokes an entry call. And, when the called task accepts this entry call, the rendezvous is built. Therefore, the testing is complete when we execute all entry calls (i.e. all **EC** nodes) at least once. The first criterion, *All-EC criterion*, is defined to represent the requirement for the rendezvous testing.

Criterion 1. *All-EC criterion*:

> *All-EC criterion* is satisfied iff when all entry calls in an Ada program are tested at least once, i.e. each EC node of the modified space-time graph must be traced at least once.

One of the important characteristics of concurrent programs is *nondeterminacy*. Nondeterminacy happens when a concurrent/parallel program with the same input data yields different results on different runs. Any nondeterminacy in a concurrent/parallel program makes it difficult to detect the cause of program errors.

Ada programs allow a called task with multiple acceptance statements for the same entry. An example is extracted from [9] and shown as the following. There are two tasks: *A*, and *B*. Task *A* provides an entry **E** to accept the other tasks and processes the coming entry call. There are two accept statements in task *A* for entry **E**, labeled <L1> and <L2>, respectively. Task *B* will call entry **E**, labeled <L3>, for its entry call statement.

Example 2:
Tasks specifications are :

```
task A is
  entry E(x : in out integer);
end;
```

```
task B;
```

The tasks bodies are :

```
task body A is
  u, v : integer;
  begin
    ...
<L1>  accept E(x : in out integer) do
        x := x + u;
      end accept;
        ...
<L2>  accept E(x : in out integer) do
        x := x + v;
      end accept;
end A;

task body B is
 b : integer;
    ...
<L3>   A.E(b);
       PUT(b);
  ...
end B;
```

The modified space-time diagram is shown in figure 3-3. In this case, the *All-EC criterion* will be satisfied when entry call ***A*.E( )** in task ***B*** is executed once, e.g., (L3 vs. L1). However, it is not enough to cover all possible synchronization among tasks. Likewise, (L3 vs. L2) is lost. Thus, we propose the second criterion, the *All-Possible-EA* criterion.

Criterion 2. ***All-Possible-EA criterion***:
All-Possible-EA criterion is satisfied iff each entry call activates all corresponding entry acceptance statements at least once, i.e. each edge from an EC node to different EA nodes of the modified space-time graph must be traced at least once.

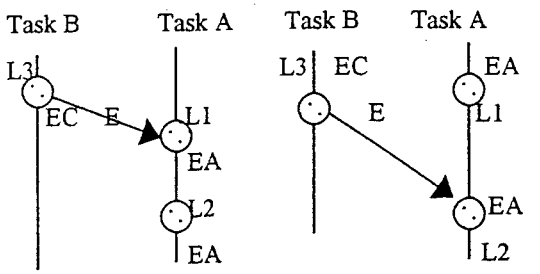
Another testing problem of concurrent programs is determining race. The races are behavior of nondetermination. A race occurs at an entry acceptance that contains at least two calls in its receiving queue. In Example 3, Task ***T*** is a monitor displayer that accepts and displays a message. Task ***B*** and ***C*** are two sensor receivers that obtain states from hardware devices and send them to Task ***T***. The modified space-time diagram is shown in figure 3-4.

Example 3:
The tasks specifications are :

```
task T is
  entry Display(m : in LINE);
end;

task B;

task C;
```

The tasks bodies are :

```
task body T is
  i:INTEGER
  begin
    ...
   accept Display(m : in LINE) do
     i := 1;
     loop
       display character m(i);
       exit when m(i) = LF;
       i := i + 1;
     end loop
  end accept;
    ...
end T;

task body B is
  L : LINE(1..254);
    ...
  T.Display(L);
    ...
end B;

task body C is
  X : LINE(1..254);
    ...
  T.Display(X);
    ...
end C;
```



Note: L1 and L2 are different entry acceptance, but they accept the same entry

Fig. 3-3. Two possible modified Space-time diagrams for Example 2

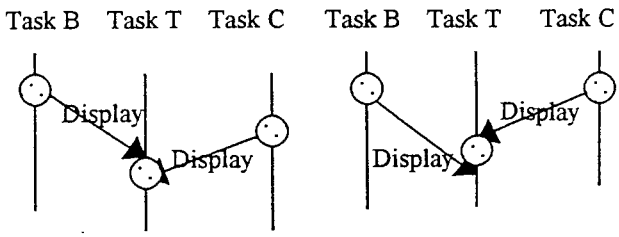


Fig. 3-4. Two possible modified space-time diagrams for Example 3

If we need to consider the ordering relationship, the race of messages displaying from Task *B* and Task *C* will occur. For testing races, we propose the third criterion, *All-EC-Permute* criterion.

Criterion 3. ***All-EC-Permutation criterion***:

All-EC-Permutation criterion is satisfied iff all possible permutations in the receiving queue of each entry acceptance are tested at least once, i.e., the permutation of all edges from different EC nodes to an EA node of the modified space-time graph must be traced at least once.

Thus, testing cases include not only **{(EC1, EA) and (EC2, EA)}** but also **{(EC2, EA) and (EC1, EA)}**, i.e., the number of test cases of an entry acceptance is the number of all possible entry calls ***permutations***.

Multiple tasks may send the same entry calls to a receiving task that has multiple entry acceptance statements of the same entry name. If the executed ordering among the entry calls and the activated acceptance statements are dependent, then the *All-EC-Permutation* criterion is not enough because it just tests the permutation of the individual entry acceptance. It cannot test the permuted relationship between different entry acceptance statements. Thus, we propose the fourth criterion to test the potential ordering-dependent permutation of all entry calls in all entry acceptance statements of the same entry name.

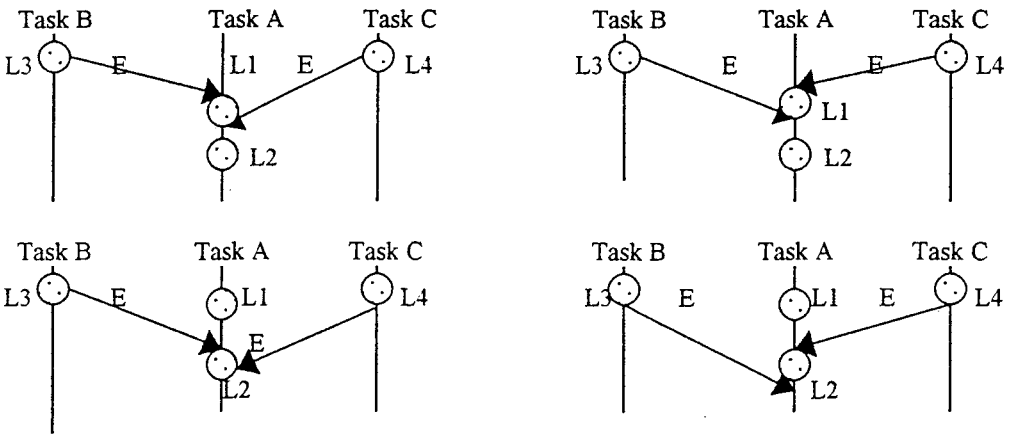
In Example 4, extended from Example 2, there is another Task *C* which also calls entry E (labeled <L4> for its entry call statement). Figure 3-5 depicts the possible modified space-time diagrams.

Example 4:
The tasks specifications are :

```
task A is
   entry E(x : in out integer);
end;

task B;

task C;
```

The tasks bodies are :

```
task body A is
   u, v : integer;
   begin
      ...
<L1>    accept E(x : in out integer) do
            x := x + u;
         end accept;
      ...
<L2>    accept E(x : in out integer) do
            x := x + v;
         end accept;
end A;

task body B is
   b : integer;
   ...
<L3>  A.E(b);
      PUT(b);
   ...
end B;

task body C is
   c : integer;
   ...
<L4>  A.E(c);
      PUT(c);
   ...
end C;
```



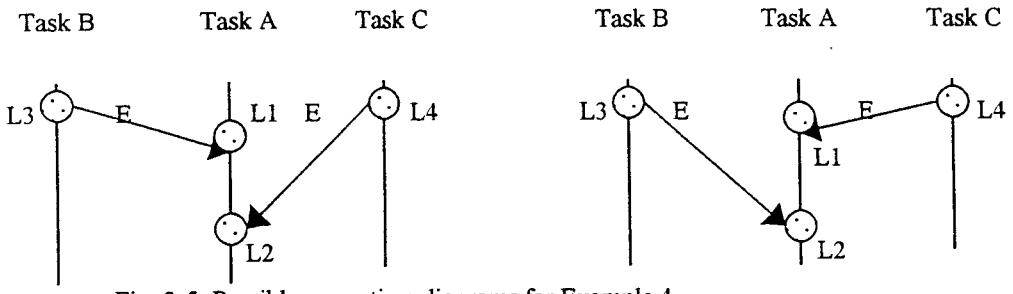


Fig. 3-5. Possible space-time diagrams for Example 4

The fourth criterion is described as the following.

Criterion 4. ***All-EC-Dependency-Permutation criterion***:
All-EC-Dependency-Permutation criterion is satisfied iff all possible permutations in received queue of all entry acceptance statements with the same entry name are tested at least once, i.e. the permutation of all edges from different EC nodes to each EA node with the same entry name of modified space-time graph must be traced at least once.

The test cases are: **{(L3, L1)-(L4, L1), (L4, L1)-(L3, L1), (L3, L2)-(L4, L2), (L4, L2)-(L3, L2), (L3, L1)-(L4, L2), (L4, L1)-(L3, L2)}**, i.e., the number of test cases is the summary of the permutations of all possible entry calls of individual entry acceptance plus the permutations of all possible entry calls in different entry acceptance statements.

4. Coverage Criteria Hierarchy

In this section, we present the coverage criteria hierarchy of the criteria proposed in section 3.2 and justify the coverage relationship among them. First, some definitions of terms used in the following theorems are presented.

M: denotes the total number of different entries in an Ada program.

$\mathbf{m_E}$: denotes the number of entry calls that call the same entry E from the same or different tasks in an Ada program.

$\mathbf{n_E}$: represents the number of entry acceptance that accept the same entry E in an Ada program.

S or S': means the set of tested rendezvous satisfying some criteria.

N or N': means the total number of elements of S or S', respectively.

strictly subsume: Criterion A *strictly subsumes* criterion B if the set of tested rendezvous that satisfies criterion A also satisfies criterion B, and the set of tested rendezvous that satisfies criterion B does not satisfy criterion A.

Theorem 1

All-EC-Dependency-Permutation criterion strictly subsumes All-EC-Permutation criterion.

proof:

For each entry E in an Ada program, let S is the set of All-EC-Dependency-Permutation criteria and S' be the set of All-EC-Permutation criteria.

Each acceptance statement of the entry E possibly has $\mathbf{m_E}$ synchronization. The number of permutations of all possible rendezvous at an acceptance statement is $\mathbf{P(m_E)}$, i.e., $\mathbf{m_E!}$, where $m_E! = m_E * (m_{E-1}) *...*2*1$. The number of acceptance statements of the entry E in an Ada program is $\mathbf{n_E}$, and the summation of permutation of individual entry acceptances with the same entry name is $\mathbf{(n_E * m_E)!}$. Furthermore, considering the ordering dependency of all entry calls in all entry acceptance statements of the same entry name, we get m_E acceptance nodes from $\mathbf{n_E}$ for permuting m_E entry calls. Thus, there are $\mathbf{C(n_E, m_E)* m_E!}$, where $C(n_E, m_E) = n_E!/((n_E-m_E)!(m_E!))$, permutations from n_E to choose $\mathbf{m_E}$. The total number of S is the permutations of all entry calls in each individual entry acceptance plus the dependent permutations of all entry calls in all entry acceptance statements, i.e., $N = \mathbf{n_E * m_E ! + C(n_E, m_E) * m_E!}$.

To satisfy All-EC-Permutation criterion, each acceptance statement of entry E possibly has m_E synchronization. The permutation of all possible rendezvous at an acceptance statement is $\mathbf{P(m_E)}$, i.e., $m_E!$. The number of acceptance statements of the entry E in an Ada program is $\mathbf{n_E}$. Therefore, the total number of S' is $n_E * m_E!$, i.e., $N' = \mathbf{n_E * m_E!}$.

Due to S and S' are the sets of tested rendezvous for the same entry, clearly S' is included in S. Therefore, All-EC-Dependency-Permutation criterion strictly subsumes All-EC-Permutation criterion. ■

Theorem 2

All-EC-Permutation criterion strictly subsumes All-Possible-EA criterion.

proof:

For each entry E in an Ada program, let S is the set of All-EC-Permutation criteria and S' be the set of All-Possible-EA criteria.

According to Theorem 1, the total number of S is $n_E * m_E!$, i.e., $N = \mathbf{n_E * m_E!}$. To satisfy All-Possible-EA criterion, each entry call statement of the entry E must

execute $\mathbf{n_E}$ times because there are n_E acceptance statements of the entry E. Since there are $\mathbf{m_E}$ entry call statements of the entry E, the total number of S' is $m_E * n_E$, i.e., $N' = \mathbf{m_E} * \mathbf{n_E}$.

Due to S and S' are the sets of tested rendezvous for the same entry, clearly S' is included in S. Therefore, All-EC-Permutation criterion strictly subsumes All-Possible-EA criterion. ■

Theorem 3

All-Possible-EA criterion strictly subsumes All-EC criterion.

proof:

For each entry E in an Ada program, let S is the set of All-Possible-EA criteria and S' be the set of All-EC criteria.

According to Theorem 2, the total number of S is $m_E * n_E$, i.e., $N = \mathbf{m_E} * \mathbf{n_E}$. To satisfy All-EC criterion, each entry call statement of the entry E must execute at least once, the total number of S' is $\mathbf{m_E}$, i.e., $N' = \mathbf{m_E}$.

Due to S and S' are the sets of tested rendezvous of the same entry, clearly S' is included in S. Therefore, All-Possible-EA criterion strictly subsumes All-EC criterion. ■

The coverage criteria hierarchy is shown in figure 4-1.

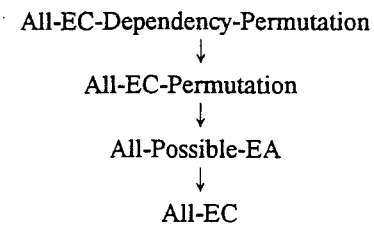


Fig. 4-1. Rendezvous-based testing coverage criteria hierarchy

5. Software Metrics for Concurrent Programs Based on Rendezvous

We propose a new mechanism to measure the complexity of a concurrent program. As mentioned in previous sections, synchronization and communication make the most distinct difference between concurrent programs and sequential programs. The complexity measurement of a concurrent program does not result without rendezvous. The number of rendezvous is an important factor to the complexity of the concurrent program. Therefore, the number of different entry, M, where each entry has m_E entry call statements and n_E entry acceptance statements, can be used to compare the complexity among concurrent programs. The first equation for measuring a concurrent program is the following:

Equation 1

$\mathbf{Cpx} = \Sigma^{\mathbf{M}}_{i=1} \mathbf{m_{Ei}}$, where *Cpx* means the complexity of a concurrent program. M and m_E are defined in the previous section, and index i (from 1 to M) represents each individual entry. ◆

The equation counts all entry calls instruction. This is the simplest case in which all entry call statements and entry acceptance statements are indicative. If different entry acceptance statements received the same entry (e.g., Example 2), we need to consider the possible rendezvous combination. Therefore, the second equation is presented as follows:

Equation 2

$\mathbf{Cpx} = \Sigma^{\mathbf{M}}_{i=1}(\mathbf{m_{Ei}} * \mathbf{n_{Ei}})$, where *Cpx* means the complexity of a concurrent program, and n_E is defined in the above section. ◆

However, the major characteristic of a concurrent program is race. Races make nondeterminism in concurrent programs and increase the difficulty in the testing task. According to Theorem 1, we can calculate the permutations of all rendezvous in an Ada program and the permutations of all rendezvous include all race cases. Thus, we propose the third equation to measure the complexity of an Ada program.

Equation 3

$\mathbf{Cpx} = \Sigma^{\mathbf{M}}_{i=1}(\mathbf{n_{Ei}} * \mathbf{m_{Ei}}!)$, where i means each individual entry, from 1 to M. ◆

When we consider the ordering dependency among entry calls, the third equation must be extended to the fourth equation as the following:

Equation 4

$\mathbf{Cpx} = \Sigma^{\mathbf{M}}_{i=1}((\mathbf{n_{Ei}} * \mathbf{m_{Ei}}!) + \mathbf{C}(\mathbf{n_{Ei}}, \mathbf{m_{Ei}}) * \mathbf{m_{Ei}}!)$,
where i means each individual entry, from 1 to M, and $\mathbf{C(x, y)}$ means combination, from x choosing y. ◆

According to these metrics equations, we make two suggestions to Ada programmers as the following:

(1) **Don't centralize all entry acceptance statements in few tasks:** This means that the loads of called tasks are heavy. Many tasks will send entry calls to the same entry acceptance of a called task. When we increase an entry call, the rendezvous complexity will increase tremendously.

(2) **Don't distribute acceptance statements to accept the same entry:** This means that there are many possibilities when a task sends an entry call. When we increase an entry acceptance statement to receive the same entry, the rendezvous will also increase tremendously.

When a concurrent program has the above two properties, the programmer is recommended to redesign

the program in order to decrease the complexity.

6. Conclusion and Future work

Recently, concurrent/parallel testing is emphasized tremendously. Testing concurrent/parallel programs is considered more difficult because concurrent programs are often nondeterministic. Thus, many concurrent testing strategies are proposed according to different properties of concurrent programs.

One major characteristic of concurrent programs compared with sequential program is rendezvous. We propose a rendezvous measurement mechanism for concurrent program testing. In our research, we present four testing criteria based on the rendezvous of concurrent/parallel programs. According to the analytic properties of entry calls and entry acceptances in tasks, programmers can choose an appropriate testing strategy to debug their concurrent programs. It is assumed that before performing concurrent test, thorough testing of the individual tasks has been done using sequential programming techniques. We also propose a coverage criteria hierarchy for the four criteria and prove the correctness of the coverage hierarchy. We provide four equations based on rendezvous to measure the software complexity of a concurrent/parallel program. Furthermore, we make two suggestions for concurrent programming based on rendezvous complexity.

As our future plan, we will consider the conjunction of rendezvous with other Ada instructions, such as *select*, *delay*, *selective-wait*, etc., and propose testing criteria to help software engineers for testing tasks. We will also extend the investigation to a general parallel programming language with explicit lexically-specified parallel constructs. We will then apply the technologies of program decomposition to conduct a quantitative analysis of the testing criteria and software metrics for concurrent/parallel programs. Finally, we will apply the methodology to other programming domains, e.g., event-driven programming, network programming, and object-oriented programming.

Acknowledges: We thank the National Scientific Committee (NSC), Taiwan, supports a project to this research, the project number is NSC 87-2213-E-032-009.

Reference

1. [Conway 63] M. E. Conway, "Design of a Separable Transition Diagram Compiler," CACM, pp. 396-408

2. [Damodaran 93] Suresh K. Damodaran-Kamal and Joan M. Francioni, "Nondeterminacy: Testing and Debugging in Message Passing Parallel Programs," ACM SIGPLAN Notices, pp. 118-128, Dec., 1993

3. [Damodaran 94] Suresh K. Damodaran-Kamal and Joan M. Francioni, "Testing Races in Parallel Programs with an OtOt Strategy," Proceeding of the 1994 International Symposium on Software Testing and Analysis(ISSA), also ACM Software Engineering Notices, special issue, pp. 216-227, Aug., 1994

4. [DoD 80] DoD, "Preliminary Ada Reference Manual," SIGPLAN Notices, Vol. 14, No. 6, Part A, Jun., 1980

5. [Gehani 91] Narain Gehani, "Ada: Concurrent Programming," 2nd Edition, AT&T Bell Lab., Silicon Press, 1991

6. [Hoare 78] C. A. R. Hoare, "Communicating Sequential Processing," Communication of ACM, Vol. 21, No. 8, pp. 666-677, Aug. 1978

7. [LeBlanc 87] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," IEEE Trans. on Computers, C-36, No. 4, pp. 471-482, Apr., 1987

8. [Morasca 90] S. Morasca and M. Peeze, "Using High Level Petri Nets for Testing Concurrent and Real Time Systems, " In Real-Time Systems: Theory and Application, pp. 119-131, 1990

9. [Moser 90] Louise E. Moser, "Data Dependency Graphs for Ada Programs," IEEE Trans. on Software Engineering, Vol. 16, No. 5, pp. 498-509, May, 1990

10. [PCF 91] Parallel Computing Forum, "PCF Parallel FORTRAN extension," FORTRAM Forum, vol. 10, No. 3, special issue, Sept. 1991

11. [Tai 91] K. C. Tai, R. H. Carver, and E. E. Obaid, "Debugging Concurrent Ada Programs by Deterministic Execution," IEEE Trans. on Software Eng., Vol. 17, No. 1, pp. 45-63, Jan. 1991

12. [Taylor 83] R. N. Taylor, "A General Purpose Algorithm for Analyzing Concurrent Programs," CACM, pp. 362-376, May, 1983

13. [Taylor 92] R. N. Taylor, D. L. Levine and C. D. Kelly, "Structural testing of Concurrent Programs," IEEE Trans. on Software Eng., Vol. 8, No. 3, pp. 206-215, March, 1992

14. [Tsai 90] J. J. Tsai, K. Y. Fang H. Y. Chen and Y. D. Bi, "A Noninterference Monitoring and Replay Mechanism for Real-time Software Testing and Debugging," IEEE Trans. on Software Eng., Vol. 16, No. 8, pp. 897-915, Aug., 1990

15. [Weiss 88] S. Weiss "A Formal Framework for The Study of Concurrent Program Testing," In Proceedings of the 2nd Workshop on Software Testing, Analysis, and Verfication, pp. 106-113, July, 1988

16. [Wang 97] Ying-Hong Wang, Timothy K. Shih, Chi-Ming Chung, Jui-Fa Chen, and Wei-Chuan Lin, March. 1-5 1997, NC, U.S.A. "Concurrent Software Testing usingTask Decomposition Mechanism," Third Joint Conference on Information Sciences, pp. 53-56