Task Decomposition Testing and Metrics for Concurrent Programs

Chi-Ming Chung, Timothy K. Shih, Ying-Hong Wang, Wei-Chuan Lin, and Ying-Feng Kou

Graduate Institute of Information Engineering, TamKang University
Tamsui, Taipei Hsien, Taiwan, R.O.C.

Abstract

Software testing and metrics are two important approaches to assure the reliability and quality of software. Testing and metrics of sequential programs have been a fairly sophisticated process, with various methodologies and tools available for use in building and demonstrating the correctness of a program being tested. 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 concurrent program testing methodologies are proposed to solve controlled execution and determinism. There are few discussions of concurrent software testing from the inter-task viewpoints. 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. In this paper, we focus the testing view on the concurrent programming through task decomposition. We propose four testing criteria to test a concurrent program. Programmer can choose an appropriate testing strategy depending on the properties of concurrent programs. Associated with the strategies, four equations are provided to measure the complexity of 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 software development life cycle. The purposes of software testing and metrics are 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 use. In the past decade, the testing issues of concurrent programming are discussed more and more, generating many new problems that cannot be solved by traditional debugging techniques of sequential programming. In this paper, we will discuss the testing problems of concurrent programs and propose new testing strategies focused on an inter-tasks view.

Concurrent programs are programs with components that can be executed in parallel. The ability to write concurrent programs has many advantages [5]. However, the testing tasks of concurrent programs are difficult. Due to nondeterminism, concurrent programs can result in many instances of 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 is that a fault occurs in only one instance of execution, and that instance of execution is never tested. Thus, any realistic parallel program testing method must be able to investigate more than one instance of execution corresponding to an input data set for a possible fault.

Testing of sequential programs has been established as a fairly sophisticated process, with various methodologies and tools available for use in building and demonstrating confidence in the program being tested. The emergence of concurrent programming in recent years [7][13], however, has presented new testing problems and difficulties which cannot be solved by regular sequential program testing techniques [11][12]. Many testing strategies are proposed based on different techniques and some shortcomings exist. We will describe the related research in the next section. However, there are few investigations to discuss concurrent software testing from an inter-task viewpoint.

The common characteristics of concurrent programming are explicit identification of large grain parallel computation units (tasks), and explicit inter-task communication via a rendezvous-style mechanism. Existing concurrent programming languages supply these capacities, such as HAL/S [11], CSP [6], Ada, and PCF FORTRAN [9], etc. Excluding the talent of inter-task communication, each parallel computation unit of a concurrent program has the same structure as sequential programs. To provide a specific basis for the further discussions, we choose Ada as our description sample, although the results are applicable to any programs that use rendezvous-like synchronization. Ada allows the specification and simultaneous execution of any 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 the rendezvous for concurrent programs. We will propose some testing strategies and metrics based on the rendezvous. To provide a focus, the discussion in the remainder of this paper will be with respect to Ada, and we assume that variables are not shared by different tasks in concurrent units.

The remainder 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 view. Section 4 presents the coverage criterion hierarchy and related proof simultaneously. Considering the rendezvous, we further propose four equations to measure the complexity of concurrent programs. They are presented in section 5. Section 6 concludes the paper and describes our plans for future work.

2. The Background of Concurrent Program Testing

The testing methodologies of concurrent programs are proposed more and more in recently years. The existing testing strategies of concurrent programs can be divided into four techniques [3].

The first one is static analysis. Taylor et al. propose a structural or white-box testing method [12]. 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 in the graph, and each transfer of control is represented by a directed edge. Weiss obtain another approach towards testing by considering a concurrent program as a set of sequential program [14].

The second technique is testing based on deterministic execution. Tai, Carver and Obaid propose a deterministic execution technique to debug concurrent Ada programs [10]. 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 testing 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. [13]. This mechanism uses the recorded execution history of a program to control the replay of the program behavior and guarantees the reproduction of its errors. The principal objective is to develop a "noninterference" software testing and debugging system to ensure 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 [15]. Its shortcoming is practically infeasible for large programs.

The last technique is testing based on controlled execution. Damodaran-Kamal and Francioni have proposed a theory for testing *nondeterminacy in message passing programs* that is based on *controlled execution* with permuted delivery of messages [2]. In general, any nondeterminacy detection strategy is intrusive when it requires instrumentation of the code. In controlled execution, the delivery of messages sent to a process and sent by a process is regulated to control the execution of the process. 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

The rendezvous in 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:

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 a specified time period for *B* to accept the entry call before withdrawing the entry call. We will discuss this in a later section. When *A* is waiting as 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 execution, presumably to record the information sent to it by *A* or to return information to *A*. They both 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 in 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 FIFO order. For instance in [5], there are two tasks, PRODUCER and CONSUMER. The PRODUCER will continue reading input from keyboard and sending each character to CONSUMER. The CONSUMER will translate all lower-case characters to upper-case characters, and then prints the characters. This is shown in Example 1 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 passings, here called rendezvous. However, it cannot represent multiple entry acceptance statements of rendezvous.

In this paper, we will use a modified space-time diagram to show the types of rendezvous. We append 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: entry call node and entry acceptance node, marked as **EC** and **EA** respectively.

In Ada programming, the rendezvous is implemented by *entry call* and *accept* among tasks. In Ada programming, there are three basic rendezvous types:

(a) **Simple rendezvous**: it is consisted of one EC node, one EA node and one edge connecting them, as shown in figure 3-2.

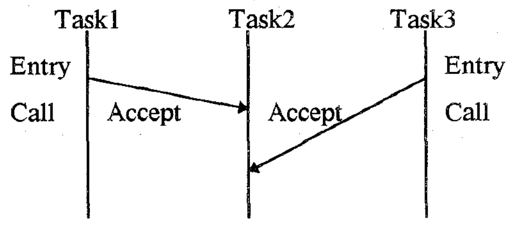

Fig. 3-1. An example of space-time diagram

(b) **Nested calling rendezvous**: the block of an acceptance statement includes other entry call statements, shown in figure 3-3. We call the node of entry acceptance EAg to mean including entry calling in the block of acceptance statement, and use a concentric circle instead of a single circle.

(c) **Nested called rendezvous**: the block of an acceptance statement includes other entry acceptance statements. It is shown in figure 3-4. The concentric circle is marked EAd to mean including entry called in the block of acceptance statement.

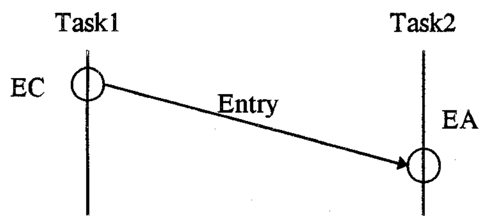

Fig. 3-2. Space-time diagram of simple

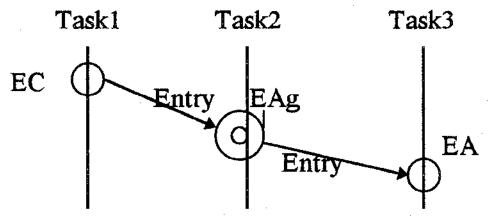

Fig. 3-3. Space-time diagram of nested calling rendezvous

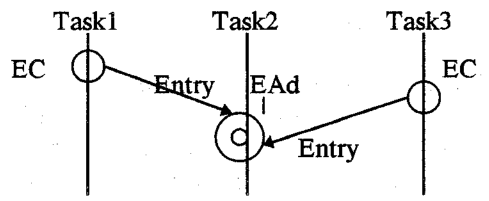

Fig. 3-4. Space-time diagram of nested called rendezvous

Clearly, no matter what types of rendezvous, the entry calls still keep the ordering. Therefore, the testing is complete when we execute all entry calls (i.e. all EC nodes) at least once. We define a criterion, *All-EC criterion*, 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 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. For example reduced from [8] is the following:

As shown in Example 2, there are two tasks: *A*, and *B*. Task *A* provides an entry E to accept the other tasks and process the coming entry call. There are two accept statements in task *A* for entry E, labeled <L1> and <L2>, respectively. The task *B* will call entry E, labeled <L3> for its entry call statement.

Example 2:

The 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 3-5. 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 for covering all possible synchronizations among tasks, likewise (L3 vs. L2) is lost. Thus, we propose the second criterion, *All-Possible-EA* criterion .

Criterion 2. ***All-Possible-EA criterion***:

All-Possible-EA criterion is satisfied iff each entry call must call all same 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.

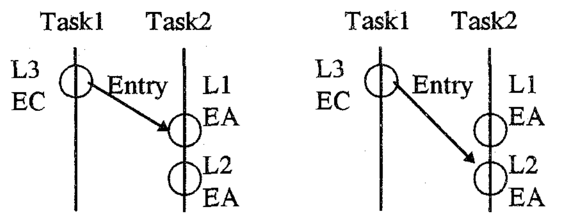

Fig. 3-5. Two possible modified space-time diagram for Example 2. Note: The L1 and L2 are different entry accepts, but they accept same entry.

Another testing problem of concurrent programs determining race. The races are one behavior of nondetermination. A race occurs at an entry acceptance that contain at least two calls in its received queue. For Example 3, Task *T* is a monitor displayer that accepts a message and displays it. Task *B* and *C* are two sensor receivers that get states from hardware and send them to Task *T*. Their modified space-time diagram is shown in figure 3-6. The tasks can be abstractly described in Example 3 below.

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;
```

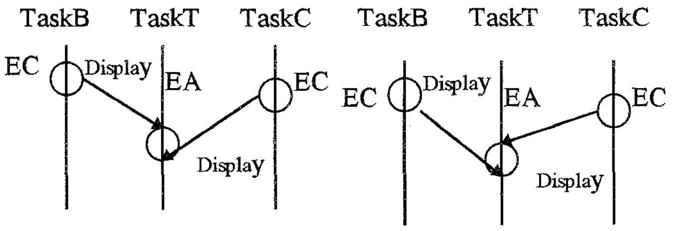

Fig. 3-6. Two possible modified space-time diagram for Example 3

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 received 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 modified space-time graph must be traced at least once.

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

Many tasks may have sent the same entry calls to a received task that has multiple entry acceptance statements for the same entry name. If the executed ordering among the entry calls and the happened acceptance statements are dependent, then the All-EC-Permutation criterion is not enough because it just tests the permutation of 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 with 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 3-7 depicts their 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;
begin
       ...
 <L3>   A.E(b);
        PUT(b);
       ...
end B;

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

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) and (L4, L1), (L3, L1) and (L3, L2), (L4, L2) and (L4, L2), (L3, L2) and (L3, L1), (L4, L2) and (L4, L1), (L3, L2)}, i.e., the number of test cases is the summary of the permutations of all possible entry call of individual entry acceptance plus the permutations of all possible entry calls in different entry acceptance statements.

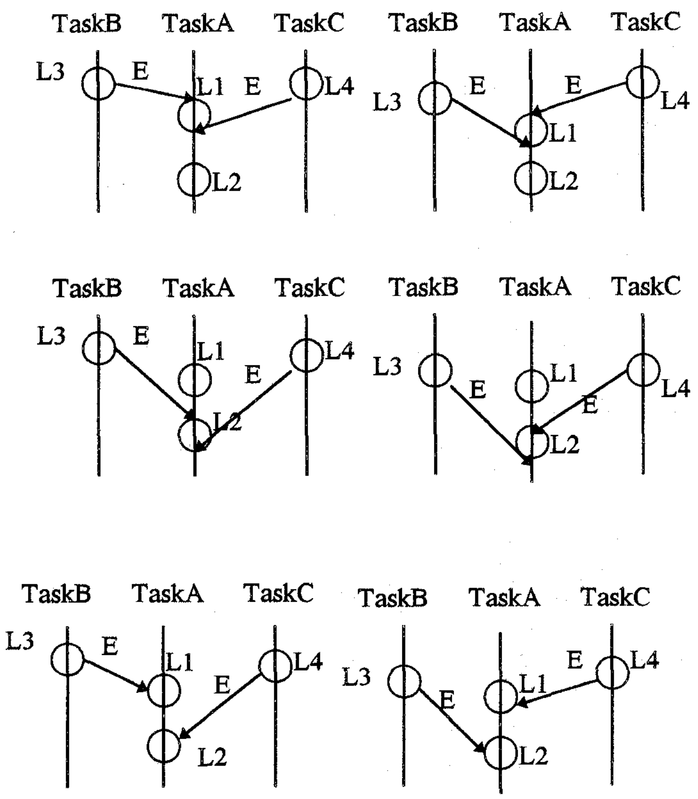

Fig. 3-7. Possible modified space-time diagram for Example 4

4. Coverage Criteria Hierarchy

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

M: The notation M means the total number of different entries in an Ada program.

m_E: The notation m_E means the number of entry calls that call the same entry E from the same or different tasks in an Ada program.
n_E: The notation n_E means the number of entry acceptance that accept the same entry E in an Ada program.
S or S': The notation S (or S') means the set of tested rendezvous satisfying some criterion.
N or N': The notation N (or N') means the total number of elements of S (or S').
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 be the set of All-EC-Dependency-Permutation criterion and S' be the set of All-EC-Permutation criterion.

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

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

Due to S and S' are the set of tested rendezvous for the same entry, it is clear that 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 be the set of All-EC-Permutation criterion and S' be the set of All-Possible-EA criterion.

According to Theorem 1, the total number of S is $n_E * m_E!$, i.e., $N = n_E * m_E!$. To satisfy All-Possible-EA criterion, each entry call statement of the entry E must execute n_E times because there are n_E acceptance statements of the entry E. Since there are m_E entry call statements of the entry E, the total number of S' is $m_E * n_E$, i.e., $N' = m_E * n_E$.

Due to S and S' are the set of tested rendezvous for the same entry, it is clear that 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 be the set of All-Possible-EA criterion and S' be the set of All-EC criterion.

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

Due to S and S' are the set of tested rendezvous for the same entry, it is clear that 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.

$$\begin{tikzcd}
\text{All-EC-Dependency-Permutation} \arrow[d] \\
\text{All-EC-Permutation} \arrow[d] \\
\text{All-Possible-EA} \arrow[d] \\
\text{All-EC}
\end{tikzcd}$$

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

5. Software Metrics for Concurrent Programs Based on Rendezvous

Finally, we propose a new view to measure the complexity of a concurrent program. As mentioned above, synchronization and communication are the major differences between concurrent programs and sequential programs. The complexity measurement of a concurrent program is also emphasized in the rendezvous. The number of rendezvous is naturally an important factor for 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

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

The equation counts all entry calls instruction. This is the most simple case in which all entry call statements and entry acceptance statements are one-to-one mapping. If different entry acceptance statements received the same entry, like Example 2, we need to consider the possible rendezvous combination. Therefore, the second equation is presented as follows:

Equation 2

$\mathbf{Cpx} = \Sigma^{M}_{i=1}(\mathbf{m_{Ei}} * \mathbf{n_{Ei}})$, 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. The races make nondeterminism in concurrent programs and increase the difficuly in the testing task. According to the proof of 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

$\mathbf{Cpx} = \Sigma^{M}_{i=1}(\mathbf{n_{Ei}} * \mathbf{m_{Ei}}\ !)$, 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

$\mathbf{Cpx} = \Sigma^{M}_{i=1}((\mathbf{n_{Ei}} * \mathbf{m_{Ei}}\ !) + \mathbf{C(n_{Ei}, m_{Ei}) * m_{Ei}!})$,

where i means each individual entry, from 1 to M, and C(x, y) means combination, from x choosing y.

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

(1) Don't centralize all entry acceptance statements in few tasks: It means the load 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: It means 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, it would be advised for redesign to decrease complexity.

6. Conclusion and Future work

Recently, concurrent/parallel programming testing is emphasized increasingly. Testing concurrent/parallel programs is considerably more difficult because concurrent programs are often not deterministic. Thus, many concurrent programming 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 point of view for concurrent program testing. In our research, we present four testing criteria based on the rendezvous for concurrent/parallel programs. According to the analytic property of entry call and entry acceptance in the tasks, programmers can choose an appropriate testing strategy to debug their concurrent programs. We also propose a coverage criteria hierarchy for the four criteria and prove the correctness of the coverage hierarchy. At the same time, we provide four equations based on the rendezvous to measure software complexity of a concurrent/parallel program. Furthermore, we make two suggestions for concurrent programming based on rendezvous complexity.

In future related work, we will consider the conjunction of rendezvous with other Ada instructions, such as *select*, *delay*, *selective-wait*, etc., and propose more testing criteria to help software engineers for testing tasks. We will also extend the investigation to 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 similar programming environments, e.g., event-driven programming, network programming and object-oriented programming.

Reference

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

2. 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. 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, "Preliminary Ada Reference Manual," SIGPLAN Notices, Vol. 14, No. 6, Part A, Jun., 1980

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

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

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

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

10. 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

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

12. 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

13. 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

14. 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

15. 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