Software Testing and Metrics for Concurrent Computation through Task Decomposition

Ying-Hong Wang, Chi-Ming Chung, Timothy K. Shih, Huan-Chao Keh, and Wei-Chuan Lin

Graduate Institute of Information Engineering, TamKang University
Tamsui, Taipei Hsien, Taiwan, China
TEL: +886-2-6215656 ext. 748, FAX: +886-2-6209749
E-mail: inhon@cs.tku.edu.tw

Abstract

Software testing is one of important approaches to assure the reliability and quality of software. Sequential programming testing is fairly sophisticated process. The emergence of concurrent programming in the recent years. Some concurrent program testing methodologies are proposed to solve controlled execution and determinism. However, there are few discussions of concurrent software testing from the inter-task viewpoints. This paper focuses the testing view on the concurrent programming through task decomposition. Four testing criteria are proposed to test a concurrent program. Programmer can choose an appropriate testing strategy depending on the properties of concurrent programs. A coverage criteria hierarchy is provided. Moreover, two suggestions for concurrent programming based on rendezvous complexity are made.

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

1. Introduction

The purpose of software testing is the assurance of software quality and software correctness. 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]. In the paper, we will discuss the testing problems of concurrent programs and try to propose new testing strategies focused on inter-tasks view.

Concurrent programs are programs with components that can be executed in parallel. 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.

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. 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. The ***rendezvous*** concept combines process synchronization and communication [1, 4]. This synchronization or communication to exchange information is called the rendezvous [5]. Thus we focus software testing in the rendezvous for concurrent programs and some testing strategies based on the rendezvous are proposed. 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, four testing criteria based on rendezvous view are proposed and the coverage criteria hierarchy is provided. In section 4, four equations for measuring complexity of concurrent program are proposed. Section 5 concludes the paper and describes our plans for future work.

2. Background of Concurrent Program Testing

The existing testing strategies of concurrent programs can be divided into some 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. Weiss obtain another approach towards testing by considering a concurrent program as a set of sequential program [14]. However, this thecnique is limited in practice because of state space explosion.

Second technique is testing based on *deterministic*

0-7803-4253-4/97/$10.00 © 1997 IEEE

execution. Tai, Carver and Obaid propose a deterministic execution technique to debug concurrent Ada programs [10]. There are some drawbacks such as selection of the appropriate SYN-sequences for covering the critical paths is a difficulty posed by this method. Also the programmer has to enumerate all possible SYN-sequences for testing a parallel program for all possible instances of execution for a given input.

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]. However, it is difficult to ensure that all possible execution instances are tested.

Yet another approach 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]. This testing algorithm has a polynomial time complexity.

3. A Rendezvous Oriented Testing for Concurrent Programs

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. However, it cannot represent multiple entry accepting of rendezvous.

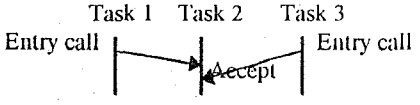


Fig. 3-1. A example of space-time diagram

In this paper, we will use a modified space-time diagram to show the rendezvous. We append a circle on the time flow to represent an entry call or entry accepting statement and label the entry name on the diagonal arrow to describe the occurring entry. The circles are divided into two classes: entry calling node and entry accepting node, marked as **EC** and **EA** respectively.

In Ada programming, the rendezvous is implemented by *entry call* and *accept* among tasks.

Clearly, the basic type of rendezvous is a task invoke an entry call, then the rendezvous is building. 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 same entry accepting. For example reduced from [8] is the following:

Example 1:
The tasks bodies are :

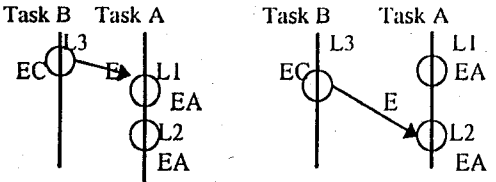
```
task body A is ...
   u, v : integer;
   begin
 L1    accept E(x : in out integer) do
           x := x + u;
       end accept;
 L2    accept E(x : in out integer) do
           x := x + v;
       end accept;
end A;
task body B is
       b : integer;
 L3    A.E(b);
       PUT(b);
end B;
```

The modified space-time diagram is shown in figure 3-2. 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 accepts at-least once, i.e. each edge from a EC node to different EA nodes of modified space-time graph must be traced at least once. □

Another testing problem of concurrent programs is race. A race occurs at an entry accepting that may contain at least two calls in its received queue. For Example 2, and their

modified space-time diagram is shown in figure 3-3.



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

Fig. 3-2. Two possible modified space-time diagrams for Example 1

The tasks can be abstractly described in Example 2 below:

Example 2:
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;
```

If we need to consider the ordering relationship, the race of messages displaying from Task B and Task C will happen. 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 permutation in received queue of each entry accept are tested at least once, i.e. the permutation of all edges from different EC nodes to a 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 accepting is the *permutation* of all possible entry calls.

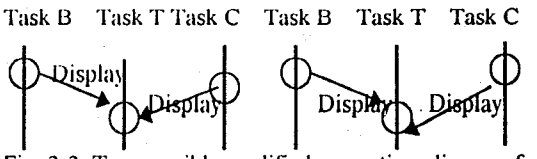


Fig. 3-3. Two possible modified space-time diagram for Example 2

Many tasks may have sent the same entry calls to a received task that has multiple entry accepting for the same entry name. If the executed ordering among the entry calls and the happened accepting statement are dependent, then the All-EC-Permute criterion is not enough because it just tests the permutation of individual entry accepting. It cannot test the permuted relationship between different entry accepting. Thus, we propose the fourth criterion to test the potential ordering-dependent permutation of all entry calls in all entry accepting with the same entry name.

In Example 3, extended from Example 1, Figure 3-4 depicts their possible modified space-time diagrams.

Example 3:
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;
```

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 accepting 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 accepting plus the permutations of all possible entry calls in different entry accepting. The coverage criteria hierarchy for proposed testing strategies is shown in figure 3-5.

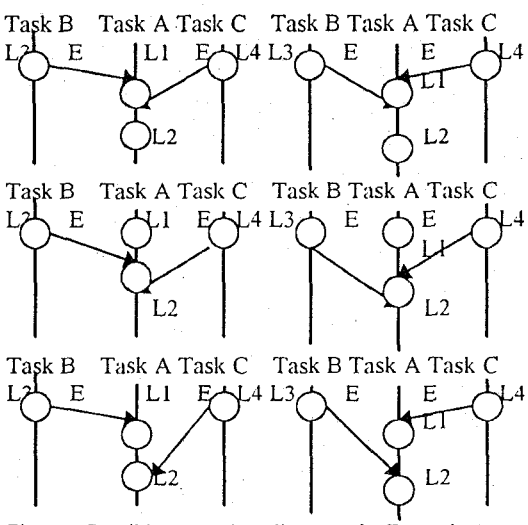


Fig. 3-4. Possible space-time diagrams for Example 4

According to the coverage criteria hierarchy and their proof, we make two suggestions as the following:

(1) **Don't centralize all entry accepting in few tasks:** It means the load of called tasks are heavy. Many tasks will send entry calls to the same entry accepting of a called task.

(2) **Don't distribute accepting statements to accept the same entry:** It means there are many possibilities when a task sends an entry call.

When a concurrent program has the above two properties, it means the difficulty of testing task will be raised. It would be advised for redesign to decreasing the complexity of testing .

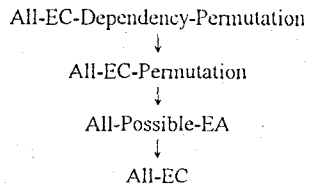


Fig. 3-5. Rendezvous-based testing coverage criteria hierarchy

4. Software Metrics for Concurrent Programs through Task Decomposition

Finally, a new view to measure the complexity of a concurrent program is proposed. 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

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

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.

5. Conclusion and Future work

In our research, a rendezvous point of view for concurrent program testing is proposed. Four testing criteria based on the rendezvous for concurrent/parallel programs are presented. A coverage criteria hierarchy for the four criteria is also provided. 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. Moreover, we will then 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