Concurrent software testing and metrics using task decomposition
Graduate Institute of Information Engineering, TamKang University, Tamsui, Taipei Hsien, Taiwan, R.O.C.
E-Mail: inhon@cs.tku.edu.tw

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 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 concurrent program testing methodologies 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. In this paper, we focus testing on the concurrent programming through a task decomposition mechanism. Four testing criteria to test a concurrent program are proposed. Associated with the strategies, four equations are provided to measure the complexity of the concurrent programs.

1. Introduction

Software testing and metrics are very important techniques in a software development life cycle. 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.

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]. Concurrent programs consist of components that can be executed in parallel. 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, testing concurrent programs is difficult and 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. 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. And 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, four testing criteria based on rendezvous are proposed. Considering rendezvous, four equations are introduced to measure the complexity of concurrent programs. The equations are presented in section 4. Section 5 concludes the paper and describes our future work.
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. 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.

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]. 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. If several tasks call the same entry of a task, then the calling tasks will rendezvous with the called task in the FIFO order in which the calls are received by the called task. Example 1 is illustrated the rendezvous mechanism of Ada [5].

Example 1:
The specifications of the two tasks are:

```ada
task PRODUCER;

task CONSUMER is
  entry RECEIVE(C: character);
```
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 RECEIYE" is called a block of accept statement.

3.2 Testing Criteria based on Rendezvous

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 tasks. And the diagonal arrows represent message passings, i.e., rendezvous. However, it cannot represent multiple entry acceptance statements of rendezvous. This is one of the nondeterministic types.

![Fig. 3-1. A example of the space-time diagram](image-url)
In this paper, we will use a modified space-time diagram 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.

![Diagram](image)

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. Example 2 is extracted from [9] and shown as the following.

**Example 2:**

Tasks specifications are:

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

task B;
```

The tasks bodies are:

```ada
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;
```
**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.

![Diagram](image)

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

Another testing problem of concurrent programs is determining race. The races are behavior of nondetermination. A race occurs at an entry acceptance that contain 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.

**Example 3:**

The tasks specifications are:

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

task B;

task C;
```

The tasks bodies are:

```plaintext
task body T is
    i:INTEGER
    begin
        ...
        accept Display(m : in LINE) do
            i := 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;
```

The modified space-time diagram is shown in figure 3-4. 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.

![Fig. 3-4. Two possible modified space-time diagram for Example 3](image)

Combining the situations of Example 2 and Example 3. In Example 4, extended from Example 2, there is another Task C which also calls entry E (labeled <L4> for its entry call statement). Multiple tasks may send the same entry calls to a receiving task that has multiple entry acceptance statements of the same entry name. Figure 3-5 depicts the possible modified space-time diagrams.

**Example 4:**

The tasks specifications are:

**task A is**

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

**task B;**

**task C;**

The tasks bodies are:

**task body A is**

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

**task body C is**

```plaintext
task body C is
  c : integer;
  begin
    ... <L4> accept E(x : in out integer) do
      A.E(c);
      PUT(c);
    end accept;
    ... end C;
```

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, the fourth
criterion is proposed to test the potential ordering-dependent permutation of all
entry calls in all entry acceptance statements of the same entry name. 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.

![Possible space-time diagrams for Example 4](image)

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

**4. Concurrent Software Metrics based on Rendezvous**

As mentioned in previous sections, synchronization and communication make
the most distinct difference between concurrent programs and sequential
programs. 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:

\[ \text{Equation 1} \]

\[
C_{px} = \sum_{i=1}^{M} m_{Ei}, \quad \text{where } C_{px} \text{ means the complexity of a concurrent program and index } i \text{ (from 1 to } M) \text{ represents each individual entry. The equation counts all entry call instructions. This is the most simple case in which all entry call statements and entry acceptance statements are injected.} \]

\[ \text{Equation 2} \]

\[
C_{px} = \sum_{i=1}^{M} (m_{Ei} \times n_{Ei}), \quad \text{for different entry acceptance statements received the same entry.} \]

\[ \text{Equation 3} \]

\[
C_{px} = \sum_{i=1}^{M} (n_{Ei} \times m_{Ei} !), \quad \text{that calculate the permutations of all rendezvous in an Ada program and the permutations of all rendezvous include all race cases.} \]

\[ \text{Equation 4} \]

\[
C_{px} = \sum_{i=1}^{M} ((n_{Ei} \times m_{Ei} !) + C(n_{Ei}, m_{Ei}) \times m_{Ei} !), \quad \text{that consider the ordering dependency among entry calls.} \]

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 load of called tasks are heavy.
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 a concurrent program has the above two properties, the programmer is recommended to redesign the program in order to decrease the complexity.

### 5. Conclusion and Future work

One major characteristic of concurrent programs compared with sequential program is rendezvous. a rendezvous measurement mechanism for concurrent program testing is proposed. In our research, four testing criteria based on the rendezvous of concurrent/parallel programs are introduced. 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. 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. We will apply the methodology to other programming domains, e.g., event-driven programming, network programming, and object-oriented programming.

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


