Test Case Generation of Concurrent Programs Based On Event Graph

Zuohua Ding[1,2], Kao Zhang[1] and Jueliang Hu[1]
[1]Center of Mathematical Computing and Software Engineering
Zhejiang Sci-Tech University
Hangzhou, 310018, P.R. China
[2]National Institute for Systems Test and Productivity
University of South Florida
Tampa, FL 33620, U.S.A
E-mail : zouhuading@hotmail.com

Abstract: This paper attempts to generate test cases for concurrent programs based on event graph. Through the analysis of state transition of event graph, sub-event-graphs can be generated. Each sub-event-graph corresponds to a test case. We may get benefits from this method in the following. 1) While executing the test case, we can monitor the state transition. 2) Every sub-event-graph is an execution path, or a simulation, thus all test cases are feasible. 3) Since the number of states in the event graph is finite, it is not likely to hit state explosion problem in the test generation process.

1. Introduction

A concurrent program specifies two or more processes (or threads) that cooperate in performing a task. Each process is a sequential program that executes a sequence statements. The processes cooperate by communicating using variables or message passing. One way to check that a concurrent program correctly implements its specification is to execute the program with a set of test sequences. A test sequence represents a sequence of actions performed by the concurrent processes in the program. These actions are often interprocess communications such as sending and receiving messages.

The testing of concurrent programs is difficult due to the inherent nondeterminism in these programs. That is, if we run a concurrent twice with the same test input, it is not guaranteed to return the same output both times. This nondeterminism causes two significant test automation problems: 1) it is hard to force the execution of a given program statements or branch and 2) it is difficult to automate the checking of test outputs.

To handle these problems, in this paper, we use event graph to simulate the behavior of concurrent programs. The test generation activities are based on the event graph model instead of the underlying source code. This abstraction model describes a program's execution by occurring state changes and their interactions on concurrently executing processes, which allows to cope equally with the programs based on the interaction mechanism.

Informally speaking, an event graph is a timed and conditioned directed graph representing discrete-event simulation models, each vertex representing an event, and each edge representing a relationship between events. The event graph can simulate both synchronous and asynchronous models with arbitrary timing delays. Thus the event graph can handle the nondeterminism caused by undecided event ordering, such as by message passing and resource sharing.

Here is the simple description of our process to generate test cases. First we model a concurrent program by event graph, then generate sub-event-graph by analyzing the migration of event states according to event graph. Finally, generate test cases according to sub-event-graph. This method has the following characteristics: 1) while executing the test case, we can monitor the state transition. 2) Since every sub-event-graph is a execution path, or a simulation, all test cases are feasible. 3) Since the number of all states in the event graph is finite, it is not likely to hit state explosion problem in the process of generating test cases.

This paper is organized as the following. Section 2 introduces event graph and the regulation of the state transition. Section 3 shows the process how to generate sub-event-graph. Section 4 gives an example. Section 5 is the conclusion of the paper.

2. Event Graph

Schruben (1983) [5] introduced Event Graphs (EGs) as a technique for graphically representing discrete-event simulation models. EGs have one basic construct with two elements: events represented by vertices, and relationships between events represented by edges. The relationships can be scheduling (i.e., execution of one event schedules another event) or canceling (i.e., execution of one event cancels another event). Additionally, they can be conditional and time dependent. Finally, the relationships can be parameterized to assign values to state variables when an event is executed (see Schruben [5] for more details on the event-graph constructs). Event graph can be used to describe program models. For example, in [1], event graphs are used to describe the model of Verilog. In this paper, with some modification, we will use event graph to describe the behavior of concurrent programs.

2.1 Event

Different definition of event may lead to different graph model. Before we give the definition to events, we give the definition of program execution states.

Definition 2.1: (Program State) A program state is defined as: a group of variables' values have been changed.

Definition 2.2: An event is defined as an activity of the program and can change the states of the program.

Events are regarded as instantaneous. If we wish to represent an activity with duration, we must introduce two events

to represent its start and finish so that other events can occur between them. The semantics of the original language determines the boundaries between events. Typical events might be *send message, receive message, user defined event/action, read, write, system events*, etc.

2.2 Edge and Timestamp

An edge represents a relationship between two events. An event may schedule another event for execution through edge. Edges contain three kinds of information: condition, duration time, and priority.

An edge with condition is used to provide the execution direction if more than one event could be scheduled. The condition is based on the boolean expression from control statement. Two types of control statements will be considered: branch and loop. An edge with duration time indicates that occurrence time from one event to another event. An edge with priority is used when processes are interacting based on the resources/services. If an edge has both time and priority, we should consider the priority first.

To order the events, we still adopt Lamport's "happened before" relation[3] between events:

- If a and b are events in the same process, and a comes before b , then $a \rightarrow b$
- If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$.
- If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

The independent relation a and b are said to be $concurrent$ if $a \not\rightarrow b$ and $b \not\rightarrow a$.

Since a duration time is attached with each edge, every event must have a timestamp, or instantaneous occurrence time. To compute the timesatmps, global clock will be used. The following are the rules in the computing. Let v_1, v_2 and v_3 be three events and the computing operation is denoted as $Cmpt(v)$.

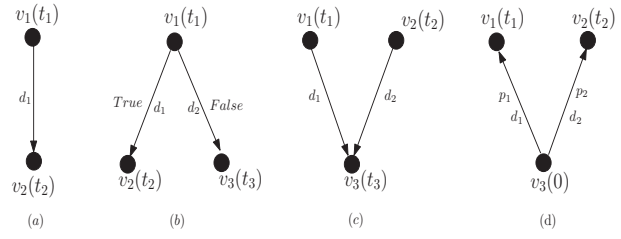

Figure 1. Cases to compute timestamps.

1. One event to one event. Assume that v_1 can activate v_2. The timestamp for v_1 is t_1. Let d_1 be the duration time on the edge from v_1 to v_2. Then the timestamp of v_2 will be $t_2 = t_1 + d_1$. See Figure 1(a).
2. One event to more events. Assume v_1 can activate v_2 or v_3 based on the condition. If the boolean expression is evaluated as true, then activate v_2, otherwise activate v_3. Let d_1 and d_2 be the duration time on the edges. Then the timestamps of v_2 and v_3 can be calculated as $t_2 = t_1 + d_1$ and $t_3 = t_1 + d_2$, respectively. See Figure 1 (b).
3. Assume that v_1 and v_2 together will activate v_3 to execute. Assume that v_1 has timestamp t_1 with duration time d_1 on the edge and v_2 has timestamp t_2 with with duration time d_2 on the edge. Then the timestamp of v_3 will be $t_3 = max(t_1 + d_1, t_2 + d_2)$. See Figure 1 (c).
4. Assume that both v_1 and v_2 need to be activated by v_3, but the edge to v_1 has higher priority p_1. Let d_1 and d_2 be the duration time on the edges. If v_1 is activated by v_3 first, then the timestamp for v_1 will be updated to $t_1 = t_1 + d_1$. If v_2 is activated by v_3 first, and $t_2 + d_2 < t_1$, then the timestamp of v_2 will be updated to $t_2 = t_2 + d_2$. If v_2 is activated by v_3 first, and $t_2 + d_2 > t_1$, then the execution of v_2 will be preempted by the execution of v_1. Thus the timestamp of v_2 would be $t_2 = t_2 + d_2 + d_1$. See Figure 1 (d).

2.3 Event Graph

Event graph is composed by events and edges between events. We use vertices to represent events and use directed edges to represent the relation between events.

An event graph is described as $G =< V, V_0, E_t, E_p, E_b >$, where

- V is the set of events,
- $V_0 \in V$ is the set of start events,
- $E_t \subset V \times V \times T$, where T is the set of duration time.
- $E_p \subset V \times V \times P$, where P is the set of priority.
- $E_b \subset V \times V \times B$, where B is the set of boolean expressions.

For more information of event graph, we refer to [5].

3. Generating Sub-Event-Graph

In the execution of event graph, we need to use memory store M to map each variable to its current value. We also assume the existence of the following operation:

- $Chgd(v, x, M)$ indicates if the execution of event v with initial memory store M changes the value of expression x.

Thus after each step of the execution, the current events and the memory store will be updated. Of course the time stamps of events and the variables are updated. Since the time stamps are associated with events and the variable values are stored in the memory store, we may use $s = (V, M)$ to denote the execution state. If we use operation $Exec$ to denote the execution, then we get $(V', M') = Exec(V, M)$. The algorithm of $Exec(V, M)$ is described as the following.

- Check the timestamp at each event of V and pick the event with the smallest timestamp. Assume this event is e_i.

1. If there is only one edge from this event, and the end event is e_{i+1}, then compute the timestamp and the variable values at e_{i+1} by $Cmpt(e_{i+1})$ and $Chgd(e_{i+1}, s, M)$.
2. If are more than two edges from this event, and assume the associated conditions are b_i^k, then apply $Eval(b_i^k, M)$. If someone is true, say b_i^1, then compute the timestamp and the variable values at e_{i+1}^1 by $Cmpt(e_{i+1}^1)$ and $Chgd(e_{i+1}^1, s, M)$.
3. If there is an edge coming into e_i with priority, and let e_0 be the event emitting this priority. Let d_{i0} be the duration time attached on the edge $e_i \rightarrow e_0$. Then check all the priorities from e_0.

(a) If the priority to e_i is the highest one, then update the timestamp and the variable values at e_i by $Cmpt(e_i)$ and $Chgd(e_i, s, M)$.

(b) If the priority to e_i is not the highest one, then check if the event has be preempted. (1) If no, then execute. If no preemption happens in the execution, then update the timestamp and the variable values at e_i by $Cmpt(e_i)$ and $Chgd(e_i, s, M)$. If some preemption happens in the execution, then denote this event as an intermediate event, still has the same priority, meanwhile compute the timestamp at e_i and the remaining execution time. (2) If yes, then continue the execution. If the execution can be successfully executed, then update the time stamp of e_i by $t_i = t_i + waiting-time+remaining-time$, and the variable values by $Chgd(e_i, s, M)$.

- If there are more than one event with the smallest timestamp, then comparing the priorities. The edge with the highest priority will be the execution direction. If no priorities associated with the edges, then pick any edge as the execution direction.

Finally, the simulation of an event graph G is a sequence of states $s_0, s_1, \ldots, \perp$, where

- $s_0 =< V_0, M_0 >$,where M_0 represents the initial memory store that maps every variable to an appropriate initial value.
- $s_{i+1} = Exec(s_i)$.
- $\perp$ is the end state.

Meanwhile, we will get a set of events. We give a name to such kind of sets, namely, SYN-sequences. Formally we have a definition:

Definition 3.1: [4] The SYN-sequence Q exercised by a concurrent execution is defined as a tuple $(Q_1, Q_2, \ldots, Q_n; \phi)$, where Q_i is the totally ordered sequence of sending and receiving events that occurred on a thread(process) or a synchronization object and ϕ is the set of synchronization pairs exercised in the execution.

For a concurrent program, a test case is actually a SYN-sequence based on the following coverage criteria:

1. Event Coverage Criterion - All events are executed at least once.
2. Edges Coverage Criterion - All Edges should be covered at least once.

The process to generate SYN-sequences can be described as the following expression:

$$V \xrightarrow[Exec(V,M)]{V} V_1 \xrightarrow[Exec(V_1,M)]{V_1} V_2 \ldots \xrightarrow[Exec(V,M)]{V_n} \perp.$$

V_n in the expression is the SYN-sequence, or the test case.

4. Case study

We takes an example from [2] as shown in Figure 2. The example is one of a single-server queueing system that has two types of customers. Of the two types of customers, customer type 1, has the highest priority. If a type 2 customer is in service when a type 1 customer arrives, the type 2 customer is preempted by the type 1 customer and will wait until all type 1 customers have been served.

The state variables for this system are defined as:

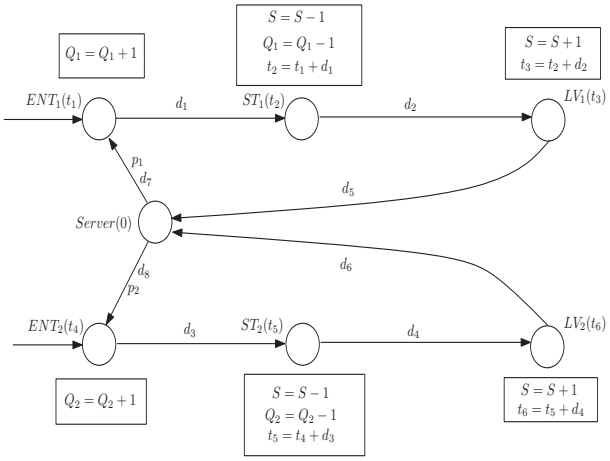

Figure 2. Single-Server Preemptive System

- Q_1 and Q_2 : The number of customers in queue 1 and queue 2, respectively.
- S: Server Status: 1 = available, 0 = busy.
- Preemptive: Preemptive status: 1 = Preemptive, 0 = no preemptive.

The events of the system are:

- ENT_1 : When a customer enters the system, ENT_1 will change Q_1 to $Q_1 + 1$.
- ST_1 : the customer starts his service and changes the values of S and Q_1.
- LV_1 : the customer finishes his service and releases the server.
- ENT_2 : When a customer enters the system, ENT_2 will change Q_2 to $Q_2 + 1$.
- ST_2 : the customer starts his service and changes the values of S and Q_2.
- LV_2 : the customer finishes his service and release the server.
- $Server$: the system event to allocate the server.
- $Preemptive$: $Preemptive = 1$ force to stop the current customer getting the server.

When a type 1 customer enters the system(ENT_1), if there is a server available, the customer starts his service(ST_1) and then finishes his service(LV_1) and releases the server. When a type 2 customer enters the system(ENT_2), if there is a server available, the customer starts his service(ST_2) and then finishes his service(LV_2) and release the server. So both ENT_1 and ENT_2 can be activated by $Server$ which are allocated by comparing priority. The customer with the higher priority will be executed first.

There are about 5 cases here and they are:

1. Each of Q_1 and Q_2 has only one customer in the queue. Event ENT_1 and ENT_2 are activated at the same time. Assume that $d_2 > d_3 + d_5$, and thus ENT_1 starts to execute after ENT_2 finishes.
2. Each of Q_1 and Q_2 has one customer in the queue. Event ENT_1 and ENT_2 are activated at the same time. Assume that $d_2 < d_3$.
3. Each of Q_1 and Q_2 has one customer in the queue. Event

ENT_1 and ENT_2 are activated at the same time. Assume that $d_3 < d_2 < d_3 + d_5$.
4. Q_1 has more than one customer and Q_2 has only one customer. Assume Q_1 has two customers. Once the first customer is done with the service, the second customer will be served immediately. Event ENT_1 and ENT_2 are activated at the same time. Assume that $d_3 < d_2 < d_3 + d_5$ and $2 \times d_2 < d_5$.
5. Q_1 has one customer 1 and Q_2 has one customer 2. Assume another customer 1 will come later and $d_2 > d_3$. Event ENT_1 and ENT_2 are activated at the same time.

For example, to generate test cases for case 5. The process can be described as the following (to save space, we use $\bar{e}$ to represent that the event e is executing, otherwise we need to assume the head and tail events):
- The state of V_0 is $V_0 = \{ENT_1, ENT_2\}$.
- ENT_1 and ENT_2 transit to the next events respectively. $d_2 > d_3$ shows that customer 2 activates ST_2 when ENT_1 is in the processing of the transition. So $V_1 = \{\overline{ENT_1}, ST_2\}$
- Server is available and ST_2 can get the Server successfully. Monitor records the pairs $(Server, ST_2)$.
- when event ST_1 is activated, ST_2 is in the transition. So V_2 is $V_2 = \{ST_1, \overline{ST_2}\}$.
- since Event ST_1 has higher priority, it can get the server which is occupied by ST_2. Thus p_1 has higher priority than p_2 to interrupt the executing of ST_2, and force ST_2 to release the server. So $V_3 = \{ST_1, \overline{ST_2}, Preemptive\}$.
- Recording the pairs $(Sever, ST_1)$ by monitor. ST_1 continues the executing after getting the sever and transits to LV_1. Meanwhile ST_2 is blocked. Thus, $V_4 = \{LV_1, \overline{ST_2}\}$.
- After finishing LV_1, LV_1 releases the server to ST_2 and the current state status of Q_1 is 0. ST_2 gets the Server and goes on executing. While ST_2 is still in transition, another customer 1 enters the system. So $V_5 = \{ENT_1^1, \overline{ST_2}\}$.
- When LV_2 is activated, ENT_1^1 starts the transition. So $V_6 = \{\overline{ENT_1^1}, LV_2\}$.
- Customer 2 is done and release the Server, and then ST_1^1 is activated. So $V_7 = \{ST_1^1, \emptyset\}$.
- Event ST_1^1 gets the server successfully and continues the execution. Monitor records pairs $(Server, ST_1^1)$. $V_8 = \{LV_1^1, \emptyset\}$.
- Customer 1 finishes all and release $Server$. $V_9 = \{\emptyset, \emptyset\}$.

The above steps can be summarized in the following expression:

$$
\begin{aligned}
& V_0 \rightarrow V_1 \xrightarrow{(Server, ST_2)} V_2 \rightarrow V_3 \xrightarrow{(Server, ST_1)} V_4 \\
& \rightarrow V_5 \rightarrow V_6 \rightarrow V_7 \xrightarrow{(Server, ST_1^1)} V_8 \rightarrow V_9 \\
= \ & \{ENT_1, ENT_2\} \rightarrow \{\overline{ENT_1}, ST_2\} \\
& \xrightarrow{(Server, ST_2)} \{ST_1, \overline{ST_2}\} \\
& \rightarrow \{ST_1, \overline{ST_2}, Preempitve\} \\
& \xrightarrow{(Server, ST_1)} \{LV_1, \overline{ST_2}\} \rightarrow \{ENT_1^1, \overline{ST_2}\} \\
& \rightarrow \{\overline{ENT_1^1}, LV_2\} \rightarrow \{ST_1^1, \emptyset\} \\
& \xrightarrow{(Server, ST_1^1)} \{LV_1^1, \emptyset\} \rightarrow \{\emptyset, \emptyset\} \\
= \ & \{\{ENT_1, ST_1, LV_1, , ENT_1^1, ST_1^1, LV_1^1\}, \\
& \{ENT_2, ST_2, LV_2\}, \{Preemptive\}; \\
& (Server, ST_2), (Server, ST_1), (Server, ST_1^1)\}
\end{aligned}
$$

For other cases, the test cases are:

1. $\{\{ENT_1, ST_1, LV_1\}, \{ENT_2, ST_2, LV_2\}; (Server, ST_2), (Server, ST_1)\}$
2. $\{\{ENT_1, ST_1, LV_1\}, \{ENT_2, ST_2, LV_2\}; (Server, ST_1), (Server, ST_2)\}$
3. $\{\{ENT_1, ST_1, LV_1\}, \{ENT_2, ST_2, LV_2\}, \{Preemptive\}; (Server, ST_2), (Server, ST_1)\}$
4. $\{\{ENT_1, ST_1, LV_1, , ENT_1^{'}, ST_1^{'}, LV_1^{'}\}, \{ENT_2, ST_2, LV_2\}, \{Preemptive, Preemptive^{'}\}; (Server, ST_2), (Server, ST_1), (Server, ST_1^{'})\}$

5. Conclusion

This paper simulated the concurrent programs by event graph. Sub-event-graphs have been generated through the analysis of state transition of event graph. Each sub-event-graph corresponds to a test case. According to the theorem 1 in[1] (The number of possible static simulation states for any event graph is finite), the process to generate test cases will not hit the problem of state explosion. For the future work we will investigate how to find the least number of sub-event-graphs to cover all the events, in other words, how to find the minimum of test cases.

Acknowledgments

This work is partially supported by the National High-Tech Research and Development Plan of China under Grant No.2006AA01Z165.

References

[1] R. S. French, M. S. Lam, A general method for compiling event-driven simulations. In: *Proceedings of 32nd ACM/IEEE Design Automation Conference*, pp. 151-156, 1995.

[2] R. G. Ingalls, D. J. Morrice, E. Yúcesan, A. B. Whinston, Execution conditions: a formalization of event cancellation in Ssimulation graphs, *Journal on Computing*, vol.15, no.4, pp.397-411, 2003.

[3] L. Lamport, Time, clocks, and the ordering of events in a distributed system, *Communications of the ACM*, vol.21, no.7, pp. 558-565, July, 1978.

[4] Y. Lei and R.H.Carver, Reachability testing of concurrent programs, *IEEE Transactions on Software Engineering* , vol.32, no.6, pp.382-403, 2006.

[5] L. W. Schruben, Simulation modeling with event graphs, *Communications of ACM*, vol.26, pp.957-963, 1983.