Statement-Coverage Testing for Concurrent Programs in Reachability Testing

Gwan-Hwan Hwang, Heng-Yi Lin, Shao-Yan Lin and Che-Sheng Lin

Department of Computer Science and Information Engineering
National Taiwan Normal University
Taipei, 106 Taiwan

In this paper we propose a scheme for reachability testing to achieve statement coverage in the dynamic testing of concurrent programs. Previous studies on reachability testing have only enumerated the feasible interleavings of a concurrent program for a given input. The proposed scheme derives inputs from SYN-sequences obtained in reachability testing and uses these inputs to perform reachability testing multiple times in order to achieve statement-coverage testing for a concurrent program. We prove formally that the proposed method can achieve statement-coverage testing if all the path conditions derived from SYN-sequences can be solved and the concurrent program contains no dead code.

Keywords: concurrent programs, nondeterministic behavior, concurrent testing, SYN-sequence, reachability testing, symbolic execution

1. INTRODUCTION

Statement coverage was among the first methods invented for systematic software testing [1]. Statement-coverage testing is designed to ensure that each statement in a program is executed at least once, which helps to identify statements or codes that are not being tested. Statement coverage is especially useful when testing logic-intensive applications with many decision points (compared to data-intensive applications, which tend to have decision paths that are much less complex). Generally speaking, the dynamic testing of a target concurrent program is insufficient if it is executed with only a single input. Symbolic execution is one of the schemes used to select a set of inputs for achieving statement coverage in dynamic testing [2]. When a program is symbolically executed, a special symbolic variable (as distinct from the program’s variables) is associated with the values returned from an execution of a statement. These symbolic variables, and expressions thereof, are tracked in a special symbolic state. At any control transfer instruction, a path condition is updated to track the branch that was taken. By negating some of the conditions in the path constraint, and by using a constraint solver to obtain satisfying assignments to the modified path constraint, it is possible to generate inputs that explore unexplored parts of the program.

Concurrent programming is becoming commonplace in modern computing. A concurrent program contains two or more processes or threads that execute concurrently and work together to perform some task. Multiple executions of a concurrent program P with the same input might produce different results. This so-called nondeterministic behavior [3, 4] means that when testing P with input X (which is a sequence of inputs for process-
es/threads in P), a single execution is insufficient to determine the correctness of P with X. Even if P has been executed successfully many times with input X, it is possible that a future execution of P with X will produce incorrect results. An execution of a concurrent program exercises a sequence of synchronization events called a synchronization sequence (or SYN-sequence). Examples of process synchronization include “wait” (acquire) and “signal” (release) primitives applied to a shared semaphore, monitor-entry procedures, sending and receiving of messages, and general sharing of memory [5, 6]. A concurrent program can exhibit nondeterministic behavior for several reasons, including process scheduling and speed differences among the processors. Such nondeterministic behavior could result in different executions of P with the same input X producing different SYN-sequences. In general, for a concurrent program with the same input, the same interleaving always produces the same SYN-sequence. This paper uses the following terminology: a test of a concurrent program P with input X refers to a single execution of P with input X to obtain a SYN-sequence and checking of the execution result.

Dynamic testing in software engineering refers to examining the physical response of a system to variables that change with time, and is used to test the dynamic behavior of program [7]. Dynamic testing involves compiling the software, running it with particular input values, and then checking if the output is as expected, and includes unit testing, integration testing, system testing, and acceptance testing methodologies. Source-code-level dynamic testing is considered to be an important step in the software life cycle (software development process) [8]. A simple approach to perform dynamic testing of a concurrent program P with nondeterministic behavior involves executing P with a fixed input many times in the hope that this will expose faults. This type of testing, called nondeterministic testing, is easy to perform, but it is usually very inefficient and has two major problems: (1) some feasible SYN-sequences of P with input X might be executed many times, and (2) some feasible SYN-sequences might never be executed [9, 10].

Systematic and exhaustive techniques have been developed for testing nondeterministic concurrent programs to explore different interleavings or SYN-sequences. Hwang et al. proposed the idea of reachability testing1 for dealing with the problem of nondeterministic behavior of concurrent programs [11, 12]. Reachability testing is an approach for the dynamic testing of a concurrent program that explores different interleavings or SYN-sequences without static analysis of the target source program. The SYN-sequences generated during testing are analyzed so that the subsequent tests explore different interleavings, thereby producing different SYN-sequences. CHESS is a concurrent testing tool that repeatedly runs a concurrent test whilst ensuring that every run takes a different interleaving [13, 14]. If an interleaving results in an error, CHESS can reproduce the interleaving (for improved debugging). CHESS uses stateless model checking techniques to systematically generate all interleavings of a given input. However, these systematic schemes can only explore interleavings for some given inputs; they cannot guarantee statement coverage. Fig. 1 shows a concurrent program with two processes. One of the input sets that fulfills statement coverage is a set of two inputs: \{(a = 1, b = 0), (a = 0, b = 2)\}. The execution of the first input, \((a = 1, b = 0)\), actually has two different interleavings: \((S_{0,0}, S_{0,2}, S_{1,0}, S_{1,2})\) and \((S_{0,0}, S_{1,0}, S_{0,2}, S_{1,1})\). There are five statements that can be reached (or executed) by this input. However, we need to execute the concurrent program with two interleavings to reach all of these statements. \(S_{0,1}\) cannot be reached by this input but can be reached by the input \((a = 0, b = 2)\).

1 Reachability testing is illustrated in Section 2.
In this paper we propose a scheme to automatically select a set of inputs to guide the reachability testing in order to perform a statement-coverage testing for concurrent programs. Our scheme does not need to analyze the structure or semantics of the target concurrent program. We derive a sequence of path conditions from the SYN-sequence obtained in reachability testing. Then, the collected path conditions are used to deploy a constraint solver to derive new inputs so that we can perform statement-coverage testing by applying reachability testing multiple times. We prove formally that the proposed scheme can achieve statement-coverage testing if all the path conditions derived from SYN-sequences can be solved and the concurrent program contains no dead code.

The paper is organized as follows. Section 2 presents the proposed scheme. Section 3 gives the formal proof to demonstrate the correctness of the proposed scheme. Experimental results are presented in Section 4. Section 5 surveys previous work, and conclusions are drawn in Section 6.

2. DERIVING INPUTS FROM SYN-SEQUENCES TO OBTAIN STATEMENT COVERAGE

Consider an input $X$ of a concurrent program $P$. Reachability testing can derive a set of SYN-sequences that represent all the feasible interleavings of $P$ with $X$. Referring to Fig. 2, reachability testing of $P$ with input $X$ involves the following steps:

1. Obtain a SYN-sequence $S$ by tracing a nondeterministic execution of $P$ with $X$.
2. Uses $S$ to derive a set of prefixes of other feasible SYN-sequences of $P$ with input $X$.
   Such prefixes (called race variants of $S$) are derived by changing the outcome of race conditions in $S$. An execution that follows a race variant of $S$ will always exercise a SYN-sequence that differs from $S$.
3. For each new race variant $R$ derived in step 1, perform a prefix-based replay of $P$ with input $X$ and $R$ to execute and collect an additional SYN-sequence for $P$ with input $X$.
   The prefix-based replay of $P$ with a race variant $R$ comprises two phases:
   (a) Replay phase: control the execution of the program by following the execution order specified in $R$.
   (b) Monitor phase: execute the program without any control and record subsequent synchronization events after replaying $R$.
4. Repeat steps 2 and 3 for each new SYN-sequence collected in step 3.

---

Due to space limitations, we do not give the formal definitions of race variant and prefix-based replay; these are available in [12, 15].
Note that \( R \) is not a complete SYN-sequence of an execution of \( P \). Thus, after replaying \( R \) we have to record the subsequent synchronization events to obtain a SYN-sequence of an execution of \( P \). This performs a test on the target concurrent program. For performing the prefix-based replay of a race variant, an entry protocol and an exit protocol must be inserted before and after each synchronization event in the original program, respectively [11, 15]. We denote the set of SYN-sequences obtained by the reachability testing of \( P \) with \( X \) as \( \text{ReachabilityTesting}(P, X) \).

![Fig. 2. Reachability testing.](image)

Given a single input of a terminating concurrent program, reachability testing can derive all the feasible SYN-sequences [12, 15]. Our scheme works by deriving new inputs from the SYN-sequences obtained from reachability testing. By conducting reachability testing again according to the derived inputs, the scheme can guide the testing process to cover more statements. The newly generated SYN-sequences are then used to derive more inputs. This process is repeated until all the statements are covered. Note that if the concurrent program contains dead codes, which are statements not reachable in any input under any interleaving, the scheme has to stop the testing and report these codes.

### 2.1 Derive an Input from a SYN-Sequence to Reach a Specified Statement

We now describe how to obtain path conditions from a SYN-sequence produced by reachability testing. A SYN-sequence consists of a sequence of synchronization events. Compared with traditional reachability testing, we need more information in the synchronization events in order to derive path conditions. A scheme for replaying the sequence of read and write operations of shared-memory concurrent programs has been presented by LeBlanc and Mellor-Crummey [16]. It is referred to as “instant replay”, in which only the version number of the accessed variables needs to be recorded. For race analysis in reachability testing, the name and version number of the accessed shared variable must be recorded [11, 15]. The read and write events are denoted \( R(U, V) \) and \( W(U, V) \), respectively, where \( U \) and \( V \) are the name and version number of the variable. In
general, the variable name and version number in a synchronization event are sufficient to generate race variants for exploring feasible interleavings, since the variable name can recognize if two events could race and we can rearrange their order by changing the version numbers to produce race variants. However, to derive the path conditions, the execution location of the read or write operation in the program context and the value of the accessed shared variable must be recorded in the synchronization event. Thus, in [17], we extend the format of SYN-sequences. The read and write events stored in the SYN-sequence are denoted $R(U, V, L, C, T)$ and $W(U, V, L, C)$, respectively, where $L$ is the execution location of the read or write operation in the program context (there can be distinct labels for each read and write operation), $C$ is the value of the accessed variable (i.e., the value read or written in a read or write operation, respectively), and $T$ is a local variable to which the value read is assigned.

In a high-level programming language, a single statement can contain multiple events. For example, if $SV_1, SV_2,$ and $SV_3$ are shared variables, then the statement "$SV_1 = SV_2 + SV_3$" will issue three synchronization events: two read operations of $SV_2$ and $SV_3$, and one write operation to $SV_1$. We have to perform a simple transformation of the target program to ensure that each statement contains at most one synchronization event [17], since the existing algorithms for generating race variants assume that a statement contains only a single synchronization event [11, 12, 15, 17]. Without loss of generality, we assume that our target concurrent programs are shared-memory concurrent programs and their executions must terminate. The statement "$tv = read(sv)$" involves reading the value of shared variable "$sv$" and storing it in local variable "$tv$", while "$write(sv, lv)$" involves writing the value of local variable "$lv$" into shared variable "$sv$". A branch statement is a typical if-then-else two-way selector that chooses between two execution paths. Fig. 3 shows a concurrent program with three processes.

Referring to Fig. 3, let $S$ be one of the feasible SYN-sequences of the execution of the concurrent program with input $(a = 0, b = 0, c = 1)$. Note that we denote $S[0], S[1],$ and $S[2]$ as the collection of synchronization events executed by processes $P0, P1,$ and $P2$, respectively:

$$S[0] = \{R(c, 0, S_{0,0}, 1, t1), W(b, 1, S_{0,3}, 2)\}$$
$$S[1] = \{R(a, 0, S_{1,0}, 0, t2), R(b, 1, S_{1,1}, 2, t3), W(c, 1, S_{1,4}, -1), R(b, 2, S_{1,5}, 1, t4)\}$$
$$S[2] = \{R(a, 0, S_{2,0}, 0, t5), R(b, 1, S_{2,1}, 2, t6), W(b, 2, S_{2,3}, 1)\}.$$
The information shown in SYN-sequence $S$ is not sufficient to derive the symbolic value and path condition after each statement. The generation of symbolic values and path conditions is supported by constructing a statement information table (SIT) for each process. Table 1 lists the SITs of the three processes. We classify the statements into three types: (1) a “BRCH” statement is an “if” statement, (2) an “IAB” statement is a statement that is executed immediately after a “BRCH” statement, and (3) other statements that are called “NIAB” statements. Note that an IAB statement is the first statement of a basic block [18]. For each IAB statement we have to record its previous statement (which is a BRCH statement) and the corresponding constraint.

**Table 1. SITs of the three processes in Fig. 3.**

(a) SIT of process 0

<table>
<thead>
<tr>
<th>Location</th>
<th>Statement</th>
<th>Type</th>
<th>Constraint</th>
<th>Previous statement</th>
<th>Branch location when True</th>
<th>Branch location when False</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_{0,0}$</td>
<td>$t_1 = c$</td>
<td>NIAB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_{0,1}$</td>
<td>if ($t_1 &lt; 0$)</td>
<td>BRCH</td>
<td></td>
<td>$S_{0,2}$</td>
<td>$S_{0,1}$</td>
<td></td>
</tr>
<tr>
<td>$S_{0,2}$</td>
<td>$b = 1$</td>
<td>IAB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_{0,3}$</td>
<td>$b = 2$</td>
<td>IAB</td>
<td></td>
<td>if ($t_1 &lt; 0$) $T$</td>
<td>$S_{0,1}$</td>
<td></td>
</tr>
</tbody>
</table>

(b) SIT of process 1

<table>
<thead>
<tr>
<th>Location</th>
<th>Statement</th>
<th>Type</th>
<th>Constraint</th>
<th>Previous statement</th>
<th>Branch location when True</th>
<th>Branch location when False</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_{1,0}$</td>
<td>$t_2 = a$</td>
<td>NIAB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_{1,1}$</td>
<td>$t_3 = b$</td>
<td>NIAB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_{1,2}$</td>
<td>if ($t_2 &gt; t_3$)</td>
<td>BRCH</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_{1,3}$</td>
<td>$c = 1$</td>
<td>IAB</td>
<td></td>
<td>if ($t_2 &gt; t_3$) $T$</td>
<td>$S_{1,2}$</td>
<td></td>
</tr>
<tr>
<td>$S_{1,4}$</td>
<td>$c = -1$</td>
<td>IAB</td>
<td></td>
<td>if ($t_2 &gt; t_3$) $F$</td>
<td>$S_{1,2}$</td>
<td></td>
</tr>
<tr>
<td>$S_{1,5}$</td>
<td>$t_4 = b$</td>
<td>NIAB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(c) SIT of process 2

<table>
<thead>
<tr>
<th>Location</th>
<th>Statement</th>
<th>Type</th>
<th>Constraint</th>
<th>Previous statement</th>
<th>Branch location when True</th>
<th>Branch location when False</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_{2,0}$</td>
<td>$t_5 = a$</td>
<td>NIAB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_{2,1}$</td>
<td>$t_6 = b$</td>
<td>NIAB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_{2,2}$</td>
<td>if ($t_5 &lt; t_6$)</td>
<td>BRCH</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_{2,3}$</td>
<td>$b = 1$</td>
<td>IAB</td>
<td></td>
<td>if ($t_5 &lt; t_6$) $T$</td>
<td>$S_{2,2}$</td>
<td></td>
</tr>
<tr>
<td>$S_{2,4}$</td>
<td>$b = -1$</td>
<td>IAB</td>
<td></td>
<td>if ($t_5 &lt; t_6$) $F$</td>
<td>$S_{2,2}$</td>
<td></td>
</tr>
</tbody>
</table>

Given a statement $m$ of a concurrent program $P$ and a feasible SYN-sequence $S$ of $P$, we can apply Algorithm 1 [**Select Input**($S$, $m$, SITs)] to select an input $I$, where the execution of $P$ with $I$ does cover statement $m$. Algorithm 1 first derives a total order sequence (TOS) $T$ from $S$. However, $T$ does not include all the events in $S$. Some irrelevant events regarding the reaching of statement $m$ are removed by Algorithm 2: [i.e., TOS($S$, $m$, SITs)]. We can derive an appropriate input by performing symbolic execution of $T$ and solving the path condition of the last event (which is the previous statement of $m$). Figs. 4 and 5 illustrate the invocation of TOS($S$, $m = S_{1,3}$, SITs) and the symbolic execu-
tion of the TOS obtained in $TOS(S, m = S_{1,3}, \text{SITs})$. The path condition of the last event is “$C < 0 \land A > 1$”. By solving this we can derive an input ($a = 2$, $c = -1$).

**Algorithm 1**: Derive an input from a SYN-sequence to reach a specified statement

**Input**: A SYN-sequence $S$ of a concurrent program $P$, a specified IAB statement $m$, SITs (the SITs of all the processes in $P$).

**Output**: An input of $P$.

**Select Input($S$, $m$, SITs)**

1. Derive a TOS from $S$. Let $T = TOS(S, m, \text{SITs})$.
2. Perform symbolic execution on $T$. We can obtain the correct symbolic values and path conditions in each event.
3. Assume that $I$ is the solution of the path condition of the last event in $T$. Note that $I = \emptyset$ if there is no solution.
4. Return $I$.

**Algorithm 2**: Obtain a TOS from a SYN-sequence $S$

**Input**: A SYN-sequence $S$, a specified IAB statement $m$, SITs (the SITs of all the processes in concurrent program $P$).

**Output**: A TOS.

$TOS(S, m, \text{SITs})$

1. Let $POG$ be the partial order graph$^4$ obtained from $S$.
2. Let $m_{\text{pre}}$ be the previous statement of $m$.
3. For each IAB statement $r$ in $POG$, insert the previous statement of $r$ in $POG$. Note that the insert node is labeled with the constraint of statement $r$.
4. Remove all the nodes that have not happened-before$^5$ $m_{\text{pre}}$ from $POG$.
5. Label the constraint of $m_{\text{pre}}$ to be the constraint of $m$.
6. Apply a topological sort to obtain a TOS from $POG$, which is designated $T$.
7. Return $T$.

---

(a) The partial order graph of $S$. (b) The partial order graph after step 3. (c) The partial order graph after step 5.

---

$^3$ $\emptyset$ is an empty set.

$^4$ The partial order graph and the algorithm that derives a partial order graph from a SYN-sequence are derived in [15].

$^5$ See Definition 2 in [15].
GWAN-HWAN HWANG, HENG-YI LIN, SHAO-YAN LIN AND CHE-SHENG LIN

<table>
<thead>
<tr>
<th>Equation/Condition</th>
<th>Symbolic Values</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 = c )</td>
<td>( t_1 = C )</td>
<td>True</td>
</tr>
<tr>
<td>( \text{if} \ (t_1 &lt; 0) ) ( T )</td>
<td>( t_1 = C )</td>
<td>( C &lt; 0 )</td>
</tr>
<tr>
<td>( b = 1 )</td>
<td>( t_1 = C, b = 1 )</td>
<td>( C &lt; 0 )</td>
</tr>
<tr>
<td>( t_2 = a )</td>
<td>( t_1 = C, b = 1, t_2 = A )</td>
<td>( C &lt; 0 )</td>
</tr>
<tr>
<td>( t_3 = b )</td>
<td>( t_2 = A, t_1 = C, t_3 = 1 )</td>
<td>( C &lt; 0 )</td>
</tr>
<tr>
<td>( \text{if} \ (t_2 &gt; t_3) ) ( T )</td>
<td>( t_2 = A, t_1 = C, t_3 = 1 )</td>
<td>( C &lt; 0 \land A &gt; 1 )</td>
</tr>
</tbody>
</table>

Fig. 5. Symbolic execution of \( \text{TOS}(S, m = S_{1,3}, \text{SITs}) \).

2.2 Derive Inputs for Statement Coverage using a Greedy Method

Algorithm 1 can be used to develop a scheme that selects a set of inputs to achieve statement coverage. Algorithm 3 performs reachability testing multiple times for given different inputs that are derived by previous executions of reachability testing. The IAB statements that had been reached in reachability testing are recorded in \( \Delta \). This is a greedy method because it only attempts to derive new inputs from IAB statements that are not recorded in \( \Delta \). However, the greedy method cannot achieve statement coverage in some situations.

**Algorithm 3:** A greedy method to derive inputs

**Input:** A concurrent program \( P \). Assume that \( \Sigma \) is the set of all the IAB statements in all the processes of \( P \) and SITs (the SITs of all the processes in concurrent program \( P \)).

**Output:** A set of inputs, \( \Omega \).

**Greedy_select_inputs(\( P \))**

1. **Let** \( \Omega = \emptyset \) and \( \Delta = \emptyset \).
2. **Arbitrarily select an input** \( X \) of \( P \). **Let** \( \Omega = \Omega \cup X \).
3. **Arbitrarily remove an input** \( I \) from \( \Omega \).
4. **Let** \( \Gamma = \text{ReachabilityTesting}(P, I) \).
5. **Let** \( \Delta = \Delta \cup \text{All}_\text{IAB}_\text{Statements}_\text{Reached}(\Gamma) \). **IF** (\( \Delta = \Sigma \)) THEN return.
6. **FOR** each IAB statement \( m \) in \( (\Sigma - \Delta) \)
   - **Assume** that \( m' \) and \( m \) have the same previous statement.
   - **FOR** each SYN-sequence \( S \) in \( \Gamma \) that reaches statement \( m' \)
     - \( \alpha = \text{Select_Input}(S, m, \text{SITs}) \).
     - **IF** (\( \alpha \neq \emptyset \) \( \land \alpha \) is a new input)
       - **THEN** \( \Omega = \Omega \cup \alpha \).
       - \( \Delta = \Delta \cup m \).
       - **EXIT FOR loop**
   - **END IF**
   - **END FOR**
   - **END FOR**
7. **IF** (\( \Omega \neq \emptyset \)) THEN GOTO step (3).
We employ the concurrent program shown in Fig. 6 to demonstrate Algorithm 3. An input $X = (a = 0, b = 0)$ is first chosen arbitrarily in step (2). The execution of ReachabilityTesting($P, (a = 0, b = 0)$) yields two SYN-sequences SYN1 and SYN2:

$\text{SYN1}[0] = \{R(b, 0, S_{0,0}, 0, t1), W(a, 1, S_{0,3}, -1)\}$
$\text{SYN1}[1] = \{R(a, 1, S_{1,0}, -1, t2), W(c, 1, S_{1,7}, 0)\}$
$\text{SYN2}[0] = \{R(b, 0, S_{0,0}, 0, t1), W(a, 1, S_{0,3}, -1)\}$
$\text{SYN2}[1] = \{R(a, 0, S_{1,0}, 0, t2), W(c, 1, S_{1,7}, 0)\}$

<table>
<thead>
<tr>
<th>Process $P_0$</th>
<th>Process $P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_{0,0}$</td>
<td>$S_{1,0}$</td>
</tr>
<tr>
<td>$t1 = \text{read}(b)$;</td>
<td>$t2 = \text{read}(a)$;</td>
</tr>
<tr>
<td>$S_{0,1}$</td>
<td>$S_{1,1}$</td>
</tr>
<tr>
<td>if ($t1 = 2$)</td>
<td>if ($t2 &gt; 0$) {</td>
</tr>
<tr>
<td>$S_{0,2}$</td>
<td>$S_{1,2}$</td>
</tr>
<tr>
<td>write($a$, $-2$);</td>
<td>write($c$, 1);</td>
</tr>
<tr>
<td>else</td>
<td>$S_{1,3}$</td>
</tr>
<tr>
<td>$S_{0,3}$</td>
<td>$S_{1,4}$</td>
</tr>
<tr>
<td>write ($a$, $-1$);</td>
<td>if ($t3 = -2$)</td>
</tr>
<tr>
<td></td>
<td>$S_{1,5}$</td>
</tr>
<tr>
<td></td>
<td>write($d$, 1);</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>$S_{1,6}$</td>
</tr>
<tr>
<td></td>
<td>write($d$, 0);</td>
</tr>
<tr>
<td></td>
<td>} else</td>
</tr>
<tr>
<td></td>
<td>$S_{1,7}$</td>
</tr>
<tr>
<td></td>
<td>Write($c$, 0);</td>
</tr>
</tbody>
</table>

Fig. 6. Example of a concurrent program with two processes.

$\Sigma = \{S_{0,2}, S_{0,3}, S_{1,2}, S_{1,5}, S_{1,6}, S_{1,7}\}$ and $\Delta = \text{All}\_\text{IAB\_Statements\_Reached}(\{\text{SYN1, SYN2}\}) = \{S_{0,3}, S_{1,7}\}$; thus, $\Sigma - \Delta = \{S_{0,2}, S_{1,2}, S_{1,5}, S_{1,6}\}$. In the first execution of the FOR loop in step (6) we have $m = S_{0,2}$ and $m' = S_{0,3}$. The first execution of the nested FOR loop activates Select_Input($\text{SYN1}, S_{0,2}, \text{SITs}$) (see Fig. 7 (a)). We derive an input $(a = 0, b = 2)$, and now $\Delta = \{S_{0,3}, S_{1,7}, S_{0,2}\}$. In the second execution of the FOR loop in step (6) we have $m = S_{1,2}$ and $m' = S_{1,7}$. The first execution of the nested FOR loop in step (6) invokes Select_Input($\text{SYN1}, S_{1,2}, \text{SITs}$) (see Fig. 7 (b)). We have to solve the path condition “$B' = 2 \Lambda - 1 > 0$”; however, there is no solution. The second execution of the nested FOR loop invokes Select_Input($\text{SYN2}, S_{1,2}, \text{SITs}$) (see Fig. 7 (c)). We have to solve the path condition “$A > 0$”. We obtain another input $(a = 1, b = 0)$, and now $\Delta = \{S_{0,3}, S_{1,7}, S_{0,2}, S_{1,2}\}$.

In the third execution of the FOR loop in step (6) we have $\Sigma - \Delta = \{S_{1,5}, S_{1,6}\}$. The first execution of the nested FOR loop activates Select_Input($\text{SYN1}, S_{1,5}, \text{SITs}$) and we have $m = S_{1,5}$ and $m' = S_{1,6}$. Since no SYN-sequence in $\Gamma$ contains $S_{1,6}$, this FOR loop terminates. In the fourth execution of the FOR loop in step (6) we have $m = S_{1,5}$ and $m' = S_{1,5}$. Since no SYN-sequence in $\Gamma$ contains $S_{1,5}$, this FOR loop terminates. In step (7), $\Omega = \{(a = 0, b = 2), (a = 1, b = 0)\}$. We go back to step (3) with $I = (a = 0, b = 2)$. The execution of ReachabilityTesting($P, (a = 0, b = 2)$) yields two SYN-sequences SYN3 and SYN4:

$\text{SYN3}[0] = \{R(b, 0, S_{0,0}, 2, t1), W(a, 1, S_{0,3}, -2)\}$
\( \Delta = \Delta \cup \text{All\_IAB\_Statements\_Reached}({\text{SYN3,SYN4}}) = \{S_0, S_1, S_2, S_3, S_4\} \); thus, \( \Sigma = \Delta \cup \{S_0, S_1, S_2, S_3, S_4\} \). In the first execution of the FOR loop in step (6) we have \( m = S_1 \) and \( m' = S_2 \). Since no SYN-sequence in \( \Gamma \) contains \( S_1 \), this FOR loop terminates. In the fourth execution of the FOR loop in step (6) we have \( m = S_3 \) and \( m' = S_4 \). Since no SYN-sequence in \( \Gamma \) contains \( S_3 \), this FOR loop terminates. In step (7) we have \( \omega = \{a = 1, b = 0\} \). We go back to step (3) with \( I = (a = 1, b = 0) \). The execution of ReachabilityTesting\( (P, (a = 1, b = 0)) \) yields three SYN-sequences SYN1, SYN5, and SYN6, where SYN5 and SYN6 are

- SYN5[0] = \{R(a, 0, S_0, 2, t_1), W(a, 1, S_1, 1)\}
- SYN5[1] = \{R(a, 0, S_1, 1, t_2), W(a, 1, S_2, 1), R(a, 1, S_3, -1, t_3), W(d, 1, S_6, 0)\}
- SYN6[0] = \{R(b, 0, S_0, 2, t_1), W(a, 1, S_1, -1)\}
- SYN6[1] = \{R(a, 0, S_1, 1, t_2), W(a, 1, S_2, 1), R(a, 0, S_3, 1, t_3), W(d, 1, S_6, 0)\}.

\( \Delta = \Delta \cup \text{All\_IAB\_Statements\_Reached}({\text{SYN1,SYN5,SYN6}}) = \{S_0, S_1, S_2, S_3, S_4, S_5\} \); thus, \( \Sigma = \Delta \cup \{S_5\} \). In the first execution of the FOR loop in step (6) we have \( m = S_5 \) and \( m' = S_6 \). The first execution of the nested FOR loop activates Select\_Input(SYN5, S_5, SITs) (see Fig. 7 (d)). We have to solve the path condition ”\( b! = 2 \Lambda A > 0 \Lambda - 1 = -2 \)”; however, there is no solution. In the third execution of the nested FOR loop we invoke Select\_Input(SYN6, S_5, SITs) (see Fig. 7 (e)). We have to solve the path condition ”\( A > 0 \Lambda A = -2 \)”; however, there is no solution. In step (7) we have \( \omega = \emptyset \); the algorithm terminates, which means that we cannot find an input to reach \( S_5 \).

---

(a) | Select\_Input(SYN1, S_0, SITs) | Symbolic values | Path condition |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 = b )</td>
<td>( t_1 = B )</td>
<td>True</td>
<td></td>
</tr>
<tr>
<td>if( (t_1 = 2) T )</td>
<td>( t_1 = B )</td>
<td>( B = 2 )</td>
<td></td>
</tr>
</tbody>
</table>

(b) | Select\_Input(SYN1, S_1, SITs) | Symbolic values | Path condition |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 = b )</td>
<td>( t_1 = B )</td>
<td>True</td>
<td></td>
</tr>
<tr>
<td>if( (t_1 = 2) F )</td>
<td>( t_1 = B )</td>
<td>( B! = 2 )</td>
<td></td>
</tr>
<tr>
<td>( a = -1 )</td>
<td>( t_1 = B, a = -1 )</td>
<td>( B! = 2 )</td>
<td></td>
</tr>
<tr>
<td>( \ell_2 = a )</td>
<td>( t_1 = B, a = -1, \ell_2 = -1 )</td>
<td>( B! = 2 )</td>
<td></td>
</tr>
<tr>
<td>if( (\ell_2 &gt; 0) T )</td>
<td>( t_1 = B, a = -1, \ell_2 = -1 )</td>
<td>( B! = 2 \Lambda - 1 &gt; 0 )</td>
<td></td>
</tr>
</tbody>
</table>

(c) | Select\_Input(SYN2, S_1, SITs) | Symbolic values | Path condition |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ell_2 = a )</td>
<td>( \ell_2 = A )</td>
<td>True</td>
<td></td>
</tr>
<tr>
<td>if( (\ell_2 &gt; 0) T )</td>
<td>( \ell_2 = A )</td>
<td>( A &gt; 0 )</td>
<td></td>
</tr>
</tbody>
</table>
2.3 Derive Inputs for Statement Coverage using an Exhaustive Method

The running example shown in Section 2.2 demonstrates that Algorithm 3 cannot derive a set of inputs to achieve statement coverage for a given concurrent program. We therefore develop another method, as shown in Algorithm 4. Step (6) of this algorithm attempts to derive inputs from all the IAB statements that cannot be reached in a single SYN-sequence.

Algorithm 4: An exhaustive method to derive inputs to achieve statement coverage

Input: A concurrent program $P$. Assume that $\Sigma$ is the set of all the IAB statements in all the processes of $P$ and SITs includes the SITs (the SITs of all the processes in concurrent program $P$).

Output: A set of inputs, $\Omega$.

Exhaustive_select_inputs($P$)

1. Let $\Omega = \emptyset$ and $\Delta = \emptyset$.
2. Arbitrarily select an input $X$ of $P$. Let $\Omega = \Omega \cup X$.
3. Arbitrarily remove an input $I$ from $\Omega$.
4. Let $\Gamma = \text{ReachabilityTesting}(P, I)$.
5. $\Delta = \Delta \cup \text{All\_IAB\_Statements\_Reached}(\Gamma)$. IF ($\Delta = \Sigma$) THEN return.
6. FOR each SYN-sequence $S$ in $\Gamma$
   
   FOR each IAB statement $m$ in ($\Sigma - \text{All\_IAB\_Statements\_Reached}(S)$)
   
   Assume that $m'$ and $m$ have the same previous statement.
   
   IF ($S$ reaches statement $m'$) THEN
   
   $\alpha = \text{Select\_Input}(S, m, \text{SITs})$.
IF ($\alpha \neq \emptyset$ AND $\alpha$ is a new input) THEN
  $\Omega = \Omega \cup \alpha$.
  $\Delta = \Delta \cup m$.
END IF
END FOR
END FOR

(7) IF ($\Omega \neq \emptyset$) THEN GOTO step (3).

We again employ the concurrent program shown in Fig. 6 to demonstrate Algorithm 4. First, we choose $X$ to be $(a = 0, b = 0)$ in step (2). The execution of ReachabilityTesting ($P, (a = 0, b = 0)$) yields two SYN-sequences $SYN1$ and $SYN2$. $\Sigma - \Delta = \{S_{0,2}, S_{1,2}, S_{1,5}, S_{1,6}\}$. In the first execution of the FOR loop in step (6) we have $S = SYN1$. $\Sigma - ll_{IAB\_Statements\_Reached}\{SYN1\} = \{S_{0,2}, S_{1,2}, S_{1,5}, S_{1,6}\}$. We invoke $Select\_Input()$ twice:

- $Select\_Input(SYN1, S_{0,2}, SITs)$: We obtain an input $(a = 0, b = 2)$.
- $Select\_Input(SYN1, S_{1,2}, SITs)$: There is no solution.

Now $\Omega = \{(a = 0, b = 2)\}$. In the second execution of the FOR loop in step (6) we have $S = SYN2$. $\Sigma - ll_{IAB\_Statements\_Reached}\{SYN2\} = \{S_{0,2}, S_{1,2}, S_{1,5}, S_{1,6}\}$. We invoke $Select\_Input()$ twice:

- $Select\_Input(SYN2, S_{0,2}, SITs)$: We obtain an input $(a = 0, b = 2)$. This input has been derived previously, and so it is not added to $\Omega$.
- $Select\_Input(SYN2, S_{1,2}, SITs)$: We obtain an input $(a = 1, b = 0)$.

Now $\Omega = \{(a = 0, b = 2), (a = 1, b = 0)\}$. We go back to step (3) with $I = (a = 0, b = 2)$. The execution of ReachabilityTesting ($P, (a = 0, b = 2)$) yields two SYN-sequences $SYN3$ and $SYN4$. $\Delta = \Delta \cup ll_{IAB\_Statements\_Reached}\{SYN3, SYN4\} = \{S_{0,3}, S_{0,2}, S_{1,7}, S_{1,2}\}$; thus, $\Sigma - \Delta = \{S_{1,5}, S_{1,6}\}$. In the first execution of the FOR loop in step (6) we have $S = SYN3$. $\Sigma - ll_{IAB\_Statements\_Reached}\{SYN3\} = \{S_{0,3}, S_{1,2}, S_{1,5}, S_{1,6}\}$. We invoke $Select\_Input()$ twice:

- $Select\_Input(SYN3, S_{0,3}, SITs)$: We obtain an input $(a = 0, b = 0)$. This input has been derived previously, and so it is not added to $\Omega$.
- $Select\_Input(SYN3, S_{1,2}, SITs)$: There is no solution.

For $m = S_{1,5}$ and $m = S_{1,6}$ we cannot find previous statement $m'$ in $SYN3$. In the second execution of the FOR loop in step (6) we have $S = SYN4$. $\Sigma - ll_{IAB\_Statements\_Reached}\{SYN4\} = \{S_{0,3}, S_{1,2}, S_{1,5}, S_{1,6}\}$. We invoke $Select\_Input()$ twice:

- $Select\_Input(SYN4, S_{0,3}, SITs)$: We obtain an input $(a = 0, b = 0)$. This input has been derived previously, and so it is not added to $\Omega$.
- $Select\_Input(SYN4, S_{1,2}, SITs)$: We obtain an input $(a = 1, b = 2)$. 


Now $\Omega = \{(a = 1, b = 0), (a = 1, b = 2)\}$. We go back to step (3) with $I = (a = 1, b = 0)$. The execution of ReachabilityTesting($P, (a = 1, b = 0)$) yields two SYN-sequences SYN5 and SYN6. $\Delta = \Delta \cup \text{All IAB Statements Reached}(\text{SYN5, SYN6}) = \{S_{0,3}, S_{0,2}, S_{1,7}, S_{1,5}, S_{1,6}\}$; thus, $\Sigma - \Delta = \{S_{1,5}\}$. In the first execution of the FOR loop in step (6) we have $S = \text{SYN5}$. $\Sigma - \text{All IAB Statements Reached}(\{\text{SYN5}\}) = \{S_{0,2}, S_{1,5}, S_{1,7}\}$. We invoke $\text{Select Input()}$ three times:

- $\text{Select Input(SYN5, S_{0,2}, SITs)}$: We obtain an input $I = (a = 0, b = 2)$. This input has been derived previously, and so it is not added to $\Omega$.
- $\text{Select Input(SYN5, S_{1,5}, SITs)}$: There is no solution.
- $\text{Select Input(SYN5, S_{1,7}, SITs)}$: We obtain an input $I = (a = 0, b = 0)$. This input has been derived previously, and so it is not added to $\Omega$.

In the second execution of the FOR loop in step (6) we have $S = \text{SYN6}$. $\Sigma - \text{All IAB Statements Reached}(\{\text{SYN6}\}) = \{S_{0,2}, S_{1,5}, S_{1,7}\}$. We invoke $\text{Select Input()}$ three times:

- $\text{Select Input(SYN6, S_{0,2}, SITs)}$: We obtain an input $I = (a = 1, b = 2)$. This input has been derived previously, and so it is not added to $\Omega$.
- $\text{Select Input(SYN6, S_{1,5}, SITs)}$: There is no solution.
- $\text{Select Input(SYN5, S_{1,7}, SITs)}$: We obtain an input $I = (a = 0, b = 0)$. This input has been derived previously, and so it is not added to $\Omega$.

Now $\Omega = \{(a = 1, b = 2)\}$. We go back to step (3) with $I = (a = 1, b = 2)$. The execution of ReachabilityTesting($P, (a = 1, b = 2)$) yields two SYN-sequences SYN7 and SYN8:

$\text{SYN7}[0] = \{R(b, 0, S_{0,0}, 2, t1), W(a, 1, S_{0,2}, -2)\}$
$\text{SYN7}[1] = \{R(a, 0, S_{1,0}, 1, t2), W(c, 1, S_{1,2}, 1), R(a, 1, S_{1,3}, -2, t3), W(d, 1, S_{1,5}, 1)\}$
$\text{SYN8}[0] = \{R(b, 0, S_{0,0}, 2, t1), W(a, 1, S_{0,2}, -2)\}$
$\text{SYN8}[1] = \{R(a, 0, S_{1,0}, 1, t2), W(c, 1, S_{1,2}, 1), R(a, 0, S_{1,3}, 1, t3), W(d, 1, S_{1,6}, 0)\}$.

$\Delta = \Delta \cup \text{All IAB Statements Reached}(\text{SYN7, SYN8}) = \{S_{0,3}, S_{0,2}, S_{1,7}, S_{1,2}, S_{1,6}, S_{1,5}\}$; thus, $\Sigma - \Delta = \emptyset$; the algorithm terminates.

The greedy method listed in Algorithm 3 cannot find an input to reach statement $S_{1,6}$. We invoke Algorithm 1 seven times as follows [note that $\text{Select Input(SYN1, S_{1,5}, SITs)}$ is invoked twice because different inputs may produce the same SYN-sequence]:

$\text{Select Input(SYN1, S_{0,2}, SITs), Select Input(SYN1, S_{1,2}, SITs)}$
$\text{Select Input(SYN2, S_{1,2}, SITs), Select Input(SYN1, S_{1,5}, SITs)}$
$\text{Select Input(SYN5, S_{1,5}, SITs), Select Input(SYN6, S_{1,5}, SITs)}$. 


However, in Algorithm 4 the following invocations of Algorithm 1 derive the input \((a = 1, b = 2)\) that can guide the reachability testing to reach statement \(S_{1,6}\):

\[
\text{Select\_Input(SYN4, } S_{1,2}, \text{ SITs), Select\_Input(SYN6, } S_{0,2}, \text{ SITs).}
\]

3. CORRECTNESS OF THE PROPOSED SCHEME

It is obvious from our running example that Algorithm 3 is not able to provide statement coverage. Moreover, although we have shown that Algorithm 4 can guide the reachability testing to achieve statement-coverage testing for our running example, a formal proof is needed to demonstrate its correctness. Theorem 1 indicates that Algorithm 4 can perform testing to reach a statement if there exists an input for the concurrent program to reach it; that is, statements that are not reached in the testing process guided by Algorithm 4 are dead codes.

**Theorem 1:**  \(P\) is a concurrent program and \(m\) is a statement of one of the processes in \(P\). If \(m\) can be reached by executing \(P\) with some input, then the application of Algorithm 4 can find an input for reachability testing to reach \(m\) if all the path conditions derived from SYN-sequences can be solved and the concurrent program contains no dead code.

**Proof:** Since \(m\) can be reached by executing \(P\) with some input, there must exist a feasible SYN-sequence \(S\) that contains \(m\). We assume that \(\text{TOS}(S, m, \text{SITs}) = \alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_k\). Note that \(\alpha_{k+1}\) is a BRCH statement and there exists an input for \(m\) to be the next statement of \(\alpha_k\). When applying Algorithm 4, we first select an input \(I_0\) arbitrarily and perform \(\Gamma = \text{ReachabilityTesting}(P, I_0)\). Let \(S_p\) be the SYN-sequence that is in \(\Gamma\) and has the maximal common feasible prefix\(^6\) of \(S\). That is, \(\text{TOS}(S_p, \alpha_q, \text{SITs})\) is the longest in \(\text{TOS}(S', \alpha, \text{SITs})\), where \(S' \in \Gamma\), \(\alpha\) is an IAB statement, and \(\alpha_q \in \text{TOS}(S, m, \text{SITs})\).

Assume that \(\text{TOS}(S_p, \alpha_q, \text{SITs}) = \alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_k\). Note that \(\alpha_{k+1}\) is the next statement of \(\alpha_k\) and \(\alpha_q \neq \alpha_{k+1}\). Step (6) of Algorithm 4 will explore each IAB statement that is not reached by the SYN-sequence. Since \(S\) is a feasible SYN-sequence of \(P\), there must exist a solution to the path condition of \(\alpha_k\) to reach \(\alpha_{k+1}\). Assume that the solution is \(I_1\); then steps (3) and (4) are repeated. The application of \(\text{ReachabilityTesting}(P, I_1)\) will yield at least one SYN-sequence that reaches \(\alpha_{k+1}\). Repeating this process produces longer feasible prefixes of \(\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_k, \alpha_{k+1}, \ldots, \alpha_{n-1}\) and must obtain \(\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_k, \alpha_{k+1}, \ldots, \alpha_{n-1}\) by applying a finite number of reachability testing.

4. EXPERIMENTAL RESULTS

We designed a preliminary implementation of Algorithm 3 and 4. The reachability testing was implemented according to the race generator in http://www.csie.ntnu.edu.tw/~ghhwang/DET/DET_RV_gen.zip\(^7\). Table 2 lists the results obtained when applying Algorithm 3 and Algorithm 4 to guide the reachability testing to achieve statement coverage in the dynamic testing of the concurrent program shown in Fig. 6. The greedy method applied fewer reachability tests than did the exhaustive method, but it could only

---

\(^6\) See Definition 5 in [15] about the definition of maximal common feasible prefix.

\(^7\) Refer to [17] for details.
derive two inputs, and one statement could not be reached. The exhaustive method derived one more input than the greedy method and could reach all the statements in the target concurrent program, but it performed more duplicate tests during the testing process.

Table 2. Comparison of Algorithm 3 and Algorithm 4 for the concurrent program in Fig. 6.

<table>
<thead>
<tr>
<th></th>
<th>Algorithm 3 Greedy method</th>
<th>Algorithm 4 Exhaustive method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of times that reachability testing was applied</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Total number of generated SYN-sequences</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Number of times that Select_Input() was invoked</td>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>Total number of derived inputs</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Number of statements not reached</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The concurrent program shown in Fig. 8 is a previously reported running example [21]. The original message-passing concurrent program was translated into a shared-memory concurrent program. It is simpler than our running example shown in Fig. 6 because it does not have a nested if-then-else branch structure. Referring to Table 3. Although Select_Input() is invoked more often by the exhaustive method than by the greedy method, the two schemes derive the same input set and the greedy method can provide statement coverage.

Fig. 8. Example of a concurrent program with two processes (from [19]).

5. RELATED WORK

Many studies have applied symbolic execution to obtain statement-coverage testing for sequential programs [1, 19, 20], but few studies have investigated how to achieve statement coverage for concurrent programs. Sen and Agha presented a testing scheme that uses simultaneous concrete and symbolic executions to consider both inputs and schedules for message-passing distributed programs [21]. Their proposed algorithm first generates a random input and a macro-step schedule that specifies the order in which the processes will be executed, and then performs the following steps within a loop: execut-
Table 3. Comparison of Algorithm 3 and Algorithm 4 for the concurrent program in Fig. 8.

<table>
<thead>
<tr>
<th></th>
<th>Algorithm 3 Greedy method</th>
<th>Algorithm 4 Exhaustive method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of times that reachability testing was applied</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Total number of generated SYN-sequences</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Number of times that <code>Select_Input()</code> was invoked</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>Total number of derived inputs</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Number of statements not reached</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Hwang proposed the idea of reachability testing for dealing with the problem of nondeterministic behavior of concurrent programs [11]. Reachability testing is an approach for the dynamic testing of a concurrent program that explores different interleavings or SYN-sequences without static analysis of the target source program. The SYN-sequences generated during testing are analyzed so that the subsequent tests explore different interleavings, thereby producing different SYN-sequences. Hwang showed how to perform reachability testing of concurrent programs that use read and write operations on shared memory and semaphores for synchronization [11]. Tai described how to apply reachability testing to concurrent programs that use message passing [23], while Lei and Tai showed how to exercise all the possible SYN-sequences for a message-passing program [24]. However, their approach still requires the sequences/variants to be stored in order to prevent a sequence from being exercised more than once. Another reachability testing algorithm exercises all the (partially ordered) SYN-sequences of a program exactly once, without requiring the sequences to be stored [25, 26]. A general execution model has been described that applies that algorithm to programs that use semaphores, locks, monitors, and/or message passing [24]. Hwang et al. described how to apply reachability testing to the exhaustive testing of a client/server database application that exhibits nondeterministic behavior [15]. The design and implementation of a distributed reachability testing algorithm for a cluster of workstations has also been presented [22]. Lei et al. presented a testing strategy, called t-way reachability testing, that selectively...
exercises a subset of SYN-sequences [27].

Carver and Lei presented an algorithm for stateful reachability testing, and showed how to store and recognize visited states when reachability testing is applied to a single monitor [28]. It has been proven that reachability testing can derive all the feasible SYN-sequences of a concurrent program if the program has a finite number of SYN-sequences for a given input [11, 15]. Lin and Hwang proposed a method for stateful reachability testing that can perform state-cover testing for nondeterministic terminating concurrent programs with an infinite number of SYN-sequences under a single input [17]. To the best of our knowledge, this paper is the first to propose a systematic scheme that can guide reachability testing to obtain statement-coverage testing for concurrent programs.

6. CONCLUSION

Previous studies had shown that reachability testing is able to generate feasible interleavings of a concurrent program with a given input. Arbitrarily choosing inputs cannot guarantee statement-coverage testing for the target concurrent program. In this paper we propose a scheme that can automatically derive a set of inputs to guide reachability testing in order to perform statement-coverage testing. We have presented algorithms to derive the path conditions from SYN-sequences obtained in reachability testing. A formal proof demonstrates that statement coverage is achieved if all the path conditions can be solved by the constraint solver and the concurrent program contains no dead code. Thus, the proposed scheme can also be used to discover dead codes in a given concurrent program.

REFERENCES

10. K. C. Tai, “Testing of concurrent software,” in *Proceedings of the 13th Annual In-

Gwan-Hwan Hwang (黃冠寰) is a Professor and Chairman in the Department of Computer Science and Information Engineering at National Taiwan Normal University, Taiwan. He received the B.S. and M.S. degrees while in the Department of Computer Science and Information Engineering at National Chiao Tung University, in 1991 and 1993, respectively, and the Ph.D. degree while in the Department of Computer Science at National Tsing Hua University, Hsinchu, Taiwan, in 1998. His research interests include internet security, concurrent software testing, groupware, service-oriented architecture, and cloud computing.

Heng-Yi Lin (林恆毅) received the M.S. degree in the Department of Computer Science and Information Engineering at National Taiwan Normal University, Taiwan, in 2012. He has been an Engineer in the Department of Connected Service of HTC Co., Ltd., Taiwan, since 2012. His research interests include concurrent program testing and Android application development.

Shao-Yan Lin (林劭彥) received the M.S. degree in the Department of Computer Science and Information Engineering at National Taiwan Normal University, Taiwan, in 2011. His research interests include concurrent software testing and validation.

Che-Sheng Lin (林哲生) received the Ph.D. degree in the Department of Computer Science and Information Engineering at National Taiwan Normal University, Taiwan, in 2013. His research interests include concurrent software testing, Java memory model, and object-oriented technologies.