Automatic Test Generation and Monitoring of Infinite States Systems

Doron Peled

University of Warwick, Department of Computer Science, Coventry CV4 7AL, UK

Abstract. Testing is the most commonly used method for debugging software. We survey here several techniques that are used for assisting the tester in generating a test suite that reflects his intuition about potential problems in the software. Then we show how to enforce and monitor these test cases.

Keywords. Model Checking, Testing, Weakest Precondition, Uniform Distribution,

1. Introduction

Software code can be developed at a rate of tens of lines of code per hour. Statistics shows also a rate of several software bugs per hour of coding, even for an experienced programmer. There are several approaches for obtaining software reliability. The oldest, and still most commonly used is software testing, where the code is exercised in a controlled way, in order for errors to manifest themselves. The main goal is to minimize the number of test cases, i.e., the size of the test suite and maximize the chance for software bugs to appear. This is obviously a difficult tradeoff.

Verification through deductive theorem proving [3,4] is historically the first method that is considered to be formal. By introducing a logic that includes both code and assertions about its behavior, one is capable of using axioms and proof rules for verifying the correctness of the software with respect to various properties (e.g., termination and mutual exclusion). This approach is also related to the introduction of formal semantics, where the meaning of code is formally defined in mathematical terms.

A more pragmatic approach is offered with model checking [2], where finite states systems are algorithmically analyzed. Model checking is fairly automatic and requires less time and human skills than deductive theorem proving. It has gained a lot of popularity among the hardware development community. However, inclusion of even a small number of word-size variables in the code can potentially generate an enormous state space, where memory and time would not be sufficient to complete the verification. Various reduction techniques (e.g., [10]) can be used in some cases to combat the state space explosion.

Obviously, there is a tradeoff in the benefits and difficulties that are offered by these methods. Our approach is to combine different formal methods in an attempt to min-

---

This research was partially supported by Subcontract UTA03-031 to The University of Warwick under University of Texas at Austin’s prime National Science Foundation Grant #CCR-0205483.
imize their difficulties and maximize the overall advantages. Of course, some of these limitations cannot be removed. Verifying the termination of a program without considering the limit on word size is undecidable. But note that when assuming a given limit on word size, the verification becomes machine dependent. Moreover, even if some realistic variable length is assumed (e.g., 32 or 64 bits), the size of the state space, and hence the complexity to perform the analysis is enormous. Word size is not the only cause for complexity problems; checking the simplest properties of concurrent systems, such as deadlocks, is in PSPACE complete in the the number of processors.

The techniques presented here exploit ideas from model checking and deductive verification in order to select a small test suite with a high probability of finding errors. Common testing tools help in generating test cases, performing the testing on the code and recording the testing activities and results. Our goal is to provide testing tools that perform deeper analysis than most existing tools. In particular, we would like to address the following:

Selection of test cases. We are using ideas from deductive verification to assist in selecting test cases. In particular, the symbolic computation of weakest precondition can assist in generating a condition for executing the code according to a particular path. This allows generating initial values for testing that path.

Adding temporal specification. Temporal specification is used to guide the selection of the test cases. The temporal assertion corresponds to the intuition that the tester has regarding the suspected cause for potential errors, rather than to a specific property that needs to be verified.

Execution of test cases. When testing concurrent systems, the choice of initial values does not prohibit the possibility that due to nondeterministic choice a different behavior than the one expected will occur. Synchronization is added in order to guarantee the appropriate execution.

Calculating the probability of a test case. Adding synchronization may guarantee that a particular behavior is selected despite nondeterministic choice. However, it also changes the tested code. If such a change is not acceptable, we may apply some calculation for the probability of executing the selected path. We perform such calculations with minimal assumption about the tested system.

2. Preliminaries

A state of a program is a function assigning values to every program variable according to their defined domains. An augmented state includes also assignments of values to the program counters. We denote the fact that a state \( g \) satisfies a first order assertion \( \varphi \) by the standard notation \( g \models \varphi \).

A flow chart of a program or a procedure is a graph, with nodes corresponding to assignments and tests taken, and edges reflect the flow of control between the nodes. A flow chart can be obtained by automatic compilation of the code. There are several kinds of nodes in a flow chart: a box containing an assignment, a diamond containing a condition, and an oval denoting the beginning or end of the program (procedure). Edges exiting from a diamond node are marked with either ‘yes’ or ‘no’ to denote the success or failure of the condition, respectively. Each node has a unique program counter value.
This value can be a label that is provided with the code, or automatically generated by a tool. Passing an edge out of one node and into another entails a corresponding change of the program counter value. A path of a program is a consecutive sequence of nodes in the flow chart.

An execution of a flow chart is a sequence of states \( g_1, g_2, \ldots, g_n \), where each state \( g_{i+1} \) is obtained from its predecessor \( g_i \) by executing a transition associated with a flow chart node, as described below. This means that the condition for the transition to execute holds in \( g_i \), and the transformation associated with the transition is applied to it. We call an execution that consists of augmented states an augmented execution. The projection of an augmented execution on the program counter values results in program counter values (labels) along a path in the flow chart. Thus, in general, a path may correspond to multiple executions, with the same sequence of program counter values.

Let \( \tau = t_1, t_2, \ldots, t_n \) be a path in a flow chart. Let \( \rho = g_1, g_2, \ldots, g_n \) be a sequence of non augmented program states (i.e., not including the program counter values). The Sequence \( \rho \) is an execution of \( \tau \) if for each \( 1 \leq i < n \) we have:

1. \( t_i \) is a diamond node, with condition \( c \) and the edge from \( t_i \) to \( t_{i+1} \) is marked with ‘yes’. Then \( g_i \models c \) and \( g_{i+1} = g_i \).
2. \( t_i \) is a diamond node, with condition \( c \) and the edge from \( t_i \) to \( t_{i+1} \) is marked with ‘no’. Then \( g_i \models \neg c \) and \( g_{i+1} = g_i \).
3. \( t_i \) is an assignment node labeled \( x := e \). Then \( g_{i+1} = g_i[x/e[g_i]] \), which denotes that \( g_{i+1} \) is the same as \( g_i \), except for the variable \( x \), which has the value of \( e \), interpreted according to the state \( g_i \).

In fact, we can also calculate \( g_{n+1} \), the state after \( t_n \), according to the above rules. However, sometimes we select not to include \( g_{n+1} \), since conceptually, if \( t_n \) is a diamond, the path does not specify whether its condition holds or not. Thus, the last node can sometimes serve merely to fix the exit edge out of a previous diamond node. The set of executions of a path \( \tau \) is denoted \( \text{exec}(\tau) \).

We use Linear Temporal Logic (LTL) syntax, as follows:

\[
\varphi ::= (\varphi) \mid \neg \varphi \mid \varphi \lor \varphi \mid \varphi \land \varphi \mid \Box \varphi \mid \Diamond \varphi \mid \varphi U \varphi \mid \varphi V \varphi \mid p
\]

where \( p \in \mathcal{P} \), with \( \mathcal{P} \) a set of basic formulas. Each proposition can be replaced by a (quantifier free) propositional formula. In particular, in our context, such formulas will refer (separately, not within the same predicate or function symbol) to the program variables and program counters. For a propositional sequence \( \sigma \) over \( 2^\mathcal{P} \), we denote the \( i \)th state (where the first state is numbered 0) by \( \sigma(i) \), and the suffix starting from the \( i \)th state by \( \sigma(i) \). Let \( |\sigma| \) be the length of the sequence \( \sigma \), which is a natural number. The semantic interpretation of LTL over finite sequences is as follows:

- \( \sigma \models C \varphi \) iff \( |\sigma| > 1 \) and \( \sigma(1) \models \varphi \).
- \( \sigma \models \varphi U \psi \) iff \( \sigma(j) \models \psi \) for some \( 0 \leq j < |\sigma| \) so that for each \( 0 \leq i < j \), \( \sigma(i) \models \varphi \).
- \( \sigma \models \neg \varphi \) iff it is not the case that \( \sigma \models \varphi \).
- \( \sigma \models \varphi \lor \psi \) iff either \( \sigma \models \varphi \) or \( \sigma \models \psi \).
- \( \sigma \models p \) iff \( |\sigma| > 0 \) and \( \sigma(0) \models p \).

The rest of the temporal operators can be defined using the above. In particular, \( \overline{\Box} \varphi = \neg \Box \neg \varphi, \varphi \land \psi = \neg((\neg \varphi) \lor (\neg \psi)), \varphi \lor \psi = \neg((\neg \varphi) \land (\neg \psi)), \text{true} = p \lor \neg p, \)
false = p ∧ ¬p, □φ = false ∨ φ, and ◻φ = true ∪ φ. The operator □ is a ‘weak’ version of the ◻ operator. Whereas ◻φ means that φ holds in the suffix of the sequence starting from the next state, □φ means that if the current state is not the last one in the sequence, then the suffix starting from the next state satisfies φ. Note that

\((◻φ) ∧ (◻ψ) = ◻(φ ∧ ψ)\)

since ◻φ already requires that there will be a next state. Another interesting observation is that the formula ◻false holds in a state that has deadlock or termination.

3. Selection of Test Cases via Path Conditions

Software testing allows sampling programs in order to find design and coding errors. It is impossible or impractical to check all the behaviors of a program. A tester thus may benefit enormously from a computerized assistance in generating test cases. Such a selection can be based on coverage criteria [11] in an attempt to sparse out the set of tests in order to reduce their number yet maintain a good probability of detecting errors. For example, we may want to cover all the flowchart nodes, the edges, or the Boolean combinations of test predicates [9].

We will describe how to assist in finding test cases based on a given path in the flow chart of the program based on [6]. This will help in generating test cases for unit testing. A path condition is a first order predicate that expresses the condition to execute the path, when starting from a given node (a path condition does not refer to the program counter values). Denote the path condition for a path τ by Γτ. Formally, this means that when starting executing from the first node t1 of τ, each execution ρ of τ must begin with a state g1 such that g1 |∈ Γτ. Moreover, when starting the execution at node t1 with a state satisfying Γτ, we can extend g1 to an execution ρ of τ. In deterministic code, there is at most one way of extending a state into an execution sequence. Hence, in this case, all the executions that start with a state satisfying Γτ are executions of the path τ (or paths with prefix τ).

Backwards calculation of Path Conditions

Define the accumulated path condition as the condition to move from the current point in the calculation to the end of the path. The current point moves at each step in the calculation of the path condition backwards, over one node to the previous point (edge). Initially, the accumulated path condition is set to true, at the end of the path (i.e., after the last node, with an empty path). When we pass over a diamond node (on our way back), we either conjoin it as is, or conjoin its negation, depending on whether we exited this node with a yes or no edge, respectively. When we pass an assignment, we “relativize” the path condition φ with respect to it; if the assignment is of the form \(x := e\), where \(x\) is a variable and \(e\) is an expression, we substitute \(e\) instead of each free occurrence of \(x\) in the path condition. This is denoted by \(φ[e/x]\).

One can also calculate path conditions forwards (see e.g., [6]). The choice of direction for calculating the path condition is affected by the direction in which the path is obtained (e.g., due to the intersection algorithm). We may want to calculate the path conditions incrementally for the prefixes of that path, or the suffixes. The reason is that path
condition calculation is rather expensive, and we may benefit from discarding a prefix (or a suffix, respectively) ‘on-the-fly’.

An extension of this method of calculating the path condition to the case of real time systems appears in [1]. There, each transition includes also time constrains (similar to those in Section 5). The condition to execute a path (actually, a partial order consistent with the path, as in Section 4) can include also timing constraints that were left parametric. One can sometimes also use these timing constraints to control the selection of the test cases.

Using Temporal Specification to Guide Test Selection

Various coverage criteria are used in order to generate test cases. Such criteria do not take into account whatever intuition and experience a tester may have about the potential location of errors. For example, array indexing is a frequent cause for problems. Texts about testing [9] suggest “rules of thumb” for dealing with various suspicious cases.

We provide a mechanism for selecting test cases based on a temporal specification that expresses the tester’s intuition about the location of errors. These specifications can mention both code locations (“program counters”) and assertions about the relationship between the program variables. The locations mentioned help to direct the test case generation through the code towards suspicious program segments or choosing to pass through different locations a particular number of times in a specific order. Thus, they limit the paths in the flow chart that are considered for the test case generation. The relations over program variables limit the different executions of the same path. Those relations affect the calculation of the path conditions. Through simplification and refutation of the resulted path condition this also affects the search, ruling out paths that have no executions satisfying the temporal property.

Using a temporal specification on program counters

The proposed use for a temporal formula is to assist a human tester in inspecting suspicious paths and executions. Denote sets of program labels using italics capitals, as in LOOP1 (for example, the labels of nodes in a particular loop), SEQ8 or IF. The predicate at l holds when the current program counter stands in front of the code labeled with l. If L = {l1, l2, … , ln}, we denote, as a shorthand, at L = at l1 ∨ at l2 ∨ … ∨ at ln. For example, we may want to test executions where control is inside LOOP1, then exit it to the code labeled by l2, then enter LOOP2 and exits it to the code labeled with l4.

\[ \psi = (\neg at \ LOOP1) \cup (at l2 \land (at \ LOOP2 \cup at l4)) \].

Recall that the interpretation of LTL formulas is over finite paths, and we are looking for finite prefixes of executions that satisfy the property.

The temporal specification ψ can be translated into a finite state automaton that accepts the same set of sequences as ψ. The algorithm is the one described in [5], and adapted to finite sequences, with further optimizations to reduce the number of nodes generated. Let \( \langle S, \Delta, I, F, L \rangle \) be a finite state automaton with nodes (states) S, a transition relation \( \Delta \subseteq S \times S \), initial nodes \( I \subseteq S \), accepting nodes \( F \subseteq S \) and a labeling function \( L \) from \( S \) to some set of labels. A run of the automaton is a finite sequence of
nodes $s_1 s_2 \ldots s_n$, where $s_1 \in I$, and for each $1 \leq i < n, (s_i, s_{i+1}) \in \Delta$. An accepting run satisfies further that $s_n \in F$.

The property automaton is $A = (S^A, \Delta^A, I^A, F^A, L^A)$. Each property automaton node is labeled by a set of negated or non-negated predicates over the program variables and program counters, as mentioned above. The flow chart can also be denoted by an automaton $B = (S^B, \Delta^B, I^B, S^B, L^B)$ (where all the nodes are accepting, hence $F^B = S^B$). Each node in $S^B$ is labeled with a single program counter value and an assignment or a condition, depending on the type of node.

The intersection between a property automaton $A$ and a flow chart $B$ is an automaton $A \times B = (S^{A \times B}, \Delta^{A \times B}, I^{A \times B}, F^{A \times B}, L^{A \times B})$, where

- the nodes $S^{A \times B} \subseteq S^A \times S^B$ have matching labels: if $(a, b) \in S^{A \times B}$ then the program counter of the flow chart node $b$ must satisfy the program counter predicates labeling the property automaton node $a$,
- the transitions are $\{(a, b), (a', b')\} | (a, a') \in \Delta^A \land (b, b') \in \Delta^B \} \cap (S^{A \times B} \times S^{A \times B})$,
- the initial states are $I^{A \times B} = (I^A \times I^B) \cap S^{A \times B}$,
- the accepting states are $F^{A \times B} = (F^A \times S^B) \cap S^{A \times B}$; thus, membership in $F^{A \times B}$ depends only on the $A$ automaton component being accepting, and
- the label on a matched pair $(a, b)$ in the intersection contains the separate labels of $a$ and $b$.

This definition takes care of sequential execution but can be extended to handle multiple flow charts that are executed concurrently. We assume that each node $s \in S^A$ of the property automaton is annotated by some set of program variables assertions whose conjunction is $\eta_s$ and some set of program counter predicates whose conjunction is $\mu_s$. This annotation is generated automatically when translating an LTL formula into an automaton.

Now consider an accepting run of the intersection of the property automaton $A$ and the flow chart $B$ of the form $\tau = t_1, t_2, \ldots, t_n$. Projecting $\tau$ over the components of the flow chart gives a path. Thus, we may observe $\tau$ as a path with some assertions $\eta_{t_i} \land \mu_{t_i}$ added to it, due to the property nodes that match the flow chart node during the intersection. We are interested in executions $\rho = g_1, g_2, \ldots, g_n$ of $\tau$ of the path $\tau$ that also satisfy the corresponding temporal property expressed using the automaton $A$, which was obtained from the specification $\psi$. Namely, for $1 \leq i \leq n, g_i = \eta_{t_i} \land \mu_{t_i}$. This means that if $A$ is constructed as a translation of an LTL property $\psi$ then $\rho \models \psi$. The fact that $g_i \models \mu_{t_i}$ is guaranteed by the matching between the labels of each pair $(a, b)$ in the intersection. We will show later how to take care that $g_i \models \eta_t$, as well. Denote a condition for executing a path $\tau$ while satisfying a temporal property $\psi$ by $\Gamma_{\tau, \psi}$. For an execution $\rho$ as above, starting with state $g_1$ we have $g_i \models \Gamma_{\tau, \psi}$. Conversely, every state satisfying $\Gamma_{\tau, \psi}$ can be extended (uniquely, in sequential code) to an execution of $\tau$ that satisfies $\psi$.

Note that since the temporal specification $\psi$ in (1) involves only the program counters but not the program variables, for each path $\tau$ there can be only two cases:

- All the executions of $\text{exec}(\tau)$ satisfy $\psi$, or
- None of the executions of $\text{exec}(\tau)$ satisfy $\psi$.

In the former case, $\Gamma_{\tau, \psi} = \Gamma_\tau$ and in the latter case $\Gamma_{\tau, \psi} = \text{false}$. By taking the intersection of the property automaton $A$, which is obtained as a translation from $\psi$ and the
flow chart $B$, the paths that are the runs of the intersection are exactly those that have all of their executions satisfying $\psi$.

We perform a DFS search that constructs the intersection of automata $A$ and $B$. We may not assume that two nodes in the flow chart with the same program counter label correspond to the same states, as they may differ because of the values of the program variables. Consequently, when reaching the same pair of nodes $(a, b)$ in the intersection, we may not immediately backtrack. One solution is to allow the user to specify a limit on the number of times that each flow chart node, i.e., a node from $S_B$, may occur on a path. Since the specification is based on finite sequences, One does not loose information by failing to identify cycles. Repeating the model checking while incrementing $n$, one can eventually cover any length of sequence. Hence, in the limit, one can cover every finite path.

By not being able to identify when states are the same, one may run into a combinatorial explosion. As a result, the number of different paths can be doubled by each diamond flow chart node (e.g., corresponding to an if statement condition) even when the paths meet at mutual nodes. Such a prohibitive case can be the result of a repeated occurrence of if-then-else statements, such as in the unfolding of the following loop:

```
while $x \neq y$ do
    if $x > y$ then $x := x - y$
    else $y := y - x$ od
```

Taking into account the program variables assertions

The specification formula (1) above mention only the program counters. Suppose that we also want to limit the test cases such that when control is at the label $l_2$ for the first time, the value of $x$ is greater or equal to the value of $y$, and when control is at the label $l_2$ for the second time, $x$ is at least twice as big as $y$. One can write this specification as follows:

$$\psi = (\neg \text{at LOOP1}) \cup (\text{at } l_2 \land x \geq y \land \bigcirc (\text{at LOOP2} \cup \text{at } l_4 \land x \geq 2 \times y))$$

(2)

The translation from a temporal formula to an automaton results in the program variables assertions $x \geq y$ and $x \geq 2 \times y$ labeling some (specifically the second and fourth) nodes of the automaton obtained as a translation for $\psi$. We need to incorporate these assertions into the calculation of the path condition $\Gamma_{r, \psi}$. According to the above construction, the conjunction of the program variables assertions labeling the property automaton nodes are assumed to hold in the path condition before the effect of the matching flow chart node.

In general, when calculating a path condition for a path $\tau$ obtained from the intersection of the property automaton $A$ for a property $\psi$ and a flow chart $B$, we need to take into account program variables assertions that appear on it (coming from the property automaton components). We can do that by transforming the path into a path $\tau'$ as follows. Observe that each node in the intersection is a pair $(a, b)$, where $a$ is a property automaton node, and $b$ is a flow chart node in the current path. From the construction above, the label of $b$ must agree with the program counter predicates in $a$. Otherwise, the path is automatically rejected (and $\Gamma_{r, \psi} = \text{false}$). We transform each such pair into two
sequential nodes. The node $b$ remains as it appears in the flow chart. We insert a new diamond node to the current path, just before $b$. The inserted node contains as its condition the conjunction of the program variables assertions labeling the node $a$ (and true if there are no program variables assertions labeling $a$). Note that the conditions on the program counter is satisfied by the matching of $a$ and $b$. The edge between the new diamond and $b$ is labeled with ‘yes’ corresponding to the case where the condition in $a$ holds. The edge that was formerly entering $b$ now enters the new diamond. A standard induction on the length of the path shows that the path condition $A_0$ obtained by this transformation is exactly $\Gamma_{\tau,0}$.

4. Testing a Concurrent Execution

An important problem in testing concurrent executions occurs due to its inherent nondeterminism; recovering an erroneous behavior on the actual program is not guaranteed even if the execution starts with an initial state satisfying the calculated path condition. Such an execution may involve some intricate scheduling and thus is hard to demonstrate. A program transformation that translates a program in a way that a specific execution can be guaranteed is described in [12]. Since the transformation implies changes to the original code, an attempt is made to minimize its effect on the original program. Any concurrency or independence between executed actions needs to be preserved by the transformation.

As an example, consider Dekker’s solution to the mutual exclusion algorithm in Figure 1. The flow chart appears in Figure 2. Starting the execution with $turn = 1$, the following execution $\sigma_1$ can be obtained, where process $P_1$ enters its critical section.

In this solution to the mutual exclusion problem, both processes proceed to signal that they want to enter their critical sections, by setting $c_1$ and $c_2$ to 0, respectively. When $turn = 1$, process $P_1$ has priority over process $P_2$. This means that process $P_2$ gives up its attempt, by setting $c_2$ to 1, while process $P_1$ insists, waiting for $c_2$ to become 1 and then enters its critical section.

$$
\begin{align*}
P_1(0) : & \text{start} \longrightarrow P_2(0) : \text{start} \longrightarrow P_1(1) : c_1 := 1 \longrightarrow \\
& P_2(1) : c_2 := 1 \longrightarrow P_2(12) : \text{true} \longrightarrow P_1(12) : \text{true} \longrightarrow \\
& P_1(2) : c_1 := 0 \longrightarrow P_2(2) : c_2 := 0 \longrightarrow P_1(8) : c_2 = 0? (\text{yes}) \longrightarrow \\
& P_2(8) : c_1 = 0? (\text{yes}) \longrightarrow P_1(7) : \text{turn} = 2? (\text{no}) \longrightarrow \\
& P_2(7) : \text{turn} = 2? (\text{yes}) \longrightarrow P_2(3) : c_2 := 1 \longrightarrow \\
& P_1(8) : c_2 = 0? (\text{yes}) \longrightarrow P_1(9) : \text{critical-1}
\end{align*}
$$

A different execution $\sigma_2$ can be obtained with the same initial state. Process $P_2$ sets $c_2$ to 0, signaling that it wants to enter its critical section. It is faster than process $P_1$, and manages also to check whether $c_1$ is 0 before $P_1$ changes it from 1. Hence $P_2$ enters its critical section.

$$
\begin{align*}
P_1(0) : & \text{start} \longrightarrow P_2(0) : \text{start} \longrightarrow P_1(1) : c_1 := 1 \longrightarrow \\
& P_2(1) : c_2 := 1 \longrightarrow P_2(12) : \text{true} (\text{yes}) \longrightarrow P_1(12) : \text{true} (\text{yes}) \longrightarrow \\
& P_2(2) : c_2 := 0 \longrightarrow P_2(8) : c_1 = 0? (\text{no}) \longrightarrow P_2(9) : \text{critical-2}
\end{align*}
$$
boolean c1, c2;
integer (1..2) turn;

\[
\begin{align*}
P_1 &:: c1:=1; \\
&\text{while true do} \\
&\begin{align*}
&\text{begin} \\
&c1:=0; \\
&\text{while c2=0 do} \\
&\begin{align*}
&\text{if turn=2 then} \\
&\begin{align*}
&\text{begin} \\
&c1:=1; \\
&\text{while turn=2 do} \\
&\begin{align*}
&\text{/* no-op */} \\
&\text{end}; \\
&c1:=0 \\
&\text{end} \\
&\text{end}; \\
&\text{/* critical section 1 */} \\
&c1:=1; \\
&\text{turn:=2} \\
&\end{align*}
\end{align*}
\end{align*}
end
\end{align*}
\end{align*}
\]

\[
\begin{align*}
P_2 &:: c2:=1; \\
&\text{while true do} \\
&\begin{align*}
&\text{begin} \\
&c2:=0; \\
&\text{while c1=0 do} \\
&\begin{align*}
&\text{if turn=1 then} \\
&\begin{align*}
&\text{begin} \\
&c2:=1; \\
&\text{while turn=1 do} \\
&\begin{align*}
&\text{/* no-op */} \\
&\text{end}; \\
&c2:=0 \\
&\text{end} \\
&\text{end}; \\
&\text{/* critical section 2 */} \\
&c2:=1; \\
&\text{turn:=1} \\
&\end{align*}
\end{align*}
\end{align*}
end
\end{align*}
\end{align*}
\]

Figure 1. Dekker’s mutual exclusion algorithm

A dependency relation \( D \subseteq A \times A \) is a reflexive and symmetric relation over the actions [8]. It captures the cases where concurrent execution of actions cannot exist. Thus, \((\alpha, \beta) \in D\) when

- \( \alpha \) and \( \beta \) are in the same process, or
- \( \alpha \) and \( \beta \) use or define (update) a mutual variable\(^1\).

This dependency corresponds to the execution model of concurrent programs with shared variables. A different dependency relation can be defined for other models of concurrency. Let \( \sigma \) be a sequence of actions from \( A \). We can index the \( k \)th time that \( \alpha \) appears on \( \sigma \) with subscript \( k \), obtaining \( \alpha_k \). We call \( \alpha_k \) the \( k \)th occurrence of \( \alpha \) in \( \sigma \). Instead of the sequence of actions \( \alpha \alpha \beta \alpha \beta \), we can equivalently denote \( \sigma \) as the sequence of occurrences \( \alpha_1 \alpha_2 \beta_1 \alpha_3 \beta_2 \). We denote the set of occurrences of a sequence \( \sigma \) by \( E_\sigma \).

In the above example, \( E_\sigma = \{\alpha_1, \alpha_2, \alpha_3, \beta_1, \beta_2\} \).

Define a binary relation \( \rightarrow_\sigma \) between occurrences on a sequence \( \sigma \). Let \( \alpha_k \rightarrow_\sigma \beta_l \) on a sequence \( \sigma \) when the following conditions hold:

1. \( \alpha_k \) occurs before \( \beta_l \) on \( \sigma \).

\(^1\)Depending on the hardware, we may allow \( \alpha \) and \( \beta \) to be concurrent even if both only use a mutual variable but none of them define it.
Thus, $\alpha_k \rightarrow_{\sigma} \beta_k$ implies that $\alpha$ and $\beta$ refer to the same variable or belong to the same process. Because of that, $\alpha_k$ and $\beta_k$ cannot be executed concurrently. According to the sequence $\sigma$, the imposed order is that $\alpha_k$ happens before $\beta_k$. Let $\rightarrow_{\sigma}^*$ be the transitive closure completion of $\rightarrow_{\sigma}$. This is a partial order, since it is transitive, asymmetric and irreflexive. The partial order view of an execution $\sigma$ is $\langle E_{\sigma}, \rightarrow_{\sigma}^* \rangle$.

We want to impose synchronization of the checked code according to the relation $\rightarrow_{\sigma}$. However, this relation may contain many pairs that are surplus to achieve the desired synchronization, which means unnecessary overhead. This can be inefficient to implement and induce many changes from the original code. We can reduce $\rightarrow_{\sigma}$ by removing pairs of actions $\alpha_k \rightarrow_{\sigma} \beta_k$ that have a chain of related (according to $\rightarrow_{\sigma}$) occurrences...
between them. The reduced relation between occurrences of $\sigma$ is denoted by $\sim_{\sigma}$. It is defined to be the (unique) relation satisfying the following conditions:

1. The transitive closure of $\sim_{\sigma}$ is $\rightarrow_{\sigma}$.
2. There are no three different elements $\alpha_k, \beta_l$, and $\gamma_m$ such that $\alpha_k \sim_{\sigma} \beta_l, \beta_l \sim_{\sigma} \gamma_m$, and $\alpha_k \sim_{\sigma} \gamma_m$.

Calculating the relation $\sim_{\sigma}$ from $\rightarrow_{\sigma}$ is simple. We can adapt the Floyd-Warshall algorithm [4] for calculating the transitive closure of $\rightarrow_{\sigma}$. Each time a new edge is discovered as a combination of existing edges (even if this edge already exists in $\rightarrow_{\sigma}$), it is marked to be removed. Subsequently, we remove all marked edges.

It is sufficient to maintain the synchronization according to the $\sim_{\sigma}$ order. We also do not need to synchronize between occurrences that belong to the same process. The rest of the orders according to $\rightarrow_{\sigma}$ are guaranteed by transitivity and by the sequentiality of the individual processes.

Figure 3 consists of the graph of occurrences that correspond to the execution sequence $\sigma_2$ earlier in this section. The nodes in this figure are labeled according to the sequence occurrences listed in $\sigma_2$. The arrow from node 3 to 8 corresponds to the update and use of the same variable ($\beta_1$) by the different processes (and according to $\sim_{\sigma}$), while the rest of the arrows correspond to ordering between actions belonging to the same process.

Although an execution is represented by a sequence of occurrences, we are in general interested in a collection of equivalent executions. To define the equivalence between executions, let $\sigma|_{\alpha, \beta}$ be the projection of the sequence $\sigma$ that keeps only occurrences of $\alpha$ and $\beta$. Then $\sigma \equiv_D \rho$ when $\sigma|_{\alpha, \beta} = \rho|_{\alpha, \beta}$ for each pair of interdependent actions $\alpha$ and $\beta$, i.e., when $(\alpha, \beta) \in D$. This also includes the case where $\alpha = \beta$, since $D$ is reflexive. This equivalence is also called partial order equivalence or trace equivalence [8]. It relates sequences of occurrences $\sigma$ and $\rho$ for the following reasons:

- The same occurrences appear in both $\sigma$ and $\rho$.
- Occurrences of actions of a single process are interdependent (all the actions of a single process use and define the same program counter). Thus, each process executes according to both $\sigma$ and $\rho$ the same actions in the same order. This represents the fact that processes are sequential.
- Any pair of dependent actions from different processes cannot be executed concurrently, and must be sequenced. Their relative order is the same according to both $\sigma$ and $\rho$.
- Occurrences of independent actions that are not separated from each other by some sequence of interdependent actions can be executed concurrently. They may appear in different orders in trace-equivalent executions. In enforcing an execution, we do not want to impose new synchronizations that will order actions that
can be executed concurrently. Distinguishing between two equivalent executions can only be done by having global clocks and making experiments that are expensive and unnatural to concurrent systems.

Transforming Shared Variables Programs

We assume a computational model of concurrent processes with shared variables. Each process is programmed in some sequential programming language. Although no explicit nondeterministic choice construct is used here, an overall nondeterministic behavior of the program can be the result of the fact that the processes can operate in different relative speeds. Hence, when we repeatedly start the execution with exactly the same initial state, we may encounter different behaviors as the sequences $\sigma_1$ and $\sigma_2$ demonstrate. The goal is then to enforce, under the given initial condition, that the program executes in accordance with the suspicious behavior.

In order to perform the transformation, the code is translated into a set of actions. Code is added to the existing actions. Since this will result in rather large actions, which cannot realistically be executed atomically, we split them in a way that is detailed and explained below.

For each pair of processes $p_i$ and $p_j$, $p_i \neq p_j$, such that for some occurrences $\alpha_k$ with $\alpha \in A(p_i)$, and $\beta_i$ with $\beta \in A(p_j)$, $\alpha_k \sim \sigma \beta_i$, one defines a variable $V_{ij}$, initialized to 0. It is used by process $p_i$ to inform process $p_j$ that it can progress. This is done in the style of the usual semaphore operations. The process $p_i$ does that by incrementing $V_{ij}$ after executing $\alpha_k$.

$$\text{Free}_{ij} : V_{ij} := V_{ij} + 1$$

The process $p_j$ waits for the value of this variable to be 1 and then decrements it.

$$\text{Wait}_{ji} : \text{wait } V_{ij} > 0; V_{ij} := V_{ij} - 1$$

Let $ Oc(p_i) \subseteq A(p_i)$ be the set of actions of process $p_i$ that have an occurrence in $\sigma$ and are related by $\sim \sigma$ to an occurrence of an action in another process. Thus, $Oc(p_i)$ are the actions that have some (but not necessarily all) occurrences that need to be synchronized. Thus, we need to check whether we are currently executing an occurrence of an action $\alpha \in S(P_i)$ that requires synchronization. Let $count_i$ be a new local counter variable for process $p_i$. We increment $count_i$ before each time an action from $Oc(p_i)$ occurs, i.e., add the code

$$count_i := count_i + 1$$

immediately after the code for $\alpha$.

We define $\sharp_\sigma \alpha_k$ to be the number of occurrences from $Oc(p_i)$ that appeared in $\sigma$ before $\alpha_k$. We can easily calculate $\sharp_\sigma \alpha_k$ according to the sequence $\sigma$. This is also the the value that the variable $count_i$ has during the execution of the code after the transformation, due to the increment statement in (3).

Suppose now $\alpha_k \sim \sigma \beta_i$, where $\alpha \in A(p_i)$, $\beta \in A(p_j)$, $p_j \neq p_j$. Then we add the following code after $\alpha_k$:

$$\text{if } count_i = \sharp_\sigma \alpha_k \text{ then } \text{Free}_{ij}$$
We add the following code before $A_C$:

\[
\text{if } count_j = \frac{1}{2} \text{ then Wait}_{ji} \tag{5}
\]

The notations $\frac{1}{2} \alpha_k$ and $\frac{1}{2} \beta_k$ should be replaced by the appropriate constants calculated from $\sigma$. Since an action may participate in several occurrences on the same sequence, different code akin to (4) and (5) for multiple occurrences can be added. One can optimize the transformation by not counting (and not checking for the value of $count_i$ for) actions that appear only once in the given execution $\sigma$. Similarly, we do not need to count actions that require the same added transformation code in all their occurrences.

Another consideration is to identify when the execution is finished. For this purpose, we add to $Oc(p_k)$ the action $\alpha$ that appears last in the execution per each process $p_k$. Thus, we count the occurrences of $\alpha$ as well in $count_i$. Let $\frac{1}{2} \alpha_k$ be the value of $count_i$ for this last occurrence $\alpha_k$ of $\alpha$ in $\sigma$. We add the following code, after the code for $\alpha$:

\[
\text{if } count_i = \frac{1}{2} \alpha_k \text{ then halt } p_i \tag{6}
\]

Again, if the last action of the process is the only occurrence of $\alpha$, one does not need to count it. Note that if we do not halt the execution of the process $p_k$ here, we may encounter and perform a later action of $p_k$ that is not in $\sigma$ and is dependent of an action of another process that did not reach its last occurrence in $\sigma$. This may lead to a behavior quite different than the one we want to obtain.

Modeling the additional code resulted from the transformation as amalgamated into the atomic actions of the original code is unrealistic. However, we now reason why it is not important that the additional code is modeled as if appearing atomically with the existing code to which it is related. Note that the only additional dependent actions are of the form $Free_{ij}$ and $Wait_{ji}$. However, when such a pair is added, $Free_{ij}$ would be preceded by some action $\alpha_k$, and $Wait_{ji}$ is succeeded by an action $\beta_i$ such that $\alpha_k \rightarrow \sigma \beta_i$. The dependency between these actions ($Free_{ij}$ and $Wait_{ji}$) are the same as the existing ones ($\alpha_k$ and $\beta_i$). Moreover, it can be shown that there cannot be any occurrence of an action between $\alpha_k$ or $Free_{ij}$ (both in $A(p_k)$) that are dependent on actions from $p_j$. Similarly, all occurrences between the occurrence of $Wait_{ji}$ and $\beta_i$ (both from $p_i$) are independent of actions from $p_j$. Hence the concurrency structure of the program is maintained, even when we break the actions of the transformed program in a more realistic way.

The transformed Dekker algorithm, which allows checking the path $\sigma_2$ is shown in Figure 4. The added code appears in boldface letters.

A corresponding transformation can be presented for concurrent programs with message passing [12]. There, instead of adding semaphores, we choose some of the non-deterministic options for receiving a message, waiting for the ones that occur according to the selected execution.

5. Calculating The Probability of a Test Case

Adding synchronization in order to test executions in the presence of nondeterministic choice may change the timing of executing transitions. In particular, such changes may alter the way a real time system interacts with some devices, and hence destroy the abil-
Figure 4. The transformed Dekker algorithm

Figure 4. The transformed Dekker algorithm
Furthermore, we also have a set of occurrences \( B = \{ \beta_1, \ldots, \beta_n \} \) that become enabled during the path, but are not executed (either they become disabled, or they remain enabled until the end of the path).

We obtain the following constraints:

- For each occurrence \( \alpha[l[\alpha], u[\alpha]] \) that becomes enabled at \( s_i \) and executed at \( s_j \) we have

  \[ l[\alpha] \leq x_j - x_i \leq u[\alpha] \]

  (Recall that \( x_j \) is also denoted by \( x[\alpha] \).)

- For each occurrence \( \beta[l[\beta], u[\beta]] \) of a transition that was enabled at some state \( s_i \) and was disabled at \( s_j \) (or lasted beyond the last state in the path if \( j = n \)), let \( x[\beta] \) be the time that \( \beta \) could have been executed, unless it became disabled (or the path terminated). Then we have:

  \[ x_j < x[\beta] \]

  Otherwise, \( \beta \) would have been selected.

  \[ l[\beta] \leq x[\beta] - x_i \leq u[\beta] \]

Let \( C \) be the collection of constraints as defined above. It defines a convex polyhedra. Under uniform distribution, the density of an occurrence \( \gamma \in A \cup B \) is \( f(\gamma) = \frac{1}{u[\gamma] - l[\gamma]} \).

The probability of the path is therefore

\[
\int \ldots \int_C f(\alpha_1) f(\alpha_2) \ldots f(\alpha_n) f(\beta_1) f(\beta_2) \ldots f(\beta_m) \, d\beta_m \ldots d\beta_1 \, d\alpha_n \ldots d\alpha_1
\]

Acknowledgement

I would like to thank my coauthors in developing the techniques described in this survey: Elsa Gunter, Marcin Judzinski and Hongyang Qu.

References