Path Exploration Tool

Elsa L. Gunter and Doron Peled
Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974, USA

December 30, 1998

Abstract

While verification methods are becoming more frequently integrated into software development projects, software testing is still the main method used to search for programming errors. Software testing approaches focus on methods for covering different execution paths of a program, e.g., covering all the statements, or covering all the possible tests. Such coverage criteria are usually approximated using some add-hoc heuristics. We present a tool for testing execution paths in sequential and concurrent programs. The tool, *path exploration tool (*PET*)*, visualizes concurrent code as flow graphs, and allows the user to interactively select an (interleaved) execution path. It then calculates and displays the condition to execute such a path, and allows the user to easily modify the selection in order to cover additional related paths. We describe the design and architecture of this tool and suggest various extensions.

1 Introduction

Software testing techniques [4] are frequently used for debugging programs. Unlike software verification techniques, software testing is usually less systematic and exhaustive. However, it is applicable even in cases where verification fails due to memory and time limitations. Many testing techniques are based on criteria for covering execution paths. Conditions are sought for executing the code from some point A to some point B, and the code is walked through or simulated. Different coverage criteria are given as a heuristic measure for the quality of testing. One criterion, for example, advocates trying to cover all the executable statements. Other criteria suggest covering all the logical tests, or all the flow of control from any setting of a variable to any of its possible uses [9]. Statistics about the effectiveness of different coverage approaches used are kept.

W.R. Cleaveland (Ed.): TACAS/ETAPS'99, LNCS 1579, pp. 405-419, 1999.

In this paper, we present a new testing approach and a corresponding testing tool. The focus of the analysis is an execution path in a sequential code, or on interleaved execution paths consisting of sequences of transitions from different concurrent processes. The system facilitates selecting such paths and calculating the conditions under which they can be executed. It also assists in generating variants of this path, such as allowing different interleavings of the path transitions.

The code of the checked programs is compiled into a collection of interconnecting flow graphs. The system calculates the most general condition for executing the path and performs formula simplification. We present the tool's architecture and demonstrate its use.

The system's architecture includes:

- An SML program that takes processes, written using Pascal-like syntax, and produces their corresponding flow graphs.
- A DOT program that is used to help obtain an optimal display of the flow graphs, representing the different processes.
- A TCL/TK graphical interface that allows selecting and manipulating paths.
- An SML program that calculates path conditions and simplifies them.
- An HOL decision procedure that is used to further simplify the path conditions by applying a Presburger arithmetic decision procedure.

2 System Architecture

Research in formal methods focuses mainly on issues such as algorithms, logics, and proof systems. Such methods are often judged according to their expressiveness and complexity. However, experience shows that the main obstacles in practically applying such technology into practice are more mundane: it is often the case that new proof techniques or decision procedures are rejected because the potential users are reluctant to learn some new syntax, or perform the required modeling process.

One approach to avoiding the need for modeling systems starts at the notation side. It provides design tools that are based on a simple notation such as graphs, automata theory (e.g., [8]), or message sequence charts [1]. The system is then refined, starting with some simplistic basic design. Such tools usually provide several gadgets that allow the system designer to perform various automatic or human assisted checks. There is some support for checking or guaranteeing the correctness of some steps in the refinement of systems. Some design tools even support automatic code generation. This approach prevents the need for modeling, by starting the design with some abstract model, and

```
begin
  y1:=x;
  y2:=1;
  while y1<=100 or y2=/=1 do
    begin
      if y1<=100 then
        begin
          y1:=y1+11;
          y2:=y2+1
        end
      else
        begin
          y1:=y1-10;
          y2:=y2-1
        end
    end;
  z:=y1-10
end.
```

Figure 1: Floyd's 101 program

refining it into a full system. Using standard notation, such as message sequence charts [3], conforms with the usual start of the design. On the other hand, automatic code generation is still add hoc, and it is not expected that the code generated would be efficient or elegant (although it is, by definition, well documented).

Our approach for testing is quite the complement to the above. After the software (or parts of it) is designed and coded, one checks its behavior under various conditions, or in various environments. One of the motivations of the PET tool is to avoid the need for modeling, and allow the testing to be performed using a notation that is natural for the user. The tool automatically translates the code of the program to be checked into one of the earliest and most useful notations for software, namely that of flow graphs. The program is written as one or more processes in a syntax similar to Pascal. Figure 1 includes the code for Floyd's 101 program, as accepted by our tool. We use the combination $= / =$ as inequality.

The graphical interface includes a window for each process, displaying the original text, and a compatible window displaying the corresponding flow graph. The flow graph is a directed graph, with some edges carrying labels. Each node in a graph is one of the following: *begin, end, test, wait, assign.* The *begin* and *end* nodes appear as ovals, the *test* and *wait* nodes appear as diamonds, labeled by the test condition, and the *assign* nodes appear as boxes labeled by

the assignment. There is no out edge from an *end* node, two out edges from a *test* node, and one out edge from all other nodes. The two out edges from a *test* node are labeled, one by "yes" and one by "no". The flow graph that is generated for the program in Figure 1 appears in Figure 2.

The focus objects of the tool are the *execution paths*. Path information is displayed using two additional windows. One window displays the recently selected execution path, and the other displays the *most general condition* to execute the selected path. In order to maintain the connection between the code and the model (the flow graph), the different windows are context sensitive: pointing at a node (e.g., a test or an assignment box) in a flow graph window would highlight the corresponding code in the process source window.[1]. A selected path in the 101 program appears in Figure 3. Each transition appears within parentheses that correspond to its shape (and color) in the flow graph. Inside the parentheses there is a pair corresponding to the process name, and the number of the transition (as appears in the flow graph). If several processes are involved, transitions of different processes appear with different amount of indentation from the left margin. If the coursor points at a transition listed in this window, the corresponding item in the flow graph and the corresponding text will be highlighted.

2.1 Path Operations

Software testing is based on inspecting paths. Therefore, it is of great importance to allow convenient selection of execution paths. Different coverage techniques suggest criteria for the appropriate coverage of a program. Our tool leaves the choice of paths to the user. (A future version, where various path selection criteria will be used to automatically suggest the tested paths, is under construction.) Once the source code is compiled into a flow graph, or a collection of flow graphs, the user can choose the test path by clicking on the appropriate constructs on the flow graphs.

The selected path appears also in a separate window, where each line lists the selected node, the process and the shape (the lines are also indented according to the number of the process to which they belong). In order to make the connection between the code, the flow graph and the selected path clear, sensitive highlighting is used. For example, when the cursor points at some node in the path window, the corresponding node in the flow graph is highlighted, as is the corresponding text of the process.

Once a path is fixed, the condition to execute it is calculated. The tool allows altering the path by removing nodes from the end, in reverse order, or appending to it new nodes. This allows, for example, the selection of an alternative choice for a condition, after the nodes that were chosen past that condition are removed.

[1] Our choice was, in the case of a test, to highlight the entire minimal programming construct that is associated with it, such as an if-then-else statement or a while loop.

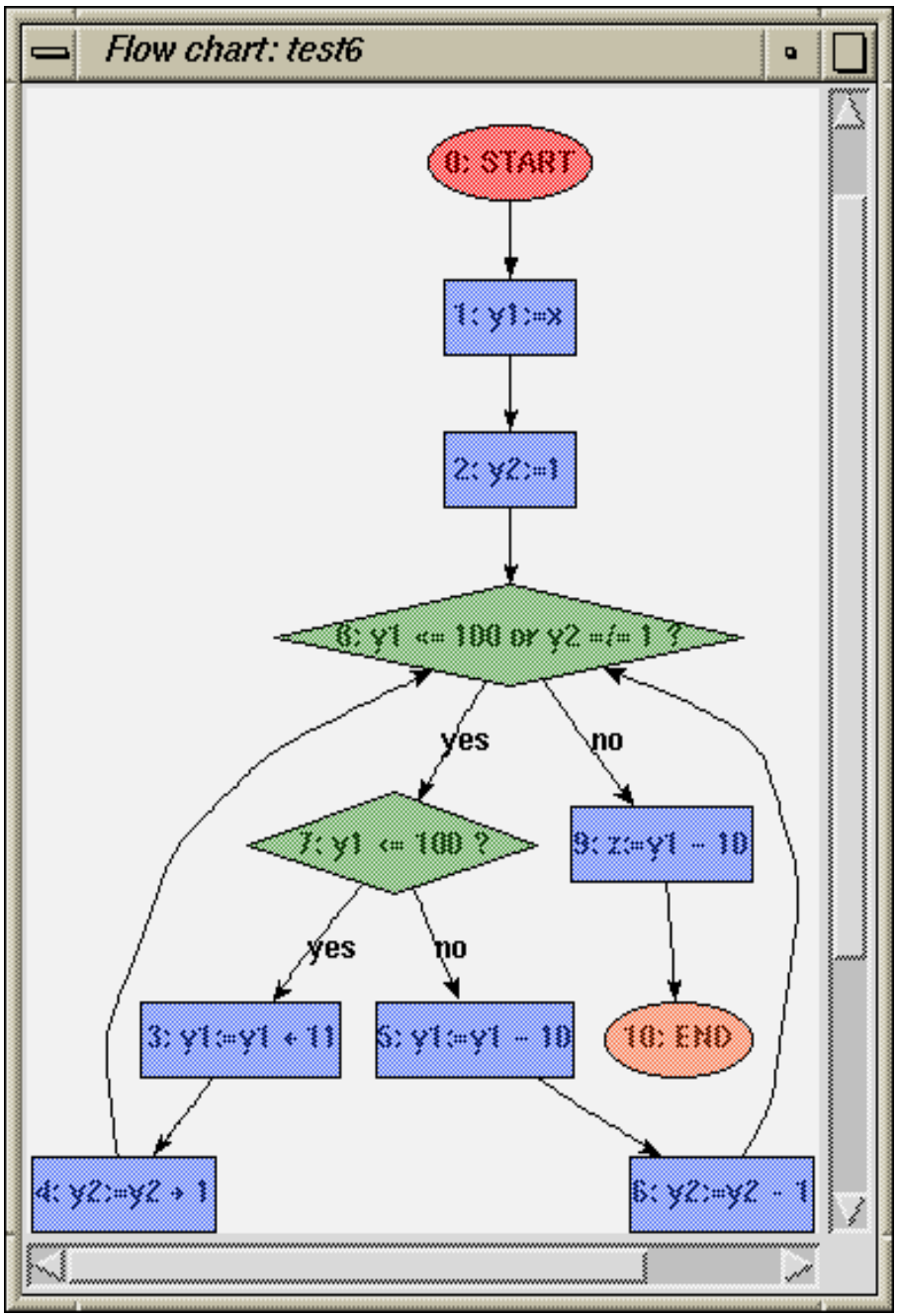

Figure 2: A flow graph for the program in Figure 1

```
(test6 : 0)
[test6 : 1]
[test6 : 2]
<test6 : 8>
<test6 : 7>
[test6 : 5]
[test6 : 6]
<test6 : 8>
[test6 : 9]
(test6 : 10)
```

Figure 3: A selected path in the 101 program

Another way to alter a path is to use the same transitions but allow a different interleaving of them. When dealing with concurrent programs, the way the execution of transitions from different nodes are interleaved is perhaps the most important source of problems. The PET tool allows the user to flip the order of adjacent transitions on the path, if they belong to different processes. It is easy to check that, by repeatedly flipping the order in this way, one can obtain any possible execution of the selected transitions.

2.2 Path Condition

The most important information that is provided by PET is the condition to execute a selected path. An important point to note is that an execution path in a set of flow graphs is really a sequences of edges, which when restricted to each of the processes involved, forms a contiguous sequence. However, when specifying an execution path, it seems most natural to give a selection of nodes to be executed. For most nodes, there is a one-to-one correspondence between the nodes in a flow graph and their out edges. The subtle case is when a test node is selected. Selecting such a node does not tell us how it executed, since the condition may be either true or false. The execution of a test is determined by whether its "yes" or "no" out edge was selected, which we can know by knowing the successor node to the test in the same process. Thus, if a test node is the last transition of some process in the selected path, it would *not* contribute to the path condition, as the information about how it is executed is not given.

Let $\xi = s_1 s_2 \ldots s_n$ be a sequence of nodes. For each node s_i on the path, we define:

$type(s_i)$ is the type of the transition in s_i. This can be one of the following: *begin, end, test, wait, assign.*

$proc(s_i)$ is the process to which s_i belongs.

$cond(s_i)$ is the condition on s_i, in case that s_i is either a *test* or a *wait* node.

$branch(s_i)$ is the label on a node s_i which is a *test* if it has a successor in the path that belongs to the same process, and is "undefined" otherwise.

$expr(s_i)$ is the expression assigned to some variable, in case that s_i is an *assign* statement.

$var(s_i)$ is the variable assigned, in case s_i is an *assign* statement.

$p[v/e]$ is the predicate p where all the (free) occurrences of the variable v are replaced by the expression e.

The following is the algorithm used to calculate the path condition. Notice that it is calculated from the tail of the path to the head.

```
current_pred := 'true';
for i := n to 1 step -1 do
    begin case type(s_i) do
        test⇒
            case branch(s_i) do
                'yes'⇒
                    current_pred := current_pred∧cond(s_i)
                'no'⇒
                    current_pred := current_pred ∧¬cond(s_i)
                'undefined'⇒
                    current_pred := current_pred
            end case
        wait⇒
            current_pred := current_pred∧cond(s_i)
        assign⇒
            current_pred := current_pred [ var(s_i)/expr(s_i) ]
      end case
      simplify(current_pred)
    end
```

It is interesting to note that the meaning of the calculated path condition is different for sequential and concurrent programs. In a sequential program, consisting of one process, the condition expresses all the possible assignments that would *ensure* executing the selected path, starting from the first selected node. When concurrency is allowed, the condition expresses the assignments that would make the execution of the selected path *possible*. Thus, when concurrency is present, the path condition does not guarantee that the selected path is executed, as there might be alternatives paths with the same variable assignments.

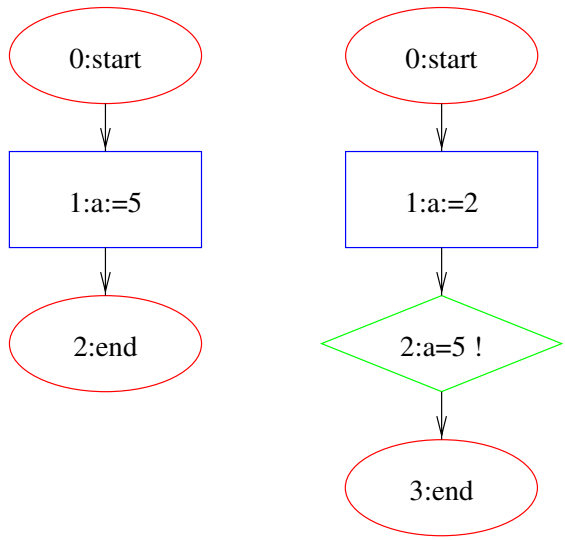

Figure 4: Two simple concurrent processes

In Figure 4, an example of two simple processes that share the variable a is given. The PASCAL code for the processes is as follows:

```
C1:                    C2:
 begin                  begin
  a:=5                   a:=2
 end                     wait a=5
                        end
```

Consider the following path:

```
(C1 : 0)
  (C2 : 0)
  [C2 : 1]
[C1 : 1]
  <C2 : 2>
  (C2 : 3)
(C1 : 2)
```

In this path, the '$a := 5$' of the first assignment is executed after the '$a := 2$' and hence the wait condition can be passed, and the path can be completed. This does not depend on the value of any variable. Thus, the path condition is 'true'. If we choose now to switch the third and the fourth lines, e.g., the two assignments to the variable a, the path cannot be passed, independent of any initial values of the variables. Thus, in this case the path condition is 'false'.

Switching the order between a pair of adjacent transitions is done by moving the mouse to the first transition in the pair and clicking a mouse button. The tool does not allow (the unreasonable choice of) permuting transitions that belong to the same process.

2.3 Formula Simplification

The primary information object that is provided by the PET tool is that of a quantifier free first order formula, describing the condition under which a path is executed. In the prototype developed, we assume that the mathematical model is that of arithmetic over the integers. As shown in the previous subsection, these conditions are calculated symbolically, and can therefore be quite complicated to understand. In most cases, the automatically generated expression is equivalent to a much simpler expression.

Simplifying expressions is a hard task. For one thing, it is not clear that there is a good measure in which one expression is simpler than the other. Another reason is that in general, deciding the satisfiability or the validity of first order formulas is undecidable. However, such limitations should not discard heuristic attempts to simplify formulas, and for some smaller classes of formulas such decision procedures do exist.

The approach for simplifying first order formulas is to try first applying several simple term-rewriting rules in order to perform some common-sense and general purpose simplifications. In addition, it is checked whether the formula is of the special form of *Presburger arithmetic*, i.e., allowing addition, multiplication by a constant, and comparison. If this is the case, one can use some decision procedures to simplify the formula.

The simplification that is performed includes the following rewriting:

- Boolean simplification, e.g., $\varphi \wedge \mathit{true}$ is converted into φ, and $\varphi \wedge \mathit{false}$ is converted into *false*.
- Eliminating constant comparison, e.g., replacing $1 > 2$ by *false*.
- Constant substitution. For example, in the formula $(x = 5) \wedge \varphi$, every (free) occurrence of x in φ is replaced by 5.
- Arithmetic cancellation. For example, the expression $(x + 2) - 3$ is simplified into $x - 1$, and $x * 0$ is replaced by 0. However, notice that $(x/2) * 2$ is not simplified, as integer division is not the inverse of integer multiplication.

In case the formula is in Presburger arithmetic, we can decide [7] if the formula φ is unsatisfiable, i.e., is constantly *false*, or if it is valid, i.e., constantly *true*. The first case is done by deciding on $\neg\exists x_1 \exists x_2 \ldots \exists x_n \, \varphi$, and the second case is done by deciding on $\forall x_1 \forall x_2 \ldots \forall x_n \varphi$, where $x_1 \ldots x_n$ are the variables that appear in φ. If the formula is not of Presburger arithmetic, one can still

try to decide whether each maximal Presburger subformula of it is equivalent to *true* or *false*.

Another way of using the decision procedure for Presburger arithmetic is to check whether there are variables that are not needed in the formula, and can hence be discarded. For example, consider a Presburger arithmetic formula $\varphi(x_1, x_2, \ldots x_n)$. We can check whether the formula depends on the variable x_n by checking

$$\forall x_1 \forall x_2 \ldots \forall x_{n-1} \forall x_n \forall {x_n}' \varphi(x_1, x_2, \ldots, x_n) \leftrightarrow \varphi(x_1, x_2, \ldots, {x_n}')$$

Then, if this formula is *true*, we can replace x_n by 0 everywhere.

2.4 Implementation

The PET system consists mainly of a graphical interface, and a program that is responsible for compilation and calculation, as described in Figure 5. The graphical interface is responsible for selection and update of execution paths. It was implemented in TCL/TK. Compilation and calculations are done via an SML program. The language SML was selected since it allows simple and efficient symbolic manipulations such as subformula substitution.

The SML program is running as a server process. It receives requests for processing from the graphical interface. One such request is of the form

file *processname*

and results in the compilation of the process to a flow graph. Another type of request is of the form

path *processname:node* ... *processname:node*

with a (reversed) selected path. The SML program calculates the weakest (most general) condition to execute the path, and returns it to the graphical interface for display.

The SML program informs the graphical interface when compilation is done, and also prepares several files, which the graphical interface uses. These files are:

- A DOT file, including the description of the flow graph that corresponds to the compiled process according to the DOT syntax (see Unix manual, or [6]. This allows using the DOT program in order to draw the graph.
- An adjacency list, specifying for each node of the graph its immediate successor. This information allows the graphical interface to control path selection.
- A list of pointers to the beginning and end of the text that corresponds to each graph item. This file allows connecting the flow graph with the text windows, so that the text corresponding to the currently selected node is highlighted.

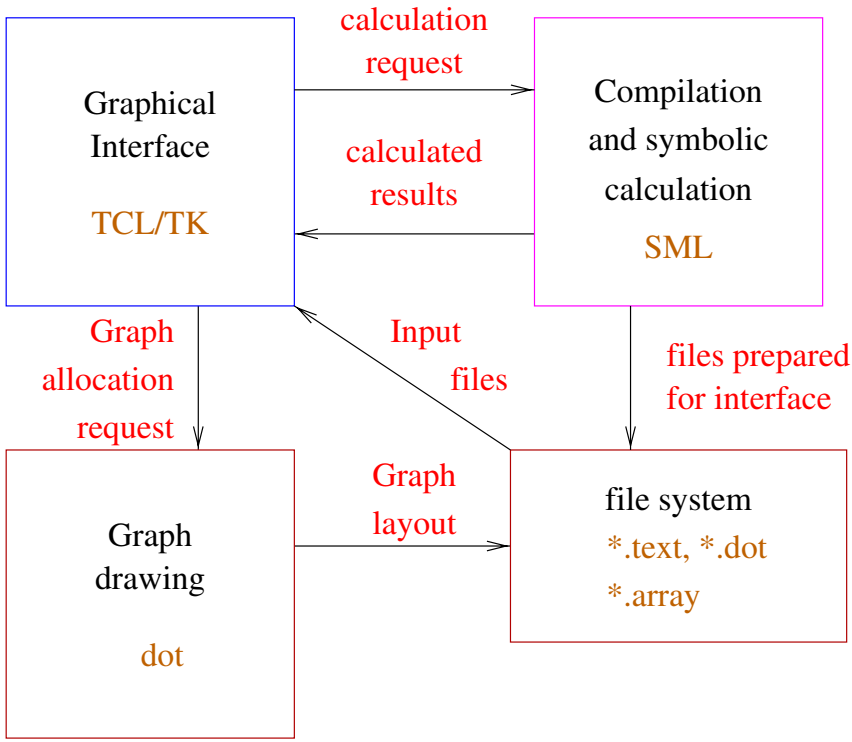

Figure 5: General architechture of PET system

The graphical interface makes use of the DOT program to draw flow graphs. The SML code prepares a DOT file, which describes the nodes, arrows and text of the flow graph. The DOT program processes this file and produces a layout for a visual description of the graph. It produces another DOT file, where the graph objects are annotated with specific coordinates. The TCL/TK graphical interface reads the latter file and uses it to draw the graph.

The SML program is compiled under the HOL environment. This allows using the Presburger Arithmetic decision procedure that is included in HOL to be used for simplifying arithmetic expressions by our system.

3 Examples

Consider the simple protocol in Figure 6, intended to obtain mutual exclusion. In this protocol, a process can enter the critical section if the value of a shared variable `turn` does not have the value of the other process. The code for the first process is as follows:

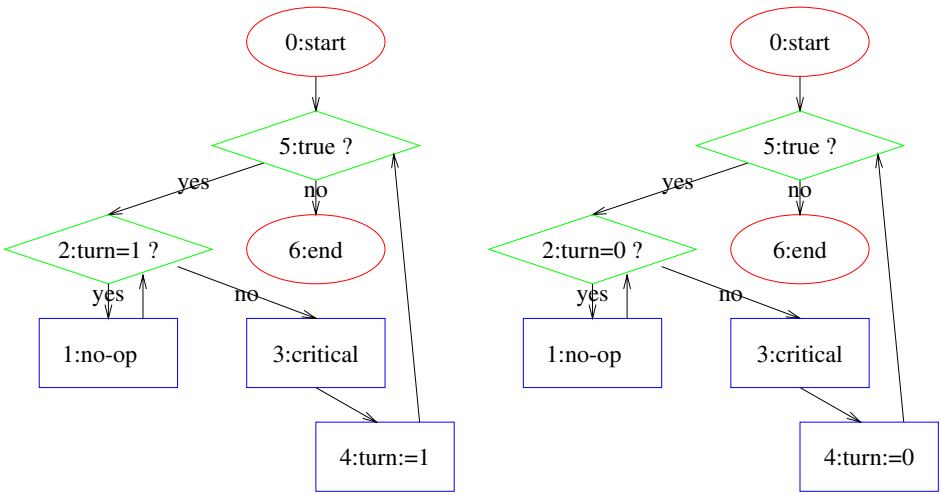

Figure 6: A mutual exclusion example

```
begin
while true do
  begin
    while turn=1 do begin (* no-op *) end;
    (* critical section *)
    turn:=1
  end
end.
```

The second process is similar, with constant values 1 changed to 0.

When we select the following path, which admits the second process *mutex1*, while the first process *mutex0* is busy waiting as follows:

```
(mutex0 : 0)
  (mutex1 : 0)
  <mutex1 : 5>
<mutex0 : 5>
  <mutex1 : 2>
<mutex0 : 2>
  [mutex1 : 3]
[mutex0 : 1]
```

we get the path condition $turn = 1$, namely that the second process will get first into its critical section if initially the value of the variable *turn* is 1. When we check a path that gets immediately into both critical sections, namely:

```
(mutex0 : 0)
```

```
  (mutex1 : 0)
  <mutex1 : 5>
<mutex0 : 5>
<mutex0 : 2>
  <mutex1 : 2>
[mutex0 : 3]
  [mutex1 : 3]
```

we get a path condition $turn \neq 1 \wedge turn \neq 0$. This condition suggests that we will not get a mutual exclusion if the initial value would be, say, 3. This indicates an error in the design of the protocol. The problem is that a process enters its critical section if *turn* is *not* set to the value of the other process. This can be fixed by allowing a process to enter the critical section if *turn* is set to its own value.

4 Extensions and Future Work

In this section, we describe work in progress, and planned extensions of the tool. The current implementation of the PET tool provides a basic framework for testing sequential and interleaved execution paths. The implementation was designed to support adding many testing techniques and features.

Software testing suggests various coverage criteria. For example, one might want to check paths that involve at least every executable statement, or paths from a statement where a variable is set to all or some of the statements where it is used [9]. Integrating such coverage techniques into our tool can be done by assisting the selection of a path according to such criteria. For example, when selecting an assignment node, the PET tool can suggest all the possible nodes where the variable that is assigned is later used. These nodes are highlighted using a color different from the other nodes, and the user can select one of them. The PET tool can then extend the current path with a shortest path from the current node to the node selected. Statistics about the quality of coverage can be collected.

Another extension deals with testing of different interleavings that are formed from a given set of transitions. Interleaving concurrent transitions in different orders is a main pitfall in concurrent programming. Currently, support is given to interleave transitions of different processes in various ways by commuting between them. An extension is being developed in order to facilitate a more efficient and thorough inspection of different interleaved sequences. The main idea is that many permutations of concurrently executed transitions do not lead to different results. For example, consider two transitions that involve completely different variables. Instead of only allowing the user to select particular adjacent transitions that will be commuted, the tool will successively generate different permutations of the selected interleaved sequences.

Using the compiled knowledge about the variables assigned and used by the transitions, a *dependency* relation between the transitions can be calculated [2]. The tool will calculate the next permutation that is *not* equivalent to the current one up to permuting adjacent independent transitions. (If it is equivalent, the path condition is guaranteed to be the same.) The new permutation will then be displayed and the new path condition will be calculated. Thus, the tool will help the user to cycle between different interleaved sequences that may give rise to different behaviors. (One can formalize this feature as presenting to the user different representatives for Mazurkiewicz traces [5].) This extension is also connected to the suggested path: if one uses the system's recommendation about how to continue a path from each given state, one does not have to worry about how to interleave these paths, as a systematic and exhaustive search of the interleavings can be performed. Of course, one has to be careful, as permuting interleaved sequences can lead to exponential number of possibilities.

Another direction of future development is to expand the programming language in which we require the programs to be written to include arrays and other data types, and to include subroutines. For arrays, the difficulty is calculating the precondition when array subscripts are given by complex arithmetic expressions.

We are also exploring different ways of presenting information to the user. Although the path conditions are in many cases simple to understand, there are cases where the user may find them difficult to use. Allowing the user to supply various finite ranges for the program variables enables the system to check whether there are values in the given range that satisfy the path conditions.

Finally, program slicing [10] can be used to extract projections of the program statements that affect a variable at a particular location. Such an analysis can also be calculated and displayed using our graphical interface.

References

[1] R. Alur, G. Holzmann, D. Peled, An Analyzer for Message Sequence Charts, *Software: Concepts and Tools*, 17 (1996), 70–77.

[2] S. Katz, D. Peled, Defining conditional independence using collapses, Theoretical Computer Science 101 (1992), 337-359.

[3] ITU-T Recommendation Z.120, Message Sequence Chart (MSC), March 1993.

[4] G.J. Myers, The Art of Software Testing, John Wiley and Sons, 1979.

[5] A. Mazurkiewicz, Trace Theory, *Advances in Petri Nets 1986*, Bad Honnef, Germany, LNCS 255, Springer, 1987, 279–324.

[6] E. Koutsofious, S.C. North, Drawing Graphs with dot, available on research.att.com in dist/drawdag/dotguide.ps.Z.

[7] D.C. Oppen, A $2^{2^{2^{pn}}}$ Upper Bound on the Complexity of Presburger Arithmetic, Journal of Computer and System Sciences 16, 1978, 323-332.

[8] B. Selic, G. Gullekson, P.T. Ward, Real-Time Object-Oriented Modeling, Wiley, 1993.

[9] S. Rapps, E.J. Weyuker, Selecting Software Test Data Using Data Flow Information, Transactions on Software Engineering 11(4): 367-375 (1985).

[10] M. Weiser, Program Slicing, IEEE Transactions on Software Engineering, 10(4), 1984, 352–357.

10.1007/b107031130028