Path Exploration Tool

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

August 26, 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 [3] techniques 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. Statistics are kept in the global level, about the effectiveness of different coverage approaches, and in the local level about the overall level of coverage achieved for the particular code tested.

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 an interleaved execution path, consisting of a sequence of transitions from different concurrent processes. The system facilitates selecting such a path, calculating the conditions under which it can be executed. It also assists in generating variants of this path, such as allowing different interleavings of the path events.

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

The system's architecture includes:

- An SML compiler that takes processes, written using Pascal-like syntax, and produces their corresponding flow-graph.
- A TCL/TK graphical interface that allow selecting and manipulating paths.
- An SML program that compiles code into a flow graph representation, calculates path conditions, and simplifies them.
- An HOL process that is used to further simplify the path conditions by applying a Presburger arithmetic decision procedure.
- The DOT program, which is used to help obtaining an optimal display of the flow-charts, representing the different processes.

2 Preliminaries

3 System Architecture

The PET system consists mainly of a graphical interface, and a program that is responsible for compilation and calculation, as described in Figure 1. 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 result in the compilation of the process. Another type of request is of the form *path process:node ... process: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 correspond to the compiled process according to the DOT syntax. 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.

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 use it to draw the graph.

3.1 Visualizing Concurrent Programs

Research in formal methods focus 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 attempting to transform such technology are more mundane. It is often the case that new proof techniques or decision procedures are rejected because the patential users are reluctant to learn some new syntax, or perform the required modeling process.

One approach to avoid the need for modeling systems starts at the notation side. It provides design tools that are based on simple notation such as graphs, automata theory (e.g., [4]), 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

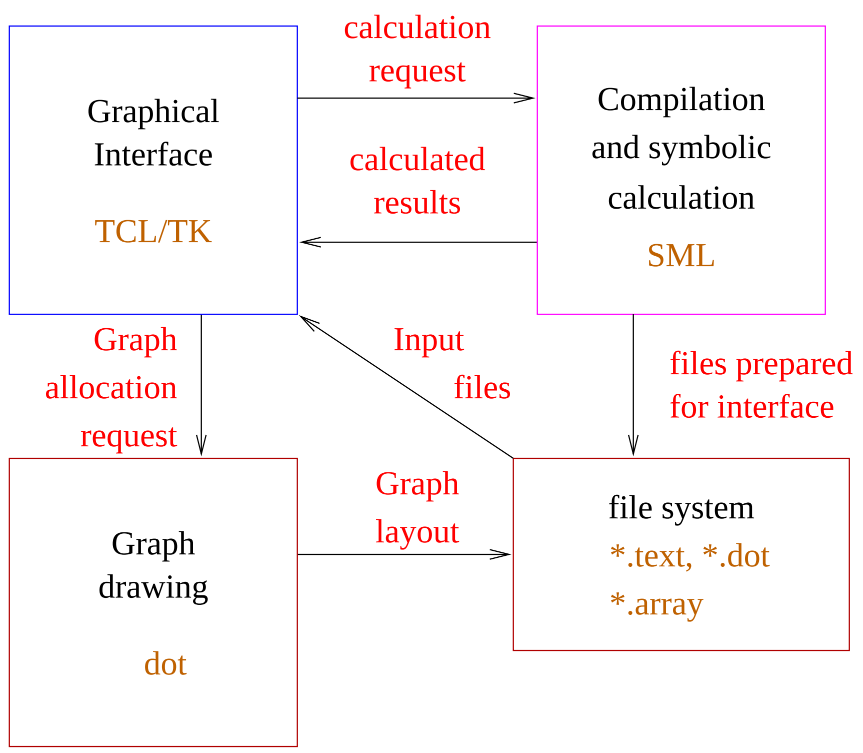

Figure 1: General architechture of PET system

modeling, by starting the design with some abstract model, and refining it into a full system. Using standard notation, such as message sequence charts [2], conforms with the usual start of the design. On the other hand, the 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) was 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 checked system into one of the earliest and most useful notations for software, namely that of flow graphs. The graphical interface includes a window for each process (or procedure), displaying the original text, and a compatible window that displays the corresponding flow graph.

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 to a node (e.g., a test or an assignment box) on a flow graph window would highlight the corresponding code in the process source window.[1]

3.2 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. Out 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 being prepared). Once the source code is compiled into a flow graph, or a collection of flow graphs, the user can choose the tested path by clicking on the appropriate constructs, namely, ovals (for *begin* and *end*), boxes (for *assignments*) and diamonds (for *conditions*).

The selected path appears also on 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 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 to some node on the path window, the corresponding shape in the flow graph is highlighted, and so does also the corresponding text of the process.

Once a path is fixed, the condition to execute it is calculated. The tool allows to alter the path by removing transitions from the end, in reverse order. This allows for example to select an alternative choice for a condition, after the nodes that were chosen past that condition are removed. Another way to alter a path, is to use the same transitions, but allow a different interleaving of the events. 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 events.

3.3 Path Condition

The most important information that is provided by PET is the condition to execute a selected path. A subtle point is understanding the execution order of the selected transitions. It is not only the appearance of the node in the path which affects the execution order, but also the appearance of a successor node from the same process. The reason is that by only looking at the appearence of

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

a node in the graph we may not be sufficient to determine *how* it is executed. Specifically, the execution of a test is determined also by whether its "yes" or the "no" exist edge was selected. 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 selected is not given. A similar case involve the *wait* statement, which has only one exit edge, followed when the condition holds. If a *wait* statement is the last statement of some process, there is no information whether the condition holds, or the process is waiting for the condition to be fulfilled.

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

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

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' transition.

pos(s_i) is a predicate that holds when node s_i is a test (i.e., the condition of an *if*, *while* or *wait*), and it has a successor in the path that belong to the same process. Furthermore, if s_i is not a *wait* statement, the edge from s_i to that successor is labeled by 'yes'.

expr(s_i) is the expression assigned to some variable, in case that s_i is an assignment transition.

var(s_i) is the variable assigned, in case s_i is an assignment 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 do
    if $command(s_i) \neq begin$ then
    begin case $type(s_i)$ do
         $test \Rightarrow$
            if $pos(s_i)$ then current_pred := current_pred$\wedge cond(s_i)$
                else current_pred := current_pred $\wedge\neg cond(s_i)$
         $wait \Rightarrow$
            if $pos(s_i)$ then current_pred := current_pred$\wedge cond(s_i)$
         $assignment \Rightarrow$
            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 *insure* 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.

In Figure 2, an example of two simple processes that share the variable a are given. The PASCAL code for the processes is as follows:

```
C1:begin
  a:=5
end.

C2:begin
  a:=2;
  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, independently of any initial values of the variables. Thus, in this case the path condition is 'false'. Notice that if the line

```
  (C2 : 3)
```

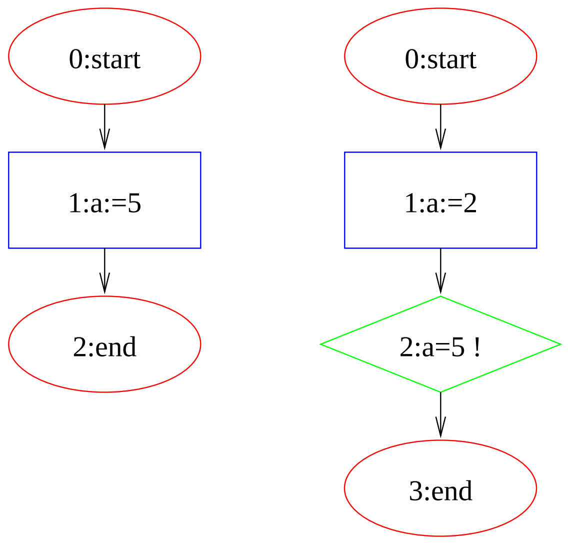

Figure 2: Two simple concurrent processes

was not present, we do not have the information about how the *wait* statement was resolved, and hence it is ignored (it is used only for the purpose of denoting that the control has passed in process $C2$ pass the '$a := 2$' assignment.

3.4 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 mathimatical 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 first try to apply 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, once can use some decision procedures to simplify the formula.

The simplification that is performed include 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 cancelation. 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 simplifed, as integer division is not the inverse of integer multiplication.

In case the formula is in Presburger arithmetic, we can decide whether 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.

4 Examples

Consider a simple mutual exclusion in Figure 3, where 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:

```
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 followin 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]
```

In this case we get a path condition $turn \neq 1$ and $turn \neq 0$. This condition suggests that we will not get a mutual exclusion if the initial value would be, say, 3.

References

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

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

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

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

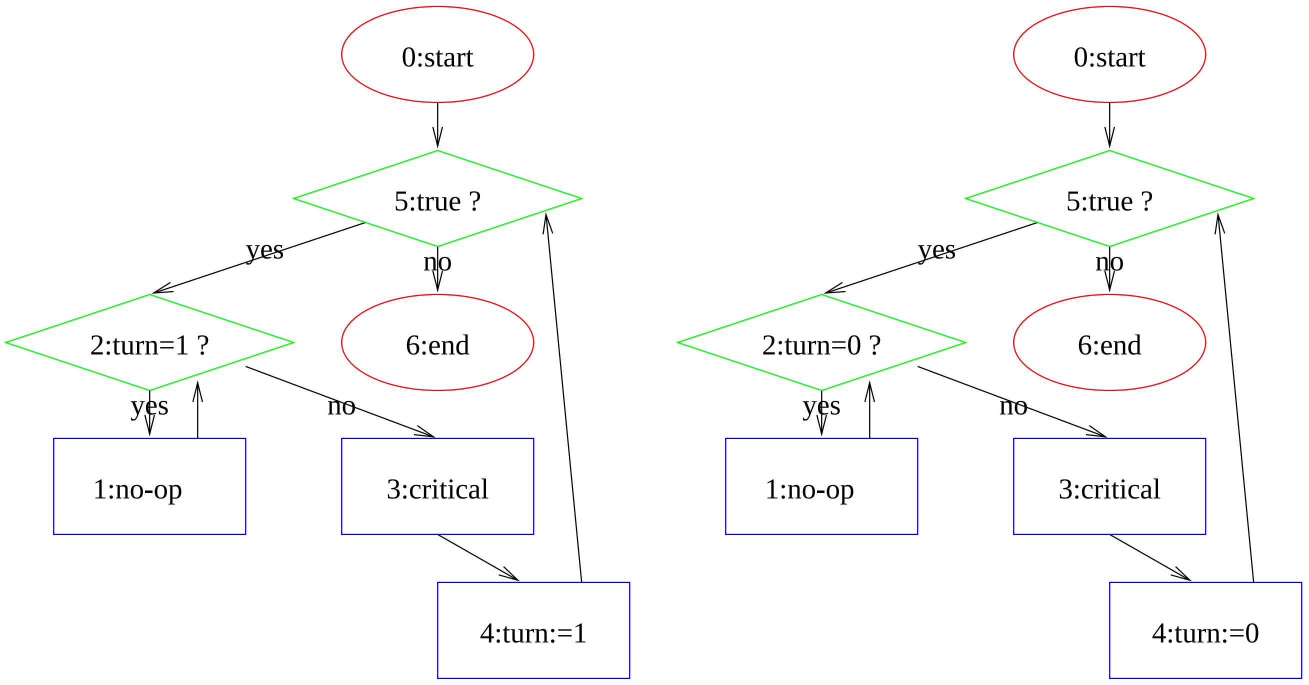

Figure 3: A Mutual Exclusion Example