PET: An Interactive Software Testing Tool

Elsa Gunter, Robert Kurshan, and Doron Peled

Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974

Abstract. We describe here the PET (standing for *path exploration tool*) system, developed in Bell Labs. This new tool allows an interactive testing of sequential or concurrent programs, using techniques taken from deductive program verification. It automatically generates and displays a graphical representation of the flow graph, and links the visual representation to the code. Testing is done by selecting execution paths, or, in the case of concurrent programs, interleaved sequences of code. The PET system calculates the exact condition to execute path being selected, in terms of the program variables. It also calculates (when possible) whether this condition is vacuous (never satisfied) or universal (always satisfied). The user can then edit the path and select variants of it by either extending it, truncating it, or switching the order of appearance of concurrent events. This testing approach is not limited to finite state systems, and hence can be used in cases where a completely automatic verification cannot be applied.

1 Introduction

Software testing is the most commonly used method for enhancing the quality of computer software. Testing is done usually in a rather informal way, such as by walking through the code or inspecting the various potential pitfalls of the program [1]. Methods that are more formal, such as model checking or deductive theorem proving, which have been used very successfully for hardware verification, have not succeeded to gain superiority in the area of software reliability. Deductive verification of actual software systems is very time consuming, while model checking suffers from the state space explosion problem, and is, in most cases, restricted to the handling finite state systems.

Concurrent programs may exhibit complicated interactions that can make debugging and testing them a difficult task. We describe here a tool that helps the user to test sequential or concurrent software using a graphical interface. It allows the user to walk through the code by selecting execution paths from the flow graph of the program. The most general relation between the program variables that is necessary in order to execute the selected path is calculated and reported back to the user. An attempt is made to decide using the path condition whether the path is at all executable. The user can edit the execution paths by adding, truncating and exchanging (in case of a concurrent program) the order of the transitions.

The PET tool is based on symbolic computation ideas taken from program verification. It allows more general ways of debugging programs than simulating

E.A. Emerson and A.P. Sistla (Eds.): CAV 2000, LNCS 1855, pp. 552–556, 2000.

one execution of the checked code at a time, as each path of the flow graph corresponds to all the executions that are consistent with it.

The input language to current implementation of PET is Pascal, extended with communication and synchronization constructs for concurrent programming. However, Pascal is just one possible choice. The main principles, on which this tool is based, can be used with other formalisms. In fact, it is quite easy to change the input language from Pascal to, e.g., C, SDL or VHDL.

Part of the problem in software testing and verification is coping with scalability. The PET tool also contains an abstraction algorithm, which can be applied during the compilation of the code. The algorithm attempts to abstract out certain variables and presents a *projection* of the program. That is, the obtained program is a simplified version in which some of the variables are abstracted away. This produces a simplified version of the program, allowing the user to better understand certain aspects of the code.

2 PET: Path Exploration Tool

A flow graph is a visual representation of a program. In the PET system [2], a node in a flow graph is one of the following: *begin, end, predicate, random, wait, assign, send* or *receive*. The *begin* and *end* nodes appear as ovals, the *predicate*, *wait* and *random* nodes appear as diamonds, labeled by a condition, or the word *random*, in the latter case. All other nodes appear as boxes labeled by the assignment, send or receive statement.

Each node, together with its output edge constitutes a *transition*, i.e., an atomic operation of the program, which can depend on some condition (e.g., the current program counter, an *if-then-else* or a while *condition* in the node, the nonemptiness of a communication queue) and make some changes to the program variables (including message queues and program counters). Notice that a *predicate* node corresponds to a pair of transitions: one with the predicate holding (corresponding to the 'yes' outedge), and one with the predicate not holding (corresponding to the 'no' outedge).

Unit testing [1] is based on examining paths. Different *coverage techniques* suggest criteria for the appropriate coverage of a program by different paths. Our tool leaves the choice of the paths to the user (a future version will allow a semi-automatic choice of the paths which uses various coverage algorithm in order to suggest the path selection, e.g., based on the coverage techniques in [1, 4]). The user can choose a path by clicking on the appropriate nodes on the flow graph.

In order to make the connection between the code, the flow chart and the selected path clearer, sensitive highlighting is used. For example, when the cursor points at some predicate node in the flow graph window, the corresponding text is highlighted in the process window. The code corresponding to a predicate node can be, e.g., an *if-then-else* or a *while* condition.

Once a path is fixed, the condition to execute it is calculated, based on repeated symbolic calculation of preconditions, as in program verification [3]. The condition is calculated backwards, starting with *true*. Thus, we proceed from a *postcondition* of a transition, in order to calculate its *precondition*. In order to

calculate the precondition given the transition and the postcondition, we apply various transformations to the current condition, until we arrive to the beginning of the paths. For a transition that consists of a predicate p with the 'yes' outedge, we transform the current condition from c to $c \wedge p$. The same predicate with the 'no' outedge, results in $c \wedge \neg p$. For an assignment of the form $x := e$, we replace in p every (free) occurrence of the variable x in the postcondition c by the expression e. We start the calculation with the postcondition *true* at the end of the selected path. Other kinds of transitions will be discussed later.

PET then allows altering the path by removing nodes from the end, in reversed order to their prior selection, or by appending new nodes. This allows, for example, the selection of an alternative choice for a condition (after the nodes that were chosen past that predicate nodes are removed). 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 foremost source of errors. The PET tool allows the user to flip the order of adjacent transitions on the path, when they belong to different processes.

The most important information that is provided by PET is the condition to execute a selected path. The meaning of the calculated path condition is different for sequential and concurrent or nondeterministic programs. In a sequential deterministic program, the condition expresses exactly the possible assignments that would *ensure* executing the selected path, starting from the first selected node. When concurrency or nondeterminism are allowed, because of possible alternative interleavings of the transitions or alternative nondeterministic choices, the condition expresses the assignments that would make the execution of the selected path *possible*. The path condition obtained in this process is simplified using rewriting rules, based on arithmetic. Subexpressions that contain only integer arithmetic without multiplication (Pressburger arithmetic) are further simplified using decision procedures (see [2]). In this case, we can also check algorithmically whether the path condition is equivalent to *false* (meaning that this path can never be executed), or to *true*.

In order to allow testing of communication protocols, one needs to add *send* and *receive* communication operations. Our choice is that of *asynchronous* communication, as its use seems to be more frequently used. The syntax of Pascal is then extended with two types of transitions:

`ch!exp` The calculated value of the expression `exp` is added to the end of the communication queue `ch`. (we will henceforce assume the queues are not limited to any particular capacity).

`ch?var` The first item of the communication queue `ch` is removed and assigned to the variable `var`. This transition cannot be executed if the queue `ch` is empty.

Typical communication protocols would allow a concurrent process to wait for the first communication arriving from one out of multiple available communications. The *random* construct represents nondeterministic choice and can be

used for that. An example for a choice of one out of three communications is as follows:

```
if random then
   if random then ch7?x
              else ch2?z
          else ch3?t
```

For the communication constraints, the translation algorithm scans the path in the forward direction. Whenever a *send* transition of the form `ch!exp` occurs, it introduces a new temporary variable, say `temp`, and replaces the transition by the assignment `temp:=exp`. It also adds `temp` to a queue, named as the communication channel `ch`. When a *receive* transition `ch?var` occurs, the oldest element `temp` in the queue `ch` is removed, and the *receive* transition is replaced by `var:=temp`. This translation produces a path that is equivalent to the original one in the case that all the queues were empty prior to the execution of the path. We can easily generalize this to allow the case where there are values already in the queues when the execution of the path begins.

Transition	queue	transformed	condition
P1:`x:=3`	ch=⟨⟩		*false*
P1:`ch1!x+y`	ch=⟨temp1⟩	`temp1:=x+y`	$x > 3$
P1:`ch1!x`	ch=⟨temp1, temp2⟩	`temp2:=x`	$x > 3$
P1:`x:=4`	ch=⟨temp1, temp2⟩		$temp2 > 3$
P2: `ch1?t`	ch=⟨temp2⟩	`t:=temp1`	$temp2 > 3$
P2: `ch1?z`	ch=⟨⟩	`z:=temp2`	$temp2 > 3$
P2: `z>3`	ch=⟨⟩		$z > 3$
P2: `z:=z+1`	ch=⟨⟩		*true*

Fig. 1. A path with its calculated condition

In Figure 1, the replacement is applied to a path with communication transitions. The first column describes the path. The second column denotes the (single in this example) queue used to facilitate the translation. The third column denotes the translated transitions (it is left empty in the cases where the translation maintain the original transition). The last column gives the calculated path condition (calculated backwards).

The user can select to project out a set of program variables. PET checks if there are assignments to other variables that use any of the projected variables. If there are, it reports to the user which additional variables need to be projected out. The projection algorithm removes the assignments to the projected variables and replaces predicates that use them by a nondeterministic choice.

References

[1] G.J. Myers, The Art of Software Testing, John Wiley and Sons, 1979.
[2] E.L. Gunter, D. Peled, Path Exploration Tool, to appear in Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Amsterdam, 1999.
[3] C.A.R. Hoare, An axiomatic basis for computer programming, Communication of the ACM 12, 1969, 576-580.
[4] S. Rapps, E.J. Weyuker, Selecting Software Test Data Using Data Flow Information, Transactions on Software Engineering 11(4): 367-375 (1985).