Testing by Narrowing

Extended Abstract

Sergio Antoy and Dick Hamlet

Portland State University
Department of Computer Science
Portland, OR 97207
{antoy,hamlet}@cs.pdx.edu

1 Introduction

Testing and debugging a program P may require computing an input I such that the execution of P on input I goes through some given path T of P. We describe how to compute such an input for programs coded in a simple imperative language with generic expressions including user-defined abstract data types.

For example, consider the following program which computes iteratively a preorder traversal of a tree. *Stack* and *tree* are user-defined types.

```
declare s : stack; t : tree;
begin
  if not(is_null(t)) then
    s := empty;
    push(t, s);
    while not(is_empty(s)) loop
      declare x, y : tree;
      begin
        y := top(s);
        pop(s);
        visit(y);
        x := left(y);
        if not(is_null(x))
          then push(x, s);
        end if;
        x := right(y);
        if not(is_null(x))
          then push(x, s);
        end if;
      end;
    end loop;
  end if;
end;
```

We may wish to compute an input that leads to the execution of two iterations through the body of the *while* statement such that during the first iteration the guard of the first *if* statement fails and the guard of the second *if* statement succeeds, whereas during the second iteration these conditions are reversed.

Problems of this kind are unsolvable in general. However, the technique that we describe is capable of finding any existing solution to the problem.

Our technique is a two-step procedure. First, given a program P, we compute the weakest precondition [4], say W, that guarantees the execution of a given path T of P. This computation is straightforward. Second, we attempt to solve the equation $W = true$ with respect to the variables of P. Operationally we use narrowing [5], a sound and complete procedure for solving equations involving symbols defined by a term rewriting system [6]. For our specific problem, if the narrowing procedure finds a solution I, then I is an input to P that executes T, and conversely, if there exists an input I to P that executes T, then the narrowing procedure finds I as a solution.

Next we describe in some detail the two steps of our technique, we outline a prototypical implementation, we discuss how more difficult problems can be solved, and finally we briefly relate our approach to similar ones previously proposed.

2 Weakest Precondition

Given a program P, a condition C, and a path T, we compute $W = wp(P, C, T)$, the weakest precondition such that the execution of P begun in any state satisfying W goes exactly through the statements of T and leaves the program in a state satisfying C. *wp* is a standard predicate transformer for a deterministic language [8], except for the presence of a third argument, the path T.

Such a path must be statically plausible, i.e., it could be executed if we were allowed to arbitrarily change both the state and the constants in the program before the execution of each statement. This

condition is easy to verify. The third argument of wp controls the choice of the next statement of a branching statement. A weakest precondition with respect to a fixed path is computed more easily than the general weakest precondition of a program.

Weakest preconditions may be huge expressions even for simple problems. The weakest precondition of the tree traversal problem discussed earlier is $and(not(is\_null(t)), \ldots)$, where there are a total of 75 occurrences of the program's operations and the variable t. In the following, we will denote this expression with $\mathcal{W}$.

3 Narrowing

The meanings of both the predefined operations of a language and the user-defined operations of a program are given by an equational specification. If $l = r$ is an equation, we stipulate that in an expression we can replace an instance of l with the corresponding instance of r, but not vice versa. For this reason we rather write our equation $l \rightarrow r$ and call it a *rewrite rule* [6].

For example, all the operation symbols occuring in $\mathcal{W}$, whose computation was discussed earlier, are specified as follows. The signature is obvious from the context. Capital letters stand for variables. The symbol '_' represents an anonymous variable.

$$\begin{aligned}
and(true, X) &\rightarrow X \\
and(false, _) &\rightarrow false \\
not(true) &\rightarrow false \\
not(false) &\rightarrow true \\
is_empty(empty) &\rightarrow true \\
is_empty(push(_, _)) &\rightarrow false \\
pop(push(_, S)) &\rightarrow S \\
top(push(E, _)) &\rightarrow E \\
is_null(null) &\rightarrow true \\
is_null(tree(_, _, _)) &\rightarrow false \\
left(tree(_, L, _)) &\rightarrow L \\
right(tree(_, _, R)) &\rightarrow R
\end{aligned}$$

A narrowing step of an expression such as $\mathcal{W}$ consists in computing a reduct of $\sigma(\mathcal{W})$, where σ is a substitution for the variables of $\mathcal{W}$ such that $\sigma(\mathcal{W})$ is reducible. For example, we may instantiate t to $tree(x, y, z)$, where x, y, and z are arbitrary values and reduce $is\_null(tree(x, y, z))$ to $false$. Thus, we solve an equation such as $\mathcal{W} = true$ by narrowing $\mathcal{W}$ all the way to $true$.

A solution for $\mathcal{W}$, computed by narrowing, is $t = tree(u, null, tree(v, tree(w, x, y), null))$, where u, v, w, x, and y are variables.

A brute-force implementation of narrowing is generally very inefficient. Relatively efficient implementations are based on strategies that limit the number of substitutions and positions that must be considered in a narrowing step. These strategies preserve the completeness of narrowing for the rewrite systems that we generally obtain in specifying the data types used in programming.

4 Implementation

Our prototypical implementation of our technique is coded in Prolog. The implementation comprises two major modules. One computes weakest preconditions and the other attempts to narrow them to $true$. The first module is small and conceptually simple, since the formal definition of our predicate transformer is easily mapped to Horn clauses. The translation of a program from its usual form to the form expected by the implementation of the predicate transformer is a non-trivial, language-dependent problem that we have not undertaken yet.

The second module is small too, but conceptually more complex. Several interesting implementation issues arise, in particular, the completeness of the narrowing procedure and the efficiency of the computation. We implement a lazy strategy that takes advantage of a particular representation of the rewrite rules and we couple our strategy with a breadth-first control regime.

Narrowing steps are "don't know" non-deterministic, whereas rewrite steps are, to a large extent, "don't care" non-deterministic. Thus, we gain efficiency by repeatedly reducing the needed redexes of an expression that is to be narrowed. Since some of these expressions may not have a normal form, some care must be taken to preserve the completeness of our implementation.

5 Advanced features

Statically plausible paths may be semantically impossible. For example, this is easy to see for the tree traversal program. Every time that *push* is invoked one iteration through the loop body must also be executed before the loop terminates. Thus, we may specify a path T that is not executed for any input. In this case, the resulting weakest precondition W has no solutions. If we try to solve $W = true$ by narrowing, the computation may or may not terminate. In practice, we fail to find inputs that reach the statements following a *while* statement.

To overcome this problem we weaken the constraints imposed on the path through a *while* statement. We specify how many iterations are executed through the body, but do not specify the path of each iteration. A weakest precondition for

such a less-constrained path is easily computed using power functions [1]. If $f : S \to S$ is a function on the state S of a program, the power function of f, $f^* : \mathbb{N} \times S \to S$, is defined by

$$f^*(k, s) = \begin{cases} s, & \text{if } k = 0; \\ f^*(k-1, f(s)), & \text{if } k > 0. \end{cases}$$

This approach allows us to find inputs to a program for reaching the statements that follow a loop.

For example, suppose that the problem is to find an input to the *while* statement of our tree traversal program that leads to the termination of the loop after exactly three iterations through the body. First, we compute the following weakest precondition, where *power* denotes the power function of the functional abstraction of the loop body.

$$\begin{array}{c} and(not(is\_empty(power(0, s)), \\ and(not(is\_empty(power(1, s)), \\ and(not(is\_empty(power(2, s)), \\ is\_empty(power(3, s)) \end{array}$$

Second, we solve this condition by narrowing, and we discover that there are only 10 distinct solutions: 5 in which s initially contains 1 tree only, 4 with 2 trees, and 1 with 3 trees.

For each solution we can find the path executed within the loop body by symbolic execution or by profiling an actual execution.

6 Related work

There are two research areas related to our work: program analysis and narrowing. The problem of finding inputs that will force the execution of a path in a program has been attacked in a number of ways. For example, [2] used symbolic execution and a linear-programming equation solver. Our approach differs from previous work in several significant ways:

1. Whereas it is usual to convert programs into control-flow graphs and compute paths in these graphs, we instead use a linear notation based on the program syntax itself.

2. Whereas it is usual to implement symbolic execution by following a flowgraph path from top to bottom, we instead use the weakest-precondition formalism that proceeds from bottom to top.

3. Whereas the usual method of solving equations is to employ some form of linear programming or matrix manipulation, we use narrowing.

The first two distinctions are matters of style only; nothing can be done using our approach that cannot be done in the previous way. Nevertheless, we believe that our approach is simpler. The final distinction is one of substance; narrowing can not only handle the numerical data types that previous solvers could, but can in principle deal with equations using arbitrary abstract types, and equations that arise from additional constraints. Thus we expect that our approach applies to a far wider class of programs, and to wider and more difficult problems.

Narrowing is the operational principle of languages that integrate the functional and logic paradigms [3]. Our application is unorthodox and the closest related work concerns the implementation of narrowing in Prolog, e.g., [7]. For our application the completeness of narrowing is crucial, thus, we have extended previous approaches by replacing the default depth-first search strategy with a breadth-first one and by interleaving narrowing with extensive rewriting.

References

[1] S. Antoy. Automatically provable specifications. Technical Report 1876, Dept. of Computer Science, University of Maryland, 1987.

[2] L. Clark. A system to generate test data and symbolically execute test programs. *Trans. on Soft. Eng.*, SE-2:215–222, 1976.

[3] D. DeGroot and G. Lindstrom, editors. *Logic Programming, Functions, Relations, and Equations*. Prentice Hall, 1986.

[4] E. W. Dijkstra. Guarded commands, nondeterminacy a formal derivation of programs. *Comm. of the ACM*, 18:453–457, 1975.

[5] M. J. Fay. First-order unification in an equational theory. In *Proc. 4th Workshop on Automated Deduction*, pages 161–167, Austin, TX, 1979. Academic Press.

[6] J. W. Klop. Term Rewriting Systems. In S. Abramsky et al., editor, *Handbook of Logic in Computer Science, Vol. II*, pages 1–112. Oxford University Press, 1992.

[7] R. Loogen, F. Lopez Fraguas, and M. Rodríguez Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. PLILP'93*. Springer LNCS, 1993. (To appear).

[8] R. T. Yeh. Verification of programs by predicate transformation. In R. T. Yeh, editor, *Current Trends in Programming Methodology*, volume 1, pages 228–247. Prentice-Hall, 1978.