Structural Testing with PathCrawler. Tutorial Paper

Nicky Williams Nikolai Kosmatov
CEA, LIST, Software Safety Laboratory, PC 174, 91191 Gif-sur-Yvette, France
E-mail: firstname.lastname@cea.fr

***Abstract*—Automatic testing tools allow huge savings but they do not exonerate the user from thinking carefully about what they want testing to achieve. To successfully use the PathCrawler-online structural testing tool, the user must provide not only the full source code, but also must set the test parameters and program the oracle. This demands a different "mindset" from that used for informal functional-style manual testing, as we explain with the help of several examples.**

I. The PathCrawler tool

PathCrawler [1] is an automatic tool for structural unit testing of C code developed at CEA LIST. The PathCrawler-online web service makes a restricted version of PathCrawler freely available for evaluation and teaching[1] The user uploads the C source code to be tested, sets the test parameters and programs an oracle and PathCrawler-online automatically constructs a set of test-cases which ensure complete coverage and displays the inputs, outputs, covered branches, path and verdict of each test-case, as well as the infeasible paths.

PathCrawler is based on the concolic or dynamic symbolic execution method and on constraint resolution. As constraint resolution (or satisfaction) is NP-complete, PathCrawler cannot guarantee to always cover a path, or demonstrate its infeasibility, within a reasonable time. When this occurs, PathCrawler reports that the corresponding path is probably infeasible but that this cannot be demonstrated. This problem is usually only posed by functions under test which implement numerical algorithms in which the branch conditions involve the results of complex calculations.

PathCrawler can be used to ensure, and demonstrate, code coverage when this is imposed by a standard. However, it can also be used even when code coverage is not imposed, as a convenient and rigorous way of debugging code fragments during development.

II. Automated testing in practice

PathCrawler-online automatically constructs the list of all possible effective input parameters for the uploaded function to be tested: these include declared parameters, global variables and the fields, elements and dimensions of all data-structures[2]. PathCrawler gives all these effective input parameters a default interval of possible values, which corresponds to the declared C type of the parameter. Variable array dimensions, including the "dimensions" of dereferenced input pointers, are given the default interval of 0..1. This means that by default, PathCrawler supposes that pointers may be `NULL` or may point to an array containing a single element. If the user does not change the test parameters then PathCrawler will construct tests which may include any combination of values from these default intervals.

The importance of test parameters. The user must survey the default test parameters and change them as necessary to ensure that the tests constructed by PathCrawler are pertinent. One possible result of inappropriate test parameters is that the generated tests reveal anomalies which the user treats as evidence of a bug in the implementation but which are actually due to an inappropriate calling context. An example of this is the function shown in Fig. 1(a), which should output an ordered array `t3` containing the 10 elements from the two input lists `t1` and `t2` of 5 elements each[3]. However, the algorithm used in this implementation of `Merge` is only correct if `t1` and `t2` are ordered. If `Merge` is tested with the default test parameters, it is probable[4] that PathCrawler will construct test-cases in which `t1` and `t2` are not ordered and so `t3` is not an ordered list. If the user supplies an oracle which checks whether `t3` is ordered then these test-cases will give a `failure` verdict and the user may waste time looking for a bug in the implementation of the algorithm.

The user may decide to include in the implementation of `Merge` a check that `t1` and `t2` are really ordered and return 1, for example, if they are not. This would be the case if lines 5-9 of the code shown in Fig. 1(a) were un-commented. This check would give rise to new branches in the code and PathCrawler will automatically generate new test-cases to activate the branches returning 1. This is an example of *robustness testing* which is naturally and automatically

[1] http://PathCrawler-online.com

[2] Pointers are not included directly, because their value is an address, but the variables to which they point are included.

[3] When input arrays have a declared dimension, as in this case, PathCrawler sets the default dimension to the declared dimension and not to 0..1, even though the C compiler ignores the declared dimension.

[4] To select input values to satisfy a certain set of constraints, and cover certain path, PathCrawler selects values randomly so the actual test-case inputs will vary each time PathCrawler is run, and the order in which the paths are covered may vary too. Below, we will refer to "probable" or "likely" properties of the generated test-cases because of this element of non-determinism.

a)

```
1  int Merge( int t1[5], int t2[5],
2             int t3[10]) {
3   int i = 0, j = 0, k = 0 ;
4  // test robustness: lines 5-9
5  /*for (i=0; i<4; i++)
6      if ( (t1[i+1] < t1[i])
7       || (t2[i+1] < t2[i]) )
8        return 1;
9    i = 0; */
10   while (i < 5 && j < 5) {
11     if (t1[i] < t2[j])
12       { t3[k] = t1[i]; i++; }
13     else
14       { t3[k] = t2[j]; j++; }
15     k++ ;
16     }
17   while (i < 5)
18     { t3[k] = t1[i]; i++; k++; }
19   while (j < 5)
20     { t3[k] = t2[j]; j++; k++; }
21   return 0 ;
22 }
```

b)

```
1  int Merge(int t1[],int t2[],int t3[],
2             int l1,int l2){
3    int i = 0, j = 0, k = 0;
4
5
6
7
8
9
10   while (i < l1 && j < l2) {
11     if (t1[i] < t2[j])
12       { t3[k] = t1[i] ; i++ ; }
13     else
14       { t3[k] = t2[j] ; j++ ; }
15     k++ ;
16     }
17   while (i < l1)
18     { t3[k] = t1[i] ; i++ ; k++ ; }
19   while (j < l2)
20     { t3[k] = t2[j] ; j++ ; k++ ; }
21   return 0;
22 }
```

c)

```
1  void Sort (int size, int * list)
2  {
3    int i, swap;
4    char done = 0;
5    int count = 0;
6    while ( !done && (count < size-1)){
7      done = 1;
8      for (i=0 ; i<size-1 ; i++)
9        if (list[i] > list[i+1]){
10         done = 0 ;
11         swap = list[i] ;
12 /* missing: list[i] = list[i + 1]; */
13         list[i + 1] = swap ;
14       }
15     count++ ;
16   }
17 }
```

Figure 1. **a–b)** Two versions of merge of two sorted arrays `t1`, `t2` into sorted array `t3`, and **c)** a buggy implementation of bubble sort.

included in the structural tests and so does not need to be carried out separately.

However, if the user knows that this implementation of `Merge` will only be called with ordered arrays, then he or she must add a precondition to the test parameters in order to prevent PATHCRAWLER constructing test-cases containing unordered arrays. Preconditions can either be entered directly, using the PathCrawler-online interface, as formulae over the input values in a subset of first-order logic, or by including in the source code files a C function with a particular name and signature which returns 1 if its inputs satisfy the precondition and 0 if not.

In the previous example, `Merge` was correctly implemented for ordered inputs and bad test parameters resulted in the user wasting time diagnosing what appeared to be a bug. Another danger of inappropriate test parameters is that, instead of resulting in incorrect outputs, test-cases provoke a runtime error before any outputs are generated. For an example of this, consider the implementation of `Merge` in Fig. 1(b), which accepts input arrays of any dimension. In this example, `l1` and `l2` represent the number of elements in `t1` and `t2`. With the default test parameters, PATHCRAWLER will construct test-cases in which `t1` and `t2` are either `NULL` or point to an array containing a single element and `l1` and `l2` take any integer value. It is likely that `l1` or `l2` will be greater than the dimension of `t1` or `t2`, provoking a segmentation fault when the test case is executed. To prevent such cases being generated, the user must add a precondition which ensures that `l1` is always less than or equal to the dimension of `t1`, `l2` less than or equal to the dimension of `t1` and `l1+l2` less than or equal to the dimension of `t3`.

In this same example, the user must enlarge the interval of dimension values for the arrays from its default value of 0..1 in order allow PATHCRAWLER to construct test-cases in which the input arrays have more than one element and `l1` and `l2` can be greater than 1. By setting the upper limits of the dimensions of `t1` and `t2`, the user influences how many inter-leavings of successive elements from `t1` and `t2` are tested and consequently the number of paths which are tested, because in this example the number of feasible paths depends on `l1` and `l2`.

Partial test coverage. Another way to limit the number of paths which are tested, when the tested function contains one or more loops whose number of iterations is input-dependent, is to set the coverage criterion to *all k-paths* instead of its default value of *all paths*. This test parameter allows the user to specify a number, k, of loop iterations so that PATHCRAWLER will just try to cover all feasible paths with up to k consecutive iterations of each loop[5]. This criterion can notably be used to limit the number of tests of functions containing loops with a number of iterations which depends on the inputs but not on the dimensions of any input array.

Defining the oracle. Once the user has decided on the test parameters, he or she must turn their attention to the oracle. PATHCRAWLER constructs a default oracle which just returns the verdict `unknown` for all tests. The user should modify the source code of this default oracle, using the macros provided to return `success` or `failure` verdicts. To decide the verdict, the oracle program must compare the values which are input to the function under test to those which the implementation returns as output[6]. The more rigorous the oracle, the more likely it is that the generated tests will detect a bug. For example, if an oracle for `Merge`

[5] In fact, PATHCRAWLER may happen to generate certain tests with more than k iterations of a loop, but given the already generated tests, it will only try to generate further tests with fewer than k iterations.

[6] The declared parameters of the oracle are the values of the declared parameters of the tested function before (renamed by prefixing their name with `pre__` and after it is called, as well as the value returned by the function, if it does return a value. The oracle program can also access the values of global variables before (renamed using the same convention) and after the execution of the function.

a)

```
int Bsearch( int A[10], int x )
{
  int low = 0 ;
  int high = 9 ;
  int found = 0, mid ;
  while( high > low ) {
    mid = (low + high) / 2 ;
    if( x == A[mid] )
      found = 1 ;
    if( x > A[mid] )
      low = mid + 1 ;
    else
      high = mid - 1 ;
    }
  mid = (low + high) / 2 ;
  if( ( found != 1)
   && ( x == A[mid]) )
    found = 1 ;
  return found ;
}
```

b)

```
int Bsearch( int A[10], int x)
{
  int low = 0 ;
  int high = 9 ;
  int found = 0, mid ;
  while( high > low ) {
    mid = (low + high) / 2 ;
    if( x == A[mid] )
      found = 1 ;
    if( x > A[mid] )
      low = mid + 1 ;
    else
      high = mid - 1 ;
    }
  mid = (low + high) / 2 ;
  if( ( found != 1)
   && ( x == A[mid]) )
    found = 0 ; // bug here
  return found ;
}
```

c)

```
int Bsearch( int A[10], int x )
{
  int low = 0 ;
  int high = 9 ;
  int found = 0, mid ;
  while( high > low ) {
    mid = (low + high) / 2 ;
    if( x == A[mid] )
      found = 1 ;
    if( x > A[mid] )
      low = mid + 1 ;
    else
      high = mid - 1 ;
    }
  mid = (low + high) / 2 ;
  if( ( found != 1)
   && ( x >= A[mid]) ) // bug here
    found = 1 ;
  return found ;
}
```

d)

```
int Bsearch( int A[10], int x)
{
  int low = 1 ; // bug here
  int high = 9 ;
  int found = 0, mid ;
  while( high > low ) {
    mid = (low + high) / 2 ;
    if( x == A[mid] )
      found = 1 ;
    if( x > A[mid] )
      low = mid + 1 ;
    else
      high = mid - 1 ;
    }
  mid = (low + high) / 2 ;
  if( ( found != 1)
   && ( x == A[mid]) )
    found = 1 ;
  return found ;
}
```

e)

```
int Spec(
  int Pre_A[10], int A[10],
  int Pre_x, int x,
  int found )
{
  int i, present = 0 ;
  for(i = 0 ; i < 10 ; i++)
      if(A[i] == x)
        present = 1 ;
  if(present != found)
    return 0 ;
  else
    return 1 ;
}

int Correct( int A[10], int x )
{
  int found = Bsearch(A,x) ;
  return Spec(A,A,x,x,found) ;
}
```

Figure 2. **a–d)** Four versions of dichotomic search of element `x` in sorted array `A`, and **e)** specification function.

just checks whether the output is sorted then bugs which prevent the output array containing all elements of the input arrays, or which cause the input arrays themselves to be modified, will not be detected.

Novice users often have difficulty understanding the relationship between the oracle, the implementation of the function to be tested and the precondition. For instance, novice users sometimes want to use the oracle to check that the precondition is respected by the test-case. This is unnecessary: the precondition is defined separately from the oracle specifically in order to ensure that PATHCRAWLER will only construct test-cases which satisfy the precondition. Some other novice users essentially copy the implementation being tested into the oracle but this just has the effect of comparing the implementation to itself, which will always give a `success` verdict whether the outputs are correct or not! Indeed, some functions have a natural “declarative” oracle which is completely different to the implementation, such as the oracle for a sorting algorithm which can check whether the output is sorted without having to calculate the “expected output”. In some cases, the oracle can be a less efficient implementation of the same function. For example, dichotomic search, which should be efficient for sorted arrays, can be checked by an oracle which implements the less efficient iterative search. In many real-life examples, the oracle can be a “golden reference” of the algorithm being implemented or a previous version of the implementation which is called “back to back” by the oracle. However, when none of these options are available, the user often just has to recode the tested function in a way as different as possible to the tested implementation or resign him- or herself to a partial oracle which may not detect all bugs. For relatively simple tested functions without too many paths, instead of using an oracle, the user can manually check the symbolic outputs and path predicate of each test-case. The *symbolic outputs* are the output values expressed as a formula over the input values and the user can check whether this formula corresponds to the expected computation. If the symbolic output is just a constant value, then the user can check the path predicate, which expresses the conditions on the input values which decided this particular output value.

Using PATHCRAWLER outputs for debugging. Indeed, PATHCRAWLER provides a wealth of information to help the user detect and localize any bugs. Where there is an oracle, the first indication is the test-case verdicts. However, the user should usually start by checking the input values of a few test-cases to see if they seem "reasonable": they can reveal a faulty precondition. Then the user should check the input and output values of the test-cases with a `failure` verdict in order to ensure that it is the implementation that is faulty and not the oracle! In some cases, the output values indicate the source of the bug, such as when the example of Fig. 1(c) (which contains a bug which systematically overwrites elements in the output array) is tested. In other cases, the paths covered by the failed cases will be the only ones to cover a particular sequential block of instructions and then the bug is likely to be situated in this block or in the condition leading to it. Consider the example of a buggy implementation of dichotomic search for a particular value in an ordered array shown in Fig. 2(b). It is the tests which cover paths including the true branch of the sub-condition in line 17 which give a `failure` verdict.

So far, we have considered bugs which produce incorrect outputs. Other potential types of bug are un-initialized local variables or runtime errors such as buffer overflow or `NULL` pointer de-referencing. These errors may always occur when a particular execution path is activated, or they may only occur when it is activated with certain input values. In the former case, or if PATHCRAWLER happens to have generated a test-case with input values which provoke the error, then PATHCRAWLER will either report to the user that the tested function crashed when executed or, if the error does not cause the program to crash, then PATHCRAWLER will detect it during the analysis of the path, after the test has been executed. However, if the test-cases generated by PATHCRAWLER do not reveal any errors of this type, it does not necessarily mean that they are not present and would not have been revealed by other input values.

Bypassing the limits of structural testing. PATHCRAWLER was designed primarily as a structural testing tool and so the test-cases which it generates suffer from the well-known limitations of structural testing. However, these limitations can be reduced to some extent by novel uses of structural testing.

Let us consider another example of a buggy implementation of dichotomic search, shown in Fig. 2(c), and first compare random testing, functional testing and structural testing. Random testing using the whole range of integer values would be very unlikely to generate test-cases in which `x` occurred in A so would be unlikely to detect many bugs. Functional testing would be likely to produce 10 test-cases in which `x` was present in A (one for each position in which it might be found), and 1 test-case in which it was not present. PATHCRAWLER creates more than 10 test-cases in which `x` is present in `A` but also several different examples in which the absent `x` is ordered compared to the elements of `A`. PATHCRAWLER can therefore detect the bug in the implementation shown in Fig. 2(c) which functional testing may not detect. However, PATHCRAWLER often fails to reveal the bug in Fig. 2(d) because it covers all paths but does not "know" that the search should include the first element in the array. This is an example of the *missing path* limitation of structural testing[7]. To combat it, the user must provide PATHCRAWLER with a specification of the expected functionality of Bsearch and check whether PATHCRAWLER can cover a path which conjoins a path in the implementation and a path which fails to satisfy the specification. This is achieved by adding the code shown in Fig. 2(e) to that of Fig. 2(d) and testing the function `Correct`. This function combines the implementation of Bsearch with a specification inspired by the oracle of the previous examples of Bsearch. The new oracle must test whether the result is 1 and the test-case which fails is the one in which `x` is equal to `A[0]`.

Similarly, PATHCRAWLER can be used to systematically find all runtime errors by inserting extra branches in the code at each point where such an error could occur. In trying to cover these branches, PATHCRAWLER will try to find test-cases which provoke runtime errors. Such branches could be inserted automatically, for example by using the RTE plugin of the FRAMA-C platform [8]. However, it is more efficient to run static analysis on the code first in order to detect all certain and possible run-time errors and then use PATHCRAWLER to either provide a test-case or demonstrate non-reachability of each one, such as in the SANTE prototype [2].

REFERENCES

[1] N. Williams, B. Marre, P. Mouy, and M. Roger, "PathCrawler: automatic generation of path tests by combining static and dynamic analysis," in *EDCC'05*.

[2] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand, "Program slicing enhances a verification technique combining static and dynamic analysis," in *SAC'12*.

[7] Indeed the only indication of this error is the number of feasible paths in the implementation. If the user's test process demands justification of the infeasibility of all uncovered paths, then some bugs may be uncovered by close examination of the feasible and infeasible paths.

[8] http://frama-c.com