A Lesson on Structural Testing with `PathCrawler-online.com`*

Nikolai Kosmatov[1], Nicky Williams[1], Bernard Botella[1],
Muriel Roger[1], and Omar Chebaro[2]

[1] CEA, LIST, Software Safety Laboratory, PC 174, 91191 Gif-sur-Yvette France
`firstname.lastname@cea.fr`
[2] ASCOLA (EMN-INRIA, LINA), École des Mines de Nantes, 44307 Nantes France
`firstname.lastname@mines-nantes.fr`

Abstract. PathCrawler is a test generation tool developed at CEA LIST for structural testing of C programs. The new version of PathCrawler is developed in an entirely novel form: that of a test-case generation web service which is freely accessible at `PathCrawler-online.com`. This service allows many test-case generation sessions to be run in parallel in a completely robust and secure way. This tool demo and teaching experience paper presents `PathCrawler-online.com` in the form of a lesson on structural software testing, showing its benefits, limitations and illustrating the usage of the tool on simple examples.

1 Introduction

Structural testing assures that the test set has thoroughly exercised the program with respect to a given coverage criterion. PathCrawler [1, 2] is a concolic test generation tool enumerating all program paths developed at CEA LIST for structural testing of C programs. This paper presents a new version of the tool developed in a novel form: that of a test-case generation web service freely accessible online [3]. This form is ideal for discovering the tool, its evaluation and teaching. We have used it in courses taught in several French universities for groups of 30 students working in parallel.

In our opinion, the benefits of automatic structural testing remain underestimated in the industry. To improve the situation, structural testing tools must be more widely taught at verification and validation courses during higher education. That was our motivation to write a teaching experience paper and to demonstrate the PathCrawler tool in the form of a small lessson where the students manipulate the tool and answer the questions. Sec. 2,3 present our experience feedback and the lesson. Sec. 4 provides some related work and concludes.

2 Teaching feedback and discussion

This lesson assumes the students have learned basic notions related to structural testing e.g. control-flow graphs (CFG), execution paths, branches and oracle. Our experience shows that theoretical courses are insufficient for learning software

* This work has been partially funded by several ANR projects.

a)

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

b)

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

Fig. 1. Two erroneous versions of binary search of element `x` in sorted array `A` of size `n`.

testing for the majority of students. The selected questions of this lesson correspond exactly to the difficult points that should be thoroughly exercised in practice. Testing with a wrong (incomplete or too strong) precondition, or without a precondition is a very common error that may be revealed by runtime-errors or wrong test results, but may remain completely unnoticed. Another common difficulty is to understand the role of an oracle. Many students do not see how to check the results of a function under test f without necessarily using the same algorithm again. Almost all students check the return values and forget to check that f does not modify variables when it does not supposed to do so. An incorrect oracle may work perfectly in an exercise and remain unnoticed, so the teacher should check the oracle of each student even if the final results seem correct. The three final questions help the students to acquire a deeper understanding of the subtleties of structural testing. Drawing the CFG helps the students to visualize and analyze test generation results, especially in the last question involving several functions.

3 The lesson

The C function `Bsearch` implements the well-known binary (or dichotomic) search algorithm. Given an ordered array of integers, `A`, an integer value to search for, `x`, and the number of elements in `A`, `n`, it should return 1 if `x` is an element of `A` and 0 if not. Let us investigate how PathCrawler can be used to test two different implementations of `Bsearch` of Fig. 1.

3.1 Testing without a precondition (test parameters)

Question 1. We start with the first implementation shown in Fig. 1a and behave as an inexperienced tester might, by just uploading this source code into `PathCrawler-online.com` via Test Your Code page and running test generation of function `Bsearch` with the default test parameters. Look at the Test Session Results and explain the errors.

Answer. The Test Session Summary shows that two test-cases provoked a runtime error. The Test-Case pages provide the input array sizes, input values and covered path of each test-case. In one of them, with `n=1873679323`, the segmentation

```
void oracle_Bsearch(int *Pre_A, int *A,
  int Pre_x, int x, int Pre_n, int n, int result_implementation){
  int i, present = 0;
  for(i=0;i<n;i++){
    if(A[i] != Pre_A[i])
      { pathcrawler_verdict_failure(); return; } /* A modified */
    if(Pre_A[i] == Pre_x)
      present = 1;
  }
  if(present==0 && present != result_implementation)
    { pathcrawler_verdict_failure(); return; } /* x wrongly found in A */
  else if(present==1 && present != result_implementation)
    { pathcrawler_verdict_failure(); return; } /* x wrongly not found in A */
  else {pathcrawler_verdict_success(); return; }
}
```

Fig. 2. Oracle for the functions of Fig. 1.

fault occurred because although the array was empty in this test-case, the loop body was executed (the path contains `+3`, i.e. the true branch at line 3 of the source code), including an out-of-bound array access at line 5. The implementation assumes that `n` is no greater than the array dimension, so this must be true for each test-case. The other test-case, with a negative value of `n`, provoked a similar out-of-bound array access at line 13.

3.2 Definition of a precondition

Question 2. Restart test generation of `Bsearch` again, but this time customize the test parameters so that the values of the elements of `A` and of `x` are restricted to the interval 0..10, the dimension of `A`, denoted by `dim(A)`, can be any value from 1 to 8 and add an unquantified precondition stating that `n` is equal to `dim(A)`. Look at the Test Session Results. Are there runtime errors? Complete the precondition if necessary. Explain what purpose a precondition serves in testing.

Answer. This time, the generated tests do not cause execution errors. However, we see that in these cases `A` is not always sorted so binary search does not necessarily work. We add to the precondition the requirement that `A` is sorted in the following quantified precondition:

$$\text{for all INDEX such that INDEX} < \text{n - 1}, \quad \text{A[INDEX]} <= \text{A[INDEX} + 1 \text{]}.$$

The preconditions ensure that the automatically generated test-cases will all respect the input domain of the implemented function. This avoids test failures due to test-cases which provoke execution errors or give the wrong result because, even if correctly implemented, the algorithm is not supposed to work on the inputs of such test-cases.

3.3 Role of an oracle in testing

Question 3. What purpose does the oracle fulfill in testing? What should a complete oracle for `Bsearch` check?

Answer. The oracle examines the inputs and outputs of each test-case and decides whether the implementation has given the expected outputs for the given inputs. A complete oracle for `Bsearch` should check that

– the array `A` is not changed by the implementation,
– the implementation returns the correct result, i.e. if `x` is really present in `A` then the implementation returns 1 and if not, it returns 0.

3.4 Using structural testing to detect a bug

Question 4. Go back to the test parameters used in Question 2 and change the default oracle, calling `pathcrawler_failure()` if the outputs are as expected and `pathcrawler_success()` if not. Rerun generation and check the verdicts and paths covered by the different test-cases. How do these help locate the bug in this implementation? Correct the bug and re-run generation with the same test parameters.

Answer. We replace the default oracle by the function of Fig. 2. The new test session results contain 8 test-cases with verdict `failure` and 15 with verdict `success`. Looking at the paths of the test cases, we see that the second condition on line 13 is only satisfied by the test-cases which failed (their path contains `+13, +13b`). This indicates that the bug is in the single statement (line 14) which is executed if this condition is satisfied. We replace it by `ret=1;` and re-run generation. All test-cases have now verdict `success`.

3.5 Limits of structural testing

Question 5. With the same test parameters as in Question 4, now generate tests for the second implementation of `Bsearch` in Fig. 1b. Are the same number of cases generated as in Question 4? What are the verdicts? The bug is in line 2. Try generation a few times to check whether the verdicts are always the same. Explain your results.

Answer. Fewer cases are generated this time and they almost always all have verdict `success` (although a run may occasionally happen to generate a test with a `failure` verdict). The bug in this implementation just causes the first element of the array not to be checked in some cases and this is why fewer tests are generated. This example shows the limits of classical structural testing. All paths in the code may be covered without revealing the error. This sort of error can only be found by taking the intended functionality of the implementation into account when generating the tests.

3.6 Testing with a specification

Question 6. Create a file with the function `Bsearch` of Fig. 1b and the functions shown in Fig. 3. Upload this new file into `PathCrawler-online.com` and generate tests for the function `CompareBsearchSpec` using the oracle of Fig. 4 and, otherwise, the same test parameters as in Question 5. Explain the significance of the test-case with a `failure` verdict (usually obtained before test generation is interrupted because of the limit on the number of partial paths in this evaluation version).

Answer. The function `spec_Bsearch` provides a specification similar to the oracle of Fig. 2, while `CompareBsearchSpec` calls `Bsearch` and `spec_Bsearch` to compare the result with the specification. There are therefore execution paths in `CompareBearchSpec` in

```
/* copy the function Bsearch above */
int spec_Bsearch(int *Pre_A, int *A,
  int Pre_x, int x, int Pre_n, int n, int result_implementation){
  int i, present = 0;
  for(i=0;i<n;i++){
    if(A[i] != Pre_A[i])
      return 0; /* A modified */
    if(Pre_A[i] == Pre_x)
      present = 1;
  }
  if(present==0 && present != result_implementation)
    return 0; /* x wrongly found in A */
  else if(present==1 && present != result_implementation)
    return 0; /* x wrongly not found in A */
  else return 1;
}

int CompareBsearchSpec(int *A, int x, int n){
  int *Pre_A = (int *)malloc(n * sizeof(int));
  int i;
  for (i = 0; i < n; i++)
    Pre_A[i] = A[i];
  int ret=Bsearch(A,x,n);
  return spec_Bsearch(Pre_A, A, x, x, n, n, ret);
}
```

Fig. 3. Specification for `Bsearch` and function `CompareBsearchSpec` to be tested in Qu. 6

```
void oracle_CompareBsearchSpec(int *Pre_A, int *A,
  int Pre_x, int x, int Pre_n, int n, int result_compare){
  if (result_compare)
    { pathcrawler_verdict_success(); return; }
  else
    { pathcrawler_verdict_failure(); return; }
}
```

Fig. 4. Oracle for function `CompareBsearchSpec` of Fig. 3

which the result returned by the implementation is not accepted by `spec_Bsearch`. In trying to generate tests to cover these paths, PathCrawler is searching for inputs which cause the implementation of `Bsearch` to give an unexpected result. In the test-case with the failure verdict, `x` is the first element of `A` and this case reveals the bug that was not detected by structural testing of the implementation of `Bsearch` alone.

4 Related work and conclusion

The PathCrawler method is related to other test generation tools combining symbolic and concrete execution of the program under test, for example, DART [4], CUTE [5], EXE [6], PEX [7], YOGI [8]. Although the idea to make tools available online for evaluation and use is very attractive, most tools are available only for download and require installation on the user's platform.

In the domain of software verification, few research teams provide an online (evaluation) version for their tool to allow (potential) users to quickly run it and to familiarize themselves with its concepts. The AgitarOne tool [9] (for unit testing of Java) and Euclide [10] (for property verification or proving reachability in C code) did have online versions allowing to try the tools. PEX for Fun [11] gives access to a limited version of the PEX [7] test generation tool in a recreational way, inviting the user to try to solve little puzzles. The online version of the Interproc static analyzer [12] illustrates static analysis for programs in a small imperative programming language accepted by the tool. In constraint programming, WebCHR [13] provides a service for solving CHR (Constraint Handling Rules) constraints online. No other software verification tool provides a testing web service for C software similar to `PathCrawler-online.com`.

Automatic structural test generators may also be used for other purposes, for example, to find execution errors [4–6], to verify conformity to specifications [7, 8, 14] or to verify non-functional properties [15]. The PathCrawler method can be efficiently combined with static analysis techniques, for example, for program debugging [16, 17]. We are currently studying uses of PathCrawler which go beyond traditional structural test-case generation, as illustrated by Question 6 above, and its novel combinations with static analysis and proof tools.

In this paper, we demonstrated `PathCrawler-online.com`, the new online version of the PathCrawler test generation tool, and illustrated by a small practical session how it can be used for teaching. We hope that this work will be helpful in teaching software testing at university level and will contribute to the introduction of automatic structural testing techniques in industry.

References

1. Williams, N., Marre, B., Mouy, P., Roger, M.: PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In: EDCC'05
2. Botella, B., Delahaye, M., Hong-Tuan-Ha, S., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: Experience with PathCrawler. In: AST'09
3. Kosmatov, N.: PathCrawler online (2010–2012) http://pathcrawler-online.com/.
4. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: PLDI'05
5. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ESEC/FSE'05
6. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: CCS'06
7. Tillmann, N., de Halleux, J.: White box test generation for .NET. In: TAP'08
8. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J.: Proofs from tests. In: ISSTA'08
9. AgitarOne Test Generator: (2012) http://www.agitar.com/.
10. Gotlieb, A.: Euclide: a constraint-based testing platform for critical C programs. In: ICST'09. http://euclide.gforge.inria.fr/.
11. Pex for fun: Online evaluation version of PEX (2011) http://pexforfun.com/.
12. Interproc online: (2012) http://pop-art.inrialpes.fr/interproc/interprocweb.cgi.
13. WebCHR online: (2012) http://dtai.cs.kuleuven.be/CHR/webchr.shtml.
14. Collavizza, H., Rueher, M.: Exploration of the capabilities of constraint programming for software verification. In: TACAS'06
15. Williams, N.: WCET measurement using modified path testing. In: WCET'05
16. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: The SANTE tool: Value analysis, program slicing and test generation for C program debugging. In: TAP'11
17. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: SAC'12