Covering User-Defined Data-flow Test Requirements Using Symbolic Execution

Marcelo Medeiros Eler[1], **André Takeshi Endo**[2], **Vinícius Durelli**[3]

[1] Escola de Artes, Ciências e Humanidades - Universidade de São Paulo
São Paulo - SP

[2]Universidade Técnica Federal do Paraná
Cornélio Procópio - PR

[3]Instituto de Ciências Matemáticas e de Computação - Universidade de São Paulo
São Carlos - SP

marceloeler@usp.br, andreendo@utfpr.edu.br, durelli@icmc.usp.br

Abstract. *Symbolic execution has been used in software testing as a effective technique to automatically generate test data. Most of approaches are based only on control-flow criteria and generate input data only for a whole program or function. However, testers may want to generate test data for covering data-flow criteria and also for satisfying specific test requirements. This paper presents an approach for generating test data to cover only test requirements selected by users, considering both control- and data-flow criteria. A prototype was implemented to support test data generation for Java programs and to perform a preliminary evaluation of the approach. The results, although in a limited context, are encouraging and motivate future experiments.*

Resumo. *A execução simbólica tem sido utilizada no teste de software como uma técnica efetiva para gerar dados de teste automaticamente. A maioria das abordagens considera apenas critérios de fluxo de controle e o teste completo da função ou do programa. Entretanto, testadores podem querer usar critérios de fluxo de dados e cobrir apenas requisitos específicos. Este artigo apresenta uma abordagem para gerar dados de teste para cobrir requisitos de teste definidos pelo usuário considerando critérios de fluxo de dados e de controle. Um protótipo foi implementado para gerar dados de teste para programas Java e realizar uma avaliação preliminar da abordagem. Os resultados, embora em um contexto limitado, são encorajadores e motivam mais experimentos.*

1. Introduction

Software testing is one of the key activities for software quality assurance. Its main goal is to execute a program with the goal of uncovering faults [Myers et al. 2004]. Selecting input values to create good test cases that are likely to reveal faults is a fundamental challenge in this field. However, testing a software with all possible input values is, in general, impossible [Vergilio et al. 2006].

Different testing techniques have been proposed to select finite but suitable sets of input values (test data) for test cases. Each technique contains a set of testing criteria which are adopted to establish test requirements that should be met by test cases. The most known testing techniques are classified as functional (black-box) or structural (white-box). Structural techniques focus on testing the inner structure of a program. Thus, testers derive test requirements from the source code of the program under test. These requirements define a testing criterion that can be used to determine whether the program under test should be more tested or not [Beizer 1990, Myers et al. 2004]. Usually, structural testing criteria are classified as control- and data-flow criteria [Rapps and Weyuker 1985, Zhu et al. 1997, Myers et al. 2004].

Generating test cases to satisfy test requirements established by structural criteria is a costly and error-prone activity. Symbolic execution and constraint solving have been used for more than three decades as an effective technique for automatic generation of test data that achieves high coverage of control-flow criteria requirements [Ramamoorthy et al. 1976, Cadar and Sen 2013]. The key idea behind symbolic execution is to represent the values of variables as functions of symbolic input values [King 1976, Cadar and Sen 2013]. Considering the control-flow criteria, for example, an execution path can be represented as a sequence of constraints that should be satisfied. Thus, a constraint solver is used to generate concrete input values that satisfy the constraint sequence for that path. If the program is executed on these concrete input values, it will take the same path and terminate in the same way.

Programs with a large number of execution paths can lead to many constraint sequences. A sizable amount of constraint sequences can negatively impact the constraint solving process, which is one of the main bottlenecks in symbolic execution [Cadar and Sen 2013]. Most heuristics focus on achieving high statement and branch coverage guiding the path exploration toward the path closest from an uncovered path [Cadar and Sen 2013]. Techniques to prune away irrelevant constraint sequences and cache constraint solving results have also been employed. Although research in symbolic execution has come a long way over the last three decades, there is much room for improvement in this area [Cadar and Sen 2013, Galler and Aichernig 2013].

A limitation of current approaches is that they only generate test data for a whole program or function. Current approaches do not generate test data to cover only a single slice of the code. We argue that generating test data to exercise only specific parts of the code would be helpful in many contexts, such as: *(i)* testers who have a test suite and want to generate test data only to meet uncovered test requirements for a given criterion; *(ii)* testers who want to generate test data only to exercise the integration between an application and specific parts of the code that interface with external components, services or systems; *(iii)* the generation of test data for regression testing only to exercise statements, branches, and data usage affected by a change; and *(iv)* approaches that generate test cases considering each user requirement (or group of requirements). Another limitation of current approaches is that they generate test data based only on control-flow criteria.

As programs become larger and more complex, covering all test requirements of these programs becomes unpractical, and hence selective testing becomes increasingly relevant. This paper proposes an approach for generating test data to cover only test requirements selected by users, considering both control and data-flow criteria. Given that this scenario is very likely in practice, the contribution of this paper is twofold. First, the tester will benefit from automatic test generation to cover complex test requirements and yet reduce the overhead of path explosion, since the test data generated are meant to cover only a subset of test requirements. In addition, another contribution is to introduce data-flow in a context where only control-flow criteria have been investigated.

This paper is organized as follows. Section 2 describes the basic concepts of structural testing and symbolic execution and discusses related work. Section 3 introduces our approach to generate test data for user-defined test requirements. We emphasize test requirements from the data-flow criterion all-uses. Section 3.1 details our prototype implementation. Section 4 provides an initial evaluation we conducted. Section 6 suggests future work and makes concluding remarks.

2. Background

The basic concepts behind structural testing and symbolic execution are presented by means of an illustrative example. Listing 1 shows a code snippet of a method that calculates the prime factorization of a number. This method receives an integer `N` as input and prints all prime factors of `N`. It starts trying to divide `N` by the first prime number `(2)` until the remaining of the division is different from `0`. Then, it calculates the next prime number and restarts the divisions until the prime calculated is greater than the half of `N`. For instance, given `N=20`, the output produced is `2 2 5`.

2.1. Structural testing

Testing the `factorization` method shown in Listing 1 according to the structural testing technique requires creating test cases that satisfy both control- and data-flow criteria. All-nodes, all-edges and all-paths are well known control-flow criteria because they consider the execution control of the program to generate test requirements [Zhu et al. 1997]. It is common to adopt an abstraction called *control-flow graph* (CFG) to represent the inner structure of the program under test and to support the analysis of structural testing criteria. CFGs are directed graphs in which each node represents a block of instructions without flow deviation (i.e., a basic block). Directed edges represent transitions in the control flow.

The all-nodes criterion requires that every node of the CFG be executed at least once, while the all-edges criterion requires the execution of every edge at least once. The all-paths criterion requires that every path of the CFG be executed once. If the CFG contains loops, it is usually impossible to execute all paths because in many cases this leads to an infinite number of paths. In such case, testers can put a limit to the number of loop iterations to be executed.

The all-uses criterion is a well-known data-flow criterion that takes information about the program data-flow to generate test requirements [Rapps and Weyuker 1982]. In this context, Rapps and Weyuker [Rapps and Weyuker 1985] proposed an extension to the CFG called *def-use graph* (DUG) to add information related to variable usage. The usage of a variable has been classified as computation use (c–use), when the variable is used in a computation, and as predicate use (p–use), when the variable is used in a conditional statement. In the DUG, c–uses are associated to nodes and p–uses are associated to edges. Figure 1 shows the DUG representation of the `factorization` method. The comments within Listing 1 indicate the DUG node number for each line of the method. The DUG signs a variable definition when its name is written inside `def=`{} next to a node. Variable uses are identified when their names are inside `c-use=`{} and `p-use=`{} next to a node or an edge, respectively.

Listing 1. A simple factorization method.

```
public void factorization(int N) {
  if (N > 0) { //1
    int prime = 2; //2
    int number = N; //2
    while (prime <= N / 2) { //3
      if (number % prime == 0) { //4
        System.out.println(prime); //5
        number = number / prime; //5
      }
      else
      {
        int nextPrime = prime; //6
        int found = 0; //6
        while (found == 0) { //7
          nextPrime = nextPrime + 1; //8
          found = 1; //8
          int d = 2; //8
          while (d <= nextPrime / 2) { //9
            if (nextPrime % d == 0) //10
              found = 0; //11
            d++; //12
          }
        }
        prime = nextPrime; //13
      }
    }
    if (number > 1) //14
      System.out.println(number); //15
  }
} //16 - return
```

Based on this data flow model, a family of testing criteria has been proposed, such as all-c-uses, all-p-uses, all-definitions, all-uses, and all-du-paths [Rapps and Weyuker 1982, Rapps and Weyuker 1985]. The all-uses criterion is by far the most known of them; it requires that every definition of a data object (variable) and its associated uses (c-use or p-use) be executed at least once (i.e., the test suite should execute all definition–use pairs). The all-uses criterion takes into account only the def-use pairs that have some path from the definition to the use in which the considered variable is not redefined. This special path is called def-clear path.

Table 1 presents the test requirements generated for structural testing criteria based on the DUG generated. Regarding control-flow criteria, 16 nodes, 22 edges and 13 paths are presented. Notice that there are only 13 paths because only one loop iteration was considered. If two iterations of each loop were considered, there would be more distinct paths. As for the all-uses criterion, there are 47 test requirements, but only the def-use pairs for the variable `number` are presented.

2.2. Symbolic Execution

The general idea behind symbolic execution is to represent the values of variables over the symbolic input values of a program or function [King 1976, Cadar and Sen 2013]. Considering the control-flow criterion, each execution path of the program can be represented as a sequence of constraints expressed as a function of the symbolic input values. Consider, for example, the following execution path of `factorization`: 1 2 3 4 6 7 8 9 7 13 3 14 15 16. To move from node 1 to node 2, `N` must satisfy the constraint `N > 0`. To move from node 3 to 4, `N` must satisfy the constraint `(prime <= N/2)`. Symbolic execution replaces `prime` by its initial value (Listing 1, Line 3), thus the constraint may be expressed as `(2 <= N/2)`. To move from node 4 to node 6, `number` and `prime` must satisfy the constraint `(number%prime != 0)`. Symbolic execution expresses this constraint as `(N%2 != 0)` after computing everything over input values.

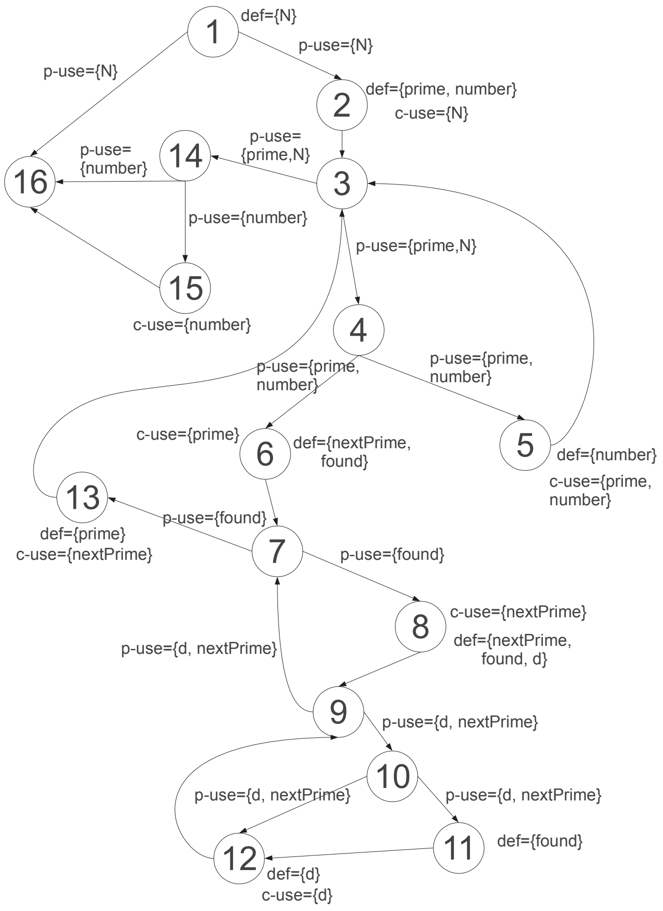

Figure 1. CFG of the `factorization` method

After symbolic execution, the aforementioned path is represented by this constraint sequence: {`N > 0`} ∧ {`2 <= N/2`} ∧ {`N%2 != 0`} ∧ {`0 == 0`} ∧ {`2 > (2+1)/2`} ∧ {`1 != 0`} ∧ {`2+1 > N/2`} ∧ {`N > 1`}. A constraint solver thus produces a solution that satisfies all constraints of the sequence. In this case, 5 is a value of `N` that satisfies all these constraint. If the number 5 is passed in as an input parameter to `factorization`, it will follow the same execution path presented above and it will cover the nodes, edges, and uses related to this path (see Table 1).

Programs under test may also have infeasible paths, i.e., they might lead to unsolvable constraints. Take the path 1 2 3 4 6 7 13 3 14 16 for instance. In node 6, `found` is set to 0 but it must satisfy the constraint `(found != 0)` so that the symbolic execution can go from node 7 to 13. Thus, this path is infeasible because symbolic execution generates the constraint `(0 != 0)`, which cannot be satisfied.

Table 1. Control and data flow test requirements

Criterion	Requirements
all-nodes	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
all-edges	(1–2), (1–16), (2–3), (3–14), (3–4), (14–15)
	(14–16), (15–16), (4-5), (4-6), (5–3), (6–7)
	(7–8), (7–13), (13–3), (8–9), (9-10), (9-7)
	(10–11), (10–12), (11–12), (12–9)
all-paths	...
(1-loop)	
	P01: 1 2 3 4 6 7 8 9 10 11 12 9 7 13 3 14 15 16
	P02: 1 2 3 4 6 7 8 9 10 11 12 9 7 13 3 14 16
	P03: 1 2 3 4 6 7 8 9 10 12 9 7 13 3 14 15 16
	P04: 1 2 3 4 6 7 8 9 10 12 9 7 13 3 14 16
	P05: 1 2 3 4 6 7 8 9 7 13 3 14 15 16
	P06: 1 2 3 4 6 7 8 9 7 13 3 14 16
	P07: 1 2 3 4 6 7 13 3 14 15 16
	P08: 1 2 3 4 6 7 13 3 14 16
	P09: 1 2 3 4 5 3 14 15 16
	P10: 1 2 3 4 5 3 14 16
	P11: 1 2 3 14 15 16
	P12: 1 2 3 14 16
	P13: 1 16
all-uses	(2–(14,16)), (2–(14,15)), (2–(4,5)), (2–(4,6)), (2–15)
(number)	(5–(14,16)), (5–(14,15)), (5–(4,5)), (5–(4,6)), (5–15)

The process of representing paths as constraint sequences to find suitable test data to execute that path can be repeated for each path of the program under test. This process generates test data to cover all test requirements generated by the structural testing criteria, except for those related to infeasible paths.

3. Approach description

Approaches that emphasize control-flow criteria generate test data only for the whole program or function [Pasareanu and Visser 2009, Godefroid 2012, Cadar and Sen 2013, Galler and Aichernig 2013]. Given the motivating scenarios in which the generation of test data to cover only a subset of test requirements is desirable (see Section 1), we present an approach to automatically generate test data for covering only test requirements selected by users. This approach generate test data to cover test requirements generated by both control and data-flow structural criteria.

Our approach is broken down and executed in five steps. It starts when a user provides the program under test along with the set of test requirements to be considered during test data generation. Testers need to inform a list of nodes and edges to generated test data to cover control-flow test requirements, and a list of def-use pairs or a list of variable names to cover data-flow test requirements. When a variable name is provided, all its def-use associations are taken into account during test data generation. We developed a prototype to support all steps of this process. Section 3.1 describes the prototype and Section 3.2 presents the details of each step of the approach supported by the prototype.

3.1. Prototype Implementation

We developed a proof-of-concept implementation to evaluate the efficiency of our approach. This prototype was implemented in Java and supports test data generation for Java programs. For its first version, we only used integer data types and we did not handle the integration between methods from the same class or package.

Figure 2 shows the architecture of our prototype, which has four components. `Instrumenter` instruments Java classes and generates the DUG to identify symbolic expressions and constraints. The instrumentation occurs at bytecode level. This is one of the advantages of this particular implementation since the source code is not required. One of the limitation of current approaches is that the source code of external libraries is usually unavailable, which ends up hampering integration testing [Cadar and Sen 2013]. In our prototype, however, it is possible to instrument external libraries since they are usually available in bytecode representations.

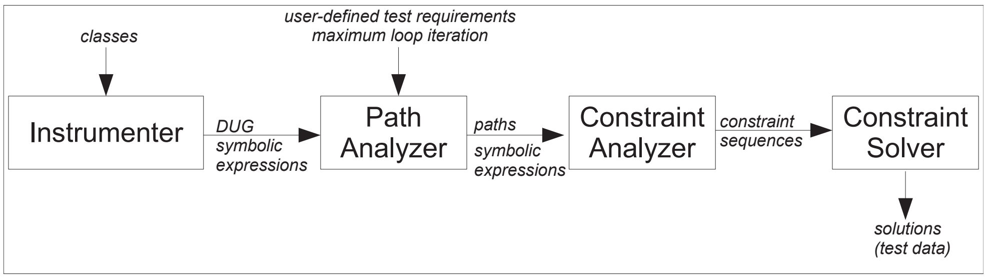

Figure 2. Architecture of the prototype

`Path Analyzer` receives a DUG and identifies all paths. Then, this component selects paths that exercise the test requirements provided by the users. Each selected path is sent to the component `Constraint Analyzer`, which generates a constraint sequence and identifies unsolvable constraints. Finally, constraint sequences are sent to the component `Constraint Solver`, which produces the test data to cover the user-defined test requirements. In this prototype, this component is implemented by Choco [Team 2008], an open source java constraint programming library.

3.2. Steps of the approach

In this section, we discuss each step of the proposed approach, as presented below. We use the example shown in Section 2 to further elucidate each step, considering that the tester wants to generate test data to cover only test requirements related to the variable `number`.

Step 1: Generate DUG. The tester must submit the name of the class to be tested, the list of test requirements and the maximum number of loop iterations to be considered. Listing 2 shows the snippet of the code used to invoke our prototype and generate test data for the `factorization` method (Listing 1). Notice that the user provided the class name (`MyMath.class`), the test requirements and the maximum number of of iterations (i.e., 3). As for the test requirements, the user decided to provide only the name of one variable (`number`), instead of selecting specific nodes, edges, or uses. The test requirements (all def-use pairs) related to this variable are shown in Table 1.

The component `Instrumenter` analyzes the program under test and generates a DUG for each method. As mentioned, the resulting DUG represents the test requirements related to control and data-flow criteria. Along with the DUG, the symbolic value of each variable and constraint over each input data is defined. The DUG for the `factorization` method is presented in Figure 1. Usually, the artifact used at this step is the source code, but it is also possible to use binary or bytecode representations. We do not provide an in-depth description of this step because each programming language or code artifact will require specific techniques to extract nodes, edges, and data usages from the program under test.

Step 2: Identify all paths. The component `PathAnalyzer` uses heuristics that employ depth first search algorithm to identify all possible paths given the DUG, the set of test requirements to be covered, and a maximum number of loop iterations. `PathAnalyzer` keeps the resultant paths in decreasing ordered according to the number of nodes. We adopted this strategy because executing paths with the highest number of nodes first may cover more test requirements than the others. Thus, few paths have to be selected and few constraints are sent to the constraint solver. At the first iteration in `PathAnalyzer`, one loop iteration is considered. At the *n*th iteration, *n* loop iterations are considered. Table 1 shows the list of paths generated for `factorization` (Listing 1) at the first iteration of `PathAnalyzer`.

Step 3: Find solutions The component `ConstraintAnalyzer` receives a list of paths produced by `PathAnalyzer` and a set of symbolic expressions related to the bytecode of the class under test. `ConstraintAnalyzer` executes an heuristic to select only paths that have potential to cover at least one of the user-defined test requirements. Starting from the first path of the list, the component generates a constraint sequence and send it to a constraint solver. If the sequence has a solution, the test requirements covered by the target path are excluded from the list of the test requirements to be covered. The next path is selected only if it covers a test requirements of the to be covered list. `ConstraintAnalyzer` select paths until one of those stop criteria are met: *(i)* there is no path left, *(ii)* all test requirements selected by the user are covered, *(iii)* the specified timeout expires, or *(iv)* the specified maximum number of loop iterations is reached.

In our example, the component `ConstraintSolver` received a set of constraint sequences sent by `ConstraintAnalyzer` and produced test data to cover the test requirements provided by the user. Figure 3 shows the six input data produced by the prototype (5, 2, 1, 10, 7 and 4) and the two test requirements that remained uncovered after 3 iterations (`5->(14-15)` and `5->15`).

4. Evaluation

In this section, we describe an initial evaluation of the proposed approach. We selected a set of nine programs found in the literature. These programs comprise branches, loops, and perform calculations on integer variables [Sedgewick and Wayne 2008, Zhu et al. 1997, Rapps and Weyuker 1982, Pasareanu and Visser 2004, JPF]. Table 2 lists the programs and some information about their DUGs. Notice that the selected programs have different characteristics, which allow us to evaluate diverse scenarios.

Using these programs and the prototype described in Section 3.1, we set up some experiments that aim to analyze the approach from different points of view. Their settings and results are discussed as follows. For the sake of mitigating sources of variability (e.g., garbage collection operations and just-in-time compilation), and thereby carrying out a more rigorous performance evaluation, the measured running times reported in this section are the average (mean) execution time across 30 runs. Execution times were measured in milliseconds and using an Intel Core i5-2450M 2.5GHZ with 6 GB RAM of physical memory running Windows 7.

Table 3 shows the average time to generate input data that cover all test requirements for all-nodes, all-edges, and all-uses. Notice that the all-uses criterion took more time in all programs. Considering `factorization`, for instance, generating all test requirements for all-uses took around 89% more time than all-nodes and approximately 78% more time than all-edges. However, in most cases, the overhead for generating test data for all-uses was not high. For instance, considering `concrete`, our prototype took around 57% more time to generate test data for the all-uses criterion than it did to create test data for the all-nodes criterion. We already expected this behavior since data-flow criteria tend to produce more test requirements and it takes extra steps to verify def-clear paths. Yet, the average time was below 7 seconds, which is quite reasonable. These results give some insights on the cost of using a symbolic execution-based approach to generate test data for data-flow criteria.

Listing 2. Snippet of code used to invoke our prototype.

```
String className = ''MyMath.class'';
TestReq[0] = new String[] {}; //nodes
TestReq[1] = new String[] {}; //edges
TestReq[2] = new String[] {}; //uses
TestReq[3] = new String[] {
  ''number''
}; //variables

int maxLC = 3; //maximum loop counts
testProject = Prototype(className,
       TestReq, maxLC);

for (ClassModel cl: testProject.getClasses()){
  for (MethodModel meth: cl.getMethods()) {
    for (TestData td: meth.getTestData()) {
      String varName = td.getVariable();
      String value = td.getValue();
      System.out.println(varName +
        '' = '' + value);
    }
  }
}
```

```
Listing 2 Output
Output: N=5, N=2, N=1, N=-10, N=7, N=4
Uncovered uses: number:5->15, number:5->(14-15)
```

Figure 3. Output of running the code in Listing 2.

Table 4 shows the average time to cover samples of test requirements (25%) derived from all-nodes, all-edges, and all-uses. These randomly selected samples cover scenarios where the test suites do not achieve maximum coverage for some criterion. In this experiment, we assume that, for instance, a test suite that covers 100% of all-edges will cover around 75% of all-uses; which leaves around 25% to be covered using symbolic execution. As shown in Table 4, our prototype achieved a reasonable performance for all programs (below 4.3 seconds). Our results would seem to demonstrate that all-uses is the most computationally intense criterion. In the worst case (`factorization`), the time spent on all-uses was around 114% longer than the time spent on all-edges. For `gcd2`, however, there was a slowdown of only 25% (Table 4).

Table 2. General information about the programs used in our evaluation.

Program	#loops	#nodes	#edges	#defs	#c-uses	#p-uses	#def-use associations	#simple paths
`concrete`	0	5	5	7	10	10	9	3
`factorial`	1	6	7	5	5	8	10	4
`factorization`	3	16	22	12	11	36	47	13
`gcd2`	1	4	4	5	4	4	5	2
`gcd`	1	7	9	4	9	12	24	4
`generatePattern`	2	12	15	5	3	20	28	5
`m`	2	10	12	7	6	18	18	8
`myMethod`	0	12	15	10	36	8	9	12
`pow`	1	9	11	10	20	10	17	8
Mean	1.22	9	11.11	7.22	11.56	14	18.56	6.56

Table 3. Average time (in ms) to generate all test requirements.

Program	All-Nodes	All-Edges	All-Uses
`concrete`	7.13	9.93	11.27
`factorial`	19.75	20.2	35.24
`factorization`	3396.1	3613.44	6431.2
`gcd2`	3.79	3.79	16.65
`gcd`	10	12.75	48.65
`generatePattern`	1496.79	1561.62	1813.17
`m`	85.96	92.48	151.79
`myMethod`	24.24	24.62	29.37
`pow`	14.37	20.03	45.55
Min	3.79	3.79	11.27
Max	3396.10	3613.44	6431.20
STD[‡]	1169.39	1240.72	2135.79

[‡]STD stands for standard deviation.

Table 5 shows the average times to cover one test requirement and all test requirements for a specific variable, respectively. These results are based on the all-uses criterion. The def-uses pairs (or variable) were randomly selected and represent two scenarios. First, the tester is trying to satisfy all-uses and need to come up with a unit test that covers a specific def-use pair. In this context, the tester needs an immediate response from the symbolic execution prototype. Second, the developer identifies a problematic variable and wants to test it with different input data. Consequently, a fast feedback from the prototype is also required. The results provide some promising insights: for the first scenario, the average time was below 1 second, and 2.7 seconds for the second scenario. Figure 4 shows the boxplots for the first scenario using the two programs in which the prototype's performance was below the average. Notice that, in the worst case (upper whisker in the boxplot representing the `factorization` method), the execution time was below 4 seconds.

Discussion. The lack of representativeness of the programs we chose to evaluate our prototype poses a threat to external validity. Apart from not being of industrial size, another potential threat to the external validity is that the chosen programs do not differ considerably in size and complexity. Consequently, we recognize that our research can be seen as an initial exploration into understanding how symbolic execution can be used to generate test data for data-flow criteria. The goal of our experiment was to provide some evidence of the efficiency and applicability of our implementation in academic settings. Therefore, further research is needed to support and expand the results of our initial evaluation, yet we believe that our initial results are promising.

Table 4. Average time (in ms) to generate 25% of the test requirements.

Program	All-Nodes	All-Edges	All-Uses
`concrete`	1.62	3.96	5.55
`factorial`	10.62	15.58	20.13
`factorization`	973.41	2003.31	4293.89
`gcd2`	3.68	3.62	4.55
`gcd`	3.86	8.1	24.58
`generatePattern`	740.1	1267.31	1423.51
`m`	31.27	43.17	48.72
`myMethod`	11.93	10.96	13.27
`pow`	9.55	12.17	22.17
Min	1.62	3.62	4.55
Max	973.41	2003.31	4293.89
STD‡	377.85	738.35	1442.97

‡STD stands for standard deviation.

Table 5. Average time (in ms) for one test requirement and one variable.

Program	One Test Requirement	One Variable
`concrete`	3.15	6.33
`factorial`	19.4	29
`factorization`	808.65	2624.6
`gcd2`	4.75	10.5
`gcd`	6.47	31
`generatePattern`	434	858.5
`m`	19.7	56.5
`myMethod`	12.37	16.5
`pow`	10.5	32.66

The current state of our prototype prevented us from conducting experiments with more complex programs [Eler et al. 2014]. The prototype handles only primitive data as integers. It does not handle issues such as concurrency and the integration with other methods or external libraries. Moreover, optimized heuristics and analysis techniques are required to enable testing complex systems and eliminate infeasible requirements. Nevertheless, it is worth mentioning that the purpose of our paper is not to present a comprehensive way of avoiding the path explosion problem, but rather to introduce data-flow criteria and related data generation in this testing context.

5. Related Work

For more than three decades, symbolic execution has been used in the context of software testing to generate test data to cover testing criteria [King 1976, Cadar and Sen 2013]. However, symbolic execution poses several challenges. Several approaches have been proposed to circumvent problems such as path explosion, concurrency, complex data, limited constraint solvers, and integration with external libraries [Pasareanu and Visser 2009, Godefroid 2012, Cadar and Sen 2013].

Many improvements have been achieved in this field and several tools and techniques were proposed, including DART [Godefroid et al. 2005], CUTE [Sen and Agha 2006], JCUTE [Sen and Agha 2006], Klee [Cadar et al. 2008], PEX [Tillmann and De Halleux 2008], LCT [Kahkonen et al. 2011], Klover [Li et al. 2011], among many others [Galler and Aichernig 2013]. Regardless the technique to mitigate symbolic execution limitations and the technology employed, most of them are generally used to generate test data to satisfy control-flow criteria (lines of code, instructions, branches, paths) for an entire program or function.

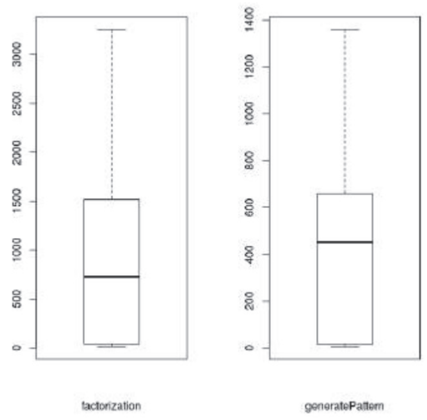

Figure 4. Execution time for one test requirement for the programs `factorization` and `generatePattern`).

From a practical standpoint, however, testers may already have a test suite implemented with test cases and oracles. Sometimes it is important to generate test data to cover only specific parts of the code, often related to data-flow. Specifically, Gupta et al. [Gupta et al. 2000] propose an approach that aims to cover a given branch (which is a test requirement for all-edges). The paper focuses on selecting paths, that lead to the branch under test, with less resistance for test data generation. Their approach dynamically searches for better paths, eliminating infeasible or likely infeasible paths along the process.

Although the theoretical foundations for data-flow criteria have been laid for more than twenty five years, there is scant tool support for the criteria. Horgan and London [Horgan and London 1991] report on the difficulties of implementing data-flow criteria for the C language. By adapting the family of data-flow coverage criteria of Rapps and Weyuker [Rapps and Weyuker 1985], which is tailored to an idealized programming language, Horgan and London defined the following data-flow coverage measures for C programs: blocks, decisions, definitions, p-uses, c-uses, all-uses, and du-paths. They also implemented these data flow coverage measures in testing tool called ATAC. This tool helps testers to improve coverage by highlighting non-covered elements. By perusing the dataflow relationships in a program, testers hand-craft tests in hopes of increasing coverage. Similarly to our approach, ATAC enables testers to focus on a given slice of code at a time. In effect, the main benefit of using ATAC is that the resulting tests are aimed at exercising non-covered elements. The main drawback is that ATAC does not support test data generation, which renders test creation for large or even moderately-sized programs burdensome.

Most data-flow criteria and implementations thereof are tailored to procedural languages. Although some of these criteria can be applied to object-oriented programs, they fail to capture the data-flow interactions that take place when users of a class invoke sequences of methods in a random order. Harrold and Rothermel [Harrold and Rothermel 1994] came up with an approach to support the testing for all types of data-flow relationships in a class. They define three levels of dataflow testing: *(i)* intra-method, which is aimed at testing individual methods, *(ii)* inter-method, whose purpose is testing methods in a class that interact through invocations, and *(iii)* intra-class, whose goal is testing sequences of method calls. In their approach, classes are represented as single-entry, single-exit programs, which makes it possible to identify def-use pairs for these types of data-flow criteria.

Several tools have been developed to perform many types of static analysis. A notable example is Frama-C [Cuoq et al. 2012]. This source code analysis platform allows to verify that C programs comply with a given formal specification. In addition, using Frama-C it is possible to slice a given program into smaller chunks of code and navigate the dataflow of these chunks, from definition to use as well as from use to definition. As pointed out by Cuoq et al. [Cuoq et al. 2012], Frama-C provides its users with a vast collection of plugins. One of these plugins is PathCrawler [Williams 2010], which automatically finds test inputs that cover all the feasible execution paths of a given C function. Differently from our tool, PathCrawler is based on a dynamic symbolic execution method. Thus, the program under test is executed under each generated test case. Afterwards, traces of the resulting execution paths are examined to determine whether the test cases activated the intended execution paths.

Although much of the groundwork for test case generation has been laid, much remains to be explored. For instance, there is a lack of approaches that aim to generate inputs to cover data-flow test requirements. Moreover, most of the existing approaches are focused on reaching a full criterion coverage. To the best of our knowledge, only the research of Gupta et al. [Gupta et al. 2000] deals with covering a given test requirement for branch coverage.

6. Concluding Remarks

This paper presented an approach to generate test data only for test requirements (for both control and data-flow) selected by users. A preliminary evaluation of the approach showed that, considering the whole program, generating input data for data-flow criteria entails more computation, especially because they produce more test requirements than control-flow criteria. Another reason is that many data-flow test requirements are infeasible and the test data generation only stops when a given time budget elapses or a specified number of loop iterations is reached. In our opinion, this is not a prohibitive problem since there are techniques to identify and remove infeasible data-flow test requirements [Vergilio et al. 2006].

However, our aim was not to show that generating test data for data-flow criteria is as effective as for control-flow. The purpose was to support testers on generating test data for covering test requirements which were not covered by existing test suites. Another contribution of our approach is that users may direct test data generation without knowing specific details about structural testing criteria. They can provide a variable name and our prototype is able to generate test data covering all test requirements related to this variable, especially def-use associations.

As future work, we intend to extend the prototype to deal with more complex data types and different criteria (like all-du-paths and MC/DC). Yet regarding our prototype, we intend to allow users to select lines of the code instead of nodes, and sequence of lines instead of edges. Moreover, heuristics and techniques to perform optimizations on the path exploration algorithm and constraint sequences generation could be investigated. Finally, we intend to explore the application of data-flow and relative input data generation in the context of dynamic symbolic execution (concolic testing) [Sen and Agha 2006].

References

Beizer, B. (1990). *Software Testing Techniques*. Van Nostrand Reinhold Co., New York, NY, USA, 2nd edition.

Cadar, C., Dunbar, D., and Engler, D. (2008). Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 209–224. USENIX Association.

Cadar, C. and Sen, K. (2013). Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90.

Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. (2012). Frama-C: A Software Analysis Perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods (SEFM)*, pages 233–247. Springer-Verlag.

Eler, M. M., Endo, A. T., and Durelli, V. H. S. (2014). Quantifying the Characteristics of Java Programs that May Influence Symbolic Execution from a Test Data Generation Perspective (to appear). In *COMPSAC 2014*, pages 1–10. Vasteras, Sweeden.

Galler, S. and Aichernig, B. (2013). Survey on Test Data Generation Tools. *International Journal on Software Tools for Technology Transfer*, pages 1–25.

Godefroid, P. (2012). Test Generation Using Symbolic Execution. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 18, pages 24–33. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Godefroid, P., Klarlund, N., and Sen, K. (2005). DART: Directed Automated Random Testing. *ACM SIGPLAN Notices*, 40(6):213–223.

Gupta, R., Mathur, A., and Soffa, M. (2000). Generating Test Data for Branch Coverage. In *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering*, pages 219–227.

Harrold, M. J. and Rothermel, G. (1994). Performing Data Flow Testing on Classes. *ACM SIGSOFT Software Engineering Notes*, 19(5):154–163.

Horgan, J. R. and London, S. (1991). Data Flow Coverage and the C Language. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, pages 87–97. ACM.

JPF. Java Path Finder: the Swiss Army Knife of Java Verification. [January 2014].

Kahkonen, K., Launiainen, T., Saarikivi, O., Kauttio, J., Heljanko, K., and Niemel, I. (2011). LCT: An Open Source Concolic Testing Tool for Java Programs. In *6th Workshop on Bytecode Semantics, Verication, Analysis and Transformation*, pages 1–6. ETAPS.

King, J. C. (1976). Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394.

Li, G., Ghosh, I., and Rajan, S. P. (2011). KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs. In *Proc. of the 23rd int. Conference on Computer aided verification*, pages 609–615. Springer-Verlag.

Myers, G. J., Sandler, C., Badgett, T., and Thomas, T. M. (2004). *The Art of Software Testing*. John Wiley & Sons, Inc., Hoboken, New Jersey.

Pasareanu, C. S. and Visser, W. (2004). Verification of java programs using symbolic execution and invariant generation. In *SPIN*, volume 2989 of *Lecture Notes in Computer Science*, pages 164–181. Springer.

Pasareanu, C. S. and Visser, W. (2009). A Survey of New Trends in Symbolic Execution for Software Testing and Analysis. *International Journal on Software Tools for Technology Transfer (STTT*, 11(4):339–353.

Ramamoorthy, C., Ho, S.-B. F., and Chen, W. (1976). On the Automated Generation of Program Test Data. *IEEE Transactions on Software Engineering,*, SE-2(4):293–300.

Rapps, S. and Weyuker, E. J. (1982). Data flow analysis techniques for test data selection. In *Proceedings of the 6th International Conference on Software Engineering*, pages 272–278. IEEE.

Rapps, S. and Weyuker, E. J. (1985). Selecting Software Test Data Using Data Flow Information. *IEEE Transaction on Software Engineering*, 11(4):367–375.

Sedgewick, R. and Wayne, K. (2008). *Introduction to programming in Java - an interdisciplinary approach*. Pearson/Addison Wesley.

Sen, K. and Agha, G. (2006). CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *In CAV*, pages 419–423. Springer.

Team, T. C. (2008). Choco: An Open Source Java Constraint Programming Library. In *Proceedings of the Workshop on Open-Source Software for Integer and Contraint Programming*, pages 1–7. ACM.

Tillmann, N. and De Halleux, J. (2008). Pex: White Box Test Generation for .Net. In *Proceedings of the 2nd International Conference on Tests and Proofs*, pages 134–153. Springer-Verlag.

Vergilio, S. R., Maldonado, J. A. C., and Jino, M. (2006). Infeasible paths in the context of data flow based testing criteria: identification, classification and prediction. *Journal of the Brazilian Computer Society*, 12:71–86.

Williams, N. (2010). Abstract Path Testing with PathCrawler. In *Proceedings of the 5th Workshop on Automation of Software Test (AST)*, pages 35–42. ACM.

Zhu, H., Hall, P. A. V., and May, J. H. R. (1997). Software Unit Test Coverage and Adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366–427.