Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 14, no. 6, June 2018, pp. 1251-1262
DOI: 10.23940/ijpe.18.06.p16.12511262

Decomposing Constraints for Better Coverage in Test Data Generation

Ju Qian*, Kun Liu, Hao Chen, Zhiyi Zhang, and Zhe Chen

College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, 210016, China

Abstract

In black-box testing, a possible choice for test data generation is to derive test data from interface constraints using some constraint solving techniques. However, directly performing constraint solving to the whole constraint formula may not be able to fully leverage the information embodied in a constraint. Thus, it is difficult to obtain a high coverage test set. For example, when solving a constraint ($a > 0$ or $b < 0$) in a whole, we cannot guarantee that data covering sub-constraint $b < 0$ will be involved in the test set. To address the problem, in this paper, we firstly define a hierarchy of coverage criteria at the specification constraint level. Then, algorithms are designed to decompose constraints according to such coverage criteria and to generate test input sets. The experiments on a set of benchmark programs show that decomposing constraints according to constraint-level coverage criteria does effectively lead to better coverage in test data generation.

Keywords: test data generation; black-box; constraint solving; constraint decomposition

(Submitted on March 6, 2018; Revised on April 21, 2018; Accepted on May 19, 2018)

1. Introduction

Software testing is a practical and widely used technique for ensuring the quality of software applications. During software development, a large amount of effort is often spent on software testing. To reduce the cost of software testing, the testing process needs to be automated as much as possible. An important step in the automation of software testing is the automatic generation of test data. The test data generation topic has been studied for decades, and still gets increasingly more attention in recent years since the software development paradigms continue to evolve and new problems continue to emerge.

The state-of-the-art test data generation approaches can be roughly categorized into white-box and black-box [4]. Although the recent advances in techniques, like symbolic execution [11, 17] have significantly improved the efficiency and usability of white-box test data generation, the black-box test data generation approaches are still of great importance. This is especially true when testing a software application running on an embedded device, the source code or even binary code may be unavailable. In that case, only black-box testing techniques can be used. The black-box test data generation techniques can be used to generate a single input passed into a software application [15] or an input sequence which can be used to test reactive software applications like a GUI window [6]. This paper focuses on the former type of test data generation problem.

In the generation of test sets consisting of each single test input, a basic approach is to generate test data by randomly selecting values from the input domain. However, there are often constraints on the input domain, randomly selecting values cannot guarantee both the reasonable inputs (valid data) and unreasonable inputs (invalid data) are selected. An improvement to such problem is using grammars [15] or formats [12] that represent the structural requirements of the software inputs to help derive more elaborated test data. With such input grammars or formats, we can only guarantee the well and bad formatted data are included in the generated test set, but cannot guarantee those logically reasonable and unreasonable data are covered.

To generate logically reasonable and unreasonable data, a possible choice is using certain interface constraints as a basis

* Corresponding author.
E-mail address: jqian@nuaa.edu.cn

for test data generation. For example, given an interface constraint $a > b$ for two inputs a and b, by conducting constraint solving for $a > b$ and $\neg(a > b)$, we can get both valid inputs and invalid inputs for the tested application. Modern SMT solvers like Z3 [10] have made such an approach practical to use.

However, when the constraints are more complex, e.g., $(a > 0 \vee b < 0)$, constraint solving to the whole constraint formula may not be able to fully leverage the information embodied in a constraint. We may only generate values that cover $a > 0$, not values covering $b < 0$. In such case, it will be difficult to obtain a high coverage test set. To get higher coverage, it is better not to depend on the internal randomness of a constraint solver, but decompose the constraint controlled input space to ensure the quality of generated data. In previous work like [1, 7, 8, 19], researches are already using the coverage of concepts like disjunction normal form (DNF) to decompose the constraints. However, the existing work does not define a clear hierarchy of coverage criteria with respect to a constraint formula to guide different grains of test data generation. What's more, none of the existing work has validated the effectiveness of constraint decomposition to test data generation via an experimental study. Do the constraint decompositions really lead to better coverage? How much improvement can be achieved by constraint decomposition compared with an approach relying on the randomness in a constraint solver to generate test data? Such problems are unclear.

To address the problems, this paper presents a constraint decomposition approach to help get high coverage test data from the specification constraints. In the first step, we define a hierarchy of coverage criteria on the specification constraints, which includes decision coverage (DC), condition coverage (CC), decision condition coverage (DCC), and normal form clause coverage (NFCC). Then, black-box algorithms are designed to obtain test data according to such criteria, with the support of Z3 SMT solver. We tested the proposed approach on GNU Coreutils benchmark set [21]. The experimental results show that the test data generation guided by constraint-level coverage criteria can achieve about 15% improvement in the source-code level coverage of test sets compared with that unguided by such criteria with the same number of test cases, and the generation time cost of using criteria CC and DCC is low. This suggests such criteria are valuable for practical use.

2. Demonstration Example

We use a modified version of the typical Triangle Classification application [5] as an example to demonstrate the proposed approach. The modified version only accepts isosceles or equilateral triangles, and its specification constraint on the three input triangle sides a, b, and c is the following:

$$C = (a + b > c) \wedge (a + c > b) \wedge (b + c > a) \wedge ((a = b) \vee (b = c) \vee (a = c)).$$

Both valid test data and invalid test data can be generated for the triangle application according to constraint C. A simple way for generation is using some constraint solver, e.g., a SMT solver, to solve constraint C to get the test data. However, directly solving constraint C cannot guarantee that all situations described by C are included in the result test data set. We may obtain a test set that only satisfies one condition $(a = b)$, without covering the cases that satisfy $(b = c)$ and $(a = c)$, respectively. Setting random solving option in the constraint solver, e.g., setting "*smt.arith.random_initial_value*" option in Z3 [10], may help relieve the problem to some extent; however, the problem still remains.

One strategy that can be used to help derive better coverage test data set is to force the test data generation to cover each atomic condition in constraint C, namely requiring these conditions to evaluate to both true and false under the generated data set. For example, by requiring the situations that atomic conditions $(a+b>c)$, $(a+c>b)$, $(b+c>a)$, $(a=b)$, $(b=c)$, and $(a = c)$, each evaluates as true and false during test generation. Non-triangles and isosceles triangles with different equal side pairs can be included in the result test set. This can obviously lead to better coverage.

However, for the triangle problem, covering each atomic condition and its negation may still not include the situation that the inputted side values form a normal triangle, which is neither isosceles nor equilateral. This is because such a normal triangle requires a combination of three conditions $\neg(a = b)$, $\neg(b = c)$, $\neg(a = c)$ to be satisfied. The combination cannot be guaranteed by an atomic condition coverage. To handle this problem, we introduce another coverage strategy for the specification condition C. The strategy requires each sub-clause in the DNF of C and each negation to the sub-clauses in the CNF (conjunction normal form) of C be covered. Under such strategy, there must be a test input satisfying constraint $\neg((a = b) \vee (b = c) \vee (a = c))$ in the generated test data set. Now, a normal triangle will be included in the result test set, and a situation underived under the covering atomic condition strategy can be covered.

As shown by the example, decomposing the constraints and defining constraint-level coverage criteria according to such

decomposition can guide the test data generation process to reach better coverage, even under a black-box testing context. This paper will introduce such an approach in detail and conduct experiments to further validate its effectiveness.

3. Decomposing Constraints according to A Hierarchy of Coverage Criteria

An overview of our approach is shown in Figure 1. The inputs of the approach include a constraint formula specifying the interface pre-condition of a given application and a selected constraint-level coverage criterion. The pre-condition constraint describes the valid input data that are supposed be processed by the tested application. It can be provided by a user or even be inferred by some tools. The output is a test data set satisfying the test criterion. The test data set may contain both the valid inputs and the invalid inputs. The whole test data generation process includes two main steps: (1) constraint decomposition according to the test criterion, and (2) test data generation based on SMT constraint solving.

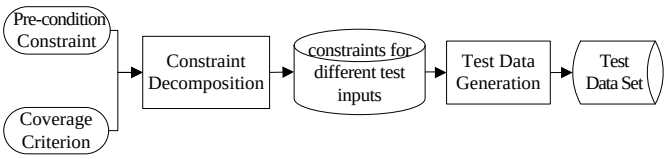

Figure 1. The test data generation process

In the first step, we decompose the interface constraint into a couple of separated constraints each corresponding to a partition of the input space. Different coverage criteria have different constraint decomposition rules. For example, if we want the test data to cover each sub-clause in the disjunction normal form (DNF) of the original interface constraint, the interface constraint will be decomposed to a set of DNF sub-clause constraints, each corresponding to a group of test inputs. In the second step, we solve the decomposed constraints with a SMT solver to generate test data.

To generate adequate test data for a given application, we define several constraint-level coverage criteria as guidance. These criteria cover different situations described by the specification constraints. Testing under such criteria can check more execution scenarios for the given application. The coverage criteria include decision coverage, condition coverage, decision condition coverage, and normal form clause coverage. Such coverage concepts are partly borrowed from the code coverage criteria in white-box testing.

3.1. Decision Coverage (DC)

In white-box testing, the decision coverage requires the test cases cover every true or false outcome of each decision. For constraint-based testing, our decision coverage requires the test data satisfies not only the specification constraint (valid data) but also its negation (invalid data).

Definition (*decision coverage*): For a pre-condition constraint *C*, a test set *T* is said to achieve *decision coverage* if the following condition is satisfied:

$$\exists t \in T, \text{eval}(C, t) = \text{true} \wedge \exists t' \in T, \text{eval}(C, t') = \text{false},$$

where eval(*C*, *t*) checks whether a test input *t* satisfies a constraint *C*.

3.2. Condition Coverage (CC)

In white-box testing, the condition coverage requires the test cases cover every true or false outcome of each condition. In constraint-based testing, our condition coverage requires the test data satisfies not only the atomic conditions in the pre-condition constraint but also their negations.

Definition (*condition coverage*): For a pre-condition *C*, a test set *T* is said to achieve condition coverage if the following condition is satisfied:

$$\forall e \in \text{BExpr}(C)\{ \exists t \in T, \text{eval}(e, t) = \text{true} \wedge \exists t' \in T, \text{eval}(e, t') = \text{false} \},$$

where BExpr(*C*) is a set of all atomic conditions in the pre-condition *C*.

3.3. Decision Condition Coverage (DCC)

In constraint-based testing, our decision condition coverage requires that test data satisfies not only the decision coverage but also the condition coverage under the given pre-condition.

Definition (*decision condition coverage*): For a pre-condition C, a test set T is said to achieve decision condition coverage if the following condition is satisfied:

$$\exists t \in T, \text{eval}(C, t) = \text{true} \wedge \exists t' \in T, \text{eval}(C, t') = \text{false}$$
$$\wedge \forall e \in \text{BExpr}(C)\{ \exists t \in T, \text{eval}(e, t) = \text{true} \wedge \exists t' \in T, \text{eval}(e, t') = \text{false} \}.$$

3.4. Normal Form Clause Coverage (NFCC)

To investigate whether decomposing to find-grained combinations can lead to more coverage, we introduce a normal form clause coverage (NFCC). This coverage criterion requires the test data satisfies not only each sub-clause in the disjunction normal form of the given pre-condition but also the negation of each sub-clause in the conjunction normal form. In this way, different situations that cause the input data to be valid and invalid can be covered.

Definition (*normal form clause coverage*): For a pre-condition C, a test set T is said to achieve normal form clause coverage if the following condition is satisfied:

$$\forall f \in \text{DNF}(C)\{ \exists t \in T, \text{eval}(f, t) = \text{true} \} \wedge \forall f \in \text{CNF}(C)\{ \exists t' \in T, \text{eval}(f, t') = \text{false} \},$$

where DNF(C) is a set of all disjunction normal form sub-clauses, while CNF(C) is a set of all conjunction normal form sub-clauses.

For example, if the pre-condition is:

$$C = (a + b < 100 \wedge a > 10) \vee a + b > 200,$$

then a test set $\{(a \mapsto 11, b \mapsto 1), (a \mapsto 102, b \mapsto 99), (a \mapsto 50, b \mapsto 100), (a \mapsto 1, b \mapsto 1)\}$ satisfies the normal form clause coverage.

3.5. A Hierarchy of Coverage Criteria for Constraint Decomposition

The relationships between the different coverage criteria introduced in this paper are shown in Figure 2. The coverage hierarchy includes four kinds of coverage criteria: DC, CC, NFCC, and DCC. If a test set T satisfies DCC, then it must satisfy DC and CC. If a test set T satisfy NFCC, then it must satisfy DC. NFCC does not have a direct relationship with DCC and CC.

We will decompose a specification constraint into atomic constraints and CNF/DNF sub-clauses according to such coverage criteria and then generate test data set. The detailed algorithms will be presented in the next section. In these four coverage criteria, DC coverage only needs to generate two test inputs, which is too small for practical use. Therefore, in this study, we mainly focus on the rest three coverage criteria: CC, DCC, and NFCC.

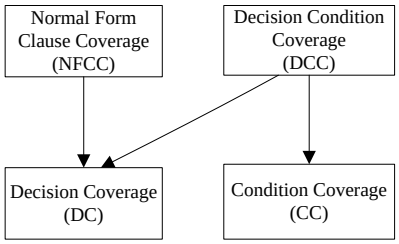

Figure 2. A hierarchy of constraint-level coverage criteria

4. Test Data Generation

This section introduces the algorithms to generate test set for the coverage criteria proposed in Section 3. These generation algorithms are based on a function Z3() which wraps Z3 SMT solver to generate data satisfying a given constraint.

4.1. Test Data Generation for CC

To generate a test set satisfying condition coverage, we need to generate test data satisfying each atomic condition e in constraint formula C ($e \in \text{BExpr}(C)$) and the negation of e, i.e., $\neg e$. Since a test data satisfying a condition e may also satisfy other conditions, directly generating test data with respect to each condition may produce redundant test inputs. Therefore, we remove other satisfied conditions after solving a constraint e. The algorithm used to generate the whole test set for CC is presented in Algorithm 1.

In the algorithm, a worklist W containing all the conditions that need to be covered are created. Then, we use Z3( ) to generate a test input t for each condition e_i in W. In order to generate a reasonable and complete input, we join e_i and pre-condition C (or its negation $\neg C$) for the input data solving. After generation, e_i will be removed from the worklist, and any condition that is also satisfied under input t will also be removed from worklist W. The generation stops when there is no more element in W.

Algorithm 1. Test data generation for condition coverage

```
function CC( C ): T
   C: condition constraint
   T: result test set
begin
   W := BExpr(C) ∪ { ¬ e | e∈BExpr(C)};
   while W ≠ ∅ do
       remove one condition e_i from W;
       t := Z3(e_i ∧ C);
      if t ≠ null then
          T := T ∪ {t};
      else
          t := Z3(e_i ∧ ¬ C);
          T := T ∪ {t};
      end
      foreach e_k ∈ W do
          if eval(e_k, t)== true then
              W := W – { e_k };
          end
      end
   end
   return T;
end
```

For example, given an interface constraint C:

$$C = a + b < 100 \wedge a > 10 \vee a + c > 200$$

A worklist $W = \{e_1 = (a + b < 100),\ e_2 = (a + c > 200),\ e_3 = (a > 10), \neg e_1, \neg e_2, \neg e_3\}$ is created. Then, we use Z3 to generate a test input $t_1 = (1, 1, 200)$ for e_1. Conditions e_2 and $\neg e_3$ are also satisfied under t_1. Therefore, we remove e_2 and $\neg e_3$ from W. Now, W becomes $\{e_3, \neg e_1, \neg e_2\}$. We use Z3 to generate a test input $t_2 = (50, 100, 100)$ for e_3. Conditions $\neg e_1$ and $\neg e_2$ are also satisfied under t_2 and hence will be remove from W. Then, W will be empty, and the generation stops. The final test data set will be $T(a, b, c) = \{(1, 1, 200), (50, 100, 100)\}$.

Sometimes a user may want to control the number of generated test inputs. For such cases, we slightly modify function CC() in Algorithm 1 to $CC_k()$ to generated only k test inputs. In $CC_k()$, if the number of generated test inputs reaches k, we immediately stop generation. If the number of generated test inputs cannot reach k, we repeat the generation process and depend on the constraint solver's internal randomness to generate additional test data.

4.2. Test Data Generation for DCC

For a program, under a pre-condition constraint C, a test set satisfying decision condition coverage can be generated by solving data to cover each condition and if necessary, add more test data to ensure decision coverage. More specifically, we first use Algorithm 1 to generate test data satisfying each condition in BExpr(C) and their negations. Then, the test set is checked to see whether condition C and its negation $\neg C$ are satisfied. If the test set does not satisfy C or $\neg C$, we will use Z3 to generate an additional test input to make such condition being satisfied. For the generated test data, when there is no limitation on the number of test inputs, all data satisfying C and satisfying $\neg C$ will be merged into the final test set; otherwise, we select half test inputs from the ones satisfying C and another half from the ones satisfying $\neg C$ to construct the final test set.

For example, given an interface constraint:

$$C = a + b < 100 \wedge a + c > 200,$$

we will get a test set of T_{cc}={(1, 1, 200), (50, 100, 100)} satisfying $a + b < 100$, $a + c > 200$, and their negations. Both the total constraint C and its negation are satisfied under T_{cc}; hence, we immediately get a test set achieving decision condition coverage.

4.3. Test Data Generation for NFCC

For a program, under a pre-condition C with a set of disjunction normal form sub-clauses DNF(C) and a set of conjunction normal form sub-clauses CNF(C), to generate a test set satisfying NFCC coverage, we need to generate test data satisfying each sub-clause in DNF(C) and test data unsatisfying each sub-clause in CNF(C). The algorithm to generate such data is presented in Algorithm 2.

Algorithm 2. Test data generation for NFCC

```
function NFCC(C, N): T
   C : condition constraint;
   N:  max number of tests
   T : result test set
begin
   F := DNF(C) ∪ {¬ e | e∈ CNF(C)};
   while F ≠ ∅ do
      remove e from F;
      if N=undefined or | T |< N then
            t := Z3(e);
            T := T ∪ { t };
      end
    end
    return T;
end
```

For example, given an interface constraint C:

$$C = (X \wedge Y) \vee Z,$$

where $X = (a + b < 100)$, $Y = (a > 10)$, and $Z = (a + b > 200)$. First, we find the set of normal form sub-clauses needing to be covered, i.e., $F = \{ f_1 = X \wedge Y, f_2 = Z, f_3 = \neg (X \vee Z), f_4 = \neg (Y \vee Z)\}$. Then, function Z3( ) is called to generate test inputs t_1 = (11, 1), t_2 = (102, 99), t_3 = (50, 100), and t_4 = (1, 1) for f_1, f_2, f_3, f_4, respectively. The final test data set will be: T(a, b) = {(11, 1), (102, 99), (50, 100), (1, 1)}.

In the computation of CNF/DNF for constraint C, the number of sub-clauses in a CNF/DNF can be of exponential size to the number atomic constraints. To avoid generating too many CNF/DNF sub-clauses and control computing time, we allow the user to set up bounds for the number of normal form sub-clauses and for the computing time. If the computing process reaches such bounds, we will stop applying the distributive laws in propositional logic in the normal form transformation. If the there are still too many CNF/DNF sub-clauses after such controlling, we will use the mechanism in Algorithm 2 to control the number of generated test cases.

5. Experiment

To validate the effectiveness of the proposed approach, we implemented the approach and tested it on Coreutils benchmark set [21] to answer the following research questions:

RQ(1): Does the approach guided by constraint-level coverage criteria outperform a normal constraint-based approach that is unguided by those criteria?

RQ(2): How is the achieved coverage of our proposed approach compared with a white-box approach that splits input constraint space according to execution paths instead of certain black-box constraint decomposition rules?

RQ(3): Which constraint-level coverage criterion leads to the best coverage?

RQ(4): How is the efficiency of our proposed approach compared with other ones?

5.1. Experiment Setup

In our experiment, we randomly select 16 GNU Coreutils 6.11 programs as the experimental subjects. They all are basic file, shell, and text manipulation utilities on the GNU system. All the experiments were conducted on Ubuntu 14 with Intel Xeon X5650 CPU. The information of the tested subjects is listed in Table 1; columns 1 to 3 show the subject's name, description, and lines of code including library code.

To answer RQ(1), we implemented an approach called CRAND unguided by constraint-level coverage criteria and compare our proposed approach with it on statement and branch coverage to see if the proposed approach outperform an approach that is unguided by those criteria. CRAND randomly explores the constraint space and alternately obtains solutions satisfying and unsatisfying the given specification constraint as the test data.

Table 1. The experiment subjects

Subject	***Description***	***LOC***
base64	encode/decode data and print to standard output	3989
chcon	change file security context	4343
chgrp	change group ownership	4278
comm	compare two sorted files line by line	3997
cut	remove sections from each line of files	4195
dircolors	color setup for ls	4093
expand	convert tabs to spaces	3916
expr	evaluate expressions	9565
fold	wrap each input line to fit in specified width	3891
mknod	make block or character special files	3840
paste	merge lines of files	3837
pathchk	check whether file names are valid or portable	3857
printf	execute programs via entries in the mailcap file	4251
split	split a file into pieces	4428
sum	checksum and count the blocks in a file	4068
tsort	perform topological sort	3856

To answer RQ(2), we compare the statement and branch coverage achieved by our proposed methods with that achieved by an optimized symbolic execution based white-box method implemented in KLEE [9]. For a subject like *paste*, KLEE's command line arguments to generate test cases are:

```
klee –optimize --output-dir=paste --only-output-states-covering-new -environ=test.env --run-in=/tmp/sandbox --max-time=3600.
-watchdog --search=random-path --search=nurs:covnew ./paste.bc --sym-args 0 1 2 --sym-args 0 1 5 --sym-stdin 8
```

For RQ (1) and RQ (2), our approach usually only generates a limited number of test cases depending on the constraint scale, while CRAND and KLEE can generate almost unlimited number of test inputs. It is difficult to compare them fairly. To this end, we set a common bar for test data generation - generating *N* number of test cases. *N* is determined according to KLEE (since it is the slowest method). We use the number of test cases generated by KLEE in one hour as *N*. The test generation of CRAND is forced to stop when the number of generated test data reaches *N*.

To answer RQ (3), let the generations guided by CC, DCC, and NFCC (they are called CC, DCC, and NFCC methods in short later) run without the test case number restriction to see which coverage criteria of our approach can lead to the best test coverage. Due to the fact that the NFCC method sometimes takes a great amount of time to find and solve the CNF sub-clauses during the experiment, we limit its constraints decomposition and solving time to 3600s.

For RQ (4), analyze the time consumed by each method when there is a restriction on the number of generated test cases and when there is no such restriction to compare the efficiency of different methods.

Our methods and CRAND are black-box approaches to generate test data under given interface constraints, while benchmarks for such constraints are hard to find. As an alternative solution, we merge the paths constraints reported by KLEE on different program paths to simulate the user-provided specification constraints. More specifically, we select *K*

number of path constraints $\{C_1, C_2, ..., C_K\}$ with widest coverage on the occurred constraint variables and constraint forms and use their disjunction C^* as the interface constraint,

$$C^* = \{ C_1 \vee C_2 \vee ... \vee C_K \}.$$

Inputs not conforming to C^* are regarded as invalid inputs. Although the valid/invalid partition of the input space depends on an application's business logic, the simulation can at least ensure that the execution paths covered by test data satisfying such specification constraints are representative. To avoid too high constraint obtaining cost, this experiment sets K=15.

In this paper, a Java version of Z3 is used to do constraint solving. The proposed approach is also implemented in Java. After executing the tested programs, we use gcov [22] to collect the coverage rates.

5.2. Experimental Results

- **Research Question (1)**

The results for RQ(1) are shown in Table 2, where the first 3 top columns present the name, statement coverage (SC), and branch coverage (BC) of each test subject, respectively. The sub-columns show the coverage of different methods, and the last row shows the average data. As stated in Section 5.1, for RQ(1), the comparison of coverage is conducted under a restriction on the number of the generated test cases. Such restriction is shown in column COUNT of Table 2.

From Table 2, we can see that for 11 of 16 subjects, the statement coverage rates of CRAND are no more than that of CC method. On average, the statement and branch coverage of the CRAND are 32.07% and 39.49%, respectively; both are less than those of our CC, DCC and NFCC methods (CC: 46.45% and 57.91%, DCC: 46.21% and 58.38%, and NFCC: 32.12% and 41.52%). CC can achieve about 15% more coverage compared with CRAND, which is not guided by constraint-level coverage criteria. The orders between different methods in average level statement coverage rates are CC > DCC > NFCC > CRAND.

In the experiment, NFCC method does not achieve as high coverage as DCC and CC. This is because for RQ(1), the number of test cases is limited. NFCC usually divides the constraints more finely, resulting in more test cases. We selected a small part of them. In that case, the achieved coverage is not necessarily higher than the other two methods. DCC method generates similar kinds and number of test data as CC. Therefore, their coverage rates can be close. The experimental results also indicate that considering the coverage of decision in addition to the coverage of atomic conditions does not make many difference for the coverage at source-code level.

- **Research Question (2)**

The coverage data in Table 2 and

Table 3 show that KLEE gets higher statement and branch coverage than the three methods proposed (KLEE: 81.70% and 85.06%, CC: 46.45% and 57.91%, NFCC: 50.04% and 60% without test case number imitation). For most of the subjects, the gaps in the statement coverage between KLEE and CC is no more than 30% of KLEE's coverage rates.

In fact, the methods proposed in this paper are black-box methods based on specification constraints, while the KLEE methods are finely tuned white-box methods. It is not strange that our methods achieves lower coverage than the KLEE methods. The advantages of our methods are that they are more efficient, do not depend on software implementations, and have fewer limitations compared to the heavy weight symbolic execution approaches. Besides, in software testing practice, it is often suggested that one should first design test cases according software specifications instead of directly generating test cases toward the coverage of source code. Under such suggestions, our approaches still can be valuable choices.

- **Research Question (3)**

For RQ(3), we generate test data freely without a restriction on the number of the generated test cases. The results are shown in

Table 3, where columns CC, DCC, and NFCC show the results of different methods, and the bottom row presents the average data. From the average data, we can see that when there is no restriction on the number of generated test cases; the orders between different methods in the achieved statement coverage rates are: NFCC > DCC > CC. For the number of generated test inputs, the ones of DCC are very close to that of CC. NFCC generates much more test inputs than CC and

DCC. Its test data sets are mostly larger than 500. However, because solving DNF/CNF sub-clauses often leads to duplicated inputs, for some subjects, the numbers of test inputs generated by NFCC are still small.

Table 2. Results of different methods under test case number restriction

Subject	Statement Coverage (SC)					Branch Coverage (BC)					TIME(s)					COUNT
	KLEE	*CRAND*	*CC*	*DCC*	*NFCC*	*KLEE*	*CRAND*	*CC*	*DCC*	*NFCC*	*KLEE*	*CRAND*	*CC*	*DCC*	*NFCC*	
base64	91.43	38.1	54.29	38.1	44.76	93.33	42.22	62.22	42.22	42.22	3605.18	7.49	38.83	40.95	479.58	70
chcon	68.21	55.38	53.85	54.36	15.38	63.89	58.33	58.33	58.33	26.39	3608.58	10.2	7.431	10.90	654.35	71
chgrp	86.67	42.22	42.22	42.22	47.78	92.16	80.39	80.39	80.39	84.31	3609.32	4.49	40.18	61.17	294.60	59
comm	85.71	39.8	38.78	62.24	27.55	94.29	45.71	45.71	85.71	31.43	3605.00	5.88	7.261	7.37	386.45	45
cut	85.47	6.08	61.49	61.49	6.08	88.71	6.45	64.52	64.52	6.45	3610.74	7.87	182.71	189.30	1084.34	88
dircolors	91.39	11.05	11.05	11.05	11.05	100.00	10.00	10.00	10.00	10.00	3569.54	4.72	125.32	114.16	597.66	56
expand	41.42	22.52	66.89	58.28	47.68	40.88	20.41	87.76	75.51	65.31	3614.93	25.28	410.12	393.12	613.42	61
expr	97.35	30.47	29.59	29.88	24.56	97.26	37.02	35.91	35.91	34.81	6.17	1.12	3.43	3.29	238.24	34
fold	95.26	21.24	73.45	64.6	43.36	98.75	31.51	94.52	91.78	57.53	3606.15	12.13	75.82	70.20	1401.60	67
mknod	91.44	42.68	40.24	39.02	35.37	95.28	50.88	50.88	50.88	47.37	3605.46	5.22	7.55	7.53	886.79	67
paste	71.97	20.86	37.43	37.43	65.24	82.11	27.56	46.46	46.46	73.23	3612.28	10.94	14.59	11.85	663.49	69
pathchk	93.39	37.12	50.00	56.82	41.67	91.59	58.95	69.47	73.68	68.22	3562.32	6.76	74.64	47.58	100.67	48
printf	65.85	33.46	61.48	61.48	36.58	68.42	37.85	68.22	68.22	44.39	3605.15	10.92	16.27	19.94	608.25	156
split	70.97	34.1	58.99	58.99	14.29	78.57	38.57	72.86	72.86	21.43	33.72	5.56	140.66	163.10	359.64	50
sum	94.74	55.79	36.84	36.84	36.84	92.45	66.04	50.94	50.94	39.62	4.46	1.3	3.67	3.68	71.68	26
tsort	75.86	22.17	26.6	26.6	15.76	83.33	20	28.33	26.67	11.67	3601.51	6.34	24.97	31.46	1037.59	48
AVG	81.70	32.07	46.45	46.21	32.12	85.06	39.49	57.91	58.38	41.52	2928.78	7.89	73.34	73.48	592.40	63

Table 3. Coverage rates achieved by different methods without test case number restriction

Subject	Statement Coverage (SC)			Branch Coverage (BC)			TIME(s)			COUNT		
	CC	*DCC*	*NFCC*	*CC*	*DCC*	*NFCC*	*CC*	*DCC*	*NFCC*	*CC*	*DCC*	*NFCC*
base64	38.1	38.1	56.19	42.22	42.22	57.78	18.44	18.33	3600.24	71	66	848
chcon	49.23	49.23	54.87	58.33	58.33	58.33	1.66	1.67	3600.50	33	33	1519
chgrp	42.22	42.22	45.56	80.39	80.39	84.31	84.99	84.38	3840.49	145	147	622
comm	62.24	62.24	65.31	85.71	85.71	85.71	1.88	1.78	1443.83	39	39	782
cut	61.49	63.85	35.47	64.52	66.94	37.9	121.11	126.59	3600.62	63	61	356
dircolors	11.05	11.05	17.89	10.00	10.00	18.75	86.61	97.32	2334.33	60	61	1363
expand	70.86	70.86	67.55	93.88	93.88	91.84	253.61	261.59	3600.16	38	38	1297
expr	29.88	29.88	31.36	35.91	35.91	38.12	1.35	1.33	3600.24	36	36	396
fold	70.8	77.88	64.6	94.52	94.52	80.82	121.65	129.31	3600.93	189	184	3714
mknod	37.8	37.8	58.54	50.88	50.88	50.88	2.17	2.15	3600.91	55	55	2566
paste	43.32	43.32	73.8	51.18	51.18	84.25	11.59	11.67	3600.11	127	129	5578
pathchk	56.06	56.06	56.82	80.00	80.00	69.47	102.26	103.38	3600.39	122	122	1953
printf	51.75	53.7	43.58	57.94	59.81	53.27	3.30	3.37	3600.18	57	59	2385
split	58.99	59.45	42.86	72.86	71.43	57.14	126.65	127.14	3600.41	56	55	235
sum	38.95	38.95	63.16	50.94	50.94	69.81	1.30	1.30	1399.26	24	24	1887
tsort	27.09	27.09	23.15	28.33	28.33	21.67	19.51	19.21	2048.40	71	71	668
AVG	46.86	47.61	50.04	59.85	60.03	60.00	59.88	61.91	3166.94	74	74	1636

Figure 3 also shows the number of subjects on which each method in CC, DCC, and NFCC achieves the best statement coverage, respectively. From Figure 3, we can see that NFCC achieves the best coverage on 10 test subjects, while the numbers for DCC and CC are 6 and 2. Here, the total number is larger than the number of subjects 16 since DCC and CC sometimes get the same coverage rates. Compared with DCC and CC, NFCC also gets better coverage for more subjects.

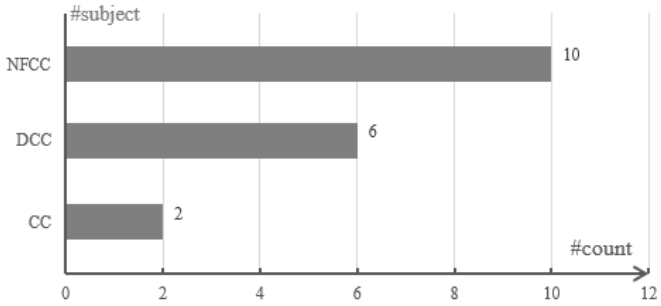

Figure 3. The number of subjects on which each method achieves the best coverage.

- **Research Question (4)**

From the TIME column of Table 2, we can see that when there is a restriction on the number of the generated test cases. On average, KLEE consumes the longest time (2928.78s), CC and DCC consume about 70 seconds, while CRAND consumes the shortest time (7.89s). On average, the orders between different methods on test data generation time is KLEE > NFCC > DCC > CC > CRAND. CC and DCC take more generation time to find the same number of test inputs compared to CRAND because we do not exclude the previous constraint solving solutions when solving the next constraint. (Two constraints sharing the same solution means that we can use one test input to cover two conditions. Hence, it is unnecessary to generate different test inputs for different constraints.) Different constraints may have the same solution; thus, we need more time solving more constraints to find different test inputs.

Column TIME in

Table 3 shows the test data generation time of the methods guided by constraint-level coverage criteria when there is no restriction on the number of the generated test cases. From the table, we can see that for all subjects, NFCC takes the largest execution time (often reaches its time upper bound), and the time consumed by CC and DCC is small. On average, DCC takes 61.91 seconds, and CC takes 59.88 seconds. CC and DCC take smaller time than that when there is a restriction on the generated test case number, because in previous case we may need to run CC and DCC more than one round to reach the common test case number. NFCC is slower because it spends more time on dividing constraints into normal form sub-clauses, and on solving these sub-constraints.

5.3. Discussion

From the experimental results, we can see that decomposing constraints can improve the test coverage by about 15%, compared with an approach that randomly solving constraints without such decompositions. The coverage-guided generation does not introduce too much cost, especially for criteria CC and DCC.

For the constraint-level coverage criteria defined according to different constraint decomposition rules, NFCC can lead to the best source-code level coverage; however, it is also time-consuming and the improvement in coverage rates achieved by NFCC compared with CC and DCC is small. Therefore, CC and DCC are more valuable choices for practical use. They can get relatively high coverage with fewer test inputs. Both achieved coverage rates and the consumed time of CC and DCC are very close. In theory, DCC can guarantee the true and false outcomes of the whole constraint are covered, so DCC will be a better choice than CC.

The threats to the validity of the experimental study include several aspects. First, the interface constraints used in this experiment are simulated by merging constraints derived from program paths. Such constraints may not be representative enough for the real specification constraints, which can possibly affect the effectiveness of experimental study. Second, our experiment subjects are mainly command line argument processing programs focusing more on strings. The form of constraints on such subjects are limited. In the future, we plan to conduct experiments on more kinds of subjects and more real specification constraints to further strengthen the experiment validation.

6. Related Work

Different from the test data generation approaches, which focus on creating a sequence of inputs according to finite state machines or labeled transition systems [4], our approach is a model-based one [18], which generates test inputs passed into an application at a single time according to interface constraints. In the generation of test data from constraints, linear programming, SAT and SMT solving, constraint programming are typical techniques to derive test data satisfying given constraints [14].

Besides, random and boundary values are also used in constraint-based test data generation when the forms of constraints are limited. Recently, researches also use search-based techniques to solve complex constraints [2, 3]. Of these techniques, SMT solving have been proven powerful for practical use; we choose to use it as a basis for test data generation.

To generate high coverage test data from constraints, some work [7, 16, 19] decomposes the constraint controlled input space into partitions and then uses random or boundary values in the partitions to create test inputs. These approaches do not rely on constraint solving techniques, but they only suite for a limited number of constraint formula forms. The work in [1] and [13] supports decomposing a constraint like $x \leq y$ into $x < y$ and $x = y$ for the sake of getting more fine-grained input space partitions. However, [1] does not discuss how the test data are generated, and [13] focuses on generating certain models to test model transformation in model-driven engineering (MDE), instead of generating normal test data. In [1, 7, 8, 19], disjunction normal form (DNF) of a constraint formula is also considered during the partitioning of input space. None of these approaches have defined a clear hierarchy of coverage criteria with respect to a constraint formula to guide different grains of test data generation. Like our work, Weibleder and Schlingloff [20] also adopt some logic coverage criteria in white-box testing, e.g., condition coverage and decision coverage, to the test data generation under OCL (Object Constraint Language) specifications. However, they do not consider the coverage of CNF and DNF in their coverage criteria.

A noticeable limitation of the previous constraint decomposition research is that no research has experimentally validated the effectiveness of the proposed decomposition rules in test data generation. When solving constraints with a SMT solver, the solver itself can have some randomness in test data generation. With such randomness, values in different partitions of the constraint controlled input space may have chance to be selected even without any constraint decomposition effort. Whether the constraint decompositions can really lead to better coverage and how much improvement can be achieved is unclear. This paper presents an experimental study to help answer these questions.

7. Conclusions

When generating test data from constraints, performing generation directly from the whole constraint cannot guarantees the coverage of the result test set. To this end, this paper defines a hierarchy of constraint-level coverage criteria for test data generation. Guided by these criteria, we decompose the original constraints into smaller ones representing fine-grained input space partitions and use SMT solver to obtain the test data set. With such approach, the generated data becomes more diverse. The experimental results show that constraint-level criteria effectively leads to better statement and branch coverage rates with the same number of test cases (about 15% improvement) and the additional cost introduced for test generation due to criteria CC and DCC is low.

In the future, we plan to bring more white-box coverage criteria, e.g., MC/DC, into constraint-level and study more constraint decomposition strategies for test data generation. Besides, we will also find more experiment subjects and more constraints to further investigate the effectiveness of the constraint decomposition approach.

Acknowledgements

This work is supported by the China Defense Industrial Technology Development Program (Grant No. JCKY2016206B001 and JCKY2014206C002) and the Science and Technology Planning Project of Jiangsu Province (BY2016003-02).

References

1. L. v. Aertryck and T. Jensen, “UML-Casting: Test Synthesis from UML Models Using Constraint Resolution,” In *Proceedings of the Approches Formelles dans l'Assistance au Développement de Logiciels* (AFADL), 2003.
2. S. Ali, M. Z. Iqbal, A. Arcuri, and L. C. Briand, “Generating Test Data from OCL Constraints with Search Techniques,” *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1376-1402, 2013
3. S. Ali, M. Z. Iqbal, M. Khalid, and A. Arcuri, “Improving the Performance of OCL Constraint Solving with Novel Heuristics for Logical Operations: a Search-Based Approach,” *Empirical Software Engineering*, vol. 21, no.6, pp.2459-2502, 2016
4. S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, “An Orchestrated Survey of Methodologies for Automated Software Test Case Generation,” *Journal of Systems and Software*, vol. 86, no. 8, pp.1978-2001, 2013
5. A. Arcuri and L. Briand, “Adaptive Random Testing: An Illusion of Effectiveness?” In *Proceedings of the International Symposium on Software Testing and Analysis* (ISSTA), pp.265-275, Toronto, Ontario, Canada, July 2011
6. I. Banerjee, B. Nguyen, V. Garousi, and A. Memon, “Graphical User Interface (GUI) testing: Systematic Mapping and Repository,” *Information and Software Technology*, vol. 55, no.10, pp.1679-1694, 2013
7. M. Benattou, J.-M. Bruel, and N. Hameurlain, “Generating Test Data from OCL Specification,” In *Proceeding of ECOOP Workshop on Integration and Transformation of UML Models*, 2002

8. A. D. Brucker and B. Wolff, "On Theorem Prover-Based Testing," *Formal Aspects of Computing*, vol. 25, no. 5, pp 683–721, 2013
9. C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), pp. 209-224, San Diego, California, USA, 2008
10. L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS), pp. 337-340, Budapest, Hungary, March 2008
11. P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pp. 213-223, Chicago, IL, USA, June 2005
12. P. Godefroid, H. Peleg, and R. Singh, "Learn&Fuzz: Machine Learning for Input Fuzzing," In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (ASE), pp. 50-59, Urbana-Champaign, IL, USA, 2007
13. C. A. González and J. Cabot, "Test Data Generation for Model Transformations Combining Partition and Constraint Analysis," In *Proceedings of the International Conference on Theory and Practice of Model Transformations*, pp. 25-41, York, UK, July 2014
14. A. Gotlieb. "Chapter Two: Constraint-Based Testing: An Emerging Trend in Software Testing," *Advances in Computers*, vol. 99, pp. 67-101, 2015
15. F. M. Kifetew, R. Tiella, and P. Tonella, "Generating Valid Grammar-Based Test Inputs by Means of Genetic Programming and Annotated Grammars," *Empirical Software Engineering*, vol.22, no.2, pp 928–961, 2017
16. H. Mei and L. Zhang, "A Framework for Testing Web Services and Its Supporting Tool," In *Proceedings of the IEEE International Workshop on Service-Oriented System Engineering* (SOSE), pp. 207-214, Beijing, China, October, 2005.
17. K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," In *Proceedings of the International Symposium on the Foundations of Software Engineering* (FSE), pp. 263-272 , Lisbon, Portugal, September 2005
18. M. Utting, A. Pretschner, and B. Legeard, "A Taxonomy of Model-Based Testing Approaches," *Software Testing, Verification and Reliability*, vol.22, no.5, pp.297-312, 2012
19. S. Weiβleder and B.-H. Schlingloff, "Deriving Input Partitions from UML Models for Automatic Test Generation," In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems* (MODELS), pp. 151-163, Nashville, TN, USA, September 2007
20. S. Weiβleder andB.-H. Schlingloff, "Quality of Automatically Generated Test Cases Based on OCL Expressions," In *Proceedings of the International Conference on Software Testing, Verification, and Validation* (ICST), pp. 517-520, Lillehammer, Norway, June 2008
21. Coreutils - GNU core utilities, 2018, http://www.gnu.org/software/coreutils/
22. gcov—a Test Coverage Program, 2018, https://gcc.gnu.org/onlinedocs/gcc/Gcov.html