Boosting Search Based Testing by using Constraint Based Testing

Abdelilah Sakti, Yann-Gaël Guéhéneuc, and Gilles Pesant

Department of Computer and Software Engineering
École Polytechnique de Montréal, Québec, Canada
{abdelilah.sakti,yann-gael.gueheneuc,gilles.pesant}@polymtl.ca

Abstract. Search-Based Testing (SBT) uses an evolutionary algorithm to generate test cases. Traditionally, a random selection is used to generate an initial population and also, less often, during the evolution process. Such selection is likely to achieve lower coverage than a guided selection. We define two novel concepts: (1) a constrained population generator (CPG) that generates a diversified initial population that satisfies some test target constraints; and (2) a constrained evolution operator (CEO) that evolves test candidates according to some constraints of the test target. Either the CPG or CEO may substantially increase the chance of reaching adequate coverage with less effort. In this paper, we propose an approach that models a relaxed version of the unit under test as a constraint satisfaction problem. Based on this model and the test target, a CPG generates an initial population. Then, an evolutionary algorithm uses a CEO and this population to generate test input leading to the test target being covered. Our approach combines constraint-based testing (CBT) and SBT and overcomes the limitations associated with each of them. Using eToc, an open-source SBT tool, we implement a prototype of this approach. We present the empirical results of applying both CPG or CEO on three open-source programs and show that CPG or CEO improve SBT performance in terms of branch coverage by 11% while reducing computation time.

Keywords: Search Based Testing, Constraint Based Testing, Initial Population Generator, Evolution Operator.

1 Introduction

Proving that some software system corresponds to its specification or exposing hidden errors in its implementation is a time consuming and tedious process, accounting for 50% of the total software cost [13]. Test case generation is one of the most expensive parts of the software testing phase. Therefore, automating testing can significantly reduce software cost, development time, and time to market [6]. Constraint Based Testing (CBT) and Search Based Testing (SBT) have become the dominant approaches to test case generation, because they achieve high code coverage. Over the last decade, theses two approaches have been extensively explored [3, 5, 9, 10, 14, 18, 19].

The main advantages of the CBT approach are its precision in test data generation and its ability to prove that some paths are unreachable. The main disadvantage of CBT is its inability to manage the dynamic aspects of a unit under test (dynamic structures, native function calls, and communication with the external environment) [19]. Using CBT, it is not always possible to generate an exact test input due to source code complexity or unavailability.

In contrast, SBT approaches handle any sort of programs but these approaches depend on the search space size, the diversity of their initial populations, and the effectiveness of their fitness functions. It is inefficient to use evolutionary testing in a large search space without a diversified population or without sufficient guidance: for example, in a program that contains conditional statements on boolean variables (flags), evolutionary testing may not have the necessary information to guide its search [11]. Even though SBT is largely used in industry, it still suffers from many problems [11]. Those problems can be dealt with by using CBT. Recently, a strong combination of constraint programming and of an evolutionary algorithm has shown great promise to solve optimization problems [8]. Such combination leads us to believe that combining SBT and CBT is interesting for an efficient test data generation.

In this paper, we propose a hybrid approach, *CSBT*, that combines *CBT* and *SBT* to generate test data. We define a novel CBT framework to replace any random generation in the SBT approach. To generate test input candidates for SBT, the CBT models and solves a relaxed version of the unit under test (UUT). Then, the SBT framework takes these input candidates and generates the actual test input. We implemented a prototype of this hybrid approach and applied it to generate test input data leading to branch coverage on a set of programs. We report the comparison of CBT, SBT, and our novel approach CSBT and show that the CSBT technique outperforms the others. These empirical results will show that CSBT reduces the effort needed to reach a given test target and that it achieves higher branch coverage than the test input generated by each approach alone.

The contributions of this paper are: a novel approach to combine both *search based* and *constraint based* techniques to generate test input data; a framework to model a relaxation of a UUT as a constraint satisfaction problem; an empirical comparison of CBT, SBT, and CSBT on some programs.

The remainder of the paper is organized as follows: Section 2 presents a brief overview of CBT and SBT. Section 3 introduces the problem and the approach by using a motivating example. Section 4 describes our testing approach. Section 5 describes how we implemented a prototype of our approach by extending the SBT tool eToc [20]. Section 6 presents the empirical study used to evaluate the CSBT and the analysis of the results. Section 7 summarizes related work. Section 8 concludes with some future work.

2 Background

Constraint-based testing is applied in two different ways: static [2] (symbolic execution) and dynamic [4] (Dynamic Symbolic Execution DSE).

Static CBT. We distinguish between two static CBT approaches: *path-oriented* [2] and *goal-oriented* [5]. The first finds test inputs of a given execution path. *Symbolic execution* (SE) is a well known path-oriented technique that was introduced by Clarke [2] in the 1970s. SE consists of statically selecting an execution path from the control-flow graph (CFG) and then executing it symbolically and creating a path constraint over the program's input variables. A solution to this path constraint is a test data that will drive the execution down the selected path. The second approach finds test input that reaches a given statement (test target). Generally, this approach translates a whole program into a constraint programming problem. The given test target is translated into a constraint. Solving the conjunction of this constraint and the generated constraint programming problem yields a test data that will reach the test target.

Dynamic CBT. In the literature, the dynamic CBT technique most used is *dynamic symbolic execution* (DSE) [4, 17]. DSE combines symbolic and concrete execution. It consists of exploring execution paths at runtime: First, it executes an instrumented version of a UUT; second it gets the executed path condition and some concrete values; and then it derives a new path condition; it uses concrete values to simplify complex (unsupported) expressions. Solving this new path condition generates test inputs to explore a new path. This procedure is repeated until the test target is covered (e.g., all-paths, all-branches, all-statements) or some condition limit is met.

Search-Based Testing. SBT was introduced by Miller and Spooner [12] in the 1970s. To generate test inputs SBT uses an evolutionary algorithm that is guided by an objective function or fitness function. The fitness function is defined according to a desirable test target. The commonly used fitness functions are branch-distance and approach-level [11]. To generate test input, SBT starts by generating a random set of test input candidates (initial population). For each test input candidate, an instrumented version of the UUT is executed and its fitness is computed. Based on the fitness ranking, the evolutionary algorithm evolves the current population to generate a new one. It continues evolving populations until the test target is achieved or a stopping criterion is reached.

3 Motivating Example

We use the program in Fig. 1 as a running example. It considers the problem of generating a test input to reach Targets 1, 2, and 3. The three targets reflect different problems of test input generation.

- Target 1 is easy to reach;
- Target 2 is unreachable because $!(x \leq 0 || y \leq 0) \& !(x < y/2) \& (y > 3 \times x)$ is unsatisfiable;
- Target 3 is hard to reach because it contains nested predicates and involves the native function call, fun, that returns x^2/y.

CBT can generate test inputs for Target 1 and prove that Target 2 is unreachable. However, if we suppose that the function fun cannot be handled by a particular constraint solver, a static CBT approach cannot generate test inputs for Target 3. A dynamic CBT can also fail to derive test inputs for Target 3 in a reasonable amount of time.

```
int foo(int X, int Y){
  if(X<=0 || Y<=0)
    return 0;
  int Z;
  if ((X < Y/2)|| (Y==0))
    Z= 1; //Target 1
  else if (Y>3*X)
    Z=2; //Target 2
  else{
    Z = fun(X,Y);
    if ((Z >8) && (Y==10))
      if(Z==Y)
        Z=3;//Target 3
  }
  return Z;
}
```

Fig. 1. running example

SBT can derive test inputs for Target 1 but it takes more time than CBT. Therefore, if the search space is large, SBT may take a very long time before generating a test input for Target 3 (it needs $x = 10$ and $y = 10$). For Target 2, SBT may search forever without proving its unreachability.

Target 3 is problematic for both approaches: it is a nested branch predicate [11] for SBT and it contains an unsupported function for CBT. However a hybrid approach can take advantage of both to generate test input for Targets 1 and 3, and it may prove the unreachability of Target 2. We confirmed this fact by generating a test input to reach Target 3 using all three approaches: SBT (eToc [20] and a hill climbing implementation), CBT (CP-SST [15], a goal-oriented static CBT approach), and a combination of these approaches. We concluded that eToc and Hill Climbing were unable to generate test inputs to reach Target 3 after 45000 fitness calculations in domain $[-20000, 20000]$. As well, CP-SST was unable to reach Target 3. However, a hybridization eToc+CP-SST (resp. HC+CP-SST) was able to generate test input just after 200 (resp. 600) fitness calculations.

The key idea of our hybridization is to proceed in two phases. First, generating a relaxed version of foo by replacing the function fun with an uninitialized variable typed as foo's return value. Then CBT uses the relaxed version to generate pseudo test inputs. In a second phase, SBT uses those pseudo test inputs as candidates to generate the actual test inputs.

Now, consider for example the program in Fig.1. In order to reach Target 3, the conjunction of the negation of the first three conditional statements and the last two conditional statements must be fulfilled. For the last two conditions, z depends on $fun(x, y)$'s return value. Using CP-SST [15] the path condition can be written as follow:

$not((x_0 \leq 0)||(y_0 \leq 0)) \wedge not((x_0 < y_0/2)||(y_0 = 0)) \wedge not(y_0 > 3 \times x_0) \wedge z_3 = fun(x_0, y_0) \wedge ((z_3 > 8) \wedge (y_0 = 10)) \wedge (z_3 = y_0)$.

The relaxed version is obtained by ignoring the constraint $z_3 = fun(x_0, y_0)$. Then, the path condition becomes $not((x_0 \leq 0)||(y_0 \leq 0)) \wedge not((x_0 < y_0/2)||(y_0 = 0)) \wedge not(y_0 > 3 \times x_0) \wedge ((z_3 > 8) \wedge (y_0 = 10)) \wedge (z_3 = y)$. As the relaxed path condition is less restrictive than the actual one, generated pseudo test inputs won't necessarily trigger Target 3, but they satisfy a big part of the path condi-

tion. By solving the relaxed constraint, we obtain solutions of the form ($x_0 \geq 5$, $y_0 = 10$). Therefore, the search space size has been reduced. Using those pseudo test inputs as an initial population for an evolutionary algorithm can reduce significantly the effort needed to generate test data.

In this example, only one path led to Target 3. If there are many paths that can reach the test target, generating the test data that allows covering different paths or different branches may assure a diversified population for SBT.

4 CSBT: Constrained Search Based Testing

The assumption underlying the research work presented in this paper supposes that using a diversified initial population that partly satisfies the predicates leading to the test target can reduce significantly the effort required to reach this test target. Our approach is called Constrained Search Based Testing (CSBT) because it starts, in a first phase, by using CBT to generate an initial population and then, in a second phase, it uses SBT to generate test inputs. We propose to replace the random generation of an initial population used by SBT with a set of pseudo test input generated using a CBT approach.

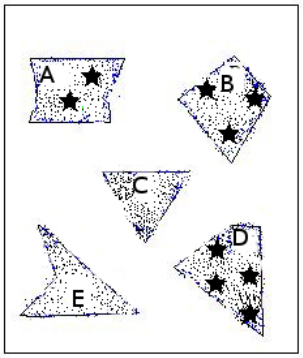

Fig. 2. Inputs search space.

Fig. 2 presents the whole inputs search space of a program P: the parts A, B, C, D, and E are the CBT solutions space of a relaxed version of P which are called *pseudo test inputs*, while stars are actual test inputs. This example shows that a random test input candidate is likely to be too far from an actual test inputs compared to some pseudo test inputs. The parts C and E don't contain any test input, so a pseudo input from these two parts may also be far from an actual test input. A population that takes its candidates from different parts is likely to be near of a test input, we call it a *diversified population*. We can offer an acceptable level of diversity by generating a pseudo test input for every consistent branch with the test target (All-branch), and a high level of diversity by generating a pseudo test input for every path leads to the test target (All-path).

4.1 Unit Under Test Relaxation

To avoid traditional CBT problems, we introduce a preliminary phase called program relaxation. We propose to apply CBT on a relaxed version of the UUT. A relaxed UUT version is a simplified version, of the original one, that contains only expressions supported by the constraint solver: the expressions that generate constraints whose consistency can be checked by the constraint solvers are kept in the relaxed version, while all the other expression are relaxed or ignored, e.g., for an integer solver, expressions that can generate constraints over string or float are ignored, unsupported operators (expressions) are relaxed, and a native function call that returns an integer is relaxed.

```
intStr(int X,int Y,
 String S1, String S2){
  int y= X<<Y;
  int x=y+X/Y;
  String s=S1+S2;
  if((s.equals("OK")
    && x>0)
    && s.length()>x)
     return 1; //Target
  return 0;
}
```

Fig. 3. intStr function

```
intStr(int X,int Y){
  int R1,R2;
  int y= R1;
  int x=y+X/Y;
  if(
    x>0)
    && R2>x)
     return 1; //Tar
  return 0;
}
```

Fig. 4. Relaxed version for an integer solver

```
intStr(String S1,
      String S2){

  String s=S1+S2;
  if((s.equals("OK")
    )
    )
     return 1; //Tar
  return 0;
}
```

Fig. 5. Relaxed version for a string solver

A relaxed version is obtained by applying the following rules:

- Any unsupported expression (function call, operator) is relaxed: the expression is replaced by a new variable. Fig. 4 shows a relaxed version of *intStr*, which is shown in Fig.3, for an integer solver. We suppose that the solver cannot handle the shift operator, so the expression that uses this operator has been replaced by a new variable $R1$. The variable $R1$ is not initialized. Therefore, when we will translate the relaxed version into a CSP, the CSP variable $R1$ can take any integer value.
- Any statement over unsupported data type is ignored. In Fig. 4, The relaxed version of *intStr* ignores the statement $String\ s = S1{+}S2$; and the condition $s.equals(\text{"}OK\text{"})$ because those two expressions are over strings.
- For each data type that needs a different solver a new relaxed version is created. The function *intStr* needs integer and string type as inputs. If we have an available solver for string, then we generate another relaxed version over string. Fig. 5 shows a relaxed version of *intStr* for a string solver.
- For loops, CBT cannot model an unlimited number of iterations. In general a constant k-path (equal 1, 2, or 3) is used to limit the number of loop iterations. We can model a loop in two different ways: first, we force a loop to stop at most after k-path iterations, in this case some feasible paths may become infeasible; second, we don't force a loop to stop and we model just k-path iterations, after which the value of a variable assigned inside a loop is unknown. We use the second case and we relax any variable assigned inside a loop just after this loop. The variable is assigned a new uninitialized variable.

4.2 Collaboration between CBT and SBT

Every test input generated is a result of a collaborative task between CBT and SBT: CBT generates a pseudo test input, and then SBT generates an actual test input. Our main contribution here is to define the information exchanged and connection points that can make CBT useful for SBT. SBT needs new test input candidates at three different points:

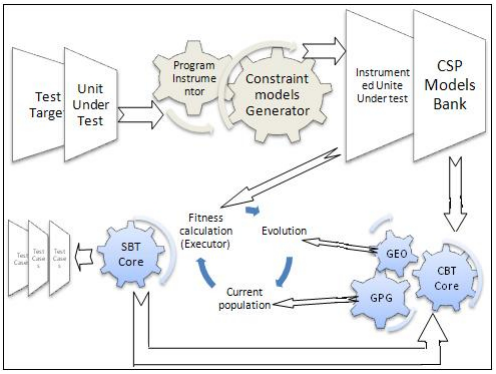

Fig. 6. Implementation overview

1. During the generation of the initial population;
2. When it restarts, i.e., when it reaches the attempt limit of evolving;
3. During its evolving procedure.

For the first and the second points, a CBT can generate the whole or a part of the population. If CBT is unable to generate the required number (population size) of candidates the usual generation procedure (random) can be called to complete the population. We called this technique *Constrained Population Generator*(CPG). For the third point, CBT can participate to guide the population evolution by discarding candidates that break the relaxed model and only allowing candidates that satisfy the relaxed model, we called this technique *Constrained Evolution Operator* (CEO). But frequent calls to CBT seem to be too expensive in practice, this can weaken the main advantage of evolutionary algorithms which is their speed. Therefore, we propose to limit the number of CBT calls during the population evolving procedure.

5 Implementation

We have implemented an integer version (with an integer solver) of our approach CSBT by using a new implementation of our CBT [15] and by extending eToc [20]. Fig. 6 shows the overview of the implementation, built over six components: Program instrumentor, Constraint models Generator, CBT Core, a CEO, a CPG, and SBT Core. The main components are CBT Core and SBT Core. These two components communicate via shared target and population. They identify sub-targets (branches) in the same way by using the Program Instrumentor component as a common preprocessing phase.

In the architecture, the SBT Core acts as the master and CBT Core plays the role of slave. When SBT Core needs test input candidates, it requests CBT Core by sending its current test target. To answer the request, CBT Core uses CEO or CPG and replies by sending a pseudo test input, proving the inaccessibility of the target, or by simply saying that the execution time limit is reached. Then SBT evolves its new population. This process is repeated until the test target is covered or some condition limit is met.

5.1 SBT Core

We use eToc that implements a genetic algorithm to generate test cases for object oriented testing. eToc begins with instrumenting the UUT to identify the test target and to keep trace of the program execution. eToc generates a random initial population whose individuals are a sequence of methods. eToc follows the GA principle: to evolve its population it uses a crossover operator and four mutation operators. The main mutation operator that interests us mutates method arguments. To adapt eToc to our requirements, we modified the argument values generator from a random generator to a CPG. Thus, we modified one mutation operator to make it a CEO.

5.2 CBT Core

Constraint Models Generator. This component takes a UUT as input optionally instrumented with eToc program instrumentor. For each Java class method a specific structure of control flow graph is generated. In addition, using the Choco[1] language a constraint model for a relaxed version of this method is generated. All generated models constitute a CSP Models Bank that is used by CBT Core during test input generation.

CEO. CEO can be implemented as crossover operator, mutation operator, or neighbourhood generator. In this work, we implemented CEO as a mutation operator. With a predefined likelihood the genetic algorithm calls CEO by sending the methods under test, the current test target, current parameters values, and the parameter to change. CEO uses this information to choose the adequate CSP model and to fix the test target and parameters except those required to change. After that the model is solved and a new value is assigned to the requested parameter. If the solver does not return a solution then all parameters are assigned arbitrary values.

CPG. When the eToc genetic algorithm starts or restarts, randomly it generates its chromosomes (methods sequences), and then it calls CPG to generate parameters values. For each function in the population, this generator check a queue of pseudo test inputs, if there is a pseudo test input for this method, then the method's parameters are assigned and this pseudo input test is deleted. Otherwise the CPG generates a pseudo test input for each test target not yet covered in this method. One of these pseudo test inputs is immediately assigned to this method's parameters, the rest are pushed into the queue. During solving test target if a target is proved infeasible, then the generator drops it from the set of target to cover.

[1] Choco is an open source java constraint programming library. urlhttp://www.emn.fr/z-info/choco-solver/pdf/choco-presentation.pdf

6 Empirical Study

The *goal* of our empirical study is to compare our proposed approach against previous work and identify the best variant among the two proposed techniques in Section 4.2 and a combination thereof. The *quality focus* is the performance of the CPG technique, the CEO technique, CPG+CEO, CBT, and SBT to generate test inputs set that covers all-branches. The *context* of our research includes three case studies: Integer, BitSet, and ArithmeticUtils were taken from the *Java* standard library and *Apache Commons project*[2]. BitSet was previously used in evaluating white-box test generation tools [20, 7]. These classes have around 1000 lines of java code. The SBT used (eToc) was not able to manage array structures and long type. We fixed the long type limitation by using *int* type instead, but we had to avoid testing methods over array structures. Therefore, we tested the whole *BitSet* and *ArithmticUtils*, but only a part from *Integer* because it contains array structure an input data for some methods. The number of all branches is 285: 38 in Integer, 145 in BitSet, and 102 in ArithmticUtils. These classes were chosen specifically because they contain function calls, loops, nested predicates, and complex arithmetic operations over integers.

6.1 Research Questions

This case study aims at answering the following three research questions:

RQ1: Can our approach CBST boost the SBT performance in terms of branch coverage and runtime? This question shows the applicability and the usefulness of our approach.

RQ2: When and at what order of magnitude is using our approach useful for SBT? This question shows the effectiveness of our approach.

RQ3: Which of the three proposed techniques is best suited to generate test inputs in an efficient way?

6.2 Parameters

As shown in the running example, during the empirical study we observed that the difference, in terms of fitness calculations, between CSBT and SBT is very large. We think that comparing approaches using this metric is unfair because CSBT uses CBT to reduce the number of fitness calculations. So to provide a fair comparison across the five approaches (CBT, SBT, CPG, CEO, CPG+CEO), we measure the cumulative branch coverage achieved by these approaches every 10 seconds for a sufficient period of *runtime* (300 s). This period was empirically determined as sufficient for all approaches. To reduce the random aspect in the observed values, we repeated each computation 10 times. We think that the default integer domain in eToc $[-100, 100]$ is unrealistically small so to get a meaningful empirical study, we chose to fix the domain for all the input variables to a larger domain $[-2 \times 10^4, 2 \times 10^4]$. We kept the rest of eToc's default parameters as is. We used identical parameters values for all techniques.

[2] The Apache Software Foundation. `http://commons.apache.org`

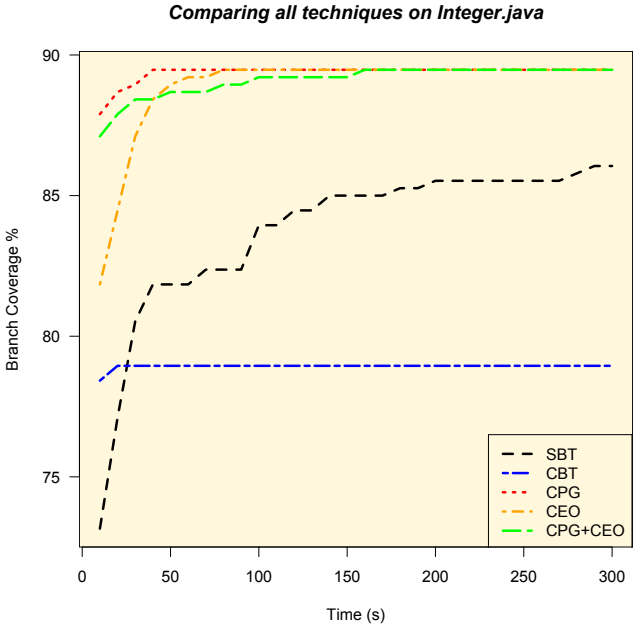

Fig. 7. Comparing all techniques on Integer.java

For CBT, the solver uses the minDom variable selection heuristic on the CSP variables that represent CFG nodes as a first goal and on CSP variables that represent parameters as a second goal. The variable value is selected randomly from the reduced domain. To make the study scalable, we restrict the solver runtime per test target to 500 ms. This avoids the solver hanging or consuming a large amount of time.

6.3 Analysis

Fig. 7 summarizes the branch coverage percentage in terms of execution time for the BitSet class. Overall, the CPG technique is the most effective, achieving 89.5% branch coverage in less than 40 s. Also, CEO and CPG+CEO reach 89.5% but after 70 s. eToc was unable to go beyond 85.78%, which is attained after 120 s. CBT performs badly, its best attained coverage is 78.94%. This figure shows that the proposed three techniques can improve the efficiency of eToc in terms of percentage coverage and execution time.

We analysed branch coverage percentage in terms of execution time on BitSet and on ArithmeticUtils. We got two graphics that resemble Fig. 7 with a slight difference: eToc starts better than the three proposed techniques for the first twenty seconds, but CPG rapidly makes up for lost time outperforming eToc just after 20 s. Also CEO and CPG+CEO outperformed eToc, but they needed a little more time especially on ArithmeticUtils. CBT performs always worse than even the weakest proposed techniques.

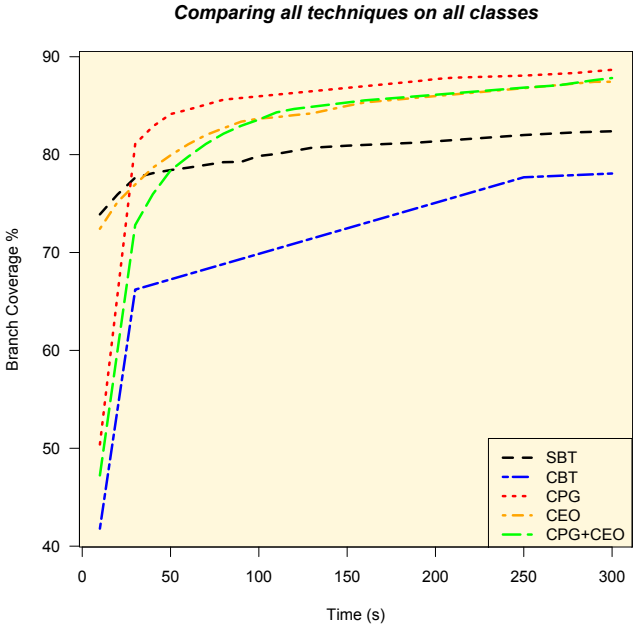

Fig. 8. Comparing all techniques on all the three java classes

Finally, Fig. 8 reflects the achieved results for all classes Integer, BitSet, and ArithmeticUtils. It confirms that the CPG technique is the most effective, and eToc starts better than the proposed techniques during the first twenty seconds.

6.4 Study Discussions

For the given classes, it is clear that CPG outperforms all techniques in terms of execution time and branch coverage. Also CEO and CPG+CEO perform better than eToc. Therefore, the proposed techniques boost SBT implemented in eToc. This result may be due to the kind of UUTs tried, which essentially use integer data types. More evidence is needed to verify whether the advantage of the proposed techniques represents a general trend. Yet, on the selected UUTs and the tool eToc **we answer RQ1 by claiming that the CBST techniques can boost the SBT**.

Over almost all graphics, we observed that eToc starts better than the proposed techniques. Therefore the latter are not useful for the first twenty seconds. This behaviour is due to the frequency of solver calls: at the start time all targets are not yet covered even the easiest one which can be covered randomly. Therefore, a combination that uses only SBT for a small lapse of time or until a certain number of fitness calculations and then uses CBST, may perform better. Also, we observed that after this time the CBST techniques quickly reach a high level of coverage. This is because at this moment CBST takes advantage of both

approaches: it includes branches which are covered either by SBT, CBT or by their combination. Thus, **we answer RQ2 by claiming that the CBST is more useful after the starting time**.

Even though CEO and CPG+CEO outperform SBT, we think that they don't perform as expected. On BitSet class these two techniques take a significant time before beating SBT. There are several factors that can influence the performance of the CEO. First, the frequency of solver calls is very high; it makes a call for every mutation. Second, in object oriented testing, a method under test does not necessarily contain the test target — this can make the mutation operator useless. Third, the mutation that we used imposes to fix part of the parameters which can make the CSP infeasible or hard to solve. These three factors are the main sources of CEO weakness. CPG+CEO is indirectly influenced by these factors by using CEO.

The CPG technique enhanced eToc by an average of 6.88% on all classes: 3.60% on Integer, 5.65% on BitSet, and 11.37% on ArithmeticUtils. These values did not take into account the proved infeasible branches: just on ArithmeticUtils CPG has proved 4 infeasible branches. We confirmed manually that these branches were infeasible because they use in their predicates some values out of the domain used. According to [7] the branches not covered by eToc are very difficult to cover. Therefore, a percentage ranging between 3.6% and 11.37% is a good performance for CPG. Thus, **we answer RQ3 by claiming that CPG is more useful to generate test inputs**.

6.5 Threats to Validity

The results showed the importance of using CSBT to generate test data, especially to improve the SBT performance in terms of runtime and coverage.

Yet, several threats potentially impact the validity of the results of our empirical study. A potential source of bias comes from the natural behaviour of any search based approach: the random aspect in the observed values. This can influence the internal validity of the experiments. In general, to overcome this problem, the approach should be applied multiple times on samples with a reasonable size. In our empirical study, each experiment took 300 seconds and was repeated 10 times. The coverage was traced every 10 s. The observed values become stable after 120 s. Each tested class contains around 1,000 LOCs and more than 100 branches. Therefore, experiments provided a reasonable size of data from which we can draw some conclusions, but more experiments are strongly recommended to confirm or refute such conclusions.

Another source of bias comes from the eToc genetic algorithm parameters: we didn't try different combinations of parameters values to show empirically that the approach is robust to eToc parameters. This can affect the internal validity of our empirical study.

Another potential source of bias includes the selection of the classes used in the empirical study, which could potentially affect its external validity. The BitSet class has been used to evaluate different structural testing approaches before [20, 7], thus it is a good candidate for comparing the different proposed

techniques. The two other classes were chosen because they feature integers and because they contain common problems in CBT and SBT (e.g, path that contains nested predicates and native function calls) and they represent widely used classes with non-trivial sizes.

7 Related Work

Several approaches have been proposed to use CBT or combine it with SBT in order to solve some input test generation problems, but they apply CBT on a complete version of the UUT [7, 9, 10]. All these approaches are limited by the size and the complexity of the UUT, and the fact that the input test generation problem is undecidable (reachability problem).

EVACON [7] was the first tool proposed that combines a CBT approach and a SBT approach. It bridges eToc [20] and jCute [16] to generate test data for classes. eToc is used to generate method sequences and jCute is used to generate test inputs. In this approach CBT and SBT work in a cooperative way: each of them has a separate task. CBT is dedicated to test input generation. In this way both approaches can complete each other but cannot solve common problems. In contrast, in our approach CBT and SBT work in a collaborative way: CBT starts the task by generating a pseudo test input and SBT complete this task by evolving pseudo test input and then generating the actual test input. A common problem is solved partially by using CBT, and then is completed by using SBT.

To solve constraints over floating point, the CBT tool PEX [19] has been extended by using FloPSy [9] which is a SBT approach that solves floating point constraints. FloPSy deals with a specific issue by using SBT to help CBT in solving constraints over floating point. In contrast our approach works in the opposite direction: it uses CBT to help SBT to improve its performance in terms of time and coverage.

Lakhotia et al. [1] propose a fitness function based on symbolic execution. The proposed fitness function analyses and approximates symbolic execution paths: some variables are approximated and any branches involving these variables are ignored. Then, the fitness value is computed based on the number of ignored conditions and the path distance (branch distance). This fitness function uses constraint programming to enhance SBT. Our approach is different because we propose an initial population generator and some evolution operators. Their fitness and our approach enhance SBT by using CBT, but in different ways. This fitness can be used in the same framework with our approach.

Recently, J. Malburg and G. Fraser [10] proposed a hybrid approach that combines GA and DSE on Java PathFinder. This approach uses GA to generate the test inputs; during the GA evolution the approach calls CBT to explore a new part in the program. The CBT is used as a mutation operator which uses SE to derive a constraint path, and then it uses DSE to solve this constraint. Therefore, the test data that are generated based on the combination are actually generated by using DSE. As explained in Section 2 DSE uses concrete values to simplify complex expressions. It is well known that these concrete values may make the path constraint infeasible. In general, these concrete values are randomly chosen,

in which case DSE falls in a random search. In contrast our CEO uses SBT to find the values of complex expressions. In addition in our approach all actual test inputs are generated by using SBT.

Our approach differs from previous work in that it provides a general framework for combining a SBT and a CBT to generate test inputs. It uses CBT to improve SBT, i.e., CBT is used to reduce the SBT search space. The initial population of SBT is generated using CBT solutions domain; and SBT evolution procedure is constrained to evolve the population in the CBT solutions space. Test inputs are generated using an incremental combination: CBT proposes pseudo test inputs, SBT materializes test inputs.

8 Conclusion

In this paper, we presented a novel combination of SBT and CBT to generate test inputs, called CBST. The novelties of our approach lie in its use of a relaxed version of a UUT with CBT and of CBT solutions (pseudo test inputs) as test input candidates in service of SBT. We identified three main points where CBT can be useful for SBT. For each point we proposed a technique of combination: a CPG that uses CBT to generate test input candidates for SBT; and a CEO that uses CBT to evolve test input candidates. We implemented a prototype of this approach. Then we compared three variants of CBST with CBT and SBT. Results of this comparison showed that CPG outperforms all techniques in terms of runtime and branch coverage. It is able to reach 89.5% branch coverage in less than 40 s. Also CEO and CPG+CEO perform better than SBT in terms of branch coverage. The obtained results are promising but more experiments must be performed using different sort of solvers (String, floating point) to confirm if the absolute advantage of the proposed techniques represents a general trend.

In the future, we will focus on extending our approach by exploring new combination techniques. In particular, we would like to enhance our technique CEO and to use several solvers at the same time.

References

1. Baars, A., Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., Tonella, P., Vos, T.: Symbolic search-based testing. In: ASE, 2011 26th IEEE/ACM International Conference on. pp. 53 –62 (nov 2011)
2. Clarke, L.: A system to generate test data and symbolically execute programs. Software Engineering, IEEE Transactions on SE-2(3), 215 – 222 (sept 1976)
3. Collavizza, H., Rueher, M., Hentenryck, P.V.: Cpbpv: a constraint-programming framework for bounded program verification. Constraints 15, 238–264 (2010)
4. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. SIGPLAN Not. 40, 213–223 (June 2005)
5. Gotlieb, A.: Euclide: A constraint-based testing framework for critical c programs. In: ICST. pp. 151–160 (2009)
6. Ince, D.C.: The automatic generation of test data. The Computer Journal 30(1), 63–69 (1987)

7. Inkumsah, K., Xie, T.: Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs. In: Proc. 22nd IEEE/ACM ASE. pp. 425–428 (November 2007)
8. Khichane, M., Albert, P., Solnon, C.: Strong combination of ant colony optimization with constraint programming optimization. In: Proceedings of the 7th international conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. pp. 232–245. CPAIOR'10, Springer-Verlag, Berlin, Heidelberg (2010)
9. Lakhotia, K., Tillmann, N., Harman, M., de Halleux, J.: Flopsy - search-based floating point constraint solving for symbolic execution. In: Petrenko, A., Simao, A., Maldonado, J. (eds.) Testing Software and Systems, Lecture Notes in Computer Science, vol. 6435, pp. 142–157. Springer Berlin / Heidelberg (2010)
10. Malburg, J., Fraser, G.: Combining search-based and constraint-based testing. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. ISSTA '11, ACM, New York, NY, USA (2011)
11. McMinn, P.: Search-based software test data generation: a survey. Software Testing Verification & Reliability 14, 105–156 (2004)
12. Miller, W., Spooner, D.: Automatic generation of floating-point test data. Software Engineering, IEEE Transactions on SE-2(3), 223 – 226 (sept 1976)
13. Myers, G.J.: The art of software testing. John Wiley and Sons (1979)
14. Păsăreanu, C.S., Rungta, N.: Symbolic pathfinder: symbolic execution of java bytecode. In: Proceedings of the IEEE/ACM international conference on Automated software engineering. pp. 179–180. ASE '10, ACM, New York, NY, USA (2010)
15. Sakti, A., Guéhéneuc, Y.G., Pesant, G.: Cp-sst : approche basée sur la programmation par contraintes pour le test structurel du logiciel. In: Septitièmes Journées Francophones de Programmation par Contraintes (JFPC). pp. 289–298 (June 2011)
16. Sen, K., Agha, G.: Cute and jcute: concolic unit testing and explicit path model-checking tools. In: Proceedings of the 18th international conference on CAV. pp. 419–423. CAV'06, Springer-Verlag, Berlin, Heidelberg (2006)
17. Sen, K., Marinov, D., Agha, G.: Cute: a concolic unit testing engine for c. SIGSOFT Softw. Eng. Notes 30, 263–272 (September 2005)
18. Staats, M., Păsăreanu, C.: Parallel symbolic execution for structural test generation. In: Proceedings of the 19th international symposium on Software testing and analysis. pp. 183–194. ISSTA '10, ACM, New York, NY, USA (2010)
19. Tillmann, N., de Halleux, J.: Pex-white box test generation for .net. In: Beckert, B., Hahnle, R. (eds.) Tests and Proofs, Lecture Notes in Computer Science, vol. 4966, pp. 134–153. Springer Berlin / Heidelberg (2008)
20. Tonella, P.: Evolutionary testing of classes. SIGSOFT Softw. Eng. Notes 29(4), 119–128 (Jul 2004)