Divide-by-Zero Exception Raising via Branch Coverage

Neelesh Bhattacharya, Abdelilah Sakti, Giuliano Antoniol
Yann-Gaël Guéhéneuc, and Gilles Pesant

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

Abstract. In this paper, we discuss how a search-based branch coverage approach can be used to design an effective test data generation approach, specifically targeting divide-by-zero exceptions. We first propose a novel testability transformation combining *approach level* and *branch distance*. We then use different search strategies, *i.e.*, hill climbing, simulated annealing, and genetic algorithm, to evaluate the performance of the novel testability transformation on a small synthetic example as well as on methods known to throw divide-by-zero exceptions, extracted from real world systems, namely Eclipse and Android. Finally, we also describe how the test data generation for divide-by-zero exceptions can be formulated as a constraint programming problem and compare the resolution of this problem with a genetic algorithm in terms of execution time. We thus report evidence that genetic algorithm using our novel testability transformation out-performs hill climbing and simulated annealing and a previous approach (in terms of numbers of fitness evaluation) but is out-performed by constraint programming (in terms of execution time).

Keywords: Exception raising, test input data generation, evolutionary testing.

1 Introduction

Consequences of uncaught or poorly-managed exception may be dire: program crashes and/or security breaches. For embedded systems, an exception can be caused by, for example, unexpected values read from a sensor and can cause catastrophic effects. Indeed, poorly-managed exceptions are at the root of the 1996 Arianne 5 incident during which an uncaught floating-point conversion exception led to the rocket self-destruction 40 seconds after launch. In aerospace, as in other domains requiring highly-dependable systems such as medical systems, poorly-managed exceptions may have severe consequences to human beings or lead to great economic losses.

In software engineering, testing have traditionally been one of the main activities to obtain highly-dependable systems. Testing activities consume about

M.B. Cohen and M. Ó Cinnéide (Eds.): SSBSE 2011, LNCS 6956, pp. 204–218, 2011.
© Springer-Verlag Berlin Heidelberg 2011

50% of software development resources [14] and any technique reducing testing costs is likely to reduce the software development costs as a whole. Although, exhaustive and thorough testing is often infeasible because of the possibly infinite execution-space and its high costs with respect to tight budget limitations, other techniques, such as code inspection are even costlier, though more effective. Therefore, testing activities should focus on the kinds of defects that, if were to slip into some deployed safety or mission-critical systems, may lead the systems to crash with possibly catastrophic consequences.

Consequently, we follow the work by Tracey *et al.* [19] on the generation of test data to raise exceptions and by others [13,20,2] on branch coverage criteria to propose a novel approach to generate test data for raising divide-by-zero exceptions for integers. In [19], the authors proposed to transform a target system so that the problem of generating test data to raise some exceptions becomes equivalent to a problem of branch coverage. They transform statements possibly leading to exceptions into branches with sufficient guard conditions. They then applied evolutionary testing to the transformed system to generate test data by traversing the branches and firing the exceptions. However, in their proposal, the search was solely guided by its *branch distance* [11], *i.e.*, the number of traversed control nodes, which may lead the search to behave like a random search.

In this paper, we propose to apply both *branch distance* as well as *approach level* [11] to generate test data to raise divide-by-zero exceptions for integers. The use of both guiding criteria, *i.e.*, *branch distance* and *approach level*, in an additive fitness function similar to previous work [20], to fire divide-by-zero exceptions yields to a reduction of the number of fitness evaluations needed to reach a given target statement [13,11]. To the best of our knowledge, this paper presents the first use of such a fitness function to generate test data to raise divide-by-zero exceptions.

We apply the novel testability transformation to generate test input data leading to divide-by-zero exceptions on the exemplary code presented in [19] and on two methods extracted from Eclipse and Android. We report the comparison of several meta-heuristic search techniques, *i.e.*, hill climbing (HC), simulated annealing (SA), and genetic algorithm (GA), with a random search (RND) and with constraint programming (CSP) on the three exemplary code samples. We thus show that the GA technique out-performs the other techniques but performs worse than the CSP technique.

Thus, the contributions of this paper are as follows:

- We propose to adopt both *branch distance* and *approach level* to generate test input data to fire divide-by-zero exceptions.
- We report the performance of HC (in three variants), SA, GA, and CSP on three systems, one synthetic and two real-world, from which we extracted three and two divide-by-zero exception-prone methods respectively.

The remainder of the paper is organized as follows: Section 2 describes the novel testability transformation. Section 3 describes the empirical study along with its settings. Sections 4 and 5 describe and discuss the results of the study. Section 6 summarizes related work. Section 7 concludes with some future work.

2 The Approach

We now present our approach, the novel testability transformation and the different techniques that we will use to generate test input data.

2.1 Example and Fitness Function

We follow previous work [19] to transform the problem of generating test input data to raise divide-by-zero exceptions into a branch coverage problem. This transformation essentially consists of wrapping divide-by-zero prone statements with a branch statement (an *if*), whose condition corresponds to the expression containing the possible division by zero. Consequently, satisfying the *if* through some branch coverage is equivalent to raising the divide-by-zero exception. For example, let us consider the following fragment of code:

```
1     int z, x=4;
2     if (Z>1 AND Z<=5)
3           return z;
4     else
5           return (x*4)/(z-1);
```

a divide-by-zero exception would be raised when z equals to 1 at line 5.

It is usually difficult to generate test data targeting a specific condition by obtaining appropriate variable values. We transform the code fragment above into a semantically-equivalent fragment in which the expression possibly leading to an exception becomes a condition. Then, it is sufficient to satisfy the new condition to obtain test input data raising the exception, as in the following fragment:

```
1     int z, x=4;
2     if (Z>1 AND Z<=5)
3           return z;
4     else
5.1         if (Z == 1)
5.2               print "Exception raised";
5.3         else
5.4               return (x*4)/(z-1);
```

where we transform the divide-by-zero prone statement at line 5 into the lines 5.1 to 5.4.

In general, such a transformation may not be trivial and different types of exceptions may require different types of transformations. Defining the types of transformations for various types of exceptions similarly to a Harman *et al.* framework [6] is future work.

To efficiently generate test data to expose the exception, it is not sufficient to reach line 5.1 using the *approach level* because between two test input data,

both reaching line 5.1, we would prefer the data making the condition at line 5.1 true and thus reaching line 5.2.

Consequently, our fitness function is an additive function using both the *approach level* and the normalized *branch distance* [1,8], where the normalized branch distance is defined by Equation 1 and used in the fitness function defined by Equation 2.

$$Normalized\ branch\ distance = 1 - 1.001^{-branch_distance} \quad (1)$$

$$Fitness\ function = Approach\ level + Normalized\ branch\ distance \quad (2)$$

2.2 Search Techniques to Generate Test Input Data

Once the code fragments under test are transformed and instrumented to collect run-time variable values needed to compute the *approach level* and normalized *branch distance*, we can generate test input data using several techniques, such as hill climbing (HC), simulated annealing (SA), and genetic algorithms (GA), or simply by a random search (RND). In the following section, we briefly describe the settings of the various techniques as well as the use of constraint programming for structural software testing (CP-SST)[16].

We assume that the input values are integers. Other more structured data types will be investigated as part of our future works, following the strategy proposed in [5].

Hill Climbing. Hill climbing (HC) is the simplest, widely-used, and probably best-known search technique. HC is a local search method, where the search proceeds from a randomly chosen point (solution) in the search space by considering the neighbours (new solutions obtained by mutating the previous solution) of the point. Once a fitter neighbour is found, this becomes the current point in the search space and the process is repeated. If, after mutating a given x number of times no fitter neighbour is found, then the search terminates and a maximum has been found (by definition). Often, to avoid local optima, the hill climbing algorithm is restarted multiple times from a random point (also called stochastic hill climbing). We have drawn inspiration from this idea and proposed three strategies.

Strategy 1: The HC1 strategy generates, for any input variable involved in the denominator of the exceptions raising statements, an immediate neighbour of the input data as the sum of the the current value of the variable with a randomly-generated value drawn from a Gaussian distribution with zero mean and an initial standard deviation (SD) of ten, which we choose after several trials to have neighbourhoods that are not too small or large.

If after a given number of moves in the neighbourhoods, the fitness values of the neighbours are always worse than the current fitness value, then we change the value of the SD to a larger value to expand the neighbourhood and give the algorithm an opportunity to get out of the, possible local optimum.

Strategy 2: In the HC1 strategy, the values of the SD may change at run-time. We observed that HC1 does not always improve the search and may lead to a slow search-space exploration. Thus, to avoid getting "trapped" in a specific region of the space, we define the HC2 strategy that forces the search to take a jump away from unsuccessful neighbourhood in an attempt to move into a more favourable neighbourhood, similarly to HC with random restarts.

For a given SD value, if the search does not improve for a given number of iterations, instead of changing SD, we force a jump to another neighbourhood using a large value and HC reiterates its process. The "length" of a jump and the number of jumps depend on the search space. For example, a search space of $[-10,000; +10,000]$ would be likely covered with 40 jumps of lengths 500. As with the previous strategy, this strategy goes on until it reaches a maximum number of iterations or generates test data firing the targeted exception.

Strategy 3: The HC3 strategy is a combination of HC1 and HC2. With HC3, we store the fitness values of the best neighbour of all previously-visited neighbourhoods before jumping to another neighbourhood. After having visited a given number of neighbourhoods as in HC2, HC3 returns to the "best" one, *i.e.*, the neighbourhood with the best fitness value among all recorded values, and then increases the SD by 25 (making the SD 35), as in HC1, to visit more of this neighbourhood. As with the previous strategies, this strategy goes on until it reaches a maximum number of iterations or generates test data firing the targeted exception.

Simulated Annealing. SA like hill climbing, is a local search method. However, simulated annealing has a 'cooling mechanism' that initially allows moves to less fit solutions if $p < m$, where p is a random number in the range $[0 \ldots 1]$ and m, acceptance probability, is a value that decays ('cools') at each iteration of the algorithm. The effect of 'cooling' on the simulation of annealing is that the probability of following an unfavourable move is reduced. This (initially) allows the search to move away from local optima in which the search might be trapped. As the simulation 'cools' the search becomes more and more like a simple hill climb. The choice of the parameters of SA are guided by two equations [21]:

$$Final\ temperature = Initial\ temperature \times \alpha^{Number\ of\ iterations}$$
$$Acceptance\ probability = e^{\frac{-Normalized\ fitness\ function}{Final\ temperature}}$$

where *Acceptance probability* if the probability that the algorithm decides to accept solution or not: if the current neighbour is "worse" than previous ones, it can still be accepted (in contrary to hill climbing) if the probability is greater than a randomly-selected value; α and $Number\ of\ iterations$ are chosen constants.

Genetic Algorithms. A GA starts by creating an initial population of n sets of test input data, chosen randomly from the domain D of the program being tested. Each chromosome represents a test set; genes are values of the input variables. In

an iterative process, the GA tries to improve the population from one generation to another using the fitness of each chromosome to perform reproduction, *i.e.*, cross-over and–or mutation. The GA creates a new generation with the l fittest test sets of the previous generation and the offspring obtained from cross-overs and mutations. It keeps the population size constant by retaining only the n best test sets in each new generation. It stops if either some test set satisfies the condition or after a given number of generations.

We implemented our GA algorithm using *JMetal*[1]. We use the binary tournament selection operator. Once parents are selected, they undergo a single point cross-over. To diversify the population, the off-springs (after cross-overs) may be mutated. We use bit-flip mutation, in which, based on a probability, the variable values are replaced with values selected randomly.

Random Search. Not using any heuristics to guide the search, this technique relies on generating initial input variable data randomly to execute the transformed program and to try firing the targeted exception. It stops if either the generated test data fires the exception or after a given number of iterations.

Constraint Programming. Constraint programming for software structural testing (CP-SST) is a generic technique for test-data generation to reach a specific target or to satisfy a test coverage criteria, for proof post-condition, or for counter-example generation.

The main idea of CP-SST is to convert the program under test and the test target into a constraint solving problem (CSP) and to solve the resulting CSP to obtain test input data. The first step of CP-SST consists of transforming the program under test into the static single assignment (SSA) form. The second step consists of modelling the program control flow graph (CFG) as a preliminary CSP. CP-SST begins by generating the CFG that features an independent node for each parameter and global variable, each control statement, each block of statements, and each join point. CP-SST labels edges among nodes depending on the origin node: an edge outgoing from a statement node is labelled by 1, an edge outgoing from a condition node is labelled by 1 if the decision is positive and -1 if the decision is negative. Then, CP-SST generates the preliminary CSP by translating each node into a CSP variable whose domain is the set of labels of its outgoing edges, except for join nodes that take their domains from the joint nodes.

In the third step, CP-SST uses the preliminary CSP, the SSA form, and the relationships among nodes and their statements to create a new global CSP, which it then solves to generate test input data or return a failure if no solution can be reached.

3 Empirical Study

The *goal* of our empirical study is to compare our novel testability transformation against previous work and identify the best technique among the five

[1] http://jmetal.sourceforge.net/

Table 1. Details of the systems under test and tested units

Systems	Versions	Class Names	LOCs	Numbers of Exceptions	Bug Tracking Numbers
Tracey's code	N/A	F	13	2	N/A
Eclipse	2.0.1	GridCanvas	10	2	205772
Android	2.0	ProcessStats	41	3	Unavailable

presented in Section 2 using synthetic as well as real systems to generate integer test input data to fire divide-by-zero exceptions. The *quality focus* is the performance of the proposed hill climbing strategies and other meta-heuristics and constraint programming techniques to raise divide-by-zero exceptions. The *perspective* includes researchers and software engineers working in search-based software testing looking to generate test data for firing exceptions in the code. The *context* of our research includes three case studies: one synthetic program and two real software systems, namely, *Eclipse* and *Android*. Table 1 summarizes the three software systems and the selected methods/functions for testing and the corresponding class names having two, two, and three divide-by-zero exception statements, respectively.

We seek answers to four research questions:

RQ1: Based on the fitness function we use, which of the three proposed hill climbing strategies is best suited to raise a divide-by-zero exception and what is the measure of its effectiveness?

RQ2: Which of all the meta-heuristic techniques is best suited to raise a divide-by-zero exception and what is the measure of its effectiveness? (Retaining the best-suited hill climbing strategy from RQ1.)

RQ3: Which of Tracey's fitness function and the fitness function we used, is best suited to raise a divide-by-zero exception and what is the measure of its effectiveness? (Retaining the best-suited meta-heuristic from RQ2.)

RQ4: Which of the best-suited meta-heuristic technique and of the CP-SST is best suited to raise a divide-by-zero exception and what is the measure of its effectiveness? (Retaining the best-suited meta-heuristic from RQ2.)

3.1 Choice of the Comparison Measure

HC (in the three variants), SA, GA, and RND can be compared with one another using their numbers of fitness evaluations to fire some divide-by-zero exception. CP-SST is based on a completely different paradigm than the meta-heuristic techniques. Thus, CP-SST cannot be compared with the other techniques using the numbers of fitness evaluations and we use execution times of the different techniques for comparison. We consider the approach requiring less execution time to reach a target to be "better" to generate test data for firing divide-by-zero exceptions.

3.2 Choice of the Targeted Exceptions

We selected three methods in three classes of three different systems, for a total of seven possible target exceptions. For the sake of space, we only report in the

following the results of our empirical study for three target exceptions, chosen to lead to the worst performance for all the techniques, among the seven possible targeted exceptions and called in the following units-under-test, UUT. All results and data for replication are available on-line[2].

3.3 General Parameters of the Techniques

We chose different ranges of values for each input variable to analyse the performance of all the techniques to deal with values ranging from very small to very large. The domains have been varied from $[-100; +100]$ to $[-50,000; +50,000]$ for all the input variables. Reaching a success, *i.e.*, raising the targeted divide-by-zero exception in the UUT, is the stopping criterion as well as a number of evaluations of 1000000. We repeated each computations 20 times to analyse the diversity in the observed values and conduct statistical tests. Table 2(a) details the values used.

Table 2. Parameters

(a) General Parameters

Input Domain	$[-100; +100]$ - $[-50,000; +50,000]$
Max # iterations	1000000
# Computations	20

(b) Hill Climbing

Strategies	**CS**	**SD**
S1	100	-
S2	100	-
S3	100	35

(c) Simulated Annealing

Params.	**Inspected Values (Chosen)**
Temper.	0.5-50 (20)
α	0.8-0.995 (0.99)
# Iter.	10-500 (100)

(d) Genetic Algorithm

Operators	**Type**	**Prob.**
Crossover	Single Point	0.9
Mutation	Bit Flip	0.09
Selection	Binary Tournament	-

3.4 Specific Parameters of the Techniques

We use the following parameters for the techniques (we do not report techniques which do not use any particular parameters):

Hill Climbing: For the first strategy, we use a parameter *checkStagnation* to control the number of iterations before changing neighbourhood. We use 100, *i.e.*, if no improvement occurs in a neighbourhood after 100 iterations, HC1 changes neighbourhood. We use 100 as SD because it led to better performance than other values.

In the second strategy, we also use *checkStagnation* parameter. We use two other important parameters: *gaussianJumpLength* and *numberOfJumps* depicting the length of the "jump" and the number of jumps, respectively. We found the values of these two parameters by trial-and-error runs.

In the third strategy, we also use *deviationValue*, the value of SD when the technique returns to the best neighbourhood. Table 2(b) depicts the parameter values.

[2] `http://web.soccerlab.polymtl.ca/ser-repos/public/div_by_zero.tar.gz`

Simulated Annealing: We base our choice of the initial temperature, α, and the number of iterations on several experiments in which we varied the initial temperature. Table 2(c) shows the values used.

Genetic Algorithm: We used single-point cross-over, bit-flip mutation, and binary-tournament selection. We choose the binary-tournament selection because its complexity is lower than that of any other selections and provides more population diversity to the cross-over operator than others [23]. Table 2(d) shows the various values.

4 Study Results

Figure 1, 2, and 3 reports the box plots of the number of fitness evaluation needed to raise a divide-by-zero exception for the Tracey exemplary code, Eclipse, and Android UTTs, respectively. We did not include results for RND as it performs always substantially worse then even the slowest HC1 strategy.

We observe that HC3 is, in all cases, better then the other two hill climbing strategies but in all cases is also performing substantially worse than SA and GA. Overall, Figure 1, 2, and 3 support the observation that GA is the most effective meta-heuristic technique as far as these UTTs are concerned.

Table 3, 4, and 5 reports the t-test values comparing the numbers of fitness evaluations needed by the different search techniques as well as the Cohen d effect size [4]. The effect size is defined as the difference between the means of two groups, divided by the pooled standard deviation of both groups. The effect size is considered small for $0.2 \leq d < 0.5$, medium for $0.5 \leq d < 0.8$, and large for $d \geq 0.8$ [4]. We chose the Cohen d effect size because it is appropriate for our variables (ratio scales) and given its different levels (small, medium, large) easy to interpret.

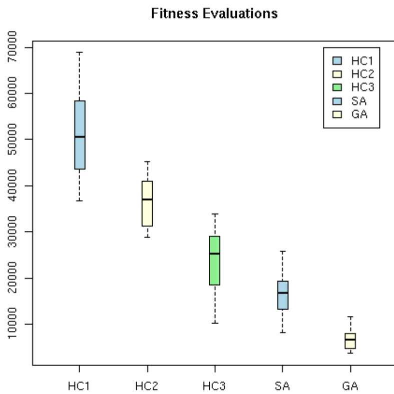

Fig. 1. Comparison on Tracey [19] UUT of the different search techniques (input domain $[-50,000; +50,000]$)

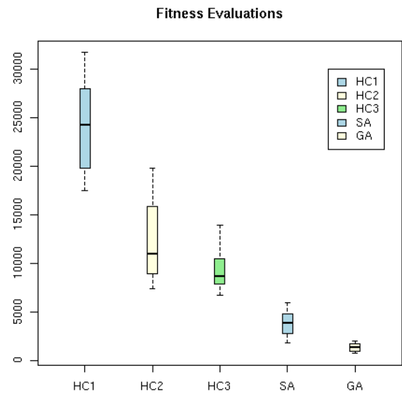

Fig. 2. Comparison on Eclipse UUT of the different search techniques (input domain $[-50,000; +50,000]$)

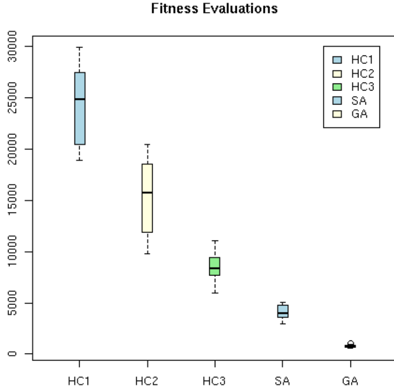

Fig. 3. Comparison on Android UUT of the different search techniques (input domain $[-50,000; +50,000]$)

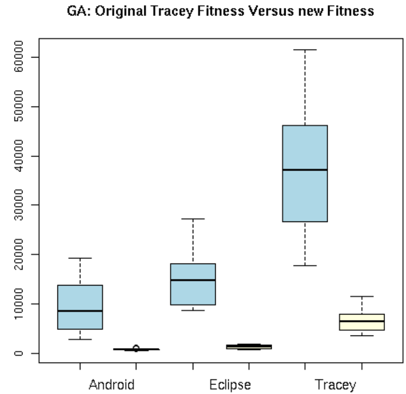

Fig. 4. GA comparison against Tracey's original fitness [19] versus the fitness function we used (input domain $[-50,000; +50,000]$)

Table 3. Results of t-test and Cohen d effect size for Tracey [19] UUT

Comparisons	p-values	Cohen d values
HC1-HC2	9.261e-10	2.55246
HC1-HC3	6.376e-16	5.951003
HC2-HC3	6.868e-08	2.428475
HC3-SA	7.049e-14	4.147889
HC3-GA	2.2e-16	8.223645
SA-GA	8.763e-15	6.254793

Table 4. Results of t-test and Cohen d effect size for Eclipse UUT

Comparisons	p-values	Cohen d values
HC1-HC2	3.245e-10	2.682195
HC1-HC3	7.167e-13	4.111778
HC2-HC3	0.003998	0.9912142
HC3-SA	2.387e-11	3.239916
HC3-GA	1.933e-13	5.258295
SA-GA	9.989e-09	2.694495

Table 5. Results of t-test and Cohen d effect size for Android UUT

Comparisons	p-values	Cohen d values
HC1-HC2	1.438e-06	1.894204
HC1-HC3	2.981e-12	3.345377
HC2-HC3	7.438e-08	2.12037
HC3-SA	0.0003169	1.266088
HC3-GA	1.283e-10	3.481401
SA-GA	4.531e-09	2.696728

Table 6. Comparison of GA against CP-SST in terms of average execution times (ms) and standard deviations for all UUTs

	Tracey's Code	Eclipse	Android
GA	8.067/1.439	2.129/1.149	1.926/1.177
CP	1.035/0.0135	0.01/0	0.01/0

As expected from Figures 1, 2, and 3, the t-test and Cohen d effect size results support with very strong statistical evidence the superiority of HC3 over HC1 and HC2 as well as the superiority of GA over SA and HC3.

Box-plots as well as tables clearly support the superiority of GA over the other techniques. Overall, **we answer RQ1 by stating that HC3 performs better than HC1 and HC2 with a large effect size.** Furthermore, **we answer RQ2 by stating, with a large effect size also, that GA outperforms the other techniques.**

We compare our fitness function with that proposed by Tracey *et al.* [19] by using two GA implementations, one using Tracey's fitness function and another using the fitness function explained before. We compare the two fitness functions in terms of the required numbers of fitness evaluations for all the UUTs in Figure 4 to reach the targeted exceptions. Figure 4 shows that our novel testability transformation allows the GA to reach the targeted exceptions in much less numbers of evaluations. Consequently, **we answer RQ3 by stating that, in comparison to Tracey's fitness function, our novel testability transformation dramatically improves the performance of a GA technique.**

Finally, Table 6 reports average and standard deviations of the execution times for twenty experiments on the three UUTs for both GA and CP. For the given UUTs, it is clear that CP out-performs GA in term of execution times. This result may be due to the size of the UUTs, which are relatively small, and to the structure of the condition to satisfy. More evidence is needed to verify if the averages in Table 6 represent a general trend. Yet, on the selected UUTs, **we answer RQ4 by claiming that the CP-SST technique out-performs the best of the meta-heuristic techniques, GA.**

5 Study Discussions

5.1 Discussions

We presented the results of three UUTs to answer the four research questions. The other four UUTs from the same systems exhibit the same trends as the ones reported in this paper, thus adding more evidence to our answers.

We also evaluated the performance of our novel testability transformation with respect to the one proposed by Tracey *et al.* [19] in terms of required numbers of fitness evaluations. The results showed the importance of having both *approach level* and *branch distance* in the fitness function, as opposed to the one proposed by Tracey *et al.* [19] which uses only the *approach level.*

5.2 Threats to the Validity

We now discuss the threats to the validity of our study.

Threats to *construct validity* concern the relationship between theory and observation. In our study, these threats can be due to the fact that one of the UUT is a synthetic code, even though previously-used to exemplify and study the divide-by-zero exception [19], and thus might represent real code. However, we extracted the two other UUTs from real-world systems (Eclipse and Android) and the method containing the divide-by-zero exceptions has been documented in the Eclipse issue tracking system. Finally, the code excerpt [19] as well as the Eclipse and Android studied methods contain multiple possible divide-by-zero statements and, in all cases, we focused on the statements leading to the worst performances, *i.e.*, the most deeply-nested statements.

Threats to *internal validity* concern external factors that may affect an independent variable. We limited the bias of intrinsic randomness of our results

by repeating each experiment 20 times and using proper statistics to compare the results. We have calibrated the HC (*i.e.*, HC1, HC2, and HC3), SA, and GA settings using a trial-and-error procedure. We chose the values of the parameters of the techniques, such as *checkStagnation*, *gaussianJumpLength* and so on, after executing the techniques several times and evaluating their performance on a toy program. We also chose the cross-over and mutation operators by doing a small study on the same toy program: although we found evidence of the superiority of specific operators, it could happen that (1) studies on different systems would lead to a different choice of cross-over and mutation operators and (2) the obtained calibration may not be the most suitable for our subject systems.

Threats to *conclusion validity* involve the relationship between the treatment and the outcome. To overcome this threat, we inspected box-plots, performed t-tests, and evaluated the Cohen d effect sizes.

Threats to *external validity* involve the generalization of our results. We evaluated the novel testability transformation on UUTs from the work of Tracey *et al.* [19] and two different Java systems. The sample size is small and, although for Eclipse code it corresponds to a documented bug, a larger evaluation is be highly desirable.

Finally, for all divide-by-zero conditions listed in Table 1 and all applied search techniques, including random search, a replication package is available on-line[3] to promote replication.

6 Related Work

Our approach stems from the work of Tracey *et al.* [19]. They proposed an approach to automatically generate test data for exceptions by (1) transforming the statements containing exceptions into a branch with guard conditions derived from the possible exception and the statement structure and (2) generating test data to traverse the added branch and thus fire the exception. The fitness function used in [19] is in essence oriented to structural coverage and uses only the *branch distance*.

Automation of structural coverage criteria and structural testing have been the most-widely investigated subjects. Local search was first used by Miller and Spooner [12] with the goal of generating input data to cover particular paths in a system. This work was later extended by Korel [10]. In brief, to cover a particular path, the system is initially executed with some arbitrary input. If an undesired branch is taken, an objective function derived from the predicate of the desired branch is used to guide the search. The objective function value, referred to as *branch distance* [8], measures how close the predicate is to being true. Baresel *et al.* [1] proposed a normalization of the *branch distance* between in $[0, 1]$ to better guide the search avoiding *branch distance* making *approximation level* useless. The idea of minimizing such an objective function was refined and extended by several researchers to satisfy coverage criteria of certain given procedural-program structures like branches, statements, paths, or conditions.

[3] `http://web.soccerlab.polymtl.ca/ser-repos/public/div_by_zero.tar.gz`

To overcome the limitations associated with local search techniques, Tracey *et al.* [**?**] applied simulated annealing and defined a more sophisticated objective function for relational predicates. The genetic algorithm was first used by Xanthakis [22] to generate input data satisfying the all branch predicate criterion. Evolutionary approaches, where search algorithms, in particular genetic algorithm, are tailored to automate and support testing activities, *i.e.*, to generate test input data [9,18,20] are often referred to as evolutionary-based software testing or simply evolutionary testing. A survey of evolutionary testing and related techniques is beyond the scope of this paper; the interested reader may refer to the survey published by McMinn [11].

In the last few years, researchers have focused on static, dynamic and hybrid approaches to identify and handle various types of exceptions in object-oriented systems. Sinha *et al.* [17] proposed an approach to reduce the complexity of a program in the presence of implicit control flow. The approach, based on static and dynamic analysis of constructs, provides information to developers in an IDE. Ryder *et al.* [15] studied the tool *JESP* and evaluated the frequency with which exception-handling constructs are used in Java programs. Their analysis found that exception-handling constructs were used in 16% of the methods that they examined. Chatterjee *et al.* [3] proposed an approach for data-flow testing. They identified the definition–use associations arising along with the exceptional control-flow paths. Jang *et al.* [7] proposed an exception analysis approach for Java, both at the expression and method level, to overcome the dependence of JDK Java compiler on developers' declarations for checking against uncaught exceptions.

Our work shares many commonalities with previous work, as we apply structural evolutionary testing developed for branch coverage to generate test input data exposing divide-by-zero exceptions in a unit under test transformed as proposed by Tracey *et al.* [19].

7 Conclusion

In this paper, we presented a novel testability transformation to generate test input data to raise divide-by-zero exceptions in software systems. We compared the performance of hill climbing, simulated annealing, genetic algorithm, random search, and constraint programming when using this fitness function. The novel testability transformation used by hill climbing, simulated annealing, and genetic algorithm is based on both *approach level* and *branch distance*. Further, we also proposed three hill climbing strategies to improve basic hill climbing search. Finally, we chose the best meta-heuristic technique (genetic algorithm) and compared its performance with that of constraint programming in terms of execution time.

We validated our novel testability transformation and compared the search technique on three software units: one synthetic code fragment taken from [19] and two methods extracted from Eclipse and Android, respectively. While comparing the meta-heuristic techniques, genetic algorithm performed best in terms

of the number of required fitness evaluations to reach the desired target for all the three units under test. Then, constraint programming out-performed the genetic algorithm in terms of execution times for all the three case studies.

In the future, we will validate our fitness function and choice of search technique with more complex input data types and different types of exceptions. We will also extend the validation part to other software systems. We would also like to integrate a chaining approach to better deal with data dependencies and study the testability transformations required to simplify and make it efficient to generate test input data to raise exceptions.

References

1. Baresel, A.: Automatisierung von strukturtests mit evolutionren algorithmen. Diploma Thesis, Humboldt University, Berlin, Germany (2000)
2. Baresel, A., Sthamer, H., Schmidt, M.: Fitness function design to improve evolutionary structural testing. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1329–1336 (July 2002)
3. Chatterjee, R., Ryder, B.G.: Data-flow-based testing of object-oriented libraries. Tech. Rep. DCS-TR-382, Department of Computer Science, Rutgers University (1999)
4. Cohen, J.: Statistical power analysis for the behavioral sciences, 2nd edn. Lawrence Earlbaum Associates, Hillsdale (1988)
5. Romano, D., Massimiliano Di Penta, G.A.: An approach for search based testing of null pointer exceptions. In: Proceedings of the Fourth International Conference on Software Testing, Verification and Validation, pp. 160–169 (March 2011)
6. Harman, M., Baresel, A., Binkley, D., Hierons, R.M., Hu, L., Korel, B., McMinn, P., Roper, M.: Testability transformation – program transformation to improve testability. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 320–344. Springer, Heidelberg (2008)
7. Jo, J.W., Chang, B.M., Yi, K., Choe, K.M.: An uncaught exception analysis for java. Journal of System Software 72, 59–69 (2004), http://portal.acm.org/citation.cfm?id=1005486.1005491
8. Joachim Wegener, A.B., Sthamer, H.: Evolutionary test environment for automatic structural testing. Information and Software Technology 43(14), 841–854 (2001)
9. Jones, B., Sthamer, H., Eyres, D.: Automatic structural testing using genetic algorithms. Software Engineering Journal 11(5), 299–306 (1996)
10. Korel, B.: Dynamic method of software test data generation. Softw. Test, Verif. Reliab. 2(4), 203–213 (1992)
11. McMinn, P.: Search-based software test data generation: a survey. Software Testing Verification and Reliability 14(2), 105–156 (2004)
12. Miller, W., Spooner, D.L.: Automatic generation of floating-point test data. IEEE Transactions on Software Engineering 2(3), 223–226 (1976)
13. Mresa, E.S., Bottaci, L.: Efficiency of mutation operators and selective mutation strategies: An empirical study. Software Testing Verification and Reliability 9(4), 205–232 (1999)
14. Pressman, R.S.: Software Engineering: A Practitioner's Approach, 3rd edn. McGraw-Hill, New York (1992)

15. Ryder, B.G., Smith, D.E., Kremer, U., Gordon, M.D., Shah, N.: A static study of java exceptions using JESP. In: Watt, D.A. (ed.) CC 2000. LNCS, vol. 1781, pp. 67–81. Springer, Heidelberg (2000), http://portal.acm.org/citation.cfm?id=647476.727763
16. Sakti, A., Guéhéneuc, Y.G., Pesant, G.: Cp-sst: approche basée sur la programmation par contraintes pour le test structurel du logiciel. Septitièmes Journées Francophones de Programmation par Contraintes (JFPC), 289–298 (June 2011)
17. Saurabh Sinha, R.O., Harrold, M.J.: Automated support for development, maintenance, and testing in the presence of implicit control flow. In: ICSE 2004, pp. 336–345. IEEE Computer Society Press, Washington, DC, USA (2004)
18. Tracey, N., Clark, J.A., Mander, K., McDermid, J.A.: Automated test-data generation for exception conditions. Software Practice and Experience 30(1), 61–79 (2000)
19. Tracey, N., Clark, J.A., Mander, K.: Automated program flaw finding using simulated annealing. In: ISSTA, pp. 73–81 (1998)
20. Wegener, J., Baresel, A., Sthamer, H.: Evolutionary test environment for automatic structural testing. Information & Software Technology 43(14), 841–854 (2001)
21. Wright, M.: Automating parameter choice for simulated annealing. Tech. Rep. 32, Lancaster University Management School, UK (2010)
22. Xanthakis, S., Ellis, C., Skourlas, C., Gall, A.L., Katsikas, S., Karapoulios, K.: Application des algorithmes genetiques au test des logiciels. In: 5th Int. Conference on Software Engineering and its Applications, pp. 625–636 (1992)
23. Zhang, B.T., Kim, J.J.: Comparison of selection methods for evolutionary optimization. Evolutionary Optimization 2(1), 55–70 (2000)