Evaluating Normalization Functions with Search Algorithms for Solving OCL Constraints

Shaukat Ali and Tao Yue

Certus Software V&V Center,
Simula Research Laboratory, P.O. Box 134,
Lysaker, Norway
{shaukat,tao}@simula.no

Abstract. The use of search algorithms requires the definition of a fitness function that guides the algorithms to find an optimal solution. The definition of a fitness function may require the use of a normalization function for various purposes such as assigning equal importance to various factors constituting a fitness function and normalizing only one factor of a fitness function to give it less/more importance than the others. In our previous work, we defined various branch distance functions (a commonly used heuristic in the literature at the code-level) corresponding to the constructs defined in the Object Constraint Language (OCL) to solve OCL constraints to generate test data for supporting automated Model-Based Testing (MBT). The definition of several of these distance functions required the use of a normalization function. In this paper, we extend the empirical evaluation reported in one of the works in the literature that compares the impact of using various normalization functions for calculating branch distances at the code-level on the performance of search algorithms.

The empirical evaluation reported in this paper assesses the impact of the commonly used normalization functions for the branch distance calculation of OCL constraints at the model-level. Results show that for one of the newly studied algorithms Harmony Search (HS) and Random Search (RS), the use of the normalization functions has no impact on the performance of the search. However, HS achieved 100% success rates for all the problems, where RS obtained very poor success rates (less than 38%). Based on the results, we conclude that the randomness in creating a solution in search algorithms may mask the impact of using a normalization function.

Keywords: OCL, Search Algorithm, Test data, Model-Based Testing, Empirical evaluation.

1 Introduction

The Object Constraint Language (OCL) is a commonly used language to specify constraints on the Unified Modeling Language (UML) for various applications, such as supporting Model-Based Testing (MBT) [1, 2] and automated code generation [3]. In the context of MBT, a typical purpose of solving constraints is to generate test data for automated generation of executable test scripts from models. In our previous work

M.G. Merayo and E. Montes de Oca (Eds.): ICTSS 2014, LNCS 8763, pp. 17–31, 2014.

[4], to solve OCL constraints on models for automated test data generation, we defined a set of novel heuristics for various OCL constructs. Some of these heuristics require the use of a normalization function, which is proposed in [4]. In this paper, we empirically evaluate the impact of two commonly used normalization functions in the literature on the performance of various search algorithms for solving OCL constraints for test data generation. The main motivation of this empirical evaluation is to assess if the use of a particular normalization function can improve the performance of solving OCL constraints for test data generation. Studying the impact of the normalization functions on the performance of search algorithms is important since any improvement in the efficiency of test data generation in our application of industrial MBT is appreciated.

Based on the above motivation, we conducted an empirical evaluation to assess the impact of the two commonly used normalization functions reported in [5, 6] on the performance of various search algorithms to solve OCL constraints. The differences of this paper from the one reported in [5, 6] are as follows: 1) We evaluate the use of the normalization functions on heuristics defined on model-level constraints as compared to the code-level branches reported in [5, 6]; 2) We empirically evaluated in total five search algorithms: Genetic Algorithm (GA) and Random Search (RS)—two algorithms that have been evaluated in [5, 6], and the other three algorithms (Alternating Variable Method (AVM), (1+1) Evolutionary Algorithm (EA), and Harmony Search (HS) [7]) which are newly evaluated in this paper.

Based on our empirical evaluation, we observed that the extent of the randomness consideration in generating a solution may mask the effect of using a particular normalization function. For HS and RS, where the extent of the randomness consideration of generating a solution is higher than the other three algorithms we studied, the effect of using a particular normalization function on performance of search is masked. However, HS achieved 100% success rate, whereas RS achieved low success rates (37%). Therefore, HS is recommended for solving OCL constraints when combined with any of the normalization function.

The rest of the paper is organized as follows. Section 2 presents the background necessary to understand the rest of the paper. Section 3 presents our empirical evaluation. Section 4 reports threats to validity of the empirical evaluation. Section 5 provides the related work and Section 6 concludes the paper.

2 Background

In this section, we briefly provide some information required to understand the rest of the paper. Section 2.1 provides a brief introduction to our existing work on test data generation from OCL constraints to provide the context in which the two normalization functions are evaluated. Section 2.2 presents the two commonly used normalization functions in search-based software engineering (SBSE), whereas Section 2.3 provides a brief introduction to Harmony Search (HS) since it is new to SBSE.

2.1 Test Data Generation from OCL Constraints

In our previous work [4], we proposed and assessed novel heuristics for the application of search-based techniques, such as Genetic Algorithms (GAs), (1+1) Evolutionary Algorithm (EA), and Alternating Variable Method (AVM), to generate test data from OCL constraints. A search-based test data generator (EsOCL) was implemented in Java and was evaluated on an industrial case study in addition to the empirical evaluation of each proposed heuristics using several artificial problems.

To guide the search for test data that satisfy OCL constraints, a heuristic tells *'how far'* input data are from satisfying the constraint. For example, let us say we want to satisfy the constraint *x=0*, and suppose we have two data inputs: *x1:=5* and *x2:=1000*. Both inputs *x1* and *x2* do not satisfy *x=0*, but *x1* is heuristically closer to satisfy *x=0* than *x2*. A search algorithm would use such a heuristic as a fitness function, to reward input data that are closer to satisfy a target constraint.

To generate test data to solve OCL constraints, we used a fitness function that was adapted from work done for code coverage (e.g., for branch coverage in C code [8]). In particular, we used the a fitness function so called branch distance (a function *d()*), as defined in [8]. The function *d()* returns *0* if the constraint is solved (an OCL constraint evaluates to *true*), a value *k* if the constraint evaluates to *undefined* (a special case for OCL that makes it three-valued logic for writing constraints [4]); otherwise a positive value (greater than *0* and less than *k*) that heuristically estimates how far the constraint was from being evaluated to be *true*. As for any heuristic, there is no guarantee that an optimal solution (e.g., in our case, input data satisfying constraints) will be found in a reasonable period of time. Nevertheless, many successful results have been reported in the literature for solving various software engineering problems by applying heuristic based approaches [9]. In cases where we wanted a constraint to be evaluated to be *false*, we simply negated the constraint and find data for which the negated constraint evaluates to *true*. For example, if we want to prevent firing a guarded transition in a state machine, we can simply negate the guard and find data for the negated guard. We defined various novel heuristics for calculating branch distances for various constructs and operations defined in OCL such as *includes()*, *oclInState()*, and *forAll()* [4] and due to space limitations we cannot provide details about them in this paper. The interested readers may consult the relevant reference [4] for more details.

2.2 Normalization Functions

A normalization function is commonly used in the branch distance calculation (a commonly used heuristic [8]) to normalize a branch distance value within range [0, 1]. The minimum value of the branch distance, i.e., 0 is known; however the maximum value for the branch distance is not known and thus it is important to normalize the branch distance in the range [0, 1].

In SBSE, the following two normalization functions are commonly used:

$$N1(x) = \frac{x}{x+\beta} \tag{1}$$

$$N2(x) = 1 - \alpha^{-x} \quad \ldots \tag{2}$$

In the above functions, $\alpha > 1$ and $\beta > 0$ and in the literature a typical value of $\alpha = 1.001$ is used [5, 6]. Both of the above functions maintain the order of inputs with outputs, i.e., a higher value of input to the normalization functions will output a higher value of the branch distance [5, 6]. Also notice that branch distance values are always greater than or equal to 0, and the minimum value $x=0$ will produce $N1(0) = 0$ and $N2(0) = 0$.

2.3 Harmony Search

Harmony Search (HS) [10] is a meta-heuristic music-inspired algorithm for global optimization. Music is produced by a group of musicians (variables) together, where each musician produces a note (value) to find a best harmony (global optimal solution). To explain how HS works, considering we have a set of decision variables $V = \{v_1, v_2, .., v_n\}$, HS starts search with *Harmony Memory* (*HM*) consisting of a set of randomly generated solutions (i.e., $HM = \{s_1, s_2, .., s_m\}$) where each s_i consists of randomly generated values for v_1, v_2, .., v_n. To create a new solution, each variable in V is set to a value from one of the solutions in *HM* with a probability *Harmony Memory Consideration Rate* (*HMCR*) and an additional modification to the selected value with *Pitch Adjusting Rate* (*PAR*). Otherwise, the variable is set to a random value. This case is referred to as random consideration with a probability of (*1-HMCR*) [10]. If the newly created solution has better fitness than the worst solution in *HM*, then it is replaced with the new solution. The process of creating new solutions continues until the termination criteria are met. On reaching the termination criteria, the solution with the best fitness from *HM* is selected. More details of the algorithm can be found in [10].

3 Empirical Evaluation

In this section, we will provide an empirical evaluation based on various artificial problems (Table 6) to evaluate the two normalization functions (Section 2.2).

3.1 Experiment Design

To empirically evaluate whether the normalization function *N1* really improves the performance of a search algorithm as compared to *N2*, we carefully defined artificial problems to evaluate each heuristic that requires a normalization function. Based on this criterion, we defined eight artificial problems (Table 6). The model we used for the experiment consists of a very simple class diagram with one class *X*, which has two attributes *x* and *y* of type *Integer*. We populated *10* objects of class *X*. The use of a single class with 10 objects was sufficient to create complex constraints since the overall solution space of each constraint is very small.

In our experiments, we compared four search algorithms: AVM, HS, GA, (1+1) EA, and used RS as a comparison baseline to assess the difficulty of the addressed problems [11]. AVM was selected as a representative of local search algorithms. HS [7] is a music-inspired algorithm, which is not commonly used in SBSE. GA was selected since it is the most commonly used global search algorithm in SBSE [11]. (1+1) EA is simpler than GAs, but in our previous works we found that it can be more effective for software testing problems (e.g., see [4, 12, 13]). For GA, we set the population size to 100 and the crossover rate to 0.75, with a 1.5 bias for rank selection. We use a standard one-point crossover, and mutation of a variable is done with the standard probability $p = 1/n$, where n is the number of variables. For HS, we used the most commonly used parameter settings, which are Harmony Memory Size (HMS) of 5, *HMCR* of 0.9, and *PAR* of 0.4 [10].

In this experiment, we address the following research questions:

RQ1: Is there any impact of using the two normalization functions on the performance of a search algorithm?

RQ2: What are the differences in terms of the performance of the algorithms when using *N1* and *N2*?

To compare the algorithms for *N1* and *N2* (RQ1), we followed the guidelines defined in [11, 14], which recommends a number of statistical procedures to assess randomized test strategies. First, we calculated the success rate for each algorithm, which is defined as the number of times that a solution was found out of the total number of runs (100 in this case). These success rates were then compared using the Fisher Exact test (with a significance level of 0.05), quantifying the effect size using an odds ratio with a 0.5 correction. We chose the Fisher Exact test since for each run of algorithms the result is binary, i.e., either the result is 'found' or 'not found'. This is exactly the condition for which the Fisher's exact test is defined. In addition to statistical significance, we used odds ratio as the results of our experiments are dichotomous. A value greater than 1 means that *N1* has more chance of success as compared to *N2*, whereas a value of 1 means no difference.

When the difference between the success rates of N1 and N2 were not significant for an algorithm, we further compared the number of iterations taken by the algorithm for N1 and N2 to solve the problems. For this purpose, we used Mann-Whitney U-test [10] at a significance level of 0.05. In addition, we report effect size measurements using Vargha and Delaney's Â12 statistics, which is a non-parametric effect size measure. In our context, the value of Â12 tells the probability for N1 to find a solution in less number of iterations than N2. This means that higher the value of Â12 than 0.5, the higher the chance that N1 will take lesser iterations to find a solution than N2. If N1 and N2 are equal then the value of Â12 is 0.5. In Table 1-Table 5, results for Mann-Whitney U-test and Â12 are only shown when the differences are not significant based on success rates. For RQ2, we used the same statistics except that instead of N1 and N2, we compared two algorithms.

Table 1. Results for *N1* vs. *N2* for the Five Algorithms*

Algo	P#	Success Rate				# Iterations		
		S(N1)	S(N2)	OR	p-value	MD	Â12	p-value
AVM	1	0.34	0	104	***<0.0001***	_	_	_
	2	0.53	0	226	***<0.0001***	_	_	_
	3	1	0	40401	***<0.0001***	_	_	_
	4	1	0	40401	***<0.0001***	_	_	_
	5	1	0	40401	***<0.0001***	_	_	_
	6	1	1	1	1	-11	0.495	0.98
	7	1	0	40401	***<0.0001***	_	_	_
	8	1	0	40401	***<0.0001***	_	_	_
	Avg.	0.859	0.125	_	_	_	_	_
HS	1	1	1	1	1	-114	0.450	0.97
	2	1	1	1	1	-18	0.497	0.71
	3	1	1	1	1	39	0.513	0.99
	4	1	1	1	1	-17	0.495	0.95
	5	1	1	1	1	81	0.498	0.80
	6	1	1	1	1	0	0.5	1
	7	1	1	1	1	37	0.515	0.83
	8	1	1	1	1	218	0.539	0.25
	Avg.	1	1	_	_	_	_	_
1+1(EA)	1	1	0.82	45	***<0.0001***	_	_	_
	2	1	0.24	628	***<0.0001***	_	_	_
	3	0.97	0.31	61	***<0.0001***	_	_	_
	4	1	0.27	537	***<0.0001***	_	_	_
	5	1	0.22	701	***<0.0001***	_	_	_
	6	1	1	1	1	0	0.50	1
	7	1	0.39	313	***<0.0001***	_	_	_
	8	1	0.8	51	***<0.0001***	_	_	_
	Avg.	0.996	0.506	_	_	_	_	_
RS	1	0	0	_	_	_	_	_
	2	0.44	0.98	0.02	***<0.0001***	_	_	_
	3	0.29	0.27	1.10	0.87	118	0.567	0.05
	4	0.32	0.39	0.74	0.37	15	0.508	0.79
	5	0	0	_	_	_	_	_
	6	1	1	1	1	0	0.50	1
	7	0.41	0.31	2	0.18	21	0.52	0.82
	8	0	0	1	1	_	_	_
	Avg.	0.308	0.369	_	_	_	_	_
GA	1	1	0.89	26	***<0.0001***	_	_	_
	2	1	1	1	1	37	0.51	0.92
	3	1	1	1	1	-29	0.48	0.98
	4	1	1	1	1	30	0.50	0.99
	5	0.55	0.14	7	***<0.0001***	_	_	_
	6	1	1	1	1	0	0.50	1
	7	1	1	1	1	-75	0.48	0.60
	8	1	0.89	26	***<0.0001***	_	_	_
	Avg.	0.944	0.865	_	_	_	_	_

*S(N1), Success rate with *N1*, OR: Odds Ratio, MD: Mean Difference *(N1-N2)*

3.2 Experiment Execution

We executed the five algorithms 100 times with both *N1* and *N2* for the eight problems. We let the algorithms run up to 20,000 fitness evaluations on each problem and collected data on whether the algorithms found solutions for *N1* and *N2*. We used a PC with Intel Core Duo CPU 2.20 GHz with 4 GB of RAM, running Microsoft Windows 7 operating system for the execution of experiment.

3.3 Experiment Results and Analysis

In this section, we will answer each of our research questions.

RQ1: Is there any impact of using the two normalization functions on the performance of a search algorithm? For AVM, we observed that when using *N2* for all the problems the success rate is 0% (Table 1) except for P6, for which both *N1* and *N2* achieved 100% success rates**.** For P6, we further compared *N1* and *N2* based on the number of iterations to solve the problem. The results in Table 1 show that *N1* took less number of iterations as *MD* is -11 but no significant difference identified since Â12 value is 0.495 and *p*-value is 0.98.

For HS, there were no significant differences in using *N1* and *N2* in terms of both success rates and number of iterations as all the *p*-values are greater than 0.05.

For (1+1) EA, *N1* is significantly better than *N2* except for P6 in terms of success rates since all *p*-values are less than 0.05 and Odds Ratio (*OR*) values are greater than 1. For P6, there was no significant difference between *N1* and *N2* in terms of number of iterations (*MD*=0, $\hat{A}12$=0.5, and *p*-value=1).

For RS, there was no significant difference between *N1* and *N2* except for P6. Overall success rates are very low, i.e., 31% for *N1* and 37% for *N2*. For P6, both *N1* and *N2* achieved 100% success rates and in terms of the number of iterations there was no significant difference between them.

For GA, for P1, P5, and P8, *N1* is significantly better than *N2* in terms of success rates. For the rest, there was no significant difference between *N1* and *N2* in terms of success rates and the number of iterations.

Based on the above results, we can answer RQ1 as follows: Using *N1* significantly improves the performance of AVM and (1+1) EA. For HS using *N1* or *N2* doesn't have any impact on its performance. For GA, *N1* either improves the effectiveness (3 out of 8 problems) or has no significant difference than *N2*.

We observe that for P6 all the algorithms with both *N1* and *N2* achieved 100% success rates since the problem was the easiest one as it can be seen from the fact even RS has 100% success rate for P6.

RQ2: What are the differences in terms of the performance of the algorithms when using *N1* and *N2*? In this section, we will compare each pair of the algorithms with *N1* and *N2*.

AVM vs HS. When *N1* was applied, HS is significantly better than AVM in terms of success rates for P1 and P2 (Table 2) since the *OR* values are less than 1 and *p*-values are less than 0.05. However, there is no significant difference for the rest of the problems and thus we further compared these two algorithms in terms of the number of iterations (Table 3). Except for P6 and P8, *N1* took significantly less number of iterations as $\hat{A}12$ values are less than 0.5 and *p*-values less than 0.05 (Table 3). For P6, there is no significant difference between the two algorithms and for P8 HS took significantly more iterations than AVM ($\hat{A}12$=0.81 and *p*-value<0.05 as shown in Table 3). For P3, *N1* took lesser number of iterations than *N2*; however the differences are not significant (Table 3). We can observe from Table 4 and Table 5 that for *N2*, HS is significantly better than AVM for all the problems except P6, where there is no significant difference identified between the two algorithms both in terms of success rates and number of iterations.

AVM vs. (1+1) EA. With *N1*, there is no significant difference for all the problems except for P1 and P2, where (1+1) EA is significantly better than AVM in terms of success rates (Table 2). As one can observe from Table 3, there is no significant difference for P3, P6, and P7; For P4 and P5, (1+1) EA took significantly more iterations than AVM; For P8, AVM took significantly more iterations than (1+1) EA. As shown in Table 4 and Table 5, with *N2*, (1+1) EA is significantly better than AVM for all the problems except for P6, where there is no significant difference between them both in terms of success rates and the number of iterations.

AVM vs. RS. With *N1*, except for P2 and P6, AVM is significantly better than RS (Table 2). For P2 and P6, in terms of the number of iterations there is no significant difference between the two algorithms (Table 3). For *N2*, RS achieved significantly higher success rates for P2, P3, P4, and P7, whereas for P1, P5, and P8 both of the algorithms have 0% success rates (Table 4). For P6, there is no significant difference in terms of success rates and number of iterations (Table 4 and Table 5).

AVM vs. GA. With *N1*, for P1, P2, and P5, GA achieved significantly higher success rates than AVM. For the rest, there is no significant difference in terms of success rates (Table 2). For these problems, GA took significantly more iterations than AVM for P4 and P5, whereas for P8 AVM took significantly more iterations than GA (Table 3). There is no significant difference between GA and AVM for P3, P6, and P7 (Table 3). With *N2*, GA achieved significantly higher success rates than AVM except for P6, for which, as for other algorithms, we didn't observe any significant difference in terms of both success rates and the number of iterations (Table 4 and Table 5).

HS vs. (1+1) EA. With *N1*, there is no significant difference in terms of success rates (Table 2). However, for P1, P5, and P8, HS took significantly less iterations than (1+1) EA (Table 3). For P2, P3, P6, and P7, no significant difference was observed between the two algorithms (Table 3). For P4, (1+1) EA took significantly less

iterations than HS as shown in Table 3. With *N2*, HS achieved significantly higher success rates for all the problems except for P6 (Table 4 and Table 5).

HS vs. RS. With *N1,* HS achieved significantly higher success rates than RS except for P6 (Table 2). With *N2*, HS obtained significantly higher success rates than RS except for P2 and P6 (Table 4).

HS vs. GA. With *N1*, there is no significant difference between the two algorithms except for P5, where HS achieved significantly higher success rates than GA (Table 2). For the rest of the problems (excluding P5), no significant difference was observed in terms of the number of iterations except for P2 where HS took significantly less number of iterations than GA (Table 3). With *N2*, for P1, P5, and P8, HS achieved significant higher success rates than GA. For the rest of the problems, where there was no significant difference in terms of success rates, we didn't even observe difference in terms of the number of iterations (Table 4 and Table 5).

(1+1) EA vs. RS. With *N1*, (1+1) EA is significantly better than RS except for P6, where there is no significant difference in terms of both success rates and iterations (Table 2 and Table 3). With *N2* for P1, P5, and P8, (1+1) EA achieved significantly higher success rates than RS. For P2, RS obtained higher success rates than (1+1) EA. For the rest, there is no significant difference in terms of both success rates and the number of iterations (Table 4 and Table 5).

(1+1) EA vs. GA. With *N1*, there is no significant difference for all the problems except for P5, where (1+1) EA achieved significantly higher success rates (Table 2). For P2 and P4, GA took significantly less iterations and vice versa for P1 (Table 3). For the rest of the problems, there is no significant difference as shown in Table 3. With *N2*, for P2, P3, P4, and P7, GA achieved significant higher success rates than (1+1) EA and for the rest of the problems, there is no significant difference in terms of both success rates and the number of iterations (Table 4 and Table 5).

RS vs. GA. With *N1*, GA is significantly better than RS in terms of success rates except for P6 (Table 2 and Table 3). With *N2*, GA is significantly better than RS in terms of success rates except for P2 and P6, where there were no significant differences in terms of success rates and iterations (Table 4 and Table 5).

Overall Conclusion. Based on the results from RQ1 and RQ2, for HS and RS, we observed that there was no significant difference in using any of the two normalization functions. However, for HS, the success rates with *N1* and *N2* were both 100%, whereas for RS, the success rates were 31% and 37% respectively. One possible explanation of this phenomenon may be due to the extent of the randomness consideration in both of the algorithms in creating a new solution. In case of RS, solutions are

created randomly and thus the effect of the normalization functions on performance is masked since the probability of the randomness for RS is always 1. Similarly for HS, there is always (*1- HMCR*), i.e., 0.1 probability in our context to pick a random value (Section 2.3 and Section 3.3). Considering there are *n* decision variables (n=10 in our case as we discussed in Section 3.1), there is roughly a probability of $(1\text{-}HMCR)^n$ randomness in a solution, which in our context equals to 0.1^{10} plus randomness due to *HMS* random solutions in *HM*. Another possible explanation could be that the excellent performance of HS masked the effect of normalization: it achieved 100% success rates for all the problems with no significant difference in terms of the number of iterations (Table 1). In case of more complicated problems, it might be possible to observe performance difference, when using different normalization functions. We will investigate this in the future with more focused experiments. Similarly for RS, its performance was consistently low for all the problems and thus difference when applying the two normalization functions was not visible.

For (1+1) EA (see [4] for detailed description of how the algorithm works), randomness is only considered in the mutation operator, which mutates each variable in the solution with a probability *p*. Suppose that we have *n* variables (n=10), then the total amount of randomness is roughly: $(1/n)^n$ (0.1^{10} in our case). This value is lower than the one for RS and HS since HS also introduces more randomness because of random solutions in HM (Section 2.3). In case of AVM (see [4] for the detailed description of the algorithm), considering that there are *n* decision variables, the extent of the randomness consideration in each solution can be roughly calculated as: $\frac{n-1}{n} * \frac{n-2}{n} * \ldots * \frac{1}{n} * p^n$ ($\frac{9!}{10^{10}} * 0.1^{10}$ in our context). This value is again less as compared with the randomness consideration in RS and HS. In summary, RS and HS have stronger randomness consideration than (1+1) EA and AVM, which might explain why HS and RS were not affected when using different normalization functions. However, to further understand this phenomenon, more focused experiments are required to test the algorithms combined with the normalization functions when given various values to parameters (e.g., *p*). This will be our future work.

As reported in [5, 6] for GA, when using *N1*, it achieved significantly better performance than N2. In our context, we however observed partially consistent results as what was reported in [5, 6]: as for 3 out of 8 problems (Table 1), *N1* showed significantly better performance than *N2* and for the rest, no difference was observed. In GA as we discussed in Section 3.1, the probability of randomness is roughly equivalent to p^n due to mutation. Considering n=10 in our context, we have a probability of 0.1^{10}. Notice that this is a rough calculation of the probability of randomness since we didn't account for randomness due to crossover. Since this probability is somewhat similar to the probability of randomness in HS, we observed similar results as HS since for 5 problems there was no significant difference, when using any normalization function in case of GA. However the performance of HS is significantly better than GA (Section 3.3). This can be explained based on how the algorithms work as discussed in [15]. The main difference between GA and HS is that HS considers all existing solutions when generating a new solution, while GA considers only two solutions (parents). This feature increases the flexibility of HS and therefore produces better solutions as we observed from our experiment results.

Based on the above discussion, we can conclude that the extent of the consideration of randomness in an algorithm may affect the use of a particular normalization function. This means that higher the randomness consideration in an algorithm in generating new solutions, higher the chance that the performance of the algorithm will not be impacted by the use of any particular normalization function. Based on these observations, we recommend using HS with any of the two normalization functions for solving OCL constraints to support MBT, considering that HS achieved 100% success rates for all the problems and was not affected by the choice of any of the two normalization functions.

4 Threats to Validity

To reduce construct validity threats, we chose an effectiveness measure: the search success rate, which is comparable across all the five search algorithms (AVM, (1+1) EA, HS, GA and RS). Furthermore, we used the same stopping criterion for all the algorithms, i.e., the number of fitness evaluations. This criterion is a comparable measure of efficiency across all the algorithms.

The most probable conclusion validity threat in experiments involving randomized algorithms is due to random variations. To address it, we repeated experiments 100 times to reduce the possibility that results were obtained by chance. Furthermore, we performed Fisher exact tests to compare proportions and determine the statistical significance of the results. We chose this test since it is appropriate for dichotomous data where proportions must be compared [14], thus matching our situation. To determine the practical significance of the results obtained, we measured the effect size using the odds ratio of success rates across the search techniques.

A possible threat to internal validity is that we have experimented with only one configuration setting for the (1+1) EA, AVM, HS and GA parameters. However, these settings are in accordance with the common guidelines in the literature and our previous experience on testing problems. Another threat is the use of fixed values for α and β in the normalization functions. Recall that to deal with this, we used the most commonly used values for α and β reported in the literature.

5 Related Work

To efficiently generate test data for OCL constraints, we previously defined in [4] novel heuristics based on branch distance [8] for various OCL constructs and operations to guide search algorithms including GA, (1+1) EA, and AVM. This is the only work in the literature that explores the use of search algorithms for solving OCL constraints as we comprehensively compared in [4].

Only related work to the empirical evaluation presented in this paper is reported in [5, 6], where the two commonly used normalization functions were studied for the first time. In this paper, we further enhanced the empirical evaluation reported in [5, 6]. The differences of these two works are as follows: 1) The work reported in this paper empirically evaluates the two normalization functions on model-level con-

straints as opposed to branches in code; 2) This work evaluates three other search algorithms, i.e., (1+1) EA, AVM and HS. The results of the empirical study reported in this paper are partially consistent with the results reported in [5, 6]; however for HS we observed that the choice of a normalization function has no impact on its performance.

6 Conclusion

Model-based testing (MBT) offers an automated and systematic alternative to manual testing and has shown promising results both in industry and academia. MBT requires developing a model of system under test with constraints to support automation of e.g., test data generation. In our previous work, we developed a search-based test generator (EsOCL) from the Object Constraint Language (OCL) to solve constraints on UML state machines to support automated MBT. We defined various heuristics to guide search algorithms to efficiently solve OCL constraints to generate test data. Several of these heuristics required using one of the commonly used normalization functions in the literature. An existing work has evaluated two commonly used normalization functions in search-based software engineering for various search algorithms. In this paper, we further extend this work with more empirical evaluations.

More specifically, in this paper, we reported an empirical evaluation of two commonly used normalization functions in the literature, along with five search algorithms, in the context of solving OCL constraints for supporting automated test data generation at the model level. In contrast, the only related work in the literature evaluates the same normalization functions in the context of program branches at the code-level. In this paper, we evaluated in total five algorithms: Random Search (RS), Genetic Algorithm (GA), (1+1) Evolutionary Algorithm, Alternating Variable Method (AVM) and Harmony Search (HS). We observed that our evaluation results are mostly consistent with what was already reported in the existing work except for HS, for which we observed that the choice of the two normalization functions doesn't affect its performance. We also provided plausible explanation for this observation: the extent of the consideration of randomness in an algorithm might affect the effect of a normalization function on the performance of a search algorithm: higher randomness consideration leads to higher chance that the performance of an algorithm will not be impacted by the use of any particular normalization function. The results also show that HS achieved 100% success rates for all the problems and therefore we recommend HS for solve OCL constraints at the model-level for generating test data, when combined with any of the two normalization functions.

References

1. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann (2006)
2. Chow, T.S.: Testing Software Design Modeled by Finite-State Machines. IEEE Transactions on Software Engineering 4, 178–187 (1978)

3. Iqbal, M.Z., Arcuri, A., Briand, L.: Environment Modeling with UML/MARTE to Support Black-Box System Testing for Real-Time Embedded Systems: Methodology and Industrial Case Studies. In: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 286–300. Springer. Heidelberg (Year)
4. Ali, S., Iqbal, M.Z., Arcuri, A., Briand, L.: Generating Test Data from OCL Constraints with Search Techniques. IEEE Transactions on Software Engineering (2013)
5. Arcuri, A.: It Does Matter How You Normalise the Branch Distance in Search Based Software Testing. In: Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation. IEEE Computer Society (2010)
6. Arcuri, A.: It really does matter how you normalize the branch distance in search-based software testing. Software Testing, Verification and Reliability (2011)
7. Zou, D., Gao, L., Wu, J., Li, S., Li, Y.: A novel global harmony search algorithm for reliability problems. Comput. Ind. Eng. 58, 307–316 (2010)
8. McMinn, P.: Search-based software test data generation: a survey: Research Articles. Softw. Test. Verif. Reliab. 14, 105–156 (2004)
9. Harman, M., Mansouri, A., Zhang, S., Search, Y.: based software engineering: A comprehensive analysis and review of trends techniques and applications. King's College,Technical Report TR-09-03 (2009)
10. Geem, Z.W.: Music-Inspired Harmony Search Algorithm: Theory and Applications. Springer Publishing Company (2009) (Incorporated)
11. Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K.: A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation. IEEE Transactions on Software Engineering 99 (2009)
12. Arcuri, A., Iqbal, M.Z., Briand, L.: Black-box system testing of real-time embedded systems using random and search-based testing. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 95–110. Springer, Heidelberg (2010)
13. Fraser, G., Arcuri, A.: Whole Test Suite Generation. IEEE Transactions on Software Engineering (2012)
14. Arcuri, A., Briand., L.: A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In: International Conference on Software Engineering, ICSE (2011)
15. Alia, O.M.D., Mandava, R.: The variants of the harmony search algorithm: an overview. Artif. Intell. Rev. 36, 49–68 (2011)

Appendix A

Table 2. Results for Success Rates for N1 for Each Pair of Algorithms

P#	AVM	HS	1+1(EA)	RS	GA	AVM vs HS		AVM vs 1+1(EA)		AVM vs RS		AVM vs GA		HS vs 1+1(EA)		HS vs RS		HS vs GA		1+1(EA) vs RS		1+1(EA) vs GA		RS vs GA	
						OR	p-value	OR	p-value	OR	p-value	OR	p-value	OR	p-value	OR	p-value	OR	p-value	OR	p-value	OR	p-value	OR	p-value
1	0.34	1	1	0	1	0.002	***<0.0001***	0.002	***<0.0001***	104	***<0.0001***	0.002	***<0.0001***	1	1	40401	***<0.0001***	1	1	40401	***<0.0001***	1	1	<0.0001	***<0.0001***
2	0.53	1	1	0.44	1	0.005	***<0.0001***	0.005	***<0.0001***	1	0.25	0.005	***<0.0001***	1	1	255	***<0.0001***	1	1	255	***<0.0001***	1	1	0.003	***<0.0001***
3	1	1	0.97	0.29	1	1	1	7	0.24	487	***<0.0001***	1	1	7	0.24	487	***<0.0001***	1	1	67	***<0.0001***	0.13	0.24	0.002	***<0.0001***
4	1	1	1	0.32	1	1	1	1	1	423	***<0.0001***	1	1	1	1	423	***<0.0001***	1	1	423	***<0.0001***	1	1	0.002	***<0.0001***
5	1	1	1	0	0.55	1	1	1	1	40401	***<0.0001***	164	***<0.0001***	1	1	40401	***<0.0001***	164	***<0.0001***	40401	***<0.0001***	164	***<0.0001***	0.004	***<0.0001***
6	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
7	1	1	1	0.41	1	1	1	1	1	288	***<0.0001***	1	1	1	1	288	***<0.0001***	1	1	288	***<0.0001***	1	1	0.003	***<0.0001***
8	1	1	1	0	1	1	1	1	1	40401	***<0.0001***	1	1	1	1	40401	***<0.0001***	1	1	40401	***<0.0001***	1	1	<0.0001	***<0.0001***
Avg.	0.86	1	0.9963	0.31	0.94	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_

Table 3. Results for the Number of Iterations for N1 for Each Pair of Algorithms

P#	AVM	HS	1+1(EA)	RS	GA	AVM vs HS		AVM vs 1+1(EA)		AVM vs RS		AVM vs GA		HS vs 1+1(EA)		HS vs RS		HS vs GA		1+1(EA) vs RS		1+1(EA) vs GA		RS vs GA	
						A12	p-value	A12	p-value	A12	p-value	A12	p-value	A12	p-value	A12	p-value	A12	p-value	A12	p-value	A12	p-value	A12	p-value
1	0.34	1	1	0	1	_	_	_	_	_	_	_	_	0.28	***<0.0001***	_	_	0.49	0.69	_	_	0.62	***<0.0001***	_	_
2	0.53	1	1	0.44	1	_	_	_	_	0.56	0.84	_	_	0.57	0.09	_	_	0.42	***0.02***	_	_	0.4	***<0.0001***	_	_
3	1	1	0.97	0.29	1	0.45	0.29	0.58	0.27	_	_	0.48	0.56	0.56	0.05	_	_	0.52	0.62	_	_	0.43	0.13	_	_
4	1	1	1	0.32	1	0.24	***<0.0001***	0.41	***0.01***	_	_	0.19	***<0.0001***	0.67	***<0.0001***	_	_	0.51	0.73	_	_	0.27	***<0.0001***	_	_
5	1	1	1	0	0.55	0.21	***<0.0001***	0.32	***<0.0001***	_	_	_	_	0.43	***0.03***	_	_	_	_	_	_	_	_	_	_
6	1	1	1	1	1	1	***<0.0001***	0.5	1	0.5	1	0.5	1	0.5	1	0.5	1	0.5	1	0.5	1	0.5	1	0.5	1
7	1	1	1	0.41	1	0.4	***0.02***	0.41	0.1	_	_	0.38	0.18	0.52	0.34	_	_	0.55	0.23	_	_	0.47	0.93	_	_
8	1	1	1	0	1	0.81	***<0.0001***	0.71	***<0.0001***	_	_	0.72	***<0.0001***	0.33	***0.001***	_	_	0.46	0.63	_	_	0.57	***0.005***	_	_
Avg.	0.86	1	0.9963	0.31	0.94	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_

Table 4. Results for Success Rates for N2 for Each Pair of Algorithms

P#	AVM	HS	1+1(EA)	RS	GA	AVM vs HS		AVM vs 1+1(EA)		AVM vs RS		AVM vs GA		HS vs 1+1(EA)		HS vs RS		HS vs GA		1+1(EA) vs RS		1+1(EA) vs GA		RS vs GA	
						A12	p-value	A12	p-value	A12	p-value	A12	p-value	A12	p-value	A12	p-value	A12	p-value	A12	p-value	A12	p-value	A12	p-value
1	0	1	0.82	0	0.89	-	-	-	-	-	-	-	-	0.55	0.15	-	-	-	-	-	-	0.6	0.05	-	-
2	0	1	0.24	0.98	1	-	-	-	-	-	-	-	-	0.6	***0.001***	0.49	0.88	0.46	0.23	-	-	-	-	0.45	0.18
3	0	1	0.31	0.27	1	-	-	-	-	-	-	-	-	0.6	***0.002***	-	-	0.49	0.88	0.46	0.33	-	-	-	-
4	0	1	0.27	0.39	1	-	-	-	-	-	-	-	-	0.64	***0.001***	-	-	0.52	0.32	0.5	0.76	-	-	-	-
5	0	1	0.22	0	0.14	-	-	-	-	-	-	-	-	0.45	0.94	-	-	-	-	-	-	0.48	0.32	-	-
6	1	1	1	1	1	0.5	1	0.5	1	0.5	1	0.5	1	0.5	1	0.5	1	0.5	1	0.5	1	0.5	1	0.5	1
7	0	1	0.39	0.31	1	-	-	-	-	-	-	-	-	0.61	***0.01***	-	-	0.51	0.93	0.5	0.73	-	-	-	-
8	0	1	0.8	0	0.89	-	-	-	-	-	-	-	-	0.54	0.22	-	-	-	-	-	-	0.57	0.11	-	-
Avg.	0.13	1	0.50625	0.37	0.87	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 5. Results for the Number of Iterations for N2 for Each Pair of Algorithms

P#	AVM	HS	1+1(EA)	RS	GA	AVM vs HS		AVM vs 1+1(EA)		AVM vs RS		AVM vs GA		HS vs 1+1(EA)		HS vs RS		HS vs GA		1+1(EA) vs RS		1+1(EA) vs GA		RS vs GA	
						OR	p-value	OR	p-value	OR	p-value	OR	p-value	OR	p-value	OR	p-value	OR	p-value	OR	p-value	OR	p-value	OR	p-value
1	0	1	0.82	0	0.89	<0.0001	***<0.0001***	0.001	***<0.0001***	1	1	0.0006	***<0.0001***	45	***<0.0001***	40401	***<0.0001***	25	***<0.0001***	896	***<0.0001***	0.57	0.22	0.0006	***<0.0001***
2	0	1	0.24	0.98	1	<0.0001	***<0.0001***	0.01	***<0.0001***	0.0001	***<0.0001***	<0.0001	***<0.0001***	627	***<0.0001***	5	0.49	1	1	0.008	***<0.0001***	0.001	***<0.0001***	0.19	0.49
3	0	1	0.31	0.27	1	<0.0001	***<0.0001***	0.01	***<0.0001***	0.01	***<0.0001***	<0.0001	***<0.0001***	443	***<0.0001***	537	***<0.0001***	1	1	1.2	0.64	0.002	***<0.0001***	0.001	***<0.0001***
4	0	1	0.27	0.39	1	<0.0001	***<0.0001***	0.01	***<0.0001***	0.007	***<0.0001***	<0.0001	***<0.0001***	537	***<0.0001***	312	***<0.0001***	1	1	0.58	0.09	0.001	***<0.0001***	0.003	***<0.0001***
5	0	1	0.22	0	0.14	<0.0001	***<0.0001***	0.01	***<0.0001***	1	1	<0.0001	***<0.0001***	701	***<0.0001***	40401	***<0.0001***	1199	***<0.0001***	57	***<0.0001***	1.7	0.19	0.02	***<0.0001***
6	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
7	0	1	0.39	0.31	1	<0.0001	***<0.0001***	0.007	***<0.0001***	0.01	***<0.0001***	<0.0001	***<0.0001***	312	***<0.0001***	443	***<0.0001***	1	1	1.4	0.29	0.003	***<0.0001***	0.002	***<0.0001***
8	0	1	0.8	0	0.89	<0.0001	***<0.0001***	0.001	***<0.0001***	1	1	0.0006	***<0.0001***	51	***<0.0001***	40401	***<0.0001***	25	***<0.0001***	789	***<0.0001***	0.5	0.11	0.0006	***<0.0001***
Avg.	0.13	1	0.5063	0.37	0.87	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 6. Artificial Problems Used in the Experiment

P#	Example
1	let c = Set{0,1,2,3} in X.allInstances() → select(b|b.y>0 and b.y<5) → size()>=5 and X.allInstances() → select(b|b.y>0 and b.y<5) → collect(b|b.y) → excludesAll(c)
2	X.allInstances() → select(b|b.y > 90) → size() > 4 and X.allInstances() → select(b|b.y > 90) → exists(b|b.y=92)
3	X.allInstances() → select(b|b.y > 90) → size() > 4 and X.allInstances() → select(b|b.y > 90) → isUnique(b|b.y)
4	X.allInstances() → select(b|b.y > 90) → size() > 4 and X.allInstances() → select(b|b.y > 90) → one(b|b.y=95)
5	X.allInstances() → select(b|b.y=0) → size()>6
6	X.allInstances() → select(b|b.y=0) → size()<=1
7	X.allInstances() → select(b|b.y > 90) → size() > 4 and X.allInstances() → select(b|b.y > 90) → select(b|b.y=92) → size() <> 0
8	X.allInstances() → select(b|b.y=0) → size() = 5