# Improved Heuristics for Solving OCL Constraints using Search Algorithms

Shaukat Ali
Simula Research Laboratory
Norway
shaukat@simula.no

Muhammad Zohaib Iqbal
Quest Lab, National University of Computer & Emerging Sciences, Pakistan and
SnT Luxembourg, Luxembourg
zohaib.iqbal@nu.edu.pk

Andrea Arcuri
Simula Research Laboratory
Norway
arcuri@simula.no

## ABSTRACT

The Object Constraint Language (OCL) is a standard language for specifying constraints on Unified Modeling Language (UML) models. The specified constraints can be used for various purposes including verification, and model-based testing (e.g., test data generation). Efficiently solving OCL constraints is one of the key requirements for the practical use of OCL. In this paper, we propose an improvement in existing heuristics to solve OCL constraints using search algorithms. We evaluate our improved heuristics using two empirical studies with three search algorithms: Alternating Variable Method (AVM), (1+1) Evolutionary Algorithm (EA), and a Genetic Algorithm (GA). We also used Random Search (RS) as a comparison baseline. The first empirical study was conducted using carefully designed artificial problems (constraints) to assess each individual heuristics. The second empirical study is based on an industrial case study provided by Cisco about model-based testing of Video Conferencing Systems. The results of both empirical evaluations reveal that the effectiveness of the search algorithms, measured in terms of time to solve the OCL constraints to generate data, is significantly improved when using the novel heuristics presented in this paper. In particular, our experiments show that (1+1) EA with the novel heuristics has the highest success rate among all the analyzed algorithms, as it requires the least number of iterations to solve constraints.

## Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

## General Terms

Algorithms, Performance, Experimentation, Theory, Verification

## Keywords

UML; OCL; Search-based testing; Test data; Empirical evaluation

## 1. INTRODUCTION

The Object Constraint Language (OCL) [1] is a standard language to specify constraints on Unified Modeling Language (UML) models such as Class Diagrams and State Machines [2]. There are several purposes for writing constraints on the UML models, such as to give precise meaning to models and support automation. The constraints specified on the UML models may require to be solved for various applications, for example proving properties of a system and test data generation to support model-based testing. Solving complex industrial OCL constraints can be difficult due to a large number of clauses in the constraints, which might include specialized OCL operations such as *oclInState()*, *includesAll*, *forAll()* and a large number of attributes or collections of attributes used in those constraints.

To efficiently solve OCL constraints, a set of heuristics were proposed in [3, 4] based on branch distance [5] for various OCL constructs and operations to guide search algorithms including Genetic Algorithm (GA), (1+1) Evolutionary Algorithm (EA), and Alternating Variable Method (AVM). In this paper, we propose novel, improved heuristics for solving constraints on Boolean operations defined in OCL including *and*, *or*, *xor*, and *implies*. The aim is to further improve the performance of search algorithms in order to efficiently solve constraints. Our motivating assumption is that instead of solving a complicated constraint consisting of several clauses all together, we can improve the performance by solving independent clauses separately and, then, combining the results of the independent clauses to solve the complicated constraint.

We empirically evaluate the performance of the improved heuristics by conducting two sets of experiments. The first set of experiments compares the improved heuristics with existing heuristics using 28 artificial problems (constraints) carefully designed to evaluate each heuristic. The results showed that the improved heuristics significantly improve the effectiveness of all the algorithms to solve the constraints. However, GA and RS showed worse performance when compared to (1+1) EA and AVM for both improved and existing heuristics. The second set of experiments was performed on a real industrial case study of model-based robustness testing of a Video Conferencing System developed by Cisco. We evaluated our improved heuristics on 42 constraints from the industrial case study using the three search algorithms and RS. The results were consistent with the results from the artificial problems, where the novel heuristics significantly improved the performance of the search algorithms. Overall, our experiments revealed that (1+1) EA with the novel heuristics was the most efficient among all the studied algorithms.

The rest of the paper is organized as follows: Section 2 provides background information, Section 3 describes our novel heuristics, whereas Section 4 and Section 5 present the results of two empirical evaluations. Section 6 provides overall discussion on the empirical evaluations followed by describing threats to validity in Section 7.

*GECCO'14*, July 12–16, 2014, Vancouver, BC, Canada.
Copyright © 2014 ACM 978-1-4503-2662-9/14/07...$15.00.
http://dx.doi.org/10.1145/2576768.2598308

We discuss related work in Section 8 and finally we conclude the paper in Section 9.

## 2. BACKGROUND

In [1, 6], the authors 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 set of heuristics were defined to guide a search algorithm to find test data that satisfy OCL constraints. The heuristics were implemented as a fitness function that is adapted from work done for code coverage (e.g., for branch coverage in C code [5]). In particular, branch distance (a function *d()*) was used as defined in [5]. The function *d()* returns *0* if the constraint is solved, a value *k* if the constraint evaluates to undefined (a special value in OCL that makes OCL a three-valued logic [1]) otherwise a positive value greater than 0 and less that *k* that heuristically estimates how far the constraint was from being evaluated to true. The work reported in [1, 6] defined various novel heuristics for calculating branch distances for various constructs and operations defined in OCL such as *includes()*, *oclInState()*, and *forAll()*.

The operations defined in OCL to combine *Boolean* clauses are *or*, *xor*, *and*, *not*, *if then else*, and *implies*. For these operations, branch distances were extended (Table I) from the ones proposed in [5], since the ones proposed in [5] work only in programming languages with two-valued Boolean logic. The operations *implies* and *xor* are syntactic sugars that usually do not appear in programming languages such as C and Java, and can be re-expressed using combinations of *and* and *or* operators. The evaluation of *d()* on a constraint composed by two clauses is specified in Table I and can simply be computed for more than two clauses recursively. In this paper, we provide improvement in calculating branch distance for these operations as we will discuss in the next section.

Table I. Branch Distance Calculation for OCL's Boolean Operations

| Operation | Distance function |
|---|---|
| A | **if** A is **true**<br>    **then** d(A)=0<br>**else** A is **false**<br>    **then** d(A)>0 and d(A)< k<br>    **else**<br>    **then** d(A)=k |
| not A | **if** A is **false**<br>    **then** d(A)=0<br>**else** A is **true**<br>    **then** d(A)>0 and d(A)< k<br>**else**<br>    **then** d(A)=k |
| A and B | numberOfUndefinedClauses + nor (d(A)+d(B)) |
| A or B | numberOfUndefinedClauses + nor (min (d(A),d(B))) |
| A implies B | d(not A or B) |
| if A then B else C | d((A and B) or (not A and C)) |
| A xor B | d((A and not B) or (not A and B)) |

## 3. IMPROVED BRANCH DISTANCE CALCULATION FOR LOGICAL OPERATIONS

This section presents our improved branch distance calculation for the *Boolean* operations defined in OCL. As mentioned in Section 2, in total there are six operations in OCL for *Boolean* clauses. Among them, *and, or, xor,* and *implies* are binary operations and therefore can be used to combine different constraints. Our main assumption behind defining the branch distance for these operations is that, by dividing a complex constraint into a set of small independent constraints and then solving each of them individually, we can bring significant improvement over solving the whole constraint at the same time. By independent constraint, we mean that the variables used in one constraint are not used in the other constraints belonging to the main complex constraint. Mathematically, we can say that a complicated constraint *C* consisting of *n* clauses combined using any of the *Boolean* operations (e.g., *or, and*) can be divided into a set *CI* of independent constraints, where $CI= \{CI_1, CI_2, ..CI_m\}$, where $m<=n$. Each $CI_i$ is independent, if all variables *v* involved in $CI_i$ are not used in any variable involved in $CI_x$, where *x* is from *1* to *m* excluding *i*. Below, we provide improved branch distance calculations for each of the *Boolean* operations defined in OCL. In this section, we show calculation of branch distance on a constraint consisting of two clauses and separated by a *Boolean* operation. However, this calculation can simply be computed for more than two clauses joined using different *Boolean* operations recursively.

### 3.1 Improved-and Branch Distance Calculation

In the case of *and*, since each $CI_i$ is conjuncted, each $CI_i$ must be solved to obtain the overall result for the complicated constraint *C*. If any $CI_i$ was not solved, then this means that *C* was not solved successfully.

### 3.2 Improved-or Branch Distance Calculation

Each $CI_i$ in this case is disjuncted and solving only one $CI_i$ will solve *C*. To improve the performance, first we calculate branch distance for each $CI_i$ based on a randomly generated test case. Then we select an independent constraint $CI_j$ from *CI,* which has the minimum branch distance. After selecting $CI_j$, we instruct the search algorithm to only solve $CI_j$, thus avoiding the cost of solving and evaluating the remaining independent constraints. In this case, we need to calculate *n* branch distance calculations corresponding to *n* independent constraints in *C,* prior to solving the selected $CI_j$. For example, if we have a constraint A *or* B, then branch distance is calculated as follows:

```
If (d(A) <= d(B)) then Solve for A
else     Solve for B
```

However, it could be possible that the chosen $CI_j$ is not feasible. In such cases, if within a predefined number of fitness evaluations the fitness does not improve, then the search is re-directed toward one of the other clauses.

### 3.3 Improved-implies Branch Distance Calculation

The result of *implies* operation is *true* in the following two cases: (1) *false implies true/false;* (2) *true/false implies true*. To explain branch distance calculation in this case, consider a simple constraint *A implies B*, where *A* and *B* are two operands. By exploiting the definition of *implies*, we calculate branch distance as follows.

```
If (d(not A)<= d(B)) then Solve for not A
 else Solve for B
```

Notice that we need one branch distance calculation to calculate *d(not A)* and one to calculate *d(B)* based on a randomly generated test case. This means that we need two calculations at the beginning to select either *not A* or *B*, depending on which one has the minimum distance. Later on, the selected one is solved by a search algorithm.

### 3.4 Improved-xor Branch Distance Calculation

The result of *xor* operator is *true* if any one of the operands is *true* and the other is *false* at the same time. In this case, by exploiting the definition of *xor*, we calculate branch distance as follows:

```
Considering the example: A xor B
  If (d(A)+d(not B) <= d(not A) +d (B)) then
```

```
    Solve for (A and not B) using improved-and
else
    Solve for (not A and B) using improved-and
```

This means that we will calculate the branch distance for both *A and not B* and *not A and B* based on a randomly generated test case. We then make the search algorithm solve the constraint that has minimum branch distance. In this case, we need four branch distance calculations at the beginning based on a randomly generated test case to select *d(A)+d(Not B)* or *d(notA) +d (B)*.

## 4. EMPIRICAL EVALUATION ON ARTIFICAL PROBLEMS

### 4.1 Experiments Design

To empirically evaluate whether the improved heuristics (*IM*) defined in Section 3 really improve the effectiveness of a search algorithm as compared to existing heuristics, *Not IMproved* (*NIM*), we carefully defined artificial problems to evaluate each improved heuristic. For each of the four *Boolean* operations, we defined seven artificial problems covering various significant OCL operations such as *forAll*, *include*, and *one*. The model we used for the experiment consists of a very simple class diagram with one class *B*. *B* has two attributes *x* and *y* of type *Integer*. Due to space limitations, we do not provide full details of all the artificial problems used in the paper. However, the readers are requested to see the corresponding technical report, where we provide the problems in full [7].

In our experiments, we compared three search algorithms: AVM, GA, (1+1) EA, and used RS as a comparison baseline to assess the difficulty of the addressed problems [8]. AVM was selected as a representative of local search algorithms. GA was selected since it is the most commonly used global search algorithm in search-based software engineering [8]. (1+1) EA is simpler than GAs, but we found that it can be more effective for software testing problems (e.g., see [9, 10]). 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 1/n, where n is the number of variables. Different settings would lead to different performance of a search algorithm, but standard settings usually perform well [11].

In this first set of experiments, we address the following research questions:

**RQ1**: Does *IM* improve the effectiveness of search over *NIM*?

**RQ2**: Among the considered search algorithms (AVM, GA, (1+1) EA), which one fares best in solving OCL constraints using *IM* and how do they compare to RS?

To compare the algorithms for *IM* and *NIM*, we followed the

Table II. Results for artificial problems for each algorithm for *IM* and *NIM**

| | AVM | | | | (1+1) EA | | | | RS | | | | GA | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P# | S(IM) | S(NIM) | O/A | p-value | S(IM) | S(NIM) | O/A | p-value | S(IM) | S(NIM) | O/A | p-value | S(IM) | S(NIM) | O/A | p-value |
| 1 | 0.54 | 0.31 | 2 | 0.001 | 1 | 0.43 | 265 | 1.51E-22 | 0 | 0 | *0.5* | *1* | 0 | 0 | *0.5* | *1* |
| 2 | 0.95 | 1 | *0.23* | *1.07E-07* | 0.81 | 0.19 | 17 | 4.09E-19 | 0 | 0 | *0.5* | *1* | 0 | 0 | *0.5* | *1* |
| 3 | 1 | 1 | *0.63* | *0.006* | 1 | 0.87 | *0.758* | *2.38E-05* | 0 | 0 | *0.5* | *1* | 0 | 0 | *0.5* | *1* |
| 4 | 1 | 1 | *0.59* | *0.049* | 1 | 1 | *0.972* | *5.85E-18* | 0 | 0 | *0.5* | *1* | 0 | 0 | *0.5* | *1* |
| 5 | 1 | 1 | *0.48* | *0.47* | 1 | 1 | *0.961* | *1.02E-16* | 0 | 0 | *0.5* | *1* | 0 | 0 | *0.5* | *1* |
| 6 | 0 | 0.01 | 1 | 0.33 | 0.74 | 0.62 | *0.538* | *0.04* | 0 | 0 | *0.5* | *1* | 0 | 0 | *0.5* | *1* |
| 7 | 0.35 | 0.09 | 5 | 1.27E-05 | 1 | 1 | *0.907* | *5.48E-15* | 0 | 0 | *0.5* | *1* | 0 | 0 | *0.5* | *1* |
| 8 | 0.51 | 0.71 | 0.42 | 0.0051 | 1 | 1 | *0.51* | *4.20E-02* | 0.31 | 0.16 | 2 | 0.01 | 1 | 0.91 | 20 | 0.003 |
| 9 | 0.94 | 1 | 0.07 | 0.02 | 0.93 | 0.12 | 88 | 4.81E-34 | 0.14 | 0.18 | *0.52* | *4.34E-01* | 1 | 0.86 | *0.71* | *9.86E-07* |
| 10 | 1 | 1 | *0.61* | *0.08* | 1 | 0.86 | 33 | 7.49E-05 | 0.17 | 0.16 | *0.49* | *0.8* | 1 | 0.86 | *0.68* | *1.87E-06* |
| 11 | 1 | 1 | *0.8* | *4.15E-12* | 1 | 1 | *0.544* | *3.23E-01* | 1 | 1 | *0.51* | *0.5* | 1 | 1 | *0.5* | *0.86* |
| 12 | 1 | 1 | *0.5* | *1* | 1 | 1 | *0.5* | *1* | 1 | 1 | *0.5* | *1* | 1 | 1 | *0.5* | *1* |
| 13 | 0.02 | 0 | 7.51 | 8.20E-11 | 0.83 | 0.72 | *0.421* | *7.16E-02* | 0.01 | 0.02 | *0.5* | *0.78* | 1 | 0.99 | *0.79* | *2.29E-10* |
| 14 | 0.22 | 0.35 | 12.33 | 0.001 | 1 | 1 | *0.296* | *7.40E-09* | 0 | 0 | *0.5* | *1* | 0.68 | 0.19 | 8 | 2.53E-12 |
| 15 | 1 | 0.8 | 51 | 6.64E-07 | 1 | 1 | *1* | *3.96E-18* | 1 | 0.33 | 405 | 4.05E-28 | 1 | 0.92 | 18 | 0.006 |
| 16 | 1 | 0.99 | *1* | *3.95E-18* | 1 | 0.19 | 840 | 1.08E-37 | 1 | 0.13 | 1303 | 8.48E-43 | 1 | 0.77 | 61 | 5.71E-08 |
| 17 | 1 | 0.99 | *1* | *3.87E-18* | 1 | 0.85 | 36 | 3.46E-05 | 1 | 0.24 | 628 | 5.75E-34 | 1 | 0.82 | 45 | 3.29E-06 |
| 18 | 1 | 1 | *1* | *1.55E-23* | 1 | 1 | *1* | *3.93E-18* | 1 | 1 | 1 | 3.90E-18 | 1 | 1 | 1 | 3.90E-18 |
| 19 | 1 | 1 | *1* | *3.88E-18* | 1 | 1 | *1* | *1.55E-23* | 1 | 1 | 1 | 1.55E-23 | 1 | 1 | 1 | 1.55E-23 |
| 20 | 1 | 0 | 40401 | 2.21E-59 | 1 | 0.73 | 75 | 1.96E-09 | 1 | 0.01 | 13333 | 2.23E-57 | 1 | 0.88 | 28 | 0.0003 |
| 21 | 1 | 0.32 | 424 | 1.01E-28 | 1 | 1 | *1* | *3.95E-18* | 1 | 0 | 40401 | 2.21E-59 | 1 | 0.24 | 628 | 5.75E-34 |
| 22 | 1 | 0.78 | 57 | 1.30E-07 | 1 | 1 | *0.705* | *3.29E-06* | 0 | 0.26 | 0.01 | 4.60E-09 | 0 | 0.86 | 0.0008 | 6.91E-42 |
| 23 | 1 | 1 | *0.27* | *3.43E-05* | 1 | 0.11 | 1564 | 1.05E-44 | 0 | 0.17 | 0.02 | 7.26E-06 | 0 | 0.74 | 0.001 | 1.39E-32 |
| 24 | 1 | 0.97 | *0.34* | *0.002* | 1 | 0.9 | 23 | 0.001 | 0 | 0.18 | 0.02 | 3.29E-06 | 0 | 0.81 | 0.001 | 1.08E-37 |
| 25 | 1 | 1 | *0.97* | *1.68E-14* | 1 | 1 | *0.785* | *1.13E-10* | 0 | 1 | 2.48E-05 | 2.21E-59 | 0 | 1 | 2.48E-05 | 2.21E-59 |
| 26 | 0.23 | 1 | 0.001 | 1.11E-34 | 1 | 1 | *0* | *3.96E-18* | 0 | 1 | 2.48E-05 | 2.21E-59 | 0 | 1 | 2.48E-05 | 2.21E-59 |
| 27 | 1 | 0 | 40401 | 2.21E-59 | 1 | 0.72 | 79 | 8.25E-10 | 0 | 0.05 | 0.08 | 0.059 | 0 | 0.98 | 0.0001 | 1.14E-55 |
| 28 | 1 | 0.45 | 245 | 1.59E-21 | 1 | 1 | *0.5* | *1.68E-02* | 0 | 0 | *0.5* | *1* | 0 | 0.14 | 0.029 | 7.49E-05 |
| M | 0.81 | 0.71 | NA | | 0.98 | 0.80 | NA | | 0.34 | 0.28 | NA | | 0.49 | 0.61 | NA | |

*M: Mean, P: Problem, S(IM): Success rate for IM, S(NIM): Success rate for NIM, O/A: Represents O, i.e., Odds Ratio if results were significant based on success rates (non-italicized) otherwise represents A, i.e., Â12 when the results are not significant based on success rates (Italics), p-value: Italicized for Mann-Whitney U and non-Italicized for Fisher's Exact test, NA: Not Applicable guidelines defined in [12], 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 a solution was found out of the total number of runs (100 in this case). These success rates are 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 Fisher Exact test since for each run of algorithms the result is binary, i.e., either the result is *'found'* or *'not found'* and this is exactly the situation for which the Fisher's exact test is defined. In addition to statistical significance, we used odds ratios as the results of our experiments are dichotomous. A value greater than 1 means that *IM* has more chances of success as compared to *NIM*, whereas a value of 1 means no differences.

When the differences between the success rates of *IM* and *NIM* were not significant for an algorithm, we further compared the number of iterations taken by the algorithm for *IM* and *NIM* to solve the problems. For this purpose, we used Mann-Whitney U-test [13], 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, a value of Â12 less than 0.5 tells that *NIM* is better than *IM* in finding solutions in lesser number of iterations. A value of 0.5 tells that there is no difference between *IM* and *NIM*, whereas a value greater than 0.5 tells that *IM* is better than *NIM* in finding solutions in lesser number of iterations. In Table II, Table III, and Table IV, results for Mann-Whitney U-test and Â12 are italicized and are only shown when the differences are not significant based on success rates.

Table III. Results for artificial problems for *IM* across algorithms*

| | AVM vs (1+1) EA | | AVM vs RS | | AVM vs GA | | (1+1) EA vs RS | | 1+1(EA) vs GA | | RS vs GA | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P# | p-value | O/A | p-value | O/A | p-value | O/A | p-value | O/A | p-value | O/A | p-value | O/A |
| 1 | 3.17E-17 | 0.005 | 5.05E-21 | 236 | 5.05E-21 | 236 | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 1 | 1 |
| 2 | 0.003 | 4 | 2.13E-51 | 3490 | 2.13E-51 | 3490 | 1.08E-37 | 840 | 1.08E-37 | 840 | 1 | 1 |
| 3 | *0.11* | *0.43* | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 1 | 1 |
| 4 | *3.13E-05* | *0.32* | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 1 | 1 |
| 5 | *0.0005* | *0.33* | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 1 | 1 |
| 6 | 1.39E-32 | 0.001 | 1 | 1 | 1 | 1 | 1.39E-32 | 565 | 1.39E-32 | 565 | 1 | 1 |
| 7 | 6.16E-27 | 0.002 | 1.56E-12 | 109 | 1.56E-12 | 109 | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 1 | 1 |
| 8 | 1.32E-18 | 0.005 | 0.006 | 2 | 1.32E-18 | 0 | 2.44E-29 | 443 | 1 | 1 | 2.44E-29 | 0.002 |
| 9 | 0.08 | 0.51 | 2.22E-33 | 87 | 0.02 | 0 | 2.58E-32 | 74 | 0.01 | 0.06 | 6.91E-42 | 0.001 |
| 10 | 0.13 | 0.59 | 2.64E-39 | 959 | 1 | 1 | 2.64E-39 | 959 | 1 | 1 | 2.64E-39 | 0.001 |
| 11 | 0.002 | 0.61 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 12 | 1 | 0.50 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 13 | 6.91E-36 | 0.005 | 1 | 2 | 1.14E-55 | 0.0001 | 1.90E-37 | 317 | 7.26E-06 | 0.02 | 2.23E-57 | 7.50E-05 |
| 14 | 2.08E-35 | 0.001 | 1.30E-07 | 58 | 7.19E-11 | 0.13 | 2.21E-59 | 40401 | 2.41E-11 | 95 | 1.01E-28 | 0.002 |
| 15 | *1* | *0.5* | *1* | *0.5* | *1* | *0.5* | *1* | *0.5* | *1* | *0.5* | 1 | 1 |
| 16 | *1* | *0.5* | *1* | *0.5* | *1* | *0.5* | *1* | *0.5* | *1* | *0.5* | 1 | 1 |
| 17 | *1* | *0.5* | *1* | *0.5* | *1* | *0.5* | *1* | *0.5* | *1* | *0.5* | 1 | 1 |
| 18 | *1* | *0.5* | *1* | *0.5* | *1* | *0.5* | *1* | *0.5* | *1* | *0.5* | 1 | 1 |
| 19 | *1* | *0.5* | *1* | *0.5* | *1* | *0.5* | *1* | *0.5* | *1* | *0.5* | 1 | 1 |
| 20 | *1* | *0.5* | *1* | *0.5* | *1* | *0.5* | *1* | *0.5* | *1* | *0.5* | 1 | 1 |
| 21 | *1* | *0.5* | *1* | *0.5* | *1* | *0.5* | *1* | *0.5* | *1* | *0.5* | 1 | 1 |
| 22 | *0.03* | *0.42* | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 1 | 1 |
| 23 | *0.001* | *0.39* | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 1 | 1 |
| 24 | *3.15E-05* | *0.34* | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 1 | 1 |
| 25 | *0.0004* | *0.32* | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 1 | 1 |
| 26 | 1.11E-34 | 0.002 | 5.71E-08 | 61 | 5.71E-08 | 61 | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 1 | 1 |
| 27 | *8.37E-05* | *0.37* | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 1 | 1 |
| 28 | *0.005* | *0.38* | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 2.21E-59 | 40401 | 1 | 1 |

* O/A: Represents O, i.e., Odds Ratio if results were significant based on success rates (non-italicized) otherwise represents A, i.e., Â12 when the results are not significant based on success rates (Italics), p-value: Italicized for Mann-Whitney U and non-Italicized for Fisher's Exact test

## 4.2 Experiment Execution

We executed each of the four algorithms 100 times with both *IM* and *NIM* for the 28 problems. We let the algorithms run up to 10,000 fitness evaluations on each problem and collected data on whether the algorithms found solutions for *IM* and *NIM*. 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.

Table IV. Results for industrial constraints for each algorithm for *IM* and *NIM**

| | **AVM** | | | | **(1+1) EA** | | | | **RS** | | | | **GA** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **P#** | **S(IM)** | **S(NIM)** | **O/A** | **p-value** | **S(IM)** | **S(NIM)** | **O/A** | **p-value** | **S(IM)** | **S(NIM)** | **O/A** | **p-value** | **S(IM)** | **S(NIM)** | **O/A** | **p-value** |
| 1 | 1 | 1 | 0.68 | *5.73E-07* | 1 | 0.5 | 201 | 4.45E-19 | 1 | 0.1 | 1732 | 1.04E-45 | 1 | 0.3 | 465 | 5.77E-30 |
| 2 | 1 | 1 | *0.74* | *3.05E-05* | 1 | 0.7 | 87 | 1.43E-10 | 1 | 0.1 | 1732 | 1.04E-45 | 1 | 0.2 | 789 | 6.51E-37 |
| 3 | 1 | 1 | *0.83* | *2.20E-13* | 1 | 0.61 | 129 | 3.54E-14 | 1 | 0.03 | 5599 | 3.91E-54 | 1 | 0.25 | 595 | 2.87E-33 |
| 4 | 1 | 1 | *0.85* | *8.05E-12* | 1 | 0.59 | 140 | 5.03E-15 | 1 | 0.01 | 13333 | 2.23E-57 | 1 | 0.19 | 840 | 1.08E-37 |
| 5 | 1 | 1 | *0.75* | *1.21E-07* | 1 | 0.68 | 95 | 2.41E-11 | 1 | 0.02 | 7919 | 1.14E-55 | 1 | 0.17 | 959 | 2.64E-39 |
| 6 | 1 | 1 | *0.80* | *6.77E-10* | 1 | 0.65 | 109 | 1.56E-12 | 1 | 0 | 40401 | 2.21E-59 | 1 | 0.14 | 1199 | 6.91E-42 |
| 7 | 1 | 1 | *0.83* | *1.02E-13* | 1 | 1 | *0.34* | *0.005* | 1 | 0.8 | 51 | 6.64E-07 | 1 | 0.7 | 87 | 1.43E-10 |
| 8 | 1 | 1 | *0.73* | *2.67E-08* | 1 | 0.8 | 51 | 6.64E-07 | 1 | 0.4 | 300 | 3.90E-24 | 1 | 0.9 | 23 | 0.001 |
| 9 | 1 | 1 | *0.72* | *4.49E-06* | 1 | 0.87 | 31 | 0.0001 | 1 | 0.42 | 277 | 4.54E-23 | 1 | 0.93 | 16 | 0.01 |
| 10 | 1 | 1 | *0.71* | *6.65E-06* | 1 | 0.83 | 42 | 7.26E-06 | 1 | 0.46 | 236 | 5.05E-21 | 1 | 0.85 | 36 | 3.46E-05 |
| 11 | 1 | 1 | *0.62* | *0.008* | 1 | 0.89 | 26 | 0.0007 | 1 | 0.26 | 565 | 1.39E-32 | 1 | 0.79 | 54 | 2.95E-07 |
| 12 | 1 | 1 | *0.70* | *2.40E-06* | 1 | 0.85 | 36 | 3.46E-05 | 1 | 0.35 | 371 | 6.16E-27 | 1 | 0.83 | 42 | 7.26E-06 |
| 13 | 1 | 1 | *0.64* | *6.20E-07* | 1 | 0.8 | 51 | 6.64E-07 | 1 | 0.1 | 1732 | 1.04E-45 | 1 | 0.4 | 300 | 3.90E-24 |
| 14 | 1 | 1 | *0.72* | *3.20E-05* | 1 | 0.7 | 87 | 1.43E-10 | 1 | 0.1 | 1732 | 1.04E-45 | 1 | 0.4 | 300 | 3.90E-24 |
| 15 | 1 | 1 | *0.74* | *3.60E-07* | 1 | 0.69 | 91 | 5.90E-11 | 1 | 0.05 | 3490 | 2.13E-51 | 1 | 0.37 | 340 | 8.62E-26 |
| 16 | 1 | 1 | *0.75* | *1.54E-06* | 1 | 0.66 | 104 | 3.91E-12 | 1 | 0.12 | 1423 | 9.76E-44 | 1 | 0.38 | 326 | 3.13E-25 |
| 17 | 1 | 1 | *0.77* | *5.02E-08* | 1 | 0.76 | 64 | 2.48E-08 | 1 | 0.06 | 2922 | 3.77E-50 | 1 | 0.17 | 959 | 2.64E-39 |
| 18 | 1 | 1 | *0.70* | *1.17E-05* | 1 | 0.69 | 91 | 5.90E-11 | 1 | 0.03 | 5599 | 3.91E-54 | 1 | 0.22 | 701 | 2.08E-35 |
| 19 | 1 | 1 | *0.76* | *3.87E-08* | 1 | 0.68 | 95 | 2.41E-11 | 1 | 0.12 | 1423 | 9.76E-44 | 1 | 0.43 | 266 | 1.51E-22 |
| 20 | 1 | 1 | *0.71* | *5.96E-05* | 1 | 0.89 | 26 | 0.0007 | 1 | 0.18 | 896 | 1.73E-38 | 1 | 0.64 | 114 | 6.13E-13 |
| 21 | 1 | 1 | *0.81* | *2.19E-10* | 1 | 0.66 | 104 | 3.91E-12 | 1 | 0.05 | 3490 | 2.13E-51 | 1 | 0.3 | 465 | 5.77E-30 |
| 22 | 1 | 1 | *0.80* | *6.85E-09* | 1 | 0.66 | 104 | 3.91E-12 | 1 | 0.02 | 7919 | 1.14E-55 | 1 | 0.31 | 443 | 2.44E-29 |
| 23 | 1 | 1 | *0.00* | *3.95E-18* | 0.55 | 1 | 0.01 | 8.93E-17 | 0.05 | 1 | 0.0002 | 2.13E-51 | 0.21 | 1 | 0.001 | 3.75E-36 |
| 24 | 1 | 1 | *0.74* | *3.73E-07* | 1 | 0.72 | 79 | 8.25E-10 | 1 | 0 | 40401 | 2.21E-59 | 1 | 0.21 | 743 | 3.75E-36 |
| 25 | 1 | 1 | *0.72* | *3.82E-06* | 1 | 0.85 | 36 | 3.46E-05 | 1 | 0.66 | 104 | 3.91E-12 | 1 | 0.94 | 14 | 0.02 |
| 26 | 1 | 1 | *0.65* | *0.01* | 1 | 0.95 | *0.48* | *0.65* | 1 | 0.92 | *0.43* | *0.11* | 1 | 0.97 | *0.53* | *0.49* |
| 27 | 1 | 1 | *0.72* | *4.49E-06* | 1 | 0.87 | 31 | 0.0001 | 1 | 0.42 | 277 | 4.54E-23 | 1 | 0.93 | 16 | 0.01 |
| 28 | 1 | 1 | *0.71* | *6.65E-06* | 1 | 0.83 | 42 | 7.26E-06 | 1 | 0.46 | 236 | 5.05E-21 | 1 | 0.85 | 36 | 3.46E-05 |
| 29 | 1 | 1 | *0.62* | *0.008* | 1 | 0.89 | 26 | 0.0007 | 1 | 0.26 | 565 | 1.39E-32 | 1 | 0.79 | 54 | 2.95E-07 |
| 30 | 1 | 1 | *0.74* | *3.73E-07* | 1 | 0.72 | 79 | 8.25E-10 | 1 | 0 | 40401 | 2.21E-59 | 1 | 0.21 | 743 | 3.75E-36 |
| 31 | 1 | 1 | *0.76* | *3.87E-08* | 1 | 0.68 | 95 | 2.41E-11 | 1 | 0.12 | 1423 | 9.76E-44 | 1 | 0.43 | 266 | 1.51E-22 |
| 32 | 1 | 1 | *0.71* | *5.96E-05* | 1 | 0.89 | 26 | 0.0007 | 1 | 0.18 | 896 | 1.73E-38 | 1 | 0.64 | 114 | 6.13E-13 |
| 33 | 1 | 1 | *0.81* | *2.19E-10* | 1 | 0.66 | 104 | 3.91E-12 | 1 | 0.05 | 3490 | 2.13E-51 | 1 | 0.3 | 465 | 5.77E-30 |
| 34 | 1 | 1 | *0.80* | *6.85E-09* | 1 | 0.66 | 104 | 3.91E-12 | 1 | 0.02 | 7919 | 1.14E-55 | 1 | 0.31 | 443 | 2.44E-29 |
| 35 | 1 | 1 | *0.73* | *2.52E-07* | 1 | 0.69 | 91 | 5.90E-11 | 1 | 0.03 | 5599 | 3.91E-54 | 1 | 0.18 | 896 | 1.73E-38 |
| 36 | 1 | 1 | *0.74* | *3.73E-07* | 1 | 0.72 | 79 | 8.25E-10 | 1 | 0 | 40401 | 2.21E-59 | 1 | 0.21 | 743 | 3.75E-36 |
| 37 | 1 | 1 | *0.77* | *9.70E-09* | 1 | 0.73 | 75 | 1.96E-09 | 1 | 0.12 | 1423 | 9.76E-44 | 1 | 0.52 | 186 | 3.85E-18 |
| 38 | 1 | 1 | *0.76* | *2.48E-09* | 1 | 0.82 | 45 | 3.29E-06 | 1 | 0.22 | 701 | 2.08E-35 | 1 | 0.66 | 104 | 3.91E-12 |
| 39 | 1 | 1 | *0.74* | *3.60E-07* | 1 | 0.69 | 91 | 5.90E-11 | 1 | 0.05 | 3490 | 2.13E-51 | 1 | 0.37 | 340 | 8.62E-26 |
| 40 | 1 | 1 | *0.75* | *1.54E-06* | 1 | 0.66 | 104 | 3.91E-12 | 1 | 0.12 | 1423 | 9.76E-44 | 1 | 0.38 | 326 | 3.13E-25 |
| 41 | 1 | 1 | *0.77* | *5.02E-08* | 1 | 0.76 | 64 | 2.48E-08 | 1 | 0.06 | 2922 | 3.77E-50 | 1 | 0.17 | 959 | 2.64E-39 |
| 42 | 1 | 1 | *0.70* | *1.17E-05* | 1 | 0.69 | 91 | 5.90E-11 | 1 | 0.03 | 5599 | 3.91E-54 | 1 | 0.22 | 701 | 2.08E-35 |
| M | 1 | 1 | NA | | 0.99 | 0.75 | NA | | 0.98 | 0.20 | NA | | 0.98 | 0.48 | NA | |

*M: Mean, P: Problem, S(IM): Success rate for IM, S(NIM): Success rate for NIM, O/A: Represents O, i.e., Odds Ratio if results were significant based on success rates (non-italicized) otherwise represents A, i.e., Â12 when the results are not significant based on success rates (Italics), p-value: Italicized for Mann-Whitney U and non-Italicized for Fisher's Exact test, NA: Not Applicable

## 4.3 Experiment Results and Analysis

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

### 4.3.1 Comparision of *IM* and *NIM* for each Algorithm (RQ1)

Table II shows the results for all the algorithms for *IM* and *NIM* for 28 (P1-P28) artificial problems. P1-P7 are designed for evaluating the *improved-and* heuristic, P8-P14 for the *improved-or* heuristic, P15-P21 for the *improved-implies* heuristic, and P22-P28 for the *improved-xor* heuristic.

First we discuss the performance of AVM for *IM* and *NIM.* As we observe in Table II that for the *Boolean* operation *and* (P1-P7), *IM* is significantly better than *NIM* for four problems (P1, P3, P4, and P7) and there are no significant differences for two problems (P5-P6) between *IM* and *NIM.* In case of P2, there is no significant difference between *IM* and *NIM* in terms of success rates, but *NIM* finds the solution faster than *IM*. For the operator *or*, we observed that for three problems (P10, P11, and P13) *IM* is significantly better than *NIM*, whereas for P8, P10, and P12, there were no significant differences. However, for P9 *NIM* is significantly better than *IM*. In case of *implies* operator, *IM* was significantly better than *NIM* for all problems, either based on success rates or number of iterations. For operator *xor*, *IM* was better than *NIM* for four problems and was worse for three problems. Out of two of these three problems, there were no significant differences between *IM* and *NIM*, though *NIM* found solutions quickly. Overall, we can conclude that *IM* improved the performance of search in 18 problems. For six of the problems, there were no significant differences between *IM* and *NIM*. For four of the remaining, there were no significant differences between *NIM* and *IM* in terms of success rates, though *NIM* found solutions in lesser number of iterations.

For (1+1) EA, in the case of *and*, *IM* is significantly better than *NIM* for all the problems. For *or*, *IM* is significantly better than *NIM* for all the problems except two. For P12, there were no significant differences. For P13, *IM* has higher success rate (0.83) than *NIM* (0.72), but not significant (*p-value* = 0.08). However, in terms of number of iterations for P13, *NIM* took significantly less number of iterations. For *implies*, *IM* is significantly better than *NIM* for all problems. For *xor*, *IM* is significantly better than *NIM* for all problems, except 22, and 26 in which cases *NIM* is better. But in all these three cases P19, P22, and P26, the success rates for both *IM* and *NIM* were equal to 100%. Therefore, we can conclude that for all these operators, (1+1) EA with *IM* shows significantly better results than *NIM*, except for the problems that were either simple for the algorithm (i.e., the success rate was 100%) or for P13, where the success rate of *IM* is better, but *NIM* finds the solution faster.

For RS, in the case of *and,* success rates for both *IM* and *NIM* are 0%. For *or*, we observed that for most of the problems (P10-P14), there are no significant differences between *IM* and *NIM*. For *implies*, *IM* is significantly better than *NIM* for all problems either based on success rates or number of iterations. Similar results are observed for *xor*, where *NIM* is significantly better than *IM* (P22-P26) or there are no significant differences (P27-P28). Although, success rates are again very low for RS, i.e., less than equal to 34% as it can be seen in Table II. Therefore, we can conclude that there is no significant benefit of using *IM* rather than *NIM* for RS as for both cases, the overall success rates are very low.

For GA, in case of *and*, both *NIM* and *IM* achieved 0% success rates. In case of *or*, *IM* performed significantly better than *NIM* either in terms of success rates or number of iterations except for P11 and P12. For *implies* and *xor*, success rates for *IM* are 0% for all the problems and thus *NIM* seems to perform significantly better than *IM* for these cases.

To summarize, we can answer our RQ1 as follows: (1+1) EA with *IM* significantly improve effectiveness of solving OCL constraints for generating test data. (1+1) EA has higher average success rates with *IM* than *NIM*, i.e., 0.98 (Table II). In addition, when there were no significant differences between *IM* and *NIM*, most of the time the algorithm managed to solve constraints in significantly lesser number of iterations than *NIM* for successful runs as it can be seen in italicized Â12 values are greater than 0.5 (eight column in Table II) with p-values less than 0.05. For AVM, for more than half of the problems (18 out of 28), *IM* significantly improved the effectiveness of search and for six problems no significant differences were observed in terms of both success rates and number of iterations. For most of the remaining problems, *IM* and *NIM* have no significant differences in success rates. For RS and GA, overall we obtained very low success rates for both *IM* and *NIM,* for instance, 39% and 49% for RS and GA respectively for *NIM*.

#### *4.3.2 Comparision of Algorithms for IM (RQ2)*

Table III presents results for *IM* across each pair of algorithms for the four algorithms. For each pair, whenever Odds Ratio (O) is greater than 1, this means that the first algorithm has more chances of success than the second one. For instance in comparison of (AVM vs RS), when O>1, this means that AVM has more chances of success than RS. In Table III, a value of Â12 (*A*) less than 0.5 means that the second algorithm is better than the first one in finding solutions in lesser number of iterations. A value of Â12=0.5 means no differences and a value greater than 0.5 means that first algorithm is better than the second one.

First we discuss the performance of AVM in comparison to other algorithms. In case of AVM vs (1+1) EA, for 18 out of 28 problems (1+1) EA showed significantly better results than AVM. For the ten remaining problems, there were no significant differences between the two algorithms. For AVM vs RS, in 17 problems AVM shows significantly better results than RS (indicated in Table III by *O* values greater than 1 and p-values less than 0.05). For the remaining 11 problems, there is no difference between the two algorithms. For AVM vs GA, in the case of 16 problems, AVM showed significantly better results than GA, for the other 11 there were no significant differences, whereas for P9, RS was significantly better than GA.

Next, we discuss the comparison of (1+1) EA with the remaining algorithms (GA and RS). For (1+1) EA vs RS, in case of 19 problems, (1+1) EA shows significantly better performance than RS. For the remaining nine problems, there were no significant differences. In case of (1+1) EA vs GA, for 16 problems (1+1) EA showed significantly better performance. For the 11 problems, there were no statistically significant differences, whereas for P9, GA was significantly better than (1+1) EA.

Finally, in case of RS vs GA, for 23 problems, there were no significant differences and for the remaining five problems (P8-P10, P13, P14), GA showed significantly better performance.
Based on the above results, we can conclude that (1+1) EA with *IM* is significantly better than the other algorithms and is best suited for generating test data for OCL constraints, followed by AVM.

## 5. INDUSTRIAL CASE STUDY

### 5.1 Introduction of the Case Study

Our industrial case study is part of a project aiming at supporting automated, model-based robustness testing of a core subsystem of a video conference system (VCS) called Saturn developed by Cisco Systems, Inc, Norway. The standard behavior of the system is modeled as a UML 2.0 state machine. In addition, we modeled robustness behavior separately as aspect state machines. The robustness behavior is modeled based on different functional and non-functional properties, whose violations lead to erroneous states. Such properties can be related to the VCS or its environment such as the network and other systems interacting with the SUT. In our case study, in total we have 57 constraints and 42 out of these 57 constraints have more than one clause and thus we included them in this experiment. These constraints contain between 6 and 8 clauses and all of these clauses were conjuncted, therefore we could only evaluate the *improved-and* heuristic. We followed exactly the same design and analysis methods as for artificial problems except that we ran each algorithm for 2,000 iterations due to time limitation to execute the experiment.

### 5.2 Results and Analysis

#### *5.2.1 Comparision of IM and NIM for each Algorithm (RQ1)*

Table IV presents results for *IM* and *NIM* for each of the 42 constraints for all the four algorithms. We observed that mean success rates using *IM* for all the algorithms are greater than equal to 98% ; however for *NIM* mean success rates are 100%, 80%, 20%, and 50% for AVM, (1+1) EA, RS, and GA, respectively.

For AVM, for both *IM* and *NIM*, success rates are 100%, therefore, we compared *IM* and *NIM* based on number iterations to solve problems. All Â12 values are greater than 0.5 and p-values less than 0.05 suggesting that *IM* took significantly less number of iterations to solve the problems. For the rest of the three algorithms, *IM* has either significantly higher success rates than *NIM* or took significantly lower iterations to solve the problems as it can be seen in Table IV except for Problem 23.

Based on the above results we can conclude that *IM* brought significant improvement to all the algorithms.

### *5.2.2 Comparision of Algorithms for IM (RQ2)*

The results of this research question are provided in [7] due to space limitations. In terms of *IM* across algorithms as we can observe in Table IV that success rates for all four algorithms are greater than or equal to 98%; and thus the differences in terms of success rates are not significant. Therefore, we performed comparisons based on number of iterations for each pair of algorithms as shown in Table V of supplementary file. In Table V of supplementary file, a value of Â12 less than 0.5 means that the second algorithm has higher chances of finding a solution in lesser number of iterations than the first one. A value greater than 0.5 means first algorithm is better than the second one.

For AVM vs (1+1) EA, most of Â12 values are greater than 0.5 with p-values less than 0.05 (32 out of 42) suggesting that AVM has significantly higher chances of solving constraints in lesser number of iterations than (1+1) EA. Similar results can be seen for AVM vs RS (28 out of 42) and AVM vs GA (26 out of 42) in Table V of supplementary file. For (1+1) EA vs RS, for 23 out of 42 problems, (1+1) EA took significantly less iterations than RS, whereas in nine it took significantly more iterations than RS, and no differences were observed in 10 problems. For (1+1) EA vs GA, (1+1) EA took significantly less iterations than GA for 16 problems, where (1+1) EA took significantly more iterations than GA in 13 problems, and no differences were observed for the rest of the problems. For RS vs GA, in case of 16 problems Â12 values are below 0.5 with p-values less than 0.05 suggesting RS has significantly higher chances of solving constraints with more iterations than GA. For 12 problems, RS took significantly less iterations than GA, whereas for the remaining 14 problems no significant differences were observed.

Based on the above results, we can conclude that for the industrial case study, AVM has higher chances of solving constraints with lesser number of iterations with *IM* than other algorithms.

## 6. OVERALL DISCUSSION

Based on the results of our empirical analyses, we recommend using (1+1) EA with *IM* for solving constraints since the effectiveness of the algorithm was better than all the other algorithms we compared, as we discussed in Section 4 based on artificial problems on which all four heuristics were applied. However, we found that AVM was the best algorithm for the industrial case study as we discussed in Section 5. Notice that, in the industrial case study, we could only evaluate the heuristic for *and* operator and, even in this case, there were no significant differences across success rates of AVM and (1+1) EA, which are 100% and 99% respectively. Based on these observations, we suggest using AVM when the constraints are conjuncted and there is limited time as it may find solutions quicker than (1+1) EA. If we are flexible with time budget (e.g., the constraints need to be solved only once, and the cost of doing that is negligible compared to other costs in the testing phase), we rather recommend running (1+1) EA for as many iterations as possible as we notice that the success rate for (1+1) EA was 99% on average for the industrial case study, whereas for the artificial problems, it has mean success rate of 98% with *IM* and AVM has 81% of average success rate (Table II).

The difference in performance between AVM and (1+1) EA has a clear explanation. AVM works like a sort of greedy local search. If the fitness function provides a clear gradient towards the global optima, then AVM will quickly converge to one of them. On the other hand, (1+1) EA puts more focus on the exploration of the search landscape. When there is a clear gradient toward global optima, (1+1) EA is still able to reach those optima in reasonable time, but it will spend some time in exploring other areas of the search space. This latter property becomes essential in difficult landscapes where there are many local optima. In these cases, AVM gets stuck and has to re-start from other points in the search landscape. On the other hand, (1+1) EA, thanks to its mutation operator, has always a non-zero probability of escaping from local optima.

## 7. THREATS TO VALIDITY

To reduce construct validity threats, we used search success rate measure, which is comparable across the three search algorithms (AVM, (1+1) EA, GA) and RS. Furthermore, we used the same stopping criterion for all algorithms, i.e., number of iterations, which is comparable measure 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 the results were obtained by chance. Furthermore, we performed Fisher exact tests to compare proportions and determine the statistical significance of the results. To determine the practical significance of the results obtained, we measured the effect size using the odds ratio of success rates across search techniques, we used Mann-Whitney U-test [13] to determine significance of results and Vargha and Delaney's Â12 statistics for report effect sizes, using the guidelines for reporting results of randomized algorithms presented in [12].

A possible threat to internal validity is that we have experimented with only one configuration setting for the GA parameters. However, these settings are in accordance with the common guidelines in the literature and our previous experience on testing problems. Parameter tuning can improve the performance of GAs, although default parameters often provide reasonable results [11].

We ran the experiments on an industrial case study to generate test data for 42 different OCL constraints, ranging from constraints having just six to eight clauses. Although the empirical analysis is based on a real industrial system, our results might not generalize to other case studies. However, such threats to external validity are common to all empirical studies. In addition to the industrial case study, we also conducted an empirical evaluation of each proposed branch distance calculation using small yet complex artificial problems to demonstrate that the effectiveness of our heuristics holds even for more complex problems.

## 8. RELATED WORK

A number of approaches use constraint solvers for analyzing OCL constraints for various purposes. These approaches usually translate constraints and models into a formalism (e.g., Alloy [14], temporal logic BOTL [15], FOL [16], graph constraints [17]), which can then be analyzed by a constraint analyzer (e.g., Alloy constraint analyzer [18], model checker [15], Satisfiability Modulo Theories (SMT) Solver [16], theorem prover [16]). Satisfiability Problem (SAT) solvers have also been used for evaluating OCL specifications ,e.g., for OCL operation contracts (e.g., [19]).

Some approaches are reported in the literature to solve OCL constraints and generate data that evaluate the constraints to true. The data generated can then be used as test data. Most of these approaches do not handle important features of OCL (e.g., Collections) and UML, and are based on formal constraint solving techniques, such as SAT solving (e.g., [20]), constraint satisfaction problem (CSP) (e.g., [21]), higher order logic (HOL) [22], and partition analysis (e.g., [23]). The work presented in [21] is one of the most sophisticated approaches reported so far. However, its focus is on the verification of correctness properties, though it generates an instantiation of the model as part of its process. The major limitation of the approach is that the search space is bounded and, as the bounds are raised, CSP faces a quickly increasing combinatorial explosion (as discussed in [21]). The task of determining the optimal bounds for verification is left to the user and considering that models of industrial applications can have a large number of attributes, manually finding bounds for individual attributes is often impractical.

Existing approaches for OCL constraint solving do not fully fit the needs we identified with our industrial partners. Most of the approaches are only limited to simple numerical expressions and do not handle collections, which are used widely to specify expressions that navigate over associations. These limitations are due to the high expressiveness of OCL that makes the definitions of constraints easier, but their analysis more difficult. The conversion of OCL to a SAT formula or a CSP instance can easily result in a combinatorial explosion as the complexity of the model and constraints increases (as discussed in [21]). Most of the discussed approaches listed in this section either do not support the OCL constructs present in the constraints that we have in our industrial case study or are not efficient to solve them. Hence, existing techniques based on conversion to lower-level languages seem impractical in the context of large scale, real-world systems.

## 9. CONCLUSION

This paper presented four new heuristics for solving OCL constraints, which are based on the process of dividing a given constraint into its independent sub constraints and solving them separately, by first focusing on the ones that are easier to solve. The new heuristics correspond to the four primary operations for OCL constraints (*and, or, implies,* and *xor*). We conducted two empirical studies, one on 28 artificial constraints and the other on 42 constraints from an industrial case study, to evaluate the performance of improved heuristics and comparing them with previously defined heuristics. For this purpose, we used three search algorithms: (1+1) Evolutionary Algorithm (EA), Genetic Algorithms (GA), and Alternating Variable Method (AVM). The results of the empirical studies show that the novel heuristics lead to significantly better performance for each algorithm, on both the artificial problems and the industrial case study. Overall, (1+1) EA with the improved heuristics showed best performance among all the algorithms and must be used in practice.

## 10. REFERENCES

[1] OCL. (2011). *Object Constraint Language Specification, Version 2.2*. Available: http://www.omg.org/spec/OCL/2.2/

[2] R. V. Binder, Testing object-oriented systems: models, patterns, and tools: Addison-Wesley Longman Publishing Co., Inc., 1999.

[3] S. Ali, M. Z. Iqbal, A. Arcuri, and L. Briand, "A Search-based OCL Constraint Solver for Model-based Test Data Generation," in Proceedings of the 11th International Conference On Quality Software (QSIC 2011), 2011, pp. 41-50.

[4] S. Ali, M. Z. Iqbal, A. Arcuri, and L. Briand, "Generating Test Data from OCL Constraints with Search Techniques," IEEE Trans. Softw. Eng., vol. 39, pp. 1376-1402, 2013.

[5] P. McMinn, "Search-based software test data generation: a survey: Research Articles," Softw. Test. Verif. Reliab., vol. 14, pp. 105-156, 2004.

[6] S. Ali, M. Z. Iqbal, A. Arcuri, and L. Briand, "Generating Test Data from OCL Constraints with Search Techniques," Simula Reserach Laboratory, Technical Report (2010-16)2012.

[7] S. Ali, M. Z. Iqbal, and A. Arcuri, "Empirically Evaluating Improved Heuristics for Test Data Generation from OCL Constraints using Search Algorithms," Simula Research Laboratory2012.

[8] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation," IEEE Transactions on Software Engineering, vol. 99, 2009.

[9] A. Arcuri, "It really does matter how you normalize the branch distance in search-based software testing," Software Testing, Verification and Reliability, 2011.

[10] M. Z. Iqbal, A. Arcuri, and L. Briand, "Empirical Investigation of Search Algorithms for Environment Model-Based Testing of Real-Time Embedded Software " in International Symposium on Software Testing and Analysis (ISSTA), 2012.

[11] A. Arcuri and G. Fraser, "On Parameter Tuning in Search Based Software Engineering," presented at the International Symposium on Search Based Software Engineering (SSBSE), 2011.

[12] A. Arcuri and L. Briand., "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering," presented at the International Conference on Software Engineering (ICSE), 2011.

[13] D. J. Sheskin, Handbook of Parametric and Nonparametric Statistical Procedures: Chapman and Hall/CRC, 2007.

[14] B. Bordbar and K. Anastasakis, "UML2Alloy: A tool for lightweight modelling of Discrete Event Systems," presented at the IADIS International Conference in Applied Computing, 2005.

[15] D. Distefano, J.-P. Katoen, and A. Rensink, "Towards model checking OCL," presented at the ECOOP-Workshop on Defining Precise Semantics for UML, 2000.

[16] M. Clavel and M. A. G. d. Dios, "Checking unsatisfiability for OCL constraints," presented at the In the proceedings of the 9th OCL 2009 Workshop at the UML/MoDELS Conferences, 2009.

[17] J. Winkelmann, G. Taentzer, K. Ehrig, and J. M. ster, "Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars," Electron. Notes Theor. Comput. Sci., vol. 211, pp. 159-170, 2008.

[18] D. Jackson, I. Schechter, and H. Shlyahter, "Alcoa: the alloy constraint analyzer," presented at the Proceedings of the 22nd international conference on Software engineering, Limerick, Ireland, 2000.

[19] M. Krieger and A. Knapp, "Executing Underspecified OCL Operation Contracts with a SAT Solver," presented at the 8th International Workshop on OCL Concepts and Tools., 2008.

[20] L. v. Aertryck and T. Jensen, "UML-Casting: Test synthesis from UML models using constraint resolution," presented at the Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2003), 2003.

[21] J. Cabot, R. Claris, and D. Riera, "Verification of UML/OCL Class Diagrams using Constraint Programming," presented at the Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop, 2008.

[22] A. D. Brucker, M. P. Krieger, D. Longuet, and B. Wolff, "A Specification-Based Test Case Generation Method for UML/OCL," presented at the Worksshop on OCL and Textual Modelling, MoDELS, 2010.

[23] L. Bao-Lin, L. Zhi-shu, L. Qing, and C. Y. Hong, "Test case automate generation from uml sequence diagram and ocl expression," presented at the International Conference on cimputational Intelligence and Security, 2007.