

# Solving OCL Constraints for Test Data Generation in Industrial Systems with Search Techniques

*Shaukat Ali, Muhammad Zohaib Iqbal, Andrea Arcuri, Lionel Briand*

**Abstract**— Metaheuristic Model-based testing (MBT) aims at automated, scalable, and systematic testing solutions for complex industrial software systems. To increase chances of adoption in industrial contexts, software systems should be modeled using well-established standards such as the Unified Modeling Language (UML) and Object Constraint Language (OCL). Given that test data generation is one of the major challenges to automate MBT, this is the topic of this paper with a specific focus on solving OCL constraints, which is a necessary step to generate appropriate test data. Though search-based software testing has been applied to test data generation for white-box testing (e.g., branch coverage), its application to the MBT of industrial software systems has been limited. In this paper, we propose a set of search heuristics targeted to OCL constraints to guide test data generation and automate MBT in industrial applications. These heuristics are used to develop an efficient OCL solver exclusively based on search. In this paper, we evaluate these heuristics for search algorithms such as Genetic Algorithms, (1+1) Evolutionary Algorithm and Alternating Variable Method. We empirically evaluate our heuristics using complex artificial problems followed by empirical analyses to evaluate the feasibility of our approach on one industrial system. Though the focus is on OCL constraints, many of the principles introduced here could be adapted to other high level constraint languages based on first-order logic and set theory.

## 1. Introduction

Model-based testing (MBT) has recently received increasing attention in both industry and academia [1]. MBT leads to systematic, automated, and thorough system testing, which would often not be possible without models. However, the full automation of MBT, which

is a requirement for scaling up to real-world systems, requires supporting many tasks, including preparing models for testing (e.g., flattening state machines), defining appropriate test strategies and coverage criteria, and generating test data to execute test cases. Furthermore, in order to increase chances of adoption, using MBT for industrial applications requires using well-established standards, such as the Unified Modeling Language (UML) and its associated language to write constraints: the Object Constraint Language (OCL) [2].

OCL is a standard language that is widely accepted for writing constraints on UML models. OCL is based on first order logic and set theory. It is at a higher expressive level than Boolean predicates written in programming languages such as C and Java. The language allows modelers to write constraints at various levels of abstraction and for various types of models. For example, it can be used to write class and state invariants, guards in state machines, constraints in sequence diagrams, and pre and post conditions of operations. A basic subset of the language has been defined that can be used with meta-models defined in Meta Object Facility (MOF) [3] (which is a standard defined by Object Management Group (OMG) for defining meta-models). This subset of OCL has been largely used in the definition of UML for constraining various elements of the language. Moreover, the language is also used in writing constraints while defining UML profiles, which is a standard way of extending UML for various domains using pre-defined extension mechanisms.

Due to the ability of OCL to specify constraints for various purposes during modeling, for example when defining guard conditions or state invariants in state machines, such constraints play a significant role when testing is driven by models. For example, in state-based testing, if the aim of a test case is to execute a guarded transition (where the guard is written in OCL based on input values of the trigger and/or state variables) to achieve full transition coverage, then it is essential to provide input values to the event that triggers the transition such that the values satisfy the guard. Another example can be to generate valid parameter values based on the pre-condition of an operation.

Test data generation is an important component of MBT automation. For UML models, with constraints in OCL, test data generation is a non-trivial problem. A few approaches in the literature exist that address this issue. But most of them, as we will explain in more details later in the paper, either target only a small subset of OCL [4, 5], are not scalable, or

lack proper tool support [6]. This is a major limitation when it comes to the industrial application of MBT approaches that use OCL to specify constraints on models.

This paper provides and assesses 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 the solving of OCL constraints (covering the entire OCL 2.2 semantics [2]) in order to generate test data. A search-based OCL constraint solver is implemented and evaluated on the first reported, industrial case study on this topic. Note that many of the principles introduced here could be easily adapted to other high-level constraint languages based on first-order logic and set theory, in other modeling languages, and this makes our contribution of general value for test data generation based on models with constraints.

The rest of the paper is organized as follows: Section 2 discusses the background and Section 3 discusses related work. In Section 4, we present the definition of distance function for various OCL constructs. Section 5 discusses the case study and analysis of results of the application of the approach on an industrial case study, whereas Section 6 provides an empirical evaluation of heuristics on a set of artificial problems, and Section 7 provides an overall discussion of the both empirical evaluations. Section 8 discusses the tool support, Section 9 addresses the threats to validity of our empirical study, and finally Section 10 concludes the paper.

## 2. Background

Several software engineering problems can be reformulated as a search problem, such as test data generation [7]. An exhaustive evaluation of the entire search space (i.e., the domain of all possible combinations of problem variables) is usually not feasible. There is a need for techniques that are able to produce “good” solutions in reasonable time by evaluating only a tiny fraction of the search space. Search algorithms can be used to address this type of problem. Several successful results by using search algorithms are reported in the literature for many types of software engineering problems [8-10].

To use a search algorithm, a fitness function needs to be defined. The fitness function should be able to evaluate the quality of a candidate solution (i.e., an element in the search space). The fitness function is problem dependent, and proper care needs to be taken for developing adequate fitness functions. The fitness function will be used to guide the search

algorithms toward fitter solutions. Eventually, given enough time, a search algorithm will find a satisfactory solution.

There are several types of search algorithms. Genetic Algorithms (GAs) are the most well-known [8], and they are inspired by the Darwinian evolution theory. A population of individuals (i.e., candidate solutions) is evolved through a series of generations, where reproducing individuals evolve through crossover and mutation operators. (1+1) Evolutionary Algorithm (EA) is simpler than GAs, in which only a single individual is evolved with mutation. Alternating Variable Method (AVM) is a local search algorithm, which is similar to the Hill Climbing algorithm, with the main difference that it can have larger modifications. To verify that search algorithms are actually necessary because they address a difficult problem, it is a common practice to use Random Search (RS) as comparison baseline [8].

### 3. Related Work

There are a number of approaches that deal with the evaluation of OCL constraints. The basic aim of most of these approaches is to verify whether the constraints can be satisfied. Though most of the approaches do not generate test data, they are still related to our work since they require the generation of values for validating the constraints. These approaches can therefore be adapted for generating test data. In Section 3.1, we discuss the relationship between OCL solvers and OCL evaluators. In Section 3.2, we discuss the OCL-based constraint solving approaches in the literature, whereas Section 3.3 discusses the approaches that use search-based heuristics for testing.

#### 3.1 Comparison with OCL Constraints Evaluation

An OCL evaluator tells whether a constraint on a class diagram satisfies an instantiation of the class diagram provided to it. Several OCL evaluators are currently available that can be used to evaluate OCL constraints such as the IBM OCL evaluator [22], OCLE 2.0 [23], EyeOCL [24], and the OCL evaluation in CertifyIt by Smartesting [25]. Our work is different from these works since we automatically generate instances of a class diagram with the aim of finding a particular instantiation that solves a provided constraint. Note that for our purpose, i.e., solving OCL constraints to generate test data, an OCL evaluator is a necessary component because of two reasons: 1) an evaluator tells if a constraint is solved,

2) an evaluator helps in calculating the fitness (e.g., using a branch distance [26]) of an OCL expression that guides a search algorithm to find a solution. Note that any OCL evaluator can be integrated with our tool.

**Table 1: Summary of OCL Constraint Solving Approaches**

Technique	Translation to Formalism	Intermediate Representation	Complete OCL	OCL Parts Missing or Additional Requirements
Alloy Analyzer [11]	Yes	Alloy	No	Real, String, Enumerations, Limited operations on collections, attributes
Aertryck & Jensen [4]	Yes	FSA	No	Collections, Real, String, Enumerations
Diestefano et al. [12]	Yes	BOTL	No	String, real, enumerations
Clavel et al. [13]	Yes	FOL	No	String, Real, collections other than Set, Enumeration
Bao-Lin et al. [6]	No	DNF	No	Not discussed in the paper
Benattou et al. [5]	No	DNF	No	Class Inheritance, Generalization, Association
Aichernig [14]	Yes	CSP	No	Handles a small subset, collections iterators, Bag, Sequence,
UMLtoCSP [15]	Yes	CSP	No	Enumerations
Queralt et al [16]	Yes	FOL	No	Operations that cannot be converted to select() or size() operations, e.g., collect.
Winkelman [17]	Yes	Graph constraints	No	Collection operations except size(), isEmpty(). Enumerations
Kyas et al [18]	Yes	PVS	No	Not discussed in the paper
Kreiger [19]	Yes	SAT in CNF	No	Adds a non standard extension, String, Real, Enumerations
Weißler [20]	No	Test Tree	No	Collections, Enumerations
Gogolla [21]	Yes	Formal Logic	No	Desired properties of snapshot to be specified in a language ASSL

### 3.2 OCL-based Constraint Solvers

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 [11], temporal logic BOTL [12], FOL [13], Prototype Verification System (PVS) [18], graph constraints [17]), which can then be analyzed by a constraint analyzer (e.g., Alloy constraint analyzer [27], model checker [12], Satisfiability Modulo Theories (SMT) Solver [13], theorem prover [13], [18]). Satisfiability Problem (SAT) solvers have also been used for evaluating OCL specifications ,e.g., for OCL operation contracts (e.g., [28], [19]).

**Table 2: Summary of OCL Constraint Solving Approaches**

<b>Technique</b>	<b>Tool Support</b>	<b>Application on Case Study</b>	<b>Approach Type</b>	<b>Test Data Generation</b>
Alloy Analyzer [11]	Yes	Simple example	SAT Solver	No
Aertryck & Jensen [4]	Yes	Simple example	SAT Solver	Yes
Diestefano et al. [12]	Yes	Simple example	Model Checking	No
Clavel et al. [13]	Yes	Simple example	SMT Solver	No
Bao-Lin et al. [6]	No	Simple example	Partition Analysis	Yes
Benattou et al. [5]	No	Simple example	Partition Analysis	Yes
Aichernig [14]	Yes	Simple example	CSP Solving	No
UMLtoCSP [15]	Yes	Simple example	CSP Solving, Instance Generation	No
Queralt et al [16]	No	No	Reasoning	No
Winkelman [17]	No	No	Instance Generation	No
Kyas et al [18]	Yes	Simple example	Theorem Proving, Interactive	No
Kreiger [19]	Yes	Simple example	SAT Solver	No
Weißler [20]	Yes	Simple example	Partition Testing	Yes
Gogolla [21]	Yes	Simple example	Interactive	No

Some approaches are reported in the literature to solve OCL constraints and generate data that evaluates the constraints to true. The data generated can then be used as test data. Most of these approaches only handle a small subset of OCL and UML, and are based on formal constraint solving techniques, such as SAT solving (e.g., [4]), constraint satisfaction problem (CSP) (e.g., [14], [15]), higher order logic (HOL) [29], and partition analysis (e.g., [6], [5]). The work presented in [15] 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 [15]). The task of determining the optimal bounds for verification is left to the user, which is not simple and requires repeated interactions with the user. Models of industrial applications can have hundreds of attributes and manually finding bounds for individual attributes is often impractical. We present the results of an experiment that we conducted to compare our novel approach with this approach in Section 5.2. Existing approaches for OCL constraint solving do not fully fit the needs we identified with our industrial partners. Almost all of the existing works only support a small, insufficient subset of OCL (Table 1 and Table 2). Most of the

approaches, as shown in Table 1, 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 [15]). For instance, one factor that could easily lead to a combinatorial explosion, when converting an OCL constraint into an instance of SAT formula, is when the number of variables and their ranges increase in a constraint. Conversion to a SAT formula requires that a constraint must be encoded into Boolean formulae at the bit-level and as the number of variables increases in the constraint, chances of a combinatorial explosion increase. For industrial scale systems, as in our case, this is a major limitation, since the models and constraints are generally quite complex. Most of the discussed approaches 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 (see Section 5.2). Hence, existing techniques based on conversion to lower-level languages seem impractical in the context of large scale, real-world systems.

Earlier we discussed approaches that convert OCL expressions to other constraint languages. There are a number of other constraint solvers (such as in [30] [31]) that have their own constraint solving languages. To the best of our knowledge, mappings from OCL constraints to these constraint languages have not been reported. If such mappings were provided, they might entail the same limitations as the existing approaches based on mappings and translation, due to the gap in abstraction and level of expressiveness between OCL and the target languages. For instance, in Gecode [30], only the *Set* data type is supported, whereas the OCL supports many other collection data types, e.g., *Bag*, which cannot be directly translated into a *Set* (bags allow duplicated elements, whereas sets do not). As a consequence, it does not seem trivial (if possible at all) to translate the constraints using bags. COMET [30] is another constraint solver that requires constraints to be programmed in its own constraint programming language, which is a superset of Java/C++. Similar to Gecode, full translation of OCL into this language is either complex or not possible at all, e.g., COMET also only supports *Set*. Moreover, OCL supports OCL-specific data types such as *OCLAny*, *OCLVoid*, and *OCLInvalid* and UML-specific data types such as *OCLState* and *OCLMessage*, which may not be directly translated. In

addition, there are several OCL-specific operations such as *oclIsValid()* and *oclIsUndefined()* and UML-specific operations such as *oclIsInState()* and *isSignalSent()*, which are dependent on UML semantics.

Even when a translation of OCL to other constraint languages is feasible, such translation would incur a significant computational overhead, particularly in cases where there are significant differences in abstractions and no straightforward mappings. Depending on the time that a constraint solver takes to solve a constraint, such extra overhead might not be negligible when comparisons are made with solvers that work directly on OCL. To complicate things even further, even if we wanted to use a constraint solver that does not handle OCL directly, we would not only need to translate OCL constraints, but we would also need to translate metamodels/models (e.g., state machines) into the respective language of those constraint solvers. On the other hand, our constraint solver fully supports UML and the UML profiling mechanism, thus enabling the solving of constraints even on profiled models. This is one of the requirements in many of the case studies of our industrial partners, where we have to solve constraints on profiled UML models.

Most of the above approaches are different from our work, since we want to generate test data based on OCL constraints provided by modelers on UML state and class diagrams. These diagrams may be developed for environment models (for example, as in [32]) or system models (for example [33]) and the modeler should be allowed to use the entire standard (OCL 2.2). We want to provide inputs for which the constraints are satisfied, and not just verify if inputs comply with them. We also want a tool that can be easily integrated with different state-based testing approaches and is completely automated.

### **3.3 Search-based Heuristics for Model Based Testing**

The application of search-based heuristics for MBT has received significant attention recently (e.g., [34], [35]). The idea of these techniques is to apply heuristics to guide the search for test data that should satisfy different types of coverage criteria on state machines, such as state coverage. Achieving such coverage criteria is far from trivial since guards on transitions can be arbitrarily complex. Finding the right inputs to trigger these transitions is not simple. Heuristics have been defined based on common practices in

white-box, search-based testing, such as the use of branch distance and approach level [26]. Our goal is to tailor these heuristics to OCL constraint solving for test data generation. Instead of using search algorithms, another possible approach to cope with the combinatorial explosion faced in solving OCL constraints could be to use hybrid approaches that combine formal techniques (e.g., constraint solvers) with random testing (e.g. [36]). However, we are aware of no work on this topic for OCL and, even for common white-box testing strategies, performance comparisons of hybrid techniques with search algorithms are rare [37].

## 4. Definition of the Fitness Function for OCL

To guide the search for test data that satisfy OCL constraints, it is necessary to define a set of heuristics. 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 the target constraint.

In this paper, to generate test data to solve OCL constraints, we use a fitness function that is adapted from work done for code coverage (e.g., for branch coverage in C code [26]). In particular, we use the so called branch distance (a function  $d()$ ), as defined in [26]. The function  $d()$  returns 0 if the constraint is solved, otherwise a positive value that heuristically estimates how far the constraint was from being evaluated to true. As for any heuristic, there is no guarantee that an optimal solution (e.g., in our case, input data satisfying the constraints) will be found in reasonable time, but nevertheless many successful results based on such heuristics are reported in the literature for various software engineering problems [7]. In cases where we want a constraint to evaluate to false, we can simply negate 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.

In this section, we give examples of how to calculate the branch distance for various kinds of OCL expressions including primitive data types (such as *Real* and *Integer*) and collection-related types (such as *Set* and *Bag*). In OCL, all data types are subtypes of *OCLAny*, which is categorized into two subtypes: primitive types and collection types.

Primitive types are *Real*, *Integer*, *String*, and *Boolean*, whereas collection types include *Collection* as super type with subtypes *Set*, *OrderedSet*, *Bag*, and *Sequence*. A constraint can be seen as an expression involving one or more *Boolean* clauses connected with logical operators such as and and or. A constraint can be defined on variables of different types, such as equalities of integers and comparisons of strings. As an example, consider the UML class diagram in Figure 1 consisting of two classes: *University* and *Student*. Constraints on the class *University* are shown in Figure 2.



Figure 1. Example class diagram

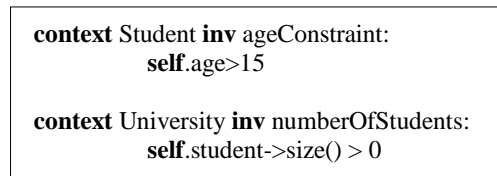


Figure 2. Example constraints

The first constraint states that the age of every Student should be greater than 15. Based on the type of attribute age of the class Student, which is Integer, the comparison in the clause is determined to involve integers. The second constraint states that the number of students in the university should be greater than 0. In this case, the *size()* operation, which is defined on collections in OCL and returns an Integer denoting the number of elements in a collection, is called on collection student (containing elements of the class Student). Even though an operation is called on a collection, the comparison is between two integers (return value from operation *size()* and 0).

Following we will discuss branch distance functions for different types of clauses in OCL.

#### 4.1 Primitive types

A Boolean variable *b* is either true (when the branch distance is 0, i.e.,  $d(b)=0$ ), or false (when  $d(b)=k$ , where *k* is an arbitrary positive constant, for example  $k=1$ ). If a *Boolean* variable is obtained from an operation call, then in general the branch distance would take one of only two possible values (0 or *k*). For example, when the operation *isEmpty()* is

called on a collection, the branch distance would either take  $0$  or  $k$ , unless a more fine grained specialized distance calculation is specified (e.g., returning the number of elements in the collection). For some types of OCL operations (e.g., *forall()*) we can provide more fine grained heuristics. We will provide more details on these operations and their corresponding branch distance calculations later in Section 4.2.

**Table 3. Branch distance calculations for OCL's operations for Boolean**

Boolean operations	Distance function
A	if A is true then 0 otherwise k
not A	if A is false then 0 otherwise k
A and B	$d(A)+d(B)$
A or B	$\min(d(A),d(B))$
A implies B	$d(\text{not } A \text{ or } B)$
if A then B else C	$d((A \text{ and } B) \text{ or } (\text{not } A \text{ and } C))$
A xor B	$d((A \text{ and } \text{not } B) \text{ or } (\text{not } A \text{ and } B))$

\* A and B are Boolean expressions or variables.

**Table 4. Branch distance calculations of OCL's relational operations for numeric data**

Relational operations	Distance function
$x=y$	if $\text{abs}(x-y) = 0$ then 0 otherwise $\text{abs}(x-y)+k$
$x \langle \rangle y$	if $\text{abs}(x-y) \langle \rangle 0$ then 0 otherwise k
$x < y$	if $x-y < 0$ then 0 otherwise $(x-y)+k$
$x \leq y$	if $x-y \leq 0$ then 0 otherwise $(x-y)+k$
$x > y$	if $(y-x) < 0$ then 0 otherwise $(y-x)+k$
$x \geq y$	if $(y-x) \leq 0$ then 0 otherwise $(y-x)+k$

The operations defined in OCL to combine *Boolean* clauses are *or*, *xor*, and, *not*, *if then else*, and *implies*. For these operations, branch distances are adopted from [26] since they work in a similar way as in programming languages and are shown in Table 3. Operations *implies*, and *xor* are syntax 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 predicate composed by two clauses is specified in Table 3 and can simply be computed for more than two clauses recursively.

```

if not (C1.ocllsKindOf(C2))
  d(C1=C2) := 1
otherwise if C1→size() <> C2→size()
  d(C1=C2) := 0.5 + 0.5*nor(d (C1→size()=C2 → size()))
otherwise
  d(C1 = C2) := 0.5 *  $\sum_{i=1}^{C1 \rightarrow size()} \text{nor}(d(\text{pair}_i)) / C1 \rightarrow size()$ 
  where, d(pairi) is the distance between elements in the ith
  position in the two sorted collections, e.g.,
  d(C1.at(i)=C2.at(i)) and nor is a normalizing function [38]
  defined as nor(x)=x/(x+1). Suppose C1 and C2 are two OCL
  collections.

```

**Figure 3. Branch distance equality of collections**

When a predicate or one of its parts is negated, then the predicate is transformed by moving the negation inward to the basic clauses, e.g., *not (A and B)* would be transformed into *not A or not B*.

For the numeric data types, i.e., *Integer* and *Real*, the relational operations that return *Booleans* (and so can be used as clauses) are *<*, *>*, *<=*, *>=*, and *<>*. For these operations, we adopted the branch distance calculation from [26] as shown in Table 4.

In OCL, several other operations are defined on *Real* and *Integer* such as *+*, *-*, *\**, */*, *abs()*, *div()*, *mod()*, *max()*, and *min()*. Since these operations are not used to compare two numerical values in clauses, there is no need to define a branch distance for them. For example, considering *a* and *b* of type *Integer* and a constraint  $a+b*3 < 4$ , then the operations *+* and *\** are used only to define the constraint. The overall result of the expression  $a+b*3$  will be an *Integer* and the clause will be considered as a comparison of two values of *Integer* type. For the *String* type, OCL defines several operations such as *=*, *+*, *size()*, *concat()*, *substring()*, and *toInteger()*. There are only three operations that return a Boolean: equality operator *=*, inequality *<>* and *equalsIgnoreCase()*. In these cases, instead of using *k* if the comparisons are false, we can return the value from any string matching distance function to evaluate how close any two strings are. In our approach, we implemented the edit distance [9] function, but any other string matching distance function can easily be incorporated.

## 4.2 Collection-Related Types

Collection types defined in OCL are *Set*, *OrderedSet*, *Bag*, and *Sequence*. Details of these

types can be found in the standard OCL specification [2]. OCL defines several operations on collections. An important point to note is that, if the return type of an operation on a collection is *Real* or *Integer* and that value is used in an expression, then the distance is calculated in the same way as for primitive types as defined in Section 4.1. An example is the *size()* operation, which returns an *Integer*. In this section, we discuss branch distances for operations in OCL that are specific to collections, and that usually are not common in programming languages for expressing constraints/predicates and hence are not discussed in the literature.

#### 4.2.1 Equality of collections (=)

In OCL constraints, we may need to compare the equality of two collections. We defined a branch distance for comparing collections as shown in Figure 3. The main goal is to improve the search process by providing a more fine grained heuristic than using a simple heuristic which simply calculates 0 if the result of an evaluation is true and  $k$  otherwise. In Figure 3, a branch distance for equality (=) of collections is calculated in one of the following three ways.

First, if collections  $C1$  and  $C2$  are not of the same kind, i.e., *not* ( $C1.oclIsKindOf(C2)$ ) evaluates to *true*, then the distance is simply 1. Note that any other constant could have been used to represent the maximum distance. Whenever, the distance is 1, it means that the collections are of different types, and the search algorithms must be guided to make the two collections of the same types.

Once the first condition is satisfied, the search algorithms must be guided such that the collections have equal number of elements. The second condition in the formula checks if the collections, which are of the same type, have different sizes. In that case, the search is guided to generate collections of equal size, i.e.,  $C1 \rightarrow size()=C2 \rightarrow size()$ . We compute  $d(C1 \rightarrow size()=C2 \rightarrow size())$  and since *size()* returns an integer, this distance calculation is simply performed using the equality operation on numerical data as shown in Table 4. The maximum distance value that can be taken by  $d(C1=C2)$  in this case can be derived as follows:

$$d(C1=C2) = 0.5 + 0.5 * \text{nor}(d(C1 \rightarrow size()=C2 \rightarrow size()))$$

using the formula of equality for numerical data from Table 4, we can derive:

$$d(C1=C2) = 0.5 + 0.5 * \text{nor}(\text{abs}(C1 \rightarrow \text{size}() - C2 \rightarrow \text{size}()) + k)$$

using the definition of  $\text{nor}(X)$ , and suppose  $Y = \text{abs}(C1 \rightarrow \text{size}() - C2 \rightarrow \text{size}()) + k$ , we can derive

$$d(C1=C2) = 0.5 + 0.5 * (Y / (Y+1))$$

In the above equation,  $Y/(Y+1)$  always computes a value less than  $1$ . Equation  $0.5 + 0.5 * (Y / (Y+1))$  therefore always takes a value between  $0.5$  to and  $1$ . Whenever,  $d(C1, C2)$  is greater than  $0.5$  and less than  $1$ , this means that collections do not have the same number of elements.

For the third condition, i.e., if collections are of the same type and have equal numbers of elements, the distance is calculated based on comparing elements in both collections. First, we sort both collections based on their elements, regardless of type of the collection (i.e., whether they are sets, bags or sequences). For sorting, a natural order among the elements must be defined. For instance, if collections consist of integers, then we simply sort based on the integer values. However, for other types of elements (e.g., enumerations), there is no pre-defined natural order and, in these cases, we sort using the name of the identifiers of elements (e.g., a sequence of enumerations  $\{B, A, D, C\}$  would be sorted into  $\{A, B, C, D\}$ ). If a collection consists of collections, we flatten the structure until we reach the primitive types and sort them based on all primitive types. Notice that how the sorting is done is not important. The important property that needs to be satisfied is that, if two collections are equal (regardless of the type of collection), then the sorting algorithm should produce the same paired alignment. For example, the set  $\{B, A, C\}$  is equal to  $\{C, B, A\}$  (the order in the sets has no importance), and their alignment using the name of enumeration elements produces the same sorted sequence  $\{A, B, C\}$ .

**Table 5. Minimum and Maximum Distance Values For Distance Calculation for Equality Of Collections**

Condition	Minimum	Maximum
not (C1.ocIsKindOf(C2))	1	1
$C1 \rightarrow \text{size}() \neq C2 \rightarrow \text{size}()$	$\geq 0.5$	$< 1$
not (C1.ocIsKindOf(C2)) and $C1 \rightarrow \text{size}() = C2 \rightarrow \text{size}()$	0	$< 0.5$

Once the element of both collections are sorted, we sum the distances between each pair of elements in the same position in the collections (i.e., distance between the  $i_{th}$  element of  $C1$  with the  $i_{th}$  element of  $C2$ ) and finally take the average by dividing the sum with the

number of elements in  $C1$ . When all elements of  $C1$  are equal to  $C2$ , then  $d(pair)$  yields 0 and as a result  $d(C1=C2) = 0$ . The maximum value  $d(C1=C2)$  can take in this case can be derived as follows:

$$d(C1 = C2) := 0.5 * \sum_{i=1}^{C1 \rightarrow size()} \text{nor}(d(\text{pair}_i)) / C1 \rightarrow size()$$

Using definition of  $\text{nor}$ , the above equation can be rewritten as:

$$d(C1 = C2) := 0.5 * \left( \sum_{i=1}^{C1 \rightarrow size()} d(\text{pair}_i) / (d(\text{pair}_i) + 1) \right) / C1 \rightarrow size()$$

$d(\text{pair}_i) / (d(\text{pair}_i) + 1)$  will always compute a value less than 1. Considering a simple example, in which collections consists of Boolean values, using the formula from Table 3,  $d(\text{pair}_i)$  can take  $k$  as the maximum value. So the formula will be reduced to:

$$d(C1 = C2) := 0.5 * \left( \sum_{i=1}^{C1 \rightarrow size()} k / (k + 1) \right) / C1 \rightarrow size()$$

$$d(C1=C2) := 0.5 * (k / (k + 1))$$

Since  $(k / (k + 1))$  computes a value below one, the above formula will always compute a value below 0.5. To further explain the computation of branch distance, when condition  $\text{not}(C1.\text{oclIsKindOf}(C2))$  and  $C1 \rightarrow size() = C2 \rightarrow size()$  is *true*, we provide an example below:

*Example 1:* Suppose  $C1 = \{2, 1, 3\}$ ,  $C2 = \{5, 4, 9\}$ , then the distance will be calculated as follows:

$$d(C1 = C2) := 0.5 * \sum_{i=1}^{C1 \rightarrow size()} \text{nor}(d(\text{pair}_i)) / C1 \rightarrow size()$$

$$d(C1 = C2) := 0.5 * \sum_{i=1}^3 \text{nor}(d(\text{pair}_i)) / 3$$

After sorting,  $C1$  and  $C2$  will be  $\{1, 2, 3\}$  and  $\{4, 5, 9\}$  respectively.

$$d(C1 = C2) := 0.5 * (\text{nor}(d(1 = 4)) + \text{nor}(d(2 = 5)) + \text{nor}(d(3 = 9))) / 3$$

Using  $k=1$  and formula of equality of two numeric values from Table 4

$$d(C1=C2) := 0.5 * (\text{nor}(4) + \text{nor}(4) + \text{nor}(7)) / 3$$

$$d(C1=C2) := 0.28$$

**Table 6. Branch distance calculation for operations checking objects in collections**

Operation	Distance function
includes (object:T): Boolean, where T is any OCL type	$\min_{i=1 \text{ to } self \rightarrow size() } d(\text{object} = self.at(i))$
excludes (object:T): Boolean, where T is any OCL type	$\sum_{i=1}^{self \rightarrow size()} d(\text{object} \langle \rangle self.at(i))$
includesAll (c:Collection(T)): Boolean, where T is any OCL type	$\sum_{i=1}^{self \rightarrow size()} \min_{j=1 \text{ to } c \rightarrow size()} d(c.at(i) = self.atj)$
excludesAll(c:Collection(T)): Boolean, where T is any OCL type	$\sum_{i=1}^{self \rightarrow size()} \sum_{j=1}^{self \rightarrow size()} d(c.at(i) \langle \rangle self.atj)$
isEmpty(): Boolean	$d(self \rightarrow size() = 0)$
notEmpty(): Boolean	$d(self \rightarrow size() \langle \rangle 0)$

As illustrated in Table 5 the three conditions in Figure 3 match three distinct value ranges, thus ensuring that the distance is always superior in the first case and the lowest in the third case, thus properly guiding the search.

#### 4.2.2 Operations checking existence of one or more objects in a collection

OCL defines several operations to check the existence of one or more elements in a collection such as *includes()* and *excludes()*, which check whether an object does or does not exist in a collection, respectively. Whether a collection is empty is checked with *isEmpty()* and *notEmpty()*. Such operations can be further processed for a more refined calculation of branch distance than simply calculating a distance  $0$  when an expression is *true* and  $k$  otherwise. The refined calculations of branch distances for these operations are described in Table 6.

For *includes (object:T)*, a branch distance is the minimum distance from all distances (calculated using the heuristic for equality as listed in Table 4) between object and each element of the collection (*self*) on which *includes* is invoked. When any element of *self* is equal to object, the distance will be  $0$ , and the overall distance will therefore be  $0$ . When none of the collection elements is equal to object, then we select the element in the collection with minimum distance. The example below illustrates how branch distance is calculated:

*Example 2:* Suppose  $C = \{1,2,3\}$  and we have an expression  $C \rightarrow \text{includes}(4)$ , then the branch distance will be calculated as:

$$d(C \rightarrow \text{includes}(4)) := \min_{i=1 \text{ to } \text{self} \rightarrow \text{size}() } d(\text{object} = \text{self.at}(i))$$

$$d(C \rightarrow \text{includes}(4)) := \min_{i=1 \text{ to } 3} d(\text{object} = C.at(i))$$

$$d(C \rightarrow \text{includes}(4)) := \min (d(1 = 4), d(2 = 4), d(3 = 4))$$

Using k=1, and formula of the equality of two integers from Table 4

$$d(C \rightarrow \text{includes}(4)) := \min (4,3,2)$$

$$d(C \rightarrow \text{includes}(4)) := 2$$

For *excludes (object:T)*, a branch distance is calculated in a similar way as *includes*, except that we use the distance heuristic for inequality (<>) and sum up the distances of all elements in the collection, which are equal to object. The example below illustrates how a branch distance is computed using the formula.

**Table 7. Branch distance for forAll and exists**

Operation	Distance function
forAll(v1,v2, ...vm exp)	if (self→size()) = 0 then 0 otherwise $\frac{\sum_{i_1=1}^{\text{self} \rightarrow \text{size}()} \sum_{i_2=1}^{\text{self} \rightarrow \text{size}()} \dots \sum_{i_m=1}^{\text{self} \rightarrow \text{size}()} d(\text{expr}(\text{self.at}(i_1), \dots, \text{self.at}(i_m)))}{(\text{self} \rightarrow \text{size}())^m}$
exists( v1,v2, ...vm exp)	$\min_{i_1, i_2, \dots, i_m \in 1 \text{ to } \text{self} \rightarrow \text{size}()} d(\text{expr}(\text{self.at}(i_1), \dots, \text{self.at}(i_m)))$
isUnique(v1 exp)	$\frac{\sum_{i=1}^{(\text{self} \rightarrow \text{size}() - 1)} \sum_{j=i+1}^{(\text{self} \rightarrow \text{size}())} d(\text{expr}(\text{self.at}(i)) <> \text{expr}(\text{self.at}(j)))}{((\text{self} \rightarrow \text{size}()) * (\text{self} \rightarrow \text{size}() - 1)) / 2}$
one( v1 exp)	$d(\text{self} \rightarrow \text{select}(\text{exp}) \rightarrow \text{size}() = 1)$

*Example 3:* Suppose  $C = \{1,2,2\}$  and we have an expression  $C \rightarrow \text{excludes}(2)$ , then

$$d(C \rightarrow \text{excludes}(2)) := \sum_{i=1}^{\text{self} \rightarrow \text{size}()} d(\text{object} <> \text{self.at}(i))$$

$$d(C \rightarrow \text{excludes}(2)) := \sum_{i=1}^3 d(\text{object} <> C.at(i))$$

$$d(C \rightarrow \text{excludes}(2)) := d(1 <> 2) + d(2 <> 2) + d(2 <> 2)$$

Using k=1, and formula from Table 4

$$d(C \rightarrow \text{excludes}(2)) := 0 + 1 + 1$$

$$d(C \rightarrow \text{excludes}(2)) := 2$$

In a similar fashion, we calculate branch distance of *includesAll* and *excludesAll* (Table 6), where we check if all elements of one collection are present/absent in another collection. For *includesAll*, we sum, over all elements of a collection, their minimum distance among all the elements of another collection as shown in the formula for *includesAll* in Table 6. For *excludesAll*, we sum all distances between all possible pairs of elements across the two collections, as shown in the formula for *excludesAll* in Table 6. Branch distance calculations for *isEmpty* and *notEmpty* are also defined in Table 6.

### 4.2.3 Branch distance for iterators

OCL defines several operations to iterate over collections. Below, we will discuss branch distances for these iterators.

The *forall()* iterator operation is applied to an OCL collection and takes as input a *boolean-expression*, then it determines whether the expression holds for all elements in the collection. To obtain a fine grained branch distance, we calculate the distance of the *boolean-expression* by computing the distance on all elements in the collection and summing the results. The function for *forall* presented in Table 7 is generic for any number of iterators. For the sake of clarity in the paper, we assume that function  $\text{expr}(v_1, v_2, \dots, v_m)$  in Table 7 evaluates an expression *expr* on a set of elements  $v_1, v_2, \dots, v_m$ . To explain *expr*, suppose we have a collection  $C = \{1, 2, 3\}$  and an expression  $C \rightarrow \text{forall}(x, y \mid x * y > 0)$ , then  $\text{expr}(C.\text{at}(1), C.\text{at}(2))$  entails calculating “ $d(x * y > 0)$ ”, where  $x = C.\text{at}(1)$ , i.e., 1 and  $y = C.\text{at}(2)$ , i.e., 2. The keyword *self* in the table refers to the collection on which an operation is applied,  $\text{at}(i)$  is a standard OCL operation that returns the  $i_{th}$  element of a collection, and  $\text{size}()$  is another OCL operation that returns the number of elements in a collection. The denominator  $(\text{self} \rightarrow \text{size}())^m$  is used to compute the average distance over all element combinations of size  $m$  since we have  $(\text{self} \rightarrow \text{size}())^m$  distance computations. Notice that calculating the average distance is important to avoid bias towards decreasing the size of the collection. For example, since it is a minimization problem (i.e., we want to minimize the branch distance), there would be a bias against larger collections as they would tend to have a higher branch distance (there is a number of branch distance additions that is polynomial in the number of iterators and collection size). A search operator that removes one element from the collection would always produce a better fitness function, so it would have a clear gradient toward the empty collection. An empty collection would make the

constraint true, but it can have at least two kinds of side effects: first, if a clause is conjuncted with other clauses that depend on the size (e.g.,  $C \rightarrow \text{forAll}(x/x>5)$  and  $C \rightarrow \text{size}()=10$ ), then there would likely be plateaus in the search landscape (e.g., gradient to increase the size towards 10 would be masked by the gradient towards the empty collection); second, because in our context we solve constraints to generate test data, we want to have useful test data to find faults, and not always empty collections. In general, to avoid side effects such as unnecessary fitness plateaus, our branch distance functions are designed in a way that, if there is no need to change the size of a collection to solve a constraint on it, then the branch distances should not have bias toward changing its size in one direction or another.

Below, we further illustrate the branch distance for *forAll* with the help of examples:

*Example 4:* Suppose we have a collection  $C = \{1,2,3\}$  and the expression is  $C \rightarrow \text{forAll}(x/x=0)$ . In this example, we have just one iterator  $x$ , and therefore  $m=1$ . In this case, the formula will be:

$$d(C \rightarrow \text{forAll}(x/x=0)) := \sum_{i_1=1}^{C \rightarrow \text{size}()} d(\text{expr}(C.\text{at}(i_1)) / C \rightarrow \text{size}())$$

The branch distance in this case will be calculated as:

$$d(C \rightarrow \text{forAll}(x \mid x=0)) := (d(\text{expr}(C.\text{at}(1))) + d(\text{expr}(C.\text{at}(2))) + d(\text{expr}(C.\text{at}(3))))/3$$

$$d(C \rightarrow \text{forAll}(x \mid x=0)) := (d(\text{expr}(1)) + d(\text{expr}(2)) + d(\text{expr}(3)))/3$$

Considering  $k=1$  and using the definition of *expr* and formulae from Table 3 and Table 4

$$d(C \rightarrow \text{forAll}(x \mid x=0)) := (2+3+4)/3$$

$$d(C \rightarrow \text{forAll}(x \mid x=0)) := 9/3 = 3$$

*Example 5:* Suppose we have a collection  $C = \{1,2\}$  and the expression is  $C \rightarrow \text{forAll}(x,y/x*y > 0)$ . In this case, we have two iterators  $x$  and  $y$  and thus the formula will become:

$$d(C \rightarrow \text{forAll}(x,y/x*y > 0)) := \frac{\sum_{i_1=1}^{C \rightarrow \text{size}()} \sum_{i_2=1}^{C \rightarrow \text{size}()} d(\text{expr}(C.\text{at}(i_1), C.\text{at}(i_2)))}{(C \rightarrow \text{size}())^2}$$

The branch distance in this case will be calculated as:

$$d(C \rightarrow \text{forAll}(x,y \mid x*y > 0)) := (d(\text{expr}(C.\text{at}(1), C.\text{at}(1))) + d(\text{expr}(C.\text{at}(1), C.\text{at}(2))) + d(\text{expr}(C.\text{at}(2), C.\text{at}(1))) + d(\text{expr}(C.\text{at}(2), C.\text{at}(2))))/4$$

$$d(C \rightarrow \text{forAll}(x,y \mid x*y > 0)) := (d(\text{expr}(1, 1)) + d(\text{expr}(1, 2)) + d(\text{expr}(2, 1)) + d(\text{expr}(2, 2)))/4$$

$$d(C \rightarrow \text{forAll}(x,y \mid x*y > 0)) := (d(1*1 > 0) + d(1*2 > 0) + d(2*1 > 0) + d(2*2 > 0))/4$$

Considering  $k=1$  and using formulae from Table 3 and Table 4

$$d(C \rightarrow \text{forAll}(x,y \mid x*y > 0)) := (0+0+0+0)/4$$

$$d(C \rightarrow \text{forAll}(x,y \mid x*y > 0)) := 0$$

In a similar fashion, the formula can be used for any number of iterators ( $m$ ).

The *exists()* iterator operation determines whether a *boolean-expression* holds for at least one element of the collection on which this operation is applied. The generic distance form for *exists()* is shown in Table 7. The definition of *exists()* is very similar to *forAll()* except for two differences. First, instead of summing distances across all element combinations of size  $m$ , we compute the minimum of these distances, since any element satisfying *exp* makes *exists()* *true*. Second, we do not have a denominator since no average needs to be computed. The *expr()* function works in the same way as for *forAll()*. Below we further illustrate branch distance calculation using two examples.

*Example 6:* Suppose we have a collection  $C = \{1,2,3\}$  and the expression is  $C \rightarrow \text{exists}(x/x=0)$ . In this example, we have just one iterator, i.e.,  $x$ . The formula will be:

$$d(C \rightarrow \text{exists}(x \mid x = 0)) := \min_{i_1 \in 1 \text{ to } 3} d(\text{expr}(C.\text{at}(i_1)))$$

The branch distance in this case will be calculated as:

$$d(C \rightarrow \text{exists}(x \mid x=0)) := \min (d(\text{expr}(C.\text{at}(1))), d(\text{expr}(C.\text{at}(2))), d(\text{expr}(C.\text{at}(3))))$$

$$d(C \rightarrow \text{exists}(x \mid x=0)) := \min (d(\text{expr}(1)), d(\text{expr}(2)), d(\text{expr}(3)))$$

Considering  $k=1$ , the definition of *expr*, and using formulae from Table 3 and Table 4

$$d(C \rightarrow \text{exists}(x \mid x=0)) := \min (2,3,4)$$

$$d(C \rightarrow \text{exists}(x \mid x=0)) := 2$$

*Example 7:* Suppose we have a collection  $C = \{1,2\}$  and the expression is  $C \rightarrow \text{exists}(x,y/x*y > 1)$ . In this case, we have two iterators  $x$  and  $y$  and thus the formula will become:

$$d(C \rightarrow \text{exists}(x,y \mid x * y > 0)) := \min_{i_1, i_2 \in 1 \text{ to } C \rightarrow \text{size}() } d(\text{expr}(C.\text{at}(i_1), C.\text{at}(i_2)))$$

The branch distance in this case will be calculated as:

$$d(C \rightarrow \text{exists}(x,y \mid x*y > 0)) := \min (d(\text{expr}(C.\text{at}(1), C.\text{at}(1))), d(\text{expr}(C.\text{at}(1), C.\text{at}(2))), \\ d(\text{expr}(C.\text{at}(2), C.\text{at}(1))), d(\text{expr}(C.\text{at}(2), C.\text{at}(2))))$$

$$d(C \rightarrow \text{exists}(x,y \mid x*y > 0)) := \min (d(\text{expr}(1, 1)), d(\text{expr}(1, 2)), d(\text{expr}(2, 1)), d(\text{expr}(2, 2)))$$

$$d(C \rightarrow \text{exists}(x,y \mid x*y > 0)) := \min (d(1*1 > 1), d(1*2 > 1), d(2*1 > 1), d(2*2 > 1))$$

Considering  $k=1$ , the definition of *expr*, and using formulae from Table 3 and Table 4

$$d(C \rightarrow \text{exists}(x,y \mid x*y > 0)) := \min (1,0,0,0)$$

$$d(C \rightarrow \text{exists}(x,y \mid x*y > 0)) := 0$$

In a similar fashion, as explained with Example 6 and Example 7, the formula can be used for any number of iterators ( $m$ ).

In addition, we also provide branch distance for *one()* and *isUnique()* operations in Table 7. The *one* operation returns true only if *exp* evaluates to true for exactly one element of the collection. The *isUnique()* operation returns true if *exp* on each element of the source collection evaluates to a different value. In this case, the distance is calculated by computing and summing the distances between each element of the collection and every other element in the collection. Since in this formula, we are computing  $((self \rightarrow size()) * (self \rightarrow size() - 1)) / (2)$  distances, we compute the average distance by using this formula in the denominator. Again, calculating the average distance is important to avoid bias in the search towards decreasing the size of the collection as we discussed for *forAll*. Below we provide an example of how we calculate branch distance for *isUnique()*.

**Table 8. Special Rules for Select() Followed By Size() when exp is false**

Operation	Distance function
>, >=	$d(\text{exp}) = \begin{cases} \text{if } C \rightarrow \text{size}() \leq z() & \text{then } (z() - C \rightarrow \text{size}()) + k \\ \text{else} & \text{nor}((z() - C \rightarrow \text{select}(P) \rightarrow \text{size}()) + k + \text{nor}(d(P))) \end{cases}$
<, <=	$d(\text{exp}) = \begin{cases} \text{if } C \rightarrow \text{size}() \geq z() & \text{then } (C \rightarrow \text{size}() - z()) + k \\ \text{else} & \text{nor}((C \rightarrow \text{select}(P) \rightarrow \text{size}() - z()) + k + \text{nor}(d(\text{not } P))) \end{cases}$
<>	$d(\text{exp}) = \begin{cases} \text{if } C \rightarrow \text{select}(P) \rightarrow \text{size}() = 0 & \text{then } d(P) \\ \text{if } C \rightarrow \text{select}(P) \rightarrow \text{size}() = C \rightarrow \text{size}() & \text{then } d(\text{not } P) \\ \text{if } 0 < C \rightarrow \text{select}(P) \rightarrow \text{size}() < C \rightarrow \text{size}() & \text{then } \min(d(P), d(\text{not } P)) \end{cases}$
=	$d(\text{exp}) = \begin{cases} \text{if } C \rightarrow \text{select}(P) \rightarrow \text{size}() > z() & \text{then } (C \rightarrow \text{select}(P) \rightarrow \text{size}() - z()) + k + \text{nor}(d(\text{not } P)) \\ \text{if } C \rightarrow \text{select}(P) \rightarrow \text{size}() < z() & \text{then } (z() - C \rightarrow \text{size}()) + k + \text{nor}(d(P)) \end{cases}$

\* In the above table,  $k=1$ ,  $\text{nor}(x) = x/(x+1)$  and  $d(P)$  is simply the sum of  $d()$  over the elements in  $A$

*Example 8:* Suppose we have a collection  $C = \{1,1,3\}$  and the expression is  $C \rightarrow isUnique(x/x)$ . In this example, we have just one iterator, i.e.,  $x$ . Using the formula of branch distance for  $isUnique()$ ,

$$d(C \rightarrow isUnique(x|x)) := \frac{\sum_{i=1}^{(C \rightarrow size() - 1)} \sum_{j=i+1}^{(C \rightarrow size())} d(\text{expr}(C.at(i)) \langle \rangle \text{expr}(C.at(j)))}{((C \rightarrow size()) * (C \rightarrow size() - 1)) / 2}$$

$$d(C \rightarrow isUnique(x|x)) := (d(\text{expr}(C.at(1)) \langle \rangle \text{expr}(C.at(2))) + d(\text{expr}(C.at(1)) \langle \rangle \text{expr}(C.at(3))) + d(\text{expr}(C.at(2)) \langle \rangle \text{expr}(C.at(3)))) / ((3 * 2) / 2)$$

$$d(C \rightarrow isUnique(x|x)) := (d(1 \langle \rangle 1) + d(1 \langle \rangle 3) + d(1 \langle \rangle 3)) / 3$$

$$d(C \rightarrow isUnique(x|x)) := (1 + 0 + 0) / 3$$

$$d(C \rightarrow isUnique(x|x)) := 1 / 3$$

Select, reject, collect operations select a subset of elements in a collection. The *select()* operation selects all elements of a collection for which a *Boolean* expression is true, whereas *reject()* selects all elements of a collection for which a *Boolean* expression is false. In contrast, the *collect()* iterator may return a subset of elements that does not belong to the collection on which it is applied. Since all these iterators (like the generic iterator operation) return a collection and not a *Boolean* value, we do not need to define branch distance for them, as discussed in Section.4.1. However, an iterator operation (such as *select()*) followed by another OCL operation, for instance *size()*, can be combined to make a Boolean expression of the following form:

$$\text{exp} = C \rightarrow \text{selectionOp}(P) \rightarrow \text{size}() \text{ RelOp } z()$$

Where  $C$  is a collection, *selectionOp* is either select, reject, or collect,  $P$  is a *boolean-expression*, *RelOp* is a relational operation from set  $\{\langle, \leq, =, \langle \rangle, \rangle, \geq\}$ , and  $z()$  is a function that returns a constant. A simple way of calculating branch distance for the above example, when *RelOp* is  $=$ , and *selectionOp* is *select* would be as follows:

$$\text{exp} = C \rightarrow \text{select}(P) \rightarrow \text{size}() = z()$$

If  $\text{exp} = \text{true}$  then

$$d(\text{exp}) = 0$$

else

$$d(\text{exp}) = |C \rightarrow \text{select}(P) \rightarrow \text{size}() - z()| + k$$

An obvious problem of calculating branch distance in this way is that it does not give any gradient at all to help search algorithms solve  $P$ , which can be arbitrarily complex. To

optimize branch distance calculation in this particular case, we need special rules that are defined specifically for each *RelOp*.

For  $>$  and  $\geq$ , when *exp* is *false*, this means that the size of resultant collection of the expression  $C \rightarrow \text{select}(P)$  is less than the size which will make the branch distance 0. In this case, first we need a collection with size greater than  $z()$ , and then we need to obtain those elements of  $A$  that increase the value of  $\text{size}()$  returned by  $C \rightarrow \text{select}(P) \rightarrow \text{size}()$ . This can be achieved by the rule shown in the first row of Table 8. The normalization function  $\text{nor}()$  is necessary because the branch distance should first reward any increase in  $C \rightarrow \text{size}()$  until it is greater than  $z()$  regardless of the evaluation of  $P$  on its elements. Then, once the collection  $C$  has enough elements, we need to account for the number of elements for which  $P$  is true by using  $((z() - C \rightarrow \text{select}(P) \rightarrow \text{size}()) + k)$ . The function  $d(P)$  returns the sum of branch distance evaluations of a predicate  $P$  over all the elements in  $C$  and provides additional gradient by quantifying how close are collection elements from satisfying  $P$ . Below, we further illustrate this case with an example:

*Example 9:* Suppose we have a collection  $C = \{1, 1, 3\}$  and the expression is  $C \rightarrow \text{select}(x|x > 1) \rightarrow \text{size}() \geq 3$ . Using the formula of branch distance for the case when *RelOp* is  $>$ ,  $\geq$ .

In this case,  $C \rightarrow \text{size}()$  is 3, which is equal to  $z()$ , i.e., 3, so the formula for branch distance calculation is:

$$d(C \rightarrow \text{select}(x|x > 1) \rightarrow \text{size}() \geq 3) := \text{nor}((z() - C \rightarrow \text{select}(P) \rightarrow \text{size}()) + k + \text{nor}(d(P)))$$

Assuming  $k=1$ ,

$$d(C \rightarrow \text{select}(x|x > 1) \rightarrow \text{size}() \geq 3) := \text{nor}((3-1)+1 + \text{nor}(d(x > 1)))$$

$$d(C \rightarrow \text{select}(x|x > 1) \rightarrow \text{size}() \geq 3) := \text{nor}(3 + \text{nor}(d(x > 1)))$$

$$d(C \rightarrow \text{select}(x|x > 1) \rightarrow \text{size}() \geq 3) := \text{nor}(3 + \text{nor}(d(1 > 1) + d(1 > 1) + d(3 > 1)))$$

Using  $k=1$ , and formula from Table 4

$$d(C \rightarrow \text{select}(x|x > 1) \rightarrow \text{size}() \geq 3) := \text{nor}(3 + \text{nor}(1+1+0))$$

$$d(C \rightarrow \text{select}(x|x > 1) \rightarrow \text{size}() \geq 3) := \text{nor}(3 + 0.667)$$

$$d(C \rightarrow \text{select}(x|x > 1) \rightarrow \text{size}() \geq 3) := 0.78$$

For  $<$  and  $\leq$ , when *exp* is *false*, this means that  $C \rightarrow \text{select}(P) \rightarrow \text{size}()$  is greater than the size which will make the branch distance 0. Similar to the previous case, the

distance computation account for those elements of  $C$  that decrease the value of  $size()$  returned by  $C \rightarrow select(P) \rightarrow size()$ , and uses  $nor(d(not P))$  to provide additional gradient to the search, as shown in second row of Table 8.

For the cases when the value of  $RelOp$  is inequality ( $<>$ ), the rule is shown in the third row of Table 8. Recall that our expression is in the following format:  $exp = C \rightarrow selectionOp(P) \rightarrow size() RelOp z()$ . For this rule, there are three cases based on the value of  $C \rightarrow select(P) \rightarrow size()$ . Recall that  $d(P)$  is simply the sum of all  $d()$  on all elements of  $C$ . The first case is when  $C \rightarrow select(P) \rightarrow size() = 0$ , where  $P$  does not hold for any element in  $C$ . To guide the search towards increasing the size of the collection,  $d(exp)$  will be  $d(P)$  so as to minimize the sum of distances of all elements with  $P$ . The second case is when  $P$  is *true* for all elements of  $C$ , which means that  $C \rightarrow select(P) \rightarrow size() = C \rightarrow size()$ . To guide the search in decreasing the size of the collection, for reasons that are similar to the first case, we define  $d(exp)$  as  $d(not P)$ . When  $0 < C \rightarrow select(P) \rightarrow size() < C \rightarrow size()$ , we can guide the search to either increase or decrease the size of the collection and thus define  $d(exp)$  as  $min(d(P), d(not P))$ .

For the cases when the value of  $RelOp$  is equality ( $=$ ), the rule is shown in the fourth row of Table 8. There are two important cases, which work in a similar way as the first and second cases as reported in Table 8. The first case is when  $C \rightarrow select(P) \rightarrow size() > z()$ , where we need to decrease  $C \rightarrow select(P) \rightarrow size()$ , which can be achieved by minimizing  $(C \rightarrow select(P) \rightarrow size() - z()) + k + nor(d(not P))$ . The second case is when  $C \rightarrow select(P) \rightarrow size() < z()$ . For this case, we need to increase the number of elements in  $C$  for which  $P$  holds and must minimize  $(z() - C \rightarrow select(P) \rightarrow size()) + k + nor(d(P))$ .

Note that we only presented formulae in Table 8 for the cases when the iterator operation considered  $selectionOp$  is *select*, however, the formulae can simply be extended for other iterator operations. The *collect* operation works in the same way as *select*, and hence the formulae in Table 8 can simply be adapted by replacing *select* with *collect* in the formulae. For instance, for the case when  $RelOp$  is  $>$  or  $>=$ , formula for *collect* would be:

$$d(exp) = (z() - C \rightarrow collect(P) \rightarrow size()) + k + nor(d(P))$$

The *reject* operation works in a different way than *select* since it rejects all those elements for which a *Boolean* expression is *true*. But  $reject(P)$  can be simply transformed into  $select(not P)$ .

In addition to the rules for an iterator followed by *size()*, we defined two new rules when a *select()* is followed by *forall()* or *exists()* that are shown in Table 9. For example,  $C \rightarrow \text{select}(P1) \rightarrow \text{forall}(P2)$  (first row in Table 9) implies that for all elements of  $C$  for which  $P1$  holds,  $P2$  should also hold. In other words,  $P1$  implies  $P2$ . Therefore,  $C \rightarrow \text{select}(P1) \rightarrow \text{forall}(P2)$  can simply be transformed into  $C \rightarrow \text{forall}(P1 \text{ implies } P2)$ . Similarly, a *select()* followed by an *exists()* can simply be transformed into *exists()* ( $P1$  and  $P2$ ). This means that there should be at least one element in  $C$  for which  $P1$  and  $P2$  holds. Notice that a sequence of selects can be simply combined, e.g.,  $C \rightarrow \text{select}(P1) \rightarrow \text{select}(P2)$  is equivalent to  $C \rightarrow \text{select}(P1 \text{ and } P2)$ .

The effectiveness of all these rules for calculating branch distance is empirically evaluated in Section 6.

### 4.3 Tuples in OCL

In OCL several different values can be grouped together using tuples. A tuple consists of different parts separated by a comma and each part specifies a value. Each value has an associated name and type. For example, consider the following example of a tuple in OCL:

```
Tuple{firstName = "John", age= 29}
```

This tuple defines a *String* *firstName* of value “John” and an *Integer* *age* of value 29. Each value is accessed via its name. For example, `Tuple{firstName = “John”,age= 29}.age` returns 29. There are no operations allowed on tuples in OCL because they are not subtypes of *OCLAny*. However, when a value in a tuple is accessed and compared, a branch distance is calculated based on the type of the value and the comparison operation used. For example, consider the following constraint:

```
Tuple{String: firstName = “John”, Integer: age= 29}.age > 20
```

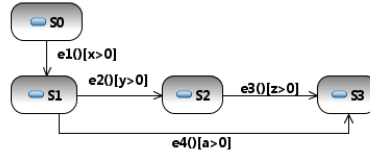
In this case, since *age* is an *Integer* and comparison operation is  $>$ , we use the branch distance calculation of numerical data for the case of  $>$  as defined in Table 4.

### 4.4 Special Cases

In this section, we will discuss branch distance calculations for some special cases including enumerations and other special operations provided by OCL, such as for example *oclInState*.

**Table 9. Special Rules for Select() Followed by ForALL and Exists**

Operation	Distance function
$C \rightarrow \text{select}(P1) \rightarrow \text{forAll}(P2)$	$d(C \rightarrow \text{forAll}(P1 \text{ implies } P2))$
$C \rightarrow \text{select}(P1) \rightarrow \text{exists}(P2)$	$d(C \rightarrow \text{exists}(P1 \text{ and } P2))$



**Figure 4. A dummy example to explain *oclInState()***

#### 4.4.1 Enumerations

Enumerations are datatypes in OCL that have a name and a set of enumeration literals. An enumeration can take any one of the enumeration literals as its value. Enumerations in OCL are treated in the same way as enumerations in programming languages such as Java. Because enumerations are objects with no specific order relation, equality comparisons are treated as basic Boolean expressions, whose branch distance is either 0 or k.

#### 4.4.2 *oclInState*

The *oclInState(s:OclState)* operation returns *true* if an object is in a state represented by *s*, otherwise it returns *false*. This operation is valid in the context of UML state machines to determine if an object is in a particular state of the state machine. *OclState* is a datatype similar to enumeration. This datatype is only valid in the context of *oclInState* and is used to hold the names of all possible states of an object as enumeration literals. In this particular case, the states of an object are not precisely defined, i.e., each state of the object is uniquely identified based on the names of the states. For example, a class *Light* having two states: *On* and *Off*, is modeled as an enumeration with two literals *On* and *Off*. In this example, *s:OclState* takes either *On* or *Off* value and the branch calculation is same as for enumerations. However, if the states are defined as state invariants, which is a common way of defining states in a UML state machine as an OCL constraint [2], then the branch distance is calculated based on two special cases depending on whether we can directly set the state of an object by manipulating the state variables or not. Below, we will discuss each case separately.

The first case is when the state of an object can be manipulated by directly setting its state defining attributes (or properties) to satisfy a state invariant. In this case, state invariants—which are OCL constraints—can be satisfied by solving the constraints based on heuristics defined in the previous sections. Note that each state in a state machine is uniquely identified by a state invariant and there is no overlapping between state invariants of any two states (strong state invariants [39]). For instance, in our industrial case study, we needed to emulate faulty situations in the environment for the purpose of robustness testing, which were modeled as OCL constraints defined on the properties of the environment. In this case, it was possible to directly manipulate the properties of the environment emulator based on which the state of the environment is defined and each state was uniquely identified based on its state invariant. A simple example of such state invariant for the environment is given below:

$$\text{self.packetLoss.value} > 5 \text{ and } \text{self.packetLoss.value} \leq 10$$

The above state invariant defines a faulty situation in the environment, when the value of packet loss in the environment is greater than 5% and less or equal to 10%. This constraint can easily be solved using the heuristics defined in the previous sections and the value of packetLoss generated by our constraint solver can be directly set for the environment.

In the second case, when it is not possible to directly set the state of an object, the approach level heuristic [26] can be used in conjunction with branch distance to make the object reach the desired state. We will explain this case using a dummy example of a UML state machine shown in Figure 4. The approach level calculates the minimum number of transitions in the state machine to reach the desired state from the closest executed state. For instance, in Figure 4, if the desired state is *S3* and currently we are in *S1*, then the approach level is *1*. By calculating the approach level for the states that the object has reached, we can obtain a state that is closest to the desired state (i.e., it has the minimum approach level). In our example, the closest state based on the approach level is *S1*. Now, the goal is to transition in the direction of the desired state in order to reduce the approach level to *0*. This goal is achieved with the help of branch distance. The branch distance is used to heuristically score the evaluation of the OCL constraints on the path from the current state to the desired state (e.g., guards on transitions leading to the desired state). The distance is calculated based on the heuristics defined in this paper. The branch

distance is used to guide the search to find test data that satisfy these OCL constraints. An event corresponding to a transition can occur several times but the transition is only triggered when the guard is true. The branch distance is calculated every time the guard is evaluated to capture how close the values used are from solving the guard. In the example, we need to solve guard ' $a > 0$ ' so that whenever  $e4()$  is triggered we can reach  $S3$ . Since the guards are written in OCL, they can be solved using the heuristics defined in the previous sections. In the case of MBT, it is not always possible to calculate the branch distance when the related transition has never been triggered. In these cases, we assign to the branch distance its highest possible value. More details on this case can be found for example in [40].

#### 4.4.3 *oclIsTypeOf(),oclIsKindOf(), and oclIsNew()*

These three operations are special operations defined for all objects in the OCL. The *oclIsTypeOf( $t:Classifier$ )* returns true if  $t$  and the object on which this operation is called have the same type. The *oclIsKindOf( $t:Classifier$ )* operation returns true if  $t$  is either the direct type or one of the supertypes of the object on which the operation is called. The operation *oclIsNew()* returns true if the object on which the operation is called is just created. These three operations are defined to check the properties of objects and hence are not used for test data generation, therefore we do not explicitly define branch distance calculation for these operations. However, whenever these operations are used in constraints, the branch distance is calculated as follows: if the invocation of an operation evaluates to true, then the branch distance is 0, else the branch distance is  $k$ , as for any *boolean* function for which more fine grained heuristic is not provided.

#### 4.4.4 *User-defined Operations*

Apart from the operations defined in the standard OCL library, OCL also provides a facility for the users to define new operations. Body of these operations is written using OCL expressions and may call the standard OCL library operations. As we discussed in Section 4, we only provide specialized branch distance calculations for the operations defined in the standard OCL library. For user-defined operations, we calculate a branch distance according to the return types of these operations. If a user-defined operation returns a Boolean, to provide more fine grained fitness functions, it is possible to use testability transformations on those operations, as for example in search-based software

testing of Java software [41]. In our tool, we have not implemented and evaluated this type of testability transformations, and further research would be needed to study their applications in OCL. For any other return type but Boolean, we define a branch distance using the rules defined in Section 4.4. For instance, consider a user-defined OCL operation named *operation1()*, which is defined on a collection and returns a collection, and the following constraint defined on it:

$$c1 \rightarrow \text{operation1}() \rightarrow \text{isEmpty}()$$

In this case, the branch distance is calculated based on the heuristic for *isEmpty()* as defined in Section 4.2.

**Table 10. Statistics of Complexity of Constraints**

# of Clauses	Frequency
8	1
7	8
6	23
5	10
2	6
1	9

**Table 11. OCL Data Types Used in Constraints**

OCL Data Types Used	Frequency
Integer	13
Boolean	2
Integer and Enumeration	31
Integer, Enumeration, and Boolean	11

```
context Saturn inv synchronizationConstraint:
  self.media.synchronizationMismatch.value > self.media.synchronizationMismatchThreshold.value)
```

**Figure 5. A constraint checking synchronization of audio and video in a videoconference**

## 5. Case study: Robustness Testing of Video Conference System

This case study is part of a project aiming to support automated, model-based robustness testing of a core subsystem of a video conference system (VCS) called Saturn [42], developed by Cisco Systems, Inc, Norway. Saturn is modeled as a UML class diagram meant to capture information about APIs and system (state) variables, which are required to generate executable test cases in our application context. The standard behavior of the system is modeled as a UML 2.0 state machine. In addition, we used Aspect-oriented

Modeling (AOM) and more specifically the AspectSM profile [33] to model 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 SUT or its environment such as the network and other systems interacting with the SUT. A weaver later on weaves robustness behavior into the standard behavior and generates a standard UML 2.0 state machine. More details and models of the case study, including a partial woven state machine, are provided in [33]. The woven state machine produced by the weaver is used for test case generation. In the current, simplified case study, the woven state machine has 12 states and 103 transitions. Out of these 103 transitions, only 83 transitions model robustness behavior as change events and 57 transitions out of these 83 have identical change conditions, including 42 constraints using *select()* and *size()* operations. A change event is defined with a ‘when’ condition and it is triggered when this condition is met during the execution of a system. An example of such a change event is shown in Figure 5. This change event is fired during a videoconference when the synchronization between audio and video passes the allowed threshold. *synchronizationMismatch* is a non-functional property defined using the MARTE profile [43], which measures the synchronization between audio and video in time. In order to traverse these transitions appropriate test data is required that satisfies the constraints specified as guards and when conditions (in case of change events). The complexity of these constraints, which are all in a conjunctive normal form, is reported in Table 10 in terms of number of clauses. Most constraints contain between 6 and 8 clauses. The different OCL data types used in these constraints are shown in Table 11 and we can see that all primitive types are being used in our case study.

In our case study, we target test data generation for model-based robustness testing of the VCS. Testing is performed at the system level and we specifically target robustness faults, for example related to faulty situations in the network and other systems that comprise the environment of the SUT. Test cases are generated from the system state machines using our tool TRUST [42]. To execute test cases, we need appropriate data for the state variables of the system, state variables of the environment (network properties and in certain cases state variables of other VCS), and input parameters that may be used in the following UML state machine elements: (1) guard conditions on transitions, (2) change events as triggers on transitions, and (3) inputs to time events. We have successfully used

the TRUST tool to generate test cases using different coverage criteria on UML state machines, such as all transitions, all round trip, modified round trip strategy [44].

## 5.1 Empirical Evaluation

This section discusses the experiment design, execution, and analysis of the evaluation of the proposed OCL test data generator on the VCS case.

### 5.1.1 Experiment Design

We designed our experiment using the guidelines proposed in [8, 45]. The objective of our experiment is to assess the efficiency of search algorithms such as GAs to generate test data by solving OCL constraints. In our experiments, we compared four search techniques: AVM, GA, (1+1) EA, and RS (Section 4). 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 in previous software testing work we found that it can be more effective in some cases (e.g., see [38]). We used RS as the comparison baseline to assess the difficulty of the addressed problem [8].

From this experiment, we want to answer the following research questions.

**RQ1:** Are search-based techniques effective and efficient at solving OCL constraints in industrial system models?

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

### 5.1.2 Experiment Execution

We ran experiments for 57 OCL predicates from the VCS industrial case study that we discussed earlier. The number of clauses in each predicate varies from one to eight and the median value is six. The complexity of the problems is summarized in Table 10, where we provide details on the distribution of numbers of clauses. In Table 11, we summarized the data types and OCL specific operations used in the problems.

Fitness evaluations are computationally expensive, as they require the instantiation of models on which the constraints are evaluated on. Each algorithm was run 100 times to account for the random variation inherent to randomized algorithms [46], which for our

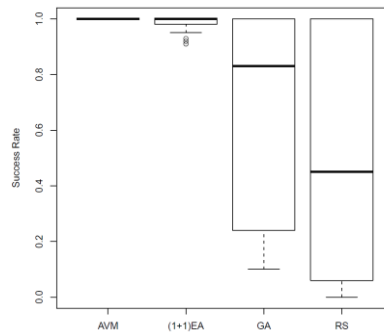
case study was enough to gain enough statistical confidence on the validity of our results. We ran each algorithm up to 2000 fitness evaluations on each problem and collected data on whether an algorithm found a solution or not. On our machine (Intel Core Duo CPU 2.20 GHz with 4 GB of RAM, running Microsoft Windows 7 operating system), running 2000 fitness evaluations takes on average 3.8 minutes for all algorithms. The number of fitness evaluations should not be too high to enable enough experimentations on different constraints within feasible time, but should still represent a reasonable “budget” in an industrial setting (i.e., the time the software testers are willing to wait when solving constraints to generate system level test cases).

Instead of putting a limit to the number of fitness evaluations, in practice we can put a limit on time depending on practical constraints. This mean we can run a search algorithm with as many iterations as possible and stop once a predefined time threshold is reached (e.g., 10 minutes) if the constraint has not been solved yet. The choice of this threshold could be driven by the testing budget. However, though useful in practice, using a time threshold would make it significantly more difficult and less reliable to compare different search algorithms (e.g., accurately monitoring the passing of time, side effects of other processes running at same time, inefficiencies in implementation details).

A solution is represented as an array of variables, the same variables that appear in the OCL constraint we want to solve. For the used 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 [46]. As we will show, our constraint solver is already very effective in solving OCL constraints, so we did not feel the need for tuning to improve the performance even further.

To compare the algorithms, we calculated their success rates. The success rate of an algorithm is defined in general as the number of times it was successful in finding a solution out of the total number of runs. In our context, it is the success rate in solving constraints.

Figure 6 shows a box plot representing the success rates of the 57 problems for AVM, (1+1) EA, GA, and RS. For each search technique, the box-plot is based on 57 success rates, one for each constraint. The results show that AVM not only outperformed all the other three algorithms, i.e., (1+1) EA, RS, and GA but in addition achieved a consistent success rate of 100%. (1+1) EA outperformed GA and RS and achieved an average success rate of 98%. Finally, GA outperformed RS, where GA achieved an average success rate of 65% and RS attained an average success rate of 49%. We can observe that, with an upper limit of 2000 iterations, (1+1) EA achieves a median success rate of 98% and GA exceeds a median of roughly 80%, whereas RS could not exceed a median of roughly 45%. We can also see that all success rates for (1+1) EA are above 90% and most of them are close to 100%.



**Figure 6. Success rates for various algorithms**

Table 12 shows success rates for individual problems to further analyze the results. We observe that problems 42 to 56 were solved by all the algorithms. The reason is that these problems are the simplest problems comprising of either one or two clauses, as it can be seen from the complexity column in Table 12 and Table 15. The problems with higher complexity (higher number of clauses) are the most difficult to solve for GA and RS, as shown in Table 15. As the complexity is increasing, the success rates of GA and RS are decreasing. However, in the case of AVM and (1+1) EA, we do not see a similar pattern. AVM managed to maintain the average success rate of 100% even for the most complex problems. In the case of (1+1) EA, the minimum average success rates are for the problems with complexity of seven clauses, which is 95%. Based on these results, we can see that our approach is effective and efficient, and therefore practical, even for difficult constraints (RQ1).

**Table 12 . Success rates For Individual Problems**

Problem Id	Complexity	AVM	(1+1)EA	GA	RS
0	8	1	0,98	0,21	0,02
1	5	1	1	0,95	0,83
2	7	1	0,91	0,17	0,01
3	7	1	0,95	0,15	0,01
4	7	1	0,92	0,1	0,01
5	7	1	0,96	0,11	0
6	6	1	1	0,87	0,68
7	6	1	0,99	0,88	0,59
8	5	1	0,98	0,84	0,53
9	5	1	1	0,83	0,45
10	5	1	1	0,81	0,33
11	5	1	0,98	0,78	0,39
12	7	1	1	0,29	0,07
13	6	1	1	0,54	0,3
14	6	1	0,95	0,3	0,06
15	6	1	0,95	0,25	0,1
16	6	1	1	0,19	0,02
17	6	1	0,98	0,24	0,04
18	7	1	0,96	0,34	0,11
19	6	1	1	0,6	0,12
20	6	1	0,98	0,25	0,04
21	6	1	0,97	0,23	0,04
22	6	1	0,99	0,18	0,04
23	6	1	1	0,17	0,05
24	6	1	1	0,91	0,67
25	5	1	1	1	0,93
26	5	1	0,99	0,88	0,42
27	5	1	1	0,75	0,51
28	5	1	1	0,77	0,4
29	6	1	0,99	0,16	0,08
30	7	1	0,96	0,37	0,13
31	6	1	1	0,55	0,15
32	6	1	0,96	0,19	0,02
33	6	1	0,93	0,21	0,07
34	6	1	0,96	0,21	0,02
35	6	1	0,98	0,23	0,04
36	6	1	1	0,95	0,93
37	5	1	1	0,99	1
38	5	1	0,99	0,89	0,76
39	5	1	1	0,86	0,7
40	6	1	1	0,9	0,59
41	5	1	1	0,84	0,65
42	1	1	1	1	1
43	1	1	1	1	1
44	1	1	1	1	1
45	1	1	1	1	1
46	1	1	1	1	1
47	1	1	1	1	1
48	2	1	1	1	1
49	1	1	1	1	1
50	1	1	1	1	1
51	2	1	1	1	1
52	2	1	1	1	1
53	1	1	1	1	1
54	2	1	1	1	1
55	1	1	1	1	1
56	2	1	1	1	1

Table 13. Results for The Fisher's Exact Test at Significance Level of 0.05

ID	AVM vs (1+1) EA		AVM vs GA		AVM vs RS		(1+1 EA) vs GA		(1+1) EA vs RS		GA vs RS	
	P-Value	OR	p-Value	OR	p-Value	OR	P-Value	OR	p-Value	OR	p-Value	OR
0	0,49	5	3,75E-36	743	1,14E-55	7919	8,33E-33	146	5,41E-52	1552	2,50E-05	1
1	1	1	0,059	12	7,26E-06	42	0,05	12	7,26E-06	42	0,01	3
2	0,003	21	2,64E-39	959	2,23E-57	13333	5,39E-28	46	7,96E-45	639	7,48E-05	13
3	0,059	12	5,29E-41	1108	2,23E-57	13333	1,15E-33	96	1,95E-49	1151	0,0003	12
4	0,006	18	1,04E-45	1732	2,23E-57	13333	7,47E-35	94	6,70E-46	722	0,009	8
5	0,12	9	1,05E-44	1564	2,21E-59	40401	2,01E-38	167	1,02E-52	4310	0,0007	26
6	1	1	0,0001	31	2,41E-11	95	0,0001	31	2,41E-11	95	0,002	3
7	1	3	0,0003	28	5,03E-15	140	0,002	9	1,35E-13	46	4,85E-06	5
8	0,49	5	1,59E-05	39	1,11E-17	178	0,0007	8	5,69E-15	35	3,59E-06	5
9	1	1	7,26E-06	42	1,59E-21	245	7,26E-06	42	1,59E-21	245	2,91E-08	6
10	1	1	1,48E-06	48	4,05E-28	405	1,48E-06	48	4,05E-28	405	6,86E-12	8
11	0,49	5	1,30E-07	57	1,12E-24	312	1,21E-05	11	1,14E-21	61	3,17E-08	5
12	1	1	1,33E-30	487	5,76E-49	2505	1,33E-30	487	5,76E-49	2506	7,42E-05	5
13	1	1	3,17E-17	171	5,77E-30	465	3,17E-17	171	5,77E-30	465	0,0009	2
14	0,059	12	5,77E-30	465	3,77E-50	2922	2,68E-23	40	2,01E-42	252	1,26E-05	6
15	0,059	12	2,87E-33	595	1,04E-45	1732	2,26E-26	51	3,71E-38	150	0,008	3
16	1	1	1,08E-37	840	1,14E-55	7919	1,08E-37	840	1,14E-55	7919	0,0001	9
17	0,49	5	5,75E-34	627	1,02E-52	4310	1,13E-30	123	4,47E-49	844	5,87E-05	7
18	0,12	9	1,60E-27	387	1,05E-44	1564	4,57E-22	41	2,01E-38	167	0,0001	4
19	1	1	1,34E-14	134	9,76E-44	1423	1,34E-14	134	9,76E-44	1423	9,47E-13	11
20	0,49	5	2,87E-33	595	1,02E-52	4310	5,40E-30	116	4,47E-49	845	2,99E-05	7
21	0,24	7	1,11E-34	663	1,02E-52	4310	4,93E-30	92	1,42E-47	597	0,0001	7
22	1	3	1,73E-38	896	1,02E-52	4310	1,22E-36	296	9,48E-51	1422	0,002	5
23	1	1	2,64E-39	959	2,13E-51	3490	2,64E-39	959	2,13E-51	3490	0,01	4
24	1	1	0,003	20	9,76E-12	99	0,003	21	9,76E-12	100	4,55E-05	5
25	1	1	1	1	0,01	16	1	1	0,01	16	0,01	16
26	1	3	0,0003	28	4,54E-23	276	0,002	9	1,90E-21	91	6,44E-12	10
27	1	1	1,07E-08	68	1,32E-18	193	1,07E-08	68	1,32E-18	193	0,0007	3
28	1	1	5,71E-08	61	3,90E-24	300	5,71E-08	61	3,90E-24	300	1,67E-07	5

**Table 14. Results for The Fisher’s Exact Test At Significance Level of 0.05**

ID	AVM vs (1+1) EA		AVM vs GA		AVM vs RS		(1+1 EA) vs GA		(1+1) EA vs RS		GA vs RS	
	p-Value	OR	p-Value	OR	p-Value	OR	p-Value	OR	p-Value	OR	p-Value	OR
29	1	3	3,84E-40	1029	7,78E-48	2187	2,82E-38	339	6,70E-46	721	0,12	2
30	0,12	9	8,62E-26	340	8,48E-43	1302	1,89E-20	36,	1,39E-36	138	0,0001	3
31	1	1	8,93E-17	164	5,29E-41	1108	8,93E-17	164	5,29E-41	1108	3,55E-09	6
32	0,12	9	1,08E-37	840	1,14E-55	7919	1,09E-31	89	4,47E-49	844	0,0001	9
33	0,01	16	3,75E-36	743	5,76E-49	2505	5,21E-27	46	5,69E-39	155	0,007	3
34	0,12	9	3,75E-36	743	1,14E-55	7919	3,22E-30	79	4,47E-49	844	2,50E-05	10
35	0,49	5	1,11E-34	662	1,02E-52	4310	2,27E-31	129	4,47E-49	84	0,0001	6,
36	1	1	0,059	11	0,01	16	0,05	11	0,01	16	0,76	1
37	1	1	1	3	1	1	1	3	1	1	1	0,3
38	1	3	0,0007	25	2,48E-08	64	0,004	8	3,64E-07	21	0,02	2
39	1	1	7,49E-05	33	1,43E-10	86	7,49E-05	33	1,43E-10	86	0,009	3
40	1	1	0,001	23	5,03E-15	140	0,001	23	5,03E-15	140	6,14E-07	6
41	1	1	1,59E-05	39	1,56E-12	108	1,59E-05	39	1,56E-12	108	0,003	3
42	1	1	1	1	1	1	1	1	1	1	1	1
43	1	1	1	1	1	1	1	1	1	1	1	1
44	1	1	1	1	1	1	1	1	1	1	1	1
45	1	1	1	1	1	1	1	1	1	1	1	1
46	1	1	1	1	1	1	1	1	1	1	1	1
47	1	1	1	1	1	1	1	1	1	1	1	1
48	1	1	1	1	1	1	1	1	1	1	1	1
49	1	1	1	1	1	1	1	1	1	1	1	1
50	1	1	1	1	1	1	1	1	1	1	1	1
51	1	1	1	1	1	1	1	1	1	1	1	1
52	1	1	1	1	1	1	1	1	1	1	1	1
53	1	1	1	1	1	1	1	1	1	1	1	1
54	1	1	1	1	1	1	1	1	1	1	1	1
55	1	1	1	1	1	1	1	1	1	1	1	1
56	1	1	1	1	1	1	1	1	1	1	1	1

To check the statistical significance of the results, we carried out a paired Mann-Whitney U-test (paired per constraint) at the significance level of 0.05 on the distributions of the success rates for the four algorithms. In all the four distribution comparisons, p-values were very close to 0, as shown in Table 16. This shows a strong statistical difference among the four algorithms when applied on all 57 constraints of our case study.

In addition, we performed a Fisher's exact test at the significance level of 0.05 between each pair of algorithms based on their success rates for the 57 constraints. The results for the Fisher's exact test are shown in Table 13 and Table 14. In addition to statistical significance, we also assessed the magnitude of the improvement by calculating the effect size in a standardized way. We used odds ratio [45] for this purpose, as the results of our experiments are dichotomous. Table 13 and Table 14 also show the odds ratio for various pairs of approaches for all 57 problems.

**Table 15 . Average Success Rates For Problems of Varying Complexity**

Complexity	AVM	(1+1) EA	GA	RS
1	1	1	1	1
2	1	1	1	1
5	1	0,995	0,86	0,60
6	1	0,98	0,43	0,20
7	1	0,95	0,21	0,04
8	1	0,98	0,21	0,02

**Table 16. Results for The paired Mann-Whitney U-test At Significance Level of 0.05**

Pair of approaches	p-Value
AVM vs (1+1) EA	2.653988e-05
AVM vs GA	2.507670e-08
AVM vs RS	2.485853e-08
(1+1) EA vs GA	2.506828e-08
(1+1) EA vs RS	2.480008e-08
GA vs RS	1.822280e-08

For AVM vs (1+1) EA, we did not observe significant differences for most of the problems, except for Problem 2 and Problem 33, where AVM significantly performed better than (1+1) EA. In addition, odds ratios between AVM and (1+1) EA for 23 problems are greater than 1, implying that AVM has more chances of success than (1+1) EA. For 35 problems out of 57, the odds ratio is 1 suggesting that there is no difference between these two algorithms. For AVM vs GA, for 38 problems AVM significantly performed better than GA as p-values are below 0.05 (our chosen significance level). The odds ratios for most of the problems, except for the problems with 1 or 2 clauses, are greater than one, thus suggesting that AVM has more chances of success than GA. Similar results were observed for (1+1) EA, where for 38 problems it significantly outperformed GA. For AVM vs RS, for almost all of the problems except the ones with one or two clauses, AVM

performed significantly better than RS. Similar results were observed for (1+1) EA vs RS and GA vs RS.

To check the complexity of the problems, we repeated the experiment on the negation of each of the 57 problems. All algorithms managed to find solutions for all these problems very quickly. Most of the time and for most of the problems, each algorithm managed to find solutions in a single iteration. This result confirmed that the actual problems we targeted with search were difficult to solve.

Based on the above results, we recommend using AVM and (1+1) EA for as many iterations as possible (RQ2). We can see from the results that, even when we set the number of iterations to 2000, AVM managed to achieve a 100% success rate with 26 iterations on average. On the other hand, (1+1) EA managed to achieve a 98% success rate with an average of 743 iterations. Note that in case studies with more complex problems, a larger number of iterations may be required to eventually solve the problems.

```

context Saturn inv synchronizationConstraint:
  self.systemUnit.NumberOfActiveCalls > 1 and
  self.systemUnit.NumberOfActiveCalls <= self.systemUnit.MaximumNumberOfActiveCalls and
  self.media.synchronizationMismatch.unit = TimeUnitKind::s and (self.media.synchronizationMismatch.value >= 0 and
  self.media.synchronizationMismatch.value <= self.media.synchronizationMismatchThreshold.value) and
  self.conference.PresentationMode = Mode::Off and
  self.conference.call→select(call | call.incomingPresentationChannel.Protocol <> VideoProtocol::Off)→size()=2 and
  self.conference.call→select(call | call.outgoingPresentationChannel.Protocol <> VideoProtocol::Off)→size()=2

```

**Figure 7. Condition for a change event which is fired when synchronization between audio and video is within threshold**

## 5.2 Comparison with UMLtoCSP

UMLtoCSP [15] is the most widely used and referenced OCL constraint solver in the literature. To assess the performance of UMLtoCSP to solve complex constraints such as the ones in our current industrial case study, we conducted an experiment. We repeated the experiment for 57 constraints from our industrial application, whose complexity is summarized in Table 10. An example of such constraint, modeling a change event on a transition of Saturn’s state machine, is shown in Figure 7. This change event is fired when Saturn is successful in recovering the synchronization between audio and video. Since UMLtoCSP does not support enumerations, we converted each enumeration into an Integer and limited its bound to the number of literals in the enumeration. We also used the MARTE profile to model different non-functional properties, and since UMLtoCSP does not support UML profiles, we explicitly modeled the used subset of MARTE as part of our

models. In addition, UMLtoCSP does not allow writing constraints on inherited attributes of a class, so we modified our models and modeled inherited attributes directly in the classes. We set the range of Integer attributes from 1 to 100. Since the UML2CSP tool did not support UML 2.x diagrams, we also needed to recreate our models in a UML 1.x modeling tool.

**Table 17. Average Time Took By Algorithms to Solve the Problems**

Algorithm	Average Time to Solve Constraints (Seconds)
AVM	2.96
(1+1) EA	99
GA	182
RS	423

**Table 18. Statistics of Complexity of Constraints**

Problem #	# of Clauses	OCL Data Type Used (Number of variable is 1)	Search-based Solver with (1+1)EA (Seconds)	Search-based Solver with AVM (Seconds)	UML2CSP (Seconds)
I43	1	Boolean	0.26	0.07	0.01
I44	1	Boolean	0.10	0.07	0.01
I45	1	Integer	0.07	0.03	0.01
I46	1	Integer	0.07	0.03	0.01
I47	1	Integer	0.07	0.03	0.01
I48	1	Integer	0.13	0.04	0.01
I49	2	Integer	1.41	0.26	0.01
I50	2	Integer	1.56	0.4	0.01
I51	1	Integer	0.12	0.04	0.01
I52	2	Integer	1.76	0.25	0.01
I53	2	Integer	1.72	0.26	0.01
I54	1	Integer	0.09	0.04	0.01
I55	2	Integer	1.25	0.24	0.01
I56	1	Integer	0.08	0.04	0.01
I57	2	Integer	1.48	0.23	0.01

We ran the experiment on the same machine as we used in the experiments reported in the previous section. Though we let UMLtoCSP attempt to solve each of the selected constraints for one hour each, it was not successful in finding any valid solution for the 42 problems comprising of 5-8 clauses. A plausible explanation is that UMLtoCSP is

hampered by a combinatorial explosion problem because of the complexity of the constraints in the model. However, such constraints must be expected in real-world industrial applications as our Cisco example is in no way particularly complex by industrial standards. In contrast, our constraint solver managed to solve each constraint within at most 2.96 seconds using AVM and 99 Seconds using (1+1) EA, as shown in Table 17. For the remaining 15 problems, which are simple constraints comprising of either one or two clauses, UMLtoCSP managed to find solutions. Each of these constraints has one variable of either Integer or Boolean type. The results of the comparison of UMLtoCSP with our tool for these simple clauses (problems 42-56) are shown in Table 18. We provide the time taken by UML2CSP to solve each problem in seconds, which is reported by the tool itself and 0.01 second (maximum precision) for all fifteen constraints. For these same 15 problems, we ran our tool 100 times for each of them. In Table 18, we report the average time taken by our tool to solve each problem over 100 runs. Since we used the same machine to run experiments for both tools, it is clear from the results that for all fifteen simple problems, UMLtoCSP took less time than our tool (which is on average less than one second and in the worst case less than two seconds). But considering that UMLtoCSP fails to solve the more complex problems and its issues regarding limited support of OCL constructs (as already discussed), we conclude it is not practical to apply UMLtoCSP in large systems having complex constraints.

## 6. Empirical Evaluation of Optimization Defined as Fitness Functions

In this section, we empirically evaluate the fine grained fitness functions that we defined in Section 4 for various OCL operations to see if they really improve performance of search algorithms as compared to using simple branch distance functions, yielding  $0$  if an expression is *true* and  $k$  otherwise.

### 6.1 Experiment Design

To empirically evaluate whether the functions defined in Section 4 really improve the branch distance, we carefully defined artificial problems to evaluate each heuristic since not all of the OCL constructs were used in the industrial case study. The model we used for

the experiment consists of a very simple class diagram with one class  $X$ .  $X$  has one attribute  $y$  of type *Integer*. The range of  $y$  was set to -100 to 100. We populated 10 objects of class  $X$ . The use of a single class with 10 objects was sufficient to create complex constraints. For each heuristic, we created an artificial problem, which was sufficiently complex to remain unsolved by random search. We checked this by running all the artificial problems (100 times per problem) using random search for 20,000 iterations per problem, and random search could not manage to solve most of the problems most of the times, except for problems  $A9$  and  $A10$ . Table 19 lists the artificial problems and the corresponding heuristics that we used in the experiments. We prefixed each problem with  $A$  to show that it is an artificial problem. For the evaluation, we used the best algorithms among search algorithms used in the industrial case study (Section 5.1 and in other works [38]): (1+1) EA and AVM. In this experiment, we address the following research question:

**RQ3:** Does optimized branch distance calculations improve the effectiveness of search over simple branch distance calculations?

To answer this research question, we compared branch distance calculations based on heuristics defined in Section 4 and without heuristics (i.e., branch distance calculations either return 0 when a constraint is solved or  $k$  otherwise). We will refer to them here as Optimized ( $Op$ ) and Non-Optimized ( $NOp$ ) branch distance calculations, respectively.

## 6.2 Experiment Execution

We ran experiments 100 times for (1+1) EA and AVM, with  $Op$  and  $NOp$ , and for each problem listed in Table 19. We let (1+1) EA and AVM run up to 2000 fitness evaluations on each problem and collected data on whether the algorithms found solutions for  $Op$  and  $NOp$ . 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. To compare the algorithms for  $Op$  and  $NOp$ , we calculated the success rate, which is defined as the number of times a solution was found out of the total number of runs (100 in this case).

**Table 19. Artificial Problems for Heuristics**

<b>Problem #</b>	<b>Heuristic</b>	<b>Example</b>
A1	forall()	$X.allInstances() \rightarrow forall(b b.y=47)$
A2	exists()	$X.allInstances() \rightarrow select(b b.y > 90) \rightarrow size() > 4$ and $X.allInstances() \rightarrow select(b b.y > 90) \rightarrow exists(b b.y=92)$
A3	isUnique()	$X.allInstances() \rightarrow select(b b.y > 90) \rightarrow size() > 4$ and $X.allInstances() \rightarrow select(b b.y > 90) \rightarrow isUnique(b b.y)$
A4	one()	$X.allInstances() \rightarrow select(b b.y > 90) \rightarrow size() > 4$ and $X.allInstances() \rightarrow select(b b.y > 90) \rightarrow one(b b.y=95)$
A5	select()size()	$X.allInstances() \rightarrow select(b b.y=0) \rightarrow size()>6$
A6	select()size()	$X.allInstances() \rightarrow select(b b.y=0) \rightarrow size()<=1$
A7	select()size()	$X.allInstances() \rightarrow select(b b.y > 90) \rightarrow size() > 4$ and $X.allInstances() \rightarrow select(b b.y > 90) \rightarrow select(b b.y=92) \rightarrow size() < 0$
A8	select()size()	$X.allInstances() \rightarrow select(b b.y=0) \rightarrow size() = 5$
A9	includes()	$X.allInstances() \rightarrow collect(b b.y) \rightarrow includes(17)$
A10	excludes()	$X.allInstances() \rightarrow collect(b b.y) \rightarrow excludes(0)$
A11	includesAll()	let $c = Set\{-1,87,19,88\}$ in $X.allInstances() \rightarrow collect(b b.y) \rightarrow includesAll(c)$
A12	excludesAll()	let $c = Set\{0,1,2,3\}$ in $X.allInstances() \rightarrow select(b b.y>0$ and $b.y<5) \rightarrow size()>=5$ and $X.allInstances() \rightarrow select(b b.y>0$ and $b.y<5) \rightarrow collect(b b.y) \rightarrow excludesAll(c)$
A13	select()forall()	$X.allInstances() \rightarrow select(b b.y<>47) \rightarrow forall(b b.y*b.y=-100)$

### 6.2.1 Results and Analysis

Table 20 shows the results of success rates for *Op* and *NOp* for each problem and both algorithms ((1+1) EA and AVM). To compare if the differences of success rates among *Op* and *NOp* are statistically significant, we performed the Fisher's exact test [47] at the significance level of 0.05. We chose this 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. We performed the test only for the problems having success rates greater than 0 and not equal to each other for both *Op* and *NOp* (i.e., for problems A2, A3, and A4 in the case of 1+1 (EA) and problem A9 for AVM)). For 1+1 (EA), the p-values for all the three problems (A2, A3, and A4) are 0.0001, thus indicating that the success rate of *Op* is significantly higher than *NOp*. For problems A1, A5, A6, A7, A8, A12, and A13, the results are even more extreme as *Op* shows a 100% success rate,

whereas *NOp* has 0% success rate. For the problems *A9* and *A10*, the success rates are 100% for both *Op* and *NOp* and hence conclusions cannot simply be drawn based on these rates. For these problems, we further compared the number of iterations taken by (1+1) EA for *Op* and *NOp* to solve the problems. We used Mann-Whitney U-test [[47], at a significance level of 0.05, to determine if significant differences exist between *Op* and *NOp*. We chose this test based on the guidelines for performing statistical tests for randomized algorithms [45]. Table 21 shows the results of the test. The p-values are bold-faced when the results are significant. In Table 21, we also show the mean differences for the number of iterations and execution time between *Op* and *NOp* to show the direction in which the results are significant. In addition, we report effect size measurements using Vargha and Delaney's  $\hat{A}_{12}$  statistics, which is a non-parametric effect size measure. We chose this effect size measure using again the guidelines reported in [45]. In our context, the value of  $\hat{A}_{12}$  tells the probability for *Op* to find a solution in more iterations than *NOp*. This means that the higher the value of  $\hat{A}_{12}$ , the higher the chances that *Op* will take more iterations to find a solution than *NOp*. If *Op* and *NOp* are equal then the value of  $\hat{A}_{12}$  is 0.5. With 1+1 (EA), for *A9* and *A10*, *Op* took significantly less iterations to solve the problems (Table 21) as both p-values are below 0.05. In addition, for *A9* and *A10*, values of  $\hat{A}_{12}$  are 0.19 and 0.46, respectively, thus showing that the only 19% and 46% of the time *Op* took more iterations to solve the problem than *NOp*.

For AVM, the results of success rates for *A10* were tied between *Op* and *NOp* (Table 20). Therefore, we further compared *Op* and *NOp* for these problems based on the number of iterations AVM took to solve these problems. As discussed before, we applied Mann-Whitney U-test [47] at significance level of 0.05 to determine if significant differences exist between *Op* and *NOp*. Table 22 shows mean differences, p-values, and  $\hat{A}_{12}$  values. We observed that for the problem *Op* took less iterations to solve the problems and significant differences were observed for *A10* as the p-value is 0.04, which is less than our significance level of 0.05.

Based on the above results, we can answer our research question presented earlier: does the optimized branch distance calculation improve the effectiveness of search? We can clearly see from the results that (1+1) EA and AVM with optimized branch distance calculations significantly improve the success rates. In worst cases, when there is no

differences in success rates between *Op* and *NOp*, (1+1) EA and AVM took significantly less iterations to solve the problems.

**Table 20. Results of Fisher Exact Test for Success Rate of Optimized and Non-Optimized at alpha=0.05**

Problem #	Success Rate (1+1)EA ( <i>NOp</i> ) in %	Success Rate for (1+1)EA ( <i>Op</i> ) in %	Fisher Exact Test for (1+1)EA (p-value)	Success Rate for AVM ( <i>NOp</i> ) in %	Success Rate for AVM ( <i>Op</i> ) in %	Fisher Exact Test for AVM (p-value)
A1	0	100	-	0	100	-
A2	2	100	<b>0,0001</b>	0	59	-
A3	1	95	<b>0,0001</b>	0	99	-
A4	3	100	<b>0,0001</b>	0	100	-
A5	0	100	-	0	100	-
A6	0	100	-	0	100	-
A7	0	100	-	0	100	-
A8	0	100	-	0	100	-
A9	100	100	-	16	100	<b>0,0001</b>
A10	100	100	-	100	100	-
A11	0	94	-	0	99	-
A12	0	100	-	0	34	-
A13	0	100	-	0	100	-

**Table 21. Results of t-test at alpha=0.05 ((1+1)EA)**

Problem #	Mean Difference (OP-NOP)	$\hat{A}12$	p-value
A9	-654,38	0,19	<b>0,0001</b>
A10	-1,01	0,46	<b>0,004</b>

**Table 22. Results of t-test at alpha=0.05 (AVM)**

Problem #	Mean Difference (OP-NOP)	$\hat{A}12$	p-value
A10	-0,23	0,52	<b>0,04</b>

## 7. Overall Discussion

In this section, we provide an overall discussion based on the results of the experiments on the industrial case study and the artificial problems. Based on the results from the industrial case study, we observe that AVM and (1+1) EA perform better as compared to

GA and RS since the algorithms achieve 100% and 98% success rates for all 57 constraints on average, respectively (Section 5.1). For the experiments based on artificial problems (Section 6.2.1), we observe that AVM and (1+1) EA with optimized branch distance calculations outperform non-optimized branch distance calculations. However, we notice that for certain artificial problems, the performance of (1+1) EA is significantly better than AVM. For instance, in Table 20 for *A2*, (1+1) EA manages to find solutions for all 100 runs, whereas AVM could only manage to find solutions for 59 runs. We further perform a Fisher's exact test to determine if the differences are statistically significant at the significance level of 0.05 between these two algorithms. We obtain a p-value of 0.001 suggesting that (1+1) EA is significantly better than AVM for *A2*. Since AVM is a local search algorithm and *A2* is a complex problem, AVM can be expected to be less efficient than (1+1) EA. Similar results are obtained for *A12*. Conversely, for other problems, i.e., for *A3* and *A11*, AVM seems more successful than (1+1) EA. For *A3*, AVM manages to find solutions 99 times, whereas (1+1) EA manages to find solutions 95 times (Table 20). In this case, we obtain a p-value of 0.21 when we applied the Fisher's exact test, hence suggesting that the differences are not statistically significant between the two algorithms. Similarly, for *A11*, AVM found solutions 99 times, whereas (1+1) EA found solutions for 94 times (Table 20). In this case, we obtain again a p-value of 0.11, which is lower than our chosen significance level (0.05); hence suggesting that the differences are not significant between these two algorithms.

Based on the results of our empirical analysis, we provide the following recommendations about using AVM and (1+1) EA: If the constraints need to be solved quickly, we recommend using AVM, since it is quicker in finding solutions as we discussed in Section 6, even though its performance was worse than (1+1) EA for two artificial problems. 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 98% on average for the industrial case study, whereas for the artificial problems, it either fares better or equal to AVM.

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 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.

## 8. Tool Support

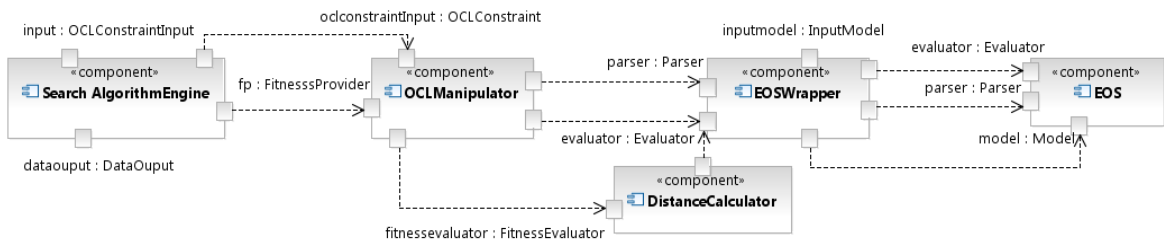
To efficiently solve OCL constraints, we developed a search-based OCL constraint solver, since current OCL solvers were not able to handle the complexity of the constraints in our models for the industrial case study within reasonable time (Section 5.2). Figure 8 shows the architecture diagram for our Search-based Constraint solver. We developed a tool in Java that interacts with an existing library, an OCL evaluator, called the EyeOCL Software (EOS) [24]. EOS is a Java component that provides APIs to parse and evaluate an OCL expression based on an object model. Our tool only requires interacting with EOS for the evaluation of constraints. We selected to use EOS as it is one the most efficient evaluators currently available. Any other OCL evaluator can also be easily integrated with our tool. Our tool implements the calculation of branch distance (*DistanceCalculator*) for various expressions in OCL as discussed in Section 4, which aims at calculating how far are the test data values from satisfying constraints. The search algorithms employed are implemented in Java as well and include Genetic Algorithms, (1+1) Evolutionary Algorithm, and Alternating Variable Method (AVM).

## 9. Threats to Validity

To reduce construct validity threats, we chose as an effectiveness measure the search success rate, which is comparable across all four search algorithms (AVM, (1+1) EA, GA and RS). Furthermore, we used the same stopping criterion for all algorithms, i.e., number of fitness evaluations. This criterion is a comparable measure of efficiency across all the algorithms because each iteration requires updating the object diagram in EyeOCL and evaluating a query on it.

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. We chose Fisher exact test because it is appropriate for dichotomous data where proportions must be compared [45], 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 search techniques.

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 [46].



**Figure 8. Architecture diagram for search-based constraint solver**

We ran our experiments on an industrial case study to generate test data for 57 different OCL constraints, ranging from simpler constraints having just one clause to complex constraints having eight clauses. Although the empirical analysis is based on a real industrial system our results might not generalize to other case studies. However, such threat to external validity is 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. In addition, empirically evaluating all proposed branch distance calculations on artificial problems was necessary since it was not possible to evaluate them for all features of OCL in the industrial case study due to its inherent properties.

In the empirical comparisons with UMLtoCSP, we might also have wrongly configured the tool. To reduce the probability of such an event, we contacted the authors of UMLtoCSP who were very helpful in ensuring its proper use. From our analysis of UMLtoCSP, we cannot generalize our results to traditional constraint solvers in general when applied to solve OCL constraints. However, empirical comparisons with other constraints solvers were not possible because, to the best of our knowledge, UMLtoCSP is not only the most referenced OCL solver but also the only one that is publically available. However, because the problems encountered with UMLtoCSP are due to the translation to a lower-level constraint language, we expect similar issues with the other constraint solvers.

## 10. Conclusion

In this paper, we presented a search-based constraint solver for constraints written in the Object Constraint Language (OCL). The goal is to achieve a practical, scalable solution to support test data generation for Model-based Testing (MBT). Existing OCL constraint solvers have one or more of the following problems that make them difficult to use in industrial applications: (1) they support only a subset of OCL; (2) they translate OCL into formalisms such as first order logic, temporal logic, or Alloy, and thus result into combinatorial explosion problems. These problems limit their practical adoption in industrial settings.

To overcome the abovementioned problems, we defined a set of heuristics based on OCL constraints to guide search-based algorithms (Genetic Algorithms (GAs), (1+1) Evolutionary Algorithm (EA), Alternating Variable Method (AVM)) and implemented them in our search-based OCL constraint solver. More specifically, we defined branch distance functions for various types of expressions in OCL to guide search algorithms. We demonstrated the effectiveness and efficiency of our search-based constraint solver to generate test data in the context of the model-based, robustness testing of an industrial case study of a video conferencing system. Even for the most difficult constraints, with research prototypes and no parallel computations, we obtain test data within 2.96 seconds on average.

As a comparison, we ran 57 constraints from the industrial case study on one well-known, downloadable OCL solver (UMLtoCSP) and the results showed that, even after running it for one hour, no solutions could be found for most of the constraints. Similar to all existing OCL solvers, because it could not handle all OCL constructs and UML features, we had to transform our constraints to satisfy UMLtoCSP requirements.

We also conducted an empirical evaluation in which we compared four search algorithms using two statistical tests: Fisher's exact test between each pair of algorithms to test their differences in success rates for each constraints and a paired Mann-Whitney U-test on the distributions of the success rates (paired per constraint). Results showed that AVM was significantly better than the other three search algorithms, followed by (1+1) EA, GA and RS respectively. We also empirically evaluated each proposed branch distance calculation using small yet complex artificial problems. The results showed that the proposed branch distance calculations significantly improve the performance of solving OCL constraints for the purpose of test data generation when compared to simple branch distance calculations. Based on the results of our empirical analyses, we recommend using AVM if the constraints need to be solved quickly since it is quicker in finding solutions, even though its performance was worse than (1+1) EA for two complex artificial problems with difficult search landscapes. In other cases, if we are flexible with time budget (e.g., the constraints need to be solved only once), we rather recommend using (1+1) EA for as many iterations as possible since (1+1) EA has 98% success rate on average for the industrial case study, whereas for the artificial problems, it either fares better or equal to AVM.

Though focused on OCL in this paper, the general search-based solution and heuristics we propose here to make the search more efficient could be adapted to other high level constraint languages based on first-order logic and set theory. In the future, we are also planning to extend our solver to automatically instantiate models by solving constraints defined on their metamodels for the purpose of model-transformation testing, which is an increasingly important challenge.

## ACKNOWLEDGEMENT

The work described in this paper was supported by the Norwegian Research Council. This paper was produced as part of the ITEA-2 project called VERDE. We thank Marius Christian Liaaen (Cisco Systems, Inc Norway) for providing us the case study. We are also grateful to Jordi Cabot, an author of UML2CSP, for helping us to run UML2CSP on our industrial case study.

## REFERENCES

- [1] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*: Morgan-Kaufmann, 2007.
- [2] OCL, "Object Constraint Language Specification, Version 2.2," Object Management Group (OMG), 2011.
- [3] MOF, "Meta Object Facility (MOF)," 2006.
- [4] L. v. Aertryck and T. Jensen, "UML-Casting: Test synthesis from UML models using constraint resolution," in *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2003)*, 2003.
- [5] M. Benattou, J. Bruel, and N. Hameurlain, "Generating test data from OCL specification," Citeseer, 2002.
- [6] L. Bao-Lin, L. Zhi-shu, L. Qing, and C. Y. Hong, "Test case automate generation from uml sequence diagram and ocl expression," in *International Conference on Computational Intelligence and Security*, 2007, pp. 1048-1052.
- [7] M. Harman, S. A. Mansouri, and Y. Zhang, "Search based software engineering: A comprehensive analysis and review of trends techniques and applications," King's College, Technical Report TR-09-032009.
- [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] M. Alshraideh and L. Bottaci, "Search-based software test data generation for string data using program-specific search operators: Research Articles," *Softw. Test. Verif. Reliab.*, vol. 16, pp. 175-203, 2006.
- [10] A. Andrea, "Longer is Better: On the Role of Test Sequence Length in Software Testing," *International Conference on Software Testing, Verification, and Validation*, 2010, pp. 469-478.
- [11] B. Bordbar and K. Anastasakis, "UML2Alloy: A tool for lightweight modelling of Discrete Event Systems," in *IADIS International Conference in Applied Computing*, 2005.
- [12] D. Distefano, J.-P. Katoen, and A. Rensink, "Towards model checking OCL," in *ECOOP-Workshop on Defining Precise Semantics for UML*, 2000.
- [13] M. Clavel and M. A. G. d. Dios, "Checking unsatisfiability for OCL constraints," in *In the proceedings of the 9th OCL 2009 Workshop at the UML/ModelS Conferences*, 2009.

- [14] B. K. Aichernig and P. A. P. Salas, "Test Case Generation by OCL Mutation and Constraint Solving," in *Proceedings of the Fifth International Conference on Quality Software*: IEEE Computer Society, 2005.
- [15] J. Cabot, R. Claris, and D. Riera, "Verification of UML/OCL Class Diagrams using Constraint Programming," in *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*: IEEE Computer Society, 2008.
- [16] D. Berardi, D. Calvanese, and G. D. Giacomo, "Reasoning on UML class diagrams," *Artif. Intell.*, vol. 168, pp. 70-118, 2005.
- [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] M. Kyas, H. Fecher, F. S. d. Boer, J. Jacob, J. Hooman, M. v. d. Zwaag, T. Arons, and H. Kugler, "Formalizing UML Models and OCL Constraints in PVS," *Electron. Notes Theor. Comput. Sci.*, vol. 115, pp. 39-47, 2005.
- [19] M. P. Krieger, A. Knapp, and B. Wolff, "Automatic and Efficient Simulation of Operation Contracts," in *9th International Conference on Generative Programming and Component Engineering*, 2010.
- [20] S. Weißleder and B.-H. Schlingloff, "Deriving Input Partitions from UML Models for Automatic Test Generation," in *Models in Software Engineering*: Springer-Verlag, 2008, pp. 151-163.
- [21] M. Gogolla, F. Bttner, and M. Richters, "USE: A UML-based specification environment for validating UML and OCL," *Sci. Comput. Program.*, vol. 69, pp. 27-34, 2007.
- [22] IBM, "IBM OCL Parser," IBM, 2011.
- [23] D. Chiorean, M. Bortes, D. Corutiu, C. Botiza, and A. Cârçu, "OCLE," V2.0 ed, 2010.
- [24] M. Egea, "EyeOCL Software," 2010.
- [25] CertifyIt, "CertifyIt," Smarttesting, 2011.
- [26] P. McMinn, "Search-based software test data generation: a survey: Research Articles," *Softw. Test. Verif. Reliab.*, vol. 14, pp. 105-156, 2004.
- [27] D. Jackson, I. Schechter, and H. Shlyachter, "Alcoa: the alloy constraint analyzer," in *Proceedings of the 22nd international conference on Software engineering* Limerick, Ireland: ACM, 2000.
- [28] M. Krieger and A. Knapp, "Executing Underspecified OCL Operation Contracts with a SAT Solver," in *8th International Workshop on OCL Concepts and Tools*. vol. 15: ECEASST, 2008.
- [29] A. D. Brucker, M. P. Krieger, D. Longuet, and B. Wolff, "A Specification-Based Test Case Generation Method for UML/OCL," in *Workshop on OCL and Textual Modelling, MoDELS: Lecture Notes in Computer Science*, Springer, 2010.
- [30] Gecode, "Gecode," 2011.
- [31] COMET, "COMET," 2011.
- [32] M. Iqbal, A. Arcuri, and L. Briand, "Environment Modeling with UML/MARTE to Support Black-Box System Testing for Real-Time Embedded Systems: Methodology and Industrial Case Studies," in *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2010.

- [33] S. Ali, L. C. Briand, and H. Hemmati, "Modeling Robustness Behavior Using Aspect-Oriented Modeling to Support Robustness Testing of Industrial Systems," Simula Research Laboratory, Technical Report (2010-03)2010.
- [34] C. Doungsa-ard, K. Dahal, A. Hossain, and T. Suwannasart, "GA-based Automatic Test Data Generation for UML State Diagrams with Parallel Paths," *Advanced Design and Manufacture to Gain a Competitive Edge*, pp. 147-156, 2008.
- [35] R. Lefticaru and F. Ipate, "Functional Search-based Testing from State Machines," in *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*: IEEE Computer Society, 2008.
- [36] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 263-272, 2005.
- [37] K. Lakhotia, P. McMinn, and M. Harman, "An empirical investigation into branch coverage for C programs using CUTE and AUSTIN," *Journal of Systems and Software*, vol. 83, pp. 2379-2391.
- [38] A. Arcuri, "It really does matter how you normalize the branch distance in search-based software testing," *Software Testing, Verification and Reliability*, 2011.
- [39] R. V. Binder, *Testing object-oriented systems: models, patterns, and tools*: Addison-Wesley Longman Publishing Co., Inc., 1999.
- [40] A. Arcuri, M. Z. Iqbal, and L. Briand, "Black-box System Testing of Real-Time Embedded Systems Using Random and Search-based Testing," in *IFIP International Conference on Testing Software and Systems (ICTSS)*, 2010.
- [41] H. Li and Gordon, "Bytecode Testability Transformation " in *Symposium on Search based Software Engineering Co-located with ESEC/FSE*: ACM SIGSOFT, 2011.
- [42] S. Ali, H. Hemmati, N. E. Holt, E. Arisholm, and L. C. Briand, "Model Transformations as a Strategy to Automate Model-Based Testing - A Tool and Industrial Case Studies," Simula Research Laboratory, Technical Report (2010-01)2010.
- [43] MARTE, "Modeling and Analysis of Real-time and Embedded systems (MARTE)," 2010.
- [44] S. Ali, L. Briand, A. Arcuri, and S. Walawege, "An Industrial Application of Robustness Testing using Aspect-Oriented Modeling, UML/MARTE, and Search Algorithms," in *ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (Models 2011)*, 2011.
- [45] A. Arcuri and L. Briand., "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering," in *International Conference on Software Engineering (ICSE)*, 2011.
- [46] A. Arcuri and G. Fraser, "On Parameter Tuning in Search Based Software Engineering," in *International Symposium on Search Based Software Engineering (SSBSE)*: Springer's Lecture Notes in Computer Science (LNCS) 2011.
- [47] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*: Chapman and Hall/CRC, 2007.