Generating Test Data from OCL Constraints with Search Techniques

Shaukat Ali[1], Muhammad Zohaib Iqbal[1], Andrea Arcuri[1], Lionel Briand[2]
[1]Certus Software V&V Center
Simula Research Laboratory, Norway
[2]University of Luxembourg, SnT Centre, Luxembourg
{shaukat, zohaib, arcuri }@simula.no, lionel.briand@uni.lu

Abstract— 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 the Object Constraint Language (OCL). Given that test data generation is one of the major challenges to automate MBT, we focus on test data generation from OCL constraints in this paper. 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. We evaluate these heuristics for three search algorithms: Genetic Algorithms, (1+1) Evolutionary Algorithm, and Alternating Variable Method. We empirically evaluate our heuristics using complex artificial problems, followed by empirical analyses of the feasibility of our approach on one industrial system in the context of test data generation targeting robustness. Our approach is also compared with the most widely referenced OCL solver (UMLtoCSP) in the literature and shows to be significantly more efficient.

Index Terms— OCL, Search-based testing, Test data generation, Empirical evaluation, Search-based software engineering

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 modeling 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. 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 predefined extension mechanisms. OCL has been used in many industrial projects for various purposes such as for con-

figuration management [4] and test case generation [5, 6]. OCL is also being used as the language for writing constraints on models in many commercial MBT tools such as CertifyIt [7] and QTronic [8].

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 in the paper, either do not handle important features of OCL such as collections or their operations [9, 10], are not scalable, or lack proper tool support [11]. 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 generate test data from OCL constraints [2]. We implemented our test data generator in Java and evaluated its application to robustness testing of industrial Video Conferencing Systems developed by Cisco Systems, Inc., Norway.

This paper is an extended version of the conference paper presented in [12]. The differences of this paper from the conference version include: 1) The handling of three-valued *Boolean* logic supported by OCL 2.2 in contrast to the two-valued logic reported earlier; 2) New heuristics such as heuristics for operations on collections, special operations (e.g., *oclInState*), and user-defined operations; 3) An augmented empirical evaluation based on an industrial case study, which is extended with new constraints and an additional search algorithm (Alternating Variable Method); 3) The empirical evaluation of the individual heuristics on several artificial problems; 4) A comprehensive comparison with the existing work in the literature on test data generation from OCL constraints; 5) A detailed empirical evaluation comparing our work with the most widely used and referenced OCL constraint solver in the literature (UMLtoCSP [13]). Note that even though UMLtoCSP was developed for constraint solving, it can be applied for test data generation.

The rest of the paper is organized as follows: Section 2 discusses the background of our work and Section 3 discusses related work. In Section 4, we first present the representation of our test data generation problem in the context of OCL followed by the definition of distance functions for various OCL constructs. Section 5 describes our tool support and presents a running example demonstrating test data generation in the context of MBT. Section 6 discusses our industrial case study,

a Video Conferencing System, and reports the results of the application of our approach, whereas Section 7 provides an empirical evaluation of heuristics on a set of artificial problems, and Section 8 provides an overall discussion of both empirical evaluations presented in Section 6 and Section 7. Section 9 addresses the threats to validity of our empirical study, and finally Section 10 concludes the paper.

2. BACKGROUND

In this section, we will provide background information on topics that will help understanding the remainder of this paper.

2.1 Search-based Software Testing

The main aim of software testing is to detect as many faults as possible, especially the most critical ones, in the system under test (SUT). To gain sufficient confidence that most faults are detected, testing should ideally be exhaustive. Since in practice this is not possible, testers resort to test models and coverage/ adequacy criteria to define systematic and effective test strategies that are fault revealing. A test case typically consists of test data and the expected output [14]. The test data can take various forms such as values for input parameters of a function and values of input parameters for a sequence of method calls.

In order to perform test case generation, systematically and efficiently, automated test case generation strategies should be employed. Bertolino [15] addresses the need for 100% automatic testing as a means to improve the quality of complex software systems that are becoming the norm of modern society. A comprehensive testing strategy must address a number of activities that should ideally be automated: the generation of test requirements, test case generation, test oracle generation, test case selection, or test case prioritization. A promising strategy for tackling this challenge comes from the field of search-based software engineering [16] [17, 18].

Search-based software engineering attempts to solve a variety of software engineering problems by reformulating them as search problems [17]. A major research area in this domain is the application of search algorithms to test case generation [18]. Search algorithms are a set of generic algorithms that are used to find optimal or near optimal solutions to problems that have large complex search spaces [17]. There is a clear match between search algorithms and software test case generation. The process of generating test cases can be seen as a search or an optimization process: there are possibly hundreds of thousands of test cases that could be generated for a particular SUT and from this pool we need to select, systematically and at a reasonable cost, those that comply to certain coverage criteria and are expected to be fault revealing, at least for certain types of faults. Hence, we can reformulate the generation of test cases as a search that aims at finding the required or optimal set of test cases from the space of all possible test cases. When software testing problems are reformulated into search problems, the resulting search spaces are usually very complex, especially for realistic or real-world SUTs. For example, in the case of white-box testing, this is due to the non-linear nature of software resulting from control structures

such as if-statements and loops [19]. In such cases, simple search strategies may not be sufficient and global search algorithms may, as a result, become a necessity as they implement global search and are less likely to be trapped into local optima [20]. The use of search algorithms for test case generation is referred to as search-based software testing (SBST) [21]. Mantere and Alander [22] discuss the use of search algorithms for software testing in general and McMinn [23] provides a survey of some of the search algorithms that have been used for test data generation.

SBST has been shown to produce results that are comparable to human competence [24], and the field has reached a state mature enough to obtain real-world results of interest for practitioners. For example, there exist SBST tools (e.g., EvoSuite [25]) that have been succesfully used to automatically generate test suites for open source projects *randomly* selected from open source repositories (e.g., SourceForge), and not just small, artificial, hand-picked case studies [26].

2.2 Description of a Selection of Search Algorithms

The most common search algorithms that have been employed for search-based software testing are evolutionary algorithms, simulated annealing, hill climbing (HC), ant colony optimization, and particle swarm optimization [27]. Among these algorithms, HC is a simpler, local search algorithm. The SBST techniques using more complex, global search algorithms are often compared with test case generation based on HC and random search to determine whether their complexity is warranted to address a specific test case generation problem. The use of the more complex search algorithm may only be justified if it performs significantly better than, for instance, random search. 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 toward fitter solutions. Below, we provide a brief description of the search algorithms that we used in this paper.

2.2.1 Genetic Algorithms

Genetic Algorithms (GAs) are the most well-known [28] and 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. GAs are the most commonly used algorithms and hence we do not provide further details; however an interested reader may consult the following reference for more details [29].

2.2.2 (1+1) Evolutionary Algorithm

(1+1) Evolutionary Algorithm (EA) [30] is simpler than GAs. In (1+1) EA, population size is one, i.e., we have only one individual in the population and the individual is represented as a bit string. As opposed to GAs, we do not use the crossover operator but only rely on a bitwise mutation operator for exploring the search space. To produce an offspring, this operator independently flips each bit in the bit string with a probability (p) based on the length of the string. If the fitness

of the child is better than that of the parent (bit string of the child before mutation), the child is retained for the next generation.

2.2.3 Alternating Variable Method

Alternating Variable Method (AVM) is a local search algorithm first introduced by Korel [31]. The algorithm works in the following way: Suppose we have a set of variables $\{v_1, v_2, \ldots, v_n\}$, we then try to maximize fitness of v_1, while keeping the rest of the variables constant, which are generated randomly. The search is stopped if a solution is found; otherwise, if the solution is not found, but we found a minimum with respect to v_1, we switch to the second variable. Now, we fix v_1 at the found minimum value and try to minimize v_2, while keeping the rest of the variables constant. The search continues in this way, until we find a solution or we have explored all the variables.

3. Related Work

The Object Constraint Language is based on first-order logic (FOL), but has some distinct features that differentiate it from FOL, such as the internal object representation and support for three-valued logic (i.e., *undefined*, *true*, and *false* for constraint evaluation) [2, 9]. There are a number of approaches that have been proposed in the literature that deal with evaluation of OCL constraints and solving OCL constraints for various purposes such as for test data generation, model checking, and theorem proving. In this section, we analyze how these approaches relate to our approach, even though their objectives may differ. In Section 3.1, we relate our test data generator with OCL evaluators. In Section 3.2, we discuss related OCL constraint solving approaches that may be used for test data generation. In Section 3.3, we consider the approaches that specifically support test data generation from OCL constraints, whereas Section 3.4 discusses the use of 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 OCLE 2.0 [32], Eclipse OCL [33], Dresden OCL Toolkit [34], USE [35], EyeOCL [36], and the OCL evaluation in CertifyIt by Smartesting [7]. Our work requires an OCL evaluator for 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 [29]) of an OCL expression to guide a search algorithm towards a solution. Any of the abovementioned OCL evaluators can be extended with our proposed heuristics (Section 4) for test data generation since these heuristics are defined on standard OCL constructs.

3.2 OCL-based Constraint Solvers

Table I and Table II present various approaches in the literature that may be used for test data generation from OCL

constraints. In Table I, in the second column we present if an approach translates OCL into another formalism for solving followed by the name of the formalism (*Intermediate Representation*) in the third coulumn (e.g., Alloy [37]). The fourth column presents the OCL subset supported by an approach followed by the support of three-valued logic in the last column. In Table II, we present the following information about an approach; 1) tool support (second column); 2) type of approach for solving such as theorem proving (third column); 3) support for test data generation.

Table I: Summary of OCL constraint solving approaches

Technique	Translation to Formalism	Intermediate Representation	OCL Parts Missing or Additional Requirements	Three-Valued Logic
Alloy Analyzer [37]	Yes	Alloy	Real, String, Enumerations, Limited operations on collections, attributes	No
Aertryck & Jensen [9]	Yes	SAT	Collections, Real, String, Enumerations	No
Diestefano *et al*. [38]	Yes	BOTL	String, real, enumerations	No
Clavel *et al*. [39]	Yes	FOL	String, Real, collections other than Set, Enumeration	No
Bao-Lin *et al*. [11]	No	DNF	Not discussed in the paper	No
Benattou *et al*. [10]	No	DNF	Class Inheritance, Generalization, Association	No
Aichernig [40]	Yes	CSP	Handles a small subset, collections iterators, Bag, Sequence	No
UMLtoCSP [13]	Yes	CSP	Enumerations	No
Queralt *et al* [41]	Yes	FOL	Operations that cannot be converted to select() or size() operations, e.g., collect.	No
Winkelmann [42]	Yes	Graph constraints	Collection operations except size(), isEmpty(). Enumerations	No
Kyas *et al* [43]	Yes	PVS	Not discussed in the paper	Yes
Kreiger [44]	Yes	SAT in CNF	Adds a non-standard extension, String, Real, Enumerations	Yes
Weißleder [45]	No	Test Tree	Collections, Enumerations	No
Gogolla [46]	Yes	Formal Logic	Desired properties of snapshot to be specified in a language ASSL	Yes
HOL-OCL [47]	Yes	Higher-Order Logic (HOL)	OclMessage, Enumeration, Tuple, OrderedSet, oclInState, iterate()	Yes
KeY Project [48]	Yes	Typed dynamic logic	Supports OCL 1.4	No

Table II: Summary of OCL constraint solving approaches

Technique	Tool Support	Approach Type	Test Data Generation
Alloy Analyzer [37]	Yes	SAT Solver	No
Aertryck & Jensen [9]	Yes	SAT Solver	Yes
Diestefano *et al*. [38]	Yes	Model Checking	No
Clavel *et al*. [39]	Yes	SMT Solver	No
Bao-Lin *et al*. [11]	No	Partition Analysis	Yes
Benattou *et al*. [10]	No	Partition Analysis	Yes
Aichernig [40]	Yes	CSP Solving	No
UMLtoCSP [13]	Yes	CSP Solving, Instance Generation	No
Queralt *et al* [41]	No	Reasoning	No
Winkelmann [42]	No	Instance Generation	No
Kyas *et al* [43]	Yes	Theorem Proving, Interactive	No
Kreiger [44]	Yes	SAT Solver	No
Weißleder [45]	Yes	Partition Testing	Yes
Gogolla [46]	Yes	Interactive	No
HOL-OCL, HOL-TestGen [47]	Yes	Interactive theorem proving	Yes
KeY Project [48]	Yes	Symbolic program execution	No

Many approaches are proposed in the literature (Table I and Table II) to solve OCL constraints for various purposes

such as checking unsatifiability [39], model-checking [38], and reasoning about models [41]. These approaches usually translate constraints and associated models into another formalism (e.g., Alloy [37], temporal logic BOTL [38], FOL [39], Prototype Verification System (PVS) [43], graph constraints [42], Higher-Order Logic (HOL) [47]), which can then be analyzed by a constraint analyzer (e.g., Alloy constraint analyzer [49], model checker [38], Satisfiability Modulo Theories (SMT) Solver [39], theorem prover [39], [43], [47]). Satisfiability Problem (SAT) solvers have also been used for evaluating OCL specifications ,e.g., OCL operation contracts (e.g., [50], [44]). The approaches listed in Table II that do not support test data generation (*Test Data Generation* column) may normally be extended for test data generation. In Section 6.2, we compare our approach with one such approach (UMLtoCSP), which is one of the most referenced works in the literature and for which a tool can be downloaded. The results showed that our approach is significantly better than UMLtoCSP when solving OCL constraints for generating test data. We do not compare our approach with the rest of OCL constraint solving approaches since their main aim is not test data generation.

3.3 OCL Test Data Generation Approaches

From all of the approaches listed in Table I and Table II, only five approaches ([9], [10], [11], [45], and [47]) are specifically developed for test data generation from OCL constraints and thus are directly related to our approach. Below, we compare our approach with them based on the different criteria listed in Table I and Table II.

3.3.1 Translation to Formalism

All the five approaches considered solve OCL constraints to generate data by translating OCL into another formalism such as HOL [47] and SAT formulae [9]. Such translation is an additional overhead and our approach aims at avoiding such overhead by directly solving OCL constraints to generate test data. Moreover, in many cases it is not always possible to translate all features of OCL and associated models into a given target formalism. In addition, such translation may result in combinatorial explosion. For example, 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 more details in [13]). 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 therefore increase. For industrial scale systems, as in our case study, this is a major limitation since the models and constraints are generally quite complex.

3.3.2 OCL Subset Supported

As it can be seen in Table I, the five approaches considered do not handle important features of OCL (e.g., Collections [9] [45], Associations [10], Enumerations [9] [47], three-valued logic [9] [10] [11] [45]). Most of the OCL features that they

do not support are either used in our industrial case study or are not solved by the existing approaches efficiently (we will discuss them in Section 5.2). In comparison, our approach supports a much more comprehensive subset of OCL.

3.3.3 Approach Type

The five approaches we consider here use various types of techniques for test data generation after translating OCL constraints into other formalisms, including SAT solving [9], partition testing [10] [11] [45], and theorem proving [47]. In contrast, we propose a new approach, which does not require translation and uses search-algorithms based on novel heuristics to generate test data. As further discussed in Section 3.3.1, such translation is an extra overhead and is a major limitation for large-scale industrial systems.

3.3.4 Tool Support

Automation is one of the key features for test data generation. From the five considered approaches, no tool support is mentioned in [10] and [11]. Our approach is automated as detailed in Section 5.

3.3.5 Three-Valued Logic

The OCL supports three-valued logic [2], i.e., each constraint in addition to being *true* or *false* may evaluate to *undefined*. The *undefined* value is commonly used to indicate an error in the evaluation of an expression such as divide by zero. Among the five considered approaches, only one of them ([47]) handles the OCL three-valued logic, like we do.

3.3.6 Summary

To summarize, none of the five test data generation approaches entirely meet the requirements of large scale real-world industrial applications, such as the video conferencing case study discussed in this paper (Section 6). Automation is a fundamental requirement for such adoption, which only three of these approaches support. None of these approaches support all OCL constructs, may of them (e.g., operations on collections) being important for successful industrial adoption. All the five approaches translate OCL constraints into other formalisms to generate test data. Such translation is either not entirely possible or is not scalable for industrial systems as we discussed in Section 3.3.1.

3.4 Search-based Heuristics for Model Based Testing

The application of search-based heuristics for MBT has received significant attention recently (e.g., [51], [52]). 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 [29]. Our goal is to tailor these heuristics for test data generation from OCL constraints.

There are several advances in the field of SBST (especially for test data generation at the unit level) that can be adopted to solve OCL constraints and integrated in our tool. For example, Alshraideh and Bottaci provided fitness functions to handle constraints involving string comparisons [53]. McMinn *et al* used web queries to further help the solving of constraints involving strings [54]. Fraser and Arcuri evaluated different seeding strategies to boost the search by starting from fitter individuals [55]. Issues with variable length representations can be addressed with bloat control techniques [56]. When generating test data to obtain full coverage on a state machine, instead of trying to solve each constraint individually, the *whole test suite* approach [57] could be used instead to further improve performance. The probability of applying the different search operators can be updated a trun time based on fitness feedback (e.g., as done by Poulding *et al* in [58]). Furthermore, search algorithms can be easily run in parallel, even on inexpensive Graphical Processing Units (GPUs) [59] .

4. Heuristics for Test Data Generation

In this section, we provide novel heuristics that are used by search-algorithms to generate test data from OCL constraints. In Section 4.2.1, we provide representation of the problem followed by the definitions of fitness functions in Section 4.2.2.

4.1 Representation of Problem

In the context of test data generation from OCL constraints, a problem is equivalent to an OCL constraint. An OCL constraint *C* (problem) is composed of a set of *Boolean* clauses $\{B_1, B_2, .. B_n\}$ joined using *Boolean* operations, i.e., {*and*, *or*, *implies, xor, not*}. Each *Boolean* clause B_i is defined over a set of variables $\{B_{i1}, B_{i2}, .. B_{im}\}$. The problem (*C*) to be solved by a search-algorithm can be represented as a set of variables: $C = \bigcup_{x=1}^{n} \bigcup_{y=1}^{m_x} \{B_{xy}\}$, where *n* is the number of clauses in a constraint and m_x is the number of variables involved in the x_{th} clause and m_x may be different for each x_{th} clause.

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*. Therefore, a clause in an OCL constraint may use primitive data types or collection-related types. 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 *University* and *Student* are shown in Figure 2.

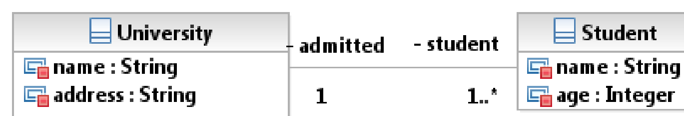

Figure 1. Example class diagram

```
context Student inv ageConstraint:
        self.age>15

context University inv numberOfStudents:
        self.student->size() > 0
```

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).

4.2 Definition of the Fitness Function for OCL

To guide the search for test data satisfying OCL constraints, it is necessary to define a set of heuristics. A heuristic indicates '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 the work targeting code coverage (e.g., for branch coverage in C code [29]). In particular, we use the so-called branch distance (a function *d()*), as defined in [29]. 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 [60]. 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 the firing of a guarded transition in a state machine, we can simply negate the guard and find data for the negated guard.

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

4.2.1 Primitive types

Primitive types supported by OCL includes *Integer*, *Real*, *String*, and *Boolean*. The *Integer* type represents a set of integer values, the *Real* type holds a set of real numbers, *Boolean* either *true* or *false*, and *String* contains strings over a set of alphabet. In the OCL, the domain of each of these data types also contains a special value called *undefined* (⊥). According to the

OCL specifications [2] this value is used for the following purposes; 1) An *undefined* value can be assigned to an attribute for which the value is currently unknown and may be assigned later, when it is known; 2) An *undefined* value may be assigned to an attribute for a particular instance for which this attribute cannot have any value; 3) An *undefined* value may represent an error in the evaluation of an expression such as divide by zero. In our context of test data generation, only the third purpose is relevant, where an *undefined* evaluation of expression indicates an error. To resolve such situations without stopping the search, we define specialized heuristics, which are discussed next.

When a *Boolean* variable b is true then the branch distance is 0, i.e., $d(b)=0$, when b is false then $d(b) > 0$ *and* $d(b) < k$, where k is an arbitrary positive constant, for example $k=1$, and when b is ⊥, then $d(b)$ is k. If a *Boolean* variable is obtained from an operation call, then in general the branch distance would take one of these three possible values. For example, when the operation *isEmpty()* is called on a collection, the branch distance would take *0, a value between 0 and k*, 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 in Section 4.2.2. Note that OCL supports two other special values, which are *null* and *invalid*. These two values have truth tables identical to *undefined* (Section 7.4 in [2]), and so these values are treated exactly the same way as we treat *undefined* in our approach. In the rest of the paper, we only provide branch distance calculations corresponding to a *Boolean* clause evaluating to *true*, *false*, or *undefined*. Whenever, a clause evaluates to *invalid* or *null*, the distance is k, which is same as *undefined*.

The operations defined in OCL to combine *Boolean* clauses are *or*, *xor*, and, *not*, *if then else*, and *implies*. The three-valued truth values for these operations are shown in Table III. For these operations, branch distances are adopted from [29], but are extended to handle *undefined* (⊥). For example, for *A and B*, the branch distance is calculated as follows:

$$d(\text{A and B}) := \text{numberOfUndefinedClauses} + \text{nor}\,(d(A)+d(B))$$

Notice that the standard definition of d(A and B) is d(A)+d(B) as specified in [29], but this calculation only accounts for two truth values, i.e., *true* and *false*. By following the same definition to account for *undefined* (⊥), if B is *undefined*, then the branch distance will be d(A) + k. However, in this case, there is no gradient in avoiding the undefined value, i.e., turning from ⊥ to either *false* or *true*. To provide a gradient for turning ⊥ to either *true* or *false*, we extended the standard branch distance calculation by adding *numberOfUndefinedClauses*. This variables holds the number of *undefined* clauses in an expression and we normalize d(A)+d(B) between 0 and 1, so that the search gives priority to minimize *numberOfUndefinedClauses*, i.e., turning ⊥ to *false*/ *true*. If the search manages to do that, the distance will be decreased by 1. In the same, way we extend the standard branch distance calculation of *or* as specified in [29].

Operations *implies*, and *xor* are syntactic sugars that usually do not appear in programming languages such as C and Ja-

va, and can be re-expressed using combinations of *and* and *or* operators as shown in Table III. A side effect of our extended branch distance for some *Boolean* operations such as *implies* and *xor* is that an expression *P* may be evaluated to *true* even if *d(P)*!=0. For example, *A implies B,* evaluates to true, when *A* is *false* and *B* is ⊥. In this case, *d(A implies B)* will be 1, but the expression is *true*. To deal with this, our implementation stops the search when an expression is *true*, even if its distance is still not zero. The evaluation of *d()* on an expression composed by two clauses is specified in Table IV and can simply be computed for more than two clauses recursively.

Table III. Truth values for Boolean operations [2]

b1	**b2**	**b1 and b2**	**b1 or b2**	**b1 xor b2**	**b1 implies b2**	**not b1**
false	false	false	false	false	true	true
false	true	false	true	true	true	true
true	false	false	true	true	false	false
true	true	true	true	false	true	false
false	⊥	false	⊥	⊥	true	true
true	⊥	⊥	true	⊥	⊥	false
⊥	false	false	⊥	⊥	⊥	⊥
⊥	true	⊥	true	⊥	true	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥

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

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

* A and B are Boolean expressions or variables, nor(x)=x/x+1

When an expression or one of its parts is negated, then the expression 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 extended the branch distance calculation from [29] for three-valued logic as shown in Table V.

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 dis-

tance 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 [53] function, but any other string matching distance function can easily be incorporated.

Table V. Branch distance calculations of OCL relational operations for numeric data

Relational operations	Distance function
x=y	**if not** x.oclIsUndefined() **and not** y.oclIsUndefined() **and** abs (x-y=0) **then** 0 **else not** x.oclIsUndefined() **and not** y.oclIsUndefined() **and** abs (x-y<>0) **then** k*nor(abs(x-y)) **else** **then** k
x<>y	**if not** x.oclIsUndefined() **and not** y.oclIsUndefined() **and** abs (x-y<>0) **then** 0 **else not** x.oclIsUndefined() **and not** y.oclIsUndefined() **and** abs (x-y=0) **then** k*nor(abs(x-y)) **else** **then** k
x<y	**if not** x.oclIsUndefined() **and not** y.oclIsUndefined() **and** abs (x-y < 0) **then** 0 **else** not x.oclIsUndefined() **and not** y.oclIsUndefined() **and** abs (x-y>=0) **then** k*nor(abs(x-y)) **else** **then** k
x<=y	**if not** x.oclIsUndefined() **and not** y.oclIsUndefined() **and** abs (x-y <= 0) **then** 0 **else not** x.oclIsUndefined() **and not** y.oclIsUndefined() **and** abs (x-y>0) **then** k*nor(abs(x-y)) **else** **then** k
x>y	**if not** x.oclIsUndefined() **and not** y.oclIsUndefined() **and** abs ((y-x) < 0) **then** 0 **else not** x.oclIsUndefined() **and not** y.oclIsUndefined() **and** abs ((y-x) >= 0) **then** k*nor(abs(x-y)) **else** **then** k
x>=y	**if not** x.oclIsUndefined() **and not** y.oclIsUndefined() **and** abs ((y-x) <= 0) **then** 0 **else not** x.oclIsUndefined() **and not** y.oclIsUndefined() **and** abs ((y-x) > 0) **then** k*nor(abs(x-y)) **else** **then** k

4.2.2 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 4.2.1. 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/ expressions and hence are not discussed in the litera-

ture.

4.2.2.1 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 finer grained heuristic than using a simple heuristic which simply calculates *0* if the result of an evaluation is *true*, a value between *0* and *k* if the result is *false*, and *k* otherwise. In Figure 3, a branch distance for equality (=) of collections is calculated in one of the following four ways.

```
if (C1.oclIsUndefined() or C2.oclIsUndefined())
    then d(C1=C2) := 1
if not (C1.oclIsKindOf(C2))
    then d(C1=C2) >= 0.75 and d(C1=C2) < 1
otherwise if C1→size() <> C2→size()
    then d(C1=C2) := 0.5 + 0.25*nor(d (C1→size()=C2 → size()))
otherwise
    then d(C1 = C2) := 0.5 * Σ_{i=1}^{C1→size()} nor(d(pair_i))/C1 → size()
where, d(pair_i) is the distance between elements in the i^th position in the two
sorted collections, e.g., d(C1.at(i)=C2.at(i)) and nor is a normalizing function
[61] defined as nor(x)=x/(x+1). Suppose C1 and C2 are two OCL collections.
```

Figure 3. Branch distance equality for collections

First, if any of Collections, i.e., *C1* and *C2* are *undefined*; then the distance is simply 1. Second, if collections *C1* and *C2* are not of the same kind, i.e., *not (C1.oclIsKindOf(C2))* evaluates to *true*, then the distance is a value from starting from 0.75 and less than 1. Note that any other constant (e.g., *k*) could have been used to represent the maximum distance (1 in this case). Whenever, the distance is less than *1* and greater than equal to 0.75, 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 second condition is satisfied, the search algorithms must be guided such that the collections have equal number of elements. The third 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 → size()=C2 → size()*. We compute *d(C1 → size()=C2 → size())* and since *size()* returns an integer, this distance calculation is simply performed using the equality operation on numerical data as shown in Table V. 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 + 0.25*nor(d (C1 → size()=C2 → size()))

⇔ 0.5 + 0.25*nor(abs(C1 → size()-C2 → size())+k) [Using formula for equality for numerical data from Table V]

⇔ 0.5 + 0.25*(Y/ (Y+1)) [Using the definition of nor(X), and assuming Y= abs(C1 → size()-C2 → size())+k]

In the above equation, *Y/(Y+1)* always computes a value less than *1*. Equation *0.5 + 0.25*(Y/ (Y+1))* therefore always takes a value between *0.5* to and 0.75. Whenever, *d(C1,C2)* is greater than *0.5* and less than *0.75*, this means that collections

do not have the same number of elements.

For the fourth 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. If the collections are either of kind *Set* or *Bag*, we first sort both collections based on their elements. Sorting the two collections facilitates their equality evaluation by reducing the number of needed comparisons of elements. After sorting, we only need to compare each pair of elements from the two collections at the same index (*C1→size()* comparisons) instead of comparing all elements from one collection with all elements in the other collection (*C1→size()*C1→size()* comparisons). 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 recursively traverse the collection and sort elements in each contrained collection that are of 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}*. When a collection consists of objects, a similar process is followed except that we perform sorting based on the values of the instance variables of the objects.

Table VI. Minimum and maximum distance values for distance calculation for equality of collections

Condition	Minimum	Maximum
C1.oclIsUndefined() **or** C2.oclIsUndefined()	1	1
not (C1.oclIsKindOf(C2))	>=0.75	<1
C1 → size() <> C2 → size()	>=0.5	<0.75
not (C1.oclIsKindOf(C2)) and C1 → size() = C2 → 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 ith element of *C1* with the ith 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:

$$d(C1 = C2) \Leftrightarrow 0.5 * \sum_{i=1}^{C1 \to size()} nor(d(pair_i)) \Big/ C1 \to size()$$

$$\Leftrightarrow 0.5 * (\sum_{i=1}^{C1 \to size()} d(pair_i)/(d(pair_i)+1) \Big/ C1 \to size()) \text{ [Using definition of } nor\text{]}$$

$d(pair_i)/ (d(pair_i)+1)$ will always compute a value less than *1*. Considering a simple example, in which collections consist of *Boolean* values, using the formula from Table IV, $d(pair_i)$ can take *k* as the maximum value. So the formula will be re-

duced to:

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

$$\Leftrightarrow 0.5 \; * \; (k/(k+1))$$

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 *not (C1.oclIsKindOf(C2))* and *C1 → size() = C2 → 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:

$$d(C1 = C2) \Leftrightarrow 0.5 * \sum_{i=1}^{C1\rightarrow size()} nor(d(pair_i))/C1 \rightarrow size()$$

$$\Leftrightarrow 0.5 * \sum_{i=1}^{3} nor(d(pair_i))/3$$

$$\Leftrightarrow 0.5 * (nor(d(1 = 4)) + nor(d(2 = 5)) + nor(d(3 = 9)))/3 \text{ [Sorted C1 and C2 will be \{1,2,3\} and \{4,5,9\}]}$$

$$\Leftrightarrow 0.5*(nor(4)+nor(4)+nor(7))/3 \text{ [Using k=1 and formula of equality of two numeric values from Table V]}$$

$$\Leftrightarrow 0.28$$

Table VII. Branch distance calculation for operations checking objects in collections

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

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

4.2.2.2 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 VII.

For *includes(object:T)*, a branch distance is the minimum distance in the set of all distances (calculated using the heuristic for equality as listed in Table V) between *object* and each element of the collection (*self*) on which includes is invoked. Getting the minimum distance will effectively guide a search algorithm to generate data that will make an element in *self*

equal to the *object*. 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 → includes(4),* then the branch distance will be calculated as:

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

$$\Leftrightarrow \min_{i=1 \text{ to } 3} d(\text{object} = \text{C.at}(i))$$

$$\Leftrightarrow \min(d(1 = 4), d(2 = 4), d(3 = 4)))$$

$$\Leftrightarrow \min(4,3,2) \text{ [Using k=1, and formula of the equality of two integers from Table V]}$$

$$\Leftrightarrow 2$$

For *excludes(object:T)*, a branch distance is calculated in a similar way as *includes*, except: (1) we use the distance heuristic for inequality (<>); (2) sum up the distances of all elements in the collection, which are equal to *object* in order to ensure that *object* is not present even once in *self.* The example below illustrates how a branch distance is computed using the formula.

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

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

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

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

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

$$\Leftrightarrow d(1 <> 2) + d(2 <> 2) + d(2 <> 2)$$

$$\Leftrightarrow 0 + 1 + 1 \text{ [Using k=1, and formula from Table V]}$$

$$\Leftrightarrow 2$$

In a similar fashion, we calculate branch distance of *includesAll* and *excludesAll* (Table VII), 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 VII. For *excludesAll*, we sum all distances between all possible pairs of elements across the two collections, as shown in the for-

mula for *excludesAll* in Table VII. Branch distance calculations for *isEmpty* and *notEmpty* are also defined in Table VII.

4.2.2.3 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 distances are summed up because *boolean-expression* must be *true* for all elements of the collection on which *forAll* is invoked. This means that more the elements for which *boolean-expression* is false, the higher will be the branch distance. The function for *forAll* presented in Table VIII is generic for any number of iterators. For the sake of clarity in the paper, we assume that function *expr(v1,v2, ... vm)* in Table VIII evaluates an expression *expr* on a set of elements *v1,v2, ... vm*. To explain *expr*, suppose we have a collection *C={1,2,3}* and an expression *C → forAll(x,y | x*y >0)*, then *expr(C.at(1),C.at(2))* entails calculating *"d(x*y>0)"*, where *x=C.at(1)*, i.e., *1* and *y= C.at(2)*, i.e., *2*. The keyword *self* in the table refers to the collection on which an operation is applied, *at(i)* is a standard OCL operation that returns the ith element of a sequence or an ordered set, and *size()* is another OCL operation that returns the number of elements in a collection. The denominator $(self \rightarrow size())^m$ is used to compute the average distance over all element combinations of size *m* since we have $(self \rightarrow size())^m$ 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 → forAll(x| x>5)* and *C → 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 → forAll(x| x=0)*. In this example, we have

just one iterator *x*, and therefore *m=1*. In this case, the formula will be:

$$d(C \to forAll(x|x = 0)) \Leftrightarrow \sum_{i_1=1}^{C\to size()} d(expr(C.at(i_1)) / C \to size()$$

The branch distance in this case will be calculated as:

d(C→forAll(x | x=0)) ⇔ (d(expr(C.at(1))+ d(expr(C.at(2))+ d(expr(C.at(3)))/3

⇔ (d(expr(1)+ d(expr(2)+ d(expr(3))/3

⇔ (2+3+4)/3[k=1, definition of *expr* and formulae from Table IV and Table V]

⇔ 9/3 = 3

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

$$d(C \to forAll(x, y|x * y > 0)) \Leftrightarrow \frac{\sum_{i_1=1}^{C\to size()} \sum_{i2=1}^{C\to size()} d(expr(C.at(i_1), C.at(i_2))}{(C \to size())^2}$$

The branch distance in this case will be calculated as:

⇔ (d(expr(C.at(1), C.at(1)))+ d(expr(C.at(1), C.at(2)))+ d(expr(C.at(2), C.at(1)))+ d(expr(C.at(2), C.at(2))))/4 C.at(2))))/4

⇔ (d(expr(1, 1))+ d(expr(1, 2))+ d(expr(2, 1))+ d(expr(2, 2)))/4

⇔ (d(1*1>0)+ d(1*2>0)+ d(2*1>0)+ d(2*2>0))/4

⇔ (0+0+0+0)/4 [Considering k=1 and using formulae from Table IV and Table V]

⇔ 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 VIII. 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 → exists(x/ x=0)*. In this example, we have just one iterator, i.e., *x*. The formula will be:

$$d(C \to exists(x|x = 0)) \Leftrightarrow \min_{i_1 \in 1 to\ 3} d(expr(C.at(i_1))$$

The branch distance in this case will be calculated as:

⇔ min (d(expr(C.at(1)), d(expr(C.at(2)), d(expr(C.at(3)))

⇔ min (d(expr(1), d(expr(2), d(expr(3))

⇔ min (2,3,4) [Using k=1, *expr*, and formulae from Table IV and Table V]

⇔ 2

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

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

The branch distance in this case will be calculated as:

⇔ min (d(expr(C.at(1), C.at(1))), d(expr(C.at(1), C.at(2))), d(expr(C.at(2), C.at(1))), d(expr(C.at(2), C.at(2))))

⇔ min (d(expr(1, 1)), d(expr(1, 2)), d(expr(2, 1)), d(expr(2, 2)))

⇔ min (d(1*1>1), d(1*2>1), d(2*1>1), d(2*2>1))

⇔ min (1,0,0,0) [Considering k=1, the definition of expr, and using formulae from Table IV and Table V]

⇔ 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 VIII. 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 → size())*(self → 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()*.

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

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

$\Leftrightarrow (d(expr(C.at(1)) <> expr(C.at(2))) + d(expr(C.at(1) <> expr(C.at(3))) + d(expr(C.at(2)) <> expr(C.at(3))))/((3*2)/2)$

$\Leftrightarrow (d(1 <> 1) + d(1 <> 3) + d(1 <> 3))/3$

⇔ (1+0+0)/3

⇔ 1/3

Table IX. Special rules for *Select()* followed by *Size()* when *exp* is *false*

Operation	Distance function
>, >=	d(exp)= if C → size() <= z() then (z()-C → size()) + k else nor((z()- C → select(P) → size())+k + nor(d(P)))
<,<=	d(exp) = if C → size() >= z() then (C → size()-z())+k else nor((C → select(P)→size()-z())+k + nor(d(not P)))
<>	d(exp) = if C → select(P) → size() = 0 then d(P) if C → select(P) → size() = C → size() then d(not P) if 0 < C → select(P) → size() < C → size() then min(d(P), d(not P))
=	d(exp) = if C → select(P) → size() > z() then (C → select(P) → size()-z())+k + nor(d(not P)) if C → select(P) → size() < z() then (z()- C → size())+k + nor(d(P))

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

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 4.2.1. 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:

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

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

```
exp := C → select (P) → size()  =  z()
If  exp = true then
        d(exp)=0
else
        d(exp)= | C → select(P) → 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 >=, when *exp* is *false*, this means that the size of resultant collection of the expression *C → 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 that *z()*, and then we need to obtain those elements of *C* that increase the value of *size()* returned by *C → select(P) → size()*. This can be achieved by the rule shown in the first row of Table IX. The normalization function *nor()* is necessary because the branch distance should first reward any increase in *C → 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 → select(P) → size())+k)*. The function *d(P)* returns the sum of branch distance evaluations of an expression *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 → select(x| x>1) → size()>=3*. Using the formula of branch distance for the case when *RelOp* is >, >=.

In this case, *C→size()* is *3*, which is equal to *z()*, i.e., *3*, so the formula for branch distance calculation is:

d(C → select(x|x>1) →size()>=3) ⇔ nor((z()- C → select(P) → size())+k + nor(d(P)))

⇔ nor((3-1)+1 + nor(d(x>1))) [Assuming k=1]

⇔ nor(3 + nor(d(x>1)))

⇔ nor(3 + nor(d(1>1)+d(1>1)+d(3>1)))

⇔ nor(3 + nor(1+1+0)) [Using k=1, and formula from Table V]

⇔ nor(3 + 0.667)

⇔ 0.78

For < and <=, when *exp* is *false*, this means that *C → select(P) → 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 → select(P) → size()*, and uses *nor(d(not P))* to provide additional gradient to the search, as shown in second row of Table IX.

For the cases when the value of *RelOp* is inequality (<>), the rule is shown in the third row of Table IX. Recall that our expression is in the following format: *exp = C → selectionOp (P) → size() RelOp z()*. For this rule, there are three cases based on the value of *C → select(P) → size()*. Recall that *d(P)* is simply the sum of all *d()* on all elements of *C*. The first case is when *C → select(P) → 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 → select(P) → size() = C → 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 → select(P) → size() < C → 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 IX. There are two important cases, which work in a similar way as the first and second cases as reported in Table IX. The first case is when *C → select(P) → size() > z()*, where we need to decrease *C → select(P) → size()*, which can be achieved by minimizing *(C → select(P) → size()-z())+k + nor(d(not P))*. The second case is when *C → select(P) → size() < z()*. For this case, we need to increase the number of elements in *C* for which *P* holds and must minimize *(z()-C → select(P) → size())+k + nor(d(P))*.

Note that we only presented formulae in Table IX 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 IX 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 X. For example, $C \rightarrow$ *select(P1)* $\rightarrow$ *forAll(P2)* (first row in Table X) implies that for all elements of *C* for which *P1* holds, *P2* should also hold. In other words, *P1 implies P2*. Therefore, $C \rightarrow$ *select(P1)* $\rightarrow$ *forAll(P2)* can simply be transformed into $C \rightarrow$ *forAll(P1 implies P2)*. Similarly, a *select(P1)* followed by an *exists(P2)* 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., $C \rightarrow$ *select(P1)* $\rightarrow$ *select (P2)* is equivalent to $C \rightarrow$ *select(P1and P2)*.

The effectiveness of all these rules for calculating branch distance is empirically evaluated in Section 7.

4.2.3 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 V.

4.2.4 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 *oclInState*.

Table X. Special rules for *Select()* followed by *forALL* and *exists*

Operation	Distance function
C → select(P1) → forAll(P2)	d(C → forAll(P1 implies P2))
C → select(P1) → exists(P2)	d(C → exists (P1 and P2))

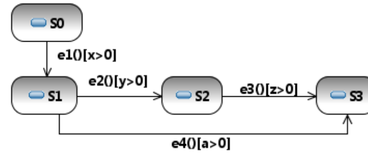

Figure 4. A dummy example to explain *oclInState()*

4.2.5 Enumerations

Enumerations are data types 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 *0, a value between 0 and k,* or *k*.

4.2.6 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 data type similar to enumeration. This data type 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 the 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 [62]). 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 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:

self.packetLoss.value > 5 and self.packetLoss.value <=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 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 [29] 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 the 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 [6].

4.2.7 Miscellaneous Operations

OCL defines several special operations that can be used with all types of objects: *oclIsTypeOf()*, *oclIsKindOf()*, *oclIsNew()*, *oclIsUndefined()*, and *oclIsInvalid()*. 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 super types 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 op-

eration evaluates to *true*, then the branch distances is *0*, if the operation evaluates to *false* then the branch distance is a value between 0 and *k*, and *k* when the operation evaluates to *undefined*. We deal with *oclIsUndefined()* and *oclIsInvalid* operations in the same way.

4.2.8 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. The pre and post conditions of these operations are 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 [63]. 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 4.2.4. For instance, consider a user-defined OCL operation named *operation1()*, which is defined on a collection and returns a collection, and is used in the following constraint:

c1 → operation1() → isEmpty()

In this case, the branch distance is calculated based on the heuristic for *isEmpty()* as defined in Section 4.2.2.

5. TOOL SUPPORT AND RUNNING EXAMPLE

In this section, we present our implementation of search-based test data generation and a running example to demonstrate how the generated data is used by our Model-based testing tool TRUST [64] to generate executable test cases.

5.1 Tool Support

To efficiently generate test data from OCL constraints, we implemented a search-based approach. Figure 5 shows the architecture diagram for our search-based test data generator. We developed a tool in Java that interacts with an existing library, an OCL evaluator, called the EyeOCL Software (EOS) [36]. 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 use EOS as it is one the most efficient evaluators currently available; however any other OCL evaluator may be used. 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. For a constraint, the search space is defined by those attributes that are used in the constraint. This is determined by statically parsing a constraint before solving it and improves the search efficiency in a similar fashion to the concept of input size reduction [65]. The search algorithms employed are implemented in Java as well and include Genetic Algorithms, (1+1) Evolu-

tionary Algorithm, and Alternating Variable Method (AVM). Note that our implementation of branch distance calculation corresponds to OCL semantics as specified in [2]. As a result, any other OCL evaluator can be plugged into our implementation without affecting our implementation of branch distance calculation.

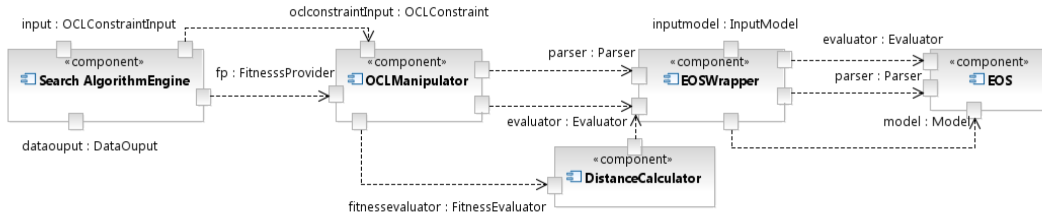

Figure 5. Architecture diagram for the search-based test data generator

5.2 Running Example of Test Data Generation

In this section, we present a running example to demonstrate how the generated test data is used in the test cases generated using our model-based testing tool, called TRUST [64]. Figure 6 shows a class diagram for a very simplified version of the Video Conferencing System (VCS) class diagram, with one class *VideoConferencingSystem*. This class has one attribute *NumberOfActiveCalls*, which holds the number of current participants in a videoconference. The class has two operations *dial(Number:Integer)*, which is used to dial to a particular VCS using an Integer *Number* (valid values range from 4000 to 5000). The state machine in Figure 6 models the behavior of VCS, where it dials to another VCS using *dial()* subject to guard (*Number>=4000 and Number<=5000*) evaluating to *true*, before transiting to the *Connected* state. From *Connected*, when *disconnect()* is called the VCS transits to *Idle*. Notice that in this example, we need to solve the guard and generate data for *Number*. Each state in Figure 6 has a state invariant written in OCL, which serves as a test oracle.

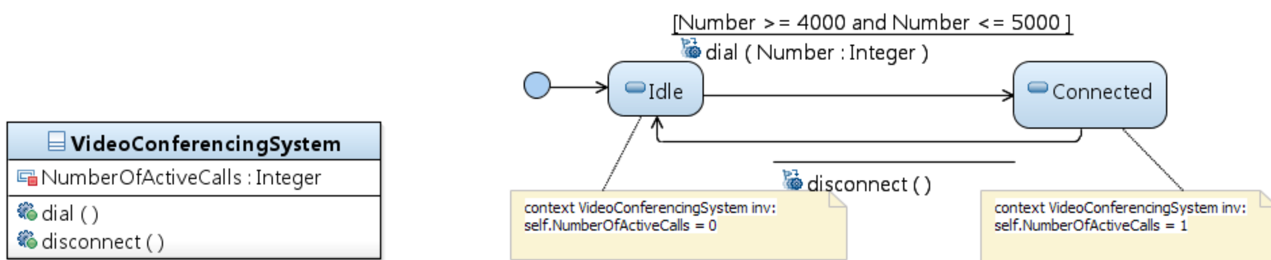

Figure 6. A small running example for Video Conferencing System

TRUST works in a series of steps to generate executable test cases including test data. For test case generation, TRUST manipulates models in three sequential steps as shown in Table XI and Figure 7. In the first step (implemented as a set of Kermeta [66] model-model transformations), TRUST flattens UML state machines with hierarchy and concurrency. Details and algorithms for this step can be found in [64]. In our running example, the state machine does not have hierarchy or concurrency, so the input and output for this step are the same. In Step 2, the flattened state machine produced by Step 1 is converted into an instance of TestTree ([64]) representing a set of abstract test paths based on coverage criteria, such as *All*

Transitions and *All States* coverage. In our running example, using *All States* coverage, we obtain an abstract test path shown in Row 3 of the *Output* column in Table XI. Notice that the actual representation of an abstract test path is in XML, but to demonstrate the test case generation process and for readability reasons, we show it as text. Step 2 is also implemented as a set of Kemeta transformations. In the third step (Test case generation), each abstract test path is transformed to an executable test case using a set of model-to-text transformations implemented in MOFScript [67]. For this step, as it can be seen in Figure 7, TRUST uses our test data generator by passing it the guard written in OCL (*Number>=4000 and Number <=5000*) and obtains a valid value for *Number*, which in our running example is 4005 as shown in Line 3 of the test script in Row 3, Output column in Table XI. This value is used in *dial()* to dial to another VCS in Line 4. Line 2 and Line 6 of the test script checks the current state of the VCS with the state invariant (OCL constraint) passed as a *String* in *checkState()* (Line 2 and Line 6). These state invariants are evaluated using EyeOCL [36] at run time and the result of the evaluation (*true* or *false*) tells whether the VCS was in an incorrect state or not.

Table XI. Steps for test case generation using TRUST

Test Case Generation Step	Inputs	Output
State machine flattening	Class diagram and state machine (Fig.5)	Class diagram and state machine (Fig.5)
Abstract test path generation	Class diagram and state machine (Fig.5), All states coverage	Idle->[Number>=4000 and Number <=5000].dial(Number:Integer)->Connected
Test case generation	Idle->[Number>=4000 and Number <=5000].dial(Number:Integer)->Connected	1. # Check state invariant for Idle 2. boolean result = checkState("self.NumberOfActiveCalls =0") 3. **number = 4005** 4. dial(number) 5. wait(20sec) 6. result = checkState("self.NumberOfActiveCalls =1")

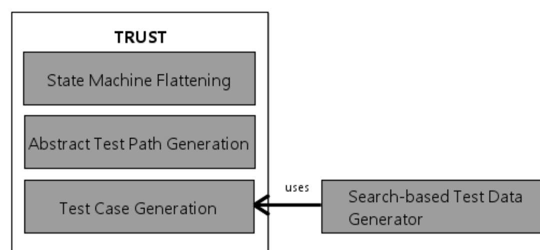

Figure 7. Integration of TRUST with the Search-based Test Data Generator

6. 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 [64], 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 [68] 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 subsequently weaves robustness behavior into the standard behavior and generates a standard UML 2.0 state machine. Some details and models of the case study, including a partial woven state machine, are provided in [68]; however, due to confidentiality reasons, we cannot provide further details. Interested readers may also visit the official website of the system we tested for more information [69]. 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' keyword and associated condition, and is triggered when this condition is met during the execution of a system. An example of such a change event is shown in Figure 8. This change event is fired during a videoconference when the synchronization between audio and video passes the allowed threshold. The non-functional property *synchronizationMismatch* is defined using the MARTE profile [70], and measures the synchronization between audio and video over 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 characteristics of these constraints, in terms of number of conjuncted clauses and their frequency of occurrence are reported in Table XII. Most constraints contain between 6 and 8 clauses. The different OCL data types used in these constraints are shown in Table XIII and we can see that most of the primitive types are being used in our case study. Notice that in our industrial case study, we needed to generate test data only for guards and change events of UML state machines. Due to the nature of the case study, the number of objects for each class were fixed (i.e., fixed upper limit of multiplicity) since objects corresponds to actual hardware features of a VCS such as audio and video channels.

Table XII. Characteristics of constraints

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

Table XIII. 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 synchronozationConstraint:
  self.media.synchronizationMismatch.value > self.media.synchronizationMismatchThreshold.value
```

Figure 8. A constraint checking synchronization of audio and video in a videoconference

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 the TRUST tool [64]. 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 [5].

6.1 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.

6.1.1 Experiment Design

We designed our experiment using the guidelines proposed in [28, 71]. The objective of our experiment is to assess the efficiency of the selected search algorithms 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 [28]. (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 [61]). We used RS as the comparison baseline to assess the difficulty of the addressed problem [28].

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

RQ1: Are search-based techniques effective and efficient at solving OCL constraints?

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?

6.1.2 Experiment Execution

We ran experiments for 57 OCL expressions from the VCS industrial case study that we discussed earlier. The number of clauses in each expression varies from one to eight and the median value is six. The characteristics of the problems are summarized in Table XII, where we provide details on the distribution of numbers of clauses. In Table XIII, we summarized the data types used in the problems.

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

[72], 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 for 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 runs on all 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).

Though putting a limit on the number of fitness evaluations is necessary for experimental purposes, in practice one would instead 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 if the constraint was not yet solved. For example, the choice of this threshold could be driven by the testing budget. However, though adequate in practice, in an experimental context, 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. We selected steady state GA with a population size of 100 and a crossover rate of 0.75, with a 1.5 bias for rank selection. We used 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 [72]. As we will show, since our search-based test data generator is already very effective in solving OCL constraints, 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 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.

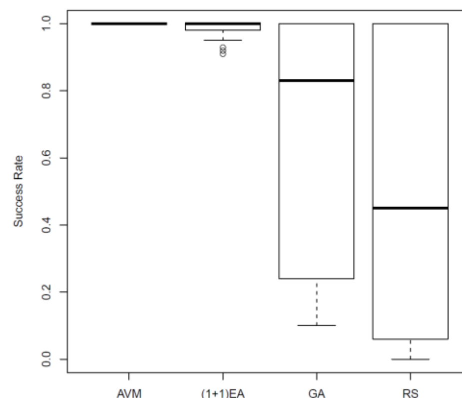

Figure 9. Success rates for various algorithms

Table XIV . Success rates for individual problems

Problem Id	# of conjuncted clauses	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

6.1.3 Results and Analysis

Figure 9 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%.

Table XIV 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 composed of just one or two clauses, as it can be seen from the number of conjuncted clauses column in Table XV. The problems with higher number of clauses are the most difficult to solve for GA and RS, as shown in Table XV. As the number of conjuncted clauses 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 an average success rate of 100% even for problems with higher numbers of conjuncted clauses. In the case of (1+1) EA, the minimum average success rates are for the problems with 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) which are likely to be encountered in industrial systems.

Table XV . Average success rates for problems of varying characteristics

# of conjuncted clauses	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 XVI. 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

To check the statistical significance of the results, we carried out a paired Mann-Whitney U-test (paired per constraint) at the significance level (α) 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 XVI. This shows a strong statistical significance in the

differences among the four algorithms when applied to all 57 constraints in our case study. In addition, we performed a Fisher's exact test with α=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 XVII and Table XVIII. 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 [71] for this purpose, as the results of our experiments are dichotomous. Table XVII and Table XVIII also show the odds ratios for various pairs of approaches for all 57 problems. For AVM vs. (1+1) EA, we did not observe practically significant differences for most of the problems, except for Problem 2 and Problem 33, where AVM performed significantly 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 performed significantly 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.

Table XVII. Results for the Fisher's Exact test at significance level of 0.05

	AVM vs. (1+1) EA		**AVM vs. GA**		**AVM vs. RS**		**(1+1 EA) vs. GA**		**(1+1) EA vs. RS**		**GA vs. RS**	
ID	**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

Based on the above results, we recommend using AVM 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. Note that in case studies with more complex problems, a larger number of iterations may be required to eventually solve the problems.

Table XVIII. Results for the Fisher's Exact test at significance level of 0.05

	AVM vs. (1+1) EA		**AVM vs. GA**		**AVM vs. RS**		**(1+1 EA) vs. GA**		**(1+1) EA vs. RS**		**GA vs. RS**	
ID	**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	0,001542052	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

```
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 10. Condition for a change event which is fired when synchronization between audio and video is within threshold

6.2 Comparison with UMLtoCSP

UMLtoCSP [13] 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 characteristics are summarized in Table XII. An example of such constraint, modeling a change event on a transition of Saturn's state machine, is shown in Figure 10. 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 UMLtoCSP tool did not support UML 2.x diagrams, we also needed to recreate our models in a UML 1.x modeling tool.

Table XIX. Results of comparison with UMLtoCSP

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)**	**UMLtoCSP (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 test data generator managed to solve each constraint within at most 2.96 seconds using AVM and 99 Seconds using (1+1) EA, as shown in Table XX. For the remaining 15 problems 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 XIX. We provide the time taken by UMLtoCSP to solve each problem in seconds, which is reported by the tool itself and is 0.01 second (maximum precision) for all fifteen constraints. For these same 15 problems, we ran our tool 100 times and also report the average time taken by our tool to solve each problem over 100 runs in Table XIX. Since we used the same machine to run experiments for both tools, it is clear that for all fifteen problems comprising of one or two clauses, 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 consid-

ering 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.

Table XX. 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

7. Empirical Evaluation of Optimized Fitness Functions

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

7.1 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 (with small solution space) 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 XXI 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 search algorithms among the ones used in the industrial case study (Section 5 and in other works [61]): (1+1) EA and AVM. In this experiment, we address the following research question:

RQ3: Do 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-1* otherwise). We will refer to them here as Optimized (*Op*) and Non-Optimized (*NOp*) branch distance calculations, respectively.

7.2 Experiment Execution

We ran experiments 100 times for (1+1) EA and AVM, with *Op* and *NOp*, and for each problem listed in Table XXI. 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 XXI. Artificial problems for heuristics

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

7.3 Results and Analysis

Table XXII 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 [73] with α=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*. 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 [73], with α=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 [71]. Table XXIII shows the results of the test and p-values are bold-faced when results are significant. In Table XXIII, 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 Â12 statistics, which is a non-parametric effect

size measure. We chose this effect size measure using again the guidelines reported in [71]. In our context, the value of Â12 tells the probability for *Op* to find a solution in more iterations than *NOp*. This means that the higher the value of Â12, 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 Â12 is 0.5. With 1+1 (EA), for A9 and A10, *Op* took significantly less iterations to solve the problems (Table XXIII) as both p-values are below 0.05. In addition, for A9 and A10, values of Â12 are 0.19 and 0.46, respectively, thus showing that *Op* took more iterations to solve the problem than *NOp* in 19% and 46% of the time

Table XXII. Results of Fisher Exact Test for success rate of Optimized and Non-Optimized at α=0.05

Problem #	Success Rate (1+1)EA (*NOp*) in %	Success Rate for (1+1)EA (*Op*) in %	Fisher Exact Test for (1+1)EA (p-value)	Success Rate for AVM (*NOp*) in %	Success Rate for AVM (*Op*) in %	Fisher Exact Test for AVM (p-value)
A1	0	100	-	0	100	-
A2	2	100	**0,0001**	0	59	-
A3	1	95	**0,0001**	0	99	-
A4	3	100	**0,0001**	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	**0,0001**
A10	100	100	-	100	100	-
A11	0	94	-	0	99	-
A12	0	100	-	0	34	-
A13	0	100	-	0	100	-

Table XXIII. Results of t-test at α=0.05 ((1+1)EA)

Problem #	Mean Difference (OP-NOP)	Â12	p-value
A9	-654,38	0,19	**0,0001**
A10	-1,01	0,46	**0,004**

Table XXIV. Results of t-test at α=0.05 (AVM)

Problem #	Mean Difference (OP-NOP)	Â12	p-value
A10	-0,23	0,52	**0,04**

For AVM, the results of success rates for A10 were tied between *Op* and *NOp* (Table XXII). 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 [73] with α=0.05 to determine if significant differences exist between *Op* and *NOp*. Table XXIV shows mean differences, p-values, and Â12 values. We observed that for these problems *Op* took less iterations to solve them and significant differences were observed for A10.

Based on the above results, we can clearly see that (1+1) EA and AVM with optimized branch distance calculations significantly improve the success rates. In the worst cases, when there is no difference in success rates between *Op* and *NOp*, (1+1) EA and AVM took significantly less iterations to solve the problems.

8. 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 than GA and RS since the algorithms achieve 100% and 98% success rates for all 57 constraints on average, respectively (Section 6.1). For the experiments based on artificial problems (Section 7.3), 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 that of AVM. For instance, in Table XXII 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 performed a Fisher's exact test to determine if the differences are statistically significant with α=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 XXII). In this case, we obtain a p-value of 0.21 when we apply 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 XXII). 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 7, 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.

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 as we discussed in Section 6.1 and Section 7.1.

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 this test since it is appropriate for dichotomous data where proportions must be compared [71], 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 [72].

We ran our experiments on an industrial case study to generate test data for 57 different OCL constraints, ranging from 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 and such industrial case studies are nevertheless very rarely reported in the research literature. 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 characteristics.

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 since 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 test data generator 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) when this is relying on the UML modeling standard or extensions, as it is often the case in industrial contexts. Existing approaches in the literature have one or more of the following problems that make them difficult to use in industrial applications: (1) they miss key features of OCL such as collections, which are common in industrial problems; (2) they translate OCL into formalisms such as first order logic, temporal logic, or Alloy, and thus often result into combinatorial explosion problems that 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 test data generator. 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 test data generator 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 standard and 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.

11. 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 UMLtoCSP, for helping us to run UMLtoCSP on our industrial case study. Lionel Briand was in part supported by a PEARL grant from the Fonds National de la Recherche, Luxembourg

12. 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*. Available: http://www.omg.org/spec/OCL/2.2/ , Accessed: April, 2012
[3] MOF. *Meta Object Facility (MOF)*. Available: http://www.omg.org/spec/MOF/2.0/ , Accessed: April, 2012
[4] T. Yue, L. Briand, B. Selic, and Q. Gan., "Experiences with Model-based Product Line Engineering for Developing a Family of Integrated Control Systems: an Industrial Case Study," Simula Research Laboratory, Technical Report(2012-06)2012.
[5] S. Ali, L. Briand, A. Arcuri, and S. Walawege, "An Industrial Application of Robustness Testing using Aspect-Oriented Modeling, UML/MARTE, and Search Algorithms," presented at the ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (Models 2011), 2011.
[6] A. Arcuri, M. Z. Iqbal, and L. Briand, "Black-box System Testing of Real-Time Embedded Systems Using Random and Search-based Testing," presented at the IFIP International Conference on Testing Software and Systems (ICTSS), 2010.
[7] CertifyIt. *CertifyIt*. Available: http://www.smartesting.com/ , Accessed: April, 2012
[8] QTRONIC. *QTRONIC*. Available: http://www.conformiq.com/qtronic.php, Accessed: April, 2012
[9] L. v. Aertryck and T. Jensen, "UML-Casting: Test synthesis from UML models using constraint resolution," presented at the Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2003), 2003.
[10] M. Benattou, J. Bruel, and N. Hameurlain, "Generating test data from OCL specification," 2002.
[11] L. Bao-Lin, L. Zhi-shu, L. Qing, and C. Y. Hong, "Test case automate generation from uml sequence diagram and ocl expression," presented at the International Conference on cimputational Intelligence and Security, 2007.
[12] S. Ali, M. Z. Iqbal, A. Arcuri, and L. Briand, "A Search-based OCL Constraint Solver for Model-based Test Data Generation," in *Proceedings of the 11th International Conference On Quality Software (QSIC 2011)*, 2011, pp. 41-50.
[13] J. Cabot, R. Claris, and D. Riera, "Verification of UML/OCL Class Diagrams using Constraint Programming," presented at the Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop, 2008.
[14] A. P. Mathur, *Foundations of Software Testing*: Pearson Education, 2008.
[15] A. Bertolino, "Software testing research: achievements, challenges, dreams," presented at the 2007 Future of Software Engineering, 2007.
[16] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology,* vol. 43, pp. 833-839, 2001.
[17] J. Clarke, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd, "Reformulating software engineering as a search problem," *IEE Software* vol. 150, pp. 161-175, 2003.
[18] M. Harman, A. Mansouri, and Y. Zhang, "Search Based Software Engineering: Trends, Techniques and Applications " *To Appear in ACM Computing Surveys,* 2012.
[19] R. Drechsler and N. Drechsler, *Evolutionary Algorithms for Embedded System Design*: Kluwer Academic Publishers, 2002.
[20] D. A. Coley, *An Introduction to Genetic Algorithms for Scientists and Engineers*: World Scientific Publishing Company, 1997.
[21] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology,* vol. 51, pp. 957-976, 2009.
[22] T. Mantere and J. T. Alander, "Evolutionary software engineering, a review," *Applied Soft Computing,* vol. 5, pp. 315-331, 2005.
[23] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability,* vol. 14, p. 52, 2004.
[24] J. T. d. Souza, C. L. Maia, F. G. d. Freitas, and D. P. Coutinho, "The Human Competitiveness of Search Based Software

Engineering," presented at the Proceedings of the 2nd International Symposium on Search Based Software Engineering, 2010.

[25] G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software," in *ACM Symposium on the Foundations of Software Engineering (FSE)*, 2011, pp. 416-419.

[26] G. Fraser and A. Arcuri, "Sound Empirical Evidence in Software Testing," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2012.

[27] E. K. Burke and G. Kendall, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*: Springer 2006.

[28] 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.

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

[30] S. Droste, T. Jansen, and I. Wegener, "On the analysis of the (1+ 1) evolutionary algorithm," *Theor. Comput. Sci.,* vol. 276, pp. 51-81, 2002.

[31] B. Korel, "Automated Software Test Data Generation," *IEEE Trans. Softw. Eng.,* vol. 16, pp. 870-879, 1990.

[32] D. Chiorean, M. Bortes, D. Corutiu, C. Botiza, and A. Cârcu, "OCLE," V2.0 ed, 2010.

[33] *Model Development Tools*. Available: http://www.eclipse.org/modeling/mdt/?project=ocl, Accessed: April, 2012

[34] *Dresden OCL*. Available: http://www.dresden-ocl.org/index.php/DresdenOCL, Accessed: April, 2012

[35] *UML-based Specification Environment (USE)*. Available: http://sourceforge.net/apps/mediawiki/useocl/index.php?title=The_UML-based_Specification_Environment, Accessed: April, 2012

[36] M. Egea, "EyeOCL Software," ed, 2010.

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

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

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

[40] B. K. Aichernig and P. A. P. Salas, "Test Case Generation by OCL Mutation and Constraint Solving," presented at the Proceedings of the Fifth International Conference on Quality Software, 2005.

[41] D. Berardi, D. Calvanese, and G. D. Giacomo, "Reasoning on UML class diagrams," *Artif. Intell.,* vol. 168, pp. 70-118, 2005.

[42] 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.

[43] 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.

[44] M. P. Krieger, A. Knapp, and B. Wolff, "Automatic and Efficient Simulation of Operation Contracts," presented at the 9th International Conference on Generative Programming and Component Engineering, 2010.

[45] S. Weißleder and B.-H. Schlingloff, "Deriving Input Partitions from UML Models for Automatic Test Generation," in *Models in Software Engineering*, ed: Springer-Verlag, 2008, pp. 151-163.

[46] 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.

[47] A. D. Brucker, M. P. Krieger, D. Longuet, and B. Wolff, "A specification-based test case generation method for UML/OCL," presented at the Proceedings of the 2010 international conference on Models in software engineering, Oslo, Norway, 2011.

[48] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. H, #228, hnle, W. Menzel, and P. H. Schmitt, "The KeY Approach: Integrating Object Oriented Design and Formal Verification," presented at the Proceedings of the European Workshop on Logics in Artificial Intelligence, 2000.

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

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

[51] 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.

[52] R. Lefticaru and F. Ipate, "Functional Search-based Testing from State Machines," presented at the Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation, 2008.

[53] 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.

[54] P. McMinn, M. Shahbaz, and M. Stevenson, "Search-Based Test Input Generation for String Data Types Using the Results of

Web Queries," presented at the Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, 2012.
[55] G. Fraser and A. Arcuri, "The Seed is Strong: Seeding Strategies in Search-Based Software Testing," presented at the Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, 2012.
[56] G. Fraser and A. Arcuri, "It is Not the Length That Matters, It is How You Control It," presented at the Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, 2011.
[57] G. Fraser and A. Arcuri, "Whole Test Suite Generation," *IEEE Transactions on Software Engineering,* 2012.
[58] S. Poulding, J. A. Clark, and H. Waeselynck, "A Principled Evaluation of the Effect of Directed Mutation on Search-Based Statistical Testing," presented at the Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, 2011.
[59] S. Yoo, M. Harman, and S. Ur, "Highly scalable multi objective test suite minimisation using graphics cards," presented at the Proceedings of the Third international conference on Search based software engineering, Szeged, Hungary, 2011.
[60] 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.
[61] A. Arcuri, "It really does matter how you normalize the branch distance in search-based software testing," *Software Testing, Verification and Reliability,* 2011.
[62] R. V. Binder, *Testing object-oriented systems: models, patterns, and tools*: Addison-Wesley Longman Publishing Co., Inc., 1999.
[63] H. Li and Gordon, "Bytecode Testability Transformation " presented at the Symposium on Search based Software Engineering Co-located with ESEC/FSE, 2011.
[64] 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.
[65] M. Phil, "Input Domain Reduction through Irrelevant Variable Removal and Its Effect on Local, Global, and Hybrid Search-Based Structural Test Data Generation," *IEEE Transactions on Software Engineering,* vol. 38, pp. 453-477, 2012.
[66] Kermeta. *Kermeta - Breathe Life into Your Metamodels*. Available: http://www.kermeta.org/, Accessed: April, 2012
[67] *MOFScript Home page*. Available: http://www.eclipse.org/gmt/mofscript/ (September 2009), Accessed: April, 2012
[68] S. Ali, L. C. Briand, and H. Hemmati, "Modeling Robustness Behavior Using Aspect-Oriented Modeling to Support Robustness Testing of Industrial Systems," *Software and Systems Modeling,* vol. 11, 2012.
[69] *Cisco C90*. Available: http://www.cisco.com/en/US/products/ps11330/index.html, Accessed: April, 2012
[70] MARTE. *Modeling and Analysis of Real-time and Embedded systems (MARTE)*. Available: http://www.omgmarte.org/, Accessed: April, 2012
[71] A. Arcuri and L. Briand., "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering," presented at the International Conference on Software Engineering (ICSE), 2011.
[72] A. Arcuri and G. Fraser, "On Parameter Tuning in Search Based Software Engineering," presented at the International Symposium on Search Based Software Engineering (SSBSE), 2011.
[73] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*: Chapman and Hall/CRC, 2007.