2011 11th International Conference On Quality Software

A Search-based OCL Constraint Solver for Model-based Test Data Generation

Shaukat Ali[1,2], Muhammad Zohaib Iqbal[1,2], Andrea Arcuri[1], Lionel Briand[1,2]
[1]Simula Research Laboratory, Norway
[2]Department of Informatics, University of Oslo, Norway
{shaukat, zohaib, arcuri, briand}@simula.no

***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 Object Constraint Language (OCL). Given that test data generation is one of the major challenges to automate MBT, this is the topic of this paper with a specific focus on test data generation from OCL constraints. Though search-based software testing (SBST) 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 based on OCL constraints to guide test data generation and automate MBT in industrial applications. These heuristics are used to develop an OCL solver exclusively based on search, in this particular case genetic algorithm and (1+1) EA. Empirical analyses to evaluate the feasibility of our approach are carried out on one industrial system.**

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

I. INTRODUCTION

Model-based testing (MBT) has recently received increasing attention in both industry and academia. MBT promises systematic, automated, and thorough testing, which would likely not be possible without models. However, the full automation of MBT, which is a requirement for scaling up to large systems, requires solving many problems, including preparing models for testing (e.g., flattening state machines), defining appropriate test strategies and coverage criteria, and generating test data to execute test cases. Furthermore, in order to increase chances of adoption, using MBT for industrial applications requires using well-established standards, such as the Unified Modeling Language (UML) and its associated language to write constraints: the Object Constraint Language (OCL) [1].

OCL [1] is a standard language that is widely accepted for writing constraints on UML models. OCL is based on first order logic and is a highly expressive language. The language allows modelers to write constraints at various levels of abstraction and for various types of models. It can be used to write class and state invariants, guards in state machines, constraints in sequence diagrams, and pre and post condition of operations. A basic subset of the language has been defined that can be used with meta-models defined in Meta Object Facility (MOF) [2] (which is a standard defined by Object Management Group (OMG) for defining meta-models). This subset of OCL has been largely used in the definition of UML for constraining various elements of the language. Moreover, the language is also used in writing constraints while defining UML profiles, which is a standard way of extending UML for various domains using pre-defined extension mechanisms.

Due to the ability of OCL to specify constraints for various purposes during modeling, for example when defining guard conditions or state invariants in state machines, such constraints play a significant role when testing is driven by models. For example, in state-based testing, if the aim of a test case is to execute a guarded transition (where the guard is written in OCL based on input values of the trigger) 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, either target only a small subset of OCL [3, 4], are not scalable, or lack proper tool support [5]. 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 a contribution by devising novel heuristics for the application of search-based techniques, such as Genetic Algorithms (GAs) and (1+1) Evolutionary Algorithm (EA), to solving OCL constraints (covering the entire OCL 2.2 semantics [1]) in order to generate test data. A search-based OCL

1550-6002/11 $26.00 © 2011 IEEE
DOI 10.1109/QSIC.2011.17

constraint solver is implemented and evaluated on the first reported, industrial case study on this topic.

The rest of the paper is organized as follows: Section II discusses the background and Section III discusses related work. In Section IV, we present the definition of distance function for various OCL constructs. Section 0 discusses the case studies and analysis of results of the application of the approach, whereas Section VI discusses the tool support and Section VII addresses the threats to validity of our empirical study. Finally, Section VIII concludes the paper.

II. BACKGROUND

Several software engineering problems can be reformulated as a *search problem*, such as test data generation [6]. An exhaustive evaluation of the entire *search space* (i.e., the domain of all possible combinations of problem variables) is usually not feasible. There is a need for techniques that are able to produce "good'' solutions in reasonable time by evaluating only a tiny fraction of the search space. Search algorithms can be used to address this type of problem. Several successful results by using search algorithms are reported in the literature for many types of software engineering problems [7-9].

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

There are several types of search algorithms. Genetic Algorithms (GAs) are the most well-known [7], and they are inspired by the Darwinian evolution theory. A population of individuals (i.e., candidate solutions) is evolved through a series of generations, where reproducing individuals evolve through *crossover* and *mutation* operators. (1+1) Evolutionary Algorithm (EA) is simpler than GAs, in which only a single individual is evolved with mutation. To verify that search algorithms are actually necessary because they address a difficult problem, it is a common practice to use Random Search (RS) as baseline [7].

III. RELATED WORK

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

A. OCL-based Constraint Solvers

A number of approaches use constraint solvers for analyzing OCL constraints for various purposes. These approaches usually translate constraints and models into a formalism (e.g., Alloy [10], temporal logic BOTL [11], FOL [12] , Prototype Verification System (PVS) [13], graph constraints [14]), which can then be analyzed by a constraint analyzer (e.g., Alloy constraint analyzer [15], model checker [11], Satisfiability Modulo Theories (SMT) Solver [12], theorem prover [12], [13]). Satisfiability Problem (SAT) solvers have also been used for the animation of OCL operation contracts (e.g., [16], [17]).

Some approaches are reported in the literature that generates test cases based on OCL constraints. Most of these approaches only handle a small subset of OCL and UML models and are based on formal constraint solving techniques, such as SAT solving (e.g., [3]), constraint satisfaction problem (CSP) (e.g., [18], [19]) and partition analysis (e.g., [5], [4]).

The work presented in [19] is one of the most sophisticated approaches in the literature. However, its focus is on verification of correctness properties, but to achieve this, it also generates an instantiation of the model. The major limitation of that approach is that the search space is bounded and, as the bounds are raised, the CSP faces a combinatorial explosion increase (as discussed in [19]). The task of determining the optimal bounds for verification is left to the user, which is not simple and requires repeated interaction from the user. Models of industrial applications can have hundreds of attributes and manually finding bounds for individual attributes is often impractical. We present the results of an experiment that we conducted to compare our approach with this approach in Section V.B.

Most of the above approaches are different from our work, since we want to generate test data based on OCL constraints provided by modelers on UML state and class diagrams. These diagrams may be developed for environment models or system models and the modeler should be allowed to use the complete set of standard OCL 2.2 notations. We want to provide inputs for which the constraints are satisfied, and not just verify them. We also want a tool that can be easily integrated with different state-based testing approaches and manual intervention should not be required for every run.

Existing approaches for OCL constraint solving do not fully fit our needs. Almost all of the existing works

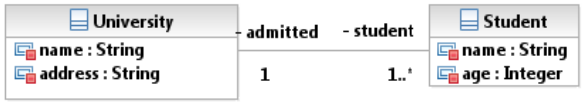

Figure 1. Example class diagram

only support a small subset of OCL. Most of the approaches are only limited to simple numerical expressions and do not handle collections (used widely to specify expressions that navigate over associations). This is generally due to the high expressiveness of OCL that makes the definitions of constraints easier, but their analysis more difficult. Conversion of OCL to a SAT formula or a CSP instance can easily result in combinatorial explosion as the complexity of the model and constraints increase (as discussed in [19]). For industrial scale systems, as in our case, this is a major limitation, since the models and constraints are generally quite complex. Most of the discussed approaches either *do not support* the OCL constructs present in the constraints that we have in our industrial case study or are not efficient to solve them (see Section V.B). Hence, existing techniques based on conversion to lower-level languages seem impractical in the context of large scale, real-world systems.

Instead of using search algorithms, another possible approach to cope with the combinatorial explosion faced in solving OCL constraints could be to use hybrid approaches that combine formal techniques (e.g., constraint solvers) with random testing (e.g. [20]). However, we are aware of no work on this topic for OCL and, even for common white-box testing strategies, performance comparisons of hybrid techniques with search algorithms are rare [21].

B. *Search-based Heuristics for Model Based Testing*

The application of search-based heuristics for MBT has received significant attention recently (e.g., [22], [23]). The idea of these techniques is to apply the 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* [24]. Our goal is to tailor this approach to OCL constraint solving for test data generation.

IV. DEFINITION OF THE FITNESS FUNCTION FOR OCL

To guide the search for test data that satisfy OCL constraints, it is necessary to define a set of heuristics. A heuristic would tell *'how far'* an input data is from satisfying the constraint. For example, let us say we

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

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

Figure 2. Example constraints

want to satisfy the constraint *x=0*, and suppose we have two data inputs: *x1:=5* and *x2:=1000*. Both inputs *x1* and *x2* do not satisfy *x=0*, but *x1* is *heuristically* closer to satisfy *x=0* than *x2*. A search algorithm would use such a heuristic as a fitness function, to reward input data that are closer to satisfy the target constraint.

In this paper, to generate test data to solve OCL constraints, we use a fitness function that is adapted from work done for code coverage (e.g., for branch coverage of code written in C [24]). In particular, we use the so called *branch distance* (a function *d()*), as defined in [24]. 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 as true. As for any heuristic, there is no guarantee that an optimal solution will be found in reasonable time, but nevertheless many successful results are reported in the literature for various software engineering problems [6].

Notice that, in some cases, we would want the constraints to evaluate to false (e.g., a transition in a state machine that should not be taken). To cope with these cases, we can simply negate the constraint and find data for which the negated constraint evaluates to true.

OCL is a constraint language that is more expressive than programming languages such as C and Java. Therefore, in this paper we extend the basic definition of branch distance to cope with *all* the features of the OCL 2.2 constraint language.

In this section, we give examples of how to calculate the branch distance for various kinds of expressions in OCL, including primitive data types (such as *Real* and *Integer*) and collection-related types (such as *Set* and *Bag)*. In OCL, all data types are subtypes of a super type *OCLAny*, which is categorized into two subtypes: primitive types and collection types. Primitive types are *Real*, *Integer*, *String*, and *Boolean*, whereas collection types include *Collection* as super type with subtypes *Set*, *OrderedSet*, *Bag*, and *Sequence*. A constraint can be seen as an expression involving one or more Boolean clauses connected with operators such as *and* and *or*. The truth value of a clause can depend on different types of properties involving variables of different types, such as equalities of integers and comparisons of strings. To explain this, consider the UML class diagram in Figure 1 consisting of two classes: *University* and *Student*.

TABLE I. BRANCH DISTANCE CALCULATIONS FOR OCL'S OPERATIONS FOR *BOOLEAN*

Boolean operations	Distance function
Boolean	**if** true **then** 0 **otherwise** k
A and B	d(A)+d(B)
A or B	**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))

TABLE II. BRANCH DISTANCE CALCULATIONS OF OCL'S RELATIONAL OPERATIONS FOR NUMERIC DATA

Relational operations	Distance function
x=y	**if abs**(x-y) = 0 **then** 0 **otherwise abs**(x-y)+k
x<>y	**if abs**(x-y) <> 0 **then** 0 **otherwise** k
x<y	**if** x-y < 0 **then** 0 **otherwise** (x-y)+k
x<=y	**if** x-y <= 0 **then** 0 **otherwise** (x-y)+k
x>y	**if** (y-x) < 0 **then** 0 **otherwise** (y-x)+k
x>=y	**if** (y-x) <= 0 **then** 0 **otherwise** (y-x)+k

TABLE III. BRANCH DISTANCE CALCULATION FOR OPERATIONS CHECKING OBJECTS IN COLLECTIONS

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

Constraints on the class *University* are shown in Figure 2.

The first constraint states that the age of a *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 is called on collection *student* of the class *Student*, which is defined on collections in OCL and returns an *Integer* denoting the number of elements in a collection. Again, we have a comparison of integers, even though a function such as *size()* is called on a collection.

In the next section, we will discuss branch distance functions based on different types of clauses in OCL.

A. Primitive types

A *Boolean* variable *b* is either true (*d(b)=0*), or false (*d(b)=k*, where for example *k=1*). If the Boolean variable is obtained from a function call, then in general the branch distance would take one of only two possible values (*0* or *k*). However, when such calls belong to the standard OCL operations (e.g., the operation *isEmpty()* called on a collection), then in some cases we can provide more fine grained heuristics (we will specify which ones in more details later in this section).

The operations defined in OCL to concatenate *Boolean* clauses are *or*, *xor*, *and*, *not*, *if then else*, and *implies*. Branch distance for operations on *Boolean* are adopted from [24] and are shown in Table II. Operations *implies*, *xor*, and *if then else* are syntax sugars that usually do not appear in programming languages, such as C and Java, and can be expressed as combinations of *and* and *or*. The evaluation of *d()* on a predicate composed by two or more clauses is done recursively, as specified in Table I.

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

For the data types defined for numerical data such as *Integer* and *Real*, the relational operations defined that return Booleans (and so can be used as clauses) are <,>, <=,>=, and <>. For these operations, we adopted the branch distance calculation from [24] as shown in Table II.

In OCL, several other operations are defined on *Real* and *Integer* such as +, -, *, /, *abs()*, *div()*, *mod()*, *max()*, and *min()*. Since these operations are used as part of the calculation of two compared numerical values in a clause, there is no need to define a branch distance for them. For example, considering *a* and *b* are of type *Integer* and the constraint *a+b*3<4*, then the operations + and * are used only to define that 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 negative, we can return the value of any string matching distance to evaluate how close two strings are, as for example the edit distance [8].

TABLE IV. BRANCH DISTANCE FOR *FORALL* AND *EXISTS*

Operation	Distance function
forAll(v_1,v_2, …v_m|exp)	if self->size() = 0 then 0 otherwise $\frac{\sum_{i_1=1}^{self->size()} \sum_{i_2=1}^{self->size()} \ldots \sum_{i_m=1}^{self->size()} d(\exp(self.at(i_1),\ldots self.at(i_m))}{(self->size())^m}$
exists(v_1,v_2, …v_m|exp)	$\min_{i_1,i_2 \ldots i_m \in 1 to\ self \to size()} d(\exp(self.at(i_1), \ldots\ self.at(i_m))$
isUnique(v_1|exp)	$\sum_{i=1}^{(self \to size()-1)} \sum_{j=i+1}^{(self \to size())} d(\exp(self.at(i)) <> \exp(self.at(j)))$
one(v_1|exp)	$d(self \to select(exp) \to size() = 1)$

Enumerations in OCL are treated in the same way as enumerations in programming languages such as Java. Because enumerations are objects with no specific order relation, equality comparisons are treated as basic Boolean expressions, whose branch distance is either *0* or *k*.

B. Collection-Related Types

Collection types defined in OCL are *Set*, *OrderedSet*, *Bag*, and *Sequence*. Details of these types can be found in [1].

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 IV.A. An example is the *size()* operation, which returns an *Integer*.

In this section, we discuss branch distance for operations in OCL that are specific to collections, and that usually are not common in programming languages for expressing constraints/predicates and hence are not discussed in the literature.

1) Equality of collections (=): In OCL constraints, we may need to compare the equality of two collections. To improve the search process by providing a more fine-grained heuristic, we defined a branch distance for comparing collections as shown in Figure 4.

2) Operations checking existence of one or more objects in a collection: OCL defines several operations to check existence of one or more elements in a collection such as *includes()* and *excludes()*, which check whether an object exists in a collection 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 calculation of branch distance to improve the search, as described in Table III.

3) Branch distance for iterators: OCL defines several operations to iterate over collections. Below, we will discuss branch distance for these iterators.

The *forAll* iterator operation is applied to an OCL collection and takes as input a *Boolean* expression and determines whether the expression holds for all elements in the collection. For branch distance, we calculate the distance of the *Boolean* expression in *forAll*. Boolean expression on all elements in the collection is conjuncted. To avoid a bias toward reducing the size of the collection on which the predicate is evaluated, we scale the resulting distance by the number of elements in the collection. The general branch distance function for *forAll* is shown in Table IV. For the sake of clarity in the paper, we assume that function *exp(v_1,v_2, …v_m)* evaluates an expression *exp* on a set of objects v_1,v_2, …v_m in Table IV. *Self* in the table refers to the collection on which an operation is applied, *at(i)* is a standard OCL operation that returns the i^{th} element of a collection, and *size()* is another OCL operation that returns the number of elements in a collection.

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 distance is computed for each element of the collection on which the *Boolean* expression is applied and the results are disjuncted. The general distance form for *exists* is shown in Table IV. In addition, we also provide branch distance for *isUnique()* and *one()* operations in the same table.

Select, reject, collect, and iterator 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, which do not belong to the collection on which it is applied. Since all these iterators return a collection and not a Boolean value, we do not need to define branch distance for them, as discussed in Section IV.A.

V. CASE STUDY: ROBUSTNESS TESTING OF VIDEO CONFERENCE SYSTEM

This case study is part of a project aiming at supporting automated, model-based robustness testing of a core subsystem of a video conference system (VCS) called Saturn [25] developed by *Tandberg AS* (now part of Cisco Systems, Inc). Saturn is modeled as

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

Figure 3. A constraint checking synchronization of audio and video in a videoconference

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 [26] 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 system or its environment such as the network and other systems interacting with the system. A weaver later on weaves robustness behavior into the standard behavior and generates a standard UML 2.0 state machine. The woven state machine is provided in [26]. This woven state machine is used for test case generation. In this current, simplified case study, the woven state machine has 11 states and 93 transitions. Out of 93 transitions, 73 transitions model robustness behavior and 47 out of 73 are unique, all of them requiring test data that satisfy the constraints to traverse them. All these 47 transitions have change events or triggers. A change event is fired when a condition is met during the operation of a system. An example of such change event is shown in Figure 3. This change event is fired during a videoconference when the synchronization between audio and video passes the allowed threshold. *SynchronizationMismatch* is a non-functional property defined using the MARTE profile, which measures the synchronization between audio and video in time.

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 targeted robustness faults, for example related to faulty situations in the network and other systems that comprise the environment of the SUT. Test cases are generated from the system state machines using our tool TRUST [25]. 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 [25].

```
if not (A.oclIsKindOf(B))
    d(A=B) := 1
otherwise if A->size() <> B->size()
    d(A=B) := 0.5 + 0.5*n(d (A->size()=B->size()))
otherwise
d(A=B) := 0.5 *  sum( n(d(pair)) )/A->size()
```

where, d(pair) = distance between each paired element in the collection, e.g., d(A.at(i)=B.at(i)) and n is a normalizing function [27], and it is defined as n(x)=x/(x+1). Suppose A and B are two collections in OCL.

Figure 4. Branch distance equality of collections

A. Empirical Evaluation

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

1) Experiment Design: We designed our experiment using the guidelines proposed in [7, 28]. The objective of our experiment is to assess the efficiency of search algorithms such as GAs to generate test data by solving OCL constraints. In our experiments, we compared three search techniques: GA, (1+1) EA, and RS. GA was selected since it is the most commonly used search algorithm in search-based software engineering [7]. (1+1) EA is simpler than GAs, but in the previous work in software testing we found that it can be more effective in some cases (e.g., see [9]). We used RS as the comparison baseline to assess the difficulty of the addressed problem [7].

In this paper, we want to answer the following research questions.

RQ1: Are search-based techniques effective and efficient at solving OCL constraints in the models of industrial systems?

RQ2: Among the considered search algorithms, which one performs best in solving OCL constraints?

2) Experiment Execution: We ran experiments for 47 OCL predicates as we discussed in Section 0. The number of clauses in each predicate varies from one to eight and the median value is six. Each algorithm was run 100 times to account for the random variation inherent to randomized algorithms.

A solution is represented as an array of variables, the same that appear in the OCL constraint we want to solve. For GA, we set the population size to 100 and the crossover rate to 0.75, with a 1.5 bias for rank selection. We use a standard one-point crossover, and mutation of a variable is done with the standard probability *1/n*, where *n* is the number of variables.

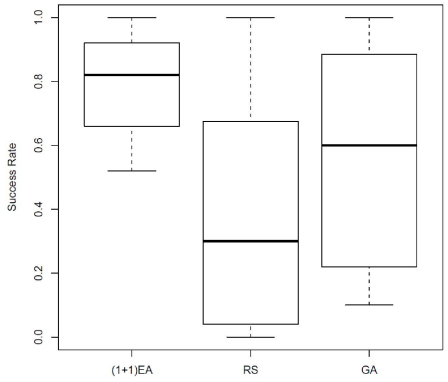

Figure 5. Success rates for various algorithms

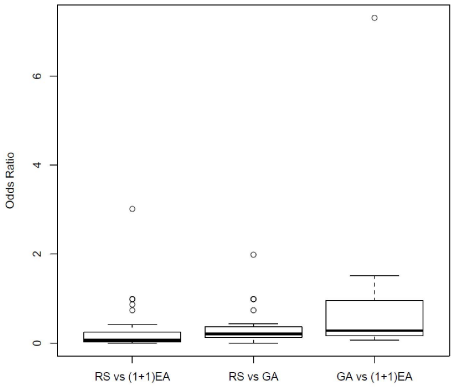

Figure 6. Odds ratio between pairs of algorithms

We ran each algorithm up to 2000 fitness evaluations on each problem and collected data on whether an algorithm found the 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. Instead of putting a limit to the number of fitness evaluations, a more practical approach would be to run as many iterations as possible, but stopping once a predefined time threshold is reached (e.g., 10 minutes) if the constraint has not been solved yet. The choice of the threshold would be driven by the testing budget. However, though useful in practice, using a time threshold would make it significantly more difficult and less reliable to compare different search algorithms (e.g., accurately monitoring the passing of time, side effects of other processes running at same time, inefficiencies in implementation details).

To compare the algorithms, we calculated their success rate, which is defined as the number of times an algorithm was successful in finding optimal solutions out of the total number of runs.

3) Results and Analysis: Figure 5 shows a box plot of the success rate of the 47 problems for (1+1) EA, GA, and RS. For each search technique, the box-plot is based on 47 success rates, one for each constraint. The results show that (1+1) EA outperformed both RS and GA, whereas GA outperformed RS. We can observe that, with an upper limit of 2000 iterations, (1+1) EA achieves a median success rate of 80% but GA does not exceed a median roughly 60%. We can also see that all success rates for (1+1) EA are above 50% and most of them are close to 100%. Constraints with the lowest success rates are seven and eight clauses long. Even taking the lowest success rates for the *most difficult* constraints (50%), this would entail that with r runs of (1+1) EA, we would achieve a success rate of $1 - (1 - 0.5)^r$. For example, with $r = 7$, we would obtain a success rate above 99%. This entails a computation time of approximately 3.8*7=27 minutes. Given that we use a slow prototype (EyeOCL) for OCL expression analysis and that we could parallelize the search, our results suggest that our approach is effective, efficient, and therefore practical, even for difficult constraints (RQ1).

To check the statistical significance of the results, we performed Fisher's exact test between each pair of algorithms based on their success rates for the 47 constraints. Due to space limitations, we do not present p-values for each problem and each pair of algorithms. In summary, we observe that for 105 times out of 141 (47*3, where 3 represent the number of algorithm pairs), results were significant at the 0.05 level. We also carried out a *paired* Mann-Whitney U-test (paired per constraint) on the distributions of the success rates for the three algorithms. In all the three distribution comparisons, p-values were very close to 0, and hence showing a strong statistical difference among the three algorithms when applied on all the 47 constraints (although on some constraints there is no statistical difference, as the 141 Fisher's exact tests show).

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 [28] for this purpose, as the results of our experiments are dichotomous. Figure 6 shows box plots of odds ratio for pairs of algorithms for the 47 constraints. Between RS and (1+1) EA (the first column in Figure 6), the value of odds ratio is less than one, thus implying that (1+1) EA has more chances of success than RS. The odds ratio between RS and GA is also similar. Therefore, there is strong evidence to claim that (1+1) EA is significantly more successful than the other analyzed algorithms since, in most of the cases, the odds ratios comparing GA and RS with (1+1) EA (first and third column in Figure 6) show values not only lower than one, but also very close to zero (RQ2).

To check the complexity of the problems, we repeat the experiment on the negation of each of the 47

predicates. All algorithms managed to find solutions for all these problems very quickly. Most of the time and for most of the problems, each algorithm managed to find solutions in a single iteration. This result confirmed that the actual problems we targeted with search were not easy to solve.

In practice, given a time budget *T*, we recommend running (1+1) EA for as many iterations as possible. An alternative is to run the algorithms several times (e.g., *r,* so each run with budget *T/r*) but this is expected to be less effective as no information is reused between runs. But, in our experiments, this latter technique is already extremely effective (99% success rate with seven runs in the worst case).

B. Comparision with UMLtoCSP

UMLtoCSP [19] is the most widely used and referenced OCL constraint solver in the literature. To check the performance of UMLtoCSP to solve complex constraints such as the ones in our current industrial case study, we conducted an experiment. We selected the 10 most complex constraints (based on the number of clauses in a constraint) from our industrial application, which comprises constraints ranging from six to eight clauses (we did not analyzed all the 47 constraints because, as we will show, these experiments took substantial computational time). An example of such constraint, modeling a change event on a transition of Saturn's state machine, is shown in Figure 7. This change event is fired when Saturn is successful in recovering the synchronization between audio and video. Since UMLtoCSP does not support enumerations, we converted each enumeration into an Integer and limited its bound to the number of literals in the enumeration. We also used the MARTE profile to model different non-functional properties, and since UMLtoCSP does not support UML profiles, we explicitly modeled the used subset of MARTE as part of our models. In addition, UMLtoCSP does not allow writing constraints on inherited attributes of a class, so we modified our models and modeled inherited attributes directly in the classes. We set the range of Integer attributes from 1 to 100.

We ran the experiment on the same machine as we used in the experiments reported in the previous section. Though we let UMLtoCSP address each of the selected constraints for *10 hours each*, it was not successful in finding any valid solution. A plausible explanation is that UMLtoCSP is negatively affected by the state explosion problem, a common problem in real-world industrial applications such as the one from Tandberg/Cisco used in this paper. In contrast, even in the worst case, our constraint solver managed to solve each constraint within at most 27 minutes, as we have reported in the previous section.

VI. Tool Support

We developed a tool in Java that interacts with an existing library, an OCL evaluator called EyeOCL [29]. EyeOCL is a Java component that provides APIs to parse and evaluate an OCL expression based on an object model. Our tool implements the calculation of branch distance as discussed in Section IV for various expressions in OCL. To calculate branch distance for an OCL expression, we send this expression for parsing to EyeOCL and obtain a parse tree of the expression. We manipulate the parse tree and call EyeOCL with the current set of values for variables in the expression and calculate the branch distance. The search algorithms employed in this paper were implemented in Java as well.

VII. Threats to Validity

To reduce *construct validity* threats, we chose the measure *success rate*, which is comparable across all three algorithms ((1+1) EA, GA and RS) that we used. Furthermore, we used the same stopping criterion for all algorithms, i.e., number of fitness evaluations. This criterion is comparable across all the algorithms that we studied because each iteration requires updating the object diagram in EyeOCL and evaluating a query on it. This time is same for all the algorithms, and it is rather expensive (approximately, 0.114 second per iteration).

The most probable *conclusion validity* threat in experiments involving randomized algorithms is due to random variation. To address it, we repeated experiments 100 times to reduce the possibility that the results were obtained by chance. Furthermore, we perform Fisher exact test to compare proportions to determine statistical significance of results. We chose Fisher's exact test because it is appropriate for dichotomous data where proportions must be compared, thus matching our case [28]. To determine practical significance of results, we measure 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 line with common guidelines in the literature and our previous experience on testing problems.

In the empirical comparisons with UMLtoCSP, there is the threat that we might have wrongly configured it. To reduce the probability of such an event, we contacted the authors of UMLtoCSP who were very helpful in ensuring its proper use.

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

Figure 7. A change event checking which is fired when synchronization between audio and video is within threshold

We ran our experiments on an industrial case study to generate test data for 47 different OCL constraints, ranging from simpler constraints having just one clause to complex constraints having eight clauses. Although the empirical analysis is based on a real industrial system and not on small artificial problems (as most work in the literature [11], [13], and [16]), our results might not generalize to other case studies. However, such threat to *external validity* is common to all empirical studies.

From our analysis of UMLtoCSP, we cannot generalize our results to traditional constraint solvers in general when applied to solve OCL constraints. However, empirical comparisons with other constraints solvers were not possible because, to the best of our knowledge, UMLtoCSP is not only the most referenced OCL solver but also the only one that is publically available.

VIII. CONCLUSION

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

To overcome the abovementioned problems, we defined a set of heuristics based on OCL constraints to guide search-based algorithms (genetic algorithms, (1+1) EA) and implemented them in our search-based OCL constraint solver. More specifically, we defined branch distance functions for various types of expressions in OCL to guide search algorithms. We demonstrated the effectiveness and efficiency of our search-based constraint solver to generate test data in the context of the model-based, robustness testing of an industrial case study of a video conferencing system. Even for the most difficult constraints, with research prototypes and no parallel computations, we obtain test data within 27 minutes in the worst case and in less than 4 minutes on average.

As a comparison, we ran the 10 most complex constraints on one well-known, downloadable OCL solver (UMLtoCSP) and the results showed that, even after running it for 10 hours, no solutions could be found. Similar to all existing OCL solvers, because it could not handle all OCL constructs, we had to transform our constraints to satisfy UMLtoCSP requirements.

We also conducted an empirical evaluation in which we compared three 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 (1+1) EA was significantly better than GA, which itself were significantly better than random search. Notice that in both empirical evaluations, the execution times were obtained on a regular PC.

Future work will consider hybrid approaches, in which traditional constraint solver techniques will be integrated with search algorithms, with the aim to overcome the current limitations that both approaches have and exploit the best of both worlds.

IX. 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 (Tandberg AS, part of Cisco Systems, Inc) for providing us the case study.

X. REFERENCES

[1] Object Constraint Language Specification, Version 2.2, Object Management Group (OMG), http://www.omg.org/spec/OCL/2.2/, 2010

[2] Meta Object Facility (MOF), http://www.omg.org/spec/MOF/2.0/, 2006

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

[4] M. Benattou, J. Bruel, and N. Hameurlain, "Generating test data from OCL specification," Citeseer, 2002.

[5] L. Bao-Lin, L. Zhi-shu, L. Qing, and C. Y. Hong, "Test case automate generation from uml sequence diagram and ocl expression," in *International Conference on cimputational Intelligence and Security*, 2007, pp. 1048-1052.

[6] 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.

[7] 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.

[8] 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.

[9] A. Andrea, "Longer is Better: On the Role of Test Sequence Length in Software Testing," International Conference on Software Testing, Verification, and Validation, 2010, pp. 469-478.

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

[11] D. Distefano, J.-P. Katoen, and A. Rensink, "Towards model checking OCL," in *ECOOP-Workshop on Defining Precise Semantics for UML*, 2000.

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

[13] 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.

[14] 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.

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

[16] M. Krieger and A. Knapp, "Executing Underspecified OCL Operation Contracts with a SAT Solver," in *8th International Workshop on OCL Concepts and Tools.* vol. 15: ECEASST, 2008.

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

[18] B. K. Aichernig and P. A. P. Salas, "Test Case Generation by OCL Mutation and Constraint Solving," in *Proceedings of the Fifth International Conference on Quality Software*: IEEE Computer Society, 2005.

[19] J. Cabot, R. Claris, and D. Riera, "Verification of UML/OCL Class Diagrams using Constraint Programming," in *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*: IEEE Computer Society, 2008.

[20] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," *SIGSOFT Softw. Eng. Notes,* vol. 30, pp. 263-272, 2005.

[21] K. Lakhotia, P. McMinn, and M. Harman, "An empirical investigation into branch coverage for C programs using CUTE and AUSTIN," *Journal of Systems and Software,* vol. 83, pp. 2379-2391.

[22] 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.

[23] R. Lefticaru and F. Ipate, "Functional Search-based Testing from State Machines," in *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*: IEEE Computer Society, 2008.

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

[25] 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.

[26] S. Ali, L. C. Briand, and H. Hemmati, "Modeling Robustness Behavior Using Aspect-Oriented Modeling to Support Robustness Testing of Industrial Systems," Simula Research Laboratory, Technical Report (2010-03)2010.

[27] A. Arcuri, "It Does Matter How You Normalise the Branch Distance in Search Based Software Testing," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*: IEEE Computer Society.

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

[29] M. Egea, EyeOCL Software, http://maude.sip.ucm.es/eos/, 2010