Regression Testing for Model Transformations: A Multi-objective Approach

Jeffery Shelburg, Marouane Kessentini, and Daniel R. Tauritz

Department of Computer Science
Missouri University of Science and Technology, Rolla, MO, USA
{jssdn2,marouanek}@mst.edu, dtauritz@acm.org

Abstract. In current model-driven engineering practices, metamodels are modified followed by an update of transformation rules. Next, the updated transformation mechanism should be validated to ensure quality and robustness. Model transformation testing is a recently proposed effective technique used to validate transformation mechanisms. In this paper, a more efficient approach to model transformation testing is proposed by refactoring the existing test case models, employed to test previous metamodel and transformation mechanism versions, to cover new changes. To this end, a multi-objective optimization algorithm is employed to generate test case models that maximizes the coverage of the new metamodel while minimizing the number of test case model refactorings as well as test case model elements that have become invalid due to the new changes. Validation results on a widely used transformation mechanism confirm the effectiveness of our approach.

Keywords: search-based software engineering, testing, model transformation, multi-objective optimization.

1 Introduction

Model-Driven Engineering (MDE) considers models as first-class artifacts during the software lifecycle. The number of available tools, techniques, and approaches for MDE are growing that support a wide variety of activities such as model creation, model transformation, and code generation. The use of different domain-specific modeling languages and diverse versions of the same language increases the need for interoperability between languages and their accompanying tools [1]. Therefore, metamodels are regularly updated along with their respective transformation mechanism.

Afterwards, the updated transformation mechanism should be validated to assure quality and robustness. One efficient validation method proposed recently is model transformation testing [1,2] which consists of generating source models as test cases, applying the transformation mechanism to them, and verifying the result using an oracle function such as a comparison with an expected result. Two challenges are: (1) the efficient generation of test cases, and (2) the definition of the oracle function. This paper focuses on the efficient generation of test cases in the form of source models.

G. Ruhe and Y. Zhang (Eds.): SSBSE 2013, LNCS 8084, pp. 209–223, 2013.

The generation of test case models for model transformation mechanisms is challenging because many issues need to be addressed. As explained in [3], testing model transformation is distinct from testing traditional implementations; the input data are models that are complex when compared to simple data types which complicates the generation and evaluation of test case models [4]. The basis of the work presented in this paper starts from the observation that most existing approaches in testing evolved transformation mechanisms regenerate all test cases from scratch. However, this can be a very fastidious task since the expected output for all test cases needs to be completely redefined by hand. Furthermore, when the number of changes made between metamodel versions is relatively small in comparison to metamodel sizes, redefining all test case output is inefficient. A better strategy is to revise existing test cases to cover new changes in metamodels to reduce the effort required to manually redefine expected test case output.

In this paper, a multi-objective search-based approach is used to generate test case models that maximizes the coverage of a newly updated metamodel while minimizing the number of refactorings applied to existing test case models in addition to minimizing the number of test case model elements that have become invalid due to the new changes. The proposed algorithm is an adaptation of multi-objective simulated annealing (MOSA) [5] and aims to find a Pareto optimal solution consisting of test case model refactorings that will yield the new test case models when applied to existing test case models of the previous version metamodel that best satisfy the three criteria previously mentioned. This approach is implemented and evaluated on a known case of transforming UML 1.4 class diagrams to UML 2.0 class diagrams [6]. Results detailing the effectiveness of the proposed approach are compared to results of a traditional simulated annealing (SA) approach (whose single objective is to maximize metamodel coverage) to create UML 2.0 test case models in two scenarios: (1) updating test case models for UML 1.4 and (2) creating new test case models from scratch. Results indicate that the proposed approach holds great promise as using MOSA from previous test case models attains slightly less metamodel coverage than using SA with, however, significantly less refactorings and invalid model elements while always outperforming both methods starting from scratch.

The primary contributions of this paper are summarized as follows: (1) A novel formulation of the model transformation testing problem is introduced using a novel multi-objective optimization technique, and (2) results of an empirical study comparing the proposed MOSA approach to a traditional SA approach in scenarios starting from previous test case models and from scratch are reported. The obtained results provide evidence supporting the claim that the proposed MOSA approach requires less manual effort to update expected output than SA and starting from existing test case models is more effective than regenerating all test case models from scratch.

2 Methodology

In this section, the three main components of any search-based approach are defined: the solution representation, change operators, and objective function.

2.1 Solution Representation

Since the proposed approach needs to modify test case models in response to changes at the metamodel level, the solution produced should yield a modified version of the original test case models that best conforms to the updated metamodel. This can be done primarily in one of two different ways: the solution could either consist of the actual updated test case model itself, or represent a structure that, when applied to the original test case models, produces the updated test case models. The latter was chosen for this problem in the form of lists of model refactorings, because it allows a sequence of refactorings to be modified at any point in the sequence.

For example, if a search-based method was employed to generate the new test case models and a suboptimal refactoring was included in the best found test case model solution at some point in the search process, it would be difficult to reverse the application of the suboptimal refactoring to the best found test case model solution if the test case model was modified directly. This is because it would need to search and find the refactoring in the space of all possible refactorings to apply to the test case model to reverse or change the suboptimal refactoring. By modifying a list of model refactorings, it is easier for the search process to remove or modify the suboptimal refactoring because it has direct access to the model refactorings included in the best found solution. Furthermore, maintaining a list of the best sequence of refactorings found during the search process gives direct information about what exactly was changed from the previous version test case models that makes updating the expected output a simpler task.

Each element in the lists of refactorings solution representation is a list of model refactorings that corresponds to a test case model. Each list of model refactorings is comprised of model refactorings that are applied to the corresponding test case model in the order in which they appear in the list. Applying these refactorings transforms the existing test case models into the updated test case models for the updated metamodel. An example of model refactorings that can be applied to UML class diagrams are shown in Table 1. Figure 1 shows an example of a possible list of UML refactorings for a test case model that moves method *getAge* from class *Employee* to class *Person*, adds a *Salary* field to the *Employee* class, and then removes the *Job* class, in that order.

MoveMethod(getAge, Employee, Person)	AddField(Salary, Employee)	RemoveClass(Job)

Fig. 1. Example list of UML class diagram model refactorings

Table 1. UML class diagram model refactorings

Add Field	Add Association	Move Field	Push Down Field
Add Method	Add Generalization	Move Method	Push Down Method
Add Class	Remove Method	Extract Class	Pull Up Field
Remove Field	Remove Association	Extract Subclass	Pull Up Method
Remove Class	Remove Generalization	Extract Superclass	Collapse Hierarchy
Change Bi- to Uni-Directional Association		Change Uni- to Bi-Directional Association	

2.2 Change Operators

The only change operator employed in MOSA is mutation. When mutating a given test case model's list of refactorings, the type of mutation to perform is first determined from a user-defined probability distribution that chooses between inserting a refactoring into the list, removing a refactoring from the list, or modifying a refactoring in the list. When inserting a refactoring into a list of refactorings, an insertion point between refactorings is first chosen, including either ends of the list. The refactorings that appear in the list before the insertion point are first applied to the test case model in the order in which they appear in the list. A refactoring is then randomly generated for the refactored test case model as it exists at the selection point, applied to the model, and inserted into the list at the insertion point. The refactorings that appear after the insertion point in the list are then validated in the order in which they appear by first checking their validity and subsequently applying them to the test case model if they are valid. If a refactoring is found to be invalid due to a conflict caused by the insertion of the new refactoring into the list, the refactoring is removed from the list. An invalid refactoring could occur, for example, if a new refactoring is inserted into the front of a list that removes a specific class attribute that is referenced in an existing refactoring later in the list. If such an occurrence happened, the existing refactoring that references the now-removed class attribute will be removed from the list. When performing a mutation that removes a refactoring from a list of refactorings, a refactoring is selected at random and removed from the list. Validation is performed in the same manner as when inserting a refactoring for those refactorings that appear after the removed refactoring in the list of refactorings.

When mutating a refactoring in the list of refactorings, a refactoring is first randomly selected. Then, one of three types of mutations is selected for application to the selected refactoring using a user-defined probability distribution: (1) replace the selected refactoring with a new randomly-generated refactoring, (2) replace the selected refactoring with a new randomly-generated refactoring of the same refactoring type (e.g., replace a *MoveField* refactoring with a new randomly generated *MoveField* refactoring), or (3) mutate a parameter of the selected refactoring. An example of a refactoring parameter mutation is changing the target class of a *MoveMethod* refactoring to another randomly chosen class in the model. Validation for all three types of refactoring mutations are performed in the same manner as described previously.

2.3 Objective Functions

Objective functions are a very important component of any search-based algorithm, because they define the metrics upon which solutions are compared that ultimately guides the search process. In the context of determining the quality of lists of refactorings to be applied to test case models in response to metamodel changes, three objective functions that define characteristics of a good solution are: (1) maximize updated metamodel coverage, (2) minimize model elements that do not conform to the updated metamodel, and (3) minimize the number of refactorings used to refactor the existing test case models.

Maximizing the coverage of the updated metamodel is imperative because the sole purpose of test case models is to ensure that the model transformation mechanisms are robust. Minimizing the number of invalid test case model elements due to metamodel changes ensures that the test case models themselves are free of defects in order to properly assess the quality of the model transformation mechanism being tested. Finally, minimizing the number of refactorings used to refactor the test case models reduces the amount of effort required to update the expected output for the test case model transformations.

Metamodel Coverage. Since UML metamodels are utilized in the experimentation described in this paper, UML metamodel coverage is described here; however, note that different methods of calculating metamodel coverage may exist for different metamodel types. The method used to derive UML metamodel coverage was first introduced in [4]. This method begins by a priori performing partition analysis in which the types of coverage criteria taken into consideration for a given problem are chosen. For metamodel coverage, an adaptation of the same three coverage criteria from [4] are used. These criteria are association-end multiplicities (AEM), class attributes (CA), and generalizations (GN). AEM refers to the types of multiplicities used in associations included in a metamodel such as *0..1*, *1..1*, or *1..N*. CA refers to the types of class attributes included in a metamodel such as *integer*, *string*, or *boolean*. Since the metamodels used in the empirical tests in this paper support class operations in addition to attributes, class method return types are included in CA. GN refers to the coverage of classes that belong to each of the following categories: *superclass*, *subclass*, *both superclass and subclass*, and *neither superclass nor subclass*.

Each coverage criterion must be partitioned into logical partitions that, when unioned together, represent all the value types each criterion could take on. These partitions are then assigned representative values to represent each coverage criterion partition. For example, if a metamodel allows for classes to have an *integer* attribute, then the *integer* class attribute element is included in the CA coverage criterion. The values an *integer* class attribute can take on can be split into partitions whose representative values are $<$*-1*, *-1*, *0*, *1*, and $>$*1*, for example. An example of partition analysis and a subset of the coverage items generated from its representative values are shown in Table 2 and Table 3, respectively.

After representative values are defined, a set of coverage items for the updated metamodel is created. In our adaptation of the coverage item set creation method

Table 2. Partition analysis example showing associated representative values for given coverage criteria

Coverage Criteria	Representative Values
CA: *boolean*	*true, false*
CA: *integer*	*<-1, -1, 0, 1, >1*
CA: *float*	*<-1.0, -1.0, 0.0, 1.0, >1.0*
CA: *string*	*Null, '', 'something'*
AEM: *1..1*	*1*
AEM: *1..N*	*1, N*
GN	*sub, super, both, neither*

Table 3. Subset of coverage items created from representative values found in Table 2

Coverage Items	
CA: *-1*	AEM: *N*
CA: *'something'*	GN: *super*
AEM: *1*	AEM: *N*
AEM: *1*	GN: *neither*
CA: *false*	CA: *>1.0*
CA: *Null*	AEM: *1*

introduced in [4], this is done by calculating all possible 2-tuple combinations of representative values from all partitions of all coverage criteria types that are included in the updated metamodel. The only exception to this are coverage items containing two different GN representative values, because they would be impossible to satisfy. This is done to ensure the robustness of the model transformation mechanism for all possible valid combinations of representative values. The metamodel coverage objective value for given test case models and updated metamodel is determined by calculating the percentage of metamodel coverage items the test case models satisfy. For example, if a given updated metamodel included associations with end multiplicities of $1..1 \rightarrow 1..N$, then the derived coverage items would include associations with end-multiplicities of $1 \rightarrow 1$ and $1 \rightarrow N$. Additionally, if a given updated metamodel also included *boolean* class attributes, then the additional coverage items would include classes with a *boolean* attribute and association end multiplicity of *true* and *1*, *false* and *1*, *true* and *N*, and *false* and *N*, respectively. For a more in-depth example of a model and the coverage items it would satisfy, refer to Figure 2 and Table 4, respectively.

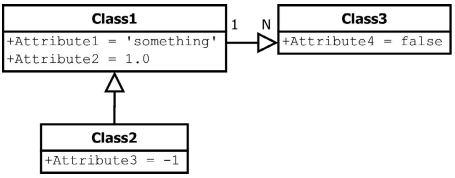

Fig. 2. Example test case model

Table 4. Coverage items satisfied by the example shown in Figure 2

Coverage Items	
CA: *'something'*	CA: *1.0*
CA: *'something'*	AEM: *1*
CA: *1.0*	AEM: *1*
CA: *'something'*	GN: *super*
CA: *1.0*	GN: *super*
AEM: *1*	GN: *super*
CA: *-1*	GN: *sub*
CA: *false*	AEM: *N*
AEM: *1*	AEM: *N*
AEM: *N*	GN: *neither*
CA: *false*	GN: *neither*

Metamodel Conformity. Unlike the bacteriological approach used to automatically generate test case models from scratch in [4], the proposed approach is initialized with test case models that were created to conform to a metamodel that may contain metamodel elements that are not compatible with the updated metamodel. Because of this, there may exist test case model elements that do not conform to the updated metamodel, and if so, should be removed or modified to improve the validity of the test case models by reducing the number of invalid model elements. Calculating the metamodel conformity objective value of given test case models and updated metamodel is done by summing up the number of test case model elements from all test case models that do not conform to the updated metamodel. For example, say Metamodel v1.0 includes *integer* class attributes while Metamodel v2.0 does not. All *integer* model elements from the test case models for Metamodel v1.0 are invalid in Metamodel v2.0, so they need to be removed or modified to a valid class attribute type to improve the validity of the test case models themselves.

Number of Refactorings. While automatically generating test case models in an attempt to maximize metamodel coverage has been previously explored and improving metamodel conformity of test case models by itself can be accomplished trivially by removing or modifying nonconforming test case model elements, performing these tasks by finding a minimal number of refactorings to apply to existing test case models has not yet been explored to our knowledge. By minimizing the number of refactorings required to update existing test case models for an updated metamodel, the task of updating expected test case model transformation output is simplified. The challenge of finding a minimal set of refactorings to apply to test case models to maximize metamodel coverage and minimize the number of nonconforming test case model elements stems from the fact that there are a multitude of different refactoring sequences that can be applied to achieve the same resulting test case models. Calculating the number of refactorings is done by summing up the number of refactorings in the lists of refactorings.

2.4 Search-Based Approach

Simulated Annealing (SA). SA is a local search heuristic inspired by the concept of annealing in metallurgy where metal is heated, raising its energy and relieving it of defects due to its ability to move around more easily. As its temperature drops, the metal's energy drops and eventually it settles in a more stable state and becomes rigid. This technique is replicated in SA by initializing a temperature variable with a "high temperature" value and slowly decreasing the temperature for a set number of iterations by multiplying it by a value α every iteration, where $0 < \alpha < 1$. During each iteration, a mutation operator is applied to a copy of the resulting solution from the previous iteration. If the mutated solution has the same or better fitness than the previous one, it is kept and used for the next iteration. If the mutated solution has a worse fitness, a probability of keeping the mutated solution and using it in the next iteration is calculated

using an acceptance probability function. The acceptance probability function takes as input the difference in fitness of the two solutions as well as the current temperature value and outputs the acceptance probability such that smaller differences in solution fitness and higher temperature values will yield higher acceptance probabilities. In effect, this means that for each passing iteration, the probability of keeping a mutated solution with worse fitness decreases, resulting in a search policy that, in general, transitions from an explorative policy to an exploitative policy. The initial lenience towards accepting solutions with worse fitness values is what allows simulated annealing to escape local minima/maxima.

Multi-Objective Simulated Annealing (MOSA). Traditional SA is not suitable for the automatic test case model generation as described previously because a solution's fitness consists of three separate objective functions and SA cannot directly compare solutions based on multiple criteria. Furthermore, even if SA had the ability to determine relative solution fitness, there would still be the problem of quantifying the fitness disparity between solutions as a scalar value for use in the acceptance probability function. MOSA overcomes these problems. When comparing the relative fitness of solutions, MOSA utilizes the idea of Pareto optimality using dominance as a basis for comparison. Solution A is said to dominate solution B if: (1) every objective value for solution A is the same or better than the corresponding objective value for solution B, and (2) solution A has at least one objective value that is strictly better than the corresponding objective value of solution B. If solution A does not dominate solution B and solution B does not dominate solution A, then these solutions are said to belong to the same non-dominating front. In MOSA, the mutated solution will be kept and used for the next iteration if it dominates or is in the same non-dominating front as the solution from the previous iteration. To determine the probability that the mutated solution dominated by the solution from the previous iteration will be kept and used for the next iteration of MOSA, there are a number of possible acceptance probability functions that can be utilized. Since previous work has noted that the average cost criteria yields good performance [5], we have utilized it. The average cost criteria simply takes the average of the differences of each objective value between two solutions, i and j, over all objectives D, as shown in Equation 1. The final acceptance probability function used in MOSA is shown in Equation 2.

$$c(i,j) = \frac{\sum_{k=1}^{|D|}(c_k(j) - c_k(i))}{|D|} \quad (1) \qquad AcceptProb(i,j,temp) = e^{\frac{-abs(c(i,j))}{temp}} \quad (2)$$

MOSA Adaptation for Generating Test Case Models. When using the number of refactorings fitness criterion along with mutations that add, modify, or remove refactorings in MOSA, a slight modification of the definition of dominance is required in order to obtain quality results. The problem with using the

traditional definition of dominance in this case is that "remove refactoring" mutations will always generate a solution that is at least in the same non-dominated front as the non-mutated solution because it utilizes less refactorings, thus making it strictly better in at least one objective. In MOSA, this means that the non-mutated solution will always be discarded in favor of the mutated solution that it will use in the following iteration. The problem with this is that the probability of an add refactoring or modify refactoring mutation yielding a mutated solution that is in the same non-dominated front or better is much less than that of a mutation removing a refactoring (100%). This is because the only way an add or modify refactoring mutation could at least be in the same non-dominated front is if it satisfied a previously unsatisfied metamodel coverage item, removed an invalid model element, or modified an invalid model element to make it valid. As a result, solutions tend to gravitate towards solutions with less refactorings that eventually results in solutions with the least possible number of refactorings, one refactoring per each test case model. This was found to be the case in experiments executed with the traditional dominance implementation. The problem is alleviated by modifying how dominance is determined in MOSA such that a mutated solution with less refactorings and less metamodel coverage or more invalid model elements than the non-mutated solution is considered to be dominated by the non-mutated solution. In other words, MOSA will only transition from the non-mutated solution from the previous iteration to the new mutated solution (using the "remove refactoring" mutation) with 100% probability if the mutated solution dominates the non-mutated solution. If the mutated solution has less refactorings but also less metamodel coverage or more invalid model elements, then it will only be accepted and used for the next iteration given the probability calculated by the acceptance probability function.

The second problem to overcome is how to use the metamodel coverage, number of invalid model elements, and number of refactoring values in the acceptance probability function in a meaningful way. As they are, these three values take on values in different scales: metamodel coverage takes on values between 0% and 100% (0.0 and 1.0), number of invalid model elements takes on values between 0 and the initial number of invalid model elements before MOSA begins, and the number of refactorings takes on the value of any nonnegative integer. In order to make the average of differences between fitness criteria values meaningful, normalization is performed. Metamodel coverage does not require any normalization as its values already lie between 0.0 and 1.0 and thus all differences between metamodel coverage values will as well. The only operation necessary is to take the absolute value of the difference to ensure it is positive as shown in Equation 3. To normalize the difference between numbers of invalid model elements, simply take the absolute value of the difference between the number of invalid model elements values and divide by the number of invalid model elements from the initial test case models as shown in Equation 4.

$$CovDiff = abs(Cov(i) - Cov(j)) \quad (3) \qquad InvDiff = \frac{abs(Inv(i) - Inv(j))}{Inv_0} \quad (4)$$

To normalize the difference in number of refactorings, the maximum number of refactorings should be used as a divisor. Since there is theoretically no upper bound to the possible number of refactorings that the lists of refactorings could have, a reasonable estimate is required. For this estimate, the sum of the initial number of unsatisfied coverage items and the number of invalid model elements of the starting test case models is used because it assumes that each coverage item and invalid model element will take one refactoring to satisfy and remove, respectively. As shown in Equation 5, the normalization of the difference in number of refactorings is calculated by taking the absolute value of the difference in number of refactorings divided by the sum of the initial number of unsatisfied coverage items and the number of invalid model elements of the starting test case models.

$$NumRefDiff = \frac{abs(NumRef(i) - NumRef(j))}{UnsatCovItems_0 + Inv_0} \tag{5}$$

2.5 Implementation

Before using MOSA to generate the lists of refactorings, a maximum model size must be declared to ensure a balance between the size of the test cases and the number of test cases is maintained. As explained in [4], smaller test cases allow for easier understanding and diagnosis when an error arises while the number of test cases should be reasonable in order to maintain an acceptable execution time and amount of effort for defining an oracle function.

After the maximum model size is declared, the automatic test case model generation begins. The algorithm iterates through all test case models once. For each test case model, its corresponding list of refactorings is initialized with one randomly-generated refactoring before the adapted MOSA algorithm is executed. After the algorithm has iterated over every test case model, the final lists of refactorings for each test case model are output along with the resulting test case models yielded from the application of the refactorings. The pseudocode for this algorithm is shown in Algorithm 1. It is important to note that although search is done for refactorings at the test case model level, the objective functions are executed on the overall running solution of the entire set of updated test case models at any given iteration. This means that, for example, if the space of refactoring lists for a particular test case model is being searched and a mutation is performed that covers a new coverage item for that test case model, but a list of refactorings for another test case model from a previous iteration already covered that particular coverage item, then there is no increase in the metamodel coverage objective function. The value yielded from the metamodel coverage objective function will only increase if a coverage item is covered that has not already been covered by any other test case model with their refactorings in the overall solution.

Algorithm 1. Pseudocode for adapted MOSA for generating test case models

```
function MOSA(testCaseModels, maxModelSize, initialTemperature, α)
    ListOfRefactorings.setMaxModelSize(maxModelSize)
    solution ← list()
    for testCaseModel in testCaseModels do
        refactorings ← ListOfRefactorings(testCaseModel)
        temp ← initialTemperature
        for iteration = 1 → maxIterations do
            newRefactorings ← copy(refactorings)
            newRefactorings.mutate()
            if newRefactorings.dominates(refactorings) then
                refactorings ← newRefactorings
            else if u[0.0,1.0] < AcceptProb(refactorings,newRefactorings,temp) then
                refactorings ← newRefactorings
            temp ← temp × α
        solution.push(refactorings)
    return listsOfRefactorings
```

3 Experimentation

3.1 Experimental Setting

To test the effectiveness of the proposed approach, experiments were carried out to evolve test case models for the UML 2.0 metamodel. In the implementation used, the UML 2.0 metamodel generated 857 coverage items that needed to be satisfied in order to obtain 100% metamodel coverage. To discover if initializing the test case models with those of a previous metamodel version was beneficial, experiments were done starting from a set of test case models that conform to UML 1.4 as well as a set of new test case models. Each test case model in the set of UML 1.4 test case models consists of between 17 and 23 model elements that collectively satisfy 46.58% of the UML 2.0 metamodel coverage items and have 60 model elements that are invalid with respect to the UML 2.0 metamodel, while each test case model in the set of new test case models consists of only five class model elements, collectively satisfy 0% of the UML 2.0 metamodel coverage items, and have no invalid model elements. Both sets are comprised of 20 test case models each.

To justify the multi-objective approach proposed in this paper, the same experiments were carried out using an SA approach utilizing only metamodel coverage like in previous works [4]. All experiments were run 30 times in order to establish statistical significance. For each of the 20 test case models, 10,000 iterations of SA were performed with a starting temperature of 0.0003 and an alpha value of 0.99965. The starting temperature and alpha values were chosen because they yielded the best results in empirical preliminary tests for both SA and MOSA. All probability distributions used by the search process (e.g., to determine the type of mutation to execute or refactoring to generate) were such that each discrete possibility had equal chance of being selected.

3.2 Results

The complete results from all four experiment configurations can be found in Table 5. The SA approaches outperformed the corresponding MOSA approaches in the metamodel coverage objective as shown in Figure 3 while, however, using a far greater number of refactorings as shown in Figure 5. Figure 4 shows that the MOSA experiment that started with the UML 1.4 test case models removed all 60 test case model elements every run while the corresponding SA experiment removed less than half of the invalid test case model elements on average. All differences in results were determined to be statistically significant employing a two-tailed t-test with $\alpha = 0.05$.

Table 5. Empirical results with standard deviations in parentheses

	From Scratch		From Existing Models	
	SA	**MOSA**	**SA**	**MOSA**
Coverage	83.82% (0.05%)	63.36% (0.04%)	96.20% (<0.01%)	91.70% (0.01%)
Invalid	-	-	35.47 (4.03)	0.00 (0.00)
Num. Ref.	1185.87 (176.69)	315.17 (18.08)	726.87 (34.15)	348.90 (13.60)

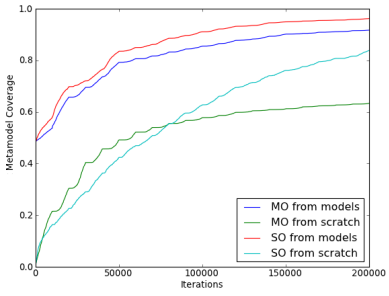

Fig. 3. Metamodel coverage versus iterations

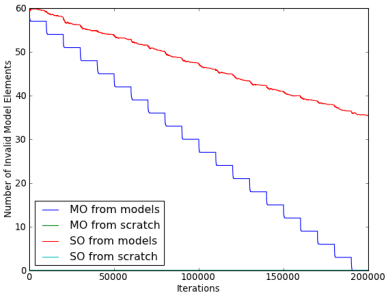

Fig. 4. Invalid model elements versus iterations

3.3 Discussion

With respect to the metamodel coverage objective, it is intuitive that the SA approaches would outperform the MOSA approaches, albeit by a relatively small margin when starting from existing test case models, because the MOSA approaches must balance conflicting objectives while the SA approaches do not. As a result, the lists of refactorings yielded from the MOSA approaches are more effective in terms of metamodel coverage per refactoring than the ones yielded from SA. Combined with the fact that the total number of refactorings yielded by the MOSA approaches are drastically less than those yielded by the SA approaches, this means that the effort required to implement the changes

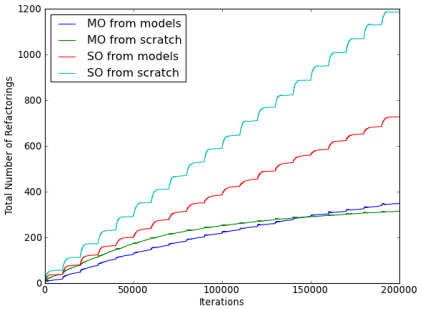

Fig. 5. Refactorings versus iterations

to expected output is less and overall more effective using the MOSA approach under the assumption that an increase in refactorings made to test case models increases the amount of effort required to update the test case expected output.

Furthermore, the results show that the approaches that start with existing test case models of a previous metamodel version outperform the same approaches that generate completely new models. This also helps reduce the effort required to update the expected test case output because portions of the expected output for the existing test cases will not need to be modified. Furthermore, if a user is already familiar with the previous test case models that were initially used as a basis for the new test case models, that knowledge can be leveraged to further decrease the amount of effort required to update expected output.

4 Related Work

Fleurey et al. [4,7] and Steel et al. [8] discuss the reasons why testing model transformations is distinct from testing traditional implementations: the input data are models that are complex in comparison to simple data types. Both papers describe how to generate test data in MDE by adapting existing techniques, including functional criteria [2] and bacteriologic approaches [3]. Lin et al. [9] propose a testing framework for model transformation built on their modeling tools and transformation engine that offers a support tool for test case construction, test execution, and test comparison; however, the test case models are manually developed in this work.

Some other approaches are specific to test case generation for graph transformation mechanisms. Küster [10] addresses the problem of model transformation validation in a way that is very specific to graph transformation by focusing on the verification of transformation rules with respect to termination and confluence. This work is concerned with the verification of transformation properties rather than the validation (testing) of their correctness. Darabos et al. [11] investigate the testing of graph transformations by considering graph transformation

rules as the transformation specification and propose to generate test data from this specification. Darabos et al. propose several faulty models that can occur when performing pattern matching as well as a test case generation technique that targets those particular faults. Compared to the multiobjective search-based approach proposed in this paper, Darabos' work is specific to graph-based transformation testing. Mottu et al. [1] describe six different oracle functions to evaluate the correctness of an output model. In [12], the authors suggest manually determining the expected transformation outcome and comparing it with the actual transformation outcome using a simple graph-comparison algorithm.

The multi-objective search-based approach proposed in this paper is inspired by contributions in the domain of Search-Based Software Engineering (SBSE) [13]. SBSE uses search-based approaches to solve optimization problems in software engineering, and once a software engineering task is framed as a search problem, many search algorithms can be applied to solve that problem. These search-based approaches are also used to solve problems in software testing [14,15,12]. The general idea behind the proposed approach is that possible test case model refactorings define a search space and multiple conflicting test case model criteria are integrated into multiple objective functions. These components guide the search approach in an attempt to find an optimal set of test case model refactorings that yields a set of adequate updated test case models.

Although the problem of generating test cases at the code level is well-studied, there are few works that generate test cases at the model level to test transformation mechanisms. To our knowledge, there is currently no other work that utilizes existing test case models of a previous metamodel to generate test case models for an updated metamodel. Furthermore, this is the first adaptation of heuristic search algorithms to take into consideration multiple objectives when generating source models (test cases) similar to the data that will be transformed.

5 Conclusion and Future Work

Empirical results show that MOSA can automatically generate quality test case models from existing test case models in response to metamodel changes. The new test case models are generated with minimal refactorings so the effort required to update expected test case model transformation output is reduced. While SA is able to achieve slightly better overall metamodel coverage, the number of refactorings, and thus required effort, is substantially greater. Furthermore, the MOSA approach is able to reliably remove test case model elements that become invalid due to metamodel changes.

To generalize our proposed approach and ensure its robustness, we plan to extend our validation to other metamodels such as Petri nets and relational schema. Furthermore, comparative studies will be performed between different multiobjective metaheuristic algorithms as well as between processing all test case models at once to yield an overall single list of refactorings and the proposed method of processing each test case model one at a time.

References

1. Mottu, J., Baudry, B., Le Traon, Y.: Model Transformation Testing: Oracle Issue. In: IEEE International Conference on Software Testing Verification and Validation Workshop, ICSTW 2008, pp. 105–112 (2008)
2. Brottier, E., Fleurey, F., Steel, J., Baudry, B., le Traon, Y.: Metamodel-based Test Generation for Model Transformations: An Algorithm and a Tool. In: 17th International Symposium on Software Reliability Engineering, ISSRE 2006, pp. 85–94 (2006)
3. Baudry, B., Fleurey, F., Jezequel, J.M., Traon, Y.L.: Automatic Test Cases Optimization Using a Bacteriological Adaptation Model: Application to.NET Components. In: Proceedings of ASE 2002 (Automated Software Engineering), Edinburgh (2002)
4. Fleurey, F., Steel, J., Baudry, B.: Validation in Model-Driven Engineering: Testing Model Transformations. In: Proceedings of First International Workshop on Model, Design and Validation, pp. 29–40 (2004)
5. Nam, D., Park, C.H.: Multiobjective Simulated Annealing: A Comparative Study to Evolutionary Algorithms. International Journal of Fuzzy Systems 2(2), 87–97 (2000)
6. Brosch, P., Egly, U., Gabmeyer, S., Kappel, G., Seidl, M., Tompits, H., Widl, M., Wimmer, M.: Towards Scenario-Based Testing of UML Diagrams. In: Brucker, A.D., Julliand, J. (eds.) TAP 2012. LNCS, vol. 7305, pp. 149–155. Springer, Heidelberg (2012)
7. Fleurey, F., Baudry, B., Muller, P.A., Traon, Y.: Qualifying Input Test Data for Model Transformations. Software & Systems Modeling 8(2), 185–203 (2009)
8. Steel, J., Lawley, M.: Model-based Test Driven Development of the Tefkat Model-Transformation Engine. In: 15th International Symposium on Software Reliability Engineering, ISSRE 2004, pp. 151–160 (2004)
9. Lin, Y., Zhang, J., Gray, J.: A Testing Framework for Model Transformations. In: Research and Practice in Software Engineering - Model-Driven Software Development, pp. 219–236. Springer (2005)
10. Küster, J.M., Abd-El-Razik, M.: Validation of Model Transformations – First Experiences using a White Box Approach. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 193–204. Springer, Heidelberg (2007)
11. Darabos, A., Pataricza, A., Varr, D.: Towards Testing the Implementation of Graph Transformations. In: Proceedings of the 5th International Workshop on Graph Transformations and Visual Modeling Techniques, pp. 69–80. Elsevier (2006)
12. McMinn, P.: Search-based Software Test Data Generation: A Survey: Research Articles. Softw. Test. Verif. Reliab. 14(2), 105–156 (2004)
13. Harman, M., Mansouri, S.A., Zhang, Y.: Search-based Software Engineering: Trends, techniques and applications. ACM Comput. Surv. 45(1), 11:1–11:61 (2012)
14. Baresel, A., Binkley, D., Harman, M., Korel, B.: Evolutionary Testing in the Presence of Loop-Assigned Flags: A Testability Transformation Approach. In: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, pp. 108–118. ACM, New York (2004)
15. Baresel, A., Sthamer, H., Schmidt, M.: Fitness Function Design To Improve Evolutionary Structural Testing. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2002, pp. 1329–1336. Morgan Kaufmann Publishers Inc., San Francisco (2002)