A multi-start constructive procedure for test case generation in model transformation testing

Alejandro Aunión*[1], Javier Yuste[1], Eduardo G. Pardo[1], and Francisco Chicano[2]

[1] GRAFO Research, Departamento de Informática y Estadística, Universidad Rey Juan Carlos, C/ Tulipán s/n, Móstoles, 28933, Spain
{alejandro.aunion,javier.yuste,eduardo.pardo}@urjc.es
[2] ITIS Software, Departamento de Languajes y Ciencias de la Computación, Universidad de Málaga, Av. de Cervantes, 2, Málaga, 29016, Spain
chicano@uma.es

Abstract. Ensuring the correctness of model transformations is a critical concern in Model-Driven Engineering, as it directly conditions the reliability of the software development activities that depend on them. To validate the correctness of model transformation processes, it is necessary to have a representative and diverse set of input testing models. However, manually constructing such a set is costly and error-prone. In this work, we propose a multi-start constructive procedure for the automatic generation of test model sets. The method starts by building a trivial feasible model and then constructs new ones by applying different operators. By identifying the feasible region of the search space, the method is able to efficiently generate feasible models. The proposed method is evaluated in comparison to the state of the art on three families of instances of varying complexity, considering both unconstrained and constrained settings. Results show that the constructive procedure effectively constructs feasible high-quality models in low computing times, even for instances for which the state-the-art method is not able to find feasible solutions at all within the time limit.

Keywords: Software quality optimization · Model-Driven Engineering · Model transformation testing · Heuristics

1 Introduction

Software systems are subject to continuous evolution and must satisfy ever-changing requirements. Moreover, this continuous evolution leads to a deterioration of their quality. To deal with the complexity of the system and ease understanding, models arise as critical tools to reduce that burden. A model is an abstraction of the system that captures its structure or behavior in a precise and manageable way [6]. Model-Driven Engineering (MDE) is a discipline that focuses on using models as executable and transformable artifacts

* Corresponding author

that drive automation throughout the entire life cycle of software development [7]. In the context of Software Engineering (SE), there are multiple situations where model transformations take place, generating models from user bug reports (text-to-model transformations [1]), generating models from other models (model-to-model transformation [5]), or generating code from models (model-to-code transformations [10]). Ensuring the correctness of the model transformation activities is key to the success of MDE.

Unfortunately, manually constructing adequate test models is a costly and error-prone task, even for domain experts [2], and poorly sets of test models can leave entire parts of the transformation unexercised, drastically reducing the fault detection capability. Thus, automatically deriving diverse sets of high-quality models to test model transformation processes is a promising step within the area of MDE [11]. We will denote this process as "Model Transformation Testing" (MTT). In this sense, the automatic generation of test input models can be tackled as an optimization problem within the context of Search-Based Software Engineering, a research area focused on reformulating SE problems as optimization problems and designing methods to solve them [8,12]. Particularly, this problem arises in the area of Search-Based Model-Driven Engineering [4].

In this work, we propose a multi-start constructive procedure to address the problem of model test case generation for MTT. First, a categorization of constraints is introduced. Then, we propose a method that starts by building a trivial feasible model and then constructs new ones by applying different operators. By identifying the feasible region of the search space, the method is able to efficiently generate feasible models in low computing times. The performance of the method is validated by comparing it to a state-of-the-art Ant Colony Optimization approach on three families of instances of varying complexity. Results show that the proposed method effectively constructs feasible models even for instances for which the state-the-art algorithm is not able to find feasible solutions.

The rest of the paper is organized as follows. In Section 2, we formally introduce the problem of automatic test model generation. In Section 3, we present the proposed multi-start constructive procedure. In Section 4, we show the experimental evaluation of the method proposed on three families of instances. Finally, in Section 5, we outline the main findings and propose directions for future work.

2 Problem definition

The problem tackled in this paper consists of generating test cases for MTT. Due to its abstract nature, a proper understanding requires the identification of its fundamental components: an instance, a solution, and the objective function. An instance is represented as a metamodel $\mathcal{M}$ that strictly defines what classes of elements may exist and how they may interact with each other. In particular, the metamodels of the problem are represented as class diagrams as defined in the Unified Modeling Language (UML) specification [3]. Fig. 1 shows an exam-

ple metamodel named `Families`. As it can be observed, the schema defines the existence of four classes (`Family`, `Member`, `Neighborhood` and `School`) and seven relationships connecting them. Relationships can be of different types, following the UML standard. In the `Families` example, composition relationships appear (represented by a filled diamond next to the containing class). These indicate a hierarchical dependency: members exist as part of a family. Conversely, reference relationships (represented by simple arrows) establish non-hierarchical associations between independent elements. For instance, the link between a `Member` and a `School` denotes a logical connection rather than containment, meaning the school exists independently of the members who attend it. Nevertheless, it is worth noting that other types of relationships may appear. Furthermore, the metamodel imposes numerical constraints known as cardinalities. For example, the figure indicates that every family must have exactly one father and one mother (cardinality `[1..1]`). In order to be considered feasible, a model must respect the cardinalities specified by the metamodel.

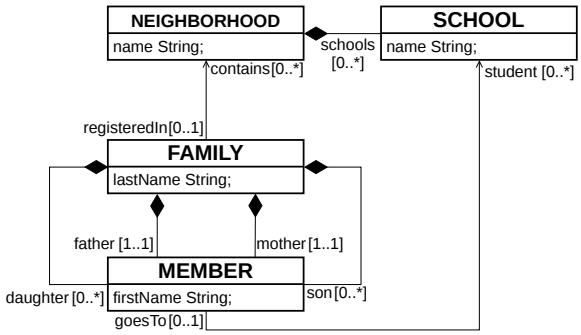
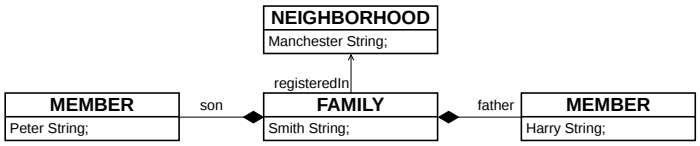
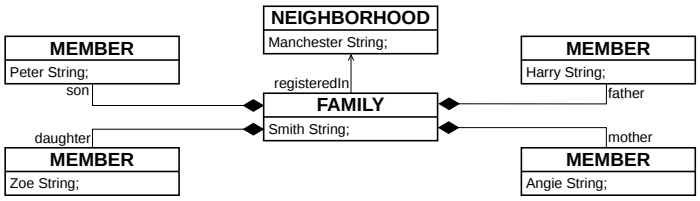


Fig. 1: UML representation of the `Families` metamodel.

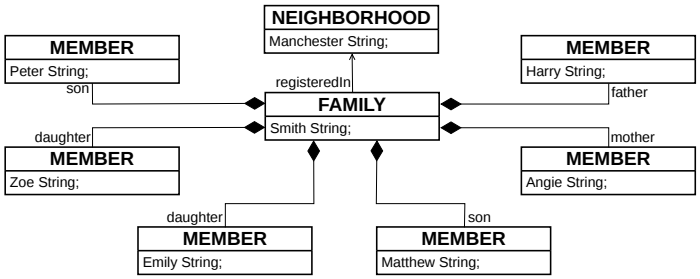
A solution to the problem consists of a set of models $M = \{m_1, m_2, \ldots, m_n\}$, where the number n of models to build is specified by the instance and each model $m_i \in M$ must be feasible. That is, it must be valid according to the domain specified by the metamodel. Considering that Fig. 1 represents a metamodel of a family, Fig. 2 presents three different models generated for the `Families` metamodel. Fig. 2a shows a model composed of one `Family`, one `Neighborhood`, and two `Member` classes. As it can be observed, the family contains one father and one son. Since this configuration lacks a mother, it violates one of the cardinalities imposed by the metamodel (each family must contain one mother), rendering the model infeasible. Fig. 2b presents a family that contains two additional members: a mother and a daughter. This model is feasible according to the aforementioned metamodel. Similarly, Fig. 2c also presents a feasible model, considering the satisfaction of the constrains imposed by the metamodel.



(a) Model m_1.



(b) Model m_2.



(c) Model m_3.

Fig. 2: Example of three models generated for the `Families` metamodel.

Metamodels might capture only abstract and general information. However, in this context it is common that an instance, apart from the constraints imposed by the design of the metamodel, contains extra specific constraints expressed verbally. For example, an instance for the problem might contain the `Families` metamodel and the following additional constraints: (1) the family must have at least six members; (2) the number of sons must be equal to the number of daughters; and (3) the family `lastName` must be in uppercase. In that case, the model shown in Fig. 2b would not be feasible, since it does not satisfy the constraint requiring a minimum of six members. In contrast, Fig. 2c presents a valid model that contains two sons and two daughters, and also six members, satisfying all the additional constraints. Finally, the instance specifies the maximum size of the models (i.e., the maximum number of classes that a model can have).

To evaluate the quality of a solution, the literature proposes considering the diversity of the testing models. The higher the diversity of the set, the greater its capability to reveal faults in the model transformation process. Thus, the goal is to maximize the diversity of the solution. In this context, the diversity between

two models measures their structural differences: the number of classes or relations that are present in one model but not in the other. For example, considering models m_2 and m_3, shown in Fig. 2b and Fig. 2c, respectively, and assuming that both are feasible, their structural difference lies in exactly two elements: m_3 contains an additional son and an additional daughter[3]. Consequently, the distance between them is equal to 2. Finally, the diversity of a solution M is the sum of the diversity between each pair of models in M.

3 Algorithmic proposal

To efficiently generate feasible models, we propose a multi-start constructive procedure. The method is based on a separation of constraints into four different categories: (1) constraints related to minimum cardinalities (e.g., at least six members); (2) constraints related to maximum cardinalities (e.g., a family cannot have more than one father); (3) constraints related to a balance between two or more elements (e.g., same number of sons and daughters); (4) and syntactic constraints, related to the names of the attributes of classes (e.g., the last name of a person must start with an uppercase letter). Note that the constraints of the fourth category do not affect the structure of the model. Therefore, this can be considered in a post-processing stage, where the names of the attributes present in the model are given appropriate values. For this reason, they are out of scope for the proposed method.

Algorithm 1 shows the pseudocode of the proposed procedure. As it can be observed, it receives three input parameters: the metamodel $\mathcal{M}$; a maximum number of models to include in the solution (max_models); and a maximum number of iterations to perform (max_iterations). The procedure starts building a trivial model (line 4). That is, a model that does not violate any of the constraints in the first category, related to minimum cardinalities. It should be noted that the trivial model is the smallest feasible model as it contains the minimum number of classes and relations required by the metamodel. The method then proceeds adding components to the trivial model (lines 7-17). At each step, the method tries to obtain a valid model by either adding a class or a relation between classes (these operators are detailed later in this section) (lines 9, 11). After each addition (line 11), the method tries to repair the model (line 12) if any constraint of the third category has been violated (that is, constraints related to a balance between two or more elements). For example, suppose that a constraint states that the model must have the same number of sons and daughters. In that case, after adding a son to the model, the repair process would add a daughter to maintain the balance demanded by the aforementioned constraint. Then, the constraints of the second category, related to maximum cardinalities, are checked (line 13). If the model is feasible, it becomes the new incumbent model (lines 14-15). The process continues by adding classes and relations to

[3] Note that attributes such as the name of the member in this case are not considered when evaluating the structural differences between models.

this obtained model. Otherwise, if the obtained model is infeasible, the procedure steps back and tries to apply any of the other operators. The iteration ends when no operator leads to a new feasible model. At that point, the current model is added to the solution (line 18) and the procedure restarts the search from the trivial initial model (lines 5-6). This multi-start nature gives the method the ability to explore different regions of the search space. The method stops when a maximum number of iterations is reached or when the solution contains the desired number of models specified by the instance (line 5). At that moment, the solution is returned (line 20).

Algorithm 1 Pseudocode of the Multi-start Constructive Search procedure

```
 1: procedure Multi-start Constructive Search(M, max_models, max_iter)
 2:     iter ← 0
 3:     sol ← ∅
 4:     x_t ← GenerateTrivialModel(M)
 5:     while iter ≤ max_iter ∧ |sol| < max_models do
 6:         x ← x_t
 7:         O ← GetPossibleOperators(x, M)
 8:         while |O| > 0 do
 9:             op ← SelectOperator(O)
10:             O ← O \ op
11:             x' ← Apply(x, op)
12:             x'' ← Repair(x')
13:             if IsFeasible(x'', M) then
14:                 x ← x''
15:                 O ← GetPossibleOperators(x, M)
16:             end if
17:         end while
18:         sol ← sol ∪ {x}
19:     end while
20:     return sol
21: end procedure
```

Note that the separation of constraints into different categories leads to an effective exploration of the search space, since many infeasible models (i.e., those that violate either minimum or maximum cardinalities) are not considered. The only constraints that might be violated by models obtained during the search are those related to balance, which are immediately restored by a repair mechanism. This identification of the feasible region in the search space drastically reduces the computational effort required to find feasible models.

Three operators are proposed to construct the solution. Fig. 3 illustrates the sequential application of these operators over an example model, showing the evolution from the model m_x to the model m_x'''. The process begins with the operator `AddRootClass`, which adds a root class to the model. A root class is a class that can exist independently, without requiring relations to other classes

in the model. In the `Families` metamodel, both `Family` and `Neighborhood` are considered root classes. As shown in Fig. 3, the operator `AddRootClass(`m_x`, Neighborhood)` is first applied to the initial model m_x, resulting in model m'_x where the class `Neighborhood` has been added.

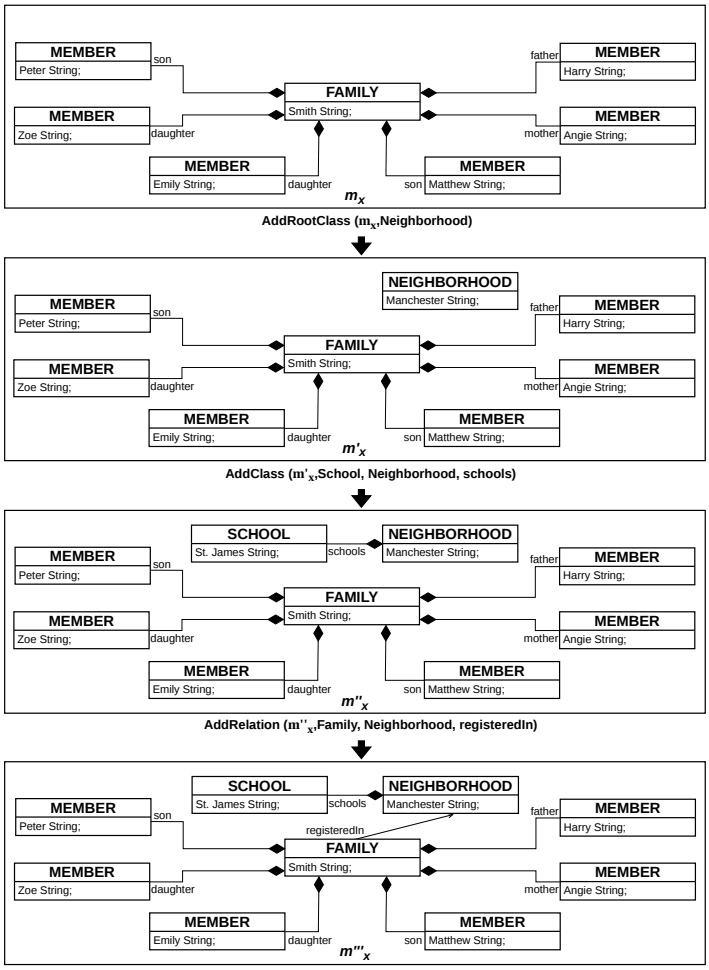


Fig. 3: Application of three different operators (`AddRootClass`, `AddClass`, and `AddRelation`) to the model m_x.

The operator `AddClass` adds a new class to the model, together with a relation connecting it to an existing class in the model. In the example, the operator `AddClass(`m'_x`, School, Neighborhood, schools)` is applied to the intermediate model m'_x. This operation introduces the class `School` and connects it to

`Neighborhood` through the relation `schools`, resulting in model m''_x. Finally, the operator `AddRelation` adds a new relation between two classes that are already present in the model. In Fig. 3, the operator `AddRelation(`m''_x`, Family, Neighborhood, registeredIn)` is applied to model m''_x, creating the relation `registeredIn` between `Family` and `Neighborhood`, resulting in model m'''_x.

Note that the operators considered by the procedure are limited to the classes and relations allowed by the metamodel. In the example of the `Families` metamodel, any relation added by the procedure between a `Member` and a `Family` would be of composition, since the metamodel does not allow any other type of relation between those classes.

4 Experimental results

In this section, we compare the performance of the proposed method with the current state of the art, an ACO algorithm [9]. All experiments were run on the same computing environment, a dedicated server equipped with an AMD EPYC 7282 processor (16 cores) and 32 GB of RAM, running the Ubuntu 20.04.1 LTS operating system. Both the proposed procedure and the state-of-the-art algorithm were implemented in Java, compiled using the Eclipse Temurin JDK 17.0.18 environment. For the ACO method, we used the implementation made publicly available by the authors [9]. Both methods perform the same number of maximum iterations, limited to fifty times the maximum number of models that the solution can contain, specified by each instance (i.e., $50 \cdot \max_sol$). For the comparison, we randomly selected a subset of instances from the benchmark proposed in the literature [9]. In particular, we study three families of instances: `Families`, `Families_Extended`, and `Grafcet`, yielding a total of 45 test cases.

Table 1 presents the experimental results, comparing the performance of the state-of-the-art ACO and the proposed constructive method (C). The table is divided into two main blocks based on the presence of additional constraints to those imposed in the metamodel (All cons.). A "✗" sign indicates that additional constraints are not considered, while a "✓" means that they are included. The values reported in each row represent the average for that group of instances. The bottom row represents the global average. For each group of instances, we report the mean diversity of the solution (Avg. Diversity), the average number of feasible models obtained (Avg. Feasible), the average deviation from the best (Avg. Dev. (%)), the number of instances for which each method achieved the best diversity (# Best), and the average running time in seconds (Avg. CPUt (s)). Note that the optima are not known. Deviation is calculated with respect to the best solution found. For readability, the best result for each metric within a given row is highlighted in bold.

As it can be observed, the proposed method is able to obtain solutions with higher diversity and a greater number of feasible models for most instances. The only exception is `Families`, where the ACO procedure achieves better performance. However, this only occurs when the additional constraints are ignored. When all constraints are considered, the ACO procedure does not generate fea-

Table 1: Comparison of the results obtained by the state-of-the-art ACO and the proposed constructive procedure (C) for different groups of instances.

Instances	All cons.	Avg. Feasible		Avg. Diversity		Avg. Dev. (%)		# Best		Avg. CPUt (s)	
		ACO	C	ACO	C	ACO	C	ACO	C	ACO	C
Families	✗	**200.00**	40.67	**1303.15**	545.81	**10.66%**	55.31%	**10**	5	16.42	**0.80**
Fam. Ext.	✗	163.53	**200.00**	1356.44	**2559.20**	58.12%	**0.00%**	0	**15**	16.24	**0.21**
Grafcet	✗	31.33	**200.00**	2822.37	**5000.52**	67.26%	**0.06%**	1	**14**	18.28	**0.51**
Families	✓	0.00	**12.07**	0.00	**641.75**	100.00%	**0.00%**	0	**15**	34.74	**1.42**
Fam. Ext.	✓	0.00	**62.20**	0.00	**814.86**	100.00%	**0.00%**	0	**15**	69.64	**0.34**
Grafcet	✓	11.80	**42.00**	131.05	**570.01**	40.59%	**16.61%**	**8**	7	75.87	**0.57**
All		67.78	**126.16**	935.50	**1688.69**	62.77%	**12.00%**	19	**71**	38.53	**0.64**

sible models for either `Families` or `Families Extended`. In addition, the proposed procedure achieves the best solution in 71 out of 90 cases in much faster times. The detailed results are presented in Table 2 (no additional constraints considered) and Table 3 (all constraints considered).

The results obtained indicate that, by guaranteeing structural validity at every step, the constructive method avoids wasting computational effort on infeasible models, thereby producing high-quality solutions efficiently. The difference in performance is higher when considering all constraints, since the probability of getting feasible models through unguided construction is lower.

5 Conclusions and future work

In this paper, we addressed the problem of automatically generating test models for model transformation testing. We proposed a categorization of constraints in four different categories. Then, we proposed an efficient constructive procedure that leverages this categorization to explore only feasible regions of the search space. The proposed method was experimentally validated over a set of benchmark instances obtained from the literature. The results show that the proposed approach achieves solutions of better quality and less computing time than the state of the art [9]. The performance of the proposed procedure is particularly superior for instances with many constraints.

Despite these promising results, there is still room for improvement. Future work will focus on incorporating mechanisms to guide the search within the identified region. Moreover, from a practical perspective, generating a very large number of models may be unrealistic in real-world applications. Thus, it would be interesting to study the trade-off between the number of models and the diversity of the solutions in a multi-objective optimization context.

Table 2: Detailed results per method without considering additional constraints in the instances.

Instance	Max. models	Max. size	# Feasible		Diversity		Best		Time (s)	
			ACO	C	ACO	C	ACO	C	ACO	C
Families	50	10	**50**	36	**270.48**	158.67	✓	✗	0.71	**0.31**
		20	**50**	**50**	399.92	**440.80**	✗	✓	1.17	**0.09**
		40	**50**	**50**	528.48	**801.20**	✗	✓	2.39	**0.17**
		80	**50**	**50**	792.08	**1277.32**	✗	✓	4.91	**0.61**
		160	**50**	**50**	1038.00	**1849.08**	✗	✓	10.35	**1.31**
		320	**50**	**50**	1457.36	**2232.04**	✗	✓	20.87	**2.80**
	100	10	**100**	36	**512.16**	158.67	✓	✗	2.11	**0.23**
	150	10	**150**	36	**862.51**	158.67	✓	✗	4.74	**0.38**
	200	10	**200**	36	**1166.22**	158.67	✓	✗	8.60	**0.52**
	250	10	**250**	36	**1290.58**	158.67	✓	✗	13.31	**0.67**
	300	10	**300**	36	**1766.01**	158.67	✓	✗	19.47	**0.81**
	350	10	**350**	36	**1953.51**	158.67	✓	✗	26.35	**0.80**
	400	10	**400**	36	**2307.43**	158.67	✓	✗	34.46	**1.04**
	450	10	**450**	36	**2248.89**	158.67	✓	✗	43.39	**1.04**
	500	10	**500**	36	**2953.68**	158.67	✓	✗	53.47	**1.22**
Fam. Ext.	50	10	34	**50**	258.53	**500.16**	✗	✓	0.62	**0.04**
		20	2	**50**	29.00	**926.20**	✗	✓	1.51	**0.04**
		40	0	**50**	0.00	**1285.08**	✗	✓	3.33	**0.04**
		80	0	**50**	0.00	**1839.36**	✗	✓	6.55	**0.06**
		160	0	**50**	0.00	**2094.72**	✗	✓	13.41	**0.12**
		320	0	**50**	0.00	**1756.96**	✗	✓	28.32	**0.29**
	100	10	51	**100**	253.26	**1042.32**	✗	✓	2.28	**0.03**
	150	10	137	**150**	1155.85	**1607.01**	✗	✓	5.00	**0.07**
	200	10	179	**200**	1530.95	**2162.92**	✗	✓	8.56	**0.11**
	250	10	226	**250**	1866.94	**2733.82**	✗	✓	12.81	**0.17**
	300	10	256	**300**	2018.33	**3329.02**	✗	✓	18.34	**0.25**
	350	10	324	**350**	2735.04	**3892.38**	✗	✓	24.35	**0.33**
	400	10	370	**400**	3149.97	**4462.37**	✗	✓	31.81	**0.43**
	450	10	411	**450**	3435.37	**5062.37**	✗	✓	38.80	**0.54**
	500	10	463	**500**	3913.32	**5693.34**	✗	✓	47.87	**0.66**
Grafcet	50	10	9	**50**	116.67	**569.40**	✗	✓	0.58	**0.02**
		20	6	**50**	180.33	**1263.08**	✗	✓	1.29	**0.03**
		40	17	**50**	1147.65	**2638.48**	✗	✓	3.01	**0.08**
		80	33	**50**	4515.58	**5223.08**	✗	✓	6.45	**0.25**
		160	39	**50**	10170.26	**10590.28**	✗	✓	13.58	**0.84**
		320	43	**50**	**21522.84**	21318.84	✓	✗	31.38	**3.03**
	100	10	16	**100**	219.63	**1175.54**	✗	✓	2.28	**0.04**
	150	10	18	**150**	257.89	**1780.85**	✗	✓	5.37	**0.09**
	200	10	24	**200**	329.58	**2422.78**	✗	✓	9.48	**0.15**
	250	10	31	**250**	444.58	**3058.34**	✗	✓	14.54	**0.23**
	300	10	37	**300**	549.08	**3705.24**	✗	✓	20.77	**0.32**
	350	10	37	**350**	526.32	**4322.09**	✗	✓	28.03	**0.43**
	400	10	53	**400**	774.57	**4989.72**	✗	✓	35.99	**0.56**
	450	10	51	**450**	755.77	**5639.88**	✗	✓	45.62	**0.70**
	500	10	56	**500**	824.79	**6310.20**	✗	✓	55.90	**0.86**

Table 3: Detailed results per method considering all constraints in the instances.

Instance	Max. models	Max. size	# Feasible		Diversity		Best		Time (s)	
			ACO	C	ACO	C	ACO	C	ACO	C
Families	50	10	0	**2**	0.00	**2.00**	✗	✓	4.38	**0.38**
		20	0	**7**	0.00	**32.00**	✗	✓	7.21	**0.56**
		40	0	**17**	0.00	**192.00**	✗	✓	14.25	**2.20**
		80	0	**37**	0.00	**912.00**	✗	✓	28.93	**8.59**
		160	0	**50**	0.00	**2759.28**	✗	✓	58.67	**1.47**
		320	0	**50**	0.00	**5711.04**	✗	✓	122.72	**3.10**
	100	10	0	**2**	0.00	**2.00**	✗	✓	9.00	**0.19**
	150	10	0	**2**	0.00	**2.00**	✗	✓	13.78	**0.28**
	200	10	0	**2**	0.00	**2.00**	✗	✓	19.17	**0.38**
	250	10	0	**2**	0.00	**2.00**	✗	✓	24.48	**0.46**
	300	10	0	**2**	0.00	**2.00**	✗	✓	29.49	**0.56**
	350	10	0	**2**	0.00	**2.00**	✗	✓	37.74	**0.65**
	400	10	0	**2**	0.00	**2.00**	✗	✓	44.33	**0.74**
	450	10	0	**2**	0.00	**2.00**	✗	✓	50.41	**0.83**
	500	10	0	**2**	0.00	**2.00**	✗	✓	56.54	**0.93**
Fam. Ext.	50	10	0	**50**	0.00	**377.04**	✗	✓	1.04	**0.13**
		20	0	**50**	0.00	**858.92**	✗	✓	1.93	**0.02**
		40	0	**50**	0.00	**1651.36**	✗	✓	6.33	**0.04**
		80	0	**50**	0.00	**1469.12**	✗	✓	25.39	**0.05**
		160	0	**50**	0.00	**1388.84**	✗	✓	145.29	**0.11**
		320	0	**50**	0.00	**1485.68**	✗	✓	612.25	**0.42**
	100	10	0	**65**	0.00	**501.32**	✗	✓	3.18	**0.16**
	150	10	0	**67**	0.00	**526.03**	✗	✓	6.80	**0.25**
	200	10	0	**70**	0.00	**551.06**	✗	✓	11.38	**0.32**
	250	10	0	**70**	0.00	**551.06**	✗	✓	18.39	**0.40**
	300	10	0	**71**	0.00	**563.58**	✗	✓	24.01	**0.48**
	350	10	0	**71**	0.00	**563.58**	✗	✓	31.88	**0.55**
	400	10	0	**71**	0.00	**563.58**	✗	✓	41.82	**0.64**
	450	10	0	**73**	0.00	**579.84**	✗	✓	52.04	**0.72**
	500	10	0	**75**	0.00	**591.87**	✗	✓	62.92	**0.80**
Grafcet	50	10	4	**32**	30.00	**112.44**	✗	✓	1.56	**0.15**
		20	2	**50**	36.00	**420.64**	✗	✓	6.55	**0.04**
		40	0	**50**	0.00	**783.28**	✗	✓	20.93	**0.06**
		80	0	**50**	0.00	**1415.76**	✗	✓	57.02	**0.13**
		160	0	**50**	0.00	**2177.88**	✗	✓	178.36	**0.35**
		320	0	**50**	0.00	**2350.72**	✗	✓	610.10	**0.79**
	100	10	8	**37**	74.75	**133.62**	✗	✓	3.92	**0.25**
	150	10	12	**38**	**148.00**	137.79	✓	✗	7.85	**0.39**
	200	10	12	**39**	**152.50**	145.44	✓	✗	12.80	**0.52**
	250	10	25	**39**	**266.48**	145.44	✓	✗	18.48	**0.66**
	300	10	26	**39**	**314.92**	145.44	✓	✗	25.64	**0.78**
	350	10	13	**39**	**167.69**	145.44	✓	✗	33.56	**0.91**
	400	10	22	**39**	**196.27**	145.44	✓	✗	43.19	**1.07**
	450	10	27	**39**	**305.41**	145.44	✓	✗	54.26	**1.17**
	500	10	26	**39**	**273.77**	145.44	✓	✗	63.87	**1.29**

Acknowledgements This research has been supported by grants: PID2024-160226OB-C22, and PID2021-125709OA-C22, funded by MCIN/AEI and "ERDF A way of making Europe"; CIRMA-CM Ref. TEC-2024/COM-404, funded by Comunidad Autónoma de Madrid; TSI-100930-2023-3, "Cátedra AI4DDS: Artificial Intelligence for Data Drive Solutions" (Ref. ID MCA07); "Cátedra de Innovación y Digitalización Empresarial entre Universidad Rey Juan Carlos y Second Episode" (Ref. ID MCA06); and "Cátedra Institucional de Transición Energética" (Ref. ID MCA09). The work is also partially funded by MICIU/AEI/10.13039/501100011033 through grants PLEC2023-010266 (SOFIA), PID2022-142964OA-I00 (EGSVAI), and RED2022-134647-T (AI4Software).

References

1. Aunión, A., Yuste, J., Pardo, E.G.: Multi-objective variable neighborhood descent for the inference of test models from user bug reports in software systems. In: Variable Neighborhood Search, Lecture Notes in Computer Science. vol. 16256, pp. 180–195. Springer Nature Switzerland (2026)
2. Baudry, B., Dinh-Trong, T., Mottu, J.M., Simmonds, D., France, R., Ghosh, S., Fleurey, F., Le Traon, Y.: Model transformation testing challenges. In: ECMDA workshop on Integration of Model Driven Development and Model Driven Testing. (2006)
3. Booch, G., Jacobson, I., Rumbaugh, J., et al.: The unified modeling language. Unix Review **14**(13), 5 (1996)
4. Boussaïd, I., Siarry, P., Ahmed-Nacer, M.: A survey on search-based model-driven engineering. Automated Software Engineering **24**, 233–294 (2017)
5. Brambilla, M., Cabot, J., Wimmer, M.: Model-driven software engineering in practice. Morgan & Claypool Publishers (2017)
6. Dennis, A., Wixom, B., Tegarden, D.: Systems analysis and design: An object-oriented approach with UML. John wiley & sons (2015)
7. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Future of Software Engineering (FOSE'07). pp. 37–54. IEEE (2007)
8. Harman, M., Jones, B.F.: Search-based software engineering. Information and software Technology **43**(14), 833–839 (2001)
9. Karimi, M., Kolahdouz-Rahimi, S., Troya, J.: Ant-colony optimization for automating test model generation in model transformation testing. Journal of Systems and Software **208**, 111882 (2024)
10. Oldevik, J., Neple, T., Grønmo, R., Aagedal, J., Berre, A.J.: Toward standardised model to text transformations. In: Lecture Notes in Computer Science. vol. 16256, pp. 239–253. Springer (2005)
11. Semeráth, O., Farkas, R., Bergmann, G., Varró, D.: Diversity of graph models and graph generators in mutation testing. International Journal on Software Tools for Technology Transfer **22**(1), 57–78 (2020)
12. Yuste, J., Pardo, E.G., Chicano, F.: Search-based software engineering problems. In: Handbook of Heuristics, pp. 1621–1649. Springer (2025)