Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Ant-colony optimization for automating test model generation in model transformation testing☆

Meysam Karimi[a], Shekoufeh Kolahdouz-Rahimi[a,b,*], Javier Troya[c]

[a] MDSE Research Group, Department of Software Engineering, University of Isfahan, Iran
[b] School of Arts, University of Roehampton, London, UK
[c] ITIS Software, Universidad de Málaga, Spain

ARTICLE INFO

Dataset link: https://github.com/MeysamKarimi/MG-ACO

Keywords:
Model-Driven Engineering
Model transformation testing
Automated model generation
Ant colony optimization

ABSTRACT

In model transformation (MT) testing, test data generation is of key importance. However, test suites are not available out of the box, and existing approaches to generate them require to provide not only the metamodel to which the models must conform, but some other domain-specific artifacts. For instance, an MT developer aiming to perform an incremental implementation of an MT may need to count on a quality test suite from the very beginning, even before all MT requirements are clear, only having the metamodels as input. We propose a black-box approach for the generation of test models where only the input metamodel of the MT is available. We propose an Ant-Colony Optimization algorithm for the search of test models satisfying the objectives of maximizing internal diversity and maximizing external diversity. We provide a tool prototype that implements this approach and generates the models in the well-established XMI interchange format. A comparison study with state-of-the-art frameworks shows that models are generated in reasonable times with low memory consumption. We empirically demonstrate the adequacy of our approach to generate effective test models, obtaining an overall mutation score above 80% from an evaluation with more than 5000 MT mutants.

1. Introduction

Model transformations (MTs) play a key role in Model-Driven Engineering (MDE). The correctness of software built using MDE techniques greatly relies on the correctness of model transformations, so it is of prime importance to test them. The core of model transformation testing is typically divided in three phases (Selim et al., 2012; Troya et al., 2022), namely test data generation, test suite assessment and oracle function. The first phase is the main focus of this research.

Novel model transformation developers, as well as students in academia, implementing an MT from scratch may require a set of input models from the beginning if they want to incrementally implement and test the model transformation. Various methods have been proposed for model generation (Gogolla et al., 2015; Guerra and Soeken, 2015; Sen et al., 2009), but the main problem is that the tester needs to be a domain expert, since many artifacts other than the source metamodel are typically required by these approaches. For instance, many approaches require a set of model fragments (Brottier et al., 2006; Fleurey et al., 2004; Sen et al., 2009) or pre-conditions (Guerra and Soeken, 2015) as input. There are other approaches that require a set of models in order to improve this initial set (Fleurey et al., 2009; Rose and Poulding, 2013). Furthermore, using specific notations or formalisms is a limitation for the usability of the test models generated by some approaches (Gogolla et al., 2005, 2015; Rose and Poulding, 2013), and there is a lack of tools to transform these models into common formats such as *XMI* (XML Metadata Interchange), which is the required input format for some well-known MT languages integrated in the Eclise ecosystem such as ATLAS Transformation Language (ATL) (Jouault et al., 2008).

For any given metamodel, there can be infinite possibilities for defining models conforming to it, so the development of appropriate test models is challenging, even for domain experts (Baudry et al., 2006, 2010). For this reason, and in order to obtain a reasonable number of generated models, there are approaches that rely on search-based algorithms for the generation of models, where the search is guided towards optimizing specific objectives (Batot and Sahraoui, 2016; Rose and Poulding, 2013; Shelburg et al., 2013; Troya et al., 2022; Wang

☆ Editor: Dr. A. Bertolino.
* Corresponding author at: MDSE Research Group, Department of Software Engineering, University of Isfahan, Iran.
E-mail addresses: meysam.karimi@eng.ui.ac.ir (M. Karimi), sh.rahimi@eng.ui.ac.ir, shekoufeh.rahimi@roehampton.ac.uk (S. Kolahdouz-Rahimi), jtroya@uma.es (J. Troya).

https://doi.org/10.1016/j.jss.2023.111882
Received 10 March 2023; Received in revised form 16 October 2023; Accepted 19 October 2023
Available online 4 November 2023

et al., 2013). An obstacle present in these approaches that hinders their usability, other than the format of the generated models in many cases, is that the tester cannot simply *hit a button* and obtain a set of models conforming to a given metamodel, but a strong domain knowledge is necessary. Furthermore, there is neither a clear search-based algorithm in this context that outperforms the others, nor any study comparing search-based algorithms for our purposes. Furthermore, there are still many search-based algorithms to be explored for model generation in the context of model transformation testing.

This paper considers all these aspects and proposes a three-fold contribution. First, we explore the use of an Ant-Colony Optimization (ACO) algorithm for model generation in the context of MT testing that, to the best of our knowledge, has not been applied in this context before. Second, we implement our approach as an easy-to-use prototype tool that can generate a set of models in XMI format by simply providing an input Ecore metamodel, which can be useful for novel MDE practitioners. Third, we compare our approach with state-of-the-art approaches in terms of performance and usability of the obtained models for the purpose of model transformation testing.

Our approach for MT testing is black-box, since we do not need the MT in order to generate test models (Troya et al., 2022). The only input is the source metamodel, as an Ecore file, which can have a set of well-formed OCL constraints defined on it, and some easy-to-set parameters with default values. The model set to be generated should have a limited number of models of manageable size, so that they can be handled by the tester. We establish two objectives for the search, namely (i) maximizing internal diversity, also known as metamodel coverage, and (ii) maximizing external diversity, also known as dissimilarity among the generated models.

In order to target this multi-objective optimization problem, we make use of an Ant-Colony Optimization (ACO) algorithm (Dorigo, 1992). The main difference of ACO with respect to other search-based approaches is that it is a constructive approach where each ant gradually constructs a solution based on swarm intelligence and heuristic information that lies in its nature. This is unlike other known algorithms such as Genetic Algorithms, which create the next solution based on update operations such as crossover and mutation. Metaheuristic algorithms usually use an encoding to represent a solution in the search space, typically in the form of binary vectors—although some works present different encodings in the field of MDE (Fleck et al., 2016b; Burdusel et al., 2018; Shariat Yazdi et al., 2016) (see Section 2.2). The problem is that if formats like XMI are to be used for the models, a customized implementation of update operators is required (Strüber, 2017). On the contrary, due to the nature of ACO, it can naturally work with formats like XMI.

Our approach is implemented by a prototype tool that takes a metamodel as input and initiates several *ants*, where each ant creates a model set. The best model set is used as output and returned as a set of models in XMI format conforming to the given metamodel, optimizing the objectives mentioned before and satisfying the optional well-formed OCL constraints expressed on the metamodel. Based on this prototype, we perform a thorough evaluation with the purpose of answering the following research questions:

- ***RQ1—Domain-independence****. Can our approach work with metamodels and OCL constraints from different domains?*
- ***RQ2—Performance****. How does our prototype perform when generating models?*
- ***RQ3—Usefulness****. Are our generated model sets able to detect faults in model transformations?*

RQ2 and RQ3 are replied not only for our approach, but we have performed a comparison study with state-of-the-art tools. In particular, we have analyzed *Viatra Solver* (Semeráth et al., 2019) and a *Random Generator* (Guerra et al., 2019). On average, our prototype tool is able to generate models 50 times faster than *Viatra Solver* and it is the one consuming the least amount of memory in the model generation process. Besides, the average mutation score of our approach is 80.49%, outperforming the other two approaches.

The remainder of this paper is structured as follows. Section 2 offers an insight into the paradigms used in this approach, namely Model-Driven Engineering and Search-Based Software Engineering. Then, Section 3 describes our approach for the generation of model sets, which is thoroughly evaluated in Section 4 in the context of model transformation testing. Section 5 introduces some related works of black-box test case generation, and finally Section 6 concludes the paper with an insight into future lines of research.

2. Preliminaries

This section introduces some concepts of the two paradigms united in this approach, namely Model-Driven Engineering (MDE) and Search-Based Software Engineering (SBSE). It also presents a running example within the context of MDE.

2.1. Model-driven engineering

Model-Driven Engineering (MDE) is a well-known methodology, which considers models as first-class artifacts for software engineering development. It is meant to increase productivity by maximizing automation and interoperability, simplifying the design process and promoting communication between stakeholders. The use of MDE principles and techniques is growing, being well established, for instance, in the development of embedded and production systems (Brambilla et al., 2017; da Silva, 2015).

2.1.1. Model transformations and (meta)models

Model transformation is a key concept in MDE for the automatic mapping of model(s) to other model(s) or code. Various types of model transformations have been defined (Czarnecki and Helsen, 2006). In this paper, we consider model-to-model (M2M) transformations, which transform one or more source model(s) to one or more target model(s).[1] A model is an abstraction of a system used to replace the system under study for a particular purpose (Kühne, 2006; Ludewig, 2003). This abstraction process allows to better manage, understand, study, and analyze models in contrast to the full system under study. In MDE, it is common that a model must conform to its metamodel. A *metamodel* defines the structure for a family of models (Mellor et al., 2004). Technically, metamodels are a special type of models that conform to a so-called meta-metamodel. In this way, metamodels are written in the language defined by their meta-metamodel. A metamodel specifies the concepts of a language, the relationships between these concepts, the structural rules that restrict the possible elements in the valid models and those combinations between elements (Brambilla et al., 2017).

M2M transformations are specified in terms of their source and target metamodels. When executed, they take as input models conforming to the source metamodels and generate models conforming to the target metamodel. In order to be able to test M2M transformations, it is helpful to count on a set of input models (Fleurey et al., 2004; Brottier et al., 2006), also referred to as *test models*, whose automatic generation is precisely the scope of this paper.

2.1.2. Running example

The simple and well-known *Families2Persons* model transformation is used as a running example. Fig. 1 presents the source metamodel of the transformation. There are two metaclasses, namely *Family* and *Member*, and four bidirectional associations between these two classes. An instance of a *Member* metaclass that specifies the members of her/his biological family can be a *familyFather*, a *familyMother*, a *familyDaughter* or a *familySon* of the biological *Family*. In turn, every instance

[1] Source/target models are also referred to as input/output models.

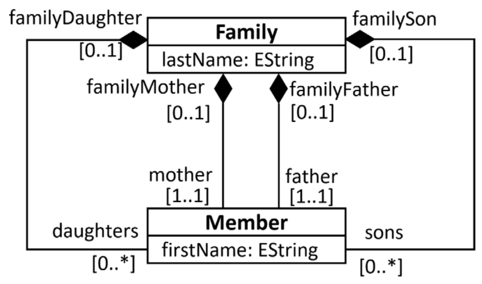

Fig. 1. Families metamodel (Burnstein, 2006).

Table 1
5 feasible test suites with 5 members created by 5 different runs.

	Equivalence Classes			
#Run	*Model1*	*Model2*	*Model3*	*Model4*
1	*2*	*2*	*0*	*0*
2	*0*	*3*	*1*	*0*
3	*2*	*1*	*1*	*0*
4	*3*	*0*	*1*	*0*
5	*1*	*2*	*0*	*1*

of a *Family* must contain a biological *father* and a biological *mother*, and any number of biological *daughters* and *sons* (Burnstein, 2006).

For exemplary purposes, Fig. 3 displays four different models conforming to the *Families* metamodel.

2.1.3. Equivalence partitioning

As mentioned before (cf. Section 2.1.1), we need to count on a set of input models to be able to test model transformations. This means that we need a set of models that conform to the input metamodel of the model transformation.

Assigning different values to each attribute in the metaclasses of a metamodel can create different valid models that conform to the metamodel. For example, if a metaclass has only two attributes of type number and string, all the different combinations of the possible values of these two properties give rise to different instances of the metaclass. If a metamodel becomes complex, the number of distinct valid models becomes very large or even infinite. This means that we need to somehow select a subset of all possible models that conform to a metamodel. A well-known technique for selecting such a subset is known as equivalence partitioning (Gogolla et al., 2015). This technique divides a metamodel into so-called equivalent classes. The idea of partitioning is that testing a member in an equivalence class is as good as testing the whole class.

Partitions can group models that are structurally different. Fig. 3 presents 4 different models with 5 objects each (apart from the root object) conforming to the *Families* metamodel showing in Fig. 1. Let us focus on the classes and the associations among them, so that each model belongs to a different partition of the *Family* metamodel—see the different number of *daughters* and *sons* in each model. Table 1 displays examples of 5 different sets of 4 models, where the 4 models in each set (shown in the rows) are categorized according to the partition they belong to (out of those in Fig. 3).

If we focus for instance on row #4, three of the four models belong to the first partition, so they all have three *members* of type *familySon*, while the other model contains two *members* of type *familyDaughter* and one *member* of type *familySon* (partition 3). Considering structural equivalence partitioning, these 4 models can be reduced to just 2 models, one belonging to partition 1 and the other to partition 3. Having a look at all rows, we can see that the sets displayed in rows 3 and 5, highlighted in the table, each cover three partitions, while the other rows only cover one or two partitions, i.e., the models are less diverse. For this reason, in this simple example, the sets of models represented in rows 3 and 5 are better choices for conforming a test suite used to test the *Families2Persons* model transformation.

2.2. Search-based software engineering

Many problems in the field of software engineering can be expressed as optimization problems (Harman and Jones, 2001). Search-Based Software Engineering (SBSE) was introduced by Harman and Jones (Harman and Jones, 2001) and is applied to optimization problems using search-based techniques. A search problem contains candidate solutions, which are searched to find (near) optimal solutions. The search is guided with a fitness function or objective function to rank the candidate solutions (Holland et al., 1992). SBSE is formed by two ingredients: a way to represent candidate solutions in a search space and a definition of the fitness function.

In recent years, several approaches have proposed to combine MDE and SBSE. For instance, the problem of finding the best orchestration for a certain set of transformation rules was considered as an optimization problem by Bill et al. (2019) and Fleck et al. (2017, 2015, 2016a). They propose the MOMoT (Marrying Optimization and Model Transformations) framework (Fleck et al., 2016b), which provides several algorithms for locally and globally searching transformation rules guided by single and multi-objective formulas. MDEOptimiser (Burdusel et al., 2018) is another tool for specifying and solving optimization problems using MDE. In this tool, the user can specify optimization problems by using a Domain-Specific Language (DSL) and the tool can run evolutionary optimization algorithms that use models as an encoding for population members and model transformations as search operators. Viatra-DSE (Hegedüs et al., 2015) is an alternative tool that uses optimization algorithms to rope transformation rules to find the most appropriate derivation chain to be applied in a prototype model. Crepe (Williams, 2013) was extended in Efstathiou et al. (2014) to support multiple objectives. It uses a Genetic Algorithm (GA) to apply mutation and crossover operations over an encoded version of models.

The proposed approach in this paper for uniting MDE and SBSE has a different goal. Our aim is to generate a set of models from scratch and use it for model transformation testing purposes. In the following, we present some SBSE concepts that are important for understanding and implementing our approach.

2.2.1. Multi-objective optimization problems

Multi-objective optimization problems are also known as Pareto-optimization problems and refer to problems with more than one objective function (Deb, 2014). These objective functions are usually in contrast with each other. Minimizing cost while maximizing performance is an easy example of a multi-objective optimization problem. In such problems, there is typically no single result solution and there are a number of Pareto-optimal solutions. A solution is called a non-dominated solution if there is a fitness value in one of its objectives that cannot be improved by other solutions (Deb, 2014). All non-dominated solutions are equally good and known as Pareto-optimal solutions (Deb, 2014).

There are algorithms that create random populations for the first iteration and then these are updated by applying different crossover and mutation operations. Examples are Getenic Algorithms (GA) and Particle Swarm Optimizations (PSO). Ant Colony Optimization (ACO) is a strong multi-agent algorithm that follows a constructive approach to generate the solutions: it explores the search space by iteratively constructing new combinations in a greedy randomized way. In this paper, an ACO-based solution is developed for generating a valid set of models based on the defined objectives. To the best of our knowledge, this is the first work using ACO for model generation. We explain its main concepts next.

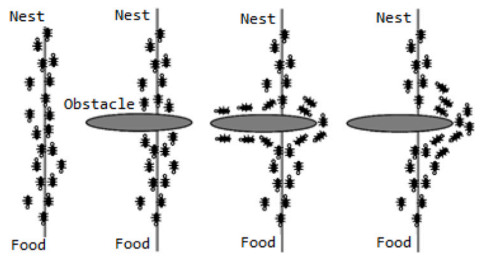

Fig. 2. Optimal route between nest and food source by ant colony (HoseinDoost et al., 2018).

2.2.2. Ant colony optimization

ACO is a well-known swarm intelligence algorithm inspired by the foraging behavior of real ants (Dorigo, 1992). In this algorithm, ants communicate indirectly with each other through stigmergy. An individual ant modifies the environment with a chemical trail called a pheromone. The other ants will respond to the new modification by choosing the paths marked by a strong amount of pheromone. This helps the ants choose the best path. After an ant finds a food source and returns to the nest, it emphasizes on the correct path by depositing pheromone again. However, if an ant does not find any food, it will not deposit pheromone on the previously passed path. Fig. 2 shows an experiment with a real colony of Argentinian ants done by Goss et al. (1989). They will choose a shorter route, gradually. When ants face an obstacle, there is an equal chance for them to choose a path (either left or right). As the left path is shorter than the right one, the ants eventually deposit a higher level of pheromone. Finally, more ants will take the left path as more levels of pheromone will be on the shorter path. There is a concept called evaporation that causes the pheromones to evaporate gradually, in order to omit the inappropriate paths. This is vital as the ants may not have chosen appropriate paths especially in the early stages.

Most concepts in ACO are very similar to other population-based meta-heuristic algorithms like GA. The most important aspect is the way in which each artificial ant selects a path. An ant considers two aspects to move from state i to j: the pheromone trail (τ), indicating how proficient it has been in the past to make that particular move, and the heuristic information (η), which indicates related knowledge of the problem that can help select a better state from the current state. For example, in the TSP problem, the heuristic information is to select a path with minimum cost for the next step. These two factors form a probability (P_{ij}) as can be seen in Eq. (1), where each ant will consider to select the next state—S indicates possible available candidate states, α and β are parameters to control the influence of (τ) and (η), respectively.

$$P_{ij} = \frac{(\tau_{ij})^{\alpha} + (\eta_{ij})^{\beta}}{\sum(\tau_{ij})^{\alpha} + (\eta_{ij})^{\beta}}, i \in [1,n], j \in S \tag{1}$$

The degree of suitability of a solution is determined by the amount of pheromone deposited by the colony according to defined fitness functions. Pheromone trail (τ) in the solution will be updated by each ant as shown in Eq. (2). Each ant deposits a positive value ($\triangle$) according to the solution fitness values. To avoid convergence to a local optimal solution, the pheromone trail evaporates over time. This is shown in Eq. (3), where ρ indicates the evaporation rate defined in the initial phase.

$$\tau_{ij} = \tau_{ij} + \triangle \tag{2}$$

$$\tau_{ij} = (1-\rho)\tau_{ij} \tag{3}$$

ACO for multi-objective problems. Several approaches have proposed the use of ACO for multi-objective problems (López-Ibáñez and Stützle, 2012; Veen et al., 2013). The methods mainly differ from each other in two aspects, namely *Pheromone trails* and *Solutions to reward*.

Pheromone trails. In single-objective problems, the pheromone amount is calculated based on the single defined objective. However, when there are multiple objectives, two different strategies are considered. The first strategy is to consider a single pheromone structure, where the amount of pheromones deposited by ants are defined according to the accumulation of different objectives (Barán and Schaerer, 2003). The second strategy is to consider several pheromone structures. In this case, different colonies of ants are associated with different objectives and pheromone structure (Doerner et al., 2003, 2004).

Solutions to reward. For updating pheromone trails, each ant has to decide about its built-in pheromone mechanisms. The first possibility is to reward the solutions that have the best fitness values for each criterion in the current iteration (Doerner et al., 2004). The second possibility is to reward any non-dominant solution in the current iteration and reward all the solutions of the Pareto set (Barán and Schaerer, 2003).

In this paper, we apply a single pheromone structure for pheromone trails and each ant rewards any non-dominant solution in each iteration.

2.2.3. Model generation using metaheuristic algorithms

When applying well-known metaheuristic algorithms, typically an initial population is formed from a set of random models that conform to the metamodel. Then, the set of models is improved according to the defined fitness functions by update operations such as crossover and mutation that are usually implemented with model transformations (Burdusel et al., 2018). Writing these model transformations requires mastery of a specific MT language, which is generally the ability of a modeler rather than a tester. This is typically done manually or via meta-learning tools (Burdusel et al., 2019; Strüber, 2017; Alhwikem et al., 2016; Mengerink et al., 2016; Hong et al., 2018), which leads to dependence on a particular technology. Furthermore, the models created as a result of the mentioned update operations may no longer conform to the metamodel (Mottu et al., 2006; Mitchel et al., 2003), so they become invalid models. An invalid model cannot be considered as a candidate solution anymore and there is usually a need for a repair operation (Burdusel et al., 2018) to generate a valid model, something that is rather costly.

3. Approach

In this section we explain our ACO-based solution for the problem of model generation, aiming to overcome some of the limitations of related approaches (cf. Section 2.2.3). To the best of our knowledge, this is the first approach that applies an ant-colony optimization algorithm for the generation of models. Our approach is technology-independent and aims to minimize human intervention. The idea is that the generated set of models is tailored to model transformation testing purposes. In the following, we first describe in detail the steps followed in our ACO-based solution. Then, we explain how OCL constraints defined on the metamodel are considered in the model generation process. Finally, we provide some details of our prototype implementation.

3.1. Test model generation applying ACO

We apply Ant-Colony Optimization (ACO) as a constructive algorithm that creates valid models at any moment in a reasonable execution time and satisfying the desired objectives, preventing the loss of good solutions (Maniezzo and Carbonaro, 1999). The objectives in our approach are (i) maximizing internal diversity and (ii) maximizing external diversity (Semeráth et al., 2020). As detailed later in Section 3.1.3, *internal diversity* refers to the percentage of the metamodel covered by each generated model, while *external diversity* indicates the degree of structural difference between models within a model set. A pseudocode of the models generation process is presented in Algorithm 1, and the different steps of our approach are displayed in Fig. 4. In the following we explain each of these steps.

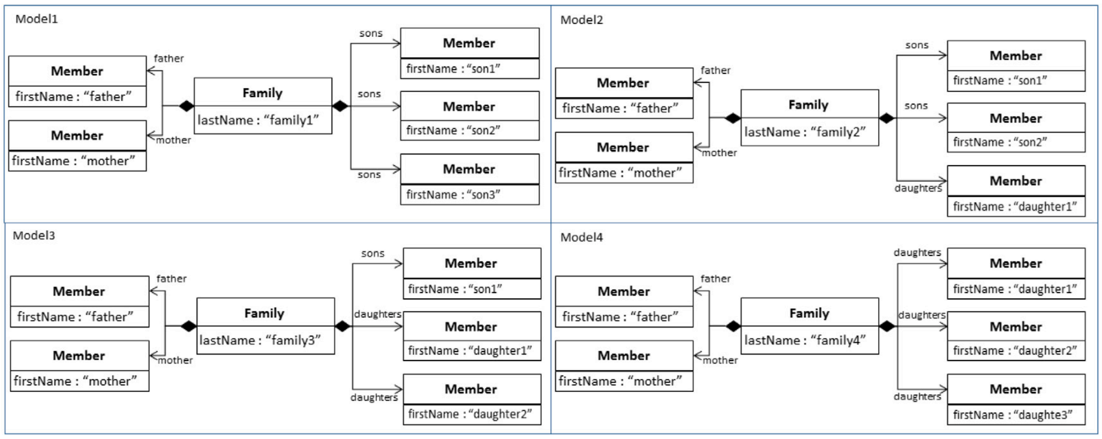

Fig. 3. Sample models conforming the *Family* metamodel.

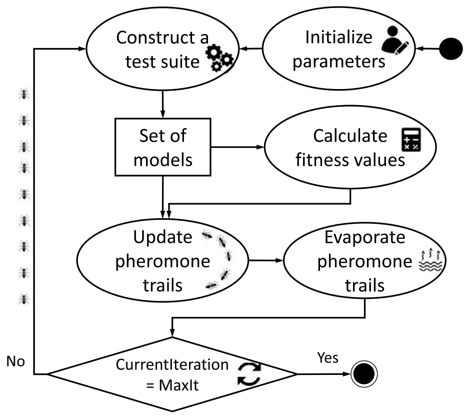

Fig. 4. Steps of our ACO-based approach.

Algorithm 1 Model generation using ACO

```
1: let $nElites = nModels$ // Number of elites to go to the next iteration
2: for $iteration = 1, 2, \ldots, MaxIt$ do
3:     for $ant = 1, 2, \ldots, NAnt$ do
4:         Construct a model using pheromone trails
5:         Add model to $CurrentPop$
6:     end for
7:     $NewPop = Merge(EliteModels, CurrentPop)$
8:     $Sort(NewPop)$ by fitness function
9:     Update pheromone trails
10:    Evaporation
11:    let $EliteModels$ = pick top $nElites$ from $NewPop$
12: end for
```

3.1.1. Step 1: Initialize parameters

In order to construct the initial set of models, the user needs to identify three inputs, namely (i) the metamodel with optional well-defined OCL constraints to which generated models must conform, (ii) the number of models to generate and (iii) the number of elements to be contained in the models, i.e., the number of *EObjects* that each generated model must have. The second and third inputs are important for narrowing down the search and achieving acceptable execution times (cf. Section 4.6). There are other parameters related to the ACO configuration, such as *nAnts*—number of ants in the algorithm—, *MaxIt*—maximum number of iterations (cf. Fig. 4)—, and α, β and ρ values (cf. Section 2.2.2). Our ACO-based framework offers some default values for these parameters, so that users do not need to have any knowledge about ACO. In any case, these parameters can also be customized by the user.

For exemplary purposes, we suppose for the running example that the tester wants to create a set of models containing 10 models of the *Family* metamodel, where each model must have 5 elements apart from the root element (cf. Section 2.1.3 and Fig. 3).

3.1.2. Step 2: Construct a test suite

At the very first iteration of our algorithm, there are no pheromone trails deposited (cf. Section 3.1.3) to guide an individual ant in constructing a suitable model. Each individual ant searches the solution space based on heuristic information and generates a semi-random model according to the number of elements specified. Our algorithm generates valid models conforming to the input metamodel, considering both cross-references and containment associations as specified by the metamodel. Let us recall that ACO is a constructive algorithm that considers heuristic information to make a solution (cf. Sections 2.2.1 and 2.2.2). The main heuristic information to construct a solution in our case is *internal diversity* when no other solution exists to calculate the external diversity—which is the case in the first iteration (cf. Section 3.1.3). This is, we prioritize internal diversity in the first iteration so that the initial set of models covers the metamodel as much as possible. For instance, if we have a look at Fig. 3, in which we consider four ants (each ant creates one model), we can see that *Model2* and *Model3* offer the highest internal diversity, since they both consider all *EObjects* – both *EClass*es and *EReference*s – of the metamodel (cf. Fig. 1).

From the second iteration onwards, each ant tries to construct a new model, given the experience in the ACO algorithm shared between the ants by *trail pheromone* as well as the heuristic information. This is, thanks to the pheromone trails deposited in previous iterations, each ant computes the probability of selecting a model element to include it in the new model under construction. This way, ACO does not apply the operations of mutation and crossover that metaheuristic algorithms typically apply, overcoming the limitations described in Section 2.2.3.

3.1.3. Step 3: Calculate fitness values and deposit pheromone trails

After models are created by the ants in the current iteration ($CurrentPop$ in Algorithm 1), a new population including current models and best models transferred from the last iteration ($EliteModels$) are merged in a new collection ($NewPop$). Recall that $EliteModels$ collection is *null* in the first iteration and only models generated by the ants in that round are considered. In this step, each model in the $NewPop$ is evaluated according to the fitness function containing our two objectives, so that the models with better fitness values deposit more pheromones on the obtained model elements. The two objectives we consider are *(i) maximizing metamodel coverage of each model (internal diversity)* and *(ii) maximizing dissimilarity among the different models (external diversity)*. For model transformation testing, diverse models need to be synthesized, where every two models must be structurally different in order to achieve high coverage or a diverse solution space (Varró et al., 2018). On the one hand, maximizing metamodel coverage is a key objective for generating test models, since a larger part of the model transformation will likely be exercised, therefore having more chances to detect errors (Semeráth et al., 2020). On the other hand, by maximizing the dissimilarity among the generated models, the overlap between the models is minimized, assuring different models exercise different parts of the model transformation under test (Semeráth et al., 2020). In the following we explain the implementation of these two objectives.

Internal diversity. As defined by Varró et al. (2018), the diversity of a model M_i is defined as the number of *(direct)* types used from its MM: M_i is more diverse than M_j if more types of MM are used in M_i than in M_j. There are various methods for measuring diversity. For instance, Semeráth et al. (2020) and Semeráth and Varró (2018) focus on graph models and propose an internal diversity metric based on neighborhood and predicate shapes. In our case, we aim at constructing models (Ecore models in our current implementation) and we try to avoid conversion between representations to optimize the performance of our approach. For this reason, we use the definition by Fleurey et al. (2009), in which internal diversity is referred to as *metamodel coverage* and it is obtained by considering class coverage, attribute coverage and association coverage. As explained in a recently published survey on model transformation testing and debugging (Troya et al., 2022), this and similar approaches have been followed by other authors (Wang et al., 2006; Finot et al., 2013; Calegari and Delgado, 2013).

In order to compute metamodel coverage, our algorithm iterates over all model elements of the metamodel. Only if all model elements are covered, we can say the metamodel is completely covered. In our approach, the ratio of the number of covered *EClasses* and the ratio of the number of *EReferences* are calculated and divided by 2, so that the result is in the range. For example, *Model2* in Fig. 3 has a metamodel coverage of 1 as it covers all the elements in the metamodel, including 2 *EClass*es named *Family* and *Member* (2/2 = 1) and 8 different *EReferences* between the *Family* and *Member EClass*es (8/8 = 1); but *Model4* has an internal diversity of $(1+0.75)/2 = 0.875$ as the *EReference*s namely *son* and *familySon* are not covered.

External diversity. External diversity (also known as *dissimilarity*) among models can be achieved by considering all possible 2-tuple combinations of models in a model set. If there is an instance of a type or coverage criterion that has a different structure in another model, then dissimilarity between them is increased. As explained by Semeráth and Varró (Semeráth and Varró, 2018), *external diversity* captures the distance between pairs of models. They explain that this diversity distance between two models is proportional to the number of different neighborhoods covered in one model but not the other. The concept of *neighborhood* is taken from graph models (Semeráth and Varró, 2018), where the neighborhood of an object describes all unary (class) and binary (reference) relations of the objects within a given range. Informally, neighborhoods can be interpreted as richer types, where the original classes are split into multiple subclasses based on the difference in the incoming and outgoing references Formally, external diversity is measured as $d_i^{ext}(M_j, M_k) = |S_i(M_j) \oplus S_i(M_k)|$, where $\oplus$ denotes the symmetric difference of two sets (Semeráth and Varró, 2018).

In our approach, each model is compared with the other models in pairs after each ant has built its own model, from which we obtain the external diversity. For instance, *Model1* and *Model2* shown in Fig. 3 differ in two elements. Specifically, the *EReferences* between the *Family* object and the last *Member* object are *son* and *familySon* in *Model1*, while these *EReferences* are *daughter* and *familyDaughter* in *Model2*. This means that $d_1^{ext}(Model1, Model2) = 2$.

Finally, *models* are ranked according to *metamodel coverage* and then sorted according to *dissimilarity*, so that *internal diversity* takes precedence over *external diversity*. Then the top *K models* (so-called *elites*) that need to go to the next iteration are stored in a shared memory between iterations. In the next iteration, the *elite models* that have reached this stage from the previous iteration and the other newly created models form the new population (by new ants) are rearranged according to *diversity metrics* and form the elite population of the next iteration. In the final iteration, this elite population will be the output of the algorithm and will include the *N* models that should appear in the output (cf. Algorithm 1).

3.1.4. Update and evaporate pheromone trails

Right before running the next iteration of the algorithm, two main phases of the ACO algorithm are executed. First, the previous *pheromone trails* are updated on the model elements, which were initially empty. For the running example, in *Run #5* in Table 1 (where we consider that *Model1* to *Model4* represent different equivalence partitions as explained in Section 2.1.3), the ant in *Model1* and the ant in *Model4* deposit more pheromones onto their model elements in comparison to the two ants in *Model2*, as they have higher external dissimilarity between their constituent models. The reason is that there are two models of type *Model2* in that run, which causes a smaller external diversity in this type of model. Let us recall that models violating any constraint on the metamodel will not deposit any pheromones at all (cf. Section 2.1.3).

After updating the pheromone trails, *evaporation* takes place, in which the pheromones poured on the model elements are evaporated at a constant rate. This causes disappearance of the pheromone on the models that satisfy the objectives to a lower extent over time. Therefore, models with better objective values will likely have more pheromones onto their model elements, and these are passed to the next iterations, leading to the convergence of the algorithm in the final rounds.

3.2. OCL constraint rules check

For a model to be considered valid, it needs to conform to its metamodel and satisfy the static constraints defined on it (Derasari, 2021). These constraints must be represented with the Object Constraint Language (OCL) (Warmer and Kleppe, 2003) in our approach, as it is a common practice in many MDE-related approaches (Troya et al., 2022). OCL is a declarative language, and it is used for defining rules on Meta-Object Facility (MOF) metamodels, including UML. In our implementation, in addition to the input parameters (cf. Section 3.1.1), the user can upload a file with OCL constraints that follow the Eclipse OCL[2] rules.

Differently from SAT solvers, which check the validity of the models regarding static constraints after the complete models are generated, our algorithm integrates incremental checks of the constraints. As it builds a model and immediately after a model element is included, only those OCL constraints related to that particular element are extracted and checked against the model that has been constructed so far. If the addition of this element maintains the model's validity, the model element can be accepted and the construction of the model with the next meta-element candidate can be continued. Otherwise, the model element is discarded and a new model element will be generated. With this solution, we do not need to wait for the complete model to be generated in order to check the constraints, which can help save time when the generated models are frequently discarded due to the unsatisfiability of the constraints. Besides, it fits with the constructive nature of the ACO algorithm.

Like with SAT solvers, the main problem of this solution is when there are constraints that are complex to be satisfied. In this case, our algorithm may get stuck in a potentially infinite loop, since no added element at a certain point can make a specific constraint be satisfied. To tackle this issue, we define a customizable value in which the user

[2] https://projects.eclipse.org/projects/modeling.mdt.ocl

controls the number of attempts to generate a new model element. The default value for this number is considered 10 in our prototype implementation. When an ant is unable to generate an element within a given effort range, this ant tries again to construct another model from scratch. If the number of attempts for including an element that satisfies a constraint is increased, then the possibility for generating valid models in complex scenarios also increases (although the algorithm could take longer). The trade-off between performance and finding valid models in our algorithm is flexible due to the inputs described above.

3.3. Prototype implementation

A prototype of the approach has been implemented in Eclipse using Java 11. It is essential for the user to provide the explained inputs in Section 3.1.1. EMF Ecore is used to define the input metamodel and Eclipse OCL is applied for specifying the constraints on the metamodel. The models generated by our tool are represented in the well-established XMI file format, so that they can be used as input for any model transformation language that accept XMI models as input. Our prototype tool is available from our project's website (Karimi et al., 2023).

4. Evaluation

In this section we present an evaluation of our approach and framework based on six case studies. In particular, we are interested in answering the following research questions (RQs):

- ***RQ1—Domain-independence***. *Can our approach work with metamodels and OCL constraints from different domains?* We want to evaluate whether our approach is applicable to metamodels and OCL constraints of different nature, size and complexity.
- ***RQ2—Performance***. *How does our prototype perform when generating models?* We want to evaluate the performance of our approach with metamodels of different sizes and compare it with the performance of state-of-the-art approaches.
- ***RQ3—Usefulness***. *Are our generated model sets able to detect faults in model transformations?* We apply mutation analysis in order to determine whether the models generated by our approach are able to detect faults in model transformations when using them as input, and we compare the results with state-of-the-art approaches.

In order to answer RQ3, we use five model transformations that target different problem areas and differ in their level of complexity regarding number and types of features used (size of input/output metamodels, number of rules, use of imperative rules, filters, helpers…). In order to answer RQ1 and RQ2, we generate models conforming to the five input metamodels of the model transformations mentioned and, for answering RQ2, we use one additional large metamodel. Also, to appropriately answer RQ1 and RQ2, we have defined sets of OCL constraints for the five metamodels that must be satisfied by the generated models. Finally, for answering RQ2 and RQ3, we do not only analyze the results given by our approach and tool, but we also compare it with state-of-the-art approaches.

The rest of this section is structured as follows. First, we describe, apart from the running example presented in Section 2.1.2, the other five case studies used in the evaluation. Second, we describe the approach and tool we have used in order to obtain mutants for model transformations, which are needed for answering RQ3. Third, we explain the state-of-the-art approaches with which we compare our approach in RQ2 and RQ3, namely *Viatra solver* and *Random Generator*. Fourth, we describe the execution environment. Fifth, we detail the evaluation process, which includes explanations on the size of the models obtained, the OCL constraints defined, the types of model transformation mutants used in the experiments, the way the different tools have been configured and the number of runs and statistical tests performed. Sixth, we present the results and the answer to the three RQs. Finally, we discuss some aspects of our approach and present the threats to the validity of this evaluation.

4.1. Case studies

Let us recall that our ACO-based solution is independent from the model transformation language, since it only depends on the metamodel to which generated models must conform. However, we do need to select actual model transformations in order to evaluate the usefulness of our approach (RQ3). We have chosen transformations written in ATL (Jouault et al., 2008), since it is currently one of the most commonly-used model transformation languages and has become a de-facto standard in MDE for implementing model transformations (Troya et al., 2022). Specifically, we have selected five ATL model transformations, as shown in Table 2. *CPL2SPL, Families2Persons* and *Grafcet2PetriNet* are available on the open-source ATL Zoo repository (ATL, 2006), while *SOOML2SOOPL* and *Families2Persons_Extended* have been taken from previous research works (Oakes et al., 2018; Troya et al., 2018b). They all differ in the application domain and have been used as case studies in several papers related to model transformation testing (Alkhazi et al., 2020; Burgueño et al., 2015; Cuadrado et al., 2017; Oakes et al., 2018; Troya et al., 2018a,b).

Table 2 summarizes some information of the model transformations (ignore *Mutants* column for now). We can see the number of classes, relationships and attributes of their input and output metamodels, as well as the number of lines of code and rules of each model transformation. The sizes of the input metamodels vary from 2 classes and 4 relationships in the *Families2Persons* case study to 33 classes and 17 relationships in the *CPL2SPL* case study. The different case studies are briefly described in the following.

- ***CPL2SPL***. This transformation, described in Jouault et al. (2006), is a relatively complex example available in the ATL Zoo (ATL, 2006). It handles several aspects of two telephony DSLs, SPL and CPL, and was created by the inventors of ATL.
- ***Families2Persons_Extended***. This is an extended version of the original Families2Persons model transformation that can be found in the ATL Zoo and that has been discussed in a number of related works on verification and testing (Gogolla and Vallecillo, 2011). It was first proposed by Oakes et al. (2018) and used for evaluating other approaches (Troya et al., 2018b).
- ***Grafcet2PetriNet***. This transformation establishes a bridge between grafcet, a mainly French-based representation support for discrete systems models, and petri net models. This transformation is available on the ATL Zoo repository (ATL, 2006).
- ***SOOML2SOOPL***. This model transformation takes as input a model representing an Object-Oriented Modeling Language and transforms it into a model representing an Object-Oriented Programming Language. This ATL model transformation has been created by the Business Informatics Group of the Institute of Software Technology and Interactive Systems at the Vienna University of Technology (TU Wien) as part of a Master Course and has been used to evaluate previous works on model transformation testing (Troya et al., 2018b).

Apart from these model transformations, we are using another metamodel for answering RQ2. In particular, it is a large metamodel related to the Maude language (Clavel et al., 2007). The *Maude metamodel*[3] consists of 45 metaclasses and covers part of the Maude language, where it is restricted to the (big set of) elements used in the formalization of models and metamodels (Troya and Vallecillo, 2011).

[3] This metamodel can be accessed from http://atenea.lcc.uma.es/projects/MaudeMM.html.

Table 2
Model transformations used as case studies.

Transf. Name	# Classes MM Inp. - Outp.	# Rel. MM Inp. - Outp.	# Attr. MM Inp. - Outp.	# LoC	# Rules	# Mutants syntactic - semantic - typing
CPL2SPL	33 - 77	17 - 70	42 - 19	503	19	1620 - 332 - 95
Families2Persons	2 - 3	4 - 2	2 - 1	49	2	34 - 55 - 15
Families2Persons_Extended	11 - 12	21 - 9	3 - 2	110	9	618 - 163 - 36
Grafcet2PetriNet	9 - 9	13 - 14	6 - 3	89	5	381 - 112 - 49
SOOML2SOOPL	15 - 10	27 - 18	5 - 5	269	10	1092 - 279 - 140
Total						5021

4.2. Model transformation mutants

In mutation analysis, a mutant is a variation of the original system under test (SUT) where an artificial bug is inserted. If the executions of a SUT and its mutant for a specific input yield different outputs, then we say that the mutant has been killed. This means that this input has been used to detect that the mutant was not the original SUT. If there is no input able to kill a mutant, then we say that the mutant remains alive. The mutation score determines the percentage of mutants that are killed by the available inputs. In the context of model transformations, it is ideal to count on a set of input models able to kill as many model transformation mutants as possible.

In order to perform mutation analysis, we need model transformation mutants where artificial bugs are seeded. For obtaining mutants, we have used different types of mutation operators, namely *syntactic operators* (Troya et al., 2015), *semantic operators* (Aranega et al., 2015) and *typing operators* (Cuadrado et al., 2017). *Syntactic operators* use create–update–delete (CUD) actions with the 18 different operators proposed by Troya et al. (2015). The aim of *semantic operators* is the same as the ones presented by Mottu et al. (2006), i.e., they try to mimic common semantic faults that programmers introduce in model transformations. Finally, *typing operators* inject typing errors (e.g., changing the return type of a helper) or faults causing runtime errors (e.g., deleting a parameter in a called rule) based on the most frequent typing errors found in the ATL zoo (Cuadrado et al., 2017).

Guerra et al. propose a Java framework (Guerra et al., 2019) that facilitates the mutation analysis of ATL model transformations. This framework provides the automatic generation of ATL model transformation mutants and includes all mentioned mutation operators. This approach does not only generate a set of mutants for a given ATL model transformation, but it also allows to apply mutation analysis given a set of input models. This means that, having the set of models generated by our approach, we can check which mutants they are able to kill. According to Guerra et al. (2019), a mutated transformation can be killed in three ways: (i) the mutant crashes when executed on a test model, (ii) the output model does not conform to the target metamodel, or (iii) the output model is not the expected one, which can be checked using a total or a partial oracle.

4.3. State-of-the-art approaches for model generation

As mentioned before, we compare the performance (RQ2) and usefulness in model transformation testing (RQ3) of our approach by comparing it with two state-of-the-art approaches, which are explained below.

4.3.1. Viatra solver

Viatra Solver[4] is an open-source tool that automatically synthesizes consistent graph models (Semeráth et al., 2019). It takes a metamodel and a set of well-formed rules as input, and it uses a SAT solver to obtain a diverse set of consistent graph models conforming to the metamodel, satisfying the consistency constraints and structurally different among each other. The model generation starts from an abstract partial model or a user-specified seed model piece. Partial models grow with unit decision and diffusion rules adapted from key SAT solving techniques. Each step during model generation modifies and develops a previous minor model, while regularly ensuring that consistency is not violated. If the constraint violation can no longer be repaired to have a valid model in hand, or the partial model has already been inspected, the model generation process will be reversed. This tool enumerates the generated models according to equivalent classes based on neighborhood shapes. Viatra Solver supports Alloy (with back-end SAT solvers like Sat4j and Minisat), Z3 SMT solver, and features a new graph solver. The framework uses the background solvers automatically, so no additional skill is required by the language developers. The generated set of models can be more diverse than those of other approaches using logic solvers in the backend, thus it is more appropriate to be applied, e.g., for mutation testing scenarios (Semeráth et al., 2019). This is the main reason why we select this tool for comparing it with ours.

[4] https://github.com/Viatra/Viatra-Generator/wiki/

4.3.2. Random generator

The tool by Guerra et al. (2019) explained in Section 4.2 also allows the creation of a set of random models given a metamodel, where the user can specify the number of models to create and the relative size of the models (size ± ratio). In the evaluation results, the model generation by this approach is referred to as *"Random Generator"*, where the ratio is set as 0.0001 to have an almost fixed size in the comparisons.

4.4. Execution environment

The machine in which the experiments have been carried out is a PC running the 64-bit OS Microsoft Windows 10 with an AMD Ryzen 4800H 2.90 GHz processor and with 16 GB of RAM. For this, we have used Eclipse Modeling Tools (IDE 2020-06 (4.16.0)), and we had to install the ATL plugin (version 4.2.0).

4.5. Evaluation process

This section explains the different models that have been generated with the different approaches in order to answer the RQs. It also describes the OCL constraints defined on the MMs, which help reply RQ1, and summarizes the model transformation mutants that have been obtained for answering RQ3. Finally, it describes the configuration of the three tools under comparison as well as the number of runs executed and statistical tests performed.

4.5.1. Generated models

In order to properly evaluate our approach and the state-of-the-art approaches regarding model generation, we need to define the models that are to be generated in this evaluation. In this sense, we drive the model generation according to two attributes, namely *model size* and *number of models*. The former refers to the number of elements in each generated model, while the latter means the number of models obtained in each run.

Table 3
Performance metrics with different model sizes and number of elements.

Metamodel	# Models	# Elements	ACO		Viatra Solver		Random Generator	
Performance indicators			Execution time (s)	Memory Consump. (MB)	Execution Time (s)	Memory Consump. (MB)	Execution Time (s)	Memory Consump. (MB)
CPL	50	10	1.5	**29.9**	39.8	359.9	**0.8**	155.4
		20	1.6	**36.6**	40.3	381.1	**0.8**	170.0
		40	1.8	**49.8**	54.8	375.0	**0.8**	232.0
		80	2.0	**75.6**	93.6	459.7	**1.0**	223.3
		160	2.4	**124.0**	139.6	451.4	**1.0**	195.6
		320	3.0	**130.4**	385.8	944.5	**1.2**	193.5
	100	10	1.7	**43.2**	51.00	655.8	**0.8**	222.2
	150		2.8	**61.2**	82.3	413.0	**1.5**	170.4
	200		2.4	**82.8**	113.0	400.3	**1.6**	167.7
	250		2.6	**109.7**	143.0	439.0	**2.3**	134.1
	300		2.9	**140.5**	200.9	423.1	**2.2**	194.6
	350		3.0	**34.2**	214.0	420.0	**2.9**	236.7
	400		3.4	**74.7**	243.3	449.4	**2.9**	233.0
	450		**3.3**	**118.0**	248.6	450.0	3.6	202.5
	500		**3.5**	**27.0**	278.7	517.7	3.6	184.4
Families	50	10	1.5	**24.3**	18.7	183.5	**0.4**	191.8
		20	1.5	**27.1**	19.8	110.1	**0.5**	170.5
		40	1.5	**34.5**	30.1	150.5	**0.6**	175.7
		80	1.7	**47.6**	52.5	267.2	**0.7**	222.5
		160	2.2	**75.6**	23.5	406.8	**0.8**	226.7
		320	2.5	**126.4**	20.0	553.3	**1.0**	191.5
	100	10	1.6	**32.1**	51.6	139.4	**1.0**	202.6
	150		1.9	**51.2**	79.8	195.6	**1.4**	277.8
	200		2.2	**71.7**	67.9	136.7	**1.3**	191.3
	250		2.5	**95.7**	81.1	215.3	**1.6**	218.9
	300		2.7	**125.7**	85.6	150.1	**2.1**	160.7
	350		3.0	**18.1**	115.6	164.9	**2.5**	233.1
	400		3.2	**56.0**	141.8	186.6	**2.6**	157.3
	450		3.5	**99.5**	130.7	191.0	**3.1**	173.9
	500		**3.5**	**99.7**	181.4	143.3	3.6	188.4
Families Extended	50	10	1.0	**26.6**	9.4	212.6	**0.8**	245.4
		20	1.1	**32.5**	–	–	**1.0**	201.5
		40	1.3	**44.1**	–	–	**0.9**	205.8
		80	1.6	**69.7**	–	–	**0.9**	232.5
		160	1.8	**131.0**	–	–	**0.9**	169.8
		320	2.1	**20.7**	–	–	**0.9**	150.5
	100	10	1.2	**38.9**	15.9	264.4	**0.8**	239.6
	150		1.5	**56.6**	25.8	291.5	**1.2**	173.8
	200		**1.7**	**78.0**	30.2	309.1	1.9	194.4
	250		2.0	**105.2**	34.2	313.9	**1.8**	185.3
	300		**2.1**	**135.9**	64.8	470.0	2.2	213.4
	350		2.4	**29.3**	59.1	372.2	**2.1**	186.0
	400		**2.6**	**69.0**	71.6	347.8	2.8	196.7
	450		2.7	**114.5**	70.0	231.3	**2.3**	165.4
	500		2.9	**22.9**	85.4	360.1	**2.6**	229.5
Grafcet	50	10	1.5	**25.7**	43.2	152.0	**0.5**	215.6
		20	1.5	**30.9**	30.9	185.5	**0.6**	207.9
		40	1.8	**41.7**	37.8	262.7	**0.7**	189.2
		80	2.1	**65.6**	60.9	200.1	**0.8**	188.3
		160	2.5	**118.6**	147.7	397.8	**0.9**	175.9
		320	3.4	**122.8**	481.8	649.1	**1.3**	198.0
	100	10	1.7	**38.5**	60.0	219.7	**0.9**	248.9
	150		2.1	**56.0**	108.8	171.5	**1.2**	246.4
	200		2.4	**79.0**	162.2	210.7	**1.2**	177.7
	250		2.6	**104.5**	216.0	262.6	**1.2**	174.2
	300		2.9	**135.1**	267.5	230.1	**1.3**	175.7
	350		3.1	**29.6**	426.0	359.4	**1.5**	175.4
	400		3.4	**68.0**	464.8.0	337.7	**1.9**	165.6
	450		3.6	**113.7**	634.6	279.4	**2.0**	229.8
	500		4.2	**21.0**	637.2	284.1	**2.1**	148.8

(*continued on next page*)

In our evaluation, the values of these two attributes are shown in columns *# Models* and *# Elements* in Table 3. For all five metamodels, we follow the same two patterns:

- First, we define a fixed number of models, 50, and we vary the number of model elements to generate in each model, $\langle 10, 20, 40, 80, 160, 320 \rangle$.
- Second, we vary the number of models to generate in each run (from 100 to 500 with an increase of 50 models), where all have a fixed size of 10 model elements.

Regarding the *Maude* metamodel, it is used especially to check the limitations of the different tools. To this aim, the model size is fixed in

Table 3 (*continued*).

Metametamodel	# Models	# Elements	ACO		Viatra Solver		Random Generator	
Performance indicators			Execution time (s)	Memory Consump. (MB)	Execution Time (s)	Memory Consump. (MB)	Execution Time (s)	Memory Consump. (MB)
SOOML	50	10	1.1	**25.2**	16.6	183.8	**0.6**	247.5
		20	1.1	**31.1**	21.9	214.4	**0.7**	205.0
		40	1.2	**41.7**	21.7	202.0	**0.7**	220.3
		80	1.4	**64.1**	26.1	221.2	**0.9**	128.8
		160	1.7	**117.1**	22.1	242.8	**1.1**	195.9
		320	1.9	**116.2**	25.9	253.9	**0.9**	192.2
	100	10	1.2	**38.3**	30.1	190.6	**0.9**	209.8
	150		1.6	**55.5**	78.2	249.0	**1.3**	241.6
	200		1.7	**76.6**	54.2	172.6	**1.5**	170.0
	250		2.0	**102.6**	32.2	216.8	**1.7**	222.7
	300		**2.1**	**132.6**	43.8	219.4	**2.1**	184.2
	350		**2.2**	**26.2**	49.1	244.0	2.4	187.3
	400		**2.4**	**64.7**	45.5	271.8	2.5	187.3
	450		**2.6**	**108.2**	63.0	269.5	2.7	193.9
	500		**2.7**	**16.9**	78.1	271.5	3.2	229.9

1000 model elements, and the number of models to generate in each run scales up to 1 million (cf. Section 4.6).

4.5.2. OCL constraints

In order to ensure that the approaches are able to work with constraints imposed on the metamodels, we have defined a set of OCL constraints for them. We use some well-known OCL operations in these constraints, such as *size()*, *oclIsUndefined()*, *allInstances()* or *notEmpty()*.

Listing 1: Examples of OCL constraints the for *Families* metamodel

```
context Family inv LastNameWithSpecificSize:
    self.lastName.size() > 10 and self.lastName.size() < 14

context Member inv FirstNameAtLeastFiveCharacters:
    self.firstName.size() > 4

context Member inv MoreThanThreeMembers:
    self.allInstances()->size() > 3

context Family inv MustHaveAtLeastOneSon:
    self.sons->size() > 0

context Family inv MustHaveAtLeastTwoDaughters:
    self.daughters->size() > 1

context Family inv SameNumberOfSonsAndDaughters:
    self.daughters->size() = self.sons->size()
```

As an example, the OCL constraints for the *Families* metamodel (cf. Fig. 1) can be found in Listing 1. The first one, called *LastNameWithSpecificSize*, states that the last name of every family must have more than 10 and less than 14 characters. The second constraint forces all members' first names to have, at least, five characters. The constraint called *MoreThanThreeMembers* forces models to have, at least, four objects of type *Member*. The next two constraints determine the minimum number of sons and daughters that each family must have, 1 and 2, respectively.

The constraints for the other metamodels can be consulted on our project's website (Karimi et al., 2023). It can happen that some constraints contradict others. For this reason, when considering the constraints in the model generation process, we may consider different sets of constraints in different runs. This is exemplified in columns *Constraints* of Table 4. In the *Grafcet* metamodel, constraints 4 and 5 contradict constraint 6. This is the reason why constraints 4 and 5 are used in one run and constraint 6 is used in a different one.

It is important to highlight that the *Random Generator* (cf. Section 4.3.2) does not support constraints defined on the metamodel. Regarding *Viatra Solver* (cf. Section 4.3.1), it does not accept constraints written in OCL, but in VQL (Viatra Query Language)[5]. VQL reuses the concepts of graph patterns, making it an easy way to specify graph model queries. A major limitation is that OCL constraints cannot be applied directly for generating models in *Viatra Solver*, since no OCL-VQL converter is currently embedded in the tool. In fact, VQL does not currently[6] support some OCL operators (Balaban et al., 2016) and, therefore, some OCL rules cannot be converted to their equivalent VQL constraints. As an example, the *LastNameWithSpecificSize* constraint explained before is not applicable in *Viatra Solver*, as VQL does not support queries which include the String data type.

As a result, a modified version of some of the OCL constraints are used for evaluating *Viatra Solver*. The VQL constraints shown in Listing 2 for the *Families* metamodel of our running example are equivalent to the two last OCL constraints shown before. For evaluating the performance, we can only consider a subset of the OCL constraints defined on the metamodels when evaluating *Viatra Solver* (see columns *OCL Constraints* and *VQL Constraints* in Table 4). The complete set of VQL constraints for the different case studies can be checked on our project's website (Karimi et al., 2023).

Listing 2: Examples of VQL constraints for the *Families* metamodel

```
pattern HaveAtLeastOneSon(family:Family, mem : Member) {
    Family.sons(family, mem1);
}

pattern HaveAtLeastTwoDaughters(family: Family, mem:Member) {
    Family.daughters(family, mem1);
    Family.daughters(family, mem2);
    mem1 != mem2;
}

@Constraint(message="rule1", severity="error", key={F})
pattern invalidFamily(family : Family) {
    neg find HaveAtLeastOneSon(family,_);
}

@Constraint(message="rule2", severity="error", key={F})
pattern invalidFamily2(family : Family) {
    neg find HaveAtLeastTwoDaughters(family,_);
}
```

4.5.3. MT mutants obtained

By using Guerra et al.'s approach (Guerra et al., 2019) and the tool provided, we have created 5021 mutants for the five case studies (cf. last column in Table 2). The table shows the number of mutants of each model transformation classified by their type (syntactic, semantic and typing; cf. Section 4.2) for each case study. Please note that this classification is different from the classification proposed by Guerra et al. (2019) on how a mutated transformation can be killed (cf.

[5] https://www.eclipse.org/viatra/documentation/query-language.html

[6] As of February 2023.

Table 4
Performance metrics with different constraints (number of models to generate is set to 50).

Metamodel	# Elements	ACO			Viatra Solver		
		OCL Constraints	Execution Time (s)	Memory (MB)	VQL Constraints	Execution Time (s)	Memory (MB)
CPL	30	1–4	1.6	32.5	3, 5	3.2	345.3
		4–8	2.5	106.3			
		1–8	3.3	61.2			
Families	7	1–6	2.8	1.1	4, 5	11.6	166.3
Families Extended	30	1–6	1.5	68.0	2, 3	17.3	176.3
Grafcet	30	1-3, 6	2.5	81.2	3, 5	43.8	220.0
		1–5	1.9	1026			
SOOML	30	1–7	4.0	94.0	2-5	27.3	253.2
		1, 8, 9	2.5	99.5			

Section 4.2). As explained before, given a set of input models for a model transformation, this tool performs mutation analysis and prints a report with the results, where the mutation score for all types of mutation operators is displayed. This means that, by relying on Guerra et al.'s framework, we do not have to deal with either the generation of the model transformation mutants or the definition of the oracles. Besides, we avoid any possible bias in the evaluation results by using a third-party tool. Therefore, this tool is used for performing the mutation analysis and replying RQ3, where the models that are fed to the tool are those obtained by our tool and *Viatra Solver*.

4.5.4. Tools configuration

When not said explicitly otherwise, the fixed parameters for the experiments in our ACO-based solution are $\alpha = 1$, $\beta = 1$, $\rho = 0.05$ (cf. Sections 2.2.2 and 3.1.1). Regarding *Viatra Solver*, we choose the default SAT solver at the back-end. As for the *Random Generator*, the ratio is set as 0.0001 as explained in Section 4.3.2.

4.5.5. Number of runs and statistical tests

Search-based algorithms may produce different results per run (Saidani et al., 2020). To deal with this random nature, it is important to evaluate their effectiveness by performing a large number of runs, which should be at least 30, as suggested by Arcuri and Briand (2011). Additionally, it is essential to use the statistical tests that provide support for rejection of the conclusions derived by analyzing the obtained results (Saidani et al., 2020). In this paper, we employ Kruskal–Wallis test (McKight and Najab, 2010) in order to detect significant performance differences between the algorithms under comparison (α is set at 0.05). In this validation, each iteration is repeated 30 times for each algorithm and each metamodel. Also, to ensure which approach is superior in comparison with the others, Wilcoxon rank sum test (McKight and Najab, 2010) is employed for pair-wise comparison of algorithms.

4.6. Results

4.6.1. RQ1 - Domain-independence

Our first RQ has to do with the compatibility of our approach to work with different case studies coming from different domains. Indeed, this RQ can be answered affirmatively. As described in Section 4.1, we have evaluated our approach and implemented prototype in five different model transformations of different nature, and we have further evaluated it with one large metamodel. All the generated models, which can be found on the project's GitHub repository (Karimi et al., 2023), conform to the corresponding metamodel and satisfy the given set of OCL constraints. Therefore, we can claim that our approach is domain-independent. The only requirement with the current implementation is that the metamodel to which models will conform must be expressed in *Ecore* and its constraints using *Eclipse OCL*.

4.6.2. RQ2 - Performance

In order to response to this research question, we have measured the execution time and memory consumption of our tool with the different models explained in Section 4.5.1. This is shown in column *ACO* of Table 3. We also measure the performance of the state-of-the-art approaches for comparison with our tool, as shown in columns *Viatra Solver* and *Random Generator* in the table. For the models shown in Table 3, no OCL constraints have been considered.

We can observe in the *ACO* column of the table that all execution times are reasonable for all metamodels and models of a variety of sizes. For instance, in the *CPL2SPL* case study, the longest time taken by our approach is 3.5 s for the generation of 500 different models, with 10 elements each, conforming to the *CPL* metamodel (which has 33 classes, 17 relationships and 42 attributes; cf. Table 2). This means that it takes an average of 7 ms to generate each model. When generating 50 models of different sizes, the average execution time when generating the smallest models (with 10 elements) is 30 ms, while the average execution time for generating the largest models (with 320 elements) is 60 ms. We can see that this difference is very small. In fact, we observe that all execution times are quite homogeneous with all the metamodels, despite these having different nature and complexity. Our hypothesis for this behavior is explained in Section 4.7.

When comparing our approach with the others in terms of execution time, we observe that the *Random Generator* outperforms our solution in most cases. However, the difference is not very big—the major difference is 2.1 s in the generation of 50 models with 320 elements for the *Grafcet* metamodel. Regarding *Viatra Solver*, it takes much longer for generating the models than our approach. Besides, in this solution, times vary more depending on the number of models to generate and the number of elements they must contain—this is normal, since it relies on a SAT solver. We can see that no numbers are added when creating models with 20 elements or more for the *Families Extended* metamodel. The reason is we started the execution, and after 5 min it had not finished, so we stopped it manually. We observed this same behavior in several different runs. From the execution times in the table we calculate that, on average, our solution is more than 50 times faster than *Viatra Solver*.

Regarding memory consumption, the numbers are quite reasonable with our approach. For instance, it only needs 22.9 MB for generating 500 models with 10 elements conforming to the *Families Extended* metamodel. Similar to the execution time values, we can see that the memory consumption is not very different in the generation of the different models for the different metamodels. Besides, our tool consumes less memory than the other two tools in all cases. *Viatra Solver* is the tool consuming more memory in most cases, although *Random Generator* is the one consuming more memory in some cases.

Let us now consider the inclusion of OCL constraints for obtaining the models, which is shown in Table 4. In all cases, 50 models are generated. As we can see in the *# Elements* column, 30 model elements are generated in all models except for models conforming to the *Families*

Table 5
Mutation analysis by case study and mutant types (cf. Section 4.2).

Transformation Name	% ACO Typing - Semantic - Syntactic	% Viatra Solver Typing - Semantic - Syntactic	% Random Generator Typing - Semantic - Syntactic
CPL2SPL	93.33 - 93.33 - 93.33	2.41 - 23.33 - 32.32	66.67 - 66.67 - 66.67
Families2Persons	100.00 - 100.00 - 76.47	86.67 - 94.54 - 76.47	86.67 - 85.45 - 76.47
Families2Persons_Extended	65.74 - 73.39 - 71.59	1.30 - 1.90 - 9.17	0.00 - 0.00 - 0.00
Grafcet2Petrinet	78.98 - 87.80 - 89.44	22.72 - 39.46 - 52.27	52.86 - 49.20 - 55.04
SOOML2SOOPL	67.33 - 73.34 - 68.33	16.38 - 22.58 - 22.92	0.00 - 0.00 - 0.00
Total	**80.49%**	**13.64%**	**34.58%**

metamodel, which contain 7 model elements. As explained in Section 4.5.2, we have defined different OCL constraints for the different metamodels. They are all available on our project's website (Karimi et al., 2023). The constraints' IDs that are considered in the generation of the models with our approach is shown in *OCL Constraints* column. Since not all these OCL constraints can be converted into VQL constraints, those that could be expressed in VQL were considered for the *Viatra Solver* executions and their IDs can be seen in column *VQL Constraints*. The *Random Generator* cannot take into account constraints expressed on the metamodels in the model generation process, reason why it is not included in the table.

Even though in the model generation with our approach we are considering more constraints than in the generation with *Viatra Solver* (columns *OCL Constraints* and *VQL Constraints*), our approach is faster in all cases except one—in the *CPL* metamodel, where 8 OCL constraints are considered versus 2 VQL constraints, our approach is 0.1 s slower. In fact, our approach can be up to 23 times faster—as in the *Grafcet* metamodel, where 5 OCL constraints versus 2 VQL constraints are considered. Regarding memory consumption, the memory consumed by our ACO approach is quite reasonable in all cases, especially considering the overload that checking constraints typically causes.

To further measure the scalability of our approach, we performed stress tests to check at which points the different tools under study would suffer from memory shortage or crash. For this, we considered the large Maude metamodel (cf. Section 4.1). The number of model elements to generate in each model was set to 1000.

We did a first experiment with our tool, in which we generated 10,000 models. The tool was able to generate 100 models per second, with a memory consumption of up to 120 MB. We continued the experiment by increasing 10 times the number of models in each execution run. Our tool was able to create models sets with an increasing number of models until we reached 1,000,000 models. At this moment, the program was unable to continue running due to an out-of-memory error. We performed the same experiment with the other two tools. *Viatra Solver* could not generate the set with 100,000 models and the *Random Generator*, like our tool, could not generate the set with 1,000,000 models; both due to memory overflow.

4.6.3. RQ3 - Usefulness

In order to determine whether our approach is able to detect faults in model transformations, we performed mutation analysis using Guerra et al.'s framework (Guerra et al., 2019) (cf. Section 4.2). For each case study and each tool under comparison, we used this framework for computing the mutation score of 30 different runs. Each of these runs takes as input one different model generated by the tool under evaluation. These models used as input have an average of 10 elements each. In these experiments, the default parameters have been selected both for ACO and *Viatra Solver*. In ACO, this means having $\alpha=1$, $\beta=0$ and $\rho=0.05$ (cf. Section 2.2.2).

Table 5 shows the results of the mutation analysis for the three tools. The framework used for this analysis (Guerra et al., 2019) outputs the mutation analysis results classified by mutant type. Let us take, for instance, the score 93.33% in the mutants of type *Typing* of the CPL2SPL case study obtained with our ACO approach—recall that 95 mutants of type *typing* were created for the CPL2SPL model transformation, as we

Table 6
Wilcoxon test—Semantic.

	ACO	RANDOM
RANDOM	2.0e−16	–
Viatra	2.0e−16	0.37

can see in the last column of Table 2. The score 93.33% means that the 30 models used in the 30 runs of the mutation analysis were able to kill an average of 88.66 *typing* mutants (out of the total of 95 mutants).

If we analyze the mutation scores for our approach, we can see that the results are quite acceptable for all case studies. All scores are above 70% except for the *typing* mutants of *Families2Persons_Extended* and the *typing* and *synatic* mutants of *SOOML2SOOPL*. In any case, all mutation scores are above 65%, and we can see that the overall mutation score is 80.49%, which is a very good number. By having a look at the mutation scores obtained with the models generated by *Viatra Solver*, we can see that results are not that good. We inspected the models obtained by *Viatra Solver* that were given as input for the mutation analysis, and observed some anomalies in the models. In particular, the root object of some models did not correspond to the root class of the metamodel, but to some inner class. For instance, in the *Families_Extended* metamodel (Oakes et al., 2018), the root class is *Country*. A *Country* is composed of *Cities* that, in turn, are composed of *Neighborhoods*. Finally, *Neighborhoods* are composed of *Schools*. We observed that the root object of some models was not of type *Country*, but of type *City*, *Neighborhood* or *School*. If the root object of a model is of type, let us say, *School*, then this model will not contain any object that is a container of *School*—in this example, it will not contain any object of type *Neighborhood*, *City* or *Country*, or any object contained by objects of these types that are not directly contained by objects of type *School* or by their contained objects. Inevitably, this reduces the object variability in the models and, therefore, the chance to kill model transformation mutants. Finally, the scores obtained with the models generated with *Random Generator* are worse than those obtained with our tool. While the mutation scores in the *Families2Persons* model transformation were quite good, with all above 75%, no mutant was killed in *Families2Persons_Extended* and *SOOML2SOOPL* model transformations. The overall mutation score was 34.58%, much smaller than our 80,49% mutation score.

Regarding the statistical tests used in our evaluation (cf. Section 4.5.5), the values of *chi-squad* per mutant type are *semantic* = 126.56, *syntactic* = 124.13 and *typing* = 106.62, all with *p-value* $< 2e{-}16$ and $df = 2$. These results entail that we would obtain meaningful results with our ACO algorithm in comparison with *Viatra Solver* and *Random Generator* solutions. This is computed by comparing the mean of the mutation scores between the five metamodels shown in Table 5. To ensure which algorithm is superior in comparison with the others, Wilcoxon rank sum test (McKight and Najab, 2010) is employed for performing pair-wise comparison between algorithms (Tables 6–8). As a summary, Table 9 shows a meaningful difference between ACO and *Viatra Solver* and between ACO and *Random Generator*. There is no significant difference between *Viatra Solver* and *Random Generator*.

Table 7
Wilcoxon test—Syntactic.

	ACO	RANDOM
RANDOM	2.0e−16	–
Viatra	2.0e−16	0.35

Table 8
Wilcoxon test—Typing.

	ACO	RANDOM
RANDOM	5.5e−15	–
Viatra	2.0e−16	0.44

Table 9
Wilcoxon test—Overall.

	ACO	RANDOM
RANDOM	2.0e−16	–
Viatra	2.0e−16	0.99

4.7. Discussion

After performing the evaluation of our approach and the comparison with state-of-the-art approaches, there are some aspects that we would like to clarify.

First, it is important to recall and highlight the aim of our contribution. Our goal was to develop an approach and easy-to-use framework for the generation of test models given a metamodel. The models generated should be optimized for model transformation testing. In the process of evaluating our tool and developed framework, a comparison with similar state-of-the-art frameworks could put the contribution of our work into context. With this in mind, we shall clarify that it was not easy to find tool prototypes or frameworks with a similar purpose as ours, i.e., we observed a lack of tools for the automatic generation of models given minimal user inputs that used standardized formats such as *XMI*. We selected the two tools that provided the closest functionality: *Viatra Solver* and *Random Generator* (cf. Sections 4.3.1 and 4.3.2). We had other candidate tools used by MDE practitioners, such as USE (Gogolla et al., 2005), but we discarded it due to the incompatibility of its inputs and outputs with EMF – for instance, USE does not accept Ecore metamodels – and to the lack of approaches proposing the generation of large model sets with USE.

Second, regarding the results obtained for RQ3, it is important to recall that our approach was tailored to model transformation testing, i.e., one of our goals was to maximize the mutation score when using the models generated by our approach. The tools with which we compare our approach were not conceived with this primary goal, so it is not surprising that our tool obtained better results (cf. Table 5). Besides, in the case of *Viatra Solver*, we observed some anomalies in the models, as we explained in Section 4.6.3. In any case, as we say, *Viatra Solver* was created with a different goal in mind, and it is primary conceived for its use with graph models. In the case of *Random Generator*, it is part of a much complex framework (Guerra et al., 2019) from which we have used its model generation functionality—recall we use another functionality of this tool for obtaining and executing all model transformation mutants in the mutation analysis. Having said this, we do not aim to replace these two tools with ours, but to complement them.

Third, we wanted to explore an algorithm, ant-colony optimization, that was – to the best of our knowledge – unexplored in the context of model generation. We are very satisfied with the results obtained, both regarding performance (cf. Section 4.6.2—RQ2) and mutation score (cf. Section 4.6.3—RQ3). We firmly believe that the framework we have developed (Karimi et al., 2023) can be used by MDE practitioners who are accustomed to work with the EMF ecosystem.

Regarding the results for performance (cf. Table 3), we observed that all execution times are quite homogeneous with our tool. Our explanation lies in the fact that ACO relies on a collective wisdom based on *Pheromone trails* (cf. Section 2.2.2). When an ant wants to select a solution, it does not search in the whole search space, but instead it considers the amount of *Pheromone trails*. Based on this and the heuristic information of the problem, a probabilistic value is calculated (cf. Eq. (1)), which will lead the ant to take the next step instead of searching the whole solution space. This way, when a metamodel gets bigger in size, the probabilistic calculation will not take much more time and the calculation complexity will remain O(1). This makes our ACO-based approach a scalable solution.

Finally, also in the reply to RQ2, we observed that our tool only crashed with the Maude metamodel when trying to generate sets with 1,000,000 models. In practice, we will rarely need such an amount of models, so our tool presented very acceptable scalability results.

4.8. Threats to validity

In the following, we describe the four types of threats that can affect the validity of current study, according to Wohlin et al. (2012).

4.8.1. Construct validity threats

Threats related to construct validity are concerned with the relationship between theory and what is observed. A possible threat could be the way we have assessed the variability in the models obtained with our approach. This assessment has been done by performing a mutation analysis—response to RQ3 (cf. Section 4.6.3). We chose to perform mutation analysis because it is the most extended technique as test adequacy criteria in the context of model transformation testing (Troya et al., 2022) for assessing generated models. Another possible way of assessing this variability could have been to compare the obtained models among them or individually computing the metamodel coverage and model transformation coverage of each generated model. However, since the goal of obtaining diverse models was to use them for model transformation testing, we considered that the best way to assess this variability in our context was through mutation analysis.

Related to the previous threat there is another one that has to do with the manner in which mutation analysis has been conducted. This analysis fully relies on Guerra et al.'s framework (Guerra et al., 2019), so we are assuming that the results yielded by this framework are correct. If they were not, large parts of our experimental results would be invalidated.

4.8.2. Conclusion validity threats

Threats to the conclusion validity are concerned with the issues that affect the ability to draw correct conclusions from the data obtained from the experiments. In our experiments, the transitory load of the machine where the performance experiments were executed can influence the performance results. To mitigate this threat, we have run all experiments 30 times on the same machine and taken the results returned by the average of the all runs.

In our evaluation, we have applied mutation analysis for assessing the capability of the different frameworks of finding bugs, so this capability has been reflected in the mutation score results. However, a threat to conclusion validity is that the mutation score might not always correlate with the capability of finding bugs. For instance, Nguyen et al. (2020) performed a study in which they could not find a direct correlation between a classical test quality metric and the capability of a fuzzing technique in finding faults.

4.8.3. Internal validity threats

These threats are related to those factors that might affect the results of our evaluation. The experiments to evaluate the usefulness of our approach require the generation of several inputs, such as model transformation mutants and appropriate oracles. We have used Guerra et al.'s framework (Guerra et al., 2019), which provides the generation of these inputs out of the box. If we had used different mutants or different oracles, we could have obtained different results. However, we think this threat is partly mitigated by the high number of mutants used in the experiments: 5021. Another internal threat can be the input parameters used for ACO and *Viatra Solver*. Having used different parameters could have drawn different results. However, to try to mitigate this threat, we have used the default parameters in both tools (cf. Section 4.6.3).

4.8.4. External validity threats

These threats have to do with the extent to which it is possible to generalize the findings of the experiments. The first threat is that the results of our experiments have been obtained with six case studies, which externally threatens the generalizability of our results. To mitigate this threat, we have tried to select a set of model transformations that differ in their domains and size of metamodels and transformation, as reflected in Table 2, and that have been used as case studies in several approaches related to model transformation testing. Another threat is that we have evaluated our approach with model transformations implemented in ATL due to its importance both in industry and academia, so it would be interesting to evaluate it with other transformation languages. However, we do believe the proposed approach would produce similar results for any model transformation language as long as they can receive *XMI* models and Ecore metamodels as input.

5. Related work

There are several approaches proposing the generation of models for model transformation testing purposes. Overall, they can be classified as black-box and white-box approaches. The former do not consider the model transformation under test in the model generation process, while the latter do. Since our approach is black-box, we only mention black-box approaches.

There exist several generators of consistent models (such as EMFtoCSP (González et al., 2012), USE (Kuhlmann et al., 2011), Formula (Jackson et al., 2015) or Clafer (Bąk et al., 2016)) that take the high-level specification of the modeling language and translate it to a logic representation and then derive consistent (graph-) models using back-end logic solvers (like KodKod Torlak and Jackson, 2007, Korat Zhong et al., 2016 or the Z3 SMT-solver De Moura and Bjørner, 2008). Despite their conceptual elegance, existing techniques only scale for tree-like models (Elkarablieh et al., 2007), but they do not scale for complex graph structures. A detailed comparison of model generator approaches and expected graph properties is provided in Varró et al. (2018), which covers graph generators developed in other disciplines (like graph databases or network science). While all these tools are able to generate consistent models, favorable scalability was only reported in Soltana et al. (2017) and Semeráth et al. (2018) to synthesize graph models with at least 1000 nodes—in our evaluation, our tool was able to generate more than 100,000 models at once with 1000 elements each.

Some approaches build on the USE framework, where models are not generated in the standard *XMI* format and input metamodels in Ecore format are not accepted. For instance, Gogolla et al. (2005) consider metamodel coverage and invariants satisfaction, and propose to use the ASSL language for the model generation. The models are not generated only from the metamodel, but according to the ASSL specification. Gogolla et al. (2015) and Hilken et al. (2018) focus on testing models and model transformations and apply the concept of classifying terms for defining equivalent classes for partitioning the source and target model spaces of the transformation. The USE tool is applied to generate object models in a goal-oriented way, where classifying terms must be manually defined, so models are not generated from scratch as in our approach. Burgueño et al. (2019) continue this line of work and propose a method to obtain classifying terms automatically. This work has a similar purpose as ours, as only the input metamodel with annotated OCL constraints is necessary. However, they do not generate a large set of models, and the generation is not completely automatic, since the domain expert is expected to decide which of the automatically identified classifying terms must be used in the generation process.

Some other lines of work also propose the generation of models, but they typically require inputs given by the domain expert. For instance, Sen et al. (2008, 2009) consider model fragments apart from the metamodel and a set of pre-conditions expressed in OCL. Guerra and Soeken (2015) drive the generation of models with a set of OCL expressions obtained from pre- and post-conditions and from invariants. Another example is WODEL (Gómez-Abajo et al., 2017), a DSL and tool for the specification and generation of model mutants. WODEL generates mutants of models that conforms to arbitrary meta-models, where seed models are needed.

Some works have specifically focus on the efficiently generation of models. For instance, Nassar et al. (2020) automate the generation of valid EMF models. The metamodel is first translated to a rule-based model transformation system and then these rules generate the models. The generation process may be further configured by the user, for instance providing initial seed models. The scalability of the approach is proved by generating large models. The objective of this approach is similar to ours, but it is addressed from a different perspective and implementation, and it is not evaluated in the context of model transformation testing. Therefore, we see this approach as complementary to ours. He et al. (2019) present an approach for efficient model generation, in which they generate models of large size in reasonable times. The major difference with our approach is that the way a valid model must be generated has to be specified with a template, for which they provide a DSL, so domain knowledge is necessary.

None of the aforementioned approaches apply search-based techniques in the generation of models, like our approach does, but there are other works that do. Rose and Poulding (2013) propose a search-based process to select a subset out of a set of random models conforming to a metamodel. This way, they aim at speeding up the testing process by relying on a smaller number of input models. The problem is they need to count on an initial set of models. Random-mutation hill climbing is used as search method. Shelburg et al.'s work (Shelburg et al., 2013) proposes to use a multi-objective search-based approach to generate input models from existing ones when the metamodel is modified, so the focus of this work is different than ours. Wang et al. (2013) propose a search-based approach for generating input models based not only on the coverage of the metamodel, but also on structural information. In particular, they propose to use a mono-objective optimization algorithm to generate input models that maximize the coverage of the metamodel while minimizing the structural distance and the number of models. The concept of structural distance is similar to ours, but the main difference with our approach is that information on structural information needs to be given as input. Finally, Alkhazi et al. (2020) propose a multi-objective approach for selecting appropriate test cases for the purpose of transformation testing. They apply the NSGA-II algorithm, but the approach is not targeted at building models from scratch.

Finally, it is worth mentioning that there are some other works related to managing models, but they have slightly different focus. For instance, Fleurey et al.'s (Fleurey et al., 2009) approach is devoted to assess the quality of a given set of input models, while the objective of our approach is precisely to obtain an optimal set of models. Another example is the work by Aranega et al. (2015), which propose to apply mutation analysis to semi-automatically improve an initial set of test models.

6. Conclusion

This paper has presented an approach to generate model sets. It is the first approach for, to the best of our knowledge, generating models by applying an ant-colony-optimization (ACO) algorithm satisfying two objectives and needing only as input the metamodel to which models must conform—together with an optional set of well-formed OCL constraints. These sets of models constitute a test model suite that the model transformation implementer or tester can use from the beginning of the implementation of a model transformation in order to test it.

Different test suites have been obtained in very reasonable times for six different case studies, and an overall mutation score of 80.49% in an evaluation with 5 different model transformations and 5021 mutants positions our approach as a very good option for its use in model transformation testing. We have compared our approach and implementation with two state-of-the-art tools, namely *Viatra Solver* and *Random Generator*, observing that our tool offers the best mutation scores. Our tool is also the best in terms of memory consumption, and it is able to generate very large model sets composed of large models in reasonable times.

As future work, we want to evaluate our tool by applying mutation analysis with model transformations written in languages different from ATL. Another line of future work is the improvement of the user interface of our framework in order to foster its usability. In fact, we want to work on creating an integrated framework with a friendly UI where users can select different algorithms and different objectives that will guide the generation of the models sets.

Verifiability

For the sake of verifiability, our prototype as well as all artifacts of the experiments are available on our project's website (Karimi et al., 2023). Detailed instructions on how to run our tool and inspect the provided files are also available there.

CRediT authorship contribution statement

Meysam Karimi: Investigation, Implementation, Writing – original draft, Writing – review & editing, Data curation. **Shekoufeh Kolahdouz-Rahimi:** Investigation, Writing – original draft, Writing – review & editing, Data curation. **Javier Troya:** Investigation, Methodology, Writing – original draft, Writing – review & editing, Data curation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

https://github.com/MeysamKarimi/MG-ACO.

Acknowledgments

This work was partially supported by the Spanish Government (FEDER/Ministerio de Ciencia e Innovación – Agencia Estatal de Investigación) under projects SoCUS [TED2021-130523B-I00] and IPSCA [PID2021-125527NB-I00], and by the Junta de Andalucia, Spain, under contract QUAL21 010UMA. We would like to thank Jesús Sánchez Cuadrado for his support using the mutation analysis tool (Guerra et al., 2019), and Oszkár Semeráth for his support with *Viatra Solver*. Their help has been key for performing the comparison study. Finally, we thank Prof. Antonio Vallecillo for his comments and very valuable feedback on an earlier draft of this paper.

References

Alhwikem, F.H.M., Paige, R.F., Rose, L.M., Alexander, R.D., 2016. A systematic approach for designing mutation operators for MDE languages. In: MoDeVVa@MoDELS 2016. pp. 54–59.

Alkhazi, B., Abid, C., Kessentini, M., Leroy, D., Wimmer, M., 2020. Multi-criteria test cases selection for model transformations. ASE 27, 91–118.

Aranega, V., Mottu, J.-M., Etien, A., Degueule, T., Baudry, B., Dekeyser, J.-L., 2015. Towards an automation of the mutation analysis dedicated to model transformation. STVR 25 (5–7), 653–683.

Arcuri, A., Briand, L., 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proc. of ICSE. pp. 1–10.

ATL, 2006. ATL Zoo. http://www.eclipse.org/atl/atlTransformations.

Bąk, K., Diskin, Z., Antkiewicz, M., Czarnecki, K., Wąsowski, A., 2016. Clafer: Unifying class and feature modeling. SoSyM 15 (3), 811–845.

Balaban, M., Bennett, P., Doan, K.-H., Georg, G., Gogolla, M., Khitron, I., Kifer, M., 2016. A comparison of textual modeling languages: OCL, alloy, FOML. In: OCL@ MoDELS. pp. 57–72.

Barán, B., Schaerer, M., 2003. A multiobjective ant colony system for vehicle routing problem with time windows. In: Applied Informatics. pp. 97–102.

Batot, E., Sahraoui, H., 2016. A generic framework for model-set selection for the unification of testing and learning MDE tasks. In: Proc. of MoDELS. pp. 374–384.

Baudry, B., Dinh-Trong, T., Mottu, J.-M., Simmonds, D., France, R., Ghosh, S., Fleurey, F., Le Traon, Y., 2006. Model transformation testing challenges. In: ECMDA Workshops.

Baudry, B., Ghosh, S., Fleurey, F., France, R., Le Traon, Y., Mottu, J.-M., 2010. Barriers to systematic model transformation testing. CACM 53 (6), 139–143.

Bill, R., Fleck, M., Troya, J., Mayerhofer, T., Wimmer, M., 2019. A local and global tour on MOMoT. SoSyM 18 (2), 1017–1046.

Brambilla, M., Cabot, J., Wimmer, M., 2017. Model-Driven Software Engineering in Practice, Second Ed..

Brottier, E., Fleurey, F., Steel, J., Baudry, B., Traon, Y.L., 2006. Metamodel-based test generation for model transformations: An algorithm and a tool. In: Proc. of ISSRE. pp. 85–94.

Burdusel, A., Zschaler, S., John, S., 2019. Automatic generation of atomic consistency preserving search operators for search-based model engineering. In: Proc. of MODELS. IEEE, pp. 106–116.

Burdusel, A., Zschaler, S., Strüber, D., 2018. MDEOptimiser: A search based model engineering tool. In: Proc. of MoDELS. pp. 12–16.

Burgueño, L., Cabot, J., Clarisó, R., Gogolla, M., 2019. A systematic approach to generate diverse instantiations for conceptual schemas. In: Proc. of ER. Springer, pp. 513–521.

Burgueño, L., Troya, J., Wimmer, M., Vallecillo, A., 2015. Static Fault Localization in Model Transformations. IESEDJ 41 (5), 490–506.

Burnstein, I., 2006. Practical Software Testing: A Process-Oriented Approach. Springer Science & Business Media.

Calegari, D., Delgado, A., 2013. Rule chains coverage for testing QVT-relations transformations. In: Proc. of AMT@MoDELS.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C., 2007. All about maude - a high-performance logical framework: How to specify, program and verify systems in rewriting logic. Springer, Berlin, Heidelberg.

Cuadrado, J.S., Guerra, E., de Lara, J., 2017. Static analysis of model transformations. IEEE TSE 43 (9), 868–897.

Czarnecki, K., Helsen, S., 2006. Feature-based survey of model transformation approaches. IBM Syst. J. 45 (3), 621–645.

da Silva, A.R., 2015. Model-driven engineering: A survey supported by the unified conceptual model. CLSS 43, 139–155.

De Moura, L., Bjørner, N., 2008. Z3: An efficient SMT solver. In: Proc. of TACAS. pp. 337–340.

Deb, K., 2014. Multi-objective optimization. In: Search Methodologies. Springer, pp. 403–449.

Derasari, R., 2021. Adding Formal Specifications to A Legacy Code Generator (Ph.D. thesis). Eindhoven University of Technology.

Doerner, K., Gutjahr, W.J., Hartl, R.F., Strauss, C., Stummer, C., 2004. Pareto ant colony optimization: A metaheuristic approach to multiobjective portfolio selection. Ann. Opera. Res. 131 (1–4), 79–99.

Doerner, K., Hartl, R., Reimann, M., 2003. Are COMPETants more competent for problem solving?–the case of a multiple objective transportation problem. CEJOR 11 (2), 115–141.

Dorigo, M., 1992. Optimization, Learning and Natural Algorithms (Ph.D. thesis). Politecnico di Milano, Italy.

Efstathiou, D., Williams, J.R., Zschaler, S., 2014. Crepe complete: Multi-objective optimization for your models. In: CMSEBA@ MoDELS. pp. 25–34.

Elkarablieh, B., Zayour, Y., Khurshid, S., 2007. Efficiently generating structurally complex inputs with thousands of objects. In: Proc. of ECOOP. pp. 248–272.

Finot, O., Mottu, J.-M., Sunyé, G., Degueule, T., 2013. Using meta-model coverage to qualify test oracles. In: Proc. of AMT. pp. 1613–0073.

Fleck, M., Troya, J., Kessentini, M., Wimmer, M., Alkhazi, B., 2017. Model transformation modularization as a many-objective optimization problem. IEEE TSE 43 (11), 1009–1032.

Fleck, M., Troya, J., Wimmer, M., 2015. Marrying search-based optimization and model transformation technology. In: Proc. of NasBASE. pp. 1–16.
Fleck, M., Troya, J., Wimmer, M., 2016a. Search-based model transformations. J. Soft-Evol. Proc. 28 (12), 1081–1117.
Fleck, M., Troya, J., Wimmer, M., 2016b. Search-based model transformations with MOMoT. In: Proc. of ICMT. pp. 79–87.
Fleurey, F., Baudry, B., Muller, P.-A., Le Traon, Y., 2009. Qualifying input test data for model transformations. SoSyM 8 (2), 185–203.
Fleurey, F., Steel, J., Baudry, B., 2004. Validation in model-driven engineering: Testing model transformations. In: Proc. of MODEVVA. pp. 29–40.
Gogolla, M., Bohling, J., Richters, M., 2005. Validating UML and OCL models in USE by automatic snapshot generation. SoSyM 4 (4), 386–398.
Gogolla, M., Vallecillo, A., 2011. Tractable model transformation testing. In: Proc. of ECMFA. pp. 221–235.
Gogolla, M., Vallecillo, A., Burgueño, L., Hilken, F., 2015. Employing classifying terms for testing model transformations. In: Proc. of MoDELS. pp. 312–321.
Gómez-Abajo, P., Guerra, E., de Lara, J., 2017. A domain-specific language for model mutation and its application to the automated generation of exercises. CLSS 49, 152–173.
González, C.A., Büttner, F., Clarisó, R., Cabot, J., 2012. EMFtoCSP: A tool for the lightweight verification of EMF models. In: Proc. of FormSERA. pp. 44–50.
Goss, S., Aron, S., Deneubourg, J.-L., Pasteels, J.M., 1989. Self-organized shortcuts in the Argentine ant. Naturwissenschaften 76 (12), 579–581.
Guerra, E., Cuadrado, J.S., de Lara, J., 2019. Towards Effective Mutation Testing for ATL. In: Proc. of MoDELS. pp. 78–88.
Guerra, E., Soeken, M., 2015. Specification-driven model transformation testing. Softw. Syst. Model. 14, 623–644.
Harman, M., Jones, B.F., 2001. Search-based software engineering. Inform. Softw. Technol. 43 (14), 833–839.
He, X., Zhang, T., Pan, M., Ma, Z., Hu, C.-J., 2019. Template-based model generation. Softw. Syst. Model. 18, 2051–2092.
Hegedüs, Á., Horváth, Á., Varró, D., 2015. A model-driven framework for guided design space exploration. ASE 22 (3), 399–436.
Hilken, F., Gogolla, M., Burgueño, L., Vallecillo, A., 2018. Testing models and model transformations using classifying terms. SoSyM 17 (3), 885–912.
Holland, J.H., et al., 1992. Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. MIT Press.
Hong, L., Drake, J.H., Woodward, J.R., Özcan, E., 2018. A hyper-heuristic approach to automated generation of mutation operators for evolutionary programming. Appl. Soft Comput. 62, 162–175.
HoseinDoost, S., Karimi, M., Rahimi, S.K., Zamani, B., 2018. Solving the quality-based software-selection and hardware-mapping problem with ACO. In: Proc. of TTC@STAF. pp. 19–30.
Jackson, E.K., Levendovszky, T., Balasubramanian, D., 2015. Automatically reasoning about metamodeling. SoSyM 14 (1), 271–285.
Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., 2008. ATL: A model transformation tool. Sci. Comput. Programm. 72 (1–2), 31–39.
Jouault, F., Bézivin, J., Consel, C., Kurtev, I., Latry, F., (2006). Building DSLs with AMMA/ATL, a Case Study on SPL and CPL Telephony Languages. In: Proc. of ECOOP-DSPD.
Karimi, M., Kolahdouz-Rahimi, S., Troya, J., 2023. Model Generation for model transformation testing applying ACO. GitHub, https://github.com/MeysamKarimi/MG-ACO.
Kuhlmann, M., Hamann, L., Gogolla, M., 2011. Extensive validation of OCL models by integrating SAT solving into USE. In: Proc. of TOOLS. pp. 290–306.
Kühne, T., 2006. Matters of (meta-) modeling. SoSyM 5 (4), 369–385.
López-Ibáñez, M., Stützle, T., 2012. An experimental analysis of design choices of multi-objective ant colony optimization algorithms. Swarm Intell. 6 (3), 207–232.
Ludewig, J., 2003. Models in software engineering – an introduction. SoSyM 2, 5–14.
Maniezzo, V., Carbonaro, A., 1999. Ant colony optimization: An overview. In: Essays and Surveys in Metaheuristics. Kluwer Academic Publishers, pp. 21–44.
McKight, P.E., Najab, J., 2010. Kruskal-Wallis test. Corsini Encyclop. Psychol. 1.
Mellor, S.J., Scott, K., Uhl, A., Weise, D., Soley, R.M., 2004. MDA Distilled: Principles of Model-Driven Architecture. O'Reilly.
Mengerink, J., Serebrenik, A., Schiffelers, R.R., Van Den Brand, M., 2016. A complete operator library for DSL evolution specification. In: Proc. of ICSME. pp. 144–154.
Mitchel, G., O'Donoghue, D., Barnes, D., McCarville, M., 2003. GeneRepair–a repair operator for genetic algorithms. In: Proceedings of the Gecco. Late Breaking Papers. pp. 235–239.
Mottu, J.-M., Baudry, B., Le Traon, Y., 2006. Mutation analysis testing for model transformations. In: Proc. of ECMDA-FA. pp. 376–390.
Nassar, N., Kosiol, J., Kehrer, T., Taentzer, G., 2020. Generating large EMF models efficiently. In: Proc. of FASE. Cham, pp. 224–244.
Nguyen, H.L., Nassar, N., Kehrer, T., Grunske, L., 2020. MoFuzz: A fuzzer suite for testing model-driven software engineering tools. In: Proc. of ASE. pp. 1103–1115.
Oakes, B.J., Troya, J., Lúcio, L., Wimmer, M., 2018. Full Contract Verification for ATL using Symbolic Execution. SoSyM 17 (3), 815–849.
Rose, L.M., Poulding, S., 2013. Efficient probabilistic testing of model transformations using search. In: Proc. of CMSBSE. p. 16–21.
Saidani, I., Ouni, A., Chouchen, M., Mkaouer, M.W., 2020. Predicting continuous integration build failures using evolutionary search. IST 128, 106392.
Selim, G.M., Cordy, J.R., Dingel, J., 2012. Model transformation testing: The state of the art. In: Proc. of AMT. pp. 21–26.
Semeráth, O., Babikian, A.A., Pilarski, S., Varró, D., 2019. VIATRA solver: A framework for the automated generation of consistent domain-specific models. In: Proc. of ICSE COmpanion. pp. 43–46.
Semeráth, O., Farkas, R., Bergmann, G., Varró, D., 2020. Diversity of graph models and graph generators in mutation testing. STTT 22 (1), 57–78.
Semeráth, O., Nagy, A.S., Varró, D., 2018. A graph solver for the automated generation of consistent domain-specific models. In: Proc. of ICSE. pp. 969–980.
Semeráth, O., Varró, D., 2018. Iterative generation of diverse models for testing specifications of DSL tools. In: Proc. of FASE. pp. 227–245.
Sen, S., Baudry, B., Mottu, J.M., 2008. On combining multi-formalism knowledge to select models for model transformation testing. In: Proc. of ICST. pp. 328–337.
Sen, S., Baudry, B., Mottu, J.-M., 2009. Automatic model generation strategies for model transformation testing. In: Proc. of ICMT. p. 148–164.
Shariat Yazdi, H., Angelis, L., Kehrer, T., Kelter, U., 2016. A framework for capturing, statistically modeling and analyzing the evolution of software models. JSS 118, 176–207.
Shelburg, J., Kessentini, M., Tauritz, D.R., 2013. Regression testing for model transformations: A multi-objective approach. In: Proc. of SSBSE. In: Lecture Notes in Computer Science, vol.8084, pp. 209–223.
Soltana, G., Sabetzadeh, M., Briand, L.C., 2017. Synthetic data generation for statistical testing. In: Proc. of ASE. pp. 872–882.
Strüber, D., 2017. Generating efficient mutation operators for search-based model-driven engineering. In: Proc. of ICMT. pp. 121–137.
Torlak, E., Jackson, D., 2007. Kodkod: A relational model finder. In: Proc. of TACAS. pp. 632–647.
Troya, J., Bergmayr, A., Burgueno, L., Wimmer, M., 2015. Towards systematic mutations for and with ATL model transformations. In: Proc. of Mutation Workshop @ ICST. pp. 1–10.
Troya, J., Segura, S., Burgueño, L., Wimmer, M., 2022. Model transformation testing and debugging: A survey. ACM CSUR 55 (4), 1–39.
Troya, J., Segura, S., Parejo, J., Ruiz-Cortés, A., 2018a. Spectrum-based fault localization in model transformations. TOSEM 27 (3), 13:1–13:50.
Troya, J., Segura, S., Ruiz-Cortés, A., 2018b. Automated inference of likely metamorphic relations for model transformations. JSS 136, 188–208.
Troya, J., Vallecillo, A., 2011. A rewriting logic semantics for ATL. JOT 10, 5:1–29.
Varró, D., Semeráth, O., Szárnyas, G., Horváth, Á., 2018. Towards the automated generation of consistent, diverse, scalable and realistic graph models. In: Graph Transformation, Specifications, and Nets. Springer, pp. 285–312.
Veen, B.v., Emmerich, M., Yang, Z., Bäck, T., Kok, J., 2013. Ant colony algorithms for the dynamic vehicle routing problem with time windows. In: Proc. of IWINAC. pp. 1–10.
Wang, W., Kessentini, M., Jiang, W., 2013. Test cases generation for model transformations from structural information. In: Proc. of MoDELS Workshops. pp. 42–51.
Wang, J., Kim, S.-K., Carrington, D., 2006. Verifying metamodel coverage of model transformations. In: Proc. of ASWEC. pp. 10–pp.
Warmer, J.B., Kleppe, A.G., 2003. The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley Professional.
Williams, J.R., 2013. A Novel Representation for Search-Based Model-Driven Engineering (Ph.D. thesis). University of York.
Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., 2012. Experimentation in Software Engineering. Springer.
Zhong, H., Zhang, L., Khurshid, S., 2016. Combinatorial generation of structurally complex test inputs for commercial software applications. In: Proc. of FSE. pp. 981–986.

Meysam Karimi is Ph.D. student at University of Isfahan. He is also a tech lead with more than 12 years of experience in the software industry. His interests are Algorithms and MDE.

Shekoufeh Kolahdouz-Rahimi is a senior lecturer at University of Roehampton. She was a lecturer at University of Isfahan (2013-2022). She completed her Ph.D. in Computer Science at Kings College London in 2013. She has published over 70 papers in international journals and conferences. Her current research interests include MDE and domain-specific languages.

Javier Troya is Associate Professor at the Universidad de Málaga, Spain. Before, he was Assistant Professor at the Universidad de Sevilla, Spain (2016-2020), and a post-doctoral researcher in the TU Wien, Austria (2013-2015). He obtained his International Ph.D. with honors at the Unviersidad de Málaga, Spain (2013). His current research interests include MDE, Software Testing and Digital Twins.