Yekta: A low-code framework for automated test models generation

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

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

Abstract

The methodology under the term model-based software engineering (MBSE) gained importance already around 20 years ago, after the publication of the MDA initiative by the OMG. This development methodology continues to evolve, giving rise to recent proposals such as *low-code* or *no-code*. Something that has not changed, as recent surveys point out, is the need for powerful testing approaches and tools for these new methodologies. In MBSE, test inputs are models, so it is key to have frameworks for model generation. However, the main shortcomings of existing model-generation frameworks are their performance limitations and the need for domain-specific knowledge, which seriously hampers their industrial adoption. In this paper, we present the Yekta low-code framework that allows to generate models in a simple way through the application of metaheuristic algorithms.

Keywords: Model-Based Software Engineering, Automated Model Generation, Automated Testing, Meta-heuristic Algorithms

1. Motivation and significance

The main principle of Model-Based Software Engineering (MBSE) for software development is to raise the level of abstraction so that models become the central artifacts, therefore minimizing the development of platform-specific code. Despite MBSE gained importance already around 20 years ago [1, 2], after the publication of the Model-Driven Architecture (MDA) initiative by the Object Management Group (OMG) [3], today many practitioners still aim at adopting its principles and many researchers make contributions in this field or apply this methodology to different software engi-

*Corresponding author

neering fields. An indicator of the adoption of MBSE is the recent definition of new terms that refer to very similar concepts, which can be interpreted as a way of modernizing the methodology and adapting it to the times. Examples of these include the terms *low-code* or *no-code* [4], which are applied to platforms [5] or programming models [6].

When developing software, testing is a crucial activity. An important phase of software testing is the generation of program inputs [7]. In MBSE, these inputs are also models, referred to as *test models* [8], and therefore the generation of models becomes a critical task [9]. Although various methods and tools have been proposed for model generation [10, 11, 12], a major shortcoming is the expert knowledge required to identify the artefacts needed for the problem at hand. Examples are sets of model fragments (they specify parts of the metamodel that should be instantiated with particular values that are interesting for testing) [12, 13, 14] or pre-conditions [11], as well as initial sets of models to be later improved [15, 16]. The use of specific notations or formalisms is yet another limitation to the usability of experimental models created by some of these approaches [17, 10, 16].

Models must conform to a metamodel, and the generation of test models conforming to a given metamodel presents a considerable challenge, even for domain experts, due to the peculiarities the models should have for their testing purposes (which objects to include in the models, which values to select for their attributes, which associations...) [8, 18]. Some of the existing model generators (such as EMFtoCSP [19], USE [20], Formula [21] or Clafer [22]) translate the modeling language to a logic representation and then derive consistent graph models using back-end logic solvers (like KodKod [23], Korat [24] or the Z3 SMT-solver [25]). However, existing techniques only scale for tree-like models [26], but not for complex graph structures. Other approaches rely on search-based algorithms for model generation [27, 16, 28, 9, 29], but, as previously mentioned, their major limitation is the domain knowledge required.

Recently, we proposed an approach for model generation applying an Ant-Colony Optimization (ACO) algorithm [30]. This generator was oriented towards model transformation testing (cf. Section 2.1), since it is one of the most common use cases of model generators [9], so that models should maximize internal and external diversity [31] (cf. Section 2.3). We evaluated its performance in the model generation process and its effectiveness for detecting faults in model transformations and compared these properties with some state-of-the-art approaches and tools (*Viatra Solver* [32] and a *Random Generator* [33]), obtaining very good results. The only input needed in this approach is the metamodel to which the generated models must conform and

an optional set of constraints defined in the Object Constraint Language[1] (OCL) [34] that the generated models must satisfy—OCL is a declarative language describing rules applying to any OMG metamodel including UML.

Following on that work, this paper presents an easy-to-use framework for model generation, named Yekta, that integrates (i) the ACO approach explained above, (ii) a new developed approach that applies a Gray-Wolf Optimizer (GWO), and (iii) the random generator presented in [33]. Following OMG's MDA initiative [3] and for usability of the models generated by our framework, Yekta generates the models in the XMI (XML Metadata Interchange) standard.

2. Background

2.1. Model-Based Software Engineering

The *Model-Based Software Engineering* (MBSE) methodology prioritizes models as key elements in software engineering. It aims to enhance productivity through automation and interoperability, streamlining design, and fostering stakeholder communication. MDE principles are increasingly popular, particularly in embedded and production systems development [35, 36].

In MBSE, *models* serve as abstractions of systems, aiding in management, understanding, and analysis [37, 38]. Models typically adhere to a metamodel, which defines their structure. *Metamodels* are special models conforming to a meta-metamodel and specify language concepts, relationships, and structural rules [39, 35]. *Model transformations* are crucial in MBSE [40]. They are software artifacts that take one more models as input and generate models or pieces of code as output. If a model transformation contains errors, these errors will be likely propagated to the generated models or code, so testing and debugging model transformations, like any other software artifact, is important [9]. Testing model transformations involves utilizing a set of test models [13, 14], so automating their generation is key [9].

2.2. Search-Based Software Engineering

Search-Based Software Engineering (SBSE), introduced by Harman and Jones [41], addresses optimization problems in software engineering using search-based techniques. In SBSE, problems are framed as search problems, where candidate solutions are explored to find optimal or near-optimal solutions. Guided by a fitness function, the search evaluates and ranks candidate

[1] `https://www.omg.org/spec/OCL/`

solutions [42]. SBSE comprises two main components: defining candidate solutions within a search space and specifying the fitness function to guide the search process.

Multi-objective optimization problems, also called Pareto-optimization problems, involve multiple objective functions that often conflict with each other [43]. Minimizing cost while maximizing performance is a typical example. In these problems, there is not usually a single optimal solution; instead, there are multiple Pareto-optimal solutions. A solution is considered non-dominated if there exists no other solution that improves upon it in at least one objective. All non-dominated solutions are equally good and termed Pareto-optimal solutions. Some metaheuristic algorithms, such as Genetic Algorithms, Particle Swarm Optimization and Grey-Wolf Optimizer (GWO), begin with random populations and then update them through operations like crossover and mutation operations. Other algorithms, like Ant-Colony Optimization (ACO), propose multi-agent solutions that construct solutions iteratively, exploring the search space by creating new combinations in a greedy randomized manner.

2.3. Application of metaheuristic algorithms for model generation

The possible number of model instances given a metamodel is infinite. For this reason, when applying metaheuristic algorithms for model generation, we need to select proper objectives so that a small number of models can represent different variations in the metamodel and cover it as much as possible. Varró et al. propose to aim for the objectives of (i) maximizing internal diversity and (ii) maximizing external diversity [44, 45]. The *internal diversity* of a model M_i is defined as the number of *(direct)* types used from its metamodel (MM): M_i is more diverse than M_j if more types of MM are used in M_i than in M_j [44]. Regarding *external diversity*, according to Sémérath and Varró [45], it captures the distance between pairs of models. External diversity among models can be enhanced by examining all possible 2-tuple combinations within a model set. By comparing instances of a type or coverage criterion across different models, external diversity between them is amplified if there are discrepancies in their structures.

Recently, we introduced a method for creating models implementing an ACO algorithm [30]. This approach was specifically designed for model transformation testing, as it represents a prevalent application of model generators [9]. The objectives were maximizing internal and external diversity of the generated models. We assessed its performance in the model generation process and its efficacy in identifying faults in model transformations. To gauge its effectiveness, we compared it with two state-of-the-art approaches

and tools, namely *Viatra Solver* [31] and a *Random Generator* [33]. The outcomes were highly promising regarding (i) its execution time, (ii) the memory consumed, and (iii) its efficacy in detecting faults in model transformations. The results were better compared with those from Viatra Solver and the Random Generator. Notably, this approach only requires the metamodel that the generated models must conform to, along with an optional set of OCL constraints that the models must satisfy.

In this work, we integrate the metaheuristic algorithms ACO and GWO into Yekta, an easy-to-use framework for model generation. ACO is inspired by the foraging behavior of real ants [46], where ants create the models based on chemical trails they leave, called *pheromones* (denoted as α), as well as on heuristic information (denoted as β). GWO, in turn, simulates a leadership structure and the hunt mechanisms used by gray wolves [47].

Our tool allows the user to optionally configure some useful parameters for the models generated that can vary depending on the models' purpose, such as the number of elements each model must have or the number of models to generate (cf. Section 3.2). This makes Yekta a versatile tool that can be used with different intentions, being the testing of MBSE artifacts one of its main purposes.

3. Software description

Our Yekta Framework is implemented as a Maven Java Project ready to be executed from any IDE, such as IntelliJ or Eclipse. The only requirement is to have installed Java (JDK Release 11[2]). Once the Yekta Framework project is cloned from GitHub and opened, all dependencies must be automatically set, so the project is ready to run.

3.1. Software architecture

The software architecture of our Yekta Framework based on its classes and packages is displayed in Figure 1. The Framework consists of six packages whose purpose and classes are described next.

- `com.dimio`: It contains the entry-point classes and the components of the user interface. Its main class, `AlgorithmUI`, uses Java Swing to design the application's user interface.

[2] `https://docs.oracle.com/en/java/javase/11/install/#Java-Platform%2C-Standard-Edition`

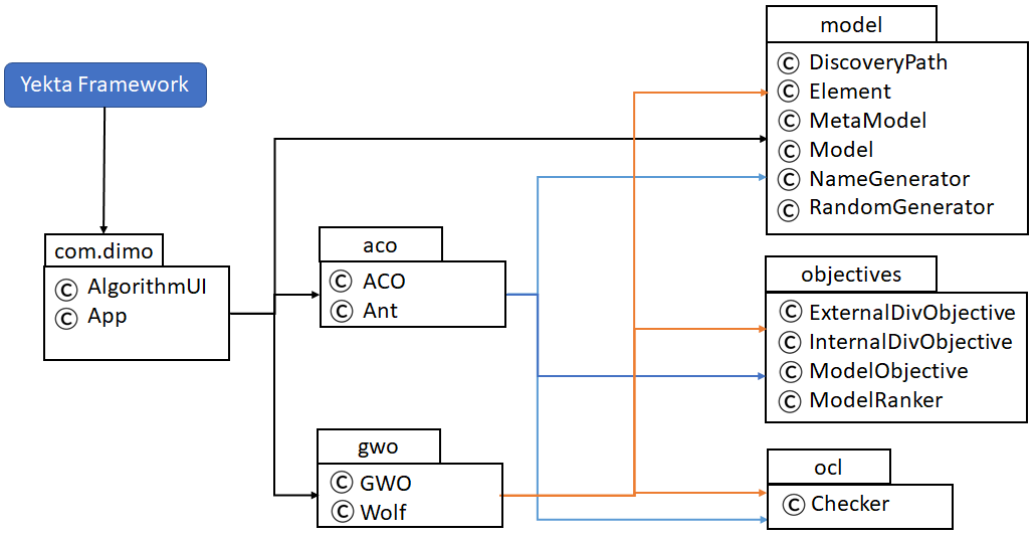


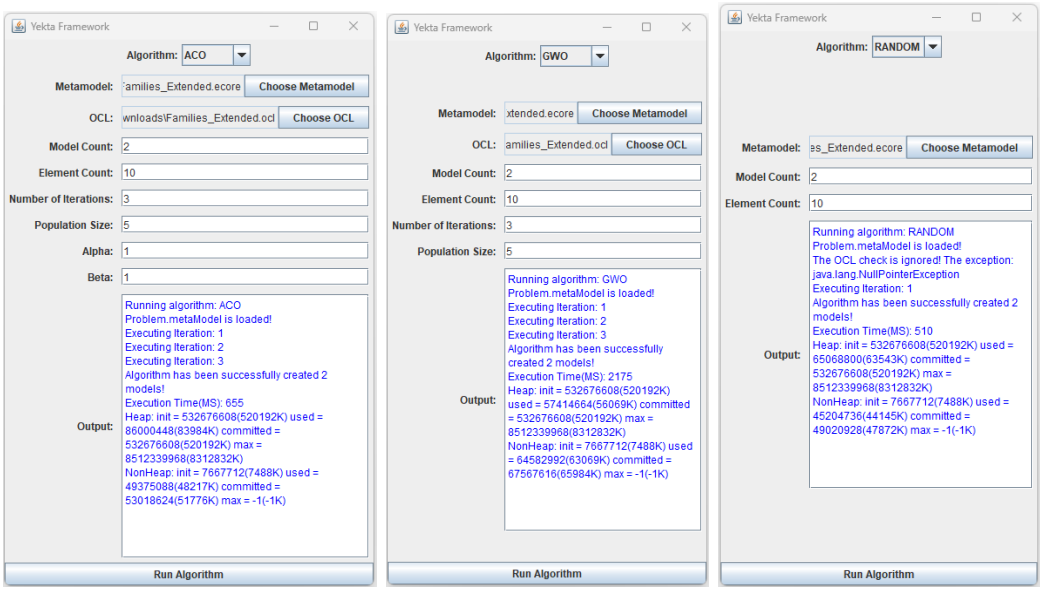
Figure 1: The structure of the packages and classes in the Yekta Framework.

- `com.dimio.aco` and `com.dimio.gwo`: These packages contain the classes that implement the ACO and GWO algorithms to generate models, respectively. The `ACO/GWO` class receives the necessary parameters of the ACO/GWO algorithm and deals with the generation of the models by relying on the `Ant/Wolf` class, which acts as the main agent responsible for the generation of the models.

- `com.dimio.model`: This package contains the logic that the various algorithms need to efficiently generate Ecore models. Class `DiscoveryPath` detects all the possible meta-objects that can be synthesized from the metamodel and, based on the selected algorithm, it selects the next item to synthesize. Class `Element` deals with the creation of `EObjects` in the models, which will be instances of class `Model`, while class `Metamodel` is used for navigating the metamodel. Finally, classes `NameGenerator` and `RandomGenerator` are used for giving meaningful names and numbers to specific attributes in the models.

- `com.dimio.objectives`: This package implements the objectives used in the solution search. Classes `ExternalDivObjective`, `Internal DivObjective` and `ModelObjetive` deal with the calculation of the two objectives explained in Section 2.3. The other important class in this package is `ModelRanker`, which ranks ants and wolves according to the objectives.

- `com.dimio.ocl`: The class in this package checks whether the optional input set of OCL constraints is satisfied by the models under generation by the different algorithms [30].



(a) ACO (b) GWO (c) Random

Figure 2: Execution with `Families_Extended` metamodel for the three different algorithms.

3.2. Software usability

The Yekta framework provides an easy-to-use interface for model generation with no deep domain knowledge, so it can be specially useful to low-code newcomers. Figures 2(a), 2(b) and 2(c) present the execution of the tool selecting the ACO, GWO and Random options, respectively. All the parameters that can be specified are displayed in Table 1, where it is explained their measurement, their type and their default value. We can see that the first three parameters are input to all algorithms, the following three parameters are input to ACO and GWO, and the last two parameters are input only to ACO.

In all three executions of the figure, we have selected the `Families_Extended` metamodel (cf. Section 4). The ACO and GWO algorithms allow to include a file with OCL constraints. Depending on the algorithm we select, the tool allows to enter the corresponding inputs, as shown in the table. If we select Random, we must indicate the number of models to generate (*Model Count*) and the number of elements to be included in each model (*Element Count*). If we select GWO, we can also specify the number of iterations to be run and the population size, i.e., the number of agents (*wolves* in this case), which must be at least the same as the number of models to generate. Finally, if we select ACO, then population size represents the number of *ants*, and we also need to specify the parameters α and β

Table 1: Yekta input parameters.

Algorithm		Paremeter	Measurement	Type	Default
ACO GWO	Random	Metamodel	File with the Ecore metamodel to which models will conform	Ecore	User input
		Model Count	Number of models to be generated	Int	User input
		Element Count	Number of elements to be included in each of the generated models	Int	User input
		OCL	File with the OCL constraints that the generated models will satisfy	OCL	User input
		# Iterations	Since we are implementing metaheuristic algorithms, this parameter specifies the number of iterations to execute	Int	10
		Population Size	Number of agents (*ants* or *wolves*) used	Int	>= model count
		Alpha	It controls the influence of the pheromone trails in the decisions taken by the ants	Double	1.0
		Beta	It regulates the heuristic information	Double	1.0

that control the influence of the pheromones and the heuristic information on the search (cf. Section 2.3) [30], which can be left as 1 as default. In all three cases, by pressing the *Run Algorithm* button we execute the algorithm, whose progress is shown in the *Output* panel, together with its memory usage and execution time. Once the algorithm terminates, the models are created in XMI format and stored in the *models* folder of the project.

4. Illustrative Example

To show some models generated by our framework, we use as input the *Families_Extended* metamodel shown in Figure 3. This metamodel was initially proposed in [48] as the input metamodel of the *Families-to-Persons_Extended* model transformation, which has also been used in other recent works [30, 49]. We choose this metamodel because its concepts are easily understandable and because it has a certain degree of complexity in terms of number of classes and associations. Having a look at Figure 3, this metamodel has the *Country* class as root element. A *Country* is made up of *companies*, *families* and *cities*. A *Family* has a *lastName*, is *registeredIn* a *Neighborhood* and can have any number of *mothers* and *fathers*, who are *Parent*s and may, in turn, work in (*worksIn*) a *Company*. It can also contain any number of *sons* and *daughters*, who are *Child*ren, and every child *goesTo* a *School*. Both parents and children are *Member*s that have a *firstName*, belong to a *family* and each of them *livesIn* a *City*. Any *City* may contain *companies*, and a *Company*, in turn, can be present in (relationship *isIn*) several distinct cities. A *City* is composed of *neighborhoods*, and these can have *schools*, where several *students* are registered. Every *School* has *Service*s, and these may be *special*, for students with special needs, or simply offer *ordinary* services. Finally, countries, cities, companies, neighborhoods and schools have a *name* attribute, which is inherited from the abstract *NamedElement* class.

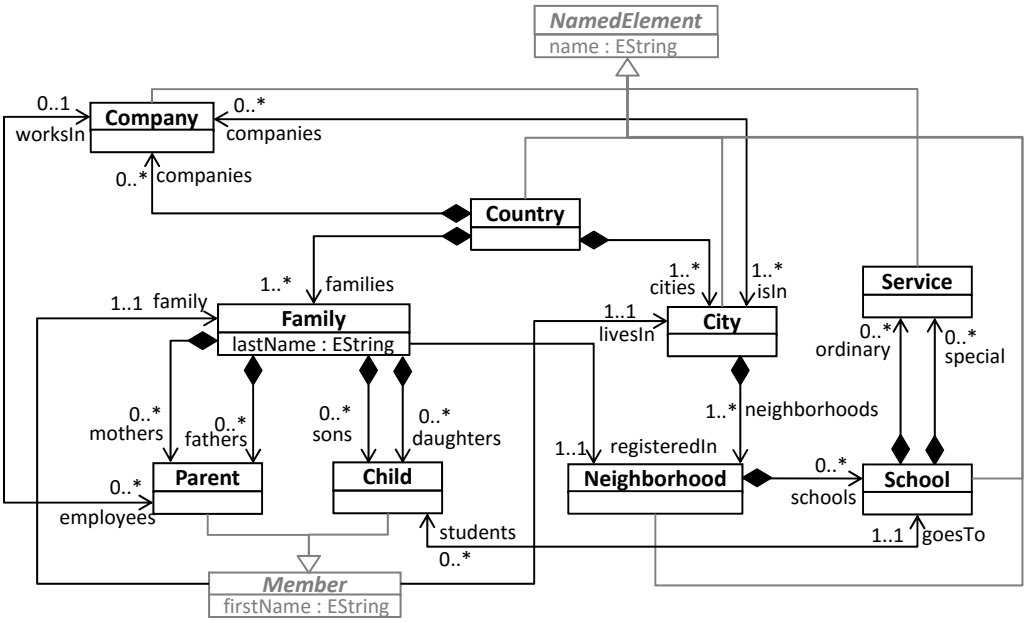


Figure 3: `Families_Extended` metamodel.

In order to ensure that our framework is able to work with constraints imposed on the metamodel, we have defined the set of OCL constraints displayed in Listing 1.

Listing 1: Examples of OCL constraints the for `Families_Extended` metamodel.

```
context Family inv familiesLastNameLengthMustBeAtLeastTwoCharacters:
        self.lastName.size() > 1
context Country inv HasAtLeastTwoCities:
        self.cities->size() > 1
context Country inv CountryMustHaveAtLeastThreeFamilies:
        self.families->size() > 2
context Country inv CountryMustHaveAtLeastOneCompany:
        self.companies->size() > 0
context Parent inv WorkInOneCompany:
        Parent.allInstances()->size() > 0 implies self.worksIn->size() = 1
```

Figures 4 and 5 display the models generated when selecting ACO and GWO algorithms, respectively, where we can see that the two models satisfy all OCL constraints—it is displayed with the default tree view available in the Eclipse IDE. Both algorithms are using 5 agents to create one model with 10 elements in three iterations, where the influence parameters of ACO are set with their default values: $\alpha = 1$ and $\beta = 1$.

As an evaluation of our framework, Table 2 displays the results of a performance experiment where different numbers of models with different sizes (in terms of number of elements) are generated with the two algorithms of our tool as well as with Viatra Solver [32], using the same input metamodel as before. Having $\#Models = 50$ and $\#Elements = 40$ indicates the generation of 50 models with

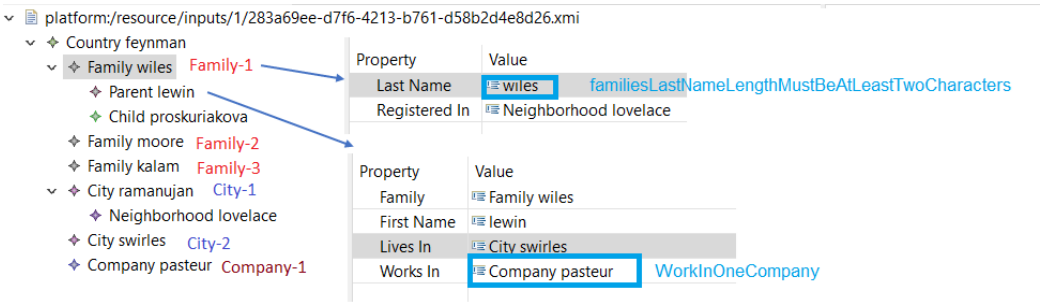


Figure 4: Model generated using ACO.

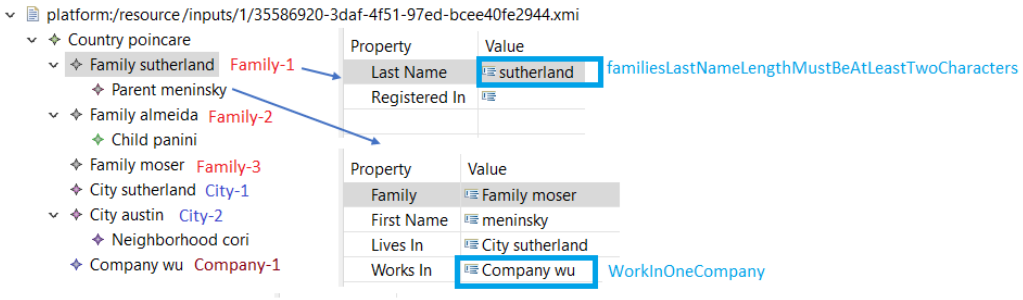


Figure 5: Model generated using GWO.

40 elements each at once—in the first and last three rows, the number of models and elements, 50 and 10 respectively, are fixed. We can see that Viatra Solver cannot deal with the generation of models with 40 elements or more. Regarding execution time, ACO and GWO are much faster than Viatra Solver, while in terms of memory consumption ACO clearly performs the best.

The reason why ACO and GWO perform better than Viatra Solver mainly lies in the fact that the latter uses a SAT solver for generating the models. Unlike SAT solvers, which are designed to find satisfying assignments meeting all constraints and can be computationally intensive, ACO and GWO focus on optimization, striking a balance between exploration and exploitation. This balance prevents them from getting stuck in local optima, accelerating the convergence process. Additionally, ACO and GWO are highly adaptable to specific problems, enabling them to exploit domain-specific knowledge more effectively. On the contrary, SAT solvers often need large data structures to handle complex logical constraints, which can slow down their performance.

Finally, we summarize the results of an evaluation to check the efficacy of the models generated to detect faults in model transformations. For this, the models generated conforming to the *Families_Extended* metamodel (cf. Figure 3) have been used as test models for the *Families-to-Persons_Extended* model transformation available from [48]. We have evaluated this efficacy by inserting faults in the model transformation, for which we have created mutants using the tool pre-

Table 2: Performance metrics.

		Execution Time (s)			Memory Consumpt. (MB)		
# Models	**# Elements**	ACO	GWO	Viatra Solver	ACO	GWO	Viatra Solver
50	10	1.0	1.5	9.4	26.6	43.2	212.6
	40	1.3	1.5	-	44.1	68.12	-
	320	2.1	2.5	-	20.7	163.77	-
100	10	1.2	1.5	15.9	38.9	139.71	264.4
350		2.4	2.7	59.1	29.3	268.22	372.2
500		2.9	3.6	85.4	22.9	348.71	360.1

Table 3: Mutation analysis for the *Families-to-Persons_Extended* model transformation [48]

	Mutant type			
	Typing	**Semantic**	**Syntactic**	**Total**
# Mutants	618	163	36	**817**
ACO	65.74%	73.39%	71.59%	**67.52%**
GWO	54.44%	63.10%	62.66%	**56.53%**
Viatra Solver	1.30%	1.90%	9.17%	**1.08%**

sented by Guerra et al. [33]. This tool receives a model transformation as input and generates several mutants divided in three categories, namely *typing*, *semantic* and *syntactic*. As we show in Table 3, we have used a total of 817 mutants of the *Families-to-Persons_Extended* model transformation. We compare the mutation scores obtained with the test models generated with ACO, GWO and Viatra Solver. We can see that ACO and GWO clearly outperform Viatra Solver. ACO performs the best, with a mutation score close to 70%. Regarding Viatra Solver, we inspected the models it generated and observed some anomalies in them, which might be the case for such a low mutation score. However, in other examples Viatra Solver yields slightly better results [30].

5. Impact

The goal of our framework is automated model generation for testing in MBSE environments. Although it is now mainly focused on low-code due to its ease of use, it was initially designed for model transformation testing [30]. In a recent published survey on model transformation testing and debugging [9], 11 challenges were identified from the analysis of 140 papers. With our framework, we try to make contributions in 5 of these challenges, which also apply to MBSE testing in general:

- *Challenge 2: Generalization of existing Approaches to MT Testing and Debugging.* There is a need for testing approaches that can be used with different modelling languages. We use the XMI standard as model representation format, so that the many modelling languages that use this format

can directly use the models generated by our tool without the need for model converters.

- *Challenge 5: Test Case Generation and Test Process Optimization.* This challenge calls for the development of new model generators that can make use of optimization algorithms and that are not based on constraint solvers for overcoming their performance limitations. Our framework applies two optimization algorithms never used in the literature, namely ACO and GWO. Table 2 shows that our framework can generate models almost 30 times faster than the well-known Viatra Solver[3] [32].

- *Challenge 8: Reusable and Realistic Evaluation Methods.* It promotes the use of realistic case studies, and not just small-scale ones. We applied a first prototype of our tool implemented with ACO [30] to a large metamodel of the Maude language [50], containing 45 classes. Our protoype was able to generate sets of up to 1,000,000 models.

- *Challenge 9: Baseline Technology Infrastructure.* Part of this challange reads "*we see that authors tend to use already available infrastructures.*" This is precisely what we seek with our framework: we provide the community with an open-source tool for MBSE.

- *Challenge 10: Comprehensive Testing and Debugging Frameworks.* This challenge is related to the previous one. It reads *"There is a lack of (...) frameworks (...). Such frameworks should define a simple way to write test cases (...)"*. Indeed, in the context of testing in MBSE, our tool provides an easy-to-use framework for model generation.

6. Conclusion

In this paper we have presented the Yekta framework. Its purpose is to generate models in the widely-used XMI format with the only input of a metamodel and an optional set of OCL constraints. The framework applies two optimization algorithms, namely ant-colony optimization and gray-wolf optimization, which have never been applied before in this context. Our tool can be used to generate test models in the general MBSE context, although its simplicity makes it very suitable for the new *low-code* and *no-code* environments.

Acknowledgements

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].

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

References

[1] S. Kent, Model driven engineering, in: M. Butler, L. Petre, K. Sere (Eds.), Proc. of IFM'02, 2002, pp. 286–298.

[2] J.-M. Favre, Foundations of model (driven)(reverse) engineering: Models, in: Int. workshop Dagsthul, Citeseer, 2004.

[3] J. Bézivin, In search of a basic principle for model driven engineering, Novatica Journal 5 (2) (2004) 21–24.

[4] J. Cabot, Positioning of the low-code movement within the field of model-driven engineering, in: Proc. of MODELS'20 Companion Proceedings, MODELS '20, 2020.

[5] A. C. Bock, U. Frank, Low-code platform, Business & Information Systems Engineering 63 (2021) 733–740.

[6] M. Hirzel, Low-code programming models, Communications of the ACM 66 (10) (2023) 76–85.

[7] J.-A. del Hoyo-Gabaldon, A. Moreno-Cediel, E. Garcia-Lopez, A. Garcia-Cabot, D. de Fitero-Dominguez, Automatic dataset generation for automated program repair of bugs and vulnerabilities through sonarqube, SoftwareX 26 (2024) 101664.

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

[9] J. Troya, S. Segura, L. Burgueño, M. Wimmer, Model transformation testing and debugging: A survey, ACM CSUR 55 (4) (2022) 1–39.

[10] M. Gogolla, A. Vallecillo, L. Burgueño, F. Hilken, Employing classifying terms for testing model transformations, in: Proc. of MoDELS, 2015, pp. 312–321.

[11] E. Guerra, M. Soeken, Specification-driven model transformation testing, Software & Systems Modeling 14 (2015) 623–644.

[12] S. Sen, B. Baudry, J.-M. Mottu, Automatic model generation strategies for model transformation testing, in: Proc. of ICMT, 2009, pp. 148–164.

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

[14] F. Fleurey, J. Steel, B. Baudry, Validation in model-driven engineering: testing model transformations, in: Proc. of MODEVVA, 2004, pp. 29–40.

[15] F. Fleurey, B. Baudry, P.-A. Muller, Y. Le Traon, Qualifying input test data for model transformations, SoSyM 8 (2) (2009) 185–203.

[16] L. M. Rose, S. Poulding, Efficient probabilistic testing of model transformations using search, in: Proc. of CMSBSE, 2013, pp. 16–21.

[17] M. Gogolla, J. Bohling, M. Richters, Validating uml and ocl models in use by automatic snapshot generation, SoSyM 4 (4) (2005) 386–398.

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

[19] C. A. González, F. Büttner, R. Clarisó, J. Cabot, Emftocsp: A tool for the lightweight verification of emf models, in: Proc. of FormSERA, 2012, pp. 44–50.

[20] M. Kuhlmann, L. Hamann, M. Gogolla, Extensive validation of ocl models by integrating sat solving into use, in: Proc. of TOOLS, 2011, pp. 290–306.

[21] E. K. Jackson, T. Levendovszky, D. Balasubramanian, Automatically reasoning about metamodeling, SoSyM 14 (1) (2015) 271–285.

[22] K. Bak, Z. Diskin, M. Antkiewicz, K. Czarnecki, A. Wasowski, Clafer: unifying class and feature modeling, SoSyM 15 (3) (2016) 811–845.

[23] E. Torlak, D. Jackson, Kodkod: A relational model finder, in: Proc. of TACAS, 2007, pp. 632–647.

[24] H. Zhong, L. Zhang, S. Khurshid, Combinatorial generation of structurally complex test inputs for commercial software applications, in: Proc. of FSE, 2016, pp. 981–986.

[25] L. De Moura, N. Bjørner, Z3: An efficient smt solver, in: Proc. of TACAS, 2008, pp. 337–340.

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

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

[28] J. Shelburg, M. Kessentini, D. R. Tauritz, Regression testing for model transformations: A multi-objective approach, in: Proc. of SSBSE, Vol. 8084 of Lecture Notes in Computer Science, 2013, pp. 209–223.

[29] W. Wang, M. Kessentini, W. Jiang, Test cases generation for model transformations from structural information, in: Proc. of MoDELS Workshops, 2013, pp. 42–51.

[30] M. Karimi, S. Kolahdouz-Rahimi, J. Troya, Ant-colony optimization for automating test model generation in model transformation testing, Journal of Systems and Software 208 (2024) 111882.

[31] O. Semeráth, R. Farkas, G. Bergmann, D. Varró, Diversity of graph models and graph generators in mutation testing, STTT 22 (1) (2020) 57–78.

[32] O. Semeráth, A. A. Babikian, S. Pilarski, D. Varró, Viatra solver: a framework for the automated generation of consistent domain-specific models, in: Proc. of ICSE Companion, pages=43–46, year=2019.

[33] E. Guerra, J. S. Cuadrado, J. de Lara, Towards Effective Mutation Testing for ATL, in: Proc. of MoDELS, 2019, pp. 78–88.

[34] J. Cabot, M. Gogolla, Object constraint language (ocl): a definitive guide, in: International school on formal methods for the design of computer, communication and software systems, Springer, 2012, pp. 58–90.

[35] M. Brambilla, J. Cabot, M. Wimmer, Model-Driven Software Engineering in Practice (2nd edition), 2017.

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

[37] T. Kühne, Matters of (meta-) modeling, SoSyM 5 (4) (2006) 369–385.

[38] J. Ludewig, Models in software engineering – an introduction, SoSyM 2 (2003) 5–14.

[39] S. J. Mellor, K. Scott, A. Uhl, D. Weise, R. M. Soley, MDA distilled: principles of model-driven architecture, O'Reilly, 2004.

[40] K. Czarnecki, S. Helsen, Feature-based survey of model transformation approaches, IBM systems journal 45 (3) (2006) 621–645.

[41] M. Harman, B. F. Jones, Search-based software engineering, Information and software Technology 43 (14) (2001) 833–839.

[42] J. H. Holland, et al., Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence, MIT press, 1992.

[43] K. Deb, Multi-objective optimization, in: Search methodologies, Springer, 2014, pp. 403–449.

[44] D. Varró, O. Semeráth, G. Szárnyas, Á. Horváth, Towards the automated generation of consistent, diverse, scalable and realistic graph models, in: Graph Transformation, Specifications, and Nets, Springer, 2018, pp. 285–312.

[45] O. Semeráth, D. Varró, Iterative generation of diverse models for testing specifications of dsl tools, in: Proc. of FASE, 2018, pp. 227–245.

[46] M. Dorigo, Optimization, learning and natural algorithms, Ph.D. thesis, Politecnico di Milano, Italy (1992).

[47] M. Karimi, S. M. Babamir, Qos-aware web service composition using gray wolf optimizer, International Journal of Information and Communication Technology Research 9 (1) (2017) 9–16.

[48] B. J. Oakes, J. Troya, L. Lúcio, M. Wimmer, Full Contract Verification for ATL using Symbolic Execution, SoSyM 17(3) (2018) 815–849.

[49] J. Troya, S. Segura, A. Ruiz-Cortés, Automated inference of likely metamorphic relations for model transformations, JSS 136 (2018) 188–208.

[50] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All about Maude - a High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic, Springer, Berlin, Heidelberg, 2007.

Code Metadata

Nr.	Code metadata description	Please fill in this column
C1	Current code version	v1.0
C2	Permanent link to code/repository used for this code version	`https://github.com/MeysamKarimi/Yekta-Framework/`
C3	Permanent link to Reproducible Capsule	-
C4	Legal Code License	MIT License
C5	Code versioning system used	git
C6	Software code languages, tools, and services used	Java
C7	Compilation requirements, operating environments & dependencies	JDK 11
C8	If available Link to developer documentation/manual	`https://github.com/MeysamKarimi/Yekta-Framework/blob/main/README.md`
C9	Support email for questions	meysam.karimi84@gmail.com

Table 4: Code metadata (mandatory)