Towards an Automated Fault Localizer while Designing Meta-models

Adel Ferdjoukh, Jean-Marie Mottu

Towards an Automated Fault Localizer while Designing Meta-models

Adel Ferdjoukh
University of Nantes, LS2N (UMR CNRS 6004)
Nantes, France
adel.ferdjoukh@univ-nantes.fr

Jean-Marie Mottu
University of Nantes, LS2N (UMR CNRS 6004)
Nantes, France
jean-marie.mottu@univ-nantes.fr

ABSTRACT

Meta-models are the centrepiece of Model Driven Engineering, required in many activities: modelling, creating DSLs (Domain Specific Languages), xDSLs (executable DSLs), or writing model transformations. Therefore, designing meta-models should be done carefully but it could be a complicated task with large ones. Meta-models should then be validated but it is mostly done considering their usability: in particular if it is possible to instantiate them. Automatic model generators are used and if they are unable to generate models it means the meta-model with its instantiation parameters (e.g. size of the models) is wrong. Several generators exist, but most of them have binary output: success or failure, without helping the meta-model debugging. In this paper, we introduce an approach, in which we statically analyse a meta-model with its instantiation parameters. In this first work, we detect inconsistencies considering each reference or each inheritance separately. Therefore we provide feedback to the meta-model designer to help her to debug the meta-model.

KEYWORDS

Meta-modelling, Model Generation, Fault Localization

1 INTRODUCTION

Model Driven Engineering (MDE) became very popular in academia and industry. Its efficiency has been proven in many real cases: medicine [7], aviation [1], automotive industry [24], *etc*. Model-based techniques are used for designing, refactoring, generating the code, testing software systems. They are very helpful in managing the complexity or heterogeneity of software systems.

A model-based process often begins by the definition of a new *meta-model*, required, for instance, to define *Domain Specific Language* (DSL). The meta-model plays a key role in all the operations that will be performed over the software system. For example, meta-models are used to write model transformations (MT) [26] which transform models into other models (M2M), or into source code or text (M2T)[1].

To ensure the correctness of any MDE process, only valid meta-models must be considered. However designing meta-models is tedious while dealing with complex systems. It could result in meta-models that cannot be instantiated in all the requested situations. Checking the validity of meta-models becomes, *de facto*, a very important issue. To validate their meta-models, domain experts try to instantiate them by creating real life models. However, that manual checking cannot be applied at a large scale and is therefore not sufficient. For this task, an existing solution is based on model generators which are already highly used in the next steps of a MDE development (e.g., generating test models to validate model transformations [21]).

Several model generation tools exist [14]: e.g., GRIMM [11], PRAMANA [25], EMFtoCSP [5], USE [13]. They instantiate a meta-model in order to build conformed models. The inputs of a generator are: a meta-model, a set of OCL constraints and *instantiation parameters*. With these last, the user configures the generator in order to choose the characteristics of instantiated model(s) (e.g., setting the number of instances generated for each class). Then, the tool transforms those inputs to send them to a solver which returns model(s) in case of success or error messages if it fails to generate a valid solution. However, even if it can alert on the non *instantiability of meta-models*, it provides too few help to debug them.

The goal of the approach that is described in this paper, is to help the meta-model designers to detect and fix invalid meta-models before facing the annoying but frequent case of generation failure. We develop an *automated fault localizer* to help them in debugging a non instantiable meta-model. Our approach uses *Systems of Linear Inequalities* (SLI) and focuses on the class' relations in the meta-model and the instantiation parameters. The structure of an *Ecore* meta-model is translated into SLI and our custom solver checks the consistency of a model generation process. In this paper, we show the first version of the translation of a meta-model into a linear system, and we describe how we use this in order to provide *fixing suggestions* for meta-model designers. The contributions are already implemented in a tool named TIWIZI which is independent from any model generation tool.

The rest of the paper is organised as follows: Section 2 gives the context and the motivation of the work. Section 3 describes the *fault localization* mechanism and presents TIWIZI, a tool that implements our contribution. Section 4 discusses existing work. Finally, Section 5 draws conclusions and opens perspectives.

2 CONTEXT & MOTIVATION

In this section, we describe the context and the motivations of the paper when tackling non instantiability of meta-models.

2.1 Meta-Modelling

The meta-models we consider are written using the *EMF/Ecore* language.

Each class can have one or more attribute. The possible types for attributes are all the usual primitive types (integer, char, string, etc.) and enumerations.

In our approach, we consider the relations between classes and their three variations in an *Ecore* diagram: *reference*, *bidirectional reference* (a reference and its opposite), *composition* (a bidirectional reference in which one class contains the other one). Each relation has two cardinalities (lower and upper bound) that bound the number of objects that could be linked to a given object. Bidirectional

[1] M2M: Model to Model transformation; M2T: Model to Text transformation

references have cardinalities in both sides. As a good practice rule, a unique *root class* is the top container and it is instantiated only once as a root of a tree of containment relations between all the objects in a model. We also consider classes inheriting from other classes. This is defined by a *super type* relation.

A meta-model also requires constraints, typically written in OCL, to specify invariants. However considering the constraints is out of the scope of this first work.

2.2 Meta-model instantiation

Depending on how the models will be used during the development of a MDE tool chain (DSLs, transformations), they are created as instances of a meta-model. *Instantiability* of a meta-model should be considered to prevent that a real life model provided by a user or another part of the chain fails to conform the meta-model, preventing it to be transformed for instance. Moreover, to validate DSLs or model transformations w.r.t. their specification, representative models should be generated to test their implementation. A model generator then requires instantiable meta-models to compute valid instances as test models.

2.2.1 Instantiation parameters. It gathers the information that is required to instantiate a meta-model, e.g. by a model generation tool. For example in the model generator GRIMM, it concerns the following information:

- Number of exact instances for each class of the meta-model.
- Bounds for unbounded references.
- Values for enumerations.
- Domains for attributes (optional but suitable for more precision).
- Probability distributions for links between classes.

Considering the instantiability of a meta-model and without anticipating with which intent a model generator will be used next (e.g. model transformation testing), we focus in this first work on two instantiation parameters that we call *candidate values* (CV).

CV are provided by the user when checking instantiability of meta-models. We currently focus on two different parameters:

- Number of exact instances for each class of the meta-model.
- Bounds for unbounded references.

2.2.2 Non instantiable Meta-models. We distinguish three major reasons why meta-model instantiation could be not possible:

- Unreachable candidate values. The instance that the user tries to generate is not reachable because given CV cannot satisfy the cardinality of at least one reference in the meta-model.
- Inconsistent meta-model structure. A combination of elements is impossible to instantiate. For example, three classes are linked by a cycle of relations and no combination of values that satisfies all the cardinalities is possible.
- Incoherent OCL constraints. The source of error in this case are the OCL constraints of the meta-model.

Remark Another obvious source of error is a faulty syntax (e.g. an unnamed class). This kind of error can be easily checked using *Ecore* validator in Eclipse for instance.

In this first work, we consider only *unreachable CV*. Our ideas for tackling the two other sources of problems are discussed in section 5.

2.3 Meta-model validation

The domain expert is supposed to design valid meta-model. She could additionally be a meta-modelling expert and manage to follow good meta-modelling practices[2]. However, a dedicated step of validation is required, such as for any development. In that case, current technique using model generators could be not a skill of the domain expert.

2.3.1 Model Generators. Usually, the first automatic non validation of a meta-model is done by a model generator when it fails to instantiate it.

Generators are based on a translation into a search-based or combinatorial technique to find models. The most famous techniques that are used are: SAT [2], CSP (Constraint Satisfaction Problem) [22], Alloy [16] and SMT (SAT Modulo Theory) [9]. Such techniques always provide existing and easy-to-use solvers to find solutions. Those solvers return a binary output: *success* if a model is found or *failure* if there is no solution. However, they give too few information about the origin of failure.

We observe that the users of model generation tools, such as GRIMM [11], PRAMANA [25] or EMFtoCSP [5], have many difficulties when they meet a *failure*, because most of them are familiar with either *Ecore* or one the combinatorial techniques that tools use: CSP, SAT or SMT. An *Ecore* expert can manually check the meta-model and its elements, mainly references. Then, she tries to debug the generation algorithm. Beside that, a CSP or SAT expert can look up into the intermediary files (eg. *xcsp* files), that are generated by the tool and try to find what is wrong inside them. All these verifications have to be done manually. They are time consuming. Worse, they have to be done each time the tool fails to generate a model. Another solution would be the use of the *Fault Localization* mechanisms that combinatorial techniques provide. For example, in Constraint Programming, there exists a sub-field called Max-CSP [17], in which the goal is to identify the subset of constraints responsible of the failure. However, Max-CSP suffers from a lack of scalability and a reverse step has to be done in order to map from *constraints* into *meta-models elements*. An equivalent mechanism exists in SAT [12] as well, but without providing more help to debug.

In summary, model generators use search-based techniques and do not provide *fault localization* mechanisms to help the users in debugging. Some of search-based and combinatorial techniques have kind of debuggers and fault localizers. Unfortunately, they are hard to use for MDE experts and reverse engineering is needed to map bugs into meta-models elements. For all these reasons, meta-model designers prefer to not use a tool for model generation. They validate their meta-models only manually. To fix that, a *Fault Localization* mechanism during meta-model instantiation is needed.

2.3.2 Tiwizi and Model Generators complementarity. The objective of this work is to reconcile meta-model designers with model generation tools. We provide a user friendly tool to assist them during design task. Concretely, we developed TIWIZI, an interactive fault localizer. The role of TIWIZI is to help to validate *Ecore* meta-models. TIWIZI does not generate models. GRIMM, PRAMANA or EMFtoCSP keep their role for meta-model instantiation.

[2] https://sites.google.com/site/metamodelingantipatterns/

3 AUTOMATED FAULT LOCALIZATION

This section presents the main contribution of our paper. We describe how to perform automated *fault localization* and provide *fixing suggestions*. As well, we present TIWIZI, the fault localizer we created.

Our fault localization mechanism is based on *Systems of Linear Inequalities* (SLI). An SLI is a set of *Linear Inequalities* between the same variables. Each inequality has the following shape:

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n <, \leq, >, \geq, =, \neq b$$

The variables are x_i. a_i are called coefficients and b is a constant term. Every inequality must have one of the inequality symbols $<, \leq, >, \geq, =, \neq$. For a more complete overview of this area, please refer to [27].

To perform fault localization, the structure of a meta-model is translated into a *system of linear inequalities* (SLI). The pair composed of the generated SLI and the input *candidate values* is checked in order to localize errors in the process of model generation. Figure 1 illustrates the steps of our method and tool (TIWIZI). It consists of three important steps: (1) generate the SLI, (2) check the consistency of the generated system and (3) deduce fixing suggestions. Each one of these steps is explained in a dedicated sub-section.

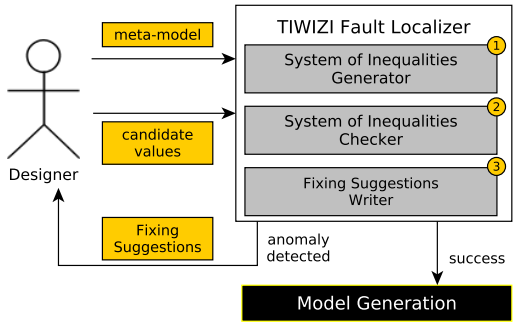


Figure 1. Steps for fault localization and suggestion generation using TIWIZI

3.1 Generation of Linear Inequalities

This section describes the *System of Linear Inequalities* (SLI) we build, in order to localize faults while designing meta-models. The work described in this paper focuses on four main elements of meta-models: *unidirectional references*, *bidirectional references*, *containments* and *inheritance*.

3.1.1 Unidirectional references. They associate two classes (possibly the same) of the meta-model in one direction onlyAn example of unidirectional reference is shown in Figure 2. In this example we want to say that objects of type *House* are connected to objects of type *Room* by a reference called *rooms*. The cardinality 1..5 means that a House is linked to at least 1 and to at most 5 Rooms.

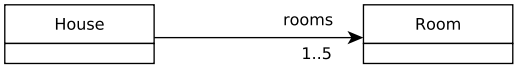


Figure 2. An example of unidirectional reference

Let us assume that we have only one instance of class House. In Figure 3, we show the minimal and maximal configurations for the previous reference. In the minimal configuration we have $\#House = \#Room$ (one room per house) and in the maximal $\#Room = 5 * \#House$ (5 rooms per house). This means that all consistent configurations are between these two bounds. So, to translate an unidirectional reference into *SLI*, we create the following pair of inequalities:

$$\begin{cases} \#House \leq \#Room \\ \#Room \leq 5 \times \#House \end{cases}$$

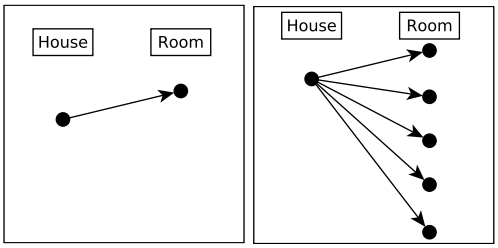


Figure 3. Minimal (left side) and maximal (right side) configuration for reference in figure 2 if we have only one House.

3.1.2 Bidirectional references. They associate two classes (possibly the same) in two opposite directions. It is modelized in *Ecore* with two references, each one being the *eOpposite* of the other one. The main difference between one *unidirectional reference* and a pair of *bidirectional references*, is that this last requests two cardinality constraints (*one for each direction*) to be checked at the same time, while instantiating them.

We cannot process this configuration by considering two unidirectional references, otherwise many false-positive examples are encountered. We then propose a way to manage the bidirectional references as a pair. Let us explain our solution by translating the bidirectional references of the Figure 4.

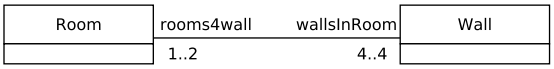


Figure 4. An example of bidirectional references

This solution is based on a particular graph, called *regular bipartite graph* [23]. A bipartite graph $G = (U, V, E)$ is regular bipartite if all the nodes of U have the same degree x and all the nodes of V have the same degree y. It is denoted x, y-regular. Regular bipartite graphs have an interesting characteristic: $x \times |U| = y \times |V|$.

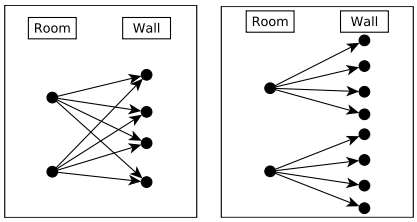
We use this kind of graphs because the instantiation of a reference always produces a bipartite graph. The instances of the first class (e.g., Room) are the first part U of the graph and the instances of the second class (e.g., Wall) are the second part V.

First, we consider instances of class *Room*, and the number of connected instances of class *Wall*. The extreme configurations are the following:

- *Minimal configuration*. Rooms share as many walls as possible. It means that each wall is connected to a maximum number of rooms (= 2). This configuration produces a $4, 2$-regular graph (Figure 5.a).

- *Maximal configuration*. Rooms do not share their walls. Each wall is connected to only one room. This configuration then produces a $4,1$-regular graph (Figure 5.b).

These two configurations remain valid whatever the number of rooms and walls. It is only related to the cardinalities of corresponding references: 1..2 & 4..4.



(a) $4,2$-regular graph (b) $4,1$-regular graph

Figure 5. Minimal and maximal configurations

The previous regular graphs are characterized by the following formulas:

$$4 \times \#Room = 1 \times \#Wall \vee 4 \times \#Room = 2 \times \#Wall$$

It means that all consistent configurations must respect the following inequalities:

$$1 \times \#Wall \leq 4 \times \#Room \vee 4 \times \#Room \leq 2 \times \#Wall$$

So, to encode the pair of opposite references, we create 4 inequalities:

$$\begin{cases} \frac{1}{4}\#Room \leq \#Wall \\ \#Wall \leq \frac{2}{4}\#Room \\ \frac{4}{2}\#Wall \leq \#Room \\ \#Room \leq \frac{4}{1}\#Wall \end{cases}$$

3.1.3 Composition references. A composition relation (A to B) with cardinalities $l..u$ is treated as a pair of bidirectional references with cardinalities 1..1 in one side and $l..u$ in the other side. This is done this way in order to force involved objects to have a unique container. This gives us the following inequalities:

$$\begin{cases} \frac{1}{u}\#A \leq \#B \\ \#B \leq \frac{1}{l}\#A \\ l \times \#B \leq \#A \\ \#A \leq u \times \#B \end{cases}$$

3.1.4 Super type relations. The methodology we propose for treating inheritance between classes is inspired by the translation of a meta-model into CSP that is performed in GRIMM tool [11]. Let us take the illustrative example in Figure 6. We can see a class A linked to a class B which had two sub-classes C and D. It means that an instance of A can be linked to 1..5 instances of B, C and D at the same time (instances of concrete classes are only considered). This case is translated into *SLI* using the following inequalities:

$$\begin{cases} \#A \leq \#B + \#C + \#D \\ \#B + \#C + \#D \leq 5 \times \#A \end{cases}$$

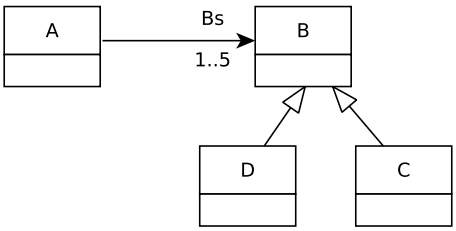


Figure 6. An example of unidirectional reference with inheritance

Remark equivalent treatments are applied for bidirectional references with inheritance.

3.2 SLI Checker

Here, we explain how we use input *candidate values* in order to check the consistency of the generated *SLI*. The algorithm in Listing 1 explains how a generated SLI is checked according to the candidate values. For each one of the inequalities, we check if the candidate values given by the user are consistent. When the checking fails, we consider that an anomaly is detected. The next step is to generate a fixing suggestion to help the user.

```
input:  SystemLinearInequalities sli
    CandidateValues cvs

output: List<DetectedAnomaly> anomalies

begin
  foreach (i:Inequality in sli) do
    if ( not check(i, cvs.get(i))) then
      a: new DetectedAnomaly(i,cvs.get(i))
      anomalies.add(a)
    endif
  endfor

  return(anomalies)
end
```

Listing 1. Algorithm for checking SLIs

Our goal is not to solve the system of linear inequalities in order to find a valid solution because we use the *CV* given by the user. For this reason, we check the SLI instead of solving it. One improvement of our current work would be mixing solving (using an existing solver) and checking. Indeed, sometimes the user does not want to give all the candidate values, because the meta-model is too big for example. In this case, we could try to solve the SLI as well, in order to suggest a whole consistent input configuration or to complete a partial one. The complexity of solving SLIs is much greater than checking that a vector of CVs is a solution. However, there exist a polynomial algorithm for checking that a homogeneous SLI has no solution [10]. An SLI is homogeneous if all constant terms (b values of inequalities) are zeros. The SLI that is created from the translation of meta-models is homogeneous.

3.3 Generation of Fixing Suggestions

Once an inconsistency between an inequality and a list of *candidate values* is met by the *SLI checker*, a *fixing suggestion* is calculated and returned to the user. The goal is to help her to quickly fix the

problem. The calculation of suggestions is not a challenging problem but we think it is very useful for users.

Here we give an example of such a suggestion. Let us reconsider again the example of reference given in Figure 2, in which an instance of class House can be linked to 1..5 instances of class Room. Listing 2 shows a fixing suggestion provided by our tool. The problem here is that the user tries to create an inconsistent model containing 1 instance of House and 6 instances of Room. This configuration violates the references between the two classes. The tool suggests to reconsider either the cardinalities or the CV.

```
--references
rooms: House []->[1..5] Room

--inequalities
House <= Room
Room <= 5*House

--candidate values
[House=1, Room=6]

--fixing suggestions
Please reconsider cardinalities for reference [rooms] >>
    upperBound++
Please reconsider number of instances >> more [House] or
    Less [Room]
```

Listing 2. An example of fixing suggestion given by TIWIZI

By default, the suggestions are written in a very succinct mode directly in the terminal. But, for a better readability, the user can ask for a *verbose* mode that generates a *tiwizi* log file for all detected anomalies (as shown in Listing 2). A pdf file summarising all the suggestions is then created using our custom syntax highlighting (calling LaTeX).

Fixing suggestions are useful because big meta-models contain dozens of classes and references. Thus, it is hard to manually localize all possible faults. Moreover, conducted user experiments [11] show that many problems come from the CV.

3.4 Tooling

All the contributions of this paper are implemented in a tool called TIWIZI. The tool is designed as a plug-in for any model generation tool: it garantee the instantiability of the meta-model before the generator try to instantiate it. Currently, TIWIZI is plugged to GRIMM as an initial step of its generation process. Such a connection can easily be done with any model generator and our code is open source.

The core part of the code that concerns the generation and the checking of a system of linear inequalities is written in *Java*. We preferred to write our own SLI checker because available checkers need big efforts to adapt them to our MDE purpose. Our checker was carefully tested on diverse meta-models to ensure it correctness. Both the pdf creation task for verbose suggestions and the connection to GRIMM tool are written in bash.

TIWIZI needs two inputs: a meta-model and a configuration file that contains the *candidate values*. The tool is fully automated and giving the input is the only manual task. The output is a list of fixing suggestions in case of failure or a call to the model generation tool in case of success. The first release of TIWIZI is available on our *github* repository (https://github.com/ferdjoukh/tiwizi/). All what you need to start using the tool can be found on the same web page.

Remark Filling the configuration file manually is very often time-consuming and boring, as the size of the meta-model is growing. In order to help the users to quickly fill all this information, our tool generates a pre-filled file. Users only need to bring their modifications if necessary.

4 RELATED WORK

In this section, we list several works that consider the quality of meta-models with different approaches and other approaches assessing the quality of modelling using Linear Algebra as well.

A first set of works identifies pattern and anti-pattern which increase and reduce the quality of the meta-models. They do not focus on the instantiability of the meta-model but prevent to build models of low quality, e.g. in [8].

In [15], Hinkel et al. confirm thanks to an empirical study the importance of the instantiability. It was already considered by Cadavid et al. in [6]. Such as most of the existing works, they request test model instances of the meta-model to validate it.

In [18] and [19], *Lopez et al.* present a language, called *mmSpec* whose the main purpose is to define quality criteria over meta-models. For example, one can ensure the strong connection of meta-models or the reachability of classes by using such criteria. The objective is both assisting in meta-modelling and analysing existing meta-models. There is a complementarity between this work and our approach. Therefore, users should use both tools for a more accurate validation of meta-models.

In [20], *Ma et al.* define metrics over meta-models with the purpose of assessing their quality. These metrics are inspired for classic object oriented metrics (e.g. depth of inheritance tree). The authors also found correlations between their metrics and quality properties, such as, reusability or understandability of meta-models.

In [3] and [4], *Boufares et al.* consider the consistency of cardinalities in UML diagrams and ER-Schemas. In both papers, they translate the cardinality systems into Integer Linear Programming (ILP). An ILP solver then solves it and answers if the cardinalities are consistent or no. However, the papers do not give much details about the fault localization mechanism which seems to be manual.

5 CONCLUSION & FUTURE WORK

In this paper, we present a method for automated fault localization during the task of designing meta-models. The main objective of our work is to help users to check if their meta-models could be instantiated and to localize the origin of failure if not. This first work focuses on errors occurring between the candidate values that are chosen by the user and some elements of the meta-model: references, bidirectional references, compositions and inheritance relations.

Our method is based on *Linear Algebra*. We model the problem of consistency of meta-models as a *System of Linear Inequalities (SLI)* and we check their consistency in order to localize bugs. The contributions of the paper are implemented in a tool called TIWIZI. TIWIZI takes a meta-model and candidate values (instantiation parameters) as input and provides fixing suggestions. Those suggestions are guidelines for the user. They are used to modify and correct the meta-model and the instantiation parameters. The role of our tool is to guarantee the success of a model generation process or to debug

it and automatically localize anomalies. Beside that, model generators keep their role for meta-model instantiation. The complexity of TIWIZI is linear and depends only on the number of references of the meta-model, not on the size of generated models. In addition, the tool is fully automated.

5.1 Limitations & Future work

This sections aims to list the limitations of our work, and the future challenges that rise from it.

The current version of our tool considers the structure of the meta-model (references between classes and inheritance) and all the fixing suggestions we generate are related to these elements of the meta-model. We did not include the attributes of classes in our translation into SLI. We think that we can add some features to our tool in order to take into account the attributes as well. All this can be inspired from what model generators do for attributes. We could for example detect anomalies in types or missing values.

As described in section 2.2.2, non-instantiability of meta-models is due to three major reasons. The first one is the topic of the current paper. However, for a complete fault localizer, we must consider the two other sources of errors. Hereinafter, we discuss the promising ideas for improving this work.

Some meta-models are faulty because of their own structure without regarding the CV. For example, in some cycles of linked classes, wrong cardinalities exclude any possibility of success (the work in [28] explains that for UML models). Therefore, we could identify, define and detect such *suspicious patterns* for *Ecore* diagrams and apply our approach on a larger scope of meta-model elements. This would help us to target larger or more accurate fixing suggestions.

The usefulness of a fault localization mechanism for OCL is undoubted. Again, we can use the OCL translators of model generators in order to tackle this interesting challenge and take into account OCL constraints of meta-models. We can imagine a solution that finds the maximum set of consistent constraints. Then, we could suggest to the user to correct only the suspicious set. This solution can for example iterate on the OCL constraints and translate only one at each step. If an inconsistency is detected then the constraint is added to the set of suspicious candidates.

The final improvement of this work is of course an experimental study to measure the benefits of our proposal. We plan to run two different steps: (i) statistical and quantitative study and (ii) user experience study.

The statistical and quantitative study consists in the running of the tool on several meta-models of different sizes and origins. The goal here is to show that TIWIZI is able to localize faults precisely whereas model generators are only able to notice failures. For example, if we create mutant of the meta-models, we can count the number of bugs that our tool discovers.

The goal of the user experience study is to show that TIWIZI really helps people to localize bugs during meta-modelling. We can measure the time that a user needs to find a bug using several existing model generation tools and compare it to the time that TIWIZI needs to automatically localize the same bugs.

REFERENCES

[1] Gérard Berry. 2008. Synchronous design and verification of critical embedded systems using SCADE and Esterel. *Lecture Notes in Computer Science* 4916 (2008).

[2] Armin Biere, Marijn Heule, and Hans van Maaren. 2009. *Handbook of satisfiability*. Vol. 185. IOS press.

[3] Faouzi Boufares and Hachemi Bennaceur. 2004. Consistency Problems in ER Schemas for Database Systems. *Information Sciences* 163, 4 (2004), 263–274.

[4] Faouzi Boufarès, Hachemi Bennaceur, and Aomar Osmani. 2003. On the Consistency of Cardinality Constraints in UML Modelling. In *ISPE, International Conference on Enhanced Interoperable Systems*. 287–292.

[5] Jordi Cabot, Robert Clarisó, and Daniel Riera. 2008. Verification of UML/OCL Class Diagrams using Constraint Programming. In *ICSTW, IEEE International Conference on Software Testing Verification and Validation Workshop*. 73–80.

[6] Juan Cadavid, Benoit Baudry, and Houari Sahraoui. 2012. Searching the boundaries of a modeling space to test metamodels. In *Fifth IEEE International Conference on Software Testing, Verification and Validation*.

[7] Catalina Martínez-Costa and Marcos Menárguez-Tortosa and Jesualdo Tomás Fernández-Breis and José Alberto Maldonado. 2009. A model-driven approach for representing clinical archetypes for Semantic Web environments. *Journal of Biomedical Informatics* 42, 1 (2009), 150–164.

[8] Hyun Cho and Jeff Gray. 2011. Design patterns for metamodels. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*. ACM, 25–32.

[9] Leonardo Mendonça De Moura and Nikolaj Bjørner. 2011. Satisfiability Modulo Theories: Introduction and Applications. *Communications of the ACM Journal* 54, 9 (2011), 69–77.

[10] Lloyd L Dines. 1919. Systems of linear inequalities. *Annals of Mathematics* (1919), 191–199.

[11] Adel Ferdjoukh, Anne-Elisabeth Baert, Eric Bourreau, Annie Chateau, and Clémentine Nebut. 2015. Instantiation of Meta-models Constrained with OCL: a CSP Approach. In *MODELSWARD*. 213–222.

[12] Zhaohui Fu and Sharad Malik. 2006. On solving the partial MAX-SAT problem. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 252–265.

[13] Martin Gogolla, Fabian Büttner, and Mark Richters. 2007. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming* 69, 1–3 (2007), 27–34.

[14] Wu Hao. 2013. *Automated Metamodel Instance Generation Satisfying Quantitative Constraints*. Ph.D. Dissertation. National University of Ireland Maynooth.

[15] Georg Hinkel, Max E Kramer, Erik Burger, Misha Strittmatter, and Lucia Happe. 2016. An Empirical Study on the Perception of Metamodel Quality.. In *MODELSWARD*. 145–152.

[16] Daniel Jackson. 2012. *Software Abstractions: logic, language, and analysis*. MIT Press.

[17] Javier Larrosa and Pedro Meseguer. 1996. Exploiting the use of DAC in Max-CSP. In *CP, International Conference on Principles and Practice of Constraint Programming*. 308–322.

[18] Jesús López-Fernández, Esther Guerra, and Juan de Lara. 2014. Assessing the Quality of Meta-models. In *MoDeVVA Workshop*. 3–12.

[19] Jesús López Fernández, Esther Guerra, and Juan Lara. 2014. Meta-Model validation and verification with MetaBest. In *ASE, ACM/IEEE International Conference on Automated Software Engineering*. 831–834.

[20] Zhiyi Ma, Xiao He, and Chao Liu. 2013. Assessing the quality of metamodels. *Frontiers of Computer Science* 7, 4 (2013), 558–570.

[21] Jean-Marie Mottu, Sagar Sen Simula, Juan Cadavid, and Benoit Baudry. 2015. Discovering Model Transformation Pre-conditions Using Automatically Generated Test Models. In *Proceedings of the 2015 IEEE 26th International Symposium on Software Reliability Engineering*. Gaithersburg, MD, USA, 88–99.

[22] Francesca Rossi, Peter Van Beek, and Toby Walsh (Eds.). 2006. *Handbook of Constraint Programming*. Elsevier Science Publishers.

[23] Edward R Scheinerman and Daniel H Ullman. 2011. *Fractional graph theory: a rational approach to the theory of graphs*. Courier Corporation.

[24] Gehan MK Selim, Shige Wang, James R Cordy, and Juergen Dingel. 2012. Model transformations for migrating legacy models: an industrial case study. In *European Conference on Modelling Foundations and Applications*. Springer, 90–101.

[25] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. 2008. On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing. In *ICST, IEEE International Conference on Software Testing, Verification and Validation*. 328–337.

[26] Shane Sendall and Wojtek Kozaczynski. 2003. Model transformation: The heart and soul of model-driven software development. *IEEE software* 20, 5 (2003), 42–45.

[27] Gerard Sierksma. 2001. *Linear and integer programming: theory and practice*. CRC Press.

[28] Robert Wille, Mathias Soeken, and Rolf Drechsler. 2012. Debugging of inconsistent UML/OCL models. In *DATE, Design, Automation and Test in Europe*. 1078–1083.