Rule-based Repair of EMF Models: An Automated Interactive Approach

Nebras Nassar[1(✉)], Hendrik Radke[2], Thorsten Arendt[3]

[1] Philipps-Universität Marburg, Germany,
nassarn@informatik.uni-marburg.de
[2] Universität Oldenburg, Germany,
Hendrik.Radke@informatik.uni-oldenburg.de
[3] GFFT Innovationsförderung GmbH, Bad Vilbel, Germany,
thorsten.arendt@gfft-ev.de

Abstract. Managing and resolving inconsistencies in models is crucial in model-driven engineering (MDE). In this paper we consider models that are based on the Eclipse Modeling Framework (EMF). We propose a rule-based approach to support the modeler in automatically trimming and completing EMF models and thereby resolving their cardinality violations. Although being under repair, the model may be viewed and changed interactively during this repair process. The approach and the developed tool support are based on EMF and the model transformation language Henshin.

Keywords: Model driven engineering · Eclipse Modeling Framework (EMF) · Model transformation · Model repair

1 Introduction

Model-driven engineering has become increasingly popular in various engineering disciplines. Although model editors are mostly adapted to their underlying domain-specific modeling language, they usually allow to edit inconsistent models. While upper bounds of multiplicities are mostly obeyed, the violation of lower bounds and further constraints requiring the existence of model patterns is usually tolerated during editing. This means that model editors often use a relaxed meta-model with less constraints than the original language meta-model [7].

The result of an editing process may be an invalid model that has to be repaired. There are a number of model repair approaches in the literature which, however, are either purely interactive such as [4,5,11,12] or fully automated such as [1,6,8,17]. Our approach intends to integrate automatic model repair into the editing process allowing user interaction during repair. It does not leave the resolution strategy completely to the modeler as in pure rule-based approaches. Instead, it is semi-automatic and guides the modeler to repair the whole model.

In our approach, we consider modeling languages being defined by meta-models based on the Eclipse Modeling Framework (EMF) [18]. We present an algorithm to model repair that consists of two tasks: (1) The meta-model of a

given language is translated to a rule-based model transformation system containing repair actions. (2) A potentially invalid model is fully repaired using the generated transformation system which configures the model repair tool according to the given language. An ordered list of used rule applications is reported as a second output. In summary, the contributions of this paper are: (1) an automatic derivation of repair rules (actions) from a given meta-model that is rooted and fully instantiable, (2) a rule-based algorithm for model repair, and (3) an Eclipse-based tool to automate the rule derivation and the model repair. Note, the correctness proof of the algorithm can be found at [10].

The paper is structured as follows: In Section 2, we introduce our running example. In Section 3, we present our repair algorithm, apply it at an example and sketch how a meta-model is translated to a rule-based transformation system. The developed Eclipse-based tools are presented in Section 4. An overview on related work is given in Section 5 while Section 6 concludes the paper.

2 Running example

The running example is a simple *Webpage* modeling language for specifying a specific kind of web pages.

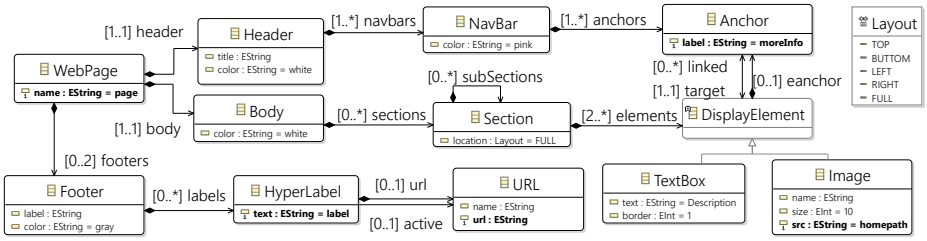

Fig. 1: *Webpage* meta-model

Our *Webpage* modeling language is defined by the meta-model shown in Figure 1. Shortly, each web page has a *name* which requires a value *(being page by default)*. A web page contains a *Header*, up to two *Footer*s, and a *Body*. A header includes at least one navigator bar *NavBar* which contains at least one *Anchor*. A body includes an arbitrary number of *Section*s. A section may contain subsections. A section is designed to contain at least two arbitrary *DisplayElement*s of type *Image* or *TextBox*. Each display element may contain an *Anchor*. An anchor has a *label* which requires a value *(being moreInfo by default)* and must be connected to one display element. A web page must not contain more than two *Footer*s. A footer may include an arbitrary number of *HyperLabel*s. A hyper label may contain one *URL*. A url node has a *url* which requires a value. A hyper label may be connected to one url node.

Figure 2 presents the abstract syntax representation of an invalid *Webpage* model repaired to a valid one. The solid-line elements represent the invalid web page model. It consists of a blue header containing a pink navigator bar and three footers: footer *Appointment* contains a hyper label with text *calendar* including

a url node with name *calurl* and a *url* attribute with empty value; the label *calendar* is activated. Footer *Address* contains a hyper label with text *floor 2*, and footer *Location* contains a hyper label with text *label3* including a *url* node with name *url3* and a *url* attribute with empty value. The label *label3* is activated.

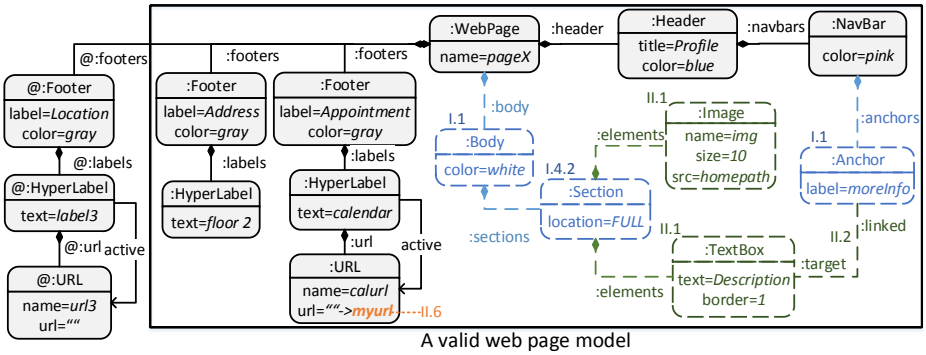

Fig. 2: The abstract syntax of the invalid *Webpage* being repaired

To repair this invalid *Webpage* model, it is first trimmed and then completed: Since there are too many footers, one *Footer*-node and its children have to be deleted. The selection of the footer can be done automatically in a non-deterministic way or determined by the user. We select the footer annotated with @. Then, the trimmed web page still contains all solid-line elements shown in Figure 2 without those annotated with @. To complete this *Webpage* model, at least the following missing elements (represented as dashed elements in Figure 2) have to be created: A *Body*-node and an *Anchor*-node are needed to fulfill the lower bounds of containment types *body* and *anchors*, respectively. An edge of type *target-linked* is needed to fulfill the lower bound of non-containment type *target*. Therefore, a *Section*-node containing two nodes of type *DisplayElement* (e.g., an image and a text box) are demanded to fulfill the lower bounds of containment type *elements* and non-containment type *target*. The *url*-attribute value of the *URL*-node *calurl* has to be set (to, e.g., *myurl*).

3 Rule-based model repair

Our approach to model repair consists of two activities: (1) *Configuration:* The meta-model of a given language is translated to a rule-based model transformation system (repair rules). (2) *Model repair:* A potentially invalid model is repaired yielding a valid model. The repair algorithm uses the generated transformation system and is presented first.

3.1 A rule-based algorithm for model repair

As pre-requisite, our approach requires an instantiable meta-model without OCL constraints. Given an invalid model, i.e., a model that does not fulfill all its

multiplicities, our algorithm is able to produce a valid one. The repair process is automatic but may be interrupted and interactively guided by the modeler.

The activity diagram in Figure 3 illustrates the overall control flow of our algorithm which consists of two main parts: The left part performs *model trimming* eliminating supernumerous model elements. The right part performs *model completion* adding required model elements.

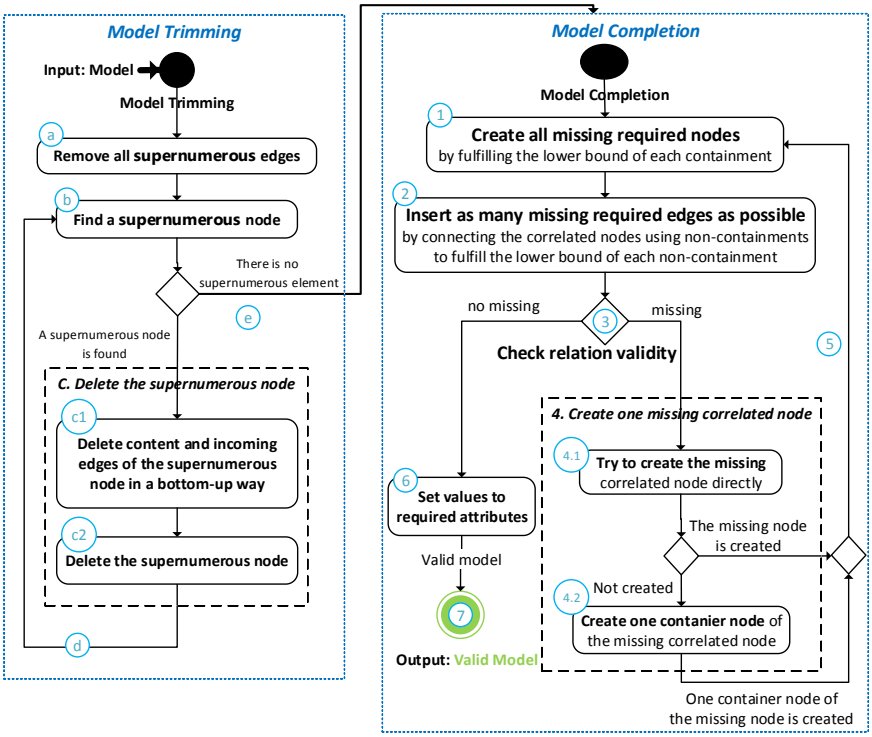

Fig. 3: Model trimming and completion algorithm

Model trimming: Step (a) in Figure 3 removes *all supernumerous edges*, i.e. edges that exceed the upper-bound invariants of each non-containment edge type. Step (b) checks if there is a *supernumerous node*, i.e., a node which exceeds the upper-bound invariants of a containment edge type. If there is none, the model is ready to be completed (Step (e)). Otherwise, this node and its content have to be deleted; this is done in Step (c). It deletes all the incoming and outgoing edges of this node and its content nodes, and then deletes the content nodes in a bottom-up way (Step (c1)); thereafter, it deletes the node itself (Step (c2)). This bottom-up deletion process starts at all the leaves and continues with their containers. Step (d) calls again Step (b) to check if there is another supernumerous node.

Model completion: Once there is no further supernumerous element in the input model, the model can be completed: Step (1) creates *all missing required*

nodes, i.e., nodes that are needed to fulfill the lower-bound of each containment edge type. Thereafter, we have a skeleton model which contains at least all the required nodes. At this stage, the model may contain nodes which are not correlated by required edges. Two node types being linked by a non-containment type are called correlated. Step (2) tries to insert *all missing required edges* by connecting the existing correlated nodes in the model. These edges are needed to fulfill the lower-bound of each non-containment edge type. This step may stop without having inserted all required edges due to potentially missing correlated nodes, i.e., it may happen that there is no further free node to correlate with.

Step (3) checks the validity of all required non-containment edge types. If all the required edges are already inserted, then we have a valid model w.r.t. all multiplicities of edge types. This also means that all the required nodes have been created. Otherwise, there is still at least one required edge missing. In this situation, Step (4) tries to add one missing node to fulfill the lower bound of a non-containment edge type. Although the number of missing nodes may be more than one, only one missing node is added in Step (4). If a missing node cannot be created directly, a (transitive) container node of a missing one is created. The added node may function as, e.g., target node for missing required edges. Hence, it may help to solve other inconsistencies in the model. The design decision of adding just one node in Step (4) helps to find a small completion.

Note that the type of a missing node may have several subtypes. In this case, there may be several possibilities to create a missing node choosing one of these subtypes. A further variation point are node types with several containers. This non-determinism may be solved by user interaction or automatically by randomly picking one. Thereafter, the next iteration of model completion starts with Step (5). Adding a node to the model may require adding further required nodes and edges. Starting a new iteration ensures that all those model elements are generated and that all missing nodes and edges will be created in the end. Once the instance model is valid w.r.t. edge type multiplicities, the values of all the empty required attributes are set (Step (6)). In Step (7) the algorithm stops.

3.2 An example repair process

We illustrate our algorithm by applying it to the invalid *Webpage* model in Figure 2. The invalid model is first trimmed and then completed. This process is described below and illustrated in Figure 2. The annotations at the model elements refer to the corresponding iteration and step numbers.

Model trimming: Since the model does not have supernumerous edges, we go directly to Step (b). Here, a supernumerous *Footer*-node is found. Assuming that the *Footer*-node *location* is selected for deletion. In Step (c1), the edge *active* is removed between the nodes *label3* and *url3*. Then, node *url3* itself is deleted. Thereafter, node *label3* can be deleted. Finally, Step (c2) deletes the selected *Footer*-node *Location*. Step (d) calls again Step (b) but there is no further supernumerous node. Hence, the process of model trimming is finished and the output model is ready to be completed (Step (e)).

Model completion is done in two iterations. *Iteration I:* In Step (1), an *Anchor*-node and a *Body*-node are created to fulfill the lower-bound invariants of containment types *anchors* and *body*. In Step (2), a *target*-edge from the created *Anchor*-node is required but cannot be inserted. (3) The required non-containment edge type *target* has not been fulfilled yet. I.e., the node of type *Anchor* has to be connected to a node of type *DisplayElement* but there is none. Step (4.1) tries to create the missing correlated node directly: The creation of a *DisplayElement*-node fails since there is no available *Section*-node (used as container node). Consequently, Step (4.2) is required which creates a *Section*-node inside of the existing *Body*-node. In Step (5), the completion is continued. I.e, a second iteration is required. *Iteration II:* In Step (1), two *DisplayElement*-nodes are created inside of the *Section*-node to fulfill the lower-bound invariant of the containment type *elements*. As an example, a node of type *TextBox* and a node of type *Image* are created. In Step (2), a non-containment edge of type *target* is inserted between the existing *Anchor*-node and one of the existing nodes of type *DisplayElement*, e.g., the *TextBox*-node. Step (3) finds out that all the required non-containment edge types are available now. In Step (6) all remaining values for the all required attributes are set. Here, the *url*-attribute value of the *URL*-node *calurl* is set to *myurl*. A valid instance model is produced (Step (7)).

3.3 Deriving a model transformation system from a meta-model

A pre-requisite of our repair algorithm is the existence of a model transformation system. It can be automatically generated from a given meta-model. In this section, we concentrate on the derivation of transformation rules and present a selection of rule kinds used for model repair. All rule schemes with examples can be found at [10]. They are generated from different kinds of meta-model patterns. The rule schemes are carefully designed to consider the EMF model constraints defined in [3]. They are generic and effective in the sense that the derived rule sets can be used to manage (fulfill) *any* lower or upper bound.

Set of required-node-creation rules: For *each* containment type *with* lower bound > 0, a rule is derived which creates a node with its containment edge if the *lower-bound invariant* of the corresponding containment type *is not yet reached*. The container is determined by the input parameter p. The user gets the possibility to select the right container, otherwise it is selected randomly by the algorithm. Figure 4 illustrates the rule scheme used to derive this kind of rules. Note that for each non-abstract class *B'* in the family of class B such a rule is derived. Figure 5 presents an example rule; it creates a required *Body*-node being contained in a *WebPage*-node if there is not already a *Body*-node contained. This set configures Step (1) of the algorithm.

Set of required-edge-insertion rules: For *each* non-containment type *with* lower bound > 0, a rule is derived which inserts a non-containment edge between two existing nodes if the *lower-bound invariant* of the corresponding non-containment type *is not yet reached*. This set configures Step (2).

Set of additional-node-creation rules: For *each* containment type of the given meta-model, a rule is generated which creates one node if *the upper-bound*

of its containment type *is not exceeded*. These rules are used to add nodes without violating upper-bound invariants. This set configures Step (4) of the algorithm.

Set of exceeding-edge-removing rules: For *each* non-containment type with a limited upper bound of the given meta-model, a rule is generated which removes one edge if the *upper-bound* of its non-containment type *is exceeded*. This set configures Step (a) of the algorithm.

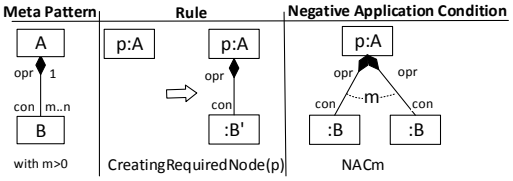

Fig. 4: Rule schema for creating a required *B*-node

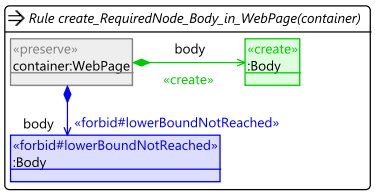

Fig. 5: An example rule for creating a required *Body*-node

4 Tool support

We have developed two Eclipse plug-ins based on EMF: The first Eclipse plug-in automates the derivation of a model transformation system from a given meta-model. The input of this tool is an Ecore meta-model that is rooted and fully instantiable, and the output is an Eclipse plug-in containing the corresponding model transformation system. The generated model transformation system is formulated in Henshin. If the meta-model contains OCL constraints, they are not taken into account yet. The second Eclipse plug-in implements the repair algorithm presented in Section 3. This plug-in has two main inputs: The model transformation system being derived from the given meta-model and an instance model of it containing one node of root type. The main output is the repaired model fulfilling all the EMF model constraints. Thus, the EMF editor can open the resulting model. Note, the tool reports the user if a new node of root type is needed to complete a given model. More information is at [10]. The algorithm can be *automatically* executed in two modes: (1) *randomly* or (2) *interactively*.

Rule-based implementation of automatic random or interactive mode: The algorithm is configured with the rule sets in a way that at any point of the algorithm where choices have to be made, they can be random, or interactive. This applies to choices of rules (repair actions), matches (model locations or targets) and attribute values. The suggested choices of each step aim at repairing the whole model. Each algorithm step uses proper rule sets, e.g., Step (1) uses the set of *required-node-creation* rules. Step (1) is ended once there is no applicable rule of the set on the model. I.e., all the required nodes with all their required child nodes are created. Since the rule sets are designed to consider the EMF model constraints, identifying the applicable rules and their matches aim at finding the proper repair actions and matches w.r.t the algorithm step and the given model state. To set attribute values of primitive types, the user can let this

be done automatically by the tool or has to provide a set of values. Note, the user can not only select each suggested choice but also can randomly perform the current algorithm step or the whole algorithm at any time.

Since model trimming may lead to information loss, the tool is developed to manage that in two ways: (a) The repair process is able to be interrupted (stopped) anytime so that the user can manage the needed data. Thereafter, the algorithm can be restarted. (b) Suggesting moving actions (if possible). For example, in Step (b) the types of exceeding nodes and their proper matches are provided as choices. At this end, the tool may provide two actions: (1) deleting an exceeding node and (2) moving an exceeding node to another container node (if exists) without violating the upper-bound of the target container node. However, if there is no available target container node, an additional container node can be automatically added (if possible) to hold the exceeding node.

An interesting test case that motivates us to carry out a scalability study in the future is that we have applied the tool to randomly resolve more than 150 inconsistencies of 3 different kinds in a model with 10,000 elements. The tool just took about 58 milliseconds to complete such a large model. The used meta-model composes 8 constrained element types and the test model is designed so that its structure is increased fairly w.r.t the model size: The test model is composed of copies of an initial model part containing elements of all given meta-model types. The initial model is first designed to be valid and then is randomly changed to have 8 inconsistencies of three different types, namely missing nodes, missing edges, and missing nodes with their edges. The test has been executed 3 times and the average is calculated, thereafter. More information can be found at [10].

5 Related work

In the following, we consider related work w.r.t. model repair and rule generation. We first relate our approach to existing model repair techniques that can be mainly categorized into syntactic and rule-based approaches on the one hand, and search-based approaches on the other hand.

Syntactic and rule-based approaches. In [11,12], the authors provide a syntactic approach for generating interactive repairs from full first-order logic formulas that constrain UML documents. The user can choose from automatically generated repair actions when an inconsistency occurs. Similarly, Egyed et al. [4,5] describe a rule-based technique for automatically generating a set of concrete changes for fixing inconsistencies at different locations in UML models and providing information about the impact of each change on all consistency rules. The designer is not required to introduce new model elements, i.e., the approach ignores the creation of new model elements as choices for fixing inconsistencies. In [14], Rabbi et al. propose a formal approach (with prototype tooling) to support software designers in completing partial models. Their approach is based on rule-based model rewriting which supports both, addition and deletion of model elements. In all these approaches, inconsistencies can be considered by the user one after the other; possible negative side effects are not taken into considera-

tion. It is up to the user to find a way to get a valid model (if any). Moreover, they are not shown to be fully consistent.

Search-based approach. A search-based repair engine starts with an inconsistent model state and tries to find a sequence of change actions which leads to a valid model. Another approach is model finding, using a constraint solver to calculate a valid model. Many approaches such as [1,6,8,17] provide support for automatic inconsistency resolution from a logical perspective. All these approaches provide automatic model completion; however, may easily run into scalability problems (as stated in [9]). Since the input model is always translated to a logical model, the performance depends on the model size. Badger [13] is a search-based tool which uses a regression planning algorithm to find a valid model. It can take a variety of parameters to let the search process be guided by the user to a certain extent. The authors argue that their generation of resolution plans is reasonably fast showing that the execution time is linear to the model size and quadratic to the number of inconsistencies. They do not show the time needed to apply a repair plan on a model. This approach may support some configuration before model repair but do not allow user interaction during model repair. Although being rule-based, the refinement approach in [16] basically translates a set of rules with complex application conditions to a logical model and thereby completes models. This completion is performed automatically by rule applications; user interaction is not considered in this process.

Our approach is designed for integrating the best of both worlds: It is a rule-based approach; therefore, it is easy to allow user interaction in the repair process, in contrast to search-based approaches. But it does not leave the resolution strategy completely to the modeler as in pure rule-based approaches. Instead, it guides the modeler in an automatic interactive way to repair the whole model. Our approach yields valid EMF models which can be opened by the EMF editor. How all the EMF constraints are specified and how valid EMF models are constructed are not clearly shown by most existing approaches. On the downside, our approach cannot yet handle OCL constraints being covered by most of the approaches mentioned above. It is promising to translate OCL constraints to graph patterns [2,15] functioning as application conditions of rules and thereby extending the automated interactive model repair approach.

Rule generation. In [7], model transformation rules are generated from a given meta-model as well. The main difference to our work is that consistency-preserving rules are generated there while we generate repair rules allowing temporarily inconsistent models w.r.t the multiplicities. Hence, rules are generated for different purposes: There, consistency-preserving rules are generated to recognize consistent edit steps, while we generate repair rules here to configure our model repair algorithm yielding consistent EMF models as results.

6 Conclusion

In this paper, we present a rule-based approach to guide the modeler in repairing models in an automated interactive way and thereby resolving all their inconsis-

tencies. Different sets of model transformation rules (repair actions) are derived from a meta-model considering it pattern-wise. A rule-based algorithm of model trimming and completion is presented yielding consistent EMF models. Two Eclipse plug-ins have been developed to automatically translate meta-models to model transformation systems and to repair corresponding instance models. First test cases show that our algorithm is fast and motivate us to carry out a scalability study. We plan to extend this approach to support OCL constraints as well. Translating OCL constraints to graph patterns [2,15] and further to application conditions of rules is promising to achieve an automated interactive model repair approach for meta-models with OCL constraints.

References

1. Apt, K.R., Wallace, M.: Constraint Logic Programming using Eclipse. Cambridge Univ. Press, Leiden (2006)
2. Bergmann, G.: Translating OCL to Graph Patterns. In: MoDELS, pp. 670–686. Springer (2014)
3. Biermann, E., Ermel, C., Taentzer, G.: Formal foundation of consistent EMF model transformations by algebraic graph transformation. (SoSyM) pp. 227–250 (2012)
4. Egyed, A.: Fixing Inconsistencies in UML Design Models. In: ICSE (2007)
5. Egyed, A., Letier, E., Finkelstein, A.: Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In: IEEE/ACM. pp. 99–108 (2008)
6. Hegedüs, Á., Horváth, Á., Ráth, I., Branco, M.C., Varró, D.: Quick fix generation for DSMLs. In: VL/HCC. pp. 17–24. IEEE (2011)
7. Kehrer, T., Taentzer, G., Rindt, M., Kelter, U.: Automatically Deriving the Specification of Model Editing Operations from Meta-Models. In: ICMT. Springer (2016)
8. Macedo, N., Guimarães, T., Cunha, A.: Model repair and transformation with Echo. In: ASE. pp. 694–697. IEEE (2013)
9. Macedo, N., Tiago, J., Cunha, A.: A Feature-based Classification of Model Repair Approaches. CoRR abs/1504.03947 (2015)
10. EMF Model Repair. http://uni-marburg.de/Kkwsr
11. Nentwich, C., Capra, L., Emmerich, W., Finkelstein, A.: xlinkit: A Consistency Checking and Smart Link Generation Service. ACM 2(2), 151–185 (2002)
12. Nentwich, C., Emmerich, W., Finkelstein, A.: Consistency management with repair actions. In: Software Engineering. pp. 455–464. IEEE (2003)
13. Puissant, J.P., Straeten, R.V.D., Mens, T.: Resolving model inconsistencies using automated regression planning. SoSyM pp. 461–481 (2015)
14. Rabbi, F., Lamo, Y., Yu, I.C., Kristensen, L.M., Michael, L.: A Diagrammatic Approach to Model Completion. In: (AMT)@ MODELS (2015)
15. Radke, H., Arendt, T., Becker, J.S., Habel, A., Taentzer, G.: Translating Essential OCL Invariants to Nested Graph Constraints Focusing on Set Operations. In: Graph Transformation, pp. 155–170. Springer (2015)
16. Salay, R., Chechik, M., Famelis, M., Gorzny, J.: A Methodology for Verifying Refinements of Partial Models. Journal of Object Technology pp. 3:1–31 (2015)
17. Sen, S., Baudry, B., Precup, D.: Partial Model Completion in Model Driven Engineering using Constraint Logic Programming. In: In Proc. INAP'07 (2007)
18. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley Professional, 2nd edn. (2009)