ReVision: A Tool for History-based Model Repair Recommendations

Manuel Ohrndorf, Christopher Pietsch, Udo Kelter
University of Siegen
{mohrndorf,cpietsch,kelter}@informatik.uni-siegen.de

Timo Kehrer
Humboldt-University of Berlin
timo.kehrer@informatik.hu-berlin.de

ABSTRACT

Models in Model-Driven Engineering are heavily edited in all stages of software development and can become temporarily inconsistent. In general, there are many alternatives to fix an inconsistency, the actual choice is left to the discretion of the developer. Model repair tools should support developers by proposing a short list of repair alternatives. Such recommendations will be only accepted in practice if the generated proposals are plausible and understandable. Current approaches, which mostly focus on fully automatic, non-interactive model repairs, fail in meeting these requirements. This paper proposes a new approach to generate repair proposals for inconsistencies that were introduced by incomplete editing processes which can be located in the version history of a model. Such an incomplete editing process is extended to a full execution of a consistency-preserving edit operation. We demonstrate our repair tool ReVision using a simplified multi-view UML model of a video on demand system, a screencast is provided at *http://pi.informatik.uni-siegen.de/projects/SiLift/icse2018/*.

CCS CONCEPTS

• **Computing methodologies → Model development and analysis**; • **Software and its engineering → Software defect analysis**;

KEYWORDS

Model repair, consistency, recommendations, history analysis

ACM Reference Format:
Manuel Ohrndorf, Christopher Pietsch, Udo Kelter and Timo Kehrer. 2018. ReVision: A Tool for History-based Model Repair Recommendations. In *Proceedings of 40th International Conference on Software Engineering , Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18 Companion),* 4 pages.
https://doi.org/10.1145/3183440.3183498

1 INTRODUCTION

This paper presents a new interactive repair tool for models which exploits information about the editing history of a model to generate useful repair recommendations. We assume a Model-Driven Engineering (MDE) approach of software development. Here, models are primary development artifacts which are frequently edited. During such development processes, models are often not "perfectly" correct in the sense that they temporarily violate advanced consistency constraints of their modeling languages. This occurs particularly often if systems are modeled from several viewpoints using different types of models, which is a widespread paradigm in managing the complexity of large-scale software-intensive systems.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3183498

1.1 Example: a Video on Demand (VoD) System

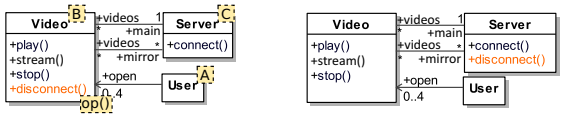
Fig. 1 shows a simplified static structure of a VoD system consisting of three classes. A user can open a video stream, a server provides the video data. The sequence diagram in Fig. 2 shows the dynamic interactions between some entities of the class diagram in a simple "start and stop the video stream" scenario. The signatures of the messages refer to operations of classes, as indicated by their names. This is a typical example on how multiple views depend on each other by means of a distributed consistency constraint. Our tool addresses inconsistencies in models introduced by isolated editing of single views or model fragments, or by model merging, or more generally, by *incomplete editing processes*. Further examples may be found in the literature [3, 4, 7, 10] and on the accompanying website of this paper (for URL, see Abstract).

1.2 Related Approaches to Model Repair

The most appropriate repair approach for the type of defects described above are recommendation systems, rather than non-interactive approaches, according to the classification of software repair approaches in [5]. The role of a recommender system is to generate a ranked list of useful repair proposals, from which a human developer can either choose one or revert to a hand-crafted solution. Typically, each single defect is repaired separately.

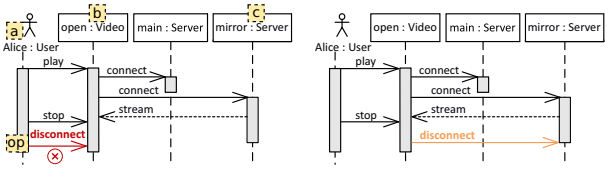
Most approaches to model repair [4] aim at a fully automated repair process. One common approach is to employ a (heuristic) search strategy, which often leads to a huge number of repair alternatives and to repair proposals which are hardly comprehensible for developers. For instance, the model repair tools *Echo* [3], *Badger* [7] and *EMF Model Repair* [6] fall into this category.

The only available recommender tool known to us is the *Model/analyzer* [8]. It handles one single consistency violation at a time and first determines the model elements affected by an inconsistency. For each element, a so-called abstract repair action is generated which gives a rough hint of how to modify this element in order to improve consistency. An inconsistency is resolved by choosing and instantiating a subset of the set of generated abstract repair actions. However, the number of proposals is still large, and the user has no guidance for assessing whether the effect of the proposed repairs is appropriate.



(a) Version A (b) Version B

Figure 1: Modification in the UML class diagram



(a) Version A (b) Version B

Figure 2: Inconsistency in the sequence diagram

All approaches to generate repairs known by us do not consider the origin of a defect, they are not aware of the change history of a model. They are likely to propose repairs which just undo former changes which have caused the inconsistency. Our tool is based on the hypothesis that plenty of these changes represent incomplete editing processes which should be retained and completed.

1.3 Contribution

We present a new model repair technique which addresses the above issues. We iteratively repair each single violation of a consistency rule (as, e.g., in [8]). Our tool, which assumes a model history to be available, has the following distinguishing features:
(a) It detects the set of former model changes which have caused this inconsistency.
(b) It generates repair proposals which extend this set of former model changes to a complete consistency-preserving edit operation (CPEO). Such a complementing edit operation is a case-specific, concrete repair operation which extends the partial execution of the CPEO into a complete one. This way, implausible repair alternatives can be largely avoided. We also prevent undoing former edit steps, which is both a feature and a current limitation of our approach. An inconsistency cannot be resolved by our approach if no suitable CPEO is available. The generated repair proposals are ranked using properties of the repair operation.

Our tool, called ReVision, is implemented on top of the widely used Eclipse modeling technology stack. It is publicly available at *http://pi.informatik.uni-siegen.de/projects/SiLift/icse2018/*.

2 CONSISTENT MODEL EDITING

2.1 Consistency Rules

Modeling languages such as the UML define many advanced well-formedness rules to which we refer to as *consistency rules*. Consistency rules are formulated using a dedicated constraint language, typically based on first-order logic such as the Object Constraint Language (OCL). A consistency rule is a logical expression that is evaluated on a *context element* of the model. Consistency checking tools can evaluate such a consistency rule on a model and report, for each inconsistency found, the context element and other model elements involved in this evaluation.

The following (informally specified) consistency rules apply in our example:

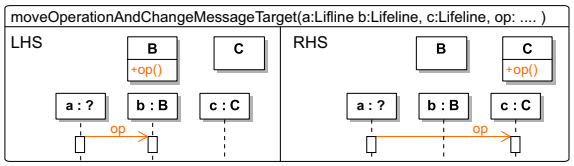


Figure 3: Move operation and change message target

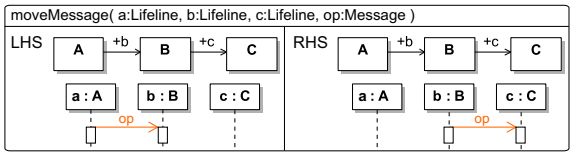


Figure 4: Move message

1. `message_signature(m:Message)`: The signature of a message in a sequence diagram must be identical to the signature of an operation of the class referenced by the receiving lifeline.
2. `message_association(m:Message)`: For a message in a sequence diagram, its sender and receiver classes in the corresponding class diagram must be connected by an association which is navigable from the sender to the receiver.

Both rules are satisfied in version A of our example in Fig. 1. Now we edit only the class diagram, we move the operation `disconnect()` from the class `Video` to the class `Server`, and obtain version B of the class diagram. The validation of the consistency rule `message_signature(m:Message)` will fail now, i.e., there is an inconsistency between version B of the class diagram and version A of the sequence diagram.

A complementary modification in the sequence diagram is required. The message `disconnect`, sent between the lifelines `Alice:User` and `open:Video`, has to be received by an object of class `Server`, which owns the operation `disconnect()`, i.e., by the lifeline `main:Server` or `mirror:Server`. To resolve the inconsistency we perform a *corrective edit step* in the sequence diagram, we change the target end of the message `disconnect` to object `mirror`.

The first consistency rule is now satisfied, however our first corrective edit step has introduced a new defect: the second rule is violated now because the object sending the message is no longer (directly) connected to the object receiving the message. This inconsistency can be fixed in various ways, one option is to move the source end of the message from lifeline `Alice` to lifeline `Video`.

We will later refer to the above edits, i.e., (1) moving the operation `disconnect()`, (2) changing the target end of message `disconnect` and (3) changing the source end of message `disconnect` as *edit step (1), (2) and (3)*, respectively.

2.2 Consistency-preserving Edit Operations

Inconsistencies caused by incomplete editing processes as in our example of Sect. 2 can be avoided by using more complex edit operations operating on multiple model fragments or views. For instance, the compound effect of edit steps (1) and (2) of our example could be performed in single edit step preserving the consistency rule `message_signature(m:Message)`. Although typically not offered by standard model editors, an according complex edit operation can be specified by a parameterized in-place model transformation rule.

An example is illustrated in Fig. 3. This rule representation, while being based on graph transformation concepts [1], uses the concrete syntax of the UML for the sake of readability. Roughly speaking, the left-hand side (LHS) of the rule depicts a model pattern to be found in a model and to be replaced by a copy of the rule's right-hand side (RHS). If we apply the rule `moveOperationAndChangeMessageTarget` onto *Version A* of our example, the operation `disconnect()` is moved from the class `Video` to the class `Server`, and the target of message `disconnect` is changed onto the lifeline `mirror:Server`. Annotations depicted in Fig. 1a and Fig. 2a indicate the occurrence of the rule's LHS in our example (e.g., rule element `op()` is mapped to model element `disconnect()`). Likewise, the shifting of the source and target end of a message (edit steps (2) and (3) of our example) can again be specified by a complex edit operation as shown in Fig. 4.

Note that complex edit operations like the ones above prevent the violation of a subset of the set of consistency rules. For example, the violation of consistency rule `message_signature(m:Message)` is prevented by the edit operation `moveOperationAndChangeMessageTarget`). We thus refer to these complex edit operations as *consistency-preserving edit operations (CPEOs)*. As we will see in the remainder of this section, we use CPEOs as the main configuration input of our tool ReVision in order to adapt its behavior to a given modeling language.

3 REPAIR RECOMMENDATIONS

3.1 Repair Operations as Complementing Edits

A restrictive editing environment would allow the developers to use only CPEOs as edit operations. However, this would make editing very clumsy, similar to syntax-driven editors which are not very popular in practice. Moreover, developers typically follow a task-oriented model editing process (e.g., focusing on a dedicated view as in our example). As a consequence, inconsistencies have to be tolerated temporarily but must be resolved eventually.

The main idea of our approach is to consider CPEOs as ideal edit operations and to recommend the "gap" between ideal edits and the edits which have caused an inconsistency as model repairs. The suggested repair operations will complement incomplete and inconsistent edit steps. The tool works offline, i.e., no monitoring of the local workspace modifications is needed. The (incomplete) edit steps are reconstructed from the model versions stored in a version control system.

Fig. 5 shows the user interface of ReVision, placed in an additional view next to the model editor. The repair process starts with the validation of the model (1). All inconsistencies that could be found in the model are reported to the developer. After that, the developer must select the inconsistency to be repaired (2). The tool now extracts from the model history the model version in which the inconsistency was non-existent and calculates the changes between the former consistent version and the current inconsistent one. If some of these changes match an incomplete application of a given CPEO (3) which can be complemented by changes improving the inconsistency (4), a repair is reported (5). In other words, a repair alternative can be considered as a specific edit operation of which some parts have been already executed and some need to be executed in order to achieve the consistency improvement.

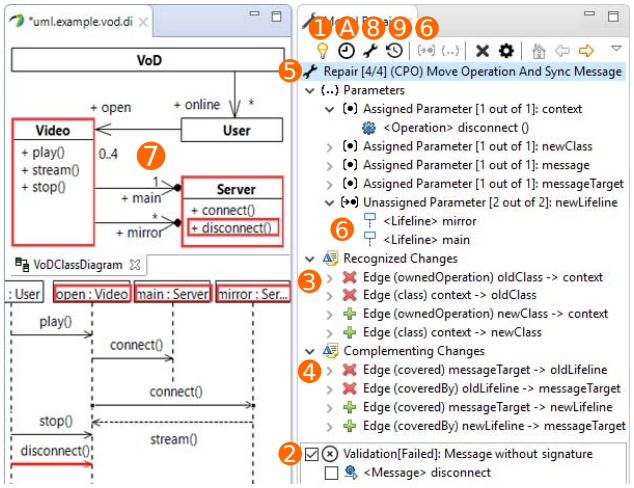


Figure 5: UI of ReVision with diagram highlighting.

Parameters are automatically assigned to a repair operation as long as their assignment is unique; otherwise a specific parameter value must be assigned (6) by the developer. In our example the parameter `newLifeline`, which is the new target for the message `disconnect`, could be assigned with the lifeline `main:Server` or `mirror:Server`. Valid parameter values are restricted to all representations of the `Server` class, since it owns the operation `disconnect()` involved in the inconsistent edit step. Furthermore, all parameters and all model elements affected by a repair can be highlighted in the model editor on the left (7). After all parameters of a repair are assigned, the repair can be applied (8).

3.2 Iterative Repair Process

Following the requirements of model repair tools which are used as recommender systems [5], the understandability of proposed repairs is of utmost importance, while partial solutions may be accepted. On purpose, a repair recommended by our approach is not guaranteed to be free of negative side effects since this would eliminate obvious repairs such as edit step (2) of our example.

Thus, performing a model repair automatically triggers a re-validation of the resulting model. If the repair had a negative side effect, then all new inconsistencies have to be repaired subsequently. In our example, this is the case for the sender of the message `disconnect` in our running example, since the class `Server` is not reachable from the class `User` through a navigable association. In the next repair step, the sender must be set to the lifeline `open:Video` (cf. manually performed edit step (3) in Sect. 2). This behavior may be achieved by the CPEO shown in Fig. 4. This time, the previous repair, that changes the receiver of the message `disconnect`, will be recognized as an incomplete edit step. ReVision will propose to change the sender of the message from lifeline `online:User` to `open:Video` to complement this edit step. After this repair is applied, the consistency of the VoD-System is restored.

All applied repairs are stored on an edit stack, so every edit step can be undone (9). This tool functionality helps the developer to understand the effect of repair proposals in a "trial an error process". A repair can also be visualized by opening and highlighting the

modified model elements in the historic version (A). This enables the developer to understand the changes causing the inconsistency.

4 CALCULATION OF RECOMMENDATIONS

We now briefly discuss the technical details of calculating repair recommendations for a given inconsistency. In the initial history analysis phase, we trace back the inconsistency to its origin. Starting from the current inconsistent model version, we search for the latest version in which the inconsistency does not occur by identifying and then validating the context element in former model versions. The identification of corresponding model elements in different versions is supported by a model matcher, the respective traces can be stored incrementally in the version control system. The next step calculates the difference, i.e., the set of changes (basically additions and deletions of model elements), between the last consistent version and the current inconsistent one.

The main novelty of ReVision is the recognition of partially applied CPEOs in this difference. The application of a complete CPEO produces a set of changes. Such a change set can be recognized by searching the corresponding pattern of changes in the difference [2]. Each syntactically correct sub-operation *SO* of a CPEO leads to a sub-pattern of changes. Basically, all possible sub-patterns of all CPEOs have to be found within the difference. If the application of the recognized sub-operation *SO* has led to changes that introduced the inconsistency, then we search for a complementing repair. The effect of a change regarding a concrete inconsistency can be determined by analyzing the evaluation of a consistency rule [8]. Given an instance of a sub-operation *SO*, a complement rule for *SO* containing the remaining changes of the CPEO can be constructed [10]. Among all possible applications of this complement rule to the current model version, those applications having an improving effect on the inconsistency are considered as potential repairs.

Finally, those complement rule applications that can improve the inconsistency are ranked by the ratio of historic and complementing changes, preferring large sub-rules and small complement rules. A large overlap with the history can be seen as an indication of an uncompleted edit step, while a small complement rule leads to a repaired model that is close to the original model.

5 VALIDATION PLAN

Interactive repair recommendation systems propose tentative patches to developers, thus the *understandability* and the *plausibility* of the proposed short list of repairs is of primary importance [5]. We demonstrated the usefulness of our repair tool by highlighting its qualitative advantages over existing approaches. To quantitatively evaluate our approach and to better assess its benefits and limitations w.r.t. related repair tools (see Sect. 1.2), we plan an offline experiment [9] using data sets obtained from real-world modeling projects, driven by the following research questions:

RQ1 (Coverage): *How many inconsistencies can be resolved by our approach?* This aims at assessing limitations of our approach to resolve inconsistency by completing partial edits.

RQ2 (Relevance): *Given a resolvable inconsistency, do we generate a repair alternative reflecting the changes that have been performed to actually resolve the respective inconsistency?* This aims at assessing whether proposed repair operations meet a developer's intention of how to resolve an inconsistency.

RQ3 (Efficiency): *How many repair alternatives must be inspected by developers until they find a relevant one?* This aims at assessing how quickly developers may pick a relevant repair from an ordered set of generated repair alternatives.

RQ4: (Performance): *How long does it take to generate the repair alternatives for a given inconsistency?* This aims at assessing the usefulness of our interactive tool which should generate repairs in acceptable latency times in the order of seconds.

Subject Selection. We plan to use Eclipse modeling projects hosted in the Eclipse Git repository as experimental data. Consistency violations and inconsistency resolutions can be observed in the version history of a model. Thus, in order to validate our repair recommendations, we will have an oracle at hand which tells us how an inconsistency has been actually resolved by developers.

Tool Configuration. An important aspect of our evaluation is the impact of CPEOs on the quality attributes addressed by our research questions. Currently, a set of CPEOs which avoid typical inconsistencies needs to be specified by a domain expert. If available, project- or language-specific editing style guides provide a promising starting point for this. In general, as illustrated in Sect. 2.2, the most interesting kinds of CPEOs for our approach are creations and deletions of complex model fragments, typically ranging over different modeling views. Further examples of this are provided on the accompanying website of this paper (for URL, see Abstract). A promising direction for future research is to learn such CPEOs from the development history of a project.

ACKNOWLEDGMENT

This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future - Managed Software Evolution.

REFERENCES

[1] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. 2006. *Fundamentals of Algebraic Graph Transformation.* Springer.
[2] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. 2011. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *Proc. Intl. Conf. on Automated Software Engineering*. IEEE CS, 163–172.
[3] Nuno Macedo, Tiago Guimaraes, and Alcino Cunha. 2013. Model repair and transformation with Echo. In *Proc. Intl. Conf. on Automated Software Engineering*. IEEE Press, 694–697.
[4] Nuno Macedo, Tiago Jorge, and Alcino Cunha. 2016. A feature-based classification of model repair approaches. *IEEE Transactions on Software Engineering* (2016).
[5] Martin Monperrus. 2014. A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In *Proc. Intl. Conf. on Software Engineering*. ACM, 234–242.
[6] Nebras Nassar, Hendrik Radke, and Thorsten Arendt. 2017. Rule-Based Repair of EMF Models: An Automated Interactive Approach. In *Intl. Conf. on Theory and Practice of Model Transformations*. Springer, 171–181.
[7] Jorge Pinna Puissant, Ragnhild Van Der Straeten, and Tom Mens. 2015. Resolving model inconsistencies using automated regression planning. *Software & Systems Modeling* 14, 1 (2015), 461–481.
[8] Alexander Reder and Alexander Egyed. 2010. Model/analyzer: a tool for detecting, visualizing and fixing design errors in UML. In *Proc. Intl. Conf. on Automated software engineering*. ACM, 347–348.
[9] Guy Shani and Asela Gunawardana. 2011. Evaluating recommendation systems. *Recommender systems handbook* (2011), 257–297.
[10] Gabriele Taentzer, Manuel Ohrndorf, Yngve Lamo, and Adrian Rutle. 2017. Change-Preserving Model Repair. In *Intl. Conf. on Fundamental Approaches to Software Engineering*. Springer, 283–299.