A Feature-based Classification of Model Repair Approaches

Nuno Macedo, Tiago Jorge, and Alcino Cunha

Abstract—Consistency management, the ability to detect, diagnose and handle inconsistencies, is crucial during the development process in Model-driven Engineering (MDE). As the popularity and application scenarios of MDE expanded, a variety of different techniques were proposed to address these tasks in specific contexts. Of the various stages of consistency management, this work focuses on inconsistency handling in MDE, particularly in model repair techniques. This paper proposes a feature-based classification system for model repair techniques, based on a systematic literature review of the area. We expect this work to assist developers and researchers from different disciplines in comparing their work under a unifying framework, and aid MDE practitioners in selecting suitable model repair approaches.

Index Terms—Model-driven Engineering, Consistency Management, Inconsistency Handling, Model Repair.

1 INTRODUCTION

MODEL-DRIVEN Engineering (MDE) is a family of development processes that focus on models as the primary development artifact. As models are modified by different stakeholders, in a possibly distributed and heterogeneous context, the consistency of the overall MDE environment must be constantly monitored and managed. Therefore, consistency management [1], [2] – which involves various activities concerned with the detection, diagnosis, handling and tracking of inconsistencies – is essential to MDE. Such activities are not only fundamental to managing intra- and inter-model consistency as models naturally evolve, but also in more specific activities, like meta-model and constraint evolution [3], model refactoring [4], variability modeling [5] or version merging [6].

1.1 Model Repair

Inconsistencies may arise due to mistakes or imprudent decisions as the developers apply changes to the models, but their impact may not be immediately perceptible, especially considering the complexity of the MDE development environment. Inconsistencies may also reflect conflicting or alternative interpretations of the requirements, or uncertainty and partial knowledge [7]. Thus, development frameworks should not forbid the introduction of inconsistencies altogether, but instead tolerate them, while still providing support for their detection [8]. Notwithstanding, as the development progresses and conflicting interpretations converge, so are the models expected to evolve to a consistent version, and thus inconsistencies must eventually be handled [2]. To be manageable, these tasks must be supported by automated techniques that help the user decide how to restore the consistency of the environment. A common solution, addressed in this study, is to rely on techniques that propose update actions that repair the models themselves, in order to ameliorate the consistency level of the MDE environment.

One of the main challenges in model repair is that for any given set of inconsistencies, there (possibly) exists an overwhelming number of updates that resolve them. Yet, since the selection of the most suitable repair update is ultimately a choice of the developer, approaches to model repair must balance the level of automation of the process with the need for user guidance in the generation of the alternative solutions. Some authors [9] advocate the use of heuristics to tackle the presence of a large search space, the need for algorithms with a low computational complexity, and the absence of known optimal solutions. Others [10] advocate against fully automatic approaches that replace the role of the human designer in repairing models. According to the latter, repairing models should be an activity that goes hand in hand with the creative process of modeling.

1.2 Need for a Unifying Taxonomy

The variety of contexts in which consistency management is necessary gave rise to an equally disparate terminology. As a clear example, techniques addressing seemingly interchangeable problems identify themselves varying as handling [11], resolving [12], fixing [13] or repairing [14] inconsistencies, among others. Moreover, to render these tasks more manageable, a variety of techniques have been developed that assume a more controlled environment with more concrete goals, including change propagation [15], model synchronization [16], bidirectional model transformation [17], [18], incremental model transformation [19] or model finding [20], each with particular terminology. Thus, there is the need for a unifying taxonomy that allows practitioners to properly compare their work with that arising from different disciplines. To be rigorous and exhaustive, such classification scheme must necessarily emerge from a systematic review of the literature relevant to the model repair problem [21].

Yet, to the best of our knowledge, the most rigorous study to date on consistency management, including incon-
sistency handling, is still the survey by Spanoudakis and Zisman [2], which, based on previous definitions from [1] and [22], surveyed and analyzed existing approaches at the time. A more recent classification of model repair techniques is presented in [23], addressing the flexibility, usability and extensibility of the approaches. However not every facet of the model repair problem is addressed, like the behavior of the repair procedure or the different mechanisms through which the user can control it. Moreover, its development was not based on a systematic review of the state-of-the-art.

Classification schemes have been proposed for related areas like model transformation [24], model synchronization [16] and bidirectional transformation [25]. While some facets of model repair overlap with facets from those disciplines, there are various topics that are specific to the former and that are not addressed by those studies.

1.3 Goals and Contributions

Motivated by the heterogeneity of approaches to model repair, this paper explores this landscape and proposes a structured taxonomy for their classification, based on a systematic literature review of the area.

We adopt the term of model repair as the focus of the study because we feel that it best defines the topic which we aim to address: techniques that handle inconsistency by acting upon software models. Here we assume a broad definition of model, encompassing any artifact that abstracts certain portions of a software system. This excludes from the study the detection of the inconsistencies and their causes, techniques that avoid the introduction of inconsistencies by enforcing consistent states, and techniques that handle inconsistencies by updating artifacts other than the models (e.g., the meta-models and associated constraints).

Following other successful classification schemes of MDE techniques (e.g., [24] for model transformation), we present our classification alternatives as feature models [26], diagrams developed with the goal of modeling alternative configurations in software product lines. This allows the presentation of the identified characteristics in a structured and formal way, rendering their dependencies explicit.

This unifying taxonomy is the main contribution of this paper, which will allow researchers and tool developers to properly locate novel approaches in the context of the state-of-the-art of the area. As a secondary contribution we provide the exhaustive classification of the studies collected during the literature review under this taxonomy [27]. We expect that this will aid researchers in identifying gaps in the field by detecting under-explored features or feature combinations representing potentially interesting approaches. Lastly, we present a detailed classification and comparison of three modern approaches to model repair to demonstrate the impact of the feature selection. Hopefully this will provide software engineering practitioners with some insight when choosing a model repair approach for their particular application domain.

This remainder of paper is structured as follows. Section 2 starts by presenting and formalizing the model repair problem, in order to clarify the artifacts that are to be classified by the taxonomy. Section 3 presents the resulting feature-based taxonomy under which model repair techniques can be classified, as well as an overview of the methodology employed to select the primary studies and extract from them the selected features. This methodology is further detailed in the Appendix. This taxonomy is used in Section 4 to classify and compare three modern techniques. Lastly, Section 5 draws conclusions and final remarks.

2 Model Repair

This section presents and formalizes the problem of model repair, the target of this study. The scheme allows us to concretely identify the artifacts that are to be classified by each facet of the taxonomy.

2.1 Overview

To provide an overview of the model repair problem and illustrate the vastness of features that model repair techniques may implement, this section introduces a couple of examples, inspired by state-of-the-art approaches to the problem [9], [10], [28].

While many approaches to model repair are designed to focus on particular classes of models (e.g., UML diagrams [29]), most modern approaches are meta-model independent: they allow the designers to restrict the model domain space on which they act, improving their versatility. This is achieved by defining well-formedness rules using meta-modeling languages provided by popular modeling frameworks like OMG’s Model-driven Architecture (MDA) or the Eclipse Modeling Framework (EMF). Fig. 1 depicts one such meta-model, for designing very simplified versions of class and sequence diagrams.

Although meta-models define which model instances are considered well-formed, there are a number of structural and behavioral properties that cannot be captured by meta-models alone. Thus, they are usually annotated with additional intra- and inter-model constraints that restrict the internal structure of individual model instances and their relationship with others, respectively. Ideally, the user should be allowed to define such constraints, typically using the well-established MDA’s OCL [30] or other similar constraint language. One such constraint over class diagrams is that class generalization links must be acyclic. In OCL, this can be defined as follows for the meta-model depicted in Fig. 1:

```
context Class acyclic_generalization:
  not self.closure(general) -> includes(self)
```
Consider, as an example, the class diagram from Fig. 2a conforming to the meta-model from Fig. 1 depicting a tentative first version of the structure of a video on demand (VOD) system (inspired by [9]), consistent under the acyclic_generalization constraint. Then, assume that at some point one of the developers, maybe oblivious of the whole inheritance tree or maybe disagreeing with previous design decisions, updated that model instance to the version depicted in Fig. 2b by introducing a new generalization link (colored red), giving rise to a violation that breaks acyclic_generalization. Since user updates can evidence conflicting interpretations of the requirements, inconsistencies should not be forbidden but rather detected, diagnosed and handled when deemed necessary.

There are a variety of updates that can be applied to the model instances to handle inconsistencies and ameliorate the consistency level of the environment. However, the alternatives generated by the model repair procedures are necessarily restricted by design choices that render the problem manageable. Should a single repair alternative be generated, even if the rational behind the choice is not clear to the user, or should the enumeration be exhaustive at the risk of overwhelming the user? Should the procedure attempt to infer all required information to repair the model instance or generate abstract plans that must be instantiated by the user? The process is also dependent on the amount of information available. Should the modeling tools work in an online setting and record the user actions that lead to violations, allowing more accurate repair alternatives? Moreover, the ability of the tool to consider domain-specific information provides additional complexity. Should the procedure be able to handle constraints specified by the stakeholders? Should the supported repair actions be defined by the stakeholders? Finally, all these design choices must also take into consideration the ability of the user to customize the procedure so that the generated alternatives prove useful. Should this be achieved by asking the user to provide repair hints or simply assigning priorities to different constraints or parts of the model? Should user input be collected at static time or should the repair procedure be interactive?

The problem becomes more complex when various constraints interfere with each other, which is the frequently the case. Consider the coexistence of class and sequence diagrams, supported by the meta-model depicted in Fig. 1. Besides internal consistency of the diagrams, consistency between them must also be maintained because some data contained in the two diagrams overlaps: messages refer to operations that must be available in the target lifeline’s class.

Since we have assumed that both kinds of diagrams share the same meta-model (much like UML diagrams), this kind of properties can still be defined as regular OCL constraints. This one in particular would take the shape:

```
context Message message_operation:
  self.target.class.operations->exists(o | o.name = self.name)
```

These constraints must coexist with those over the individual diagrams. For instance, another constraint that must hold in class diagrams is that the operations defined within a class must have unique names:

```
context Class unique_operations:
  self.operations->forall(x,y | x.name = y.name => x = y)
```

The class and sequence diagrams from Fig. 3a are consistent under the constraints that have been defined. However, if the two user updates depicted in Fig. 3b were simultaneously applied to these model instances – the introduction of a new operation and a new message (both colored red) – violations would be introduced for both message_operation and unique_operations.

When attempting to remove the violation of the message_operation constraint, the developer should be aware of the impact that each of the acceptable repair updates has on the other constraints. Fig. 4 depicts several possible repair updates that can be applied to the class diagram or to the sequence diagram that remove the message_operation violation. However, some of these repair updates have (possibly undesirable) side effects: the update applied in Fig. 4a also solves the violation caused by the unique_operations – a positive side effect – while the one applied in Fig. 4c introduces a new violation by breaking acyclic_generalization – a negative side effect. Either way, it is important that the user is aware of these side effects when choosing the fix to be applied, and thus model repair procedures should somehow consider all constraints when generating the repair updates. In this example it is also manifest that the number of valid repair updates can quickly become too large for the user to handle. Thus, a
variety of techniques have been proposed that try to balance the automation provided by the repair procedures and the user input required to reduce the number of generated repair updates. This input includes, for instance, requiring the definition of repair hints for each specified constraint, assigning different priorities to those constraints or parts of the model, or even disabling some edit operations.

As techniques were developed to handle more complex application domains, more specialized mechanisms to manage their consistency emerged. Such is the case of techniques designed to manage the consistency of models spread across heterogeneous modeling frameworks. A classical example of such scenario is the object-relational mapping, concerned with keeping class diagrams consistent with relational database schemas, so that data conforming to the former can be persisted in databases conforming to the latter. In such cases, unlike the UML sequence and class diagrams of the previous example, overlapping information can not be directly detected, and thus dedicated mechanisms to define inter-model consistency are required, like defining traceability links or consistency relations, as advocated in MDA’s QVT Relations [31]. Dedicated to manage inter-model consistency, such techniques often disregard intra-model constraints altogether.

It is easy to envision the complexity of model repair procedures over a considerable number of model instances and inter-related constraints, giving rise to multiple violations and an overwhelming number of acceptable repair updates. The goal of this paper is to interpret the myriad of solutions that have been proposed to address this kind of problems under a unifying framework.

### 2.2 Formalization

In order to properly classify model repair techniques, one must first formally define the artifacts from the MDE environment that are relevant in that context. In particular, the shape and characteristics of these artifacts has a direct impact on which functionalities a model repair framework may possess, as well as the functional properties of the comprising procedures. This section presents such scheme.

We assume that a meta-model $M$ defines a set of well-formed model instances $m \in M$, which the model repair technique may allow the user to define through a metamodeling language. The domain space of a model repair approach is defined by $k$ meta-models $M_1, \ldots, M_k$. In that sense, each state of the MDE environment is comprised by $k$ model instances $m_1, \ldots, m_k$ that conform to $M_1, \ldots, M_k$, a fact denoted by $(m_1, \ldots, m_k) \in M_1 \times \cdots \times M_k$. In practice, this product of meta-models can be seen as a single composed meta-model $M$, to which the tuple $(m_1, \ldots, m_k)$ (usually abbreviated as $m$) conforms. The shape and properties of $M$ in a model repair approach essentially determine the design space on which both the user and the repair procedures may act.

Although $M$ defines the structural consistency of model instances, semantic properties must be enforced by additional constraints defined over the meta-models. Depending on the technique, these may take different shapes and varied expressiveness (e.g., intra- vs. inter-model constraints). We denote the universe of constraints supported by a model repair technique by $C$. Only a subset of model instances from $M$ is considered consistent under a constraint $c \in C$; for the other model instances there is at least a violation to $c$. There is usually a set of constraints $\{c_1, \ldots, c_l\} \subseteq C$ specified in the MDE environment, which may or not have been defined by the user, which are abbreviated as $c$. The notion of inconsistency considered in this study is imposed by these constraints (as opposed to inconsistency rising due to uncertainty or partial knowledge, for example). This is not necessarily a limitation since formalization imposes no restriction over the expressiveness of these constraints. The shape of constraints $C$ determines the kind of properties that the framework will be able to handle, while the support to specify them affects the user’s ability to customize it.

Prior to being handled, inconsistencies must be detected and diagnosed. Since inconsistencies are introduced by the different stakeholders as the models evolve, information regarding the performed user updates may help the checking and repair procedures execute quicker and produce more accurate results. In the simplest case they amount to the model instances resulting from the user update, but they may also contain additional information, like the edit actions applied by the user. We denote the universe of user updates supported by each approach by $U$. In general, each user update $u \in U$ contains at least information about the post-state model instances $m' \in M$, which can be retrieved by $\text{post}(u)$. For instance, in frameworks that record the user’s edit actions, user updates may be represented by a pair $(m, s)$, where $m$ is the state of the environment prior to the update and $s$ denotes the applied edit actions. In such cases, the post-state model instances are retrieved by applying $s$ to $m$, i.e., $\text{post}(m, s) = s(m)$. If available, we denote the operation that retrieves the state of the environment prior to a user update $u$ by $\text{pre}(u) \in M$. The information contained in $U$ directly affects the accuracy and predictability of the repair procedures.

Given a user update, a consistency checking procedure will test whether the resulting model instances are consistent for a provided set of constraints. The information reported by these procedures may be as simple as a boolean value, or more structured information, like a set of detected violations. We denote the universe of these reports by $I$, which is instantiated by each approach. Such checking reports can be compared for their “inconsistency level”, e.g., when some violations are handled, the environment becomes “more consistent” but may still not be “fully consistent”. Following the approach proposed by Stevens [32], we assume these inconsistency levels to form a partially ordered set $(I, \sqsubseteq)$. In general, but not necessarily, this partially ordered set has a least element denoting the highest level of consistency for the environment, which will be denoted by $\bot_I$.

**Definition 1 (consistency checking).** A consistency checking procedure $\text{CHECK} : \mathbb{P}C \rightarrow U \rightarrow I$ calculates the inconsistency level $i \in I$ for an update $u \in U$ under constraints $c \subseteq \mathbb{C}$, which is denoted by $i = \text{CHECK}_c(u)$.

The features of the CHECK procedure and the information contained in the detected $I$ levels, not only affect the user’s ability to understand and control the behavior of the framework, but also define the information available to
the subsequent repair procedure when generating possible repair updates.

Model repair procedures are deployed when the stakeholders wish to decrease the level of inconsistency of the environment. Again, the generated repair updates may contain varied information, from simple model instances to a set of edit operations. The universe of repair updates of each model repair approach is denoted by \( \mathcal{R} \). The information contained in the repair updates \( \mathcal{R} \) is not necessarily the same as the user updates \( \mathcal{U} \), e.g., approaches may consider only the post-state of user updates but still propose edit sequences as repair updates. It is however assumed that from a repair update \( r \in \mathcal{R} \) and a user update \( u \in \mathcal{U} \) that led to the current model instances, an update \( u' \in \mathcal{U} \) can be derived that applies \( r \) to \( u \) (otherwise, the consistency checking procedure could not be executed after the application of repair updates). For instance, if \( u \) is simply represented by the post-state of the environment after a user update, and \( r \) is a set of edit operations, the updated \( u' \) can be retrieved by applying the \( r \) operations to the \( u \) update. We denote this operation by \( r(u) \in \mathcal{U} \). As expected, if \( \mathcal{U} \) contains the pre-state of the environment, then \( \text{pre}(r(u)) = \text{post}(u) \).

The repair procedure may return a set of alternative repair updates. Moreover, it may access the checking procedure, and retrieve the inconsistency levels \( \mathcal{I} \) of the model instances. This allows the repair procedure, for instance, to access the set of detected violations, if the \( \text{CHECK} \) procedure supports such reports.

**Definition 2 (model repair).** A model repair procedure \( \text{REPAIR} : \mathcal{P} \rightarrow \mathcal{U} \rightarrow \mathcal{P} \mathcal{R} \) calculates repair updates \( r \in \mathcal{R} \) for a user update \( u \in \mathcal{U} \) under constraints \( c \subseteq \mathcal{C} \), which is denoted by \( r \in \text{REPAIR}_c u \).

The behavior of the \( \text{REPAIR} \) procedure is fundamental to define the overall characteristics of the repair framework, while the shape of the produced repair updates \( \mathcal{R} \) affects its flexibility and effectiveness.

The generated repair updates do not necessarily recover full consistency, although they are expected to ameliorate the inconsistency level of the environment. The relation between the checking and repair procedures, as well as the properties and enumeration of the generated repair updates, are dependent on the concrete model repair approach, and are key feature to define the functionalities of the framework.

Fig. 4 presents an overview of our generic scheme for model repair. User updates \( u \) are applied to an existing model instance \( m_0 \), consisting of a tuple of model instances, from which the modified model instance \( m \) is obtained, and to which the checking procedure assigns an inconsistency level \( i \). Given a user update \( u \), and with access to the checking procedure, the repair procedure generates a set of possible repair updates \( r \), which, when applied to \( u \), result in an update \( u' \) from which the repaired model instances \( m' \) can be obtained, and whose inconsistency level \( i' \) is expected to be at least the same as the one of \( u \). (The pre operations are grayed out because the updates may not store that information.)

### 3 Feature-based Taxonomy

This section presents the identified classification features for model repair approaches, that instantiate the abstract
3.1 Methodology

Our research methodology is inspired by guidelines proposed for systematic literature reviews in software engineering, which aim to identify, evaluate, and interpret all available research relevant to a particular topic area, or phenomenon of interest [21]. One of the objectives of such review is to provide a classification framework that allows researchers to appropriately position new research activities [21], which is the goal of our study. Nonetheless, since we do not exactly aim to establish the state of evidence of the area, but rather to identify the features of existing approaches, our methodology shares characteristics with systematic mapping studies [33] as well.

Systematic reviews rely on a predefined review protocol for the selection of the primary studies (the review itself being a secondary study), that should ensure rigor and completeness of the process, as well as enable repeatability. Our protocol, depicted in Fig. 6, was inspired by previous systematic reviews on other topics of software engineering [34], [35] and is detailed in the Appendix.

Briefly, we started the process by defining the research questions that guide this study. Then we defined the search strategy employed to select the primary studies, backed by pivot searches that helped identify relevant search keywords and venues. We then specified the selection criteria used to obtain the definitive list of primary studies considered in our study and defined how the relevant data would be extracted from these studies, also backed up by pilot data extractions. Finally, we defined the procedure through which this data would be effectively synthesized into the structured taxonomy and presented in the shape of feature models, the main contribution of this work. The last two steps followed guidelines for thematic synthesis in software engineering [36]. The classification of the primary studies under the resulting features is publicly available [27].

The formalization of the model repair problem in Section 2.2 identified several artifacts whose features characterize each particular approach. The research questions aim precisely to explore alternative instantiations to these artifacts in the existing literature.

RQ1 What are the domain spaces on which approaches act, and how is the user able to customize them?
RQ2 What kind of constraints are supported by the approaches, and how are they specified?
RQ3 What kind of information regarding the repair updates is expected from the approaches?
RQ4 What is the role of the checking procedure in the overall process, and what kind of information is reported?
RQ5 What is the overall behavior of the repair procedure, and what is the shape of the generated repair updates?
RQ5.1 How can the user affect the behavior of the approaches and how are the alternative repair updates reported?
RQ5.2 What kind of semantic properties are guaranteed by the approaches?

Concretely, RQ1 refers to the specification of domain space M, RQ2 to the universe of constraints C and RQ3 to the universe of user updates U. RQ4 addresses how the checking procedure CHECK relates with the repair procedure and the shape of the reports R. RQ5 refers to behavior of the repair procedure REPAIR and the universe of repair updates R. Due to the importance of this procedure, we detail two further questions, regarding the interaction of the user with the repair procedure (RQ5.1) and the semantic properties guaranteed by the procedures (RQ5.2).

The resulting taxonomy for model repair approaches is organized under these 5 main branches, arising from the research questions and addressing different artifacts. We opted to present the resulting taxonomy as feature models. These are typically represented diagrammatically, following the notation from Table 1. A child feature may only be selected by an approach if its parent is also selected. Children features may either be mandatory (if the parent feature is selected, so must be the child), optional (if the parent feature is selected, the child may or not be selected) or arranged in or groups (if the parent is selected, at least one feature of the group must be selected) or xor groups (if the parent feature is selected, exactly one feature of the group must be selected). Every feature model has a root feature that is always present in every configuration, and may contain reference features which simply point to other feature models. Finally, feature models may also be annotated with requires and excludes constraints that allow the enforcement of cross-tree dependencies.

The top-level feature model is depicted in Fig. 7 with Repair Technique as its root, and a mandatory child feature for every main classification facet, referencing a separate and detailed feature model, which are explored in the succeeding sections. This division is purely for aesthetic purposes, and the various trees could be composed into a single one by connecting the reference features with the roots of the matching diagrams.
3.2 Domain
These features address the model domain space $M$ of the technique, i.e., which model instances the technique is able to handle, as well as whether the user is able to customize such space (RO1). The alternatives are explored below and depicted in the diagram from Fig. 8 referenced from the main diagram from Fig. 7.

3.2.1 Formalism
Apart from early human-centered approaches, that do not propose automated systems to manage consistency and informally defined artifacts, procedures CHECK and REPAIR are designed to handle model instances $m$ from $M$ represented using particular formalisms. The detected formalisms include logical representations in some abstract formal specification language \[7, 9, 11, 37, 38, 39, 40, 41, 42, 43, \text{tree-like data structures} \[41, 45, 46, 47, 48, \text{object-oriented specifications} \[10, 13, 14, 15, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, \text{support for relational data structures} \[20, 28, 60, 61, \text{or graphs} \[12, 42, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, \text{formal specification language} \[7, 9, 10, 14, 20, 28, 37, 38, 39, 41, 44, 48, 51, 54, 55, 56, 57, 58, 59, 60, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, \text{i.e., the domain space} \[M, \] these features are organized in a xor-group since our study showed that the selection of the formalism is exclusive.
The chosen formalism is tightly connected with the kind of properties that the technique is able to check. For instance, reachability properties are more easily handled in relational or graph data structures. However, the reason behind the choice of formalism tends to be ability to use previously developed techniques in the model repair approach. This is patent in the fact that most techniques based on graph formalism are built over Triple Graph Grammars (TGG) \[80, \] techniques, or that those based on logical formalisms rely on well-defined search procedures to deploy the repair procedure.

Note that, although related, this feature is not directly restricted by the technical space on which model instances are designed (Section 3.2.3), which can be internally converted to the underlying formalism by the modeling framework. For instance, techniques acting upon the MDA technical space embed UML models into relational or graph structures. Nonetheless, formalisms not closely related to the technical space may be loose relevant information regarding the application domain, which may be preserved by those over an object-oriented formalism, for instance.

3.2.2 Meta-model Independent
Model repair approaches may aim to be independent of the application domain. Such meta-model independent techniques may optionally provide the users with mechanisms to define the well-formedness rules of the model instances \[7, 9, 10, 14, 20, 28, 37, 38, 39, 41, 44, 48, 51, 54, 55, 56, 57, 58, 59, 60, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, \text{i.e., the domain space} \[M, \] this task may be delegated to different agents of the MDE process. For instance, in the ViewPoints framework \[81 \] there are two well-defined roles: the designer of the viewpoint, that defines the meta-model, the constraints and the repair plans, and the owner of the viewpoint, that manages the view according to the designer’s rules.

Meta-model independent techniques are more customizable and have wider applicability than those whose meta-model is fixed. Techniques with fixed meta-models are designed to act on specific domains, like those proposed to manage the consistency of specific UML diagrams. While with more limited applicability, knowing the shape of the model instances a priori may allow the technique to have improved effectiveness and efficiency. Moreover, meta-model independent techniques are necessarily more laborious to the user, as not only must the meta-model be defined, but also any constraint that is to be checked over the models, since there cannot be hard-coded constraints for undefined meta-models (Section 3.2.1).

3.2.3 Technical Space
This feature defines the technical space in which the user is expected to specify the various artifacts of the MDE development environment. These may be built around standard languages/architectures like XML \[45, 47, \] MDA \[12, 15, 40, 43, 52, 53, 55, 58, 61, 69, 73, 74, 79, \text{or EMF} \[12, 15, 40, 43, 52, 53, 55, 58, 61, 69, 73, 74, 79, \text{or other specific to the proposed technique} \[7, 9, 10, 11, 13, 20, 58, 59, 44, 46, 48, 49, 50, 51, 56, 59, 60, 62, 64, 66, 67, 68, 70, 71, 72, 73, 74, 76, 77, 78, 82, \text{the analyzed studies show that this is an exclusive group of features.}

The selection technical space defines the concrete model syntax that the technique is able to process, like XML, XML, UML, or a technique-specific language. These concrete model instances are translated by the technique into their representation in the underlying formalism (Section 3.2.1).

For meta-model independent techniques, this feature also specifies the meta-modeling language through which the user should specify the meta-models. Under MDA, these are expected to follow the MOF \[83, \] standard, and those under EMF, Ecore. Again, techniques may not support standard

\[1 \text{http://eclipse.org/modeling/emf/} \]
meta-modeling languages, and require the user to define them through technique-specific mechanisms. If the user is allowed to define or customize constraints (Section 3.3.1), this feature defines the language in which he is able to do so. Typically this amounts to some version of MDA’s OCL, that is also used in EMF, or it can be defined specifically for the technique. In techniques with support for inter-model constraints (Section 3.3.2), standard languages include MDA’s QVT [31].

The use of standardized technical spaces is essential if the model repair technique is to be integrated into the regular MDE development process. Techniques using specific languages are usually prototype tools that rely on a manual translation of the model instances.

### 3.2.4 Bounded

Techniques may assume a bounded universe of model elements, so that the repair procedure can be more manageable [13], [20], [28], [39], [42], [60], [61]. Such is the case of techniques that do not allow the creation of new elements, and thus are inherently bounded by the elements present in the current inconsistent state.

Some techniques impose a bounded universe in order to avoid handling possible negative side effects that may arise when new elements are created. As an alternative, many repair techniques opt to create instead abstract elements, that are to be instantiated by the user a posteriori (Section 3.6.2). In others, the bounded universe is imposed by the underlying procedure, like those relying on bounded solvers. This process can however be opaque to the user, by iteratively introducing new model elements in the universe as the process executes.

### 3.2.5 Multi-model

Model repair techniques may optionally be designed with particular concerns regarding inter-model consistency and provide dedicated support for multi-model scenarios [7], [11], [14], [28], [37], [38], [43], [44], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56], [57], [59], [60], [61], [62], [63], [64], [65], [66], [67], [68], [69], [70], [71], [72], [75], [74], [76], [77], [78], [82]. In such cases, each state \( m \in M \) is comprised by a tuple of model instances \((m_1, \ldots, m_k)\). Such is the case of techniques that were developed to manage consistency in development environments with multiple views, for model synchronization and bidirectional and multidirectional transformations.

2. Note that the domain being classified is effectively the search space available to the repair procedure.

In contrast, techniques may be defined to manage the internal consistency of a single model, in which case a state \( m \) consists of a single model instance \( m \).

Multi-model approaches focus on handling inter-model constraints (Section 3.3.2), which usually take different shapes than those used for intra-model consistency (Section 3.3.3). As a consequence, such approaches often disregard the internal consistency of the individual model instances, possibly leading to overall inconsistent states. Multi-model techniques may impose restrictions on the supported user updates (controlled update, below). Moreover, the model instances affected by the generated repair updates may also be restricted or customizable by the user (Section 3.7.2). Bidirectional transformations are a typical example of such techniques, where user updates are restricted to a source model instance, and the generated repair updates restricted to a target model instance.

Techniques without dedicated multi-model support may still handle coexisting models by merging the various model instances (and associated meta-models) into a “dummy” model conforming to a single meta-model, and expressing their seemingly inter-model constraints as that meta-model’s intra-model constraints (Section 3.3.2). Specifying inter-model consistency as an internal constraint may however prove to be more cumbersome. This is common in techniques that manage the consistency between different UML diagrams, since they share the same meta-model, as in the example from Section 2.1. In our taxonomy, such domain spaces are not considered multi-model (nor their constraints inter-model). Techniques without native support for multi-model domain spaces may simulate the controlled repair updates provided by multi-model techniques through distinguished constraints (Section 3.3.1) – by temporarily introducing a constraint that restricts the state of one of the model instances – or by assigning higher weights to certain model instances (Section 3.7.2) – promoting updates over the other model instances – if these features are supported.

Since multi-model techniques are quite common, we identified two additional optional features that such techniques may employ.

**Controlled Update:** Approaches with support for multiple models may optionally force the user to update the model instances in a controlled manner, typically only allowing updates over a single model instance so that the update propagation to the others is more easily managed [38], [59], [43], [57], [60], [63], [65], [66], [72], [73], [76], [77], [78]. This is common in bidirectional transformation or incremental transformation techniques, where the repair
updates are themselves focused on a single model: allowing concurrent updates could lead to conflicts that could not be resolved. Such techniques are less suitable for distributed and heterogeneous MDE development environments, since the different stakeholders are expected to update the various model instances concurrently.

Pairwise: Multi-model techniques may optionally focus on pairwise consistency management, since managing the consistency between only two model instances is more manageable. However, since the selection is not exclusive, techniques like these may not have hard-coded constraints. Meta-model independent techniques, however, allow the user to extend them or restrict them. Alternatively, some techniques allow the user to focus on intra-model constraints and instruct the repair procedure to focus them in detriment of the remainder constraints of the environment. Nonetheless, it is also the most direct mechanism through which the user may control the behavior of the repair procedure, one that is tightly coupled with the definition of the constraint.

Repair Hints: The model repair procedure may optionally expect each constraint to be accompanied with repair hints on how to generate the repair updates when violations to that constraint are detected. This contrasts with techniques where the repair procedures automatically derive the repair updates from the constraints. The extreme case occurs in rule-based approaches where the repair procedure expects effective resolution rules for the violations.

The definition of repair hints is often a laborious and error-prone activity that does not provide totality or correctness guarantees, since the user may not be aware of the possible side effects of the defined hints. Nonetheless, it is also the most direct mechanism through which the user may control the behavior of the repair procedure, one that is tightly coupled with the definition of the constraint. Repair hints do not necessarily reduce the enumeration of repair updates to a single alternative. Nonetheless, the technique may allow multiple repair hints to be defined for each constraint.

Distinguished: Techniques may optionally support the definition of distinguished constraints that instruct the repair procedure to focus them in detriment of the remainder constraints of the environment. These features are somehow related, and constraint prioritization could, to a degree, simulate the behavior of distinguished constraints. However, even with different priorities the repair procedure could still consider every constraint and their interaction. In contrast, distinguished constraints are effectively treated differently, and the procedure may, for instance, focus on certain constraints while discarding the others, or consider them for side effects only. This is particularly patent in the violation selection feature presented below. Incremental techniques, which rely on information from the previous executions, may not be able to support this kind of constraints.

The most common occurrence of distinguished constraints arises in techniques that allow the user to select a specific violation to be handled. This contrasts with techniques where the checking procedure reports at least the set of detected violations, as made explicit in the diagram. In such cases, the composite report typically assesses whether the selected violation was effectively removed, and the impact of that repair update on the other constraints of the environment. Since typical

3.3 Constraint

These features address the expressiveness imposed by the constraint universe $C$ and how the constraints $c$ are drawn from $C$ in the modeling framework (RQ2), the former entailing the class of problems that may be addressed by the technique and the latter its general applicability. These design choices are explored below and depicted in Fig. 9, which is referenced from the main diagram from Fig. 7. For techniques with decoupled checking procedures (Section 3.5.1), these features are assumed to regard those of the associated checker, if identified by the authors.

3.3.1 Specification

Similar to the meta-model (Section 3.2.2), techniques may either have the set of constraints $c$ hard-coded or provide the user with mechanisms to define or customize them. Techniques may even provide a set of predefined constraints but allow the user to extend them or restrict them. As a consequence, we identify these two features as an orthogonal, since their selection is not exclusive. Likewise the meta-model, many modeling frameworks delegate such tasks to a repair administrator, rendering the process opaque to the software designer.

Techniques that do not allow the user to define the constraints have limited applicability since they cannot be easily adapted to different application domains. There are typically paired with fixed meta-model techniques, where both the meta-model and the constraints are fixed a priori (techniques for managing consistency of UML diagrams being the classical example). Nonetheless, techniques with fixed meta-model may still allow the user to define the constraints. Meta-model independent techniques, however, may not have hard-coded constraints (as imposed by the excludes clause in the diagram).

Our study also identified two additional optional features that may be enforced by the model repair techniques in order to ease the generation of repair updates, as presented below.
constraint languages like OCL do not allow the specification of constraints at the model level, violation selection is performed through mechanisms internal to the technique.

This kind of approaches may be more scalable than those attempting to handle all inconsistencies at once by following a spirit of toleration. Rule-based approaches typically handle a single violation at a time, since the resolution rules are usually defined per constraint. They also provide a direct mechanism through which the user may affect the behavior of the repair procedure. However, they may also be oblivious of possible negative side effects, which may undermine the correctness of the procedure.

3.3.2 Kind

General-purpose model repair techniques act on *intra-model* constraints, interpreting the environment as a single model restricted by internal constraints [9], [10], [12], [13], [15], [20], [28], [37], [59], [40], [41], [42], [43], [44], [45], [46], [53], [58], [61], [62], [70], [79]. Nonetheless, techniques that support multi-model domain spaces (Section 3.2.5) typically support the definition of *inter-model* constraints that define the relationship between two or more models [7], [11], [14], [28], [37], [38], [43], [44], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56], [57], [59], [60], [63], [64], [65], [66], [67], [68], [69], [71], [72], [73], [74], [75], [76], [77], [78], [82]. While some of these focus on inter-model consistency and disregard the intra-model constraints, some approaches do consider both kinds of constraints. For that reason this feature is presented as an or-group.

The shape of the constraints (Section 3.3.3) is related but not exactly defined by this feature. In fact, both logical constraints and pattern matching can be used to define both intra- and inter-model constraints. Other shapes (traceability links, consistency relations and transformations) are however restricted to inter-model constraints. This dependency is made explicit in the diagram. For approaches supporting both kinds of constraints, their shape may not be identical.

The impact of supporting inter-model constraints is similar to the one of supporting multi-model domain spaces natively (Section 3.2.5). Although the definition of this kind of constraints is simplified, techniques with support for them will often disregard intra-consistency constraints, undermining the overall consistency of the environment. Inter-model constraints can usually be simulated through inter-model constrains, assuming a “dummy” meta-model composed of the individual meta-models. In this way the techniques would handle both intra- and inter-model constraints, but the definition of the latter would be more laborious to the user.

3.3.3 Shape

This feature determines the shape of the constraints supported by the model repair approach. The feature is encoded as an or-group since approaches may support more than one shape of constraints, particularly when they support both intra- and inter-model constraints (Section 3.3.2). For hard-coded constraints (Section 3.3.1) it may not be possible to determine the shape of the constraints from the primary studies alone. Note that approaches implementing the same features may still be able to address different classes of problems, since they may support constraints of varied expressiveness.

Constraints are most commonly defined as logical predicates [7], [9], [10], [11], [12], [13], [14], [15], [20], [28], [37], [41], [42], [43], [44], [45], [46], [52], [58], [61], [68], [75]. The expressiveness of such constraints is typically that of first-order logic, although they may be extended with other operators like transitive closure to allow the specification of reachability properties. These may also be used to define inter-model consistency, assuming they are able to refer to elements from different models.

Approaches built over graph data structures are often based on pattern matching, most of the times enhanced with negative application conditions (NACs) [12], [53], [54], [62], [63], [64], [65], [66], [67], [68], [69], [70], [71], [72], [74], [75], [76], [77], [79]. Pattern matching is well-suited to specify structural properties but not behavioral ones. Thus, to improve its expressiveness, some approaches allow the patterns to be attached with additional attribute constraints or imperative code snippets.

Techniques with dedicated support for inter-model consistency may rely on the definition of traceability links that connect elements from different model instances [7], [37], [38], [44], [49], [51], [54], [55], [56], [63], [64], [65], [66], [68], [69], [71], [72], [73], [74], [75], [76], [77], [82]. Constraints or patterns may then be defined over the traceability links that denote the notion of inter-model consistency (like TGGs), although some techniques assume fixed constraints over these links. The traceability links may either be explicitly de-

---

0098-5589 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
fined by the user – by manually indicating which elements correspond to each other – or be implicitly introduced either by the repair rules or by calculation. The expressiveness of such techniques depends on the ability to define properties over traceability links, like constrains restricting their multiplicity.

Inter-model consistency without traceability links is commonly defined through consistency relations, declarative predicates that define which sets of model instances are considered to be consistent with each other [28], [47], [49], [51], [55], [59], [60], [73]. Finally, some frameworks assume a notion of consistency that is implicitly defined by a transformation [38], [48], [57], [59], [78]. This is typical in multi-view frameworks with a reference model, from which each view is calculated through transformation. In such cases the model repair procedure addresses the view-update problem [63], and usually relies on the bidirectionalization of the transformation language.

The concrete syntax of the constraints is heavily dependent on the chosen technical space (Section 3.2.3). Logical constraints are commonly defined using some variant of OCL [30] standardized in MDA. Since using OCL to define inter-model constraints may be cumbersome, extensions that natively support multi-model domain spaces are also used, like Epsilon [7] from the EMF. The QVT Relations [31] from MDA is a standardized language for the definition of consistency relations between multiple model instances. Techniques relying on transformations to define the notion of consistency may also support standard transformation languages, like ATL [6] from EMF. Often however, techniques rely on internal formalisms to define the constraints.

In approaches requiring repair hints (Section 3.3.1) or rule-based approaches (Section 3.6.1), the constraints may need to be appended with additional information. In fact, in some rule-based approaches the notion of constraint is itself embedded in the definition of the repair rule (as a precondition for its application). In such cases it may not even be possible to check the consistency of constraints prior to deploying the repair procedure (Section 3.5.2).

### 3.4 Update

These feature address the universe of the user updates $U$ (RQ3), which essentially defines what information is available to the model repair procedures regarding the evolution of the models from the previous known state to the current one. These are summarized in Fig. 10 which is referenced from the diagram from Fig. 7.

#### 3.4.1 Update Representation

The simplest approach to the user update facet is to be purely state-based, where the repair procedure simply considers the post-state of the user update (i.e., the current state of the model instances), in which case user updates from $U$ simply amount to model instances $m$. [7], [11], [12], [14], [15], [20], [28], [37], [39], [40], [41], [43], [44], [45], [46], [47], [48], [51], [52], [53], [56], [57], [60], [61], [62], [64], [67], [68], [69], [70], [74], [75].

---


---

![Fig. 10. Update features.](image-url)
Visions

Goal

Reporting

Decoupled

Boolean

Violations

Number

User

Simple

Composite

Check

Checkonly

Decoupled = Distinguished

Decoupled = Checkonly

Composite = Distinguished

Decoupled = Checkonly

Fig. 11. Check features.

3.5 Check

Check features regard the model repair technique’s reliance on the checking procedure CHECK and the information contained in the inconsistency reports I (RQ4). These design options are depicted in the diagram from Fig. 11 referenced from the one in Fig. 7. Since consistency checking is not the focus of this study, these features focus mainly on classifying relationship between the checking and repair procedures.

3.5.1 Decoupled

Model repair techniques may optionally be decoupled from the consistency checking procedure [7], [9], [12], [15], [41], [44], [52], [53], [56], [61]. Such techniques may rely on external tools to detect violations to the constraints. Coupled procedures in contrast use the checking procedure as a fundamental piece in the repair procedure – sometimes in ways opaque to the user. Earlier techniques rely on the manual identification of the violations by the users of the techniques, which we interpret as a special kind of decoupled checking procedures [44], [56].

Although this feature allows the repair procedure to be extensible by deploying state-of-the-art checking procedures, coupled checking procedures typically result in more efficient techniques, since the repair procedure can exploit the potential of the checking procedure.

Decoupled checking procedures usually report structured information, like goals or violations (Section 3.5.3), that can then be processed by the repair procedure. A typical example occurs in some rule-based approaches (Section 3.6.1) that employ two classes of rules: check rules, that detect the violations and introduce some token identifying the violation, and repair rules, that detect such tokens and act upon the violation. Another instance of a decoupled procedure occurs in search-based approaches (Section 3.6.1), where the checking procedure detects a set of elements suspected to cause the inconsistency, which the repair procedure tries to remove from the model instances.

In decoupled approaches the checking procedure must somehow pass the detected information to the model repair procedure. In our scheme, this is performed through distinguished constraints (Section 3.3.1), as made explicit in the diagram. Although this feature is related to the ability to perform checkonly executions (Section 3.5.2 below), we shall see that there is not an explicit dependency between the two.

3.5.2 Checkonly

Although not directly related to the problem of model repair, it is important for the modeling framework to provide the user with information regarding the inconsistency level of the environment prior to the deployment of model repair procedures. Thus, standards like QVT enforce both repair and checkonly modes [7], [9], [10], [11], [12], [13], [14], [15], [28], [39], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [52], [53], [55], [56], [58], [59], [60], [62], [64], [66], [69], [70], [71], [72], [75], [76], [82]. In techniques that allow the selection of the violation to be handled (Section 3.3.1) such functionality is fundamental to allow the user to inspect the detected violations.

This feature is optional as some approaches do not have a proper checkonly mode. For instance, in some rule-based approaches (Section 3.6.1), the constraint may be simply defined as the pre-condition of the resolution rule.
The notion of coupled checking procedure (Section 3.5.1) is distinct from this feature, although the two are related. The best way to envision their relationship is through the 3 classes of rule-based approaches (Section 3.6.3) detected during our study. In the simplest class techniques employ repair rules only – the checking procedure is coupled to these rules as a pre-condition and cannot be run in checkonly mode. In the second class, techniques employ both check and repair rules, but these act independently of each other – thus the checking procedure is still coupled to the repair rules as a pre-condition, but the approach supports checkonly mode. In the last class techniques employ both check and repair rules, but the latter only act on tokens introduced by the former when violations are detected – thus they are decoupled and also support checkonly mode. It seems however improbable that decoupled approaches do not provide a checkonly mode, thus we enforce that dependency in the diagram.

### 3.5.3 Reporting

This group feature classifies the information reported by the checking procedure about the detected inconsistencies, i.e., the universe of inconsistency reports $I$. Techniques may just expect a basic Boolean procedure (20), (28), (37), (48), (55), (58), (59), (60), (28) that simply reports whether inconsistencies were found. This is typical for solver-based approaches (Section 3.6.3). Techniques may instead expect to know the number of violations occurring in the current state (70). Most commonly, the checking procedure returns a set of violations detected in the model instances (7), (10), (11), (12), (13), (14), (15), (39), (40), (41), (43), (44), (45), (46), (47), (49), (50), (51), (52), (63), (64), (65), (66), (67), (68), (69), (70), (71), (72), (73), (74), (75), (76), (77), (79), (82). The information contained in each violations varies, commonly containing information regarding which constraint is being broken and the model elements involved. Techniques may also report a goal that must be achieved by the repair procedure (9), (42). These may be comprised by a formula that is suspected to have rendered a constraint false – which the repair procedure must make true – or simply contain information regarding elements suspect of causing the inconsistency – that must be removed – or missing model elements – that must be created.

For some techniques with coupled checking procedures (Section 3.5.1) it may not be clear what the checking procedure reports. Such is the case of approaches where user updates are simply mapped into repair updates.

Having information about individual violations allows the user to selectively apply repair updates (Section 3.3.1), unlike with less expressive reports. This is an explicit dependency between the features.

Since the behavior of the partial order $\sqsubseteq$ over inconsistency levels $I$ is dependent on the information contained in these reports, this feature is tightly connected with the correctness criteria that the repair technique may be expected to follow (Section 3.8.2). In most cases, there is a single sensible partial order. In boolean reports, this is simply defined as

$$i \sqsubseteq i' \equiv i \iff i'$$

just enforcing that a consistent state does not regress into an inconsistent one, with the least element $\bot_I = \text{True}$. With numerical reports, the partial order takes the shape

$$i \sqsubseteq i' \equiv i \leq i'$$

where $\leq$ is the standard order over naturals, stating that the number of inconsistencies at least does not increase, with $\bot_I = 0$. For the list of violations, it simply takes the shape

$$i \sqsubseteq i' \equiv i \subseteq i'$$

meaning that no new violations are introduced, with $\bot_I = \emptyset$, the empty set of violations. Goal reports may vary in shape and content, thus the partial order will vary from approach to approach.

Generally, model repair techniques expect a single kind of inconsistency reports from the checking procedure. However, this group feature is encoded as an or-group due to the possibility of composite reports with heterogeneous information, as explained below.

**Composite**: The checking procedure may optionally report a composite inconsistency level (7), (9), (10), (11), (12), (13), (14), (28), (39), (43), (44), (45), (46), (47), (49), (50), (51), (52), (53), (54), (55), (58), (62), (65), (67), (69), (70), (79), (82). These emerge from distinguished constraints (Section 3.3.1), which are independently checked by the procedure. Typical this occurs when the approach supports violation selection, where inconsistency levels $I$ take the shape $I_1 \times I_2$, a pair whose first element states whether the selected violation was removed, and the second element provides information regarding the remainder environment constraints, allowing the user to be aware of possible side effects. Approaches with distinguished classes of constraints (like intra- and inter-model constraints) also result in composite reports.

In these composite reports there is more than a single sensible partial order over each shape of $I$. Our study identified three kinds of expected behavior in these cases. If both components are deemed equally important, the partial order takes the shape of the product order:

$$(i_1, i_2) \sqsubseteq (i'_1, i'_2) \equiv i_1 \sqsubseteq i'_1 \land i_2 \sqsubseteq i'_2$$

meaning that the inconsistency level is improved if either of the components is. The least element of this partial ordered set is simply $(\bot_I, \bot_I)$. Under violation selection this partial order is not very useful since it would allow the removal of the selected violation or any of the others. A partial order that prioritizes the amelioration of the first component is the lexicographic order, under which improvements to the first component allow arbitrary updates on the second one:

$$(i_1, i_2) \sqsubseteq (i'_1, i'_2) \equiv (i_1 = i'_1 \land i_2 \subseteq i'_2)$$

In such case, the least element is still $(\bot_I, \bot_I)$. Under violation selection this order allows the remainder violations to deteriorate, allowing negative side effects, when removing the selected one. Alternatively, techniques may prioritize the improvement of the first component but disallow damage to the second one. Under violation selection, such partial orders represent techniques that forbid negative side effects: the selected violation should be removed but avoiding the introduction of new ones in the process.

The information reported for the distinguished constraint and the remainder constraints needs not be equal.
For instance, some approaches are only concerned with not increasing the number of violations caused by the remainder constraints (even if they are not exactly the same occurring in the initial state). This is the reason why the reporting feature above is set as an or-group, and not as an exclusive selection.

3.6 Repair

These features, depicted in Fig. 7, which is referenced from Fig. 7, classify the overall behavior of the model repair procedure REPAIR, as well as the universe of repair updates $R$ (RQ5), which are at the core of the model repair approach. Due to their relevancy, the enumeration of the repair updates to the user (RQ5.1) and the functional semantics guaranteed by the approach (RQ5.2) are explored separately in Sections 3.7 and 3.8 respectively.

3.6.1 Core

This feature classifies the engine underlying the repair generation procedure. Rule-based techniques rely on a set of previously defined rules that are applied whenever an inconsistency is detected [7, 11, 12, 13, 14, 41, 42, 43, 47, 48, 50, 52, 53, 54, 56, 57, 62, 63, 64, 65, 66, 67, 68, 69, 71, 72, 73, 74, 75, 76, 77, 79, 82]. While providing full control over the resolution of inconsistencies, it puts the weight on the designer that must specify how constraints are fixed. Moreover, having a fixed set of resolution rules greatly reduces the flexibility of the technique. Generative approaches derive their transformation rules from production rules that define what is a well-formed model [63, 64, 66, 68, 71, 72, 74, 75, 76, 77, 79]. The classical example of such approaches are those based on graph grammars, where the repair rules are derived from the grammar productions.

In contrast, syntactic techniques automatically derive repair plans by syntactic analysis of the constraints [10, 15, 45, 47, 48, 51, 55, 57, 58, 78]. Typically, these repair plans are calculated at static-time and then instantiated to concrete model instances at run-time when an inconsistency is found. While these techniques may be able to generate repair updates without user input, the number of generated plans may become overwhelming for the user to choose from. Syntactic techniques are also not well suited to deal with multiple inconsistencies, nor inconsistencies that affect a large portion of the model.

Search-based approaches interpret model repair as a model search problem [9, 20, 28, 37, 38, 39, 40, 41, 42, 60, 61, 70]. These are able to automatically find fully consistent models (Section 3.8.2), but suffer from scalability issues. Moreover, they are well-suited to fix inconsistencies that affect a large portion of the model, like reachability properties. Some approaches rely on off-the-shelf solvers to search for consistent states [20, 28, 37, 41, 60, 61]. These solvers are oblivious of the application domain, and may produce unpredictable solutions. In contrast, other techniques rely on domain-specific search procedures that rely on domain-specific knowledge, like heuristics and the available edit operations, that allow a finer control on the generation of repair updates [9, 38, 39, 40, 42, 70].

Some hybrid techniques are built over more than one of these features. Such is the case of rule-based approaches that rely on search-based techniques to calculate repair plans from those rules. Thus, the selection of features from this group is not exclusive. Some earlier approaches are human-centric, relying on the user to manually flag inconsistencies and propose repair updates, focusing on the negotiation and education between different stakeholders [11, 44, 46, 49, 50, 56, 67, 82]. As expected, such approaches provide little semantic guarantees (Section 3.8).

The selection of this feature directly or indirectly affects most of the remainder features of the model repair approach. That impact is explored in the presentation of the features throughout Section 3.

Incremental: Approaches may optionally be incremental and reuse data from previous checking or repair executions, improving efficiency and localization of inconsistencies [10, 13, 37, 42, 49, 54, 55, 63, 64, 65, 66, 68, 69, 71, 72, 73, 74, 75, 76, 77]. Such techniques are typically deployed in an online setting so that the required information is preserved between executions. Thus, they are also typically delta-based (Section 3.4.2) so that this information is more easily managed.

Incrementality can be essential to preserve the consistency of the environment – as in the case of approaches that rely on implicit inter-model traceability links calculated in previous executions – or simply a mechanism to improve efficiency – by storing the instantiations of the constraints so that inconsistencies can be more efficiently checked and repaired. Frameworks that record the whole evolution history of the model instances (Section 3.4.2) may also be seen as incremental since this history may be used to guide the generation of repair updates.

3.6.2 Repair Representation

This feature regards the actual information contained in the repair updates $R$ returned by the repair procedure. We identified two exclusive features in this group. Those with state-based repair updates simply return the newly generated model instances [14, 20, 28, 37, 41, 43, 48, 49, 51, 53, 54, 57, 59, 60, 61, 62, 63, 64, 69, 73, 78, 79]. In such cases, a repair update $r \in R$ simply amounts to new model instances $m \in M$. Other procedures are operation-based, returning instead information regarding how the model instances should be changed in order to ameliorate the consistency level [7, 9, 10, 11, 12, 13, 15, 38, 39, 40, 42, 44, 45, 46, 47, 50, 52, 55, 56, 58, 63, 65, 66, 68, 70, 71, 72, 73, 74, 75, 76, 77, 79]. The shape of repair update $r \in R$ in such cases varies, as presented below.

Note that the information contained of repair updates $r \in R$ is not necessarily the same as the one of user updates $u \in U$ (Section 3.4). For instance, it is common for model repair approaches to consider state-based user updates but generate operation-based repair updates.

Operation-based: In operation-based approaches, a repair update proposed to the user may take the shape of a repair action, consisting of an atomic edit operation [7, 11, 12, 13, 44, 46, 50, 52, 56, 58, 75, 82], or of a repair plan, built from the sequential composition of valid edit operations [9, 10, 13, 38, 39, 40, 42, 43, 45, 47, 55, 63, 65, 66, 68, 70, 71, 72, 73, 76, 77]. The set
of valid edit operations that comprise these repair updates is defined elsewhere (Section 3.6.3).

This notion is different from that of multiple repair update alternatives (Section 3.7.1): in a repair plan the multiple actions aim to solve the same inconsistency, while the multiple enumeration of repair updates may represent alternative solutions to the same inconsistency (which may themselves be repair plans).

Content: The repair updates are also classified by their content. In this context, they may either be concrete, in which case they can be directly applied to the environment [9], [11], [12], [13], [14], [15], [20], [28], [37], [39], [41], [43], [45], [47], [48], [50], [51], [52], [53], [54], [55], [57], [58], [59], [60], [61], [62], [63], [64], [65], [66], [68], [69], [70], [71], [72], [73], [74], [75], [76], [77], [78], [79], [82], or abstract, requiring input from the user to be instantiated [7], [9], [10], [11], [15], [42], [44], [45], [46], [49], [50], [56], [62], [69], [82]. Most model repair procedure generate concrete repair updates when possible, and only occasionally abstract ones. Thus, the selection of these features is not exclusive.

Abstract repair updates may occur when the update requires a parameter that the model repair procedure is not able (or was not designed) to provide, relying instead on the user to define it. A typical example occurs when the technique identifies that the value of a property must be changed, but does not commit to a concrete new value. This kind of procedure may undermine the correctness of the procedure (Section 3.8.2) since the user may fail to handle the inconsistency or introduce new ones (which may be common as the complexity of the modeling environment increases).

Our study show that both abstract and concrete repair updates may be used in both state-based and operations-based repair updates.

3.6.3 Edit Operations

This feature regards the set of edit operations available to the repair procedure to calculate the repair update alternatives. For state-based repair updates (Section 3.6.2), and typically in those solver-based (Section 3.6.1), this set may be undefined, since the repair procedure simply searches for consistent model instances [37], [44], [59], [61], [78]. In rule-based approaches, this set amounts to the repair rules defined in the framework. In contrast, in syntactic and other search-based approaches, this amounts to the set of operations available to the procedure when traversing the constraints or searching for solutions. These usually amount simply to creation, modification and deletion operations, although our study shows that some do not allow the creation of elements.

Although in many techniques this set of valid edit operations is fixed [9], [10], [11], [12], [13], [15], [20], [38], [39], [40], [42], [43], [46], [47], [48], [49], [50], [51], [52], [55], [56], [58], [60], [62], [63], [64], [66], [67], [68], [69], [71], [72], [74], [75], [76], [77], [78], [79], [82], the user may also be allowed to customize it, either by being able to define the set of valid edit operations or by disabling some of those predefined [7], [14], [28], [41], [45], [53], [54], [57], [65], [70], [73], [79]. This is typical in rule-based approaches and some syntactic approaches that generate the repair updates for each constraint at static-time.

Techniques with a well-defined set of edit operations may also allow the user to assign them different costs, controlling the repair update generation in the process (Section 3.7.2), and even disable certain operations by assigning them high enough costs.

While techniques may use this set of edit operations to return operation-based repair updates (Section 3.6.2) to the user, this is not necessarily the case. For instance, techniques may internally rely on a fixed set of edit operations but still present state-based repair updates. In operation-based repair updates these can be returned as atomic repair actions or composed into repair plans.

3.7 Enumeration

This feature group defines the mechanism through which repair updates \( \mathcal{R} \) are selected and presented to the user by the repair procedure REPAIR, as well how this mechanism can be controlled (RQ5.1). These features are presented in Fig. 13, which is referenced by the general repair diagram in Fig. 12.
Fig. 13. Repair update enumeration features.

This feature selection is exclusive.

Our studies show that single repair updates can be returned by repair procedures following any of the core mechanisms (Section 3.6.1) and repair update representation (Section 3.6.2). The means through which these repair updates are selected may or not have been influenced by the user, as will be shown below.

**Complete:** Techniques that return multiple repair updates are said to be complete if they return every possible repair update within the parameters of the execution (i.e., the bounds of the search space, the allowed edit operations and any restriction imposed by the enforced semantic properties) [9], [10], [15], [20], [28], [37], [41], [45], [70]. Techniques that are not complete may discard interesting repair update alternatives or fail to handle certain inconsistencies. Again, this feature does not seem to be directly dependent on the selected core mechanism (Section 3.6.1): search-based approaches can search the whole search space, rule-based approaches may attempt to match every acceptable rule, and syntactic approaches may generate every possible alternative as the constraints are traversed.

**3.7.2 Order**

The set of the returned repair updates (Section 3.7.1), as well as the order in which they are enumerated, must be somehow selected by the repair procedures from the set of acceptable ones. This order is always defined, and can be embodied by a distance metric \( \Delta : U \times U \rightarrow \mathbb{N} \) over updates which the procedure tries to minimize.

In procedures that return a single repair update, this order determines which repair will be selected; in procedures that return multiple repair update alternatives, it determines the set of selected repair updates as well as the order in which they are enumerated. While related to least-change (Section 3.8.4), techniques with ordered repair enumeration that are not complete (Section 3.7.1) are not necessarily least-change, as the minimal repair update among the selected ones may not be the minimal repair update overall.

Such order may be internally defined and **opaque** to the user, which may render the procedure unpredictable [7], [10], [11], [12], [13], [37], [38], [39], [40], [42], [43], [44], [45], [47], [48], [50], [54], [55], [56], [58], [59], [61], [62], [63], [64], [65], [66], [68], [69], [73], [75], [78], [82]. This kind of approaches include search-based procedures returning an arbitrary repair update, e.g., the first found, or rule-based approaches that provide no control on how the rule application is selected, e.g., using some internal priory order over rules that is hidden from the user (Section 3.6.1). Some frameworks try to circumvent the unpredictability problem arising from opaque orders by providing the complete enumeration of repair alternatives (Section 3.7.1).

Other approaches have this order on repair updates predefined, rendering the technique more predictable [9], [14], [15], [20], [28], [41], [46], [49], [52], [53], [57], [60], [70], [72], [74]. Typically fixed metrics include the graph-edit distance, that counts insertions and removals of model elements, and operation-based distances, that count the number of edit steps between two models, given a set of valid edit operations (Section 3.6.3).

**Parameterizable:** Approaches with the enumeration order either opaque or predefined may allow users to parameterize the distance function \( \Delta \), thus enabling them to control the behavior of the repair procedure. For instance, under graph-edit distance, this can be achieved by assigning different weights to different parts of the meta-model [9], [41]. This allows the user to prioritize repair updates over certain types of model elements over others. Alternatively, the weights may be assigned directly to the model elements, prioritizing changes over concrete parts of the model instances [7], [9], [28], [37], [38], [44], [48], [51], [54], [55], [63], [64], [65], [66], [68], [72], [75], [74], [76], [77], [78]. An extreme form of this feature is in area selection, in techniques that allow the user to freeze portions of the model instances (as in bidirectional transformation where one of the model instances remains unchanged) [7], [28], [38], [44], [48], [51], [54], [55], [63], [64], [65], [66], [68], [72], [74], [76], [77], [78]. Instead of focusing on the models, the user may instead be allowed to control the application of the edit operations that comprise the repair updates (if these are well-defined (Section 3.6.3), as imposed by the excludes expression in the diagram) by attaching them with costs [9], [15], [41], [70]. Users may also be able to assign different priorities to the defined constraints, instructing the repair procedure to focus on different classes of violations [52], [74]. Finally, our study also found an approach where the user is able to control the procedure by relying on some additional meta-data from the environment, like authoring and versioning information [9].

Although in general this parametrization effectively affects the behavior of the repair procedure, some approaches use such features to simply provide the user with additional information regarding the impact of each possible repair update. Such weights can also be used by the checking procedure to return more informative reports.

**Interactive:** Techniques may rely on an interactive dialog with the user to refine the set of possible repair updates [51], [53], [62], [64], [71], [73], [77]. Most of the times the goal of the process is to select a single repair updates from the set of those available.

This feature contrasts with the generation of abstract repair updates (Section 3.6.2), where instead of an interactive dialog, the procedure generates repair updates that must be
they are detected; rule-based techniques with both check and repair rules are total there is at least a repair rule for each check rule. Syntactic techniques that focus on single violations at a time are typically total, while those that consider every inconsistency at once may encounter conflicts and fail to produce a repair update. Approaches with need for repair hints (Section 3.3.1) or user interaction (Section 3.7.2) may fail if the user-defined resolutions do not restore consistency.

### 3.8.2 Correctness

Since the goal of repair procedures is to remove inconsistencies from the environment’s state, they must provide some correctness guarantees. In fact, we have already defined model repair (Def. 2) under the assumption this notion can be formalized by a partial order \(\subseteq\) over inconsistency levels \(I\). Thus, the correctness of the model repair procedure is always measured in relation to that \(\subseteq\). However, as seen in Section 3.5.5, although for simple reports the shape of \(I\) entails the partial order, for composite reports that is not the case. Thus, there is the need to infer which is the expected behavior of the technique from the description of the technique.

- **Well-behaved**: A model repair procedure is said to be well-behaved if the inconsistency level at least does not increase whenever one of these repair updates is applied \([2, 39, 43, 45, 47, 48, 49, 53, 69, 78, 82]\), i.e.,

\[
\forall r \in \text{REPAIR}_e \ u \cdot \neg(\text{CHECK}_e u \sqsubseteq \text{CHECK}_e r(u))
\]

This is the minimal correctness behavior expected from a repair procedure. For instance, in boolean procedures, this means not turning completely consistent environments inconsistent ones; in those reporting the detected violations this forces procedures to not introduce new violations unless some those already detected were removed. In procedures with violation selection, this usually amounts to not introducing new violations if the selected one fails to be repaired. 

- **Consistency Improving**: Procedures that guarantee consistency improving effectively ameliorate the state of the environment, reducing its inconsistency level (unless it is already at a minimum inconsistency level) \([9, 10, 12, 13, 14, 41, 42, 52, 54, 59, 62, 64, 65, 68, 70]\). For a set of constraints \(c\) and update \(u\), this property can be specified as:

\[
\forall r \in \text{REPAIR}_e \ u \cdot \text{CHECK}_e r(u) \vee \exists i \in I \cdot i \sqsubseteq \text{CHECK}_e r(u)
\]

Consistency improving procedures are always well-behaved. If there is a single minimal inconsistency level \(I\), then it can be simplified as:

\[
\forall r \in \text{REPAIR}_e \ u \cdot \text{CHECK}_e r(u) \sqsubseteq \text{CHECK}_e u \lor \text{CHECK}_e r(u) = I
\]

Under boolean checking procedures this property degenerates into fully consistent procedures, defined below. Under more expressive checking procedures, like those reporting a set of violations, this behavior may occur in techniques that attempt to fix violations until a certain threshold is reached. Under distinguished constraints (Section 3.3.1) this is common in techniques that are only concerned with

---

0098-5589 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
a certain class of constraints (e.g., techniques dedicated to handle inter-model constraints may disregard intra-model constraints), or those supporting violation selection that effectively remove that violation (and may or not avoid side effects on the remainder violations).

**Fully Consistent:** Procedures are said to be fully consistent if they guarantee that the inconsistency level is always reduced to a minimum [15], [20], [28], [37], [38], [40], [55], [59], [60], [61], [63], [71], [72], [76], [77], i.e., for every update $u$ and set of constraints $c$:

$$\forall r \in \text{REPAIR}_c u \cdot \neg \exists i \in I \cdot i \sqsubseteq \text{CHECK}_c r(u)$$

Fully consistent procedures are always consistency improving. In case there is a least element $\bot_I$ in the inconsistency level, the law degenerates into

$$\forall r \in \text{REPAIR}_c u \cdot \text{CHECK}_c r(u) = \bot_I$$

The impact of this property depends on the minimal elements of the partially ordered set $I$. For instance, under boolean checking procedures, this amounts to setting the result to true, while under procedures that return a set of violations, this amount to fixing every violation (including possible negative side effects). This is the typical behavior of search-based approaches, that resolve all consistencies at the same time. In techniques with violation selection, this would entail fixing not only the selected violation, but also every other one identified, which would be against their essence.

Note that the definition of correctness is orthogonal to totality: procedures that fail to produce repair updates do not break correctness. In fact, some techniques enforce correctness by simply failing if generated repair update fails to ameliorate the consistency level.

Fully consistent procedures are not necessarily desirable, as the model may need to undergo inconsistent states before fully recovering consistency [7].

### 3.8.3 Stability

A repair procedure is said to be stable if for every update that does not result in an inconsistent state, it returns null repair updates [7], [9], [10], [11], [12], [13], [14], [15], [20], [28], [39], [41], [42], [43], [44], [45], [47], [48], [49], [50], [52], [53], [55], [58], [59], [60], [61], [62]. Thus, this order ordering may not be opaque, as is made explicit by the excludes expression in the diagram, and is possibly customized by the user (Section 3.7.2). This renders the approach more predictable to the designer since the set of selected repair updates is well-defined. However, while most approaches informally and loosely approximate this intuition using *ad hoc* or heuristic mechanisms, providing least-change guarantees is a complex task. In general, this technique is formalized as follows, for an update $u$ and constraints $c$:

$$\forall r \in \text{REPAIR}_c u \cdot \forall r' \in R \cdot \text{CHECK}_c r'(u) = \text{CHECK}_c r(u) \Rightarrow \Delta(r(u), u) \leq \Delta(r'(u), u)$$

Meaning that, compared with the repair updates that are equally consistent, the returned repair updates are closer to the current state of the environment. In purely state-based approaches, this degenerates into the following property, for a model $m$ and constraints $c$:

$$\forall r \in \text{REPAIR}_c m \cdot \forall r' \in R \cdot \text{CHECK}_c r(m) = \text{CHECK}_c r'(m) \Rightarrow \Delta(r(m), m) \leq \Delta(r'(m), m)$$

If the identity of indiscernibles holds for the distance function ($\Delta(m, m') = 0 \equiv m = m'$), then least-change entails stability. Otherwise there are minimal updates other than the null update.

### 3.8.4 Least-change

The principle of least-change requires repaired models to be as close as possible to the original, according to the defined order on updates $\Delta : U \times U \to \mathbb{N}$ (Section 3.7.2) [9], [15], [20], [28], [41], [47], [60]. Thus, this order order may not be opaque, as is made explicit by the excludes expression in the diagram, and is possibly customized by the user (Section 3.7.2). This renders the approach more predictable to the designer since the set of selected repair updates is well-defined. However, while most approaches informally and loosely approximate this intuition using *ad hoc* or heuristic mechanisms, providing least-change guarantees is a complex task. In general, this technique is formalized as follows, for an update $u$ and constraints $c$:

$$\forall r \in \text{REPAIR}_c u \cdot \forall r' \in R \cdot \text{CHECK}_c r'(u) = \text{CHECK}_c r(u) \Rightarrow \Delta(r(u), u) \leq \Delta(r'(u), u)$$

### 3.9 Threats to validity

The scope of the search was restricted to general-purpose software engineering venues. As a consequence, certain studies that were developed under specific application domains, but with possible general application, could have been disregarded. Our pilot searches did not identify any such study, since for every technique that was disregarded we found an extended or adapted to general-purpose techniques that were published in the venues within our scope.

Some features not explicitly covered by the authors of the primary studies may have been missed during data extraction and synthesis. The iterative nature of the coding process somehow tames this issue, since features detected in succeeding primary studies trigger a new revision of the previous studies focused on those newly identified features.

The major facet disregarded in the study regards the deployment of the identified approaches. While these are undoubtedly relevant, our pilot searches suggested that most studies do not address the deployment of the techniques, and those that do usually do not provide sufficient information in the paper. Thus, rather than having an under-explored facet, we chose to disregard deployment altogether (other than the technical space of the techniques, which is usually evident in the presentation).
4 Classifying Techniques

In the previous section we presented the taxonomy developed from the systematic literature review. In this section we classify a set of distinct model repair approaches under this taxonomy as a proof of concept [9], [10], [28]. We selected these approaches because i) they are recent approaches, based on modern, state-of-the-art techniques; ii) the primary studies presenting them were detailed enough to allow us classify with confidence most of the facets; and iii) they follow different core approaches, resulting in a varied selection of features. The goal is to demonstrate that classifying techniques through our taxonomy helps in obtaining structured and thorough descriptions which allow a better understanding and clear comparison of different approaches. Since some features may be difficult to assess for the primary studies (due to lack of information or ambiguity) our classification is conservative. The resulting classification is summarized in Tables 2, and 3 and discussed in the remainder of this section. Each of the columns represents a second-level feature. If that feature is optional, it is identified whether it was (Y) or not (N) selected by the approach; if the feature is mandatory, its selection is omitted. Every optional children feature of those second-level features selected by the approaches is also identified in that column.

To better illustrate the differences between the techniques, as well as the impact of the feature selection, we design a simple running example on which they are applied. Since we could not access the implementations of all these approaches, in order to define our profiles and infer actual repair updates, we resorted to the explicit information available in the literature and to our understanding of the techniques after a thorough study.

The example (borrowed from [9]) represents a more developed version of the VOD system and is composed by the class and sequence diagrams shown in Figs. 15a and 15b, respectively. The class diagram captures the structure of the system, while the sequence diagram describes the steps required in the process of playing a movie. Recalling constraint message_operation from Section 2.1 in order for this model to be consistent, for every message in the sequence diagram, there must exist an operation in the class of the receiver lifeline, whose name equals that of the message. Since there is no operation play in class Streamer, and display d sends message play to streamer st, this model is inconsistent. We only consider this single constraint in isolation, the repair space not being subject to any other restrictions, for instance related to some state diagram or to the associations between classes.

4.1 The Badger Approach

Badger [9] is a regression planner, implemented in Prolog, that generates repair plans for handling design model inconsistencies by applying the artificial intelligence technique of automated planning [86]. This technique aims to generate sequences of actions that lead from an initial state to a state meeting a specific predefined goal. Requiring as input a model and a set of inconsistencies, Badger performs a regression planning by starting from the negation of these inconsistencies as the goal state, and searching backwards to find a sequence of actions that reach the initial state.

Badger is based on a logical formalism, as model instances and meta-models are represented by logic facts, specified in a Prolog embedded Domain Specific Language (eDSL). The technique provides rules for defining meta-model elements, their properties and relationships, thus being meta-model independent. However, by having the Prolog eDSL as its technical space (other) and not providing any automated mechanism for the embedding of model instances nor meta-models persisted in standard languages, its integration into the MDE development process would not be seamless. This contrasts with both Model/Analyzer and Echo, which are deployed under standard technical spaces. Constraints are also user definable in the eDSL as intra-model logical constraints expressed in first-order logic with transitive closure. Since these constraints are defined in the same technical space as the model instances and meta-models, rather than being attached to the meta-model, they may refer to concrete model elements. In Badger, model instances and user updates are indistinguishable since they are not represented by the elements they contain, but rather by sequences of edit operations. The entire history record is kept (and also each pre-state), with authorship and versioning information attached to each edit step. This provides the repair procedure with rich information that is not available to those of Model/Analyzer nor Echo.

For detecting inconsistencies, Badger relies on a decoupled checking procedure proposed in [87], which returns model-level predicates corresponding to existing inconsistencies. These predicates are then negated and set as the goal of repair procedure.

Prolog’s built-in backtracking mechanism allows Badger to generate multiple repair plans, each one consisting of a set of repair actions that render the goal true. The core of the procedure is a domain-specific planner based on a recursive best-first search (RBFS) algorithm. Although this is an improvement of the well-known A* algorithm, which is known to be complete, it is not clear in the paper whether Badger provides a complete enumeration of plans or not. Badger has a fixed set of edit operations for creating and
deleting objects, as well as for creating, modifying and deleting properties or references on those objects. A benefit of using a search-based core is that the repair procedure enumerates the repair plans under a parameterizable order, which the user can control by tweaking the cost function used by the planner algorithm. For instance, the metric can be parameterized by assigning costs to edit operations, or weights to meta-model and model elements. Area selection and operation disabling can be achieved by assigning infinite costs. Since the whole history is recorded, costs over meta-data such as authors and versions can also be assigned. This contrasts with Model/Analyzer, that does not allow such customization, and with Echo, that allows the user to customize the edit operations (which are fixed in Badger). In order to avoid the multiplication of repair plans, for modifying references only (other operations are concrete), Badger resorts to temporary (abstract) elements which the user must replace by concrete ones when effectively applying the repair plan. As a consequence, repair updates cannot be automatically applied to the model instances, in contrast to fully concrete repair updates like those of Echo. Concerning semantics, Badger applies a consistency improving procedure as it generates plans transforming the erroneous model instance into one which does not have the detected violation (negated in the desired goal). However, by focusing in a single violation, it is not fully consistent, since other violations may be introduced by the repair plans (i.e., it is prone to negative side effects). Finally, the solution function used by Badger, which verifies whether there are no more unsatisfied literals in the desired goal, should ensure the stability of the procedure.

By default, the repair plans generated by Badger are ordered in terms of the number of actions they contain. For the defined example, the following eight plans to remove the violation are generated [9]:

1) modify reference target of message play
2) set property name of message play to stream
3) set property name of operation stream to play
4) set property name of message play to wait
5) set property name of operation wait to play

deleting objects, as well as for creating, modifying and deleting properties or references on those objects. A benefit of using a search-based core is that the repair procedure enumerates the repair plans under a parameterizable order, which the user can control by tweaking the cost function used by the planner algorithm. For instance, the metric can be parameterized by assigning costs to edit operations, or weights to meta-model and model elements. Area selection and operation disabling can be achieved by assigning infinite costs. Since the whole history is recorded, costs over meta-data such as authors and versions can also be assigned. This contrasts with Model/Analyzer, that does not allow such customization, and with Echo, that allows the user to customize the edit operations (which are fixed in Badger). In order to avoid the multiplication of repair plans, for modifying references only (other operations are concrete), Badger resorts to temporary (abstract) elements which the user must replace by concrete ones when effectively applying the repair plan. As a consequence, repair updates cannot be automatically applied to the model instances, in contrast to fully concrete repair updates like those of Echo.

Concerning semantics, Badger applies a consistency improving procedure as it generates plans transforming the erroneous model instance into one which does not have the detected violation (negated in the desired goal). However, by focusing in a single violation, it is not fully consistent, since other violations may be introduced by the repair plans (i.e., it is prone to negative side effects). Finally, the solution function used by Badger, which verifies whether there are no more unsatisfied literals in the desired goal, should ensure the stability of the procedure.

By default, the repair plans generated by Badger are ordered in terms of the number of actions they contain. For the defined example, the following eight plans to remove the violation are generated [9]:

1) modify reference target of message play
2) set property name of message play to stream
3) set property name of operation stream to play
4) set property name of message play to wait
5) set property name of operation wait to play
The parametrizable order results in alternative cost functions, which change the order in which repair plans are generated (disabling some if infinite costs are assigned). For instance, if one were to set a higher priority to the sequence diagram by assigning smaller costs to actions that create, modify or delete an element belonging to it – allowed by the operation costs feature – the order in which these plans would be generated becomes 1, 2, 4, 6, 8, 3, 5 and 7.

Although most generated repair plans are concrete, the first one, which suggests modifying reference target of message play, is an example of an abstract repair update. It avoids enumerating every lifeline, requiring the user to choose one when applying the repair plan. This renders the procedure more manageable by the user, at the cost of full automation.

Despite removing previously detected inconsistencies, Badger is not free from negative side effects. This is depicted in plan 7, which removes the violation for message play but introduces another violation of the same type for message connect. This is characteristic of consistency improving approaches with a loose order over the inconsistency levels, that guarantee the repair of the selected violation but disregard possible side effects. Considering the abstract syntax presented in its paper for the class and sequence diagrams, as well as all the types of repair actions supported by Badger, the enumeration of the repair plans does not seem to be complete. For instance, adding the missing operation to class Streamer or modifying reference class of st, would also be valid repair plans but are not generated.

4.2 The Model/Analyzer Approach

Model/Analyzer is a tool which follows an incremental approach to model repair, mainly focusing on efficiency. Using the syntactic structure of constraints, it determines which specific parts of a model must be checked and repaired. To achieve this, a form of profiling is used to dynamically observe constraint instances during evaluation in order to identify what model elements they must assess. Building upon this tracking mechanism, once a constraint instance is evaluated, the tool is able to generate a corresponding tree of repair actions.

Model/Analyzer is built over an object-oriented formalism and, even though the underlying repair technique is in theory applicable to any kind of models, the tool is implemented for UML (MDA) diagrams only, not providing any meta-modeling functionalities. This contrasts with Echo, which allows users to define meta-models through a standard language. Badger also allows the user to define the meta-models, but using an internal language. In the shape of intra-model logical rules, constraints are user definable by means of a generic language, called abstract rule language (ARL), to which it is possible to map arbitrary constraint languages, such as OCL. Once evaluated, the user is expected to select a specific violation to be fixed, instead of handling all inconsistencies at once. For each instance of each constraint, a consistency tree following its syntactic structure is kept in memory and dynamically evaluated in response to identified model updates (delta-based). When an element changes due to a modification in the model, every constraint instance having that element in its evaluation scope is notified. This works as a frame condition indicating the portion of the state that was effectively changed. Thus, unlike Echo and Badger, this tool is able to effectively detect elements that may cause violations.

This checking procedure is tightly coupled to the repair mechanism. In fact, it is the core of the technique, the repair procedure being built over it, and thus can be naturally run in checkonly mode. A violation is reported for each constraint instance that evaluates to false, an evaluated tree being returned. Since the involved model elements are localized through their leaves, one is able to understand where and why they failed.

The repair procedure is based on the comparison of the expected truth value of each consistency tree node, derived from its parent (ultimately, that of the root being true), with its actual observed valuation. Wherever these values differ, a corresponding repair node is generated accordingly to the type of consistency node (logical operator) and observed valuations. Since there may be more than a way to modify the valuation of a logical operator, alternative repair plans are returned for each violation, consisting of sequences of abstract and fixed edit operations (element creation, deletion, and modification). This results in repair plans which also follow the syntactic structure of the design constraint and represent enumerations of multiple repair plans. These abstract plans contrast with those generated by Badger, which are possibly concrete. The order in which the repair alternatives are enumerated is not well-defined and opaque to the user, in contrast with Badger and Echo. As a consequence, it is also not parameterizable by the user. The approach is incremental because once an update is performed, only those trees (and tree branches in particular) are evaluated which are affected by that particular change.

Regarding semantics, the repair procedure is consistency improving because it is guaranteed to remove the violation/s/trees selected by the user. Yet, similarly to Badger, it is not fully consistent because its goal is to remove only the selected violation. However, it still provides stronger guarantees than Badger since it checks for possible negative side effects (i.e., it considers a stronger partial order on inconsistency levels). Besides easily ensuring totality, this approach is also stable, as the repair update generation only occurs if the truth value of the consistency tree is false.

For the defined example, Model/Analyzer is expected to produce seven alternative repair plans, each consisting of a single repair action. Here we present the repair tree flattened into a set of alternative repair plans. Note that, unlike the list of repair plans generated by Badger, here the alternative plans are not ordered in any way clear to the user (i.e., this internal order is opaque):

5. A constraint instance corresponds to the evaluation of a constraint for a particular element of its context. For instance, in the example one would have one constraint instance per message.

6. This is done for conciseness, and possible because the tree would only include disjunction nodes.
Repair plans are generated either to fix the ranges of the quantifiers, or their predicates. In the former case, a repair action is suggested for each property referenced on the range’s expression, while in the latter case, a repair subtree is calculated for each element contained in that range. For message play in particular, the top three plans fix the range of the existential quantifier, while the other four fix its predicate. Note that modifying the class of the receiving lifeline, as well as adding an operation to its current class (respectively the second and third plan) are two particular repair updates missing in Badger’s repair plans. However, compared with that previous technique, this approach is instead missing the possibility of deleting message play itself. In fact, we did not find any information about how Model/Analyzer handles additions and removals of context elements, so the repair update enumeration might not be complete.

Unlike Badger, where a plan may suggest a concrete value to be assigned to some property, here all repair actions are abstract. For instance, the action suggesting to add an operation does not state whether this should be created anew or should come from another class, nor any suggestion to modify a name reveals what value should be used.

As a given tree is seen in isolation, one repair plan may render (once instantiated) another tree inconsistent (negative side effect). For instance, as Badger also suggests, modifying property name of operation connect (last plan) can only make message play consistent, if it also makes message connect inconsistent. Nevertheless, the authors stress that such potential side effects are detectable by checking whether a repair action of a repair tree references a model element belonging to the validation scope of other trees. In this sense, although the technique is still consistency improving (not every violation is removed), the order imposed over inconsistency levels is stronger than that of Badger.

4.3 The Echo Approach

Echo is a tool for consistency management based on the relational model finder Alloy [90], developed on top of the popular EMF. While initially built as a bidirectional model transformation framework [28], it eventually evolved to also handle intra-model consistency [91] and multidirectional transformation [60]. Thus, Echo is able to check and repair both inter- and intra-model consistency.

Since Echo’s kernel is the Alloy model finder, it is based on a relational formalism. Both model instances – following the standard structured language XMI – and meta-models – defined in EMF’s Ecore meta-modeling language – are processed into this formalism, rendering the technique meta-model independent. Moreover, Echo has support for multi-model environments, so multiple Ecore meta-models may be provided. Although its core engine is bounded, the repair procedure, presented below, guarantees that this feature is hidden from the user. Constraints are user definable, either through the embedding of OCL intra-model logical constraints as meta-model annotations, or through QVT-R specifications, a declarative language designed to specify inter-model consistency relations between related models. It also has support for the bidirectionalization of ATL transformations. All these types of constraints are expressed in first-order logic with transitive closure, and are also embedded into the Alloy core. Echo is state-based since it simply considers the post-state resulting from a user update. While this allows the technique to be run offline – since it does not need to record the user’s actions – it will render the technique less accurate than Model/Analyzer or Badger which take them into consideration. This also requires the procedure to check the consistency of the whole model instance at every execution.

The checking procedure is coupled to Echo and can be run in checkonly mode: once model instances, meta-models and the constraints are embedded into Alloy, its model checking capabilities are used to check the consistency of the environment. Thus, the checking procedure is essentially boolean. As a consequence, the user is not provided with much information regarding what caused the inconsistencies, unlike for instance Model/Analyzer that reports violations. However, intra- and inter-model constraints are distinguished, with Echo testing them independently, resulting in a composite checking report.

The core of the repair procedure is similar to that of the checking, but relying instead on Alloy’s model finding capabilities, that relies on off-the-shelf SAT solvers. Thus it is state-based, automatically calculates new model instances that satisfy the constraints. Being built over model finding, the procedure is naturally complete, enumerating multiple model instances. Despite being state-based, the user is able to customize the set of allowed edit operations that give rise to the generated instances, thus controlling their generation. Nonetheless, detecting what was effectively affected by the repair update may not be trivial, unlike in Model/Analyzer and Badger that calculate repair plans. However, contrary to those approaches, Echo’s repair updates are always concrete, which the tool converts into well-formed model instances. When acting on multiple models, Echo allows the user to select which of the model instances are to be affected by the repair updates (area selection).

The tool follows the principle of least-change, which is achieved by instructing the model finder to iteratively search for model instances at an increasing distance. Two predefined metrics are supported by Echo: graph-edit distance, that counts insertions and removals of atomic model entities, or an operation-based distance that counts the number of user-defined edit operations applied. The latter is controlled through the definition of the valid edit operations by the user. This contrasts with Badger, that allows the customization of the distance metric by assigning different weights to a fixed set of operations or model elements. Finally, due to its core based on model finding, this technique is naturally total, fully consistent and stable. This correctness guarantee is stronger than those of Badger and Model/Analyzer, that focus on removing a single violation. The trade-
off is performance, since this procedure does not scale for large model instances.

Regarding our example, it was encoded in Echo as an intra-model consistency problem to be compared with the other approaches, although the constraints could have defined as inter-model. Since Echo’s repair updates are state-based, new model instances are returned, rather than repair actions or plans. One of the consequences of this feature is that the user is not directly aware of the performed updates. Fig. 16 shows the repair update alternatives for this problem that are closest to the original model instance – due to the least-change property – under regular graph-edit distance, a predefined enumeration order. For model instances at the same distance from the inconsistent model, the order in which they are returned is arbitrary. Note that only fully consistent model instances are returned (e.g., no alternative renames operation \texttt{connect}, as it is being referred by another message), thus its characterization as fully consistent. The creation of a new operation and the deletion of the message are not among this initial set of alternatives, because they are not at minimal distance from the initial model instance. Nevertheless, once the minimal ones are enumerated, Echo starts producing the next closest ones, which would include those repair update alternatives, resulting in a complete procedure. Note that renaming the \texttt{connect} operation to \texttt{play} and changing the class of the \texttt{st} lifeline, although embodying minimal updates, are not produced by Echo, since they create negative side effects and would render the environment inconsistent.

The order of the returned solutions can be parameterizable by defining the set valid edit operations (through OCL pre- and post-conditions) and enforcing the operation-based distance. For instance, defining only operations to rename or delete messages, Echo would only return the model instances from Figs. 16a, 16b, and 16c and one where the message is deleted.

5 Conclusion

Inconsistency handling methods are vital to any software development process within the increasingly adopted MDE context. In this paper we propose a novel feature-based classification system for such techniques, that emerged from an exhaustive and systematic review of the diverse landscape of model repair, with the goal of allowing researchers and practitioners from different disciplines to properly locate and compare their work in a unifying framework. Supported by an underlying formalization of the problem of model repair, this taxonomy comprises five major classification facets, organized as hierarchical models that entail acceptable feature combinations. These facets address the shape of the relevant artifacts we set out to explore in this study, as well as the role of the user in specifying and customizing them. Despite the heterogeneity of the landscape of model repair approaches, the proposed classification is exhaustive and sufficiently flexible to classify existing approaches regarding these facets. The main relevant facet left out of the study regards the deployment of the techniques. We chose not to address such features due to the lack of information regarding the effective implementation of the approaches detected during the pilot searches.

The exhaustive classification of the primary studies selected in the literature review, published online [27], provides a snapshot of the current state-of-the-art of model repair approaches. Hopefully this can aid researchers and tool developers in identifying interesting feature combinations hitherto unexplored. For instance, each core mechanism of the repair procedures has pros and cons, but they are usually selected exclusively. Could hybrid approaches draw benefits from the various mechanisms? In any case, answering such questions would require the collection of additional information, like the approaches’ performance and scalability, which is outside the scope of this study. A quick glance at the table in [27] also shows that most techniques do not provide guarantees regarding the functional semantics of the model repair procedures. This fact, allied to the lack of information regarding the deployment of the techniques, indicates that perhaps the area has yet to reach the desirable level of maturity. We plan to keep this table up-to-date by rigorously reviewing new techniques as they are proposed, refining the taxonomy in the process with new methodologies if required, thus ensuring that it remains applicable and complete.

We thoroughly classify and explore three modern approaches to model repair under the proposed taxonomy, obtaining normative profiles which assist in understanding the techniques, and, since drawn from a common view point, make similarities and differences more obvious. The techniques are compared and the impact of feature selection is demonstrated by applying these techniques to a simple example. Although this comparison did not address every identified feature, we believe that the selected approaches are indicative of their respective classes and provide an overview of typical feature combinations. This, allied to the presentation and discussion of the various features as they are presented throughout the paper, should help MDE practitioners perform more informed decisions when selecting the model repair approach most suitable for their particular needs.

Acknowledgment

Work financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF) through project “NORTE-01-0145-FEDER-000016”.

References


