From Abstract to Concrete Repairs of Model Inconsistencies: an Automated Approach

Roland Kretschmer*, Djamel Eddine Khelladi*, Andreas Demuth*, Roberto E. Lopez-Herrejon† and Alexander Egyed*
*Institute for Software Systems Engineering
Johannes Kepler University Linz, Austria
Email: {roland.kretschmer, andreas.demuth, roberto.lopez, alexander.egyed}@jku.at
† ÉTS - University of Quebec, Notre-Dame Ouest 1100, H3C 1K3, Montreal, Canada
Email: first_name.last_name@etsmtl.ca

***Abstract*—A common task performed in model-driven software engineering is evolving models. This task is typically performed manually during the design or implementation phase of software projects and is known to cause inconsistencies. Despite extensive research on consistency checking, existing approaches either provide abstract (i.e., incomplete) repairs only, or they require manually predefined strategies on how to repair inconsistencies. In this paper, we present a novel approach that provides concrete (i.e., executable) repairs without the need of predefined repair strategies. Furthermore, our approach proposes functions which automate the generation of concrete repairs at runtime. An empirical assessment of the approach on six case studies from industry, academia and GitHub demonstrates its feasibility, and shows that the provided concrete repairs are relevant and can fix their corresponding inconsistencies automatically.**

I. Introduction

In model-driven engineering [1], [2], software models are used as the primary development artifacts to raise the level of abstraction and to make it easier for engineers to understand, analyze, and maintain software systems [3], [4]. A typical example of such software models are UML models (e.g., class, sequence, and state machine diagrams) but also domain-specific models [5], [6]. Using models has been proven to simplify communication between engineers and to have positive effects on software quality [7]. However, software models are subject to change not only during the initial development of a software system but also during maintenance [8]. Typical scenarios are: new features are added continuously to adapt the software to changing market demands, existing functionality is removed as it becomes outdated or the internal structure of a software system is changed to improve quality aspects such as maintainability or scalability. However, the benefits of models hinge on the fact that they must be kept consistent during the development process. Unfortunately as the models evolve, inconsistencies may arise which obviously need to be repaired for the models to remain useful [9], [10]. Such inconsistencies, if they remain undetected, can cause serious issues that range from project delay to failure [11], [12].

In literature, there is an extensive list of approaches for automatic analysis and detection of inconsistencies in software models (e.g., [13]–[18]). Those approaches often propose *abstract repairs* (i.e., identify an inconsistent model element, but do not say how to change it) but rarely *concrete repairs* (i.e., identifies how to change model elements with a concrete value). For example, the repair "rename class `Score`" is an abstract repair, which then can be transformed into a concrete repair "rename class `Score` to `Grade`". In this example, knowing the value `Grade` makes the abstract repair a concrete repair. This is crucial for automating the repair task of model inconsistencies since only concrete repairs can be executed automatically on the model. The challenge of providing concrete values for abstract repairs to turn them into concrete repairs is not a trivial task, because a large set of values may exist and identifying all those that will fix the inconsistency is difficult. For example, there are theoretically an infinite set of strings available for renaming the class `Score`. Yet practically, only a limited subset is able to form concrete repairs for fixing the inconsistency. Existing approaches for generating concrete repairs, on the one hand, require explicitly to define repair strategies [16], [17], [19], which can be time consuming, tedious and error prone to write and which are also limited to specific models or types of inconsistencies. On the other hand, they often provide a limited number of values which only satisfy a given set of patterns to repair the inconsistencies [18], [20], with the downside of not necessary covering all concrete repairs which lowers the repair quality and ultimately the model quality.

This paper builds on the work of Reder et al. [13], [21] that focused on computing abstract repairs. Although, in some cases concrete repairs are proposed in [13], [21] (such as to delete a particular element in the model), they focused mainly on computing abstract repairs. In this paper we primarily focus on computing concrete repairs that can fix model inconsistencies automatically. Our approach is made independent from [13], [21] and can be reused on top of approaches such as [19], [22] to obtain the abstract repairs needed for their transformation into concrete repairs. We propose a novel approach that uses internal information (i.e., values) of models to transform abstract repairs into concrete repairs automatically. The idea to rely on existing information for fixing inconsistencies has already shown to be efficient in fixing bugs [23]. This paper explores the same idea in the context of repairing model inconsistencies. The contributions of our paper are the following:

1) We propose various generator functions to generate concrete values that are retrieved from the model, which

allows us to compute potential values for abstract repairs. The generator functions are designed independently from any abstract repair, inconsistency, or model and are thus reusable.

2) We propose an algorithm to transform abstract repairs to concrete repairs. Our algorithm takes each abstract repair and a set of generator functions, and transforms the abstract repair into multiple correct concrete repairs. The algorithm is generic and can easily be extended with new generator functions.
   The algorithm explores in depth concrete repairs and keeps the ones that fix the inconsistency entirely. In particular, when several values must be combined to form an entire concrete repair, we only present to the user combinations of values that are able to fix entire inconsistencies automatically. This way the user is not overwhelmed.
3) Among the evaluated models, we evaluate three versioned models where the inconsistencies in the original models have been corrected by their users in the new version. Thus, we compare our computed concrete repairs with the actually applied user repairs to demonstrate the relevance of our approach and the computed concrete repairs.

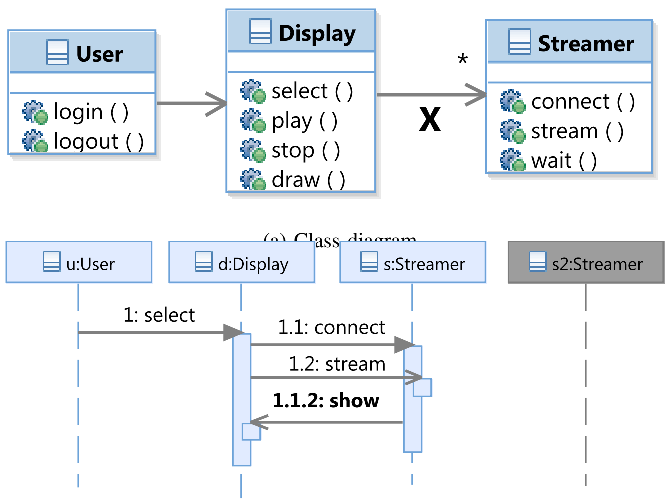
The approach has been implemented in a research prototype to demonstrate its feasibility, and it has been empirically evaluated to assess its performance. The evaluation shows that we are able to generate at least one concrete repair for 60% of all abstract repairs on average, and in 71% of the cases 1 to 5 concrete repairs are generated for each abstract repair. In total, we provide approximately 36000 concrete repairs for 3165 abstract repairs among 391 inconsistencies in 6 models taken from industry, academia and open source GitHub projects.

II. Running Example

To illustrate our approach, we use a motivating example of a *video on demand* (VOD) system which is based on a client-server architecture. Figure 1 depicts example snippets of the two different UML diagram types of this system: a class diagram and a sequence diagram.

In the class diagram in Figure 1a, class `User` initiates the process of selecting and displaying a movie. Class `Display` handles the user interaction, e.g., receiving user input and visualizing movies, and class `Streamer` manages the communication of a client with the movie server. The sequence diagram in Figure 1b describes the operation where a user selects a specific movie by calling method `select` on an instance of class `Display`, which then initiates a communication with an instance of class `Streamer`. It first connects to the `Streamer` (i.e., calls method `connect` on an instance of `Streamer`) and then starts playing the selected movie (i.e., calls method `stream` on instance `Streamer`). The `Streamer` object then sends frames to the `Display` instance via message `show`.

In this paper we use the *Object Constraint Language (OCL)* [24], a declarative language based on first order logic, to define our *consistency rules* for UML models. However, for the sake of simplicity, we will give an informal description of our consistency rules in this section. Table I shows two



(a) Class diagram

(b) Sequence diagram

Fig. 1: UML model snippets of VOD System.

TABLE I: Consistency Rules

Consistency Rule 1 (CR1)	Message direction must match class association and message must be defined as an operation in receiver's class.
Consistency Rule 2 (CR2)	Every lifeline has to have at least one message call from another class.

examples of consistency rules (CRs). *Consistency rules* define specific constraints that must hold in software models. These constraints express relations among model elements that can range from well-formedness to very domain-specific ones for non-functional properties such as maintainability or usability [25].

Let us now describe our two consistency rules which are instantiated in our model. CR1 ensures that every message direction in the sequence diagram matches the corresponding class' association, and messages in sequence diagrams are defined as operations in the message receiver's class. CR2 checks that every lifeline has at least one message call from another class. This might for example prevent dead code during instantiation. With these two CRs from Table I, the following two inconsistencies are identified in the model from Figure 1.

`I1` **Violation of CR1.** There is no corresponding operation in the class `Display` for the message `show` (message highlighted in bold text in Figure 1b) and the association from `Display` to `Streamer` forbids operation calls from `Streamer` to `Display` (highlighted with a bold X in Figure 1a).

`I2` **Violation of CR2.** Lifeline `s2:Streamer` has no message call from another class (`s2` highlighted in dark grey in Figure 1b).

Let us now take a closer look at the inconsistencies and alternatives on how they could be fixed. To fix `I1`, the association denoted with the bold **X** in Figure 1a has to be changed to be bidirectional, and the message `show` can be renamed to `play`, `stop`, `draw` or `select` to conform to class

`Display`. Another possibility would be to change association `X` to be bidirectional and to add an operation with the name `show` to class `Display`. To fix `I2`, we first need to have a look at the class diagram shown in Figure 1a. Here we see that only class `Display` is allowed to call class `Streamer` via their association. This means, that only message calls from lifeline `d:Display` to lifeline `s2:Streamer` are allowed in the sequence diagram (Figure 1b). One solution to fix `I2` would be to add a message `connect`, or `stream` or `wait` from lifeline `d:Display` to lifeline `s2:Streamer`. Other valid solutions would be to add two or three messages for the three operations.

The above repairs are all executable, i.e., they do not only suggest what to change, but how to change specific model elements. However, a challenging task is to propose concrete values (e.g., `play`, `stop`, etc). This paper proposes an approach for computing those values, combining and transforming them into model concrete repairs which are able to fix inconsistencies automatically.

III. Background

This section provides definitions and examples of the most important terms for a proper understanding of this paper. Our terminology changes substantially our previous work (see [21], [25], [26]), which we adopted and extended as follows:

Definition 1: **Model**. A model $\mathbb{M}$ consists of elements ($e \in \mathbb{M}$) where elements can have properties p that are accessed with the dot (.) operator e.g., "$e.p$".

Definition 2: **Scope Element**. A scope element is a model element and its corresponding properties ($e.p$) accessed during the validation of a consistency rule. A set of scope elements is called a scope.

Definition 3: **Cause**. A $cause_i$ of an inconsistency i is all the scope elements that violate the corresponding consistency rule. Hence, a cause is a subset of a scope.

Definition 4: **Repair Action**. A repair action defines a change of a model element property that resolves an inconsistency in part or full (often multiple repairs actions are needed to resolve an inconsistency). A repair action contains the model element (`e`), the property (`p`) that is affected by the change, the type of change (`ch`), and a value (`v`, which can be a model element $v \in \mathbb{M}$, or a primitive value $v \in \mathbb{V}$) or no value($\varnothing$) applied to the property. The following types of changes are possible: $\oplus$ adds a value, $\ominus$ deletes a value and $\odot$ modifies a model element property to the value. In addition there are the constraining changes: $\neq$, $<$, $>$, where respectively a property has to be different than the `value`, less than the `value`, or greater than the `value`. $\mathcal{P}(x)$ is the power set of x.

$$ra := \langle e, p, ch, v\rangle, ch \in \{\oplus, \ominus, \odot, \neq, <, >\},$$
$$v \in \mathcal{P}(\mathbb{V}) \cup \mathcal{P}(\mathbb{M})$$

$$execute : \mathbb{RA} \rightarrow \mathbb{M}, \langle e, p, ch, v\rangle \mapsto$$

$$x | x = (\mathbb{M} \setminus e) \cup e' | \begin{cases} e'.p = e.p \cup v & ch \text{ is } \oplus \\ e'.p = e.p \setminus v & ch \text{ is } \ominus \\ e'.p = v & ch \text{ is } \odot \\ e' = e & ch \text{ is } \neq \text{ or } < \text{ or } > \\ e' = e & v \text{ is } \varnothing \end{cases}$$

Furthermore we define the function $execute$, which performs a given repair action by applying the provided model change information to the model. This function maps from repair actions to new model states ($\mathbb{RA} \rightarrow \mathbb{M}$) by defining a new model state x in which the old model element e is replaced with a new model element e' ($(\mathbb{M} \setminus e) \cup e'$). For this new model element e' its corresponding property is changed based on the provided repair action change ch ($\oplus$, $\ominus$, $\odot$). If no value (v is $\varnothing$) has been provided or the change is different ($\neq$), greater than ($>$) or less than ($<$) nothing changes in the model ($e' = e$).

Definition 5: **Abstract Repair Action**. An abstract repair action is like a repair action, but with no concrete value ($\varnothing$). We also define the function $isAbstract$ which checks if a given repair action is abstract, i.e., if the value is equal to $\varnothing$ ($ra.v \Leftrightarrow \varnothing$) or the change is either $\neq$, $>$ or $<$.

$$isAbstract : \mathbb{RA} \rightarrow B, ra \mapsto ra.v \Leftrightarrow \varnothing \vee ra.ch \in \{\neq, >, <\}$$

As an example, the inconsistency `I2` discussed in Section II can be fixed by adding at least one unspecified message call to `s2:Streamer`. Expressed as an abstract repair action this leads to: $\langle s2, messagesReceived, \oplus, \varnothing\rangle$, which means that an unspecified value ($\varnothing$) has to be added to model element's property `s2.messageReceiver`. Note that this abstract repair action is a hint and is not executable, because we do not have a specific value to add.

Definition 6: **Concrete Repair Action**. A concrete repair action is like a repair action with always a concrete value ($v \in \mathcal{P}(\mathbb{V}) \cup \mathcal{P}(\mathbb{M})$)

We also define an operation eliminate ($\diamond$) which takes a specific set of concrete repair actions ($\mathbb{CRA}$) and removes their corresponding scope elements from the cause ($cause_i$) via execution ($execute$). Please note that $execute$ only removes the concrete repair action's corresponding scope element and not the entire cause.

$$cra := ra \in \mathbb{RA} | \neg isAbstract(ra) \wedge execute(ra) \diamond cause_i$$

To eliminate the cause for `I2` (from Section II) we have to execute the concrete repair action: $cra = \langle s2, messagesReceived, \oplus, Display.operations.connect\rangle$ that adds the message *connect*. After *cra* has been executed `s2.messagesReceived` is removed from the cause and does not take part in the inconsistency anymore. Note that it might be necessary to change multiple scope elements to fix an inconsistency. For that purpose, we define groups of repair actions as follows.

Definition 7: **Repair**. A repair is a non empty collection of repair actions (ra) that resolve a cause of a specific inconsistency (i). This set ra may contain both repair actions which can be abstract ($isAbstract(x)$) and/or concrete ($\neg isAbstract(x)$). If a repair action is concrete it also eliminates its cause by execution ($execute(x) \diamond cause_i$).

$$\langle i \in \mathbb{I}, ra \subseteq \mathbb{RA}_i | \{x | isAbstract(x) \vee \neg isAbstract(x) \Rightarrow (execute(x) \diamond cause_i)\}\rangle$$

Furthermore we define the term abstract repair which states that the set of repair actions contains at least one abstract repair action ($ra \subseteq \mathbb{RA}_i | (\exists x \in \mathbb{RA} | isAbstract(x))$), and we also define the term concrete repair which exclusively contains concrete repair actions ($ra \subseteq \mathbb{RA}_i | (\forall x \in \mathbb{RA} | \neg isAbstract(x))$). Please note that only a concrete repair is able to eliminate a cause entirely and therefore can fix an inconsistency.

As presented above, the abstract repair $\langle I2, \{\langle s2, messagesReceived, \oplus, \varnothing\rangle\}\rangle$ for the the inconsistency `I2` can be turned into a concrete repair by adding a specific message call `connect` to `s2.messagesReceived`, i.e., $\langle I2, \{\langle s2, messagesReceived, \oplus, Display.operations.connect\rangle\}\rangle$. The main contribution of this paper is the ability to transform abstract repairs to concrete ones, by generating concrete values with generator functions (see Section IV-B).

Definition 8: **Generator Function**. We define a generator function which maps the tuple set of all scope elements and model ($\langle \mathbb{SE}, \mathbb{M}\rangle$) to multiple model elements and their property values ($\mathcal{P}(\mathbb{M}) \cup \mathcal{P}(\mathbb{V})$).

$$gf : \langle \mathbb{SE}, \mathbb{M}\rangle \rightarrow \mathcal{P}(\mathbb{M}) \cup \mathcal{P}(\mathbb{V})$$

As an example, let us consider the abstract repair from the previous definition (Definition 7). It is obvious that we need concrete values (i.e., specific operations from classes) for this abstract repair to become a concrete repair. For this we use the generator function: $gf(\langle s2, messagesReceived\rangle, \mathbb{M}_{VOD})$, which returns all operations from the model: `select`, `play`, `stop`, `draw` from `Display`; `connect`, `stream`, `wait` from `Streamer` and `login` and `logout` from `User`. A more optimized way to return operations would be to return only `Display`'s operations, because this class has a direct association in the class diagram with `Streamer`. We also map this generator function to the type `lifeline` and the property `messagesReceived`, so we can get all needed operation names if we discover an instance of type `lifeline` (`s2` in this example).

IV. Overall Approach

This section presents our overall algorithm to convert abstract repairs to concrete repairs. First we give a general overview of the information flow, then we describe how we applied the concept of generator functions to our approach, and finally we discuss the implementation of the algorithm.

A. Information Flow

Overall, this section describes our approach and explains the main ideas behind the implementation of our abstract to concrete repair algorithm.

Figure 2 shows the basic workflow of our abstract to concrete repair transformation algorithm that consists of the following three stages:

The *first stage* (step 1) checks a model for `inconsistencies` with the provided consistency rules. Those inconsistencies are converted into a `tree structure` that describes the buildup of `abstract repairs`. Note that trees for abstract repairs do not have `leaves` (e.g., concrete values to repair an inconsistency) and are not able to fix an inconsistency. Those trees are constructed by parsing an OCL consistency rule and creating an abstract syntax tree for every instance of the consistency rule's context type in the model. They are then utilized to checked if the expected values of the consistency rule match those of the model. If there is a mismatch we know which model element and its corresponding property needs to be repairs (this model element is then a scope element). All model elements accessed during this process are considered the scope of an consistency rule. For more information on how consistency is checked and abstract repairs are generated, refer to [21], [25]. The *second stage* (step 2) then applies `generator functions` to retrieve sets of values (v_1, v_2, v_3) for specific model elements and their properties. Those values are then tested whether they are able to fix the corresponding inconsistency. All incorrect values are then removed from the value sets, for example 'a' from v_2 in Figure 2. In *stage three*, (step 3) we then combine the evaluated values from `stage two` with the knowledge from `stage one` and additional information about involved model elements. This finally leads to `concrete repairs`, which are then able to fix their corresponding inconsistency.

B. Generator Functions

At the core of our approach, generator functions are used to compute concrete values for generating concrete repairs. As we will see, Algorithm 2 is made generic and different generator functions can be used (from general purpose to domain-specific). We have defined 44 generator functions (21 type 1, 21 type 2, 2 type 3) so far, but engineers can easily add their specific generator function without changing our algorithm. In the following sections we give examples of different types for generator functions.

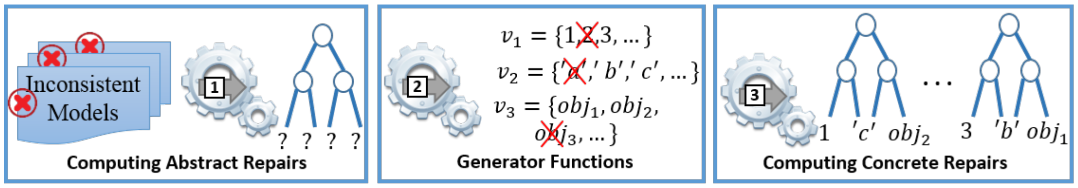
Type 1: All values of a specific type This type of generator functions returns all values of a specific type, as an example all strings or all operations. The benefit of this type of generator functions is that they are very generic and can be reused over a wide range of abstract repairs and models without changing them, as well as there are more chances that one of the values will fix the inconsistency. The disadvantage of this type of generator functions is that they may lead to a large amount of values to evaluate, because there are usually thousands of strings in medium to larger size models.

Algorithm 1 shows the implementation of a generator function of type 1, which returns all strings from a model for a property of type 'string'. The user then has to register this generator function for a specific type and property (e.g. type class and its property name). This allows our algorithm to call the generator function on all abstract repairs which involve fixing classes with names.

Algorithm 1 Generator function for all strings within a model

```
1: function GetAllStrings($m \in \mathbb{M}, p \in m, \mathbb{M}$)   ▷ m is a model element, p the corresponding property and $\mathbb{M}$ the model
2:     $values \leftarrow \varnothing$
3:     if p isType 'string' then
4:         $values \leftarrow \mathbb{M}$.getElementsOfType('string')   ▷ Utility function
5:     end if
6:     return $values$
7: end function
```

Type 2: All values of a specific type for a specific property In contrast to type 1 generator functions, this type of generator functions is tailored not only to a type but also



Inconsistent Models
1
? ? ? ?
Computing Abstract Repairs
2
v1 = {1, 2, 3, ...}
v2 = {'a', 'b', 'c', ...}
v3 = {obj1, obj2, obj3, ...}
Generator Functions
3
1 'c' obj2 ... 3 'b' obj1
Computing Concrete Repairs

Fig. 2: Overall approach of concrete repair generation.

to a specific model element property (e.g. names for classes, names for lifelines, messages for lifelines, etc). The advantage of this type of generator functions is that they return a subset of values of type 1 generator functions. For instance all class names are a subset of all strings, which leads to a reduced amount of value evaluations. The disadvantage of this type of generator functions is that in large software models this still can lead to a large amount of values, for instance there can be thousands of classes in a large software project.

Type 3: To reduce the amount of values returned by type 2 generator functions, it is possible to write generator functions for specific models or inconsistencies (type 3). However, this takes more effort in writing and also limits their versatility, because model specific information (e.g. specific class names, operations) has to be included, which limits the usability in other models drastically. Nonetheless, we implemented some type 3 generator functions and we will discuss their results in Section V. In contrast to type 1 and type 2 we are not able to provide type 3 generator functions for every model. Thus the user has to implement type 3 for her models on her own.

C. Algorithm

Algorithm 2 shows the pseudo code for our approach. The algorithm is structured into several phases for a better understanding.

Phase A: input and initialization (Lines 1–4). The input of our algorithm is a specific inconsistency `i`, a specific model element `m` and a set of generator functions `gf`. The model element `m` helps to focus only on those abstract repairs where `m` is involved.

Phase B: main iteration, preparing scope elements (Lines 6–13 and Line 34). This phase iterates over all relevant abstract repairs selected in phase one. First it selects the corresponding abstract repair actions from which scope elements are collected. This is needed to for the generator functions.

Phase C: Iteration over all scope elements (Lines 15–33). This phase iterates over all previously acquired scope elements and retrieves all values from the generator function by calling `getValues()`. Second, if the property `e.p` is not a collection (`isMultiValue()`) then the algorithm tries to reduce the amount of values by applying the abstract repair action's operation (Line 31).

For instance, consider the abstract repair action: $\langle account.money, >, 0\rangle$, which states that an account has to have at least some money on it. A generator function for `account.money` returns the following values: $values = \{-3, 5, 8\}$. The function `apply` now removes all values which do not satisfy the condition to be larger than zero. If a property is a collection then we enter phase D.

Phase D: Collection type properties (Lines 17–29). This phase computes if values have to be added or removed to property `p` (which is a collection of values), and performs the necessary generation of combinations. First the actual size of `p` is calculated (`actualSize`), and then the needed size `p` is computed to fix the corresponding inconsistency (`requiredSize`, this information is embedded in the inconsistency itself). If the collection `e.p` has too few elements ($actualSize < requiredSize$), the algorithm then calculates the needed amount of elements to be added (`diff`). We then generate all combinations of size `diff` from the value set. The combination generation process implements the binomial coefficient $\binom{n}{k}$, where the order of elements is not relevant and elements are unique. If the collection has too many values the procedure is analogously executed.

As example, consider `I2` from Section II, where `s2:Streamer` has no message call from another class. Expressed as an abstract repair we get the following: $\langle I2, \{\langle s2.messages.size, >, 2\rangle\}\rangle$, which states `s2:Streamer` has to have at least two message calls from another class. To convert this abstract repair a generator function returns the value set of all operations from class `Streamer`: `connect`, `stream`, `wait`. Now $\binom{|values|}{2} = 3$ combinations are generated: `connect`, `stream`; `stream`, `wait`; `connect`, `wait`.

Phase E: Cartesian product, validation (Lines 36–38). This phase of our algorithm performs the generation of the cartesian product (i.e., several scope elements in an abstract repair) and validates the resulting concrete repairs. First, the cartesian product is generated for all scope elements involved in the current abstract repair.

As example consider `I1` from Section II, where class `Display` has no operation `show` and the `association` between `Display` and `Streamer` is not bidirectional. Expressed as an abstract repair this leads to: $\langle I1, \{\langle s.show.name, \odot, \varnothing\rangle$, $\langle Streamer.association.bidirectional, \odot, true\rangle\}\rangle$, which states that `s.show.name` has to be renamed and the association has to be made bidirectional. To convert this abstract repair to concrete repairs a generator function for class `Display` returns its operations: `select`, `play`, `stop` and `draw`. Afterwards every operation of `Display` is combined with `Streamer.association.bidirectional`, which leads to the following concrete repairs: $\langle I1, \{\langle s.show.name, \oplus, 'select'\rangle, \langle Streamer.association.bidirectional, \odot, true\rangle\}\rangle$, $\ldots$, $\langle I1, \{\langle s.show.name, \oplus, 'draw'\rangle, \langle Streamer.association.bidirectional, \odot, true\rangle\}\rangle$.

Finally, all the combinations of scope elements and values have to be checked if they are indeed able to fix `i` [13]. This is done by applying the resulting repairs to the model, and comparing the changed model elements/values to the expected consistency rule's values in the concrete syntax tree.

Algorithm 2 Abstract repair to concrete repair algorithm

```
function CONVERT($i \in \mathbb{I}$, $m \in \mathbb{M}$, $gf \subseteq \mathbb{GF}$)    ▷ i is an inconsistency, m a model element and gf a set of generator functions
    $concreteRepairs \leftarrow \varnothing$    ▷ Contains all converted repairs in the end
    ▷ Contains all abstract repairs from i containing m
    $repairs \leftarrow \{x \in \mathbb{R} | isRepair(i, x.i) \wedge (\exists y \in x.ra | y.e \Leftrightarrow m.e \wedge y.p \Leftrightarrow m.p) \wedge isAbstract(x)\}$
    ▷ Iterate over all relevant abstract repairs in i to convert them to concrete repairs
    for all $r \in repairs$ do
        $repairActions \leftarrow \{x \in r.ra | isAbstract(x)\}$    ▷ Only convert abstract repair actions
        $scopeElements \leftarrow \varnothing$    ▷ Set of all relevant scope elements
        $se2v \leftarrow \varnothing$    ▷ Map for scope elements to values from their gf
        ▷ Collect all scope elements from repairActions to get model values
        for all $ra \in repairActions$ do
            $scopeElements \leftarrow scopeElements \cup \langle ra.e, ra.p\rangle$
        end for
        ▷ Prepare values for every scope element depending on property type
        for all $se \in scopeElements$ do
            $values \leftarrow getValues(se, \mathbb{M}, gf)$
            if $isMultiValue(ra.e.p)$ then    ▷ Property is a collection
                $actualSize \leftarrow |ra.e.p|$    ▷ Current size of the collection
                $requiredSize \leftarrow getRequiredSize(ra.e.p, i)$    ▷ Needed size to resolve i
                if $actualSize < requiredSize$ then
                    ▷ Collection has not enough values, calculate difference
                    $diff \leftarrow requiredSize - actualSize$
                    ▷ Generate value combinations
                    $values \leftarrow getCombinations(values, diff)$
                else if $actualSize > requiredSize$ then
                    ▷ Collection has too many values
                    $diff \leftarrow actualSize - requiredSize$
                    $values \leftarrow getCombinations(ra.e.p, diff)$
                end if
            else
                $values \leftarrow apply(values, ra.op, ra.v)$    ▷ For better efficiency
            end if
        end for
        $se2v \leftarrow se2v \cup \langle se, values\rangle$    ▷ Add scope element with its values
    end for
    $cp \leftarrow cartesianProduct(se2v)$
    $concreteRepairs \leftarrow getRepairs(cp, i)$    ▷ Convert Cartesian Product into concrete repairs
    return $validateRepairs(concreteRepairs, i)$
end function
```

V. Evaluation

This section evaluates our approach by assessing the correctness, applicability and usefulness. For the evaluation we applied 20 consistency rules to six models taken from three different sources: academia (VOD), industry (MVC, Dice) and GitHub (Pro11, fullAdder, activityMngr) [27]. Three models from GitHub have two versions each, where version one has inconsistencies which have been fixed in version two by a human, which allows us to compare our repairs with the actual applied ones. We selected those models randomly and without any pre-analysis to mitigate any biased evaluation. We applied the consistency rules with the Model/Analyzer, which has a compilation module integrated to check the syntactical correctness of the OCL consistency rules [25]. The model sizes range from 300 to 5000 model elements and the number of inconsistencies from 9 to 207. Table II shows details (e.g. number of model elements, number of inconsistencies, number of abstract repairs) of all models used. Theoretically the set of concrete repairs may be infinite (or too large), however, as we use concrete values from the models, it naturally limits the space to a finite set of concrete repairs that we explore. Nonetheless, we set an upper limit of 5000 generated repairs (cut off threshold) per abstract repair, which will be discussed in the threats to validity section.

A. Goals

In this section we define three research goals to evaluate our approach.

Goal 1: Abstract to concrete transformation. Investigate how many abstract repairs we are able to convert to concrete

TABLE II: Model information

Model Name	#Model Elements	#Incon-sistencies	#Abstract Repairs	Source
pro11	284	16	134	GitHub[a]
fullAdder	992	37	203	GitHub [b]
activity Manager	1185	51	270	GitHub [c]
VOD	467	9	43	Academia
MVC	1410	71	554	Industry
Dice	4485	207	1961	Industry

[a]https://github.com/11TCLC-DA-CNPM/doan-cnpm-quanly-ban-dien-thoai-pro11tclc/

[b]https://github.com/acdoorn/design-patterns/tree/master/diagrams/

[c]https://github.com/brunodevesa/DataStructures_PartThree/tree/master/diagrams

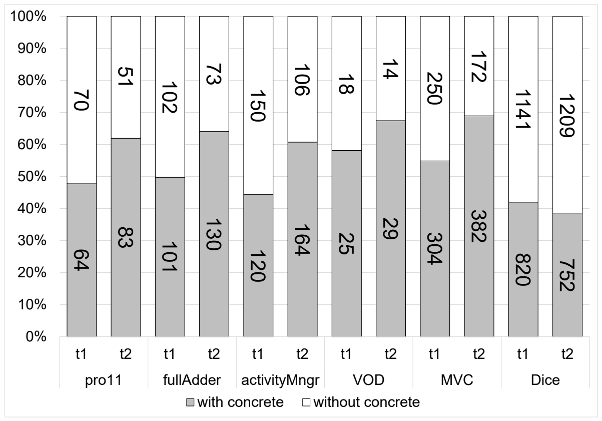


Fig. 3: Abstract Transformation.

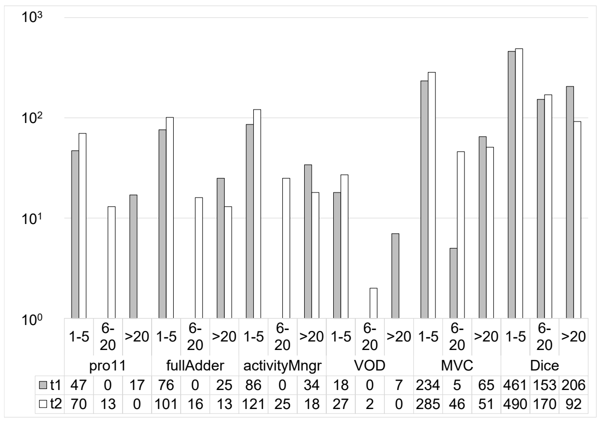


Fig. 4: Concrete Repairs Count.

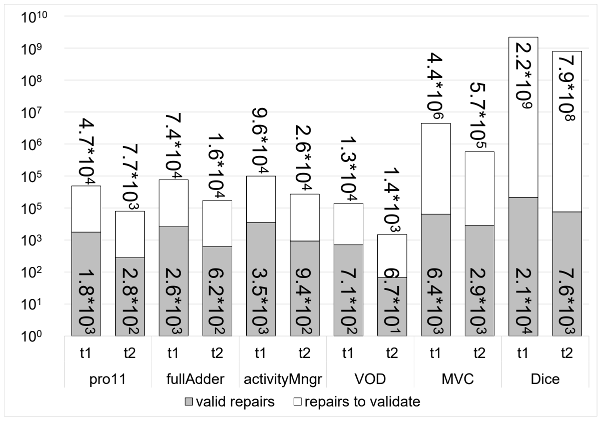


Fig. 5: Validation Count.

repairs (e.g. abstract repairs with no concrete repair vs abstract repairs with at least one concrete repair) for every model.

Goal 2: Relevant concrete repairs. Show that our generated repairs are relevant for real world consistency fixing. That means, whether we find repairs a user would also have applied to the models manually.

Goal 3: Relevant generator functions. Show that our used generator functions on the one hand are sufficient to find relevant concrete repairs, and on the other hand limit the amount of concrete repairs per abstract repair (i.e., compare results of type 1 and type 2 generator functions). Limiting the amount helps the user in the end to easily select one desired concrete repair without iterating over a large list of repairs.

B. Results

Goal 1: We applied all consistency rules to every model, and applied our approach and generator functions to all abstract repairs (we get from [13]) to transform them to concrete repairs. On average we were able to find at least one concrete repair for more than 55% (50% for type 1, 60% for type 2) of the abstract repairs.

Figure 3 shows the the percentage of abstract repairs that were transformed to concrete (grey), and the percentage of those repairs that were not transformed (white).

The numbers in the bars show the absolute amount of abstract repairs with and without concrete repairs. Note that the number of scope elements varied from 1 to 7, and we were able to find concrete repairs regardless the number of scope elements. However, the more scope elements we had, the more concrete repairs we generated. Abstract repairs for which we could not find concrete repairs needed information not present in the model but new information to be created. This was expected since only the user can create new knowledge that only she is aware of.

As an example for the Video on Demand model (VOD) we were able to find at least one concrete repair for 29 abstract repairs when applying type 2 generator functions. For 14 abstract repairs we could not find any concrete repairs, because of the reason mentioned earlier. For instance, applying a consistency rule which states that *"a message direction must match the class direction"*, to the model presented in Section II results also in the following abstract repair for lifeline `s`: $\langle I3, \{\langle s.type, \odot, \varnothing \rangle\}\rangle$. This abstract repair states that to resolve `I3` the type of `s` has to be changed to a **class** which has the correct association, and also the correct operation. But since there is no **class** already present in the model which satisfies the conditions of CR1, we did not find any concrete repair for this abstract repair. Again, only the user would be able to create this class. Creating classes based on pattern generation could fix the inconsistency. However, it could also reduce the quality of the concrete repairs [18], since there might be nonsense generated values. For instance, a pattern generator can create a class with an empty name or the name `X20SZ` together with a bidirectional association to class `Display`, which repairs *I3*. However, an empty name or the name `X20SZ` is not guaranteed to be understood by a human and most likely needs to be changed by the user after generation.

Goal 2: In this evaluation, we applied our approach to three versioned models (`pro11`, `fullAdder`, `activityMngr`) taken from GitHub. Every model contains several inconsistencies in version 1 and the model designer has manually fixed those inconsistencies in version 2 (they are not present anymore). In the set of our generated concrete repairs for version one, with both generator function types (1 and 2) we were able to find every concrete repair the designer has applied manually for every inconsistency in every model to version one. For instance, example for inconsistencies in `pro11` were messages with incorrect names (no corresponding operations) and incorrect type specifications for the lifelines. Examples of the computed concrete repairs were to rename the messages with existing operations and to change the lifeline's type with existing classes. This means that our approach covers all (both type 1 and type 2 generator functions) of the designer's needs regarding the repair of inconsistencies, with respect to our three versioned models. Of course our approach also suggested additional concrete repairs, which may be of interest for other model designers.

Goal 3: Figure 4 shows the amount of concrete repairs per abstract repair separated into three classes. Those classes range from 1 to 5, 6 to 20 and more than 20 concrete repairs. On the y-axis you can see the amount of abstract repairs in the corresponding class (note the logarithmic scale). We furthermore calculated for both generator functions types 1 and 2 the average percentage of concrete repairs per category, e.g., $\sum repairs_{1-5} / \sum repairs$. Where $repairs_{1-5}$ is the class from 1 to 5 concrete repairs. From Figure 4 we can also see that type 2 generator functions are able to convert 71%[1] of all abstract repairs into one or up to five concrete repairs, 18% have 6-20 and 11% have more than 20 concrete repairs. Type 1 was able to convert 64% of all abstract repairs into one or up to five concrete repairs, 11% have 6-20 and 25% have more than 20 concrete repairs. This means that in addition to finding relevant repairs, in 71% (64% for generator functions type 1) of the cases the user is not overwhelmed, but chooses only between one to five concrete choices for a given abstract repair to repair the inconsistency. Note that the very large number of concrete repairs per model in Figure 5 is due to the few abstract repairs for which more that 20 concert repairs were computed (e.g., 100 in some cases).

Figure 5 shows the amount of generated repairs per generator function type and model. This amount has been summed up over all abstract repairs from the specific generator function type and model (note the logarithmic scale on the y-axis). This figure shows that type 2 generator functions have a large impact (reduced by one order of magnitude) on the amount of repairs to be validated, thus they significantly improve the performance for the validation process. However, this also might lead to a reduction of concrete repairs, since they do not return every possible value. For example, for the `pro11` model and its 16 inconsistencies, type 1 generator functions produced around 50 thousand potential repairs, whereas type 2 produced 7700, which after validation resulted in about 1800 (type 1) and 280 (type 2) concrete repairs. This results on average in 13 concrete repairs per abstract repair for type 1, and two concrete repairs per inconsistency for type 2.

[1] $= \frac{70+101+121+27+285+490}{70+13+101+16+13+121+25+18+27+2+285+46+51+490+170+92}$

As mentioned in Section IV-B, we wanted to assess whether type 3 generator functions specific to a given model deliver better results. We have written three generator functions of type 3 and tested them on one model (pro11). They reduced the amount of generated repairs by 90% (compared to type 1 generator functions) while keeping the valid repairs from type 2 generator functions and the ones that were applied by the user in version 2 of model pro11. As an example, we have defined a type 3 generator function for lifeline message names, which only returns operation names of the corresponding lifeline's class. To do that, we have to go from the message to its corresponding lifeline, from the lifeline to the class, and from the class to the operation and their names. Although, the example given above of a type 3 generator function could be easily written. Depending on the user intent to retrieve targeted values writing type 3 generator functions might require more effort to write and are not universally reusable for other models, due to their tailoring to specific model elements.

Please note that our approach provides only concrete repair alternatives from which the user has to choose in the end. They are not executed randomly.

VI. Limitations

This section discusses our current approach limitations that will be addressed in future work. In this paper we rely on internal model information including changes that causes inconsistencies to compute concrete repairs (inspired from [23]). Thus, abstract repairs that require new knowledge are not transformed into concrete repairs since only the user can provide the needed new values.

As we have seen in the previous section, due to the exponential problem of computing concrete repairs, our approach explores all combinations of values and tests whether they fix the inconsistencies or not. The main reason for this is the combination of values of multiple scope elements. For instance, if we have seven scope elements to repair in an inconsistency, where every scope element has just 10 values from generator functions, this results in 10^7 combinations, and thus to the same amount of repairs to check. This results in the limitation that we also test invalid value combinations due to two main reasons, either because 1) an invalid value exists for one scope element or 2) values from two or more scope elements are contradictory with each other (although the values are valid on their own). As an example, Figure 5 shows the results obtained from analyzing model `pro11`, where for type 1 generator functions $4.7 * 10^4$ repairs are tested, but only $1.8 * 10^3$ actually fix the inconsistencies, i.e., 96% of invalid combinations not fixing the inconsistencies, whereas 4% are actually concrete repairs. However, in the end, we do not present those invalid combinations to the user, and thus, she is not overwhelmed with useless choices. We only present the concrete repairs that are able fix the inconsistencies. This limitation affects the performance of our approach, and therefore, only computation time suffers. For the model sizes from 284 elements to 1410 elements the time to generate all concrete repairs was from 0.5s to 2 minutes. Only for the largest model `Dice` it took 20 minutes overall and 6 seconds per inconsistency on average. To reduce the amount of combinations we need a mechanism to identify invalid values before combining them which is a non trivial task.

Another limitation is that our approach cannot find one single concrete repair for multiple model elements which are depending on each other, only for each model element on its own, which may not repair the whole inconsistency. In this case, the repair is not proposed to the user among alternative repairs. This is left for future work.

VII. Threats to validity

In this section we discuss internal, external and conclusion threats to validity after Wohlin et. al. [28].

Internal Validity: The internal threats to validity are centered on the cutoff threshold during the repair computation process. This threshold limits the validation process to a fixed amount of 5000 concrete repairs for one abstract repair. With this limit set, the evaluation might miss some valid concrete repairs in seldom cases, where an abstract repair contains a large amount of scope elements (in our evaluation 25% AR had more than 4 scope elements). However, only type 1 generator functions are mainly affected, because they return the largest amount of values. We chose the threshold to be 5000 after trying different sizes (500, 1000, 5000 and 10000), because we observed that after 5000 the number of validated concrete repairs becomes stable while relevant repairs (that are applied by the user in case of the three versioned models) are still computed in our case studies. Thus, we deem this threat to validity as acceptable here. Moreover, our approach depends on the expressed OCL consistency rules which are specified by the user. We only check their syntactic correctness, but not their semantic correctness/completeness since only the user knows its intent. It is also not possible to guarantee that our approach will always find concrete repairs, since the generator functions purely rely on model internal information and any arbitrary model can be used.

External Validity: We implemented our approach for UML and OCL, although we are confident that the generation of concrete repairs is also applicable to other modeling languages like SysML, XML (and XSD), we cannot generalize our results to all modeling constraint languages. However, The only requirement to apply our approach to other domains, is to check consistency rules (detect inconsistent locations), and to retrieve model internal values. In future work we plan to evaluate on other modeling languages as well.

Conclusion Validity: Our evaluation gives promising results (quantitatively and qualitatively), demonstrating that repairing a model with only internal information is possible and relevant/useful, thus we achieved all three goals from Section V-B. The results in our case studies indicate that we are able to convert a large amount of abstract repairs, which do not require new information in the model, to concrete repairs. However, we only had 3 versioned models that showed the relevance of our concrete repairs. To have more evident results, we want to evaluate on more versioned models.

VIII. Related Work

This section focuses on approaches that repair model inconsistencies. Finding concrete and executable repairs in software models is an active field of research. This section presents and discusses the works closest to ours.

Abstract repairs: As presented in Section IV, our approach relies on abstract repairs as input for finding corresponding concrete repairs. Abstract repairs have been shown to be an effective and easy way of providing inconsistency information [13], [19], [25], [29]. However, our approach would also work with triple graph grammar rules [30] or plain rule parsing. For our prototype implementation, we employed the Model/Analyzer consistency checking framework for finding inconsistencies and obtaining abstract repairs [13]. However, other approaches that provide abstract repairs may also be used as input for our approach to generate concrete repairs. For instance Xiong et al. and Jackson et al. use a very similar notation of abstract repairs, which would be suitable for our abstract to concrete repair algorithm [19], [29]. In summary, the only requirement for other approaches is that they provide affected model elements, their properties and the corresponding repair operation. Note that the Model/Analyzer is able to find concrete repairs in rare cases [21], but does not aim entirely at finding concrete repairs as in our paper.

Concrete repairs: There are multiple approaches for repairing models. For instance, da Silva et al. generate concrete repairs by defining cause detection rules combined with effect canceling functions [16]. Similarly, the approach presented by Xiong et al. requires engineers to adapt OCL constraints to provide fixing information within a consistency rule [19]. In contrast our approach does not require to manually define how certain inconsistencies should be fixed, instead it only requires defining generator functions to obtain meaningful values for the model elements and their properties.

Nentwich et al. also define repair actions and repairs, and they are able to perform consistency checking on UML models [22]. Da Silva et al. also use repairs to resolve inconsistencies in their UML models and try to find concrete versions of abstract repairs [16]. Also Xiong et al. may be used to define consistency rules and fix model inconsistencies with the Beanbag language [19]. We extended the approach of Reder et al. that was built for usage with the employed incremental consistency checker [13]. Moreover, this approach does not require fixing-related statements to be added to the applied OCL constraints, as it is in [19]. Neither does it try to execute adaptations automatically as it is proposed by [16], [19].

Kolovos et al. specify cross-model constraints to define consistency rules over model elements and provide fixing strategies for those constraints [17]. However in case of an inconsistency it is necessary to manually select which of the provided fixing strategies has to be executed. In contrast to our approach we perform this task automatically by validating all previously generated concrete repairs.

Another relevant approach for getting concrete repairs for models is shown in Hegedues et al. where a Constraint Satisfaction Problem solver (CSP solver) is used to repair inconsistencies for Domain-Specific Modeling Languages (DSMLs) [18]. In contrast to this approach we are able to get concrete repairs not only for DSMLs but any modeling language and our approach is capable of providing not only syntactically correct values, but also semantically correct values through generator functions. Puissant et al. proposed a planning technique to generate repair plans for inconsistencies while aiming at a fast computation of repairs without assessing the relevance of the repair plans [31]. In their repair plans, they allow generation of random elements which can reduce the quality of the models and they do not rely on existing information in the models. To the best of our knowledge, we are the first to compute concrete repairs with model intrinsic information, and do not need user interaction during runtime.

Le et al. propose repairs for bugs in programs by applying structured specification, deductive verification and genetic programming [32]. They also further elaborated this approach with automated example extraction and repair synthesis based on those examples [33]. Similarly Ma et al. focus on vulnerability repair in source code by learning from training sets and deducing repair templates to fix those vulnerabilities [34]. In contrast to those approaches, we do not need test cases or training sets to check the correctness/fitness of the generated repairs, and therefore no additional user input is necessary.

IX. Conclusion and Future Work

In the paper we presented a novel approach for automatically generating concrete and executable repairs for models in software development. Our approach is based on inconsistency information, abstract repairs and a set of generator functions. We proposed different types of generator functions which can be used generically for different models and inconsistencies. This means, the user can always add generator functions that are specific to her context and models. We evaluated our approach by applying 20 consistency rules to 6 models. To check the relevance of our generated repairs, we have used 3 versioned models from GitHub and showed that we are able to replicate 100% of the modeler's fixing actions. Furthermore, we have shown that on average we are able to find at least one concrete repair for more than 55% on average, and 65% for type 2 generator functions of all abstract repairs.

For future work we plan to perform an incremental generation and validation of value combinations i.e., check valid values before combining them. Additionally, our evaluation did not consider side effects of repairs (i.e., executing one repair leads to an inconsistency).

We plan to construct a graph which reflects the side effects of repairs, and a loop detection algorithm which can be applied to filter concrete repairs that lead to an endless loop. Finally, when lots of alternative repairs are computed, we plan to provide ranking heuristics to help the user in selecting an appropriate repair, e.g., ranking repairs based on the number of their model changes or possible side effects.

Acknowledgment. The research leading to these results has received funding from the Austrian Science Fund (FWF) under the grants FWF P 25289-N15.

References

[1] S. Kent, "Model driven engineering," in *IFM*, pp. 286–298, 2002.

[2] S. Beydeda, M. Book, V. Gruhn, *et al.*, *Model-driven software development*, vol. 15. Springer, 2005.

[3] A. Egyed, "Automatically detecting and tracking inconsistencies in software design models," *IEEE Trans. Software Eng.*, vol. 37, no. 2, pp. 188–204, 2011.

[4] S. H. Kan, *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[5] C. Ashbacher, ""the unified modeling language reference manual, second edition", by james rumbaugh," *JOT*, vol. 3, no. 10, pp. 193–195, 2004.

[6] R. France and B. Rumpe, "Domain specific modeling," *Software and Systems Modeling*, vol. 4, no. 1, pp. 1–3, 2005.

[7] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[8] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[9] M. Morisio, M. Ezran, and C. Tully, "Success and failure factors in software reuse," *IEEE TSE*, vol. 28, no. 4, pp. 340–357, 2002.

[10] W. B. Frakes and K. Kang, "Software reuse research: Status and future," *IEEE TSE*, vol. 31, no. 7, pp. 529–536, 2005.

[11] M. Van Genuchten, "Why is software late? an empirical study of reasons for delay in software development," *IEEE Transactions on software engineering*, vol. 17, no. 6, pp. 582–590, 1991.

[12] R. N. Charette, "Why software fails [software failure]," *IEEE Spectrum*, vol. 42, no. 9, pp. 42–49, 2005.

[13] A. Reder and A. Egyed, "Incremental consistency checking for complex design rules and larger model changes," in *MODELS*, pp. 202–218, 2012.

[14] C. Nentwich, W. Emmerich, and A. Finkelstein, "Static consistency checking for distributed specifications," in *ASE*, p. 115, 2001.

[15] X. Blanc, I. Mounier, A. Mougenot, and T. Mens, "Detecting model inconsistency through operation-based model construction," in *ICSE*, pp. 511–520, 2008.

[16] M. A. A. da Silva, A. Mougenot, X. Blanc, and R. Bendraou, "Towards automated inconsistency handling in design models," in *CAiSE*, pp. 348–362, 2010.

[17] D. S. Kolovos, R. F. Paige, and F. Polack, "Detecting and repairing inconsistencies across heterogeneous models," in *ICST*, pp. 356–364, 2008.

[18] Á. Hegedüs, Á. Horváth, I. Ráth, M. C. Branco, and D. Varró, "Quick fix generation for DSMLs," in *VL/HCC*, pp. 17–24, 2011.

[19] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei, "Supporting automatic model inconsistency fixing," in *FSE*, pp. 315–324, 2009.

[20] S. M. Shah, K. Anastasakis, and B. Bordbar, "From UML to alloy and back again," in *MODELS*, pp. 158–171, Springer, 2009.

[21] A. Reder and A. Egyed, "Computing repair trees for resolving inconsistencies in design models," in *ASE*, pp. 220–229, 2012.

[22] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency management with repair actions," in *ICSE*, pp. 455–464, 2003.

[23] M. Martinez, W. Weimer, and M. Monperrus, "Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches," in *ICSE '14, Companion Proceedings*, pp. 492–495, 2014.

[24] OMG, "Object Constraint Language," 2014.

[25] A. Reder and A. Egyed, "Determining the cause of a design model inconsistency," *IEEE TSE*, vol. 39, no. 11, pp. 1531–1548, 2013.

[26] A. Egyed, E. Letier, and A. Finkelstein, "Generating and evaluating choices for fixing inconsistencies in UML design models," in *ASE*, pp. 99–108, 2008.

[27] R. Hebig, T. H. Quang, M. R. Chaudron, G. Robles, and M. A. Fernandez, "The quest for open source projects that use UML: mining github," pp. 173–183, ACM, 2016.

[28] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[29] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, 2002.

[30] H. Giese and R. Wagner, "From model transformation to incremental bidirectional model synchronization," *Software and Systems Modeling*, vol. 8, no. 1, pp. 21–43, 2009.

[31] J. P. Puissant, R. Van Der Straeten, and T. Mens, "Resolving model inconsistencies using automated regression planning," *Software & Systems Modeling*, vol. 14, no. 1, pp. 461–481, 2015.

[32] X.-B. D. Le, Q. L. Le, D. Lo, and C. Le Goues, "Enhancing automated program repair with deductive verification," in *ICSME, 2016 IEEE International Conference on*, pp. 428–432, IEEE, 2016.

[33] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax- and semantic-guided repair synthesis via programming by examples," *FSE. ACM*, 2017.

[34] S. Ma, F. Thung, D. Lo, C. Sun, and R. H. Deng, "Vurle: Automatic vulnerability detection and repair by learning from examples," in *European Symposium on Research in Computer Security*, pp. 229–246, Springer, 2017.