Transforming Abstract to Concrete Repairs with a Generative Approach of Repair Values

Roland Kretschmer, Djamel Eddine Khelladi, Alexander Egyed

Transforming Abstract to Concrete Repairs with a Generative Approach of Repair Values

Roland Kretschmer[a], Djamel Eddine Khelladi[b], Alexander Egyed[a]

[a] *Institute for Software Systems Engineering, Johannes Kepler University, Linz, Austria*
[b] *CNRS, IRISA UMR6074, University Rennes 1, Rennes, France*

Abstract

Software models, often comprise of interconnected diagrams, change continuously, and developers often fail in keeping these diagrams consistent. Detecting inconsistencies quickly and efficiently is state of the art. However, repairing them is not trivial, because there are typically multiple model elements that need to be repaired, leading to an exponentially growing space of combinations of repair choices. Despite extensive research on consistency checking, existing approaches either provide abstract repairs only (i.e., identifying the model element but failing to describe the change), which is not satisfactory. This paper presents a novel approach that provides concrete repair choices based on values from the inconsistent models. Thus, our approach first retrieves repair values from the model, turn them to repair choices, and groups them based on their effects. This grouping lets our approach explore the repair space in its entirety, providing quick example-like feedback for all possible repairs. Our approach and its tool implementation have been empirically assessed on 10 case studies from industry, academia, and GitHub to demonstrate its feasibility and scalability. A comparison with three versioned models shows that our approach identifies useful repair values that developers have chosen.

Keywords: Model Repair, Inconsistency Repair, Abstract Repair, Concrete Repair

1. Introduction

Model-Driven Engineering (MDE) has shown to be effective in the development and maintenance of large scale and embedded systems [1, 2]. *MDE* typically puts models as a central artifact in the various phases of the development process [3, 4]. Indeed, models are used in all development stages, from specifying the customer's requirements, design, all the way to source code, with

Email addresses: roland.kretschmer@jku.at (Roland Kretschmer), djamel-eddine.khelladi@irisa.fr (Djamel Eddine Khelladi), alexander.egyed@jku.at (Alexander Egyed)

the benefits of increased productivity and reduced time to market [5, 6, 7]. These benefits, however, hinge on the assumption that models remain consistent during development. This is obviously a problem during evolution when changes happen. Changes may cause inconsistencies and subsequent errors if engineer do not recognize these inconsistencies in a timely manner. Moreover, if models are inconsistent, all automation around them is untrustworthy and likely causes further errors. Therefore, inconsistencies must not only be detected but ultimately be repaired [8, 5, 9].

A repair is typically a set of model changes that together fix a given inconsistency. Literature distinguishes *abstract* and *concrete* repairs [10, 11]. An *abstract* repair identifies a model element to repair (a location in model) but does not reveal how to change it. A *concrete* repair additionally identifies how to change the model element (with a concrete value). A *concrete* repair can thus be executed automatically on the inconsistent model to eliminate the inconsistency.

As an example, imagine that a message is passed among two components and the name of this message is inconsistent, perhaps because its name was not declared. One possible repair of this inconsistency is to change its name. Knowing this, constitutes an abstract repair because it identifies the location (i.e., the name of the message that needs repairing), but not the concrete value to what the message should be renamed to. Adding this concrete value makes an abstract repair a concrete repair. The challenge of providing concrete values for abstract repairs to turn them into concrete repairs is a non trivial task, because a large set of values may exist and identifying all that would fix the inconsistency is difficult. Continuing on the previous example, the problem is that there are infinite strings available for renaming that message. Thus, there are practically infinite concrete repairs and it is infeasible to compute and impractical to list them all. The challenge is thus to identify a subset of names that could form actual useful repairs. Whereas abstract repairs for model inconsistencies are computable in a scalable manner [12, 13, 14, 15], it is challenging to deal with concrete repairs in a scalable manner and provide a sufficient amount of choices, i.e., don't overwhelm the engineer with too many options, yet provide the repairs needed. This paper addresses these very particular challenges.

State of the art identifies two kinds of strategies for computing concrete repairs for model inconsistencies. The first kind relies on predefined repair strategies tailored to specific models or consistency rules [12, 16, 17]. Deriving these strategies can be time consuming and they are not generic/reusable. The second kind computes repairs for inconsistencies relying on solvers (e.g., SAT or CSP solvers [18, 19]). However, their downside is that not all relevant repair combinations may be explored, for example, through the use of heuristics for generating repair values.

Our approach introduces a third kind of reasoning. We argue that there are situations where exhaustive exploration of all possible repair values is unnecessary. The premise of our work lays in the fact that the inconsistent models already have the ingredients for their repairs. The idea in itself is not new. Using existing information for fixing inconsistencies has already shown to be effective in fixing bugs [20] or programming recommender systems. This paper, hence, explores the same idea in the context of the repairing of model inconsistencies. That is when an engineer evolves a model, intermittent inconsistencies can arise as a consequence of their initial changes. For example, a developer changing one diagram of a model may cause inconsistencies between this diagram and other diagrams that remain, as of yet, unchanged. Hence, these changes may be treated as correct albeit incomplete. Based on this assumption, we can use use information that already exists in the inconsistent models to repair the arising intermittent inconsistencies. Continuing on the example above, if we assume that message name and declaration were consistent originally and an engineer changed the message name such that it does not match any declaration then a repair is likely about adding or changing an existing declaration in the models such that it matches the new message name. Hence, the name of the message (a value) becomes a repair choice for the creation/renaming of a declaration.

The idea is also to use the information already present in the model for consistent change propagation. In consistent change propagation an inconsistency is introduced by an incomplete change performed by a developer. This change should have been propagated to other parts of the model, but was not carried out entirely. As our approach relies on the models' values, including those introduced by changes causing inconsistencies. Our approach, hence, can be used to compute repairs in the context of consistent change propagation.

By leveraging on values for inconsistent models, this in turn limits the number of possible repair choices. However, limiting the repair choices in this manner does not remedy the combinatorial effects of a sequence of changes. So, if there are n model elements and m concrete values for each model element to repair, then we need to explore m^n combinations. All of these must be explored to obtain a complete overview over all possible repairs even though only one is selected by the engineer at the end.

To address this scalability problem, our approach builds on the work of Reder et al. [21, 14] with the intent of understanding repair choices that have the same effect(s) and need not be considered in isolation. Our paper proposes:

1. Various generator functions to generate concrete values for changing model elements. The generator functions are designed independently from any abstract repair, inconsistency, or model and are thus reusable.
2. An algorithm to transform abstract repairs to concrete repairs using the concrete values from the generator functions. 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. A scalable exploration of combinations of repairs (i.e., possible m^n). Several key observations make this possible. When multiple values are com-

bined to form concrete repairs, one invalid value would result in many invalid combinations (i.e., incorrect concrete repairs). Even valid values on their own may contradict each other when combined. An example of an obvious invalid combination is, if the above message name was to be changed to a valid declaration name but at the same time the name of that declaration is changed as well. Hence, making the renamed message instantly inconsistent again. Moreover, if two values have the same implication onto the inconsistency then they can be grouped. Our approach combines groups of values if they have similar effects on the repair of inconsistencies. The concrete values corresponding to those valid boolean combinations (i.e., validating to true) are the only ones that are needed to compute concrete repairs.

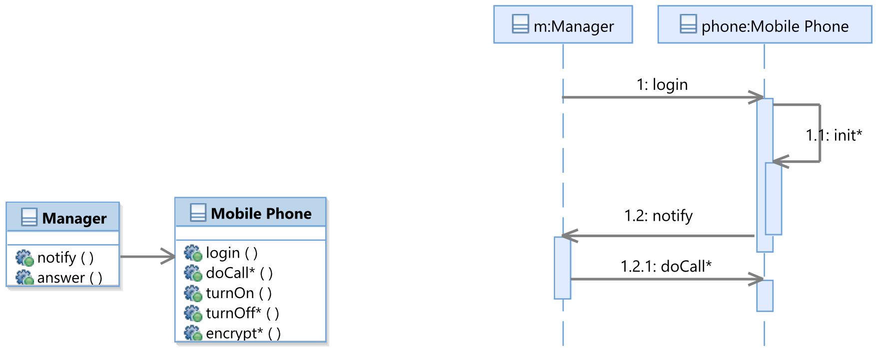
4. An evaluation of the usability of our approach by retrieving actual repairs from three versioned models. If an inconsistency in an earlier version of a model was not found in a later version then we identified the model changes that resolved these inconsistencies. We then compared these retrieved actual repairs with the ones our approach would generate. Among the inconsistencies in our versioned models, our approach identified all actual concrete repairs applied manually by the user. We also evaluated the scalability of our approach by systematically computing concrete repairs for inconsistencies found. For this part of the evaluation, we complemented the above three models with an additional seven, larger models to cover a wider range of model sizes. This evaluation showed that we were able to discover concrete repairs within a few milliseconds on average.

This paper is an extension of our previous work (Kretschmer et. al. [22]) consisting of contributions 1 and 2. Contributions 3 and 4 are the new extensions in this paper. Moreover, we added four additional models to the evaluation of the new contributions in this paper.

This paper is structured as follows. Section 1 presents the introduction and context for this paper. Section 2 introduces a running example showing the problems encountered when transforming abstract to concrete repairs. Section 3 gives definitions of the concepts used in this paper. Section 4 presents the of how to transform abstract to concrete repairs with the help of generator functions. Section 5 extends and improves this algorithm by introducing a value filtering mechanism before the transformation. Section 6 shows the evaluation and gives promising results. Section 7 presents the threats to validity. Section 8 discussed the related work. Section 9 summarizes this paper and gives ideas on future work.

2. Running Example

To illustrate our approach, we use a motivating example of a secure mobile phone environment (e.g., for managers of companies). Figure 1 depicts example snippets of two different UML diagram types of this system: a class diagram and a sequence diagram.



(a) Class diagram

(b) Sequence diagram

Figure 1: UML model snippets of secure mobile system

The class `Manager` (Figure 1a) initiates the process of performing a secure call to another phone. Class `Mobile Phone` handles the user interaction, e.g., receiving user input and ensures a secure workflow while being active. The sequence diagram in Figure 1b describes the workflow of a manager performing a call by calling operation `login` on an instance of class `Mobile Phone`, which then enables security by calling operation `init*` and notifies the manager `m` via operation `notify`. Manager `m` then performs a secure call via calling operation `doCall*`.

The models shown in Figure 1 are extended with a *UML Profile* to fit the domain of a mobile phone. The profile adds the *isSecure* attribute to messages and operations. If the *isSecure* attribute is enabled (i.e. set to true) the corresponding operation or message has an appended asterisk(*) in its name.

For the model to be correct, a secured message from a lifeline must have a secured corresponding operation in its corresponding class. This correctness criteria can be defined as a consistency rule, as shown in Listing 1. We use the *Object Constraint Language (OCL)* [23], a declarative language based on first order logic, to define *consistency rules* (CRs) for UML models [24, 14, 12, 19]. *Consistency rules* define specific constraints that must hold in software models. These constraints express relations among model elements that can range from well-formedness conditions, non-functional properties such as maintainability or usability [25, 26], or domain-specific rules such as the above one imposed by the security UML profile. Listing 1 shows one example of a CR specified with OCL. A consistency rule defines the type of model element it applies to as its context. For example, this CR applies to lifelines which means that every lifeline has to satisfy its condition.

```
Consistency Rule (CR1)
context: Lifeline
self.messages->forAll(m: self.class.operations->exists(o: o.name
     = m.name and o.isSecure implies m.isSecure = true))
```

```
//Every message in the sequence diagram has an operation in the
    corresponding class. And if the message is secure then the
    operation has to be secure as well.
```

Listing 1: OCL consistency rules for the models in Figure 1

The condition of this CR1 is to ensure that every message in the sequence diagram has an operation in the corresponding class. And if the message is secure then the operation has to be secure as well. Now imagine that an engineer renamed operation `init*` from class `Mobile Phone` to `encrypt*`, but forgot to also rename message `init*` in the sequence diagram. Doing so violated the above consistency rule and the following inconsistency occurs:

`I` **Violation of CR1** Message `init*` has no corresponding secure operation in class `Mobile Phone`.

This inconsistency occurs because the two diagrams are interconnected and changing one affects the other. To resolve this inconsistency, the developer needs to make another change – for example, to rename message init* in the sequence diagram. We may think of the inconsistency the developer caused with the first change is an intermittent inconsistency of a correct, albeit incomplete larger change (involving multiple model elements of different diagrams).

If there are multiple, alternative changes that repair an inconsistency then we speak of repair choices. The inconsistency above has at least one repair choice: To change the name of the `init*` message in the sequence diagram. This represents an abstract repair choice, since it does not reveal to which name (value) the message should be renamed to. To provide a concrete repair choice for this inconsistency we need concrete values. In this work, those values are provided through *generator functions* [22]. Our approach uses those functions to compute possible values for model elements or their attributes such as lifelines, names for messages, operations, etc. To rename message `init*`, we thus need to find values that are useful, meaningful names (regardless of whether they are able to repair the inconsistency). As an example, consider a value generator returning values for message `init*`. For simplicity, let us take names from the class `Mobile Phone` which the lifeline `phone` instantiates. These values are: {login, doCall*, turnOn, turnOff*, encrypt*}. All method names are valid message names but are all of them valid concrete repair choices?

Consider again the inconsistency `I`. On closer inspection, we find two conditions in the CR in Listing 1 which are combined through a logical and. Both conditions are violated ("o.name = m.name" and "o.isSecure implies m.isSecure = true"). All provided values from the above value generator function repair the first condition, but only some of them repair the second condition. Thus, only a subset of the values are able to fix the inconsistency (i.e., {doCall*, turnOff*, encrypt*}). This is a simple example of how values can be useful for one condition (o.name = m.name) of an inconsistency but not for another (o.isSecure implies m.isSecure = true). Among all the possible values, we thus need to find the ones which actually do repair the inconsistency.

Unfortunately, repairs get even more complicated in cases where multiple model elements are involved. In this situation, combinations of their respective, possible values have to be considered to repair the inconsistency. This leads to an exponential number of value combinations.

The combination of values is exponential with: $(\#values^{\#modelElements})$. Exploring these combinations is made even more complicated, because valid values for one model element might be invalid when combined with another valid value from a different model element. As observed by Reder et al. and our own previous work [24, 22], inconsistencies often have multiple model elements which need to be repaired, and therefore an efficient way of exploring value combinations is needed.

This paper proposes a novel approach for computing concrete repairs in a scalable manner. Our approach identifies valid values and then efficiently combines them into concrete repairs.

3. Background

This section provides definitions and examples of relevant terms used in the paper. Our terminology differs substantially from our previous work (see [21, 27, 24]), which we adopted and extended as follows:

3.1. Definitions

Definition 1. ***Model***. *A* <u>*model*</u> $\mathbb{M}$ *consists of* <u>*model elements*</u> $(e \in \mathbb{M})$ *where model elements can have* <u>*properties*</u> *p. A property of a model element is referred to by element dot (.) property name, e.g. "encrypt*.name". A diagram is simply a subset of model elements from the model. We also define the set of all possible models as* $\mathbb{M}_\Sigma$ *where* $\mathbb{M} \in \mathbb{M}_\Sigma$*.*

Examples for model elements are classes, associations, messages, etc. Examples for properties are a name of a class, the multiplicity of an association, the lifeline of a message.

Definition 2. ***Consistency Rule***. *A consistency rule is a condition defined for a context. The condition itself is a hierarchically ordered (tree-based) set of expressions, where the root expression corresponds to the condition as a whole and its subexpressions correspond to parts of the condition. An expression identifies an operation, has a single parent and one or more children and values to be validated.*

For example, CR1 from Section 2 defines a condition that every lifeline has to satisfy. Recall that the above CR has two parts connected by a logical `and`. The `and` expression has children: `o.name = m.name` and `o.isSecure implies m.isSecure = true`, where `o.name = m.name` is an equals subexpression with two leaf expression `o.name and m.name`. Typically, leaf expressions either access model elements or constants.

Definition 3. ***Expression***. *An expression is a part of a condition. Each expression has one or more children, zero to one parents, and an operation over the children.*

$$e := \langle op, children, parent, se2v \rangle$$

We define the following special kinds of expressions:

- e_r *The root expression, which is expected to validate to true (if not, there is an inconsistency).*
- e_p *The property call expression (without children) provides model values (e.g., the name of operation turnOn).*
- e_c *The constant expression, which returns a constant value.*
- e_b *The boolean expression, which performs boolean operations (e.g., NOT, AND, etc.) and returns true or false based on its validation.*
- e_v *The value comparison expression, which is used to determine if a value is suitable for fixing an inconsistency.*

For example, the `and` expression in CR1 computes the logical `and` of its two child expressions. Obviously, these two child expressions must validate to boolean results for well-formedness.

Definition 4. ***Validation Tree***. *A consistency rule validated on a specific model element is a validation. A validation tree mirrors the tree-structure of the consistency rule condition. However, in case of repetitions (e.g., forAll quantifier above) their (sub)tree-structures repeat for every iteration. Hence, the validation tree is an exact log of each operation computed during the validation of a condition.*

For example, there are two lifelines in Figure 1b: `m:Manager` and `phone:MobilePhone`. Hence, there are two validations, one for each lifeline. Each validation checks if a consistency rule's condition validates to true for its given context. This can be done recursively for every expression/subexpression of a condition. The root expression of a condition is expected to validate to true, however, as earlier work has shown, this expectation may change with subexpressions (e.g., because of negations [24]). As an example, Figure 2 shows a validation tree for CR. This validation tree will be explained in detail in the next section.

Definition 5. ***Scope Element***. *A scope element is a model element and its properties (e.p) accessed during the validation of a consistency rule. A set of scope elements is called a scope. The scope is derived from the various property call expressions of the validation tree.*

Definition 6. ***Cause***. *A* $cause_i$ *of an inconsistency i is all the expressions whose validation caused the root expression (the condition) to validate to false. Only these expressions caused the inconsistency and only these expressions need repairing. A cause is a subset of the scope.*

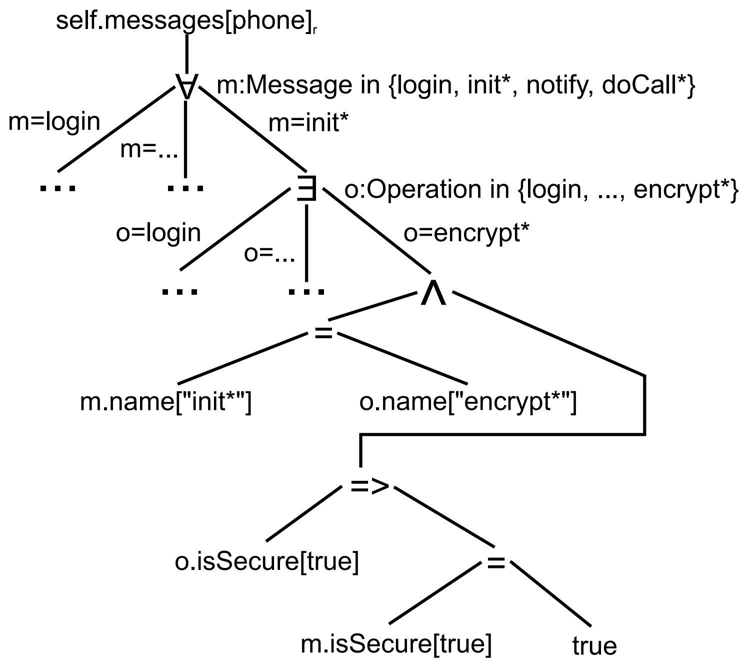


Figure 2: A Validation Tree for CR

Definition 7. ***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 identifies the model element (`e`) and the property (`p`) it applies to (that it will change), the type of operation (`op`), 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 operations are possible: $\oplus$ adds a value to a property that is a collection, $\ominus$ deletes a value from a a collection and $\odot$ modifies the value of a property. In addition there are the constraining operations: $\neq$, $<$, $>$, where respectively a property has to be different from the `value`, less than the `value`, or greater than the `value`. $\mathcal{P}(x)$ is the power set of x. Please note that the definition of an repair action's operation is not the same as the operation of an expression.*

$$ra := \langle e.p, op, v\rangle, op \in \{\oplus, \ominus, \odot, \neq, <, >\},$$
$$v \in \mathcal{P}(\mathbb{V}) \cup \mathcal{P}(\mathbb{M})$$
$$execute : \langle \mathbb{M}_\Sigma, \mathbb{RA}\rangle \to \{\mathbb{M} \in \mathbb{M}_\Sigma, \langle e.p, op, v\rangle\} \mapsto$$
$$x \in \mathbb{M}_\Sigma | x = (\mathbb{M} \setminus e) \cup e' | \begin{cases} e'.p = e.p \cup v & \textit{if op is } \oplus \\ e'.p = e.p \setminus v & \textit{if op is } \ominus \\ e'.p = v & \textit{if op is } \odot \textit{ and v is not } \varnothing \\ e' = e & \textit{if op is } \neq \textit{ or } < \textit{ or } > \\ e' = e & \textit{if v is } \varnothing \end{cases}$$

Furthermore we define the function execute, which performs a given repair action by applying the above model changes to the model. This function maps from a tuple of repair action and model ($\langle \mathbb{M}_\Sigma, \mathbb{RA} \rangle$), where $\mathbb{RA}$ is the set of all possible repair actions and $\mathbb{M}_\Sigma$ all possible models) 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 operation op ($\oplus$, $\ominus$, $\odot$). If no value (v is $\varnothing$) has been provided or the operation is different ($\neq$), greater than ($>$) or less than ($<$) nothing changes in the model ($e' = e$).

Definition 8. ***Abstract Repair Action***. *An abstract repair action is a repair action, without values ($\varnothing$). 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 operation is either $\neq$, $>$ or $<$. This function returns a boolean value (B), where* `true` *is abstract and* `false` *is not abstract.*

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

As an example the inconsistency `I` discussed in Section 2 can be fixed by changing the name of message `init*`. Expressed as an abstract repair action this leads to: $\langle init * .name, =, \varnothing \rangle$. Note that this abstract repair action is a hint and is not automatically executable yet, because we do not have a value for `init*.name`.

Definition 9. ***Concrete Repair Action***. *A concrete repair action is 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 `I` (from Section 2) we have to execute the concrete repair action: $cra = \langle init * .name, =, "encrypt*" \rangle$ that renames message `init*` to `encrypt*`. After cra has been executed message `init*` 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 10. ***Repair***. *A repair is a non empty collection of repair actions (ra) that resolve a cause of an inconsistency (i), where $\mathbb{I}$ is the set of all possible inconsistencies and $\mathbb{RA}_i$ the set of all possible repair actions for i. This set may contain abstract and/or concrete repair actions. If a repair action is concrete it also eliminates its cause by execution ($execute(x) \diamond cause_i$).*

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

We speak of an abstract repair if 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 speak of a concrete repair if all repair actions are concrete ($ra \subseteq \mathbb{RA}_i | (\forall x \in \mathbb{RA} | \neg isAbstract(x))$). 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 $\langle I, \{\langle init{*}.name, =, \varnothing\rangle\}\rangle$ for the the inconsistency `I` can be turned into a concrete repair by changing the message's name to a specific name, i.e., $\langle I, \{\langle init * .name, =, "encrypt * "\rangle\}\rangle$.

Definition 11. ***Generator Function****. We define a generator function which maps the tuple set of all scope elements and model ($\langle \mathbb{SE}, \mathbb{M}\rangle$, where $\mathbb{SE}$ is the set of all possible scope elements) to multiple model elements and their property values ($\mathcal{P}(\mathbb{M}) \cup \mathcal{P}(\mathbb{V})$, where $\mathcal{P}$ is the power set).*

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

As an example, let us consider the abstract repair action from the previous definition (Definition 10). 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: $gf(\langle init * .classes.operations\rangle, \mathbb{M})$, which returns all operations from the model ($\mathbb{M}$): `login`, `doCall*`, `turnOn`, `turnOff` and `encrypt*` from class `Mobile Phone`, and `notify` and `answer` from class `Manager`. A more optimized way to return operations would be to return only `Mobile Phone`'s operations, because instances from class `Mobile Phone` cannot call operations from class `Manager`. However, this would limit the versatility of this generator function to only inconsistencies involving class `Mobile Phone`. 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`.

3.2. Consistency Checking

Consistency checking is a well-covered topic in literature. In this section, we explain one such approach on a simple example. To illustrate the consistency checking mechanism we use a simple consistency rule CR2.

`context: Message self.class.operations->exists(o:self.name = o.name)` CR2 checks if a messages has an operation with the same name in its correpsonding class.

Figure 2 shows the instantiation of this consistency rule for the message `init*`. The root expression represents the model element, for which the consistency rule is instantiated `self[init*:Message]`. The next expression is an `exists expression` ($\exists$) where at least one of its children has to fulfill the condition defined in CR2 (`self.name = o.name`).

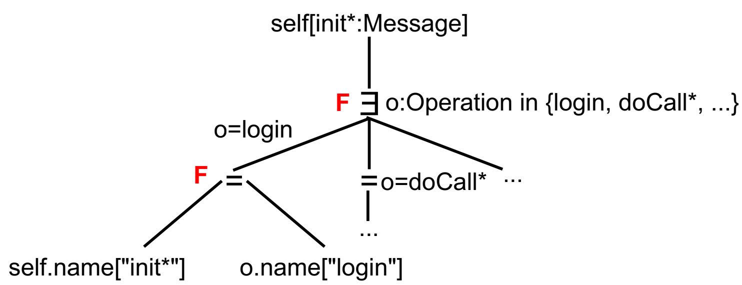


Figure 3: A Simple Validation Tree for CR2

At the `exists expression` we create one subtree for every operation in class `Mobile Phone`. For simplicity we only show the subtree for operation `login`. Please note that the subtrees for the other operations have the same subtrees, except with different names for the message and operation. The subtree for operation `login` represents an equals expression (=) which compares the values returned by their children for equality. The comparison is between the message's name `init*` and the operation's name `login`.

The scope elements of those expressions in the validation tree in Figure 3 are all operations from class `Mobile Phone`: `[login:Operation].name`, `[doCall*:Operation].name`, `[turnOn:Operation].name`, `[turnOff*:Operation].name`, `[encrypt*:Operation].name`, Message `init*` `[init*:Message].name`, `Mobile Phone:Class` and `phone:Lifeline`. Together those eight scope elements form the scope for CR2.

In the validation tree, the root expression is expected to validate to `true` (i.e., consistent) and so its children expressions. For example, the $\exists$ and $=$ expressions in Figure 3 are also expected to validate to `true`. If the root expression validates to `false`, then we detect an inconsistency. To compute the validation result of a validation tree, we start from the leafs (bottom) and start computing the validation result of the subexpressions (parent nodes) and continue this process until the root expression.

In Figure 3, since the name `"init*"` is unequal to `"login"` the subtrees' validation results is `false` (denoted with a red `F`). At the exists expression ($\exists$) there has to exist at least one subtree in its children with the validated result `true`, but in this example the subtree validates to `false`, because there is no operation with the name `init*`. Thus, the exists expression validates to `false`, which is the same validation result of the root expression. This is how our approach detects an inconsistency I2 in the model.

3.3. Repair Generation

In the previous subsection, we explained how inconsistecy I2 is detected in our model. In this section we introduce, how we can repair I2. Before to compute the repairs, we first need to identify the cause of an inconsistency.

Take again the inconsistency from CR2. To identify the *cause* for this inconsistency we take the *scope* from the previous section

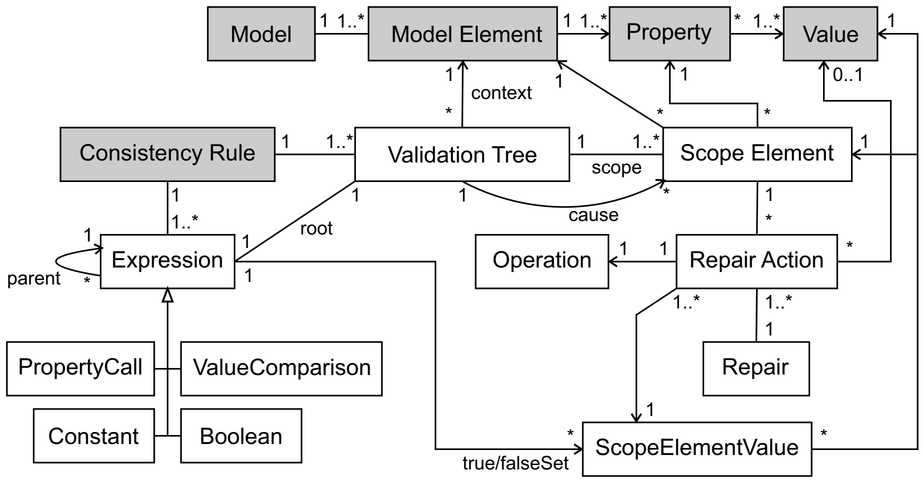


Figure 4: UML class diagram for the definitions

scope = {[login:Operation].name, [doCall*:Operation].name, [turnOn:Operation].name, [turnOff*:Operation].name, [encrypt*:Operation].name, [init*:Message].name, Mobile Phone:Class, phone:Lifeline}. We then check for every scope element, if it is part of a violated expression, i.e., validation result $\neq$ expected result (validation result of those expressions is false in Figure 3), we add it to the *cause* of the inconsistency. The *cause* of our example shown in Figure 3 is *cause* = {[login:Operation].name, [doCall*:Operation].name, [turnOn:Operation].name, [turnOff*:Operation].name, [encrypt*:Operation].name, [init*:Message].name}.

To generate repair actions we iterate over every scope element in the *cause* and look for every violated expression were the scope element is used. We then generate a repair action so that the direct violated expression is validated. For instance, the scope element [login:Operation].name is used in the violated = expression in the right hand side of the validation tree shown in Figure 3. Based on this expression we know that the equals condition = is not fulfilled, since name login is unequal to name init*. Therefore, the repair action for the scope element init* would be to rename it to login $\langle init*.name, \odot, "login" \rangle$, which leads to the repair $\langle I2, \{\langle init*.name, \odot, "login" \rangle\}\rangle$ that fixes the inconsistency when executed.

However, changing the name of message init* to login is not the only valid repair for fixing I2. Based on the = expression in Figure 3, another repair can be generated for the scope element login which is to rename operation login to init* $\langle I2, \{\langle login.name, \odot, "init*" \rangle\}\rangle$. This repair also fixes I2.

3.4. Relations of the Defined Terms

For a better understanding of how our defined terms are related to each other we given an overview in Figure 4. This figure shows a UML class diagram of

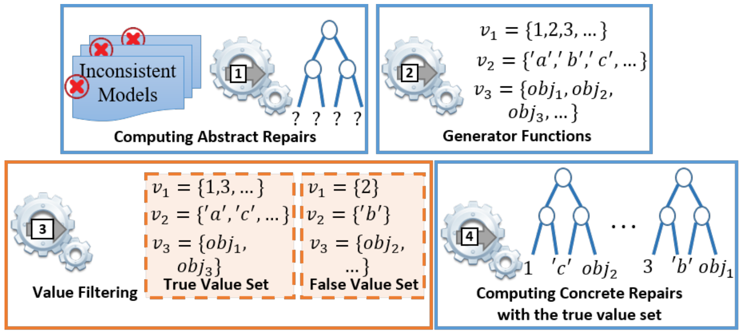


Figure 5: Overall approach.

the definitions from the previous section depicted as classes (without attributes) and their associations. For instance a `Validation Tree` has exactly one `Model Element` as `context` element, depicted as association from `Validation Tree` to `Model Element` with the name `context`. In turn, one `Model Element` can be used by multiple `Validation Trees` as their `context` element.

The boxes highlighted in grey are the foundation of our approach and are provided by an engineer.

4. Approach

This section presents our algorithm to convert abstract repairs to concrete repairs. It is important to note that the efficient computation of abstract repairs is essentially a solved problem. In providing an algorithm for efficiently computing concrete repairs, we essentially provide values to abstract repairs. Since there are often multiple useful values or value combinations, our algorithm typically provides a set of concrete repairs for any given abstract repair.

4.1. Overview

This section describes our approach and explains the main ideas of our abstract to concrete repair algorithm.

Figure 5 shows the basic workflow, which consists of the following four stages: The *first stage* (denoted with `1`) checks a model for `inconsistencies` based on the provided consistency rules. We build on the work of Reder et al. [24, 21] to compute abstract repairs for a given inconsistency. Those inconsistencies are converted into validation trees. Note that trees for abstract repairs do not yet have concrete values as leaves (e.g., concrete values to repair an inconsistency). This means that abstract repairs identify which model elements should change but not how to change them. The main purpose of validation trees is to match expected validation results with actual validation results for computing the cause(s) of an inconsistency and subsequently the repair tree. If there is a mismatch between the expected and the actual result then we know which model element and its corresponding property needs to be repaired.

The *second stage* (2) then applies `generator functions` to retrieve value sets (e.g., v_1, v_2, v_3) for model elements and corresponding properties identified by the abstract repair. Those values are then explored to test whether they are able to fix the corresponding inconsistency - in part or full. Incorrect values are removed from the value sets.

The *third stage* (3) explores repairs by starting at the leaf expressions of the validation tree from stage 1 and applies the value sets from stage 3. This is done by checking the values with the conditions of each expression and putting them in a `true` or in a `false` set recursively until the root expression is reached. This stage also provides two sets, one for values which fix the inconsistency (referred to as `true` set) and one with values that do not fix the inconsistency (referred to as `false set`). The elements of those sets are referred to as `ScopeElementValue`. The values in the `true` set are guaranteed to fix the inconsistency, since we checked them with the conditions from the validation tree, i.e., the consistency rule itself. Finally, the values in the `true` set is used to form concrete repairs that can be executed automatically on the inconsistent model.

The fourth stage (4) explores combinations of values from `stage three` using again the tree structure from `stage one`. The tree structure reveals how the values relate and whether or not the values resolve the inconsistency in part or in full. This final stage leads to combinatorial exploration to identify which value combinations do repair the inconsistency.

4.2. Generator Functions

Central to our approach are the generator functions, which provide values model element properties. These values are derived from the model and not invented. Essentially, we presume that engineers already provided the information needed for repairs through previous changes. The generator functions thus mine the model for possible values. Obviously values need to respect type compatibility (i.e., if a name is to be repaired then strings are needed).

The following identifies 42 generator functions that follow one of two types (21 type 1, 21 type 2). Engineers can easily add additional generator functions if needed, or modify existing ones. These two types of generator functions are discussed next.

Type 1: All values of a specific type: This type of generator functions returns all values of a specific type, for example, all existing strings or operations from the model. 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 having to adapt or change them. The disadvantage of this type of generator functions is that it tends to lead to a large number of values to validate. For example, consider again that a name needs repairing. A name is of type string and such a string might already exist elsewhere (e.g., in another diagram that was changed earlier). However, there are typically thousands of such strings in medium to larger size models; even if we restrict the time scale to changes within the last days or weeks, there are still many string. Still, these generator functions can provide useful values.

Algorithm 1 Generator function for all strings within a model

function GETALLSTRINGS($m \in \mathbb{M}, p \in \mathbb{M}, \mathbb{M}$) : $\mathcal{P}(\mathbb{M}) \cup \mathcal{P}(\mathbb{V})$ ▷ m is a model element, p the corresponding property and $\mathbb{M}$ the model
    $values \leftarrow \varnothing$
    if p isType 'string' **then**
        $values \leftarrow \mathbb{M}$.getElementsOfType('string') ▷ Utility function
    end if
    return *values*
end function

Algorithm 1 shows the implementation of a generator function of type 1 for a property of type 'string'. The generator function is registered 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 strings.

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 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 much smaller subset of values than type 1 generator functions. For instance all class names are a subset of all strings, which leads to a reduced amount of value validations. The disadvantage of this type of generator functions is that in large software models this still can lead to many values, for instance there can be thousands of classes in a large software project.

4.3. Transforming abstract repairs to concrete repairs

4.3.1. Main Algorithm

This section explains our main algorithm on how to transform abstract to concrete repairs.

Algorithm 2 shows the pseudo code for our approach. The algorithm is divided 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` for which we want all abstract repairs to be transformed, 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 6–14 and Line 35). 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 for the generator functions.

Phase C: Iteration over all scope elements (Lines 16–34). This phase iterates over all previously acquired scope elements and retrieves all values from

Algorithm 2 Abstract repair to concrete repair transformation algorithm

```
function ConvertAbstractRepairsForM(i ∈ 𝕀, m ∈ 𝕄, gf ⊆ 𝔾𝔽, 𝕄) : {x|x ∈ ℝ𝔸}    ▷ i is an inconsistency, m a model element contained in at least one abstract repair, and gf a set of generator functions, and 𝕄 a model
    concreteRepairs ← ∅    ▷ Contains all converted repairs in the end
    ▷ Contains all abstract repairs from i containing m
    repairs ← {x ∈ ℝ|i = x.i ∧ (∃y ∈ x.ra|y.e = m.e ∧ y.p = m.p) ∧ isAbstract(x)}
    ▷ Iterate over all relevant abstract repairs in i to convert them to concrete repairs
    for all r ∈ repairs do
        ▷ Only convert abstract repair actions
        repairActions ← {x ∈ r.ra|isAbstract(x)}
        scopeElements ← ∅    ▷ Set of all relevant scope elements
        se2v ← ∅    ▷ Map for scope elements to values from their gf
        ▷ Collect all scope elements from repairActions to get model values
        for all ra ∈ repairActions do
            scopeElements ← scopeElements ∪ ⟨ra.e⟩
        end for
        ▷ Prepare values for every scope element depending on property type
        for all se ∈ scopeElements do
            root ← validateValues(i.root, gf)
            values ← root.true[se]
            if isMultiValue(ra.e.p) then    ▷ Property is a collection
                actualSize ← |ra.e.p|    ▷ Current size of the collection
                ▷ Needed size of r.p to resolve i
                requiredSize ← getRequiredSize(ra.e.p, i)
                if actualSize < requiredSize then
                    ▷ Collection has not enough values, calculate difference
                    diff ← requiredSize − actualSize
                    ▷ Generate value combinations
                    values ← getCombinations(values, diff)
                else if actualSize > requiredSize then
                    ▷ Collection has too many values
                    diff ← actualSize − requiredSize
                    values ← getCombinations(ra.e.p, diff)
                end if
            end if
        end for
        se2v ← se2v ∪ ⟨se, values⟩    ▷ Add scope element with its values
    end for
    combinations ← combineScopeElements(se2v)
    ▷ Convert combinations into concrete repairs
    concreteRepairs ← getRepairs(combinations, i)
    return validateRepairs(concreteRepairs, i)
end function
```

the generator function by calling `getValues()`. If a property is a collection then we enter phase D.

Phase D: Collection type properties (Lines 19–32). 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 the abstract repair action $\langle phone.messages- > size(), \geq, 2\rangle$, which states that phone has to have at least two message calls (maybe one to turn the device on, and another one to turn it off). Furthermore in this example, `phone` has no message call from another class. To convert this abstract repair a generator function returns the value set of all operations from class `Mobile Phone`: `login`, `doCall*`, `turnOn`, `turnOff*` and `encrypt*`. Now $\binom{|values|}{2} = 10$ combinations are generated: `login`, `doCall*`; `login`, `turnOn`; `login`, `turnOff*`, etc.

Phase E: Combination of scope elements , validation (Lines 37–40). This phase of our algorithm performs the combination of scope elements (i.e., several scope elements in an abstract repair with the help of the Cartesian product) and validates the resulting concrete repairs. First, the Cartesian product is generated for all scope elements involved in the current abstract repair.

Finally, all the combinations have to be checked if they are indeed able to fix `I` [14]. However, this process is very time consuming in cases where generator functions return many values and multiple model elements have to be repaired at the same time. In this case abstract repairs might not be able to be transformed into concrete repairs (i.e., too many combinations). In the next Section we introduce a novel mechanism to determine which values provided by any generator function are able to fix an inconsistency before combining them in `Phase E`. This means that if the values have to be combined for concrete repairs those combinations do not have to be checked for their correctness any more.

4.3.2. Helper Functions

This section gives and explanation of the used helper functions in Algorithm 2. For simplicity, we will give the signature of those functions and one example to illustrate the usage.

Allgortihm 3 shows the signature of the `getRequiredSize` function. The inputs are a model element `me` with its property `p` and an inconsistency `i`. This function assumes that property `p` is a collection, i.e. it can store multiple values, and returns the required size of that collection to fix `i`. Consider the following

Algorithm 3 Get required size

function GETREQUIREDSIZE($me.p \in \mathbb{M}, i \in \mathbb{I}$) : integer
    ▷ Only signature given
end function

Algorithm 4 Combine scope elements

function COMBINESCOPEELEMENTS($se2v \leftarrow \{x|x \leftarrow \langle se \in \mathbb{SE}, values \leftarrow \{y|y \in$ "arbitrary value"$\}\rangle\}$) : $\{x|x \leftarrow \{y|y \in \mathbb{RA}\}\}$
    ▷ Only signature given
end function

consistency rule: `context: Class self.operations->size() < 3` This rule checks if a class has less than three operations. The function `getRequiredSize` takes now a class model element and its operation property and gets the required size from `i`. The inconsistency `i` contains the validation tree with the necessary amount of elements in the collection (contained in a less than expression).

Algorith 4 shows the signature of the function `combineScopeElements`. Here `se2v` is the input set where every element represents a tuple of one scope element `se` and its corresponding value set `values`. Such a tuple can be interpreted as a repair action, since it contains a scope element and a value. This input set is then converted to a set of repair actions (following the Definition 7).

As an example consider the following input set $\{\langle a, \{1,2\}\rangle, \langle b, \{X, Y, Z\}\rangle\}$ where `a` and `b` are scope elements and `1, 2, X, Y, Z` are valid values from the model. This input set is then converted to the following result with the help of the Cartesian product: $\{\{\langle a, \odot, 1\rangle, \langle b, \odot, X\rangle\}, \{\langle a, \odot, 1\rangle, \langle b, \odot, Y\rangle\}, \ldots, \{\langle a, \odot, 2\rangle, \langle b, \odot, Z\rangle\}\}$

Algorithm 5 shows the signature of the function `getRepairs`. It converts the set of repair actions to a repair by appending the inconsistency `i` to the corresponding repair actions.

For example, consider inconsistency `i` and the following repair action: $\{\{\langle a, \odot, 1\rangle, \langle b, \odot, X\rangle\}, \{\langle a, \odot, 1\rangle, \langle b, \odot, Y\rangle\}, \ldots, \{\langle a, \odot, 2\rangle, \langle b, \odot, Z\rangle\}\}$

This is converted to the following repairs: $\{\langle i, \{\langle a, \odot, 1\rangle, \langle b, \odot, X\rangle\}\rangle, \langle i, \{\langle a, \odot, 1\rangle, \langle b, \odot, Y\rangle\}\rangle, \ldots, \langle i, \{\langle a, \odot, 2\rangle, \langle b, \odot, Z\rangle\}\rangle\}$

4.4. Value Filtering

This section presents our automated algorithms to instantly discover valid values for all model elements that could be changed (i.e., scope elements) to repair a given inconsistency. The valid values can then be immediately used to create concrete repairs while filtering invalid values. First we give a general overview, then we describe how we divide the initial provided values into a set of valid and invalid values with the help of the validation tree.

Algorithm 5 Get repairs

function GETREPAIRS($\{x|x \leftarrow \{y|y\{x|x \ \leftarrow \{y|y \in \mathbb{RA}\}, i \in \mathbb{I}\}) : \{x|x \in \mathbb{R}\}$
  ▷ Only signature given
end function

4.4.1. Value validation

In this section, we describe in detail the first step from stage three of our value filtering mechanism. In this step, we recursively iterate over the expressions in the validation tree, and separate the provided values for the scope elements into a `true` set and a `false` set, i.e. we abstract from concrete values to boolean value sets true and false. The `true` set contains for every scope element the values which validate its corresponding expression (e.g., $\wedge$, $=$, $>$, ...) to true. The `false` set contains for every scope element the values which validated its corresponding expression to false. This step is crucial because it automatically groups concrete values based on their effects on the causes of the inconsistency into `true` and `false` sets. Algorithm 6 shows the pseudo-code of our value validation. For a better understanding, this algorithm is split into three phases.

Phase 1: (Line 2-10) This phase checks, if the current expression is not able to compare two values (e.g., AND, OR, IMPLIES expressions, etc). In this case `validateValues` is called recursively on every `child` of the current expression `e` until a comparison expression is reached (those are the parent nodes of the leafs). After a child of `e` has been validated, the `true` and `false` value sets are added to the parent expression `e`. At the end of phase 1 we combine the collected value sets, and return them to the parent expression (this step is explained in detail in the next Section 4.4.2).

Phase 2: (Line 11-21 and Line 29) This phase iterates over every scope element `se` in the current expression `e`, and over every value for `se` provided in the value set `se2v`. First, it retrieves the values of the corresponding scope element (i.e., `getValues(se, se2v)`). Then `e` is checked if it is able to compare two values while checking its type (e.g., equals(3,4) would return false). Please note that in this phase all expression types are considered that are able to compare two values (in contrast to phase 1).

Phase 3: (Line 23-27 and Line 30) This phase adds the checked values from phase 2 to the corresponding `true` or `false` value set of `e`, based on the previous validation result. After iterating over all values and scope elements the value sets have to be combined according to expression specific conditions (i.e., combinations that lead to true for AND, OR, IMPLIES, etc).

As an example for the first step from stage $\boxed{3}$ of our approach, consider the validation tree from Figure 6. It represents the instantiation of the inconsistency rule from Section 2, and the corresponding value sets `m.name`={login, doCall*, turnOn, turnOff*, encrypt*} (where `m` is a `Message`), and for the repair $\langle I, \{init * .name, =, \varnothing\}\rangle$. Please note that for simplicity we do not show the whole validation tree, but only the important parts.

Algorithm 6 Validate Values

```
function validateValues(expression e, se2v ← {x|x ← ⟨se ∈ 𝕊𝔼, {y|y ∈ "arbitrary value"}⟩}) : expression e
    if e is-not-a ValueComparsionExpression then
        ▷ Every other expression like and, or, ...
        for all child in e.children do
            expr = validateValues(child, se2v)
            e.true = e.true ∪ expr.true
            e.false = e.false ∪ expr.false
        end for
        return combineValues(e)
    end if
    for all se in e.se do
        values = getValues(se, se2v)
        for all value in values do
            if e is-a equals then
                result = equals(value, e.conditionValue)
            else if e is-a unequals then
                result = unequals(value, e.conditionValue)
            else if e is-a greaterThan then
                result = greaterThan(value, e.conditionValue)
            else if ... then
                ▷ only value comparison expressions (<, ≤, =, ...)
            end if
            if result then
                e.true = e.true ∪⟨ se, value ⟩
            else
                e.false = e.false ∪⟨ se, value ⟩
            end if
        end for
    end for
    return combineValues(e)
end function
```

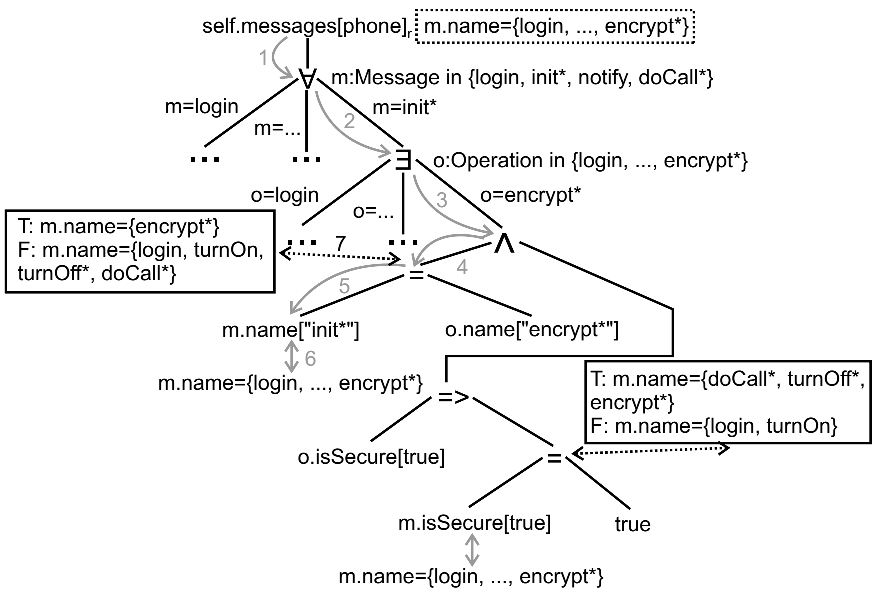


Figure 6: Example of value validation

We start at the root expression e_r, and see in phase 1 that e_r is not able to compare two values (by definition). This means that we have to iterate over all children, which is the for all expression (∀) in this example (arrow denoted with 1 in Figure 6). In for all expressions we are not able to check any values, and therefore continues with the corresponding child expression. For simplicity we show only the iteration for the right hand side sub tree, which represents the validation for message `init*` (arrow denoted with 2, the other sub trees are for other messages). The next child is the exists expression (∃), which has child expressions for every operation in class `Mobile Phone`. Also, in the exists expression (∃) we cannot check for valid values and continue with the next child expression. Again for simplicity, we focus in Figure 6 on the operation `encrypt*` sub tree (denoted with 3). However, the same would be done for the other subtrees with their corresponding operations. We continue traversing to the left hand side sub tree to the equals expression (=, steps 4 and 5), since the AND expression will only be handled in the next stage.

Finally, when we reach the equals expression (arrow denoted with 5), we are able to check for valid values, thus we enter phase 2 of our algorithm. This expression has two scope elements `m.name` (the message's name) and `o.name` (the operation's name). To determine which scope element has to be exchanged during the value validation we look at the provided values set `se2v`, and see that we only have values for messages (arrow denoted with 6). Therefore, we keep the value for `o.name` (which is in this case the operation with name "encrypt*") and check every value from the provided value set `m.name`={login, doCall*, turnOn, turnOff*, encrypt*} for equality. After validating the values for the equals expression we end up with the following `true` and `false` sets (denoted with 7): $true = \{m.name = \{encrypt*\}\}$ and $false = \{m.name = \{login, turnOn, turnOff*, doCall*\}\}$

The same process is executed at the equals expression on the right hand side subtree where we check at the equals expression if the provided operations are indeed marked as secure. At this equals expression we get additional `true` and `false` sets: $true = \{m.name = \{doCall*, turnOff*, encrypt*\}\}$ and $false = \{m.name = \{login, turnOn\}\}$

How those `true` and `false` sets from the two subtrees are then combined in the function `combineValues(e)` (Line 30) is described in the next section.

4.4.2. Back propagation

This section describes in detail the second step from stage $\boxed{3}$ of our value filtering mechanism. It combines the value sets generated in the first step according to expression specific rules. After this step, the values (and possible value combinations) in the `true` set of the root expression contains only those values for scope elements that indeed fix the corresponding inconsistency.

Algorithm 7 shows the pseudo-code of our value combination algorithm which is called at the end of stage $\boxed{3}$ of our approach. For a prompt understanding we split this algorithm into three phases.

Phase 1: (Line 3-6) This phase checks if the provided expression is a negation (logical not), and assigns the `false` set of its child to its own `true` set, and the `true` set of its child to its own `false` set. This is done, because every value of the child which validates to false, now validates to true after the negation (i.e., swapping the `true` and `false` sets). Please note that a negation always has exactly one child.

Phase 2: (Line 7-20) This phase handles the combination of values of a conjunction (logical AND). A conjunction always has exactly two children where every child has its own `true` and `false` set. To validate an AND expression to true, the `true` sets from both children have to be combined. The false values of the current expression `e` is the mathematical unification of the children's `false` sets, e.g. $\{1, 2\} \cup \{A, B\} = \{1, 2, A, B\}$.

A special case occurs when those children have the same scope element in common. Then, only the values those two `true` sets for the scope element have in common (intersection $\cap$) are able to validate the AND expression to true. To this end, we iterate over every `ScopeElementValue` of scope elements from the true set (i.e., we only consider the scope elements, not the values) and check if those scope elements are the same. We calculate the intersection of those two value sets, and assign them to the `true` set of the current expression. We then add every value which was not part of the intersection (symmetric difference Δ) to the `false` set of the current expression. Finally, we remove the true values from se2, because they are now either partly in the `true` or `false` set of `e`.

Phase 3: (Line 21-31) Here we handle the rest of the logical expression like the OR, IMPLIES, forAll, etc. To simplify the value combination process, we translate every boolean expression into negations and conjunction equivalents. Table 1 shows some examples on how those operations are translated. For example, the forAll expression (as shown in Table 1) is applied to a collection with and expression `expr`. The expression `expr` is then applied for every element in the collection, where $expr_1$, $expr_2$, ... are the applications for the first, second, ... element in the collection. Those expressions are then concatenated with conjunctions $\wedge$. Please keep in mind that an expression can consist of multiple sub expressions. After phase 3, the back propagation ends at the root expression, which now contains valid values for scope elements in the `true` set and invalid values in the `false` set.

As an example for the back propagation, consider the validation tree shown in

Algorithm 7 Combine Values

```
function COMBINEVALUES(expression e) : expression
    children = e.children
    if e is-a negation then
        ▷ Negation has exactly one child
        e.false = children[0].true
        e.true = children[0].false
    else if e is-a conjunction then
        ▷ Conjunction has exactly two children
        c1 = children[0]
        c2 = children[1]
        e.true = c1.true ∪ c2.true
        e.false = c1.false ∪ c2.false
        for all se1, se2 in e.true do
            if se1 == se2 then
                e.true[se1] = (e.true[se1] ∩ e.true[se2])
                ▷ Symmetric difference
                e.false[se1] = e.false[se1] ∪ (e.true[se1] Δ e.true[se2])
                e.true = e.true \ e.true[se1]
            end if
        end for
    else if e is-a disjunction then
        ▷ a ∨ b treated as ¬(¬a ∧ ¬b)
        ...
    else if e is-a implication then
        ▷ a ⟹ b treated as ¬a ∨ b
        ...
    else if e is-a forall then
        ▷ Same as ∧ with n children, see Table 1
    else if e is-a exists then
        ▷ Treated as ¬∀
    end if
    return e
end function
```

Table 1: NAND Logic reduction

Expression	Equivalent
$a \vee b$	$\neg(\neg a \wedge \neg b)$
$a \implies b$	$\neg a \vee b$
$\forall(exp)$	$exp_1 \wedge exp_2 \wedge \ldots$
$\exists(exp)$	$\neg\forall(exp)$
$\ldots$	

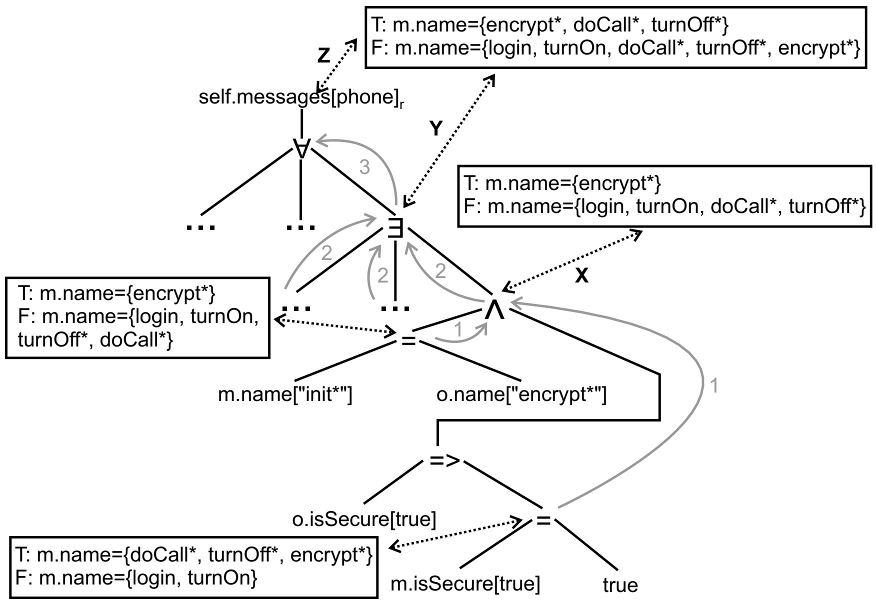


Figure 7: Example of value back propagation

Figure 7 after the value validation step shown in Figure 6. In both equals expressions we have true and false values for their corresponding scope element after the value validation phase. At those expression we do not have to combine values, since they are value comparison expressions. Also at the implication expression (=>) we do not need to perform an additional value combination, since the right hand side (`o.isSecure`) does not have a value set assigned to its scope element.

The AND expression now checks if the two child expressions do have a scope element in common (`gray` arrows denoted with 1). In this example, message `init*` from the left and right hand side subtree are the same, so the intersection set of the `true` sets is calculated and added to the `true` set of the `AND expression` (denoted with X). Since those two subtrees have only the name `encrypt*` in common this value is added to the `true` set and every other value is added to the `false` set.

The next step (denoted with three arrows labeled with 2) is the exists expression which will be handled as explained in Table 1. In this step all valid values from the sub trees `true` sets are added to the `true` set of the exists expression, which are `encrpyt*`, `doCall*` and `turnOff*`. Every message name from those subtrees are able to fix the inconsistency. Please note that the `false` set also contains the values from the `true` set. However, this is not a contradiction, since in every subtree two of the three secure operations `encrypt*`, `doCall*` and `turnOff*` are put into the `false` set (like shown in Figure 7 at the `exists expression`). We also must not remove the values contained in the `true` set from the `false` set since they might be important for subsequent expressions (e.g. a subsequent negation which swaps the `false` set with the `true` set).

As a last step (denoted with 3), the resulting `true` and `false` sets are investigated at the for all expression (∀). Here nothing has to be done, since the other subtrees (which investigate messages `login`, `notify` and `doCall*`) are already consistent, thus the `true` and `false` sets from the exists expression are propagated. At the `root` `expression` the `true` set of its child can be used for constructing concrete repairs directly (denoted with Z). An example for a concrete repair would be: $\langle I, \{\langle init{*}.name, =, "encrypt * " \rangle\}\rangle$, i.e., renaming message `init*` to `encrypt*`.

5. Evaluation

This section evaluates our abstract to concrete repair transformation, as well the new contribution of the value filtering mechanism. We do this by assessing the correctness, applicability and usefulness.

For the evaluation we applied 20 consistency rules to 10 UML models (consisting of class, sequence state machine diagrams, etc.) taken from three different sources: academia (VOD), industry (eBullition, MVC, Micro, DESI, Dice, oodt) and GitHub (Pro11, fullAdder, activityMngr) [28]. Examples for the applied consistency rules are: Parent Class should not have an Attribute referring to a Child Class, AssociationEnds must have unique Names within the Association, At most one AssociationEnd may be an Aggregation or Composition. However, during our evaluation we applied more consistency rules, which can be found in the provided evaluation data. The domains of the models range from control of a micro wave oven to a model view controller of software. Three of these models from GitHub have two versions each, where version one had inconsistencies that had been fixed in version two by a developer. This further allowed us to assess the quality of our approach and the relevance of our repairs, i.e. whether the manually applied repairs by the developers could be replicated by our approach. The model sizes ranged from 300 to 12000 model elements and the number of inconsistencies from 16 to 2000. Table 2 shows details, such as number of model elements, number of inconsistencies, number of abstract repairs for all models used in the evaluation. Note that our implementation has a compilation module integrated to check the syntactical correctness of the OCL consistency rules. Details about the dataset and consistency rules can be found on our companion web page [1].

In the following Sections we first evaluate our basic abstract to concrete transformation mechanism to measure how efficient it is. Then we evaluate our enhanced abstract repair to concrete repair transformation with the value filtering mechanism to measure its benefits.

5.1. Research Questions: Abstract to Concrete Repair Transformation

In this section we define three research questions to evaluate our abstract to concrete repair transformation approach.

RQ 1: Abstract to concrete transformation. How many abstract repairs can be converted to concrete repairs (e.g. abstract repairs with no concrete repair vs abstract repairs with at least one concrete repair) for every model?

RQ 2: Relevant concrete repairs. Are our generated repairs relevant for real world consistency fixing (e.g, are we able to find repairs a engineer would also have applied to the models manually)?

RQ 3: Relevant generator functions. Are our used generator functions on the one hand sufficient to find relevant concrete repairs, and on the other hand are they able to 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 engineer in

[1] https://figshare.com/s/a8038848c5ae1ec0faa6

Table 2: Model information

Model Name	#Model Elements	#Inconsistencies	#Abstract Repairs	Source
pro11	284	16	134	GitHub
fullAdder	992	37	203	GitHub
activity Manager	1185	51	270	GitHub
VOD	467	9	43	Academia
eBullition	1346	74	630	Industry
MVC	1410	71	554	Industry
Micro	2346	76	412	Industry
DESI	3600	276	1472	Industry
Dice	4485	207	1961	Industry
oodt	11655	2067	755	Industry

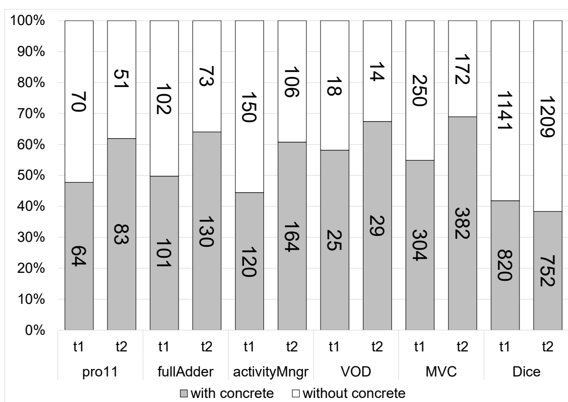


Figure 8: Abstract Transformation.

the end to easily select one desired concrete repair without iterating over a large list of repairs.

5.2. Results

RQ 1: We applied all consistency rules to every model, and applied our approach and generator functions to all abstract repairs (we get from [14]) 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 8 shows the percentage of abstract repairs that was 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.

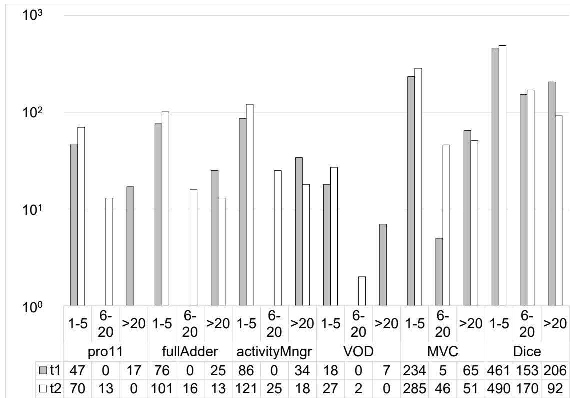


Figure 9: Concrete Repairs Count.

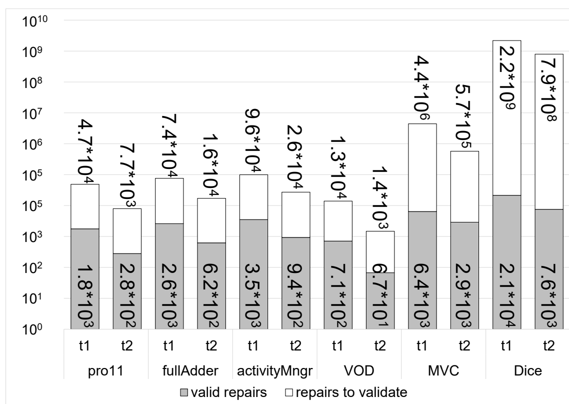


Figure 10: Validation Count.

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 engineer 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 2 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 engineer 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 engineer after generation.

Finally, although we could not transform all abstract repairs to concrete repairs, we were able to generate at least one concrete repair for every inconsistency i.e., there was at least one abstract repair per inconsistency which could be transformed to at least one concrete repair.

RQ 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.

RQ 3: Figure 9 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 9 we can also see that type 2 generator functions are able to convert 71 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 engineer 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 10 is due to the few abstract repairs for which more than 20 concert repairs were computed (e.g., 100 in some cases).

Figure 10 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 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 (this large amount is the result of the combination of multiple scope elements), which after validation, it 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.

As we have seen in this 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). Our value filtering mechanism deals with this particular limitation.

5.3. Research Questions: Value Filtering

In this section, we define three research questions (RQ) to evaluate our value filtering mechanism.

RQ 4: To what extent can our approach reduce the set of provided values? This aims to evaluate the scalability benefit that can be gained during the generation of value combinations for concrete repairs.

RQ 5: Does the `true` set of our approach lead to relevant concrete repairs? This aims to assess the quality of our approach whether it always proposes relevant values in the true set.

RQ 6: How fast does our approach separate the provided values? This aims to measure the benefit with respect to time scalability of our approach.

5.4. Results

We applied the abstract to concrete repair transformation with our value filter on all 10 models from Table 2. With the additional value filtering this allows us to run the evaluation on all 10 models. Furthermore, we used type 2 generator functions, since they provide an optimal trade-off between correct an incorrect values compared to type 1 generator functions (as can be seen in Figure 10).

RQ 4: We applied all consistency rules to every model, and performed our value discovery algorithms to every abstract repair (and therefore, to every inconsistency), and executed the corresponding concrete repairs. Figure 11 shows the amount of reduction for our value discovery approach (in percent and absolute) for every model. The numbers in the bars show the absolute amount of invalid value combinations of `false` sets in white and the absolute amount of valid value combinations of `true` sets in gray. On average we were able to reduce the amount of values by 83% of the theoretical maximum of value combinations. The theoretical maximum of value combinations per model has been determined by the formula $\sum_i^{|\mathbb{AR}_\mathbb{M}|} \prod_j^{|\mathbb{SE}_i|} |values_j|$, where $\sum_i^{|\mathbb{AR}_\mathbb{M}|}$ is the sum over all abstract repairs $\mathbb{AR}_\mathbb{M}$ for all inconsistencies of a specific model `M`, $\prod_j^{|\mathbb{SE}_i|}$ is the product over all scope elements from the abstract repair `i`, and $|values_j|$ is the amount of values the scope element `j` has. Imagine an abstract repair has three scope elements, and those scope elements have seven, eight and nine values. This leads to $7 * 8 * 9 = 504$ possible combinations of concrete repairs for this abstract repair. Then the sum of all those products for every abstract repair is calculated. The amount

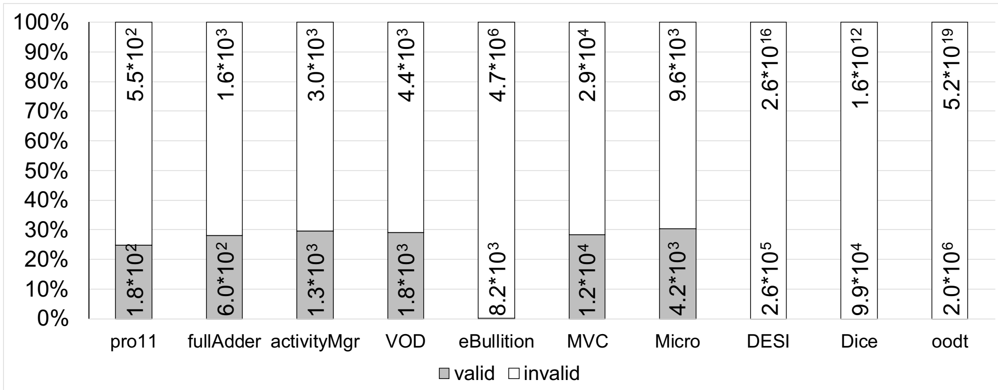


Figure 11: Reduction in % discovering valid (true set) and invalid (false set) values

Table 3: Reduction of values per scope element count

#SE	All Combinations	Valid Combinations	% of discovered valid combinations
1	$6.2 * 10^5$	$1.3 * 10^5$	21%
2	$3.2 * 10^8$	$2.2 * 10^6$	$6.6 * 10^{-3}$%
3	$5.1 * 10^{11}$	$4 * 10^4$	$7.7 * 10^{-8}$%
4	$9.4 * 10^{14}$	10^4	10^{-11}%
5	$1.2 * 10^{18}$	230	$1.7 * 10^{-16}$%
6	10^{15}	51	$4.9 * 10^{-14}$%

of valid value combinations was determined the same way, but the amount of values $|values_j|$ has been reduced by our approach (it contains only combinations of valid values). The invalid value combinations were determined by subtracting the amount of valid value combinations from the maximum. For instance, in Figure 11 for the model `pro11` the theoretical maximum was $7.3 * 10^2$ (computed with the formula previously shown) and our approach reduced that amount to $1.8 * 10^2$, which is a reduction of 75%.

The percentage of the models `eBullition`, `DESI`, `Dice`, and `oodt` was reduced by more than 99.99%. For instance for the model `oodt` in Figure 11, our approach was able to reduce the amount of value combinations from $5.2*10^{19}$ to $2*10^6$ (representing 10^{-14}% of all the theoretical maximum values), which is a reduction of 14 orders of magnitude.

Table 3 further shows the amount of reduction per scope element (needed to repair an inconsistency) summed up over all evaluated models. It shows that the more scope elements we have to repair in combination in a single inconsistency, the more efficient the reduction is performed by our approach. For instance we were able to reduce the amount of values for five scope elements from $1.2*10^{18}$ to 230 which is a reduction of 16 orders of magnitude. This is because many invalid combinations occurred due to either invalid values or invalid combinations of valid values (i.e., from Section 4.4.2 with the intersection set). Our approach was able to reduce the amount of combinations by 79% for one scope element (the maximum was $6.2*10^5$), while the reduction for five and six

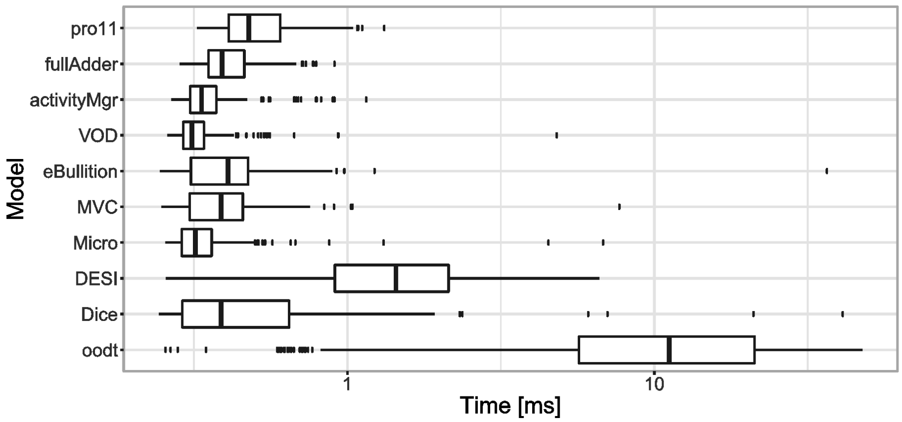


Figure 12: Runtime for the validation

scope element exceeded 99.99% leading to $1.7*10^{-16}$% (from a maximum of $1.2*10^{18}$) and $4.9*10^{-14}$% (from a maximum of 10^{15}). Ultimately, the `true` set of values are then turned into concrete repairs presenting the model elements that need to be changed and the change operations (with concrete values), which is more understandable for developers and automatically executable. In contrast to the first part of our evaluation (Section 5.2) we do not need to generate concrete repairs containing invalid values, since those values already have been filtered out (i.e., they are contained in the `false` set). Thus we do not waste time checking them.

RQ 5: This research question In this evaluation, we applied our mechanism to the three versioned models (`pro11`, `fullAdder`, `activityMngr`) taken again from GitHub. Every model contains several inconsistencies in version 1 and the model developer had manually fixed those inconsistencies in version 2 (since the inconsistencies are not present anymore). Hence, we investigated the set of our validated values in the `true` sets for version 1, and we were able to find every value the developer had used to manually repair their inconsistencies in every model of version 2. This means that our approach did not miss valid values (i.e., did not incorrectly put a valid value in the `false` set) and covered all of the developer's needs regarding the repair of inconsistencies. Of course our approach also suggested additional values in the `true` set, which may be of interest for other developers, i.e. additional repair alternatives. Example of repairs were to add a reference between classes and to rename messages and lifelines in the sequence diagram. For example, in `pro11` the engineer applied a repair by renaming an operation "search" in the class diagram, which was a repair we computed.

RQ 6: To measure the time scalability of our approach, we determined the time performance for discovering the valid values among the provided ones. We built up the validation tree (already during the inconsistency detection phase), collected the values, and then started to record the time for the value discovery for every abstract repair. Figure 12 shows the time it took to separate all values into `true` and `false` sets via box-plots. The x-axis shows the time in milliseconds, and the y-axis shows every model we have evaluated. As an example, for the model `MVC` our implementation took 1ms as median execution time, 10ms as maximum and 0.5ms as minimum recorded time (note the outliers in bold dots). The average execution time over all models for the validation was 1.5ms with a maximum of 47ms (outlier) observed for `oodt` model. We also observed that the values' size varied from 2 to 16173 with an average of 1200 values to be separated into valid and invalid values. The execution time remained stable with couple of ms even for large sets of values. This is because once we abstract from the set of values to boolean, the discovery of the `true` and `false` sets is computed instantly, which allows our approach to scale to large sets of values.

6. Threats to validity

In this section we discuss internal, external and conclusion threats to validity after Wohlin et. al. [29].

Internal Validity: The internal threats to validity for the first part of the evaluation (abstract to concrete repair transformation) 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 engineer in case of the three versioned models) are still computed in our case studies. Thus, we deem this threat to validity as acceptable here.

The internal threats to validity for both parts of the evaluation (the transformation and value validation) are centered on the generator functions [27] we used for retrieving possible values for repairing inconsistencies. Those generator functions only return existing values in the model, and cannot create new information. This means, that our evaluation did not validate values for abstract repairs which state that some new information has to be introduced to the model to fix the corresponding inconsistency. For instance, if an abstract repair states that a class with specific attributes has to be added, the used generator functions would not be able to return values for the new class name. To further mitigate this threat, in our tool we allow the engineer to provide new values as input or use a different source for values which are used to turn those abstract repairs into concrete repairs. Moreover, our approach depends on the expressed OCL consistency rules which are specified by the engineer. We only check their syntactic correctness, but not their semantic correctness/completeness since only the engineer knows its intent.

For the internal validity, we additionally performed an analysis on all detected inconsistencies to identify which type of model elements and properties they affect, i.e., what types of model element properties the corresponding repairs change. For instance a repair could change a name, add an operation to a class, or change the type of an element. Figure 13 shows the percentage of all inconsistency types. As an example, 2.32% were about the messages being received by an object (receiveEvent) and 5.58% were about names of model elements. However, keep in mind that the type of inconsistencies highly depends on the defined consistency rules, e.g., if there is no consistency rule defined that checks names, then there will not be inconsistencies about names.

External Validity: We implemented our approach for UML and OCL, although we are confident that the computation of concrete repairs and the discovery of valid

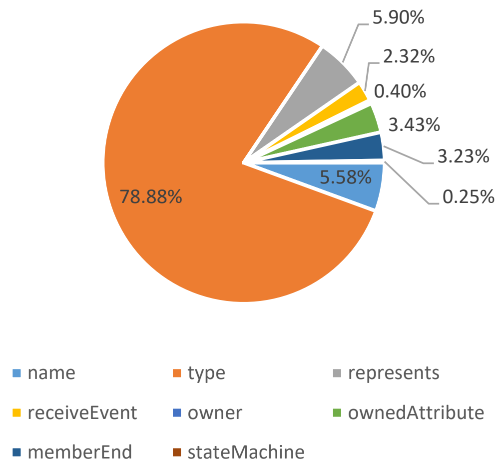


Figure 13: Inconsistency Types

values is also applicable to other modeling and constraint languages, we cannot generalize our results to all modeling constraint languages. However, The only requirement to apply our approach to other domains, is to provide an equivalent of the validation trees (i.e., AST) that has the consistency rules' conditions, and to retrieve values for fixing inconsistencies. In future work we plan to evaluate on other modeling and constraint languages as well. Moreover, we also cannot generalize our results to other sources of values (e.g., from search-based or heuristic approaches). The reduction % would vary depending on the quality of the provided values. Nonetheless our approach was able to separate instantly valid from invalid values efficiently, and the processing time is independent from the total amount of provided values and the ratio of the valid/invalid values.

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 5.2. Furthermore, we demonstrated that our value validation algorithm is very fast and reduces the amount of values significantly. The results in our case studies indicate that we are able to validate a large amount of values within milliseconds. To have more evident results, we plan to evaluate on more models.

7. 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.

Inconsistency checking: Our approach relies on validation trees and abstract repairs as input for finding corresponding concrete values, and therefore concrete repairs. All approaches that provide abstract repairs and an expression based tree structure may be used as input for our approach to generate concrete repairs. For instance Xiong et al., Reder et al. and Jackson et al. use a very similar notation of abstract repairs [12, 13, 14, 21]). In summary, the only requirement for other approaches is that they provide affected model elements, their properties and the corresponding repair operation and also expression which can be validated with the provided values.

Abstract repairs: As presented in Section 4.3, 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 [24, 14, 12, 13]. 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 [14]. 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 [12, 13]. 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 [12]. 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 [11]. 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 [12]. We extended the approach of Reder et al. that was built for usage with the employed incremental consistency checker [14]. Moreover, this approach does not require fixing-related statements to be added to the applied OCL constraints, as it is in [12]. Neither does it try to execute adaptations automatically as it is proposed by [16, 12].

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.

Le et al. propose repairs for bugs in programs by applying structured specification, deductive verification and genetic programming [31]. They also further elaborated this approach with automated example extraction and repair synthesis based on those examples [32]. Similarly Ma et al. focus on vulnerability repair in source code by learning from training sets and deducing repair templates to fix those vulnerabilities

[33]. 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.

Puissant et al. [34, 35] 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. In their repair plans, they use probabilistic generation of values [36] which can reduce the quality of the models and can lead to invalid combinations of values. Taentzer et al. [37] proposed to repair inconsistent models w.r.t. their metamodels. They relied on the model change history which helps in reducing the amount of repairs and hence does not explore all possible concrete values and repairs.

Model Transformation: Another related area of research is the multi-directional model synchronization /transformation.

Giese et al. [30] use triple graph grammar rules to bidirectionally synchronize models. In contrast to their approach, our work is not based on transformation rules, but uses consistency rules to identify inconsistencies and transform abstract to concrete repairs based on values from the model itself.

Greenyer et al. [38] compare triple graph grammar rules to query/view/transformation, and discover that those two techniques have many common concepts. Based on this observation they propose a mapping between those two techniques and show that a triple graph grammar rule transformation engine can execute query/view/transformations.

Cicchetti et al. [39] propose the Janus Transformation Language (JTL), a bidirectional model transformation language , to provide a mechanism for non-bijective transformation and change propagation. This allows the authors to propagate changes from one model to multiple other models without being restricted to the transformation direction.

Wimmer et al. [40] test model to code, and code to model transformation by extending a model to model transformation approach (Tracts [41]). The authors do this by transforming M2T/T2M transformation specifications to equivalent M2M transformations.

The main difference between our approach and model transformation approaches is that we do not transform models from one meta-model to another meta-model. Our approach performs consistency checking and repair based on the same meta-model.

Finally, in contrast to fuzzing in automated bug detection, we do not utilize randomness in our value detection.

In contrast to the existing approaches, we explore the set of potential values to discover the values that will lead to executable concrete repairs. The originality of our work, is that when transforming abstract to concrete repairs, we abstract from concrete values to boolean sets, to rapidly reason about combinations that repair given inconsistencies. Another benefit of our approach in contrast to the existing approaches is that we are able to suggest fixing inconsistencies partially as well. This means for instance if three scope elements have to be fixed, but one of them has no valid value, our approach still provides valid values for the other two scope elements. To the best of our knowledge, this is not supported by the existing approaches.

8. Conclusion and Future Work

In this 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 engineer can always add generator functions that are specific to her context and models. Furthermore we presented a novel mechanism for discovering and validating values automatically to get relevant values for concrete repairs of model inconsistencies. This mechanism utilized validation trees (i.e., instances of consistency rules) that reveal details about the causes of inconsistency but also the structure of the consistency rules. It validates the provided values based on the consistency rule's conditions, groups them into `true` and `false` sets, and then combines them based on boolean expressions found in the validation tree. Valid values are then in the `true` set to be used for concrete repairs.

Our evaluation applied 20 consistency rules to 10 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. In addition, 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. Furthermore, we have shown on larger models that our approach is scalable thanks to our value filtering mechanism. It was able to reduce the amount of combinations by 83% on average in less than 1.5ms by several orders of magnitude mostly in best cases. This saves time and effort that would have been spent on exploring invalid concrete repairs. Our contribution can thus be integrated with existing or future approaches of inconsistency repairs to accelerate significantly the process of computing concrete repairs.

For future work, we plan to further group valid values based on different criteria. For instance, group integers into positive and negative, or ranges from zero to 100, 100 to 1000, etc. This might help in those cases, where there are still a lot of valid combinations in the end. We also plan to consider constraints on the valid values to further reduce the size of the `true` set (e.g. among positive integers consider only odd values). Finally, as alternative repairs are proposed per inconsistency, in [42] we proposed one heuristic and evaluated it, we plan to provide other ranking heuristics to support the developers in quickly choosing repairs and to evaluate their benefit.

9. Acknowledgments

This research was supported by the Austrian Science Fund FWF P 31989, by Pro2Future, a COMET K1-Centre of the Austrian Research Promotion Agency (FFG), grant no. 854184, and by the LIT Secure and Correct System Lab funded by the State of Upper Austria.This research was also supported by funding from the CNRS PEPS 2019 and from the AIS Rennes Métropole under grant no. 190270

References

[1] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of mde in industry," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 471–480, ACM, 2011.

[2] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, "Assessing the state-of-practice of model-based engineering in the embedded systems domain," in *Model-Driven Engineering Languages and Systems*, pp. 166–182, Springer, 2014.

[3] J. Hutchinson, M. Rouncefield, and J. Whittle, "Model-driven engineering practices in industry," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 633–642, ACM, 2011.

[4] D. C. Schmidt, "Model-driven engineering," *COMPUTER-IEEE COMPUTER SOCIETY-*, vol. 39, no. 2, p. 25, 2006.

[5] J. Whittle, J. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE software*, vol. 31, no. 3, pp. 79–85, 2014.

[6] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "UML2alloy: A challenging model transformation," in *MODELS*, pp. 436–450, 2007.

[7] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," in *EDOC*, pp. 222–231, IEEE, 2008.

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

[9] A. Demuth, R. Kretschmer, A. Egyed, and D. Maes, "Introducing traceability and consistency checking for change impact analysis across engineering tools in an automation solution company: An experience report," in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pp. 529–538, IEEE, 2016.

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

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

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

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

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

[15] T. Mens, R. Van Der Straeten, and M. D'Hondt, "Detecting and resolving model inconsistencies using transformation dependency analysis," in *International Conference on Model Driven Engineering Languages and Systems*, pp. 200–214, Springer, 2006.

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

[20] 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.

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

[22] R. Kretschmer, D. E. Khelladi, A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "From abstract to concrete repairs of model inconsistencies: An automated approach," in *APSEC 2017*, pp. 456–465, 2017.

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

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

[25] D. Torre, Y. Labiche, and M. Genero, "Uml consistency rules: a systematic mapping study," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, p. 6, ACM, 2014.

[26] L. C. Briand, Y. Labiche, and L. O'sullivan, "Impact analysis and change management of uml models," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pp. 256–265, IEEE, 2003.

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

[28] 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.

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

[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] 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.

[32] 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.

[33] 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.

[34] 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.

[35] J. P. Puissant, R. Van Der Straeten, and T. Mens, "Badger: A regression planner to resolve design model inconsistencies," in *European Conference on Modelling Foundations and Applications*, pp. 146–161, Springer, 2012.

[36] A. Mougenot, A. Darrasse, X. Blanc, and M. Soria, "Uniform random generation of huge metamodel instances," in *European Conference on Model Driven Architecture-Foundations and Applications*, pp. 130–145, Springer, 2009.

[37] G. Taentzer, M. Ohrndorf, Y. Lamo, and A. Rutle, "Change-preserving model repair," in *International Conference on Fundamental Approaches to Software Engineering*, pp. 283–299, Springer, 2017.

[38] J. Greenyer and E. Kindler, "Comparing relational model transformation technologies: implementing query/view/transformation with triple graph grammars," *Software & Systems Modeling*, vol. 9, no. 1, p. 21, 2010.

[39] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Jtl: a bidirectional and change propagating transformation language," in *International Conference on Software Language Engineering*, pp. 183–202, Springer, 2010.

[40] M. Wimmer and L. Burgueño, "Testing m2t/t2m transformations," in *International Conference on Model Driven Engineering Languages and Systems*, pp. 203–219, Springer, 2013.

[41] M. Gogolla and A. Vallecillo, "Tractable model transformation testing," in *European Conference on Modelling Foundations and Applications*, pp. 221–235, Springer, 2011.

[42] D. E. Khelladi, R. Kretschmer, and A. Egyed, "Detecting and exploring side effects when repairing model inconsistencies," in *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20-22, 2019*, pp. 113–126, 2019.