Exploring Dependencies Among Inconsistencies to Enhance the Consistency Maintenance of Models

Luciano Marchezan
ISSE - Johannes Kepler University Linz
Linz, Austria

Wesley K. G. Assunção
North Carolina State University
Raleigh, USA

Edvin Herac
Saad Shafiq
Alexander Egyed
ISSE - Johannes Kepler University Linz
Linz, Austria

***Abstract*—Consistency maintenance is paramount for software engineering, as it improves/guarantees the quality of artifacts (e.g., models) during maintenance and evolution. To perform this maintenance, consistency rules (CR) are commonly defined and applied to evaluate model elements according to desired properties. By empirical studies, it is known that CRs commonly evaluate similar model elements (e.g., multiple CRs checking the consistency of a UML class). Thus, we hypothesize that CRs can be used as a means to identify dependencies among inconsistencies and support consistency maintenance tasks. Currently, however, no study investigates to what extent dependencies can be identified and how they can be used to repair inconsistencies. In this paper, we explore dependencies between CRs to identify and group dependent inconsistencies. For that, we define a metamodel that allows dependencies to be expressed. Furthermore, we propose a consistency maintenance and dependency analysis mechanism that uses such a metamodel. Additionally, the approach generates repairs for the inconsistencies, considering the groups of dependencies to identify overlapping and conflicting repairs. To evaluate the approach, we conducted an empirical study with 48 UML models and 27 CRs. The results show that our approach identifies dependencies between inconsistencies (46% of the inconsistencies have dependencies), within a reasonable time, 10ms on average in the worst case. Results also show that dependent inconsistencies can be grouped and used together to identify repairs that are either overlapping (26% on average) or conflicting (58% on average).**

***Index Terms*—Consistency rules, Inconsistency detection, Dependency analysis, UML model repair**

I. Introduction

Inconsistencies lead to cascading problems during the evolution of the software [1], [2], such as not meeting requirements, creating errors, and having a negative impact on the safety of the system [3]. To mitigate these problems, inconsistencies need to be identified and repaired. This process is referred to as consistency maintenance (i.e., a combination of consistency checking and repairing [4]–[9]). Consistency maintenance has proven benefits, aiding practitioners to find and repair inconsistencies in software models [3], [10]–[12]. This is evidenced by studies in the industry, reporting the importance that proper consistency maintenance in models has for the software development process [13]–[16]. In the literature and practice, we can find different strategies to perform consistency maintenance [17]. Among these strategies, the use of consistency rules (CR) is one of the most common [3]. This strategy allows the flexible application of consistency maintenance, as CRs can be defined for different types of models and domains. Furthermore, CRs can also be extended to generate repairs [18], [19], recommending changes that fix model inconsistencies.

Approaches for consistency maintenance, however, are mostly limited to considering inconsistencies and their repairs individually [20]–[23]. Only a few studies investigate how inconsistencies are related to each other, and how this may benefit the repairing process [24], [25]. Preliminary results from these studies showed the potential benefits that the relationships (in this work defined as *dependency*) between inconsistencies can bring to consistency maintenance. For instance, dependencies can be used to identify model elements that are critical (i.e., evaluated by several CRs) and deserve further attention when repairing an inconsistency. Furthermore, when multiple inconsistencies need to be repaired, the repair of one inconsistency may be used to fix another one (i.e., an overlapping repair). Similarly, a repair from one inconsistency may overwrite the repair of another, thus creating a conflict. This conflict prevents both inconsistencies from being fixed, and thus requires input from practitioners.

Once dependencies are identified, they can be used to group inconsistencies. These groups can aid practitioners in the repairing process by identifying overlapping and conflicting repairs, thus, reducing the number of repair alternatives. This is important as the number of repair alternatives may grow exponentially depending on the model and CR applied [21], [26]. Several approaches from the consistency maintenance field have investigated different strategies to reduce or rank repair alternatives [19], [27]–[29]. However, exploring the dependencies of inconsistencies to reduce possible repairs is still an open opportunity not explored by other approaches.

To explore the use of dependencies during consistency maintenance, in this work, we analyze and identify dependencies among CRs as a means to identify dependencies between inconsistencies. Based on this goal, we formulate the following assumptions (detailed in Section II). First, we argue that the properties and elements being analyzed by CRs can be compared to identify and group dependent inconsistencies (Assumption 1). This leads us to define the first research question (RQ) of this work: RQ1) To what extent can the CRs definition and execution be used to identify and group dependent inconsistencies? Second, these groups can be explored

2640-7574/24/$31.00 ©2024 IEEE
DOI 10.1109/SANER60148.2024.00023

to analyze inconsistencies that should be repaired together to identify overlapping and conflicting repair locations, thus minimizing the impact of repairs and aiding practitioners in reducing the number of repair alternatives (Assumption 2). For this assumption, we formulate the following RQ: RQ2) To what extent can grouped inconsistencies be used to identify overlapping and conflicting repair locations in a model?

The investigation of these RQs leads us to propose an approach to *analyze the definition and execution of CRs to identify dependencies between inconsistencies, as well as the possible implications of grouping inconsistencies to repair models.* The proposed approach (described in Section III) consists of: i) a metamodel that allows the definition of dependencies between CRs and inconsistencies during the maintenance and repairing; ii) an automated dependency analysis between CRs, used to identify and group dependencies between inconsistencies; iii) a mechanism to generate and identify overlapping and conflicting repairs based on the groups, aiding practitioners to reduce fixing locations.

Furthermore, this work contributes with an empirical evaluation (explained in Section IV) composed of 48 UML models and 27 CRs that have been used in recent benchmark studies [30], [31]. The results of the empirical evaluation show that 66% of the CREs and 46% of inconsistencies had dependencies. The runtime required is satisfactory as, in the worst case, the approach requires 10.35ms on average to detect dependencies. Results also reveal that dependencies between inconsistencies can be used to group and repair inconsistencies together, with 26% of overlapping repairs and 58% of conflicting repairs detected on average. These results can be used to reduce the effort of selecting repair alternatives, as 39% of shared repair locations were identified on average. Based on the analysis of the results and our experience in conducting this work, we provide a discussion about lessons learned and limitations of the approach, pointing to open research opportunities (Section V). Lastly, Section VI presents and compares related work, highlighting the limitations of existing studies. Section VII concludes this work with final remarks.

II. Background and Motivation

In this section, we describe the concepts and current limitations of existing literature that motivate this work. For that, we introduce an illustrative example to describe the motivation of this work. This example is composed of UML diagrams of a Robotic Arm, shown in Figure 1, and a set of four consistency rules (CR) applied to them, presented in Listing 1. These CRs are applied to check if the diagrams, i.e., models, are consistent according to the UML standard and requirements specification. The robotic arm has three components, defined in a class diagram: i) *RobotArm* with a gripper that can grab and release objects; ii) *Turntable* that rotates the arm; and iii) *ControlUnit* that manages both the arm and the turntable. The project also has a sequence diagram showing the messages exchanged between the components and a state chart describing how the states of an object from class *Turntable* can change.

Studies have discussed the importance and applicability of consistency maintenance in design models, such as UML [17]. Amongst the strategies to apply consistency maintenance, the use of CRs is one of the most well-known and widely applied [3]. Among the languages capable of defining CRs, OCL [32] is the second most adopted [33], as it is part of the UML standard [34]. Hence, several approaches use OCL for consistency maintenance of a variety of models from different domains [20], [21], [27], [35], [36]. In addition, UML models with CRs written in OCL are the most common in the evaluation of consistency maintenance and repair approaches [22], [37]. Based on this information, in this work, we consider OCL as the mechanism for defining CRs, although our approach is designed to be used alongside any CR language (see Section III). Despite the wide use of CRs for consistency maintenance in both research and practice, there are still opportunities to be explored, as discussed next.

Returning to the illustrative example, for checking the consistency of the models from Figure 1, four CRs are defined in Listing 1. CR1 and CR2 are applied to the classes, checking if their operations are unique and have corresponding transitions, respectively. CR3 and CR4 are applied to the messages of the sequence diagram, checking if they have corresponding operations and transitions, respectively.

Once the changes are applied in the models, the CRs are executed to check if the elements fulfill them. The execution of a CR into a model element is defined as a consistency rule evaluation (CRE), thus inconsistencies are CREs where the element does not fulfill the CR [37]. Inconsistencies are harmful to the models, and for repairing them, several approaches are proposed [19]–[21], [27], [38]. These approaches, however, treat inconsistencies individually, asking practitioners to select the ones that they want to repair based on their knowledge. A challenging characteristic of repairing inconsistencies is that a single repair may affect multiple CREs (even the consistent ones) [26], [31]. This happens because the repairs can affect the same properties of elements that are being analyzed by other CRs.

Since the CREs are found based on the execution of a CR in a model element, we assume that CRs and the model elements where they are applied can depend on other CREs. This dependency is illustrated by Change ① in Figure 1, which renames two transitions in the state diagram, from *t1[turnOrStop]* and *t2[turnOrStop]* to *t1[turn]* and *t2[stop]*, respectively. This change creates two inconsistencies. The first one is related to CR2, as the class *Turntable* does not have operations corresponding to the transitions *t1[turn]* and *t2[stop]*. The second one is related to CR4, as the messages *2.turnOrStop()* and *3.turnOrStop()* do not have corresponding transitions. Both CR2 and CR4 rely on the name of the transitions to check for consistency. Thus, changing the name of a transition will impact inconsistencies related to both CRs.

The same principle applies during the repairing of an inconsistency, as the changes performed by the repair can create inconsistencies for other CRs. For instance, to be consistent with CR1, the class *RobotArm* in Figure 1 has to be

Listing 1 A set of consistency rules applied to the UML diagram from Figure 1

```
context Class inv:
CR1: self.ownedOperation->forAll(op1, op2 : op1 <> op2 implies op1.name <> op2.name) - The class must have unique
     operation names

CR2: self.ownedOperation->exists(op| self.stateCharts->exists(sc:StateChart| sc.transitions->exists(t: t.name = op.name)))
     - All class operations must have a corresponding transition

context Message inv:
CR3: self.receiveEvent.asType(InteractionFragment).covered->forAll(r|r.represents.type.asType(Class).ownedOperation->
     exists(op| op.name=self.name)) - All messages must have a corresponding operation

CR4: self.receiveEvent.asType(InteractionFragment).covered->forAll(r|r.represents.type.asType(Class).stateCharts->exists(
     sc:StateChart| sc.transitions->exists(t: t.name = self.name))) - All messages must have a corresponding transition
```

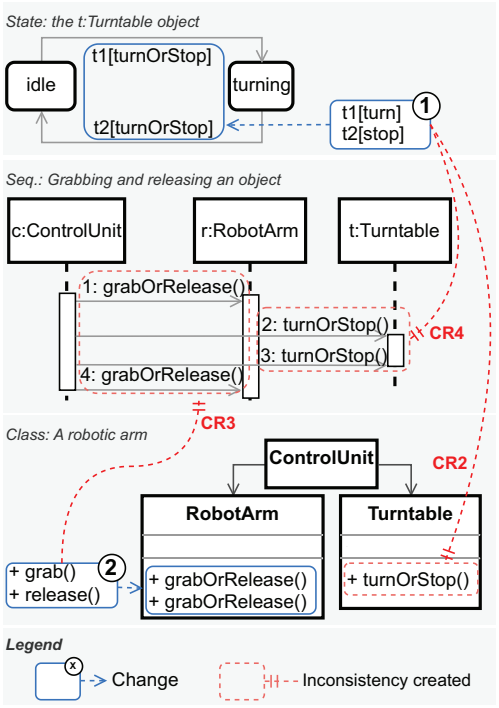

Fig. 1: Inconsistencies created due to changes in UML diagrams of a Robotic Arm

changed because it has two operations with the same name, namely *grabOrRelease*. Change ② modifies the operations to *grab* and *release*. This change repairs the inconsistency related to CR1, however, it creates a new inconsistency related to CR3. This inconsistency happens because the messages *1:grabOrRelease()* and *4:grabOrRelease()* in the sequence diagram no longer have a corresponding operation. In the case of CR1 and CR3, both rely on the name of an operation to check the consistency, thus if an operation changes its name, this change may impact both CRs. In summary, when considering repair alternatives, practitioners must be aware of their possible negative impact on the models. ***Assumption 1:*** *The definition of CRs and the generated CREs can be used to identify and group inconsistencies (or consistent CREs) that have dependencies with each other.* These groups can be shown to practitioners, giving them an indication of what elements would be impacted (in terms of consistency) by the execution of a change or repair.

Furthermore, since CR2 and CR4 are dependent, their inconsistencies may also be if they rely on the same model elements, which are the transitions *t1[turn]* and *t2[stop]*. These inconsistencies from CR2 and CR4 may be repaired by renaming the messages and operations from *turnOrStop()* to *turn()* and *stop()*, respectively. This requires changing different diagrams with multiple changes. A simpler solution, however, would be to undo Change ①, assuming that no other change was performed in the meantime, by renaming the transitions back to *turnOrStop*. This repair would fix all inconsistencies related to CR2 and CR4 while only changing one diagram. Thus, reducing the number of repairing locations, as the repair applied is overlapping for both inconsistencies.

Similarly, repair alternatives generated for one inconsistency may prevent the repairs of others at the same time. For example, to repair the inconsistencies created by Change ① in relation to CR2 (the name of the operations in the class *Turntable*) the operation *turnOrStop()* can be renamed. To match the transition *t1[turn]*, *turnOrStop()* is renamed to *turn()*. However, the *t2[stop]* transition is still inconsistent, thus the operation can be renamed again to *stop()*. This overwrites the previous repair and recreates the inconsistency related to transition *t1[turn]*. To repair both inconsistencies related to CR2, the repair alternatives have to consider both at the same time to prevent these conflicting changes. Hence, these conflicting repairs can be removed from the alternatives, aiding practitioners to reduce the number of possibilities. ***Assumption 2:*** *Grouped inconsistencies can be repaired together to identify overlapping and conflicting repair locations, aiding to reduce the number of alternatives.*

The aforementioned assumptions are the main motivation for our work. Although these are investigated in this paper by considering OCL only, these assumptions are related to most types of consistency maintenance approaches.

III. Approach

In this section, we present the approach designed to explore the identification of dependencies of CRs and inconsistencies,

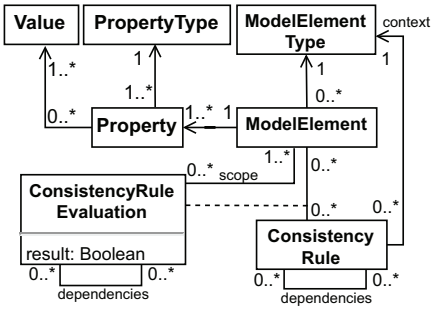

Fig. 2: Metamodel used by the approach

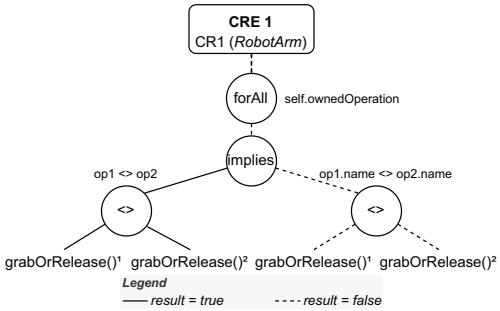

Fig. 3: CRE tree created by applying CR1 to class *RobotArm*

and the possible implications of dependencies when repairing inconsistent models.

A. A Metamodel for Inconsistency Dependencies

The definitions described in this section are illustrated by the UML class diagram presented in Figure 2. This diagram represents a metamodel that is used by our approach when applying CRs (Subsection III-B), for dependency analysis (Subsection III-C), and for repair generation and analysis (Subsection III-D).

A *ModelElement* represents an element from a given model that has a *ModelElementType* (top-right in Figure 2). For instance, a UML class diagram can have a class called "RobotArm" of the type UML Class (see Figure 1). A model element has *Property*(ies) that contain *Value*(s). A property is of a *PropertyType* which describes the cardinality of the property (e.g., Set) and the *Value* type (e.g., String). One example is the property *ownedOperation* that describes all operations from a class. Model element types are connected to the *ConsistencyRule* as the context of a CR. For instance, CR1 and CR2 from Listing 1 are defined for the context *Class*. A *ConsistencyRule* also has a relationship with the *ModelElement* class, defined as the *ConsistencyRuleEvaluation* (CRE) associative class (bottom-left in Figure 2). Each CRE represents the application of a CR in a model element of that CR's context and evaluates to a Boolean result, *true* (consistent) or *false* (inconsistent). The CRs assess the model element properties and their values to check the consistency. For example, CR1 assesses the *ownedOperation* property and generates three CREs, one for each class from Figure 1. The CREs' result is *true* for classes *ControlUnit* and *Turntable* and *false* for class *RobotArm*.

Continuing on the metamodel from Figure 2, the *dependencies* relationship between CR with itself allows the creation of dependency relationships between one or more CRs. For instance, CR1 and CR3 (Listing 1) are dependent because they both evaluate the same property from an operation, namely the operation name. The CR's dependencies are used to select the possible CREs that may be dependent, i.e., for each dependent CR, we retrieve their CREs to be analyzed (details in Section III-C). Hence, CREs can also have dependencies with each other, as the metamodel also defines *dependencies* for them. For instance, a CRE created by applying CR1 to the class *RobotArm* has a dependency to the CRE created by applying CR3 to the *r:RobotArm* lifeline in the sequence diagram. This dependency is illustrated by the negative impact of Change ② to the sequence diagram in Figure 1. Although CR1 and CR3 are dependent, not all their CREs will be dependent on each other. For example, the CRE created by applying CR1 to class *RobotArm* does not depend on the CRE created by applying CR3 to the *t:Turntable* lifeline. This happens because these CREs do not have the same model elements in their scope (details in Section III-C).

The *scope* of a CRE is defined by the relationship between CREs and model elements in Figure 2, and describes which model elements are assessed when a CRE is created. For instance, when a CRE for CR1 is created for class *RobotArm*, the class itself, and all its operations are assessed. Hence, the scope of this CRE contains the class *RobotArm* and the two operations *grabOrRelease*.

The metamodel allows our approach to be adopted for different CR languages and models. The only requirement is that these can be represented using a property-value representation.

B. Applying CRs to Models

For each CR defined for a context, our approach applies it to all model elements of that context. This is achieved by traversing the CR's expressions and evaluating the properties of the model element in context. Each expression (e.g., *forAll*, *equals*) is evaluated individually, and their results are concatenated based on the CR's definition. These results are represented as the CRE tree-like structure, illustrated in Figure 3. In this example, CR1 is applied to class *RobotArm*, creating CRE1 with each node of the tree representing an expression from the CR, namely *forAll*, *implies*, and two *not equals* (**<>**) expressions. The result of each expression is described in Figure 3 by the line connecting the tree nodes (solid for true and dashed for false).

For instance, the expression *op1.name* **<>** *op2.name*, evaluates to false (dashed line), as both operations have the same name. This leads to the parent expressions *implies* and *forAll* also resulting in *false*, meaning that CRE1 is inconsistent. The checking is performed incrementally, thus every time a model element changes, CRs that assess that model element

Algorithm 1 Dependency analysis for CRs

```
1: function DEPENDENCYANALYSIS(crs, newCr)
2:     for cr in crs do
3:         if checkDependency(cr, newCr) then
4:             cr.dependencies.add(newCr)          ▷ Oppos. relation is also created
5:     crs.add(newCr)
6:     return crs
7: function CHECKDEPENDENCY(cr1, cr2)
8:     for prop1 in cr1.propertiesChecked do
9:         for prop2 in cr2.propertiesChecked do
10:            if prop1.name = prop2.name then
11:                if prop1.element = prop2.element then
12:                    return true
13:    return false
```

Algorithm 2 Dependency analysis for CREs

```
1: function DEPENDENCYANALYSIS(cres, newCre)
2:     for cre in cres do
3:         if cre.cr.dependencies.contains(newCre.cr) or cre.cr = newCre.cr then
4:             if checkDependency(cre, newCre) then
5:                 cre.dependencies.add(newCre)    ▷ Oppos. relation is also created
6:     cres.add(newCre)
7:     return cres
8: function CHECKDEPENDENCY(cre1, cre2)
9:     for element in cre1.scope do
10:        if cre2.scope.contains(element) then
11:            return true
12:    return false
```

are applied to it, creating or updating a CRE. Additional information about this step (including algorithms) are provided in our online appendix [39].

C. Dependency Analysis

The dependency analysis is performed in two ways: i) for CRs and ii) for CREs. Considering the dependency analysis of CRs, it is triggered when a CR is created or updated. The approach analyzes all existing CRs to check if they are dependent on each other. This is performed by the *dependencyAnalysis* function described in Algorithm 1. The function takes as parameters a set of CRs, plus the newly created or updated CR. In case the CR was updated, it is first removed from the set. The function iterates over the *crs* (line 2), calling the *checkDependency* function and passing each CR from the set and the new CR as parameters (line 3).

The *checkDependency* function iterates over all *propertiesChecked* from both CRs (lines 8 and 9). The *propertiesChecked* is a set with all properties from model elements being assessed by the CRs in their expressions. If the two properties from both CRs have the same name and belong to the same model element type (lines 10 and 11), then a dependency is found, returning *true* (line 12), otherwise the function returns *false* (line 13). If the return of the *checkDependency* function is *true*, a dependency between the two CRs is created (line 4). For example, considering CR1 and CR3 in Listing 1, both rules assess the name of a class operation. Thus, a dependency between these two CRs is found and created. A dependency is always bidirectional, Figure 4 illustrates an example of dependency, showing that CR1 depends on CR3 and that CR3 depends on CR1.

The dependency analysis of CREs is performed by the *dependencyAnalysis* function described in Algorithm 2. This analysis is triggered after creating/updating a CRE. This is performed either when a CR is created/updated or when a model element is changed. The *dependencyAnalysis* of CREs is similar to the *dependencyAnalysis* of CRs. The main difference is that the function only compares CREs in case that their CRs have dependencies with each other *or* if they have the same CR (line 3). It is important to also check CREs originating from the same CR because they will share the same properties from the same model element types. Thus, it is also possible that their CREs have similar elements in their scope. The *checkDependency* function iterates over the scope of the first CRE (line 9) to check if an element from the scope is also part of the scope of the other CRE (line 10). In case this is *true*, there is a dependency between both CREs (created bidirectionally on line 5).

If we consider CREs created for CR1 and CR3 from Listing 1, since both CRs are dependent on each other, all CREs from them must be checked for dependency. As illustrated in Figure 4, dependencies are created between CRE1 and CRE4, CRE1 and CRE5, CRE4 and CRE5, CRE2 and CRE6, CRE2 and CRE7, and CR6 and CR7. The dependencies of these CREs are found because they share the operations of the same class, namely *RobotArm* for CRE1 and *Turtable* for CRE2, in their scope. Notice that CR3 also checks the operations of a class, although being defined in the message context. By supporting the identification of dependencies among CRs and their CREs, the approach can then group the inconsistent ones to be fixed together. The grouping is achieved by using one inconsistency as the origin (e.g., CRE4 after change ②) and retrieving its dependencies. Thus, one group would be composed of CRE4, CRE1, and CRE5. Since CREs 4 and 5 are inconsistent, they can be fixed together to minimize the impact of repairs as well as preventing possible conflicts (details in Section III-D). By allowing the identification and creation of dependencies and using them to group inconsistencies, our approach contributes to Assumption 1.

D. Repairing Dependent Inconsistencies

The repair generation and execution adopted in our approach are based on previous state-of-the-art approaches found in the literature [4], [7], [19]. The repairs are generated by traversing the CRE tree (Figure 3). After traversing the tree, a repair action is generated for each inconsistent node based on generator functions defined as part of the expression. Thus, a *repair action* is defined as a single change performed in a model element to fix a given inconsistency. A *repair* is a collection of repair actions that together fix the inconsistency (i.e., one inconsistency may require multiple actions). Due to the space limitation, details about repair generation are given in our online appendix [39].

In this context, the group of dependencies can be beneficial in deciding how to repair multiple inconsistencies. For example, Change ① (Figure 1) created multiple inconsistencies related to CR2 and CR4. Since these inconsistencies are dependent on each other, analyzing them together allows practitioners to check for overlapping repairs (i.e., repairs that

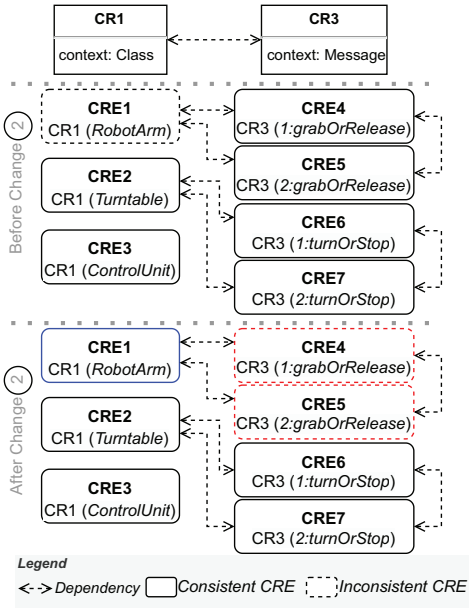

Fig. 4: Dependency analysis of CR1, CR3 and their CREs

can be applied to both inconsistencies) to reduce the number of possible alternatives. For example, consider Change ② performed in class *RobotArm* (Figure 1). This change is repairing CRE1 by modifying the name of the operations from the *RobotArm* class. Figure 4 illustrates the CREs that are impacted by Change ②, which repairs CRE1 but breaks CRE4 and CRE5 (bottom part of the figure). In this case, if instead of renaming both operations the repair deleted one of the *grabOrRelease()*, all CREs would now be consistent. This would also reduce the number of affected fixing locations, as only one change would be required.

Similarly, the groups can be used to identify conflicting repairs, such as the example of fixing the inconsistencies related to the transitions *t1[turn]* and *t2[stop]*. Our approach supports the identification of overlapping and conflicting repairs by retrieving all repair actions from one inconsistency. Then, comparing all these repair actions with the actions from dependent inconsistencies. During this comparison (shown in Algorithm 3), if two repair actions (one from each inconsistency) are exactly the same, an *overlap* relation is created between them (line 5). If not, they are compared to check if they are conflicting. A conflict is found if one repair action (ra_1) matches one of the following conditions when compared to another repair action (ra_2), given that ra_1 and ra_2 originate from different inconsistencies: i) ra_1 modifies the same property of the same model element modified by ra_2; or ii) a ra_1 deletes a model element that was added/modified by ra_2 (and vice-versa). If any of these conditions are true, the approach creates a *conflict* relationship between ra_1 a ra_2 (line 8). For both types of relationships (overlaps and conflicts), the relation is bidirectional.

Algorithm 3 Checking relationships between repair actions

```
1: function CHECKREPAIRRELATION(incon1, incon2)
2:     for ra1 in incon1.repairActions do
3:         for ra2 in incon2.repairActions do
4:             if ra1.overlapsWith(ra2) then          ▷ Oppos. relation is also created
5:                 ra1.addRelation(Operator.OVERLAP, ra2)
6:             else
7:                 if ra1.conflictsWith(ra2) then     ▷ Oppos. relation is also created
8:                     ra1.addRelation(Operator.CONFLICT, ra2)
```

The number of repair actions may grow exponentially to a point where it is not even measurable by current technology [29]. Thus, the comparison between repair actions is only executed considering a group of dependent inconsistencies. Since the dependency is found based on similar properties and elements that are part of a CRE's scope, the repair actions would likely modify similar elements and properties. Once again, conflicts can be used to reduce the number of fixing locations. For example, practitioners can ignore conflicting repair locations and focus on the non-conflicting ones, reducing the number of possibilities. Thus, we argue that by supporting the analysis and identification of overlapping and conflicting repairs, our approach contributes to Assumption 2.

IV. Evaluation

In this section, we present the study design, results, and threats to the validity to evaluate the proposed approach.

A. Prototype Implementation

To evaluate our approach, we implemented a prototype tool in Java as a service that is part of a server that supports a generic infrastructure [40]. This infrastructure supports the connection of different engineering tools by implementing tool adapters that transform artifacts created in these tools to a model representation based on the metamodel presented in Figure 2. Although multiple artifact types are supported by the server in this evaluation we focus on UML due to the majority of CRs being defined for UML models.[1]

B. Study Design

This study was designed with the goal of evaluating our approach regarding the assumptions described in Section II. For that, we pose the following research questions.

RQ1) To what extent can the CRs definition and execution be used to identify and group dependent inconsistencies? **Rationale:** to confirm Assumption 1, we investigated how the approach performs for a variety of models (i.e., ranging in size and diagram types) and with a diverse set of CRs (i.e., with different contexts and expressions). **Method:** we apply the dependency analysis of our approach in a dataset of 48 UML models with a set of 27 CRs. This dataset was created by combining state-of-the-art datasets found in recent studies [19], [29], [30]. The UML models contain different types of diagrams, including class, sequence, state, activity, and use case, among others. The set of 27 CRs is also

[1]Details about the implementation are available at https://isse.jku.at/designspace.

diverse, checking consistencies for elements from all diagram types of our dataset. **Metrics:** i) the size of the models; ii) the number of CREs (consistent and inconsistent) created by applying the CRs; iii) the number of CRE dependencies; iv) the number and size of dependency groups—a group is a unique set of dependent CREs. Thus, if two CREs have exactly the same dependencies, only one group is counted since the dependencies of these two CREs represent the same set; v) the time required to find dependencies between CREs.

RQ2) To what extent can grouped inconsistencies be used to identify overlapping and conflicting repair locations in a model? **Rationale:** we aim at observing if the group of dependencies can be used to: i) identify repairs that can be used to fix multiple inconsistencies (overlapping); ii) identify inconsistencies that cannot be fixed because repairs for other inconsistencies prevent that (conflicting); and iii) identify fixing locations shared by dependent inconsistencies (Assumption 2). **Method:** we use the dataset from RQ1 to generate repairs and compare them considering the groups of dependent inconsistencies. **Metrics:** i) number of repairs per inconsistency; ii) number of overlapping repairs and conflicting repairs in a group of inconsistencies; iii) the number of fixing locations shared by dependent inconsistencies, which is obtained by considering both conflicting and overlapping repairs since both types affect the same location. For the overlaps and conflict metrics, we compare all inconsistencies from one group incrementally, to evaluate if the number of inconsistencies considered has an impact on the results. For instance, in a group of 4 inconsistencies, we first compare only the repairs of two inconsistencies, then we compare repairs from three inconsistencies, and then from all of them.

C. Results and Analysis

In this subsection, we present the results of our study and the corresponding analysis grouped by RQ. The artifacts and results are available online [39].

Identifying and grouping dependent CREs: Table I presents the results of dependency detection for our dataset of 48 UML models, sorted by size. Notice that we have models ranging from 79 to 9,823 elements. At first, we can observe that dependencies were detected in all 48 models. However, not all models had inconsistencies with dependencies. In general, the set of 27 CRs was able to create many CREs, ranging from 94 to 14,510. The number of CREs with dependencies per model (column *#CRE w/ Dep.* in Table I) evidences that the approach can detect many dependencies, considering the number of CREs created. The time required to detect dependencies among CREs stayed between 0.55 (Model 43) and 10.35 (Model 1) milliseconds on average per CRE.

The number of inconsistencies with dependencies varied for all models (46.36% on average, bottom of Table I). When analyzing individual models (bar-plots for column *#Inc. w/ Dep.*), we notice that in some cases the percentage is lower than 50% (e.g., Models 1, 2, 5, among others). A low dependency between inconsistencies can benefit the repair activity, as non-dependent inconsistencies can be repaired without creating side effects or conflicts because the inconsistencies are not dependent on other CREs. This low dependency, however, reduces the chance of using repairs to fix multiple inconsistencies, as the inconsistencies are mostly related to the different model elements.

TABLE I: UML models sorted by size. The bar plots represent the percentage of the dependencies found. Time represents the average time (*ms*) to detect a dependency.

Id	Size	#CREs	#CRE w/ Dep.	#Inc.	#Inc. w/ Dep.	Time
1	9823	14510	9203	4053	1058	10.35
2	8757	7772	4836	2355	691	4.56
3	8074	9473	6445	3554	1285	5.49
4	5754	8990	7441	3448	2751	4.90
5	4231	2228	1389	534	63	1.25
6	3428	1748	1101	484	141	0.97
7	3216	3168	2173	932	500	1.74
8	2704	3228	2597	1708	1410	1.76
9	2628	2935	1778	776	204	1.65
10	2508	3929	2733	1214	825	2.25
11	2445	393	257	39	0	0.58
12	2245	3502	2575	1121	692	2.08
13	2059	2943	2337	1036	778	1.64
14	1970	2358	1385	546	111	1.29
15	1949	3549	2516	1076	549	2.10
16	1935	1087	654	362	102	0.65
17	1781	1559	924	213	98	0.76
18	1706	2163	1223	444	64	1.28
19	1441	2225	1665	695	503	1.32
20	1437	2106	1693	648	523	1.12
21	1211	1701	1315	602	418	0.94
22	1086	1632	1269	608	441	0.93
23	1032	1921	1580	1031	877	1.11
24	1029	1281	817	290	180	0.65
25	801	853	594	427	291	0.68
26	728	685	308	153	5	0.63
27	617	1206	950	595	493	0.66
28	575	739	513	262	194	0.59
29	560	590	319	171	53	0.64
30	536	480	296	101	0	0.78
31	516	851	703	447	363	0.61
32	452	653	459	256	178	0.66
33	446	625	489	319	249	0.62
34	427	376	180	72	10	0.76
35	425	600	520	276	220	0.75
36	382	567	399	232	149	0.61
37	370	552	283	108	7	0.61
38	254	329	258	143	118	0.73
39	248	267	119	26	0	0.70
40	206	372	282	163	119	0.67
41	202	354	227	65	38	0.89
42	189	448	439	291	285	0.66
43	125	96	43	14	0	0.55
44	103	172	88	35	1	0.62
45	96	94	30	8	0	0.62
46	96	100	50	13	0	0.62
47	88	155	120	70	54	0.55
48	79	106	57	23	0	0.65

Overall Results for the Dependencies Percentage

Dep. Type	Min	Median.	Max	Avg	Std.
CREs	31.90	68.30	98.00	66.61	13.21
Inc.	0.00	56.05	97.90	46.38	36.10

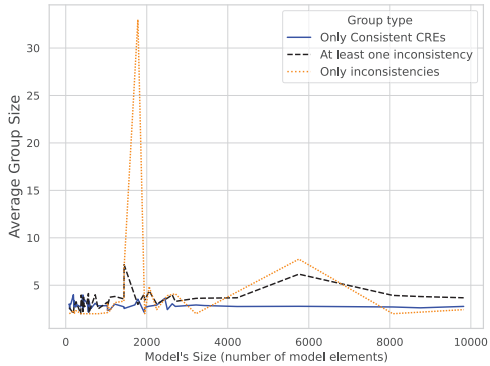

Fig. 5: Results related to the size of groups of dependency

For other models, however, the number of inconsistencies with dependencies was higher than 50% (e.g., Models 4, 8, 10, 19, 20, among others). For these cases, it is more likely that *repairing an inconsistency may have a high impact on other CREs*. We argue that in such cases, *inconsistencies can be prioritized considering the number of dependencies found*. For this, we can look at the groups of dependencies. Figure 5 summarizes the results related to the groups of CREs.[2] As illustrated, we notice that *the size of the model does not have a direct impact on the number of dependencies in a group*. This demonstrates the scalability of the approach as independently of the size of the model, the number of inconsistencies in a group remains reasonably small (with one outlier as illustrated in Figure 5). The number of dependent inconsistencies being small is important as developers have to deal with the inconsistencies in the groups, and thus having a large number can lead to the same or even more effort than just dealing with all inconsistencies (not considering any dependencies) as traditional approaches may suggest.

Answering RQ1: Our approach can detect a varied amount of dependencies (66% for all CREs and 46% for inconsistent CREs, on average) in a variety of models. The detection runtime was 10.35 ms in the worst case, thus the approach can be applied without causing overhead. As the number of inconsistencies in a group is not directly impacted by the size of the model, the approach is scalable.

Using dependencies during repair: Table II presents the repair results considering the groups with only inconsistencies grouped by CR (for some CRs, there was no group with only inconsistencies). The number of overlapping repairs was 14.57 on average, while conflicts were twice as that (32.15). This shows that identifying conflicting repairs from a group of inconsistencies is more common than identifying overlapping repairs. Furthermore, shared repair locations found between inconsistencies stayed at 21.58 on average. Considering that the total average of repairs generated was 55.40, the approach identified around 26% overlaps, 58% conflicts, and 39% shared

[2]Detailed results are available in the online appendix [39].

TABLE II: Repair results grouped by CR

CRs	Repairs		Overlaps		Conflicts		Shared Loc.	
	Avg.	Std.	Avg.	Std.	Avg.	Std.	Avg.	Std.
01	12.98	7.35	7.53	3.89	4.93	5.54	4.33	3.51
02	11.00	0	1.74	1.9	4.71	1.06	4.71	1.06
03	9.00	0	7.04	1.32	6.16	3.07	2.98	1.5
09	21.00	0	1.70	2	8.89	2.65	8.89	2.65
10	12.07	6.78	6.18	3.03	7.32	3.84	4.71	4.05
11	12.31	5.71	4.92	3.5	5.45	4.21	4.96	3.7
16	8.06	0.72	6.95	0.7	2.05	0.68	2.04	0.46
20	11.00	0	1.74	1.9	4.71	1.06	4.71	1.06
22	357.25	68.24	69.36	46.22	209.69	50.56	140.40	33.25
23	109.23	64.15	13.31	19.91	56.78	26.42	46.60	29.75
24	45.53	20.71	39.82	24.12	43.00	21.58	13.01	9.31
Avg.	**55.40**	**15.79**	**14.57**	**9.86**	**32.15**	**10.97**	**21.58**	**8.21**

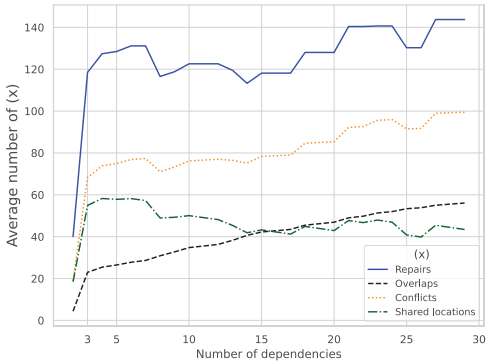

Fig. 6: Summary of results about repairs overlapping, conflicting and fixed locations from inconsistency groups.

locations on average. These results show that *practitioners can benefit from dependencies during the selection of repairs as they can focus on specific locations, select overlapping repairs that fix multiple inconsistencies, or filter out a large number of conflicts*. These results are also evident when considering the number of dependencies per group.

Figure 6 summarizes the results related to the identification of conflicts, overlaps, and fixing locations inside the groups of dependencies.[3] For both conflicting and overlapping repairs, the average number increases as more dependencies were considered from a group (recall that we collected these results using all possibilities for all groups with at least two inconsistencies being considered). As illustrated by Figure 6, the number of conflicts was higher than overlaps, staying above 50% of the total repairs when at least 3 inconsistencies were considered from the group. This behavior demonstrates that *by considering the conflicts between dependent inconsistencies, practitioners can filter out a considerable number of repairs*. The number of overlaps is lower than the conflicts, however, it also grows as the number of dependencies increases. This evidences that *when aiming at fixing multiple inconsistencies with one repair, the dependencies can be used to gather and analyze these possibilities.*

[3]Details about these results are available in the online appendix [39].

When considering the shared repair locations, increasing the number of dependencies considered from a group, reduced the shared locations. Recall that the number of shared locations (i.e., elements and properties modified by repairs) is collected by only counting the locations that are shared by all dependencies considered. Thus, *by considering the repairs of groups with more dependencies, the number of fixing locations is reduced.*

Answering RQ2: Groups of dependent inconsistencies can be used to identify conflicting repairs that can be filtered out (58% avg.). Also, inconsistencies with dependencies may be repaired together, using overlapping repairs (26% avg.). For both cases, the number increases the more dependencies are considered. Consequently, the number of fixing locations can be reduced (39% avg.).

D. Threats to Validity

In this section, we present threats related to internal, external, and conclusion validity [41], [42]. Considering the *internal validity*, we mitigated a threat related to the selection of the models by using models from different related work [19], [29], [30] (ranging from 79 to 9,823 model elements). This resulted in a range from 94 to 14,510 of CREs. Another threat is the CRs used, as the inconsistencies identified originate from them. The set of 27 CRs used was obtained from related work that systematically collected CRs that are important for UML models [33], evaluating their importance with probationers of the field [15]. These CRs contain different expression types and evaluate different contexts [39]. The results regarding the number of CREs per model support our claim that this threat was mitigated as the majority of the model elements were evaluated in most cases.

Regarding the *external validity*, a threat is related to the generalization of our results to other domains. In this evaluation, we used UML models, so we only have evidence to support the applicability of our approach with this kind of artifact. Our approach, however, can be adopted in other domains. As mentioned, the metamodel (Section III-A) aims to be generic and applicable to different model types and CR languages. The only requirement is the development of parsers to transform artifacts from different tools to be represented using our metamodel. For the evaluation presented in this paper, we have implemented a UML/EMF parser to load the UML files used into the server where our prototype implementation of the approach is running as a service.

For *conclusion validity*, a threat is related to the repairs considered for RQ2 data collection. The repair generation used as part of our approach was based on approaches from the literature [4], [7], [19]. These approaches have been evaluated with a large set of UML models in terms of scalability correctness and benefits [43]. Hence, these approaches are reliable to be used as the basis for our repair generation. This is supported by the number of repairs generated, which varied for each model. Furthermore, as different repair generation strategies may lead to different repairs being considered, we plan to consider other repair approaches [27], [28] for future evaluations to increase the generalization of the results found.

V. Lessons Learned and Research Opportunities

In this section, we discuss the limitations and lessons learned from the results of our evaluation.

Filters for CRE dependencies: Results from Table I show that the number of dependencies can be high in some cases, e.g., model 1 had more than 9k dependencies. When using dependencies to deal with inconsistencies, practitioners may be overwhelmed by these quantities. Thus, we argue that filters for CRE dependencies may be applied to retrieve a subset of focused dependencies. One possibility is to use the CRs themselves as filters, retrieving only dependencies from a specific CR. More elaborate filters can also be created, such as, only retrieving dependencies that are related to certain parts of the model. The proposal of such filters and their benefits and drawbacks remains an open research opportunity.

Using dependencies to select or filter out repairs: Repair approaches struggle to handle large sets of repair alternatives, as even one inconsistency may generate thousands of alternatives [44], [45]. The results from RQ2 show that groups of inconsistencies with a large number of dependencies, on one hand, are more difficult to repair, as one repair can generate conflicts with other repairs. On the other hand, one repair might be able to fix more than one inconsistency. A similar idea applies to groups with fewer dependencies among inconsistencies but in an opposite way. Results also evidence that the repair activity can be impacted by dependencies among inconsistencies. Knowing which are the conflicting repairs with regard to a given inconsistency enables practitioners to filter out these repairs from the set of potential repairing options. Also, the set of repairs that are similar for different inconsistencies can be prioritized by practitioners, as these repairs can fix multiple inconsistencies, thus reducing fixing locations. Evaluating the benefits of dependencies from the perspective of practitioners requires a case study with human participants and remains as future work.

Lack of dependencies for some CRs: There were cases where dependencies were not found between inconsistencies from specific CRs. These results (available at [39]), show that CRs 4-8, 12-14, and 17-19 had no inconsistencies with dependencies. This shows that the current strategy to analyze dependencies may not be relevant depending on the set of CRs used. Since the dependency analysis is based on the CRs definition, this is the main limitation of our approach. Thus, additional strategies to find dependencies have to be considered such as: user-ownership—inconsistencies that affect model elements owned by a specific user (important for collaborative engineering [2], [40]), or change-frequency—model elements that are often (or not often) changed. Since one of the benefits of dependencies is to analyze and reduce repair alternatives, having different strategies for dependency analysis can enhance the approach with more customization

options for different contexts. Investigating these different strategies is part of our future work.

Change propagation: One strategy to maintain models consistent when a change happens is the propagation of this change. For instance, by propagating the Change ② from Figure 1 to the inconsistencies created, these inconsistencies could be fixed as the values required for fixing them are those created in the change (i.e., *grab* and *release*). Repairing inconsistencies, however, may lead to ambiguous repair alternatives. For instance, undoing Change ② may be a better option than propagating the change. This, however, may still not be true for all cases as practitioners tend to have different preferences when repairing models [43]. Because of this, we argue that approaches should provide practitioners with useful information, such as dependencies, that can be used to reason about repair alternatives.

VI. Related Work

Considering the current state of the art, Torres et al. [46] collected a set of 119 CRs used for UML models. A subset of these rules is part of our set of 27 CRs applied in the evaluation. Not all rules from their set were used because they cannot be represented in OCL. We did, however, analyzed the textual similarity of their set of 119 CRs. The results show that some CRs have 100% semantic similarity with others, with the majority of CRs having around 10% semantic similarity with all others on average (results at [39]). These results support our argument regarding Assumption 1 since CRs with similar definitions show possible dependencies. This happens because dependencies are found when CRs assess the same properties of the same element types. Thus, the more similar two CRs are, the more likely they are to assess the same properties of the same elements. Other systematic studies have discussed the importance and applicability of consistency maintenance in design models, such as UML [3], [17], [33].

Consistency maintenance and repair generation are topics that have been extensively investigated [15], [20]–[23], [28], [30], [31], [47], [48]. The possible dependencies between CRs or inconsistencies, however, have only been explored by a few studies. For instance, Wu [35], [49] presents QMaxUSE, a tool that detects conflict relationships between CRs. These conflicts are found when one or more CRs have in their definition conditions that cannot be solved at the same time (i.e., contradictory). This is similar to how our approach explores inconsistency dependencies to find conflicting repairs. QMaxUSE, however, is focused on the CR level, not analyzing CREs or possible repairs, limiting its applicability depending on the set of CRs used.

Nöhrer et al. [24], [25] explore the relationship of inconsistencies to find overlapping repairs (similar to how we analyzed for RQ2). In their case, they generate repairs for all inconsistencies and randomly group them, aiming to find overlaps. Their approach shows the potential of grouping inconsistencies, however, it is inefficient as randomly grouping inconsistencies may lead to no overlaps found. Also, if random groups do not represent practical scenarios, then, their results might not be applicable to real cases. In a different direction, the work from Clariso et al. [50] extends OCL, including uncertainty in the CRs. The results show that uncertainties can be used to perform preliminary quality checks, i.e., detecting inconsistencies. The approach, however, does not explore the relationships between CRs and how inconsistencies can depend on each other.

Despite that most studies on consistency maintenance define CRs in plain English [33], these CRs are usually not supported by automated approaches. This happens because using natural language is still limited to only a few approaches that transform the written sentences into CRs of a specific language [51], [52], focusing on checking requirements [53]. There is an initial effort into using natural language processing (NLP) to transform CRs defined in natural language to OCL [54], [55]. This is a natural direction, as defining CRs in OCL is complex and tedious. Foreseeing the use of NLP for defining CRs, we defined the metamodel of our approach in a way that it can be adapted for different CR languages, including natural language. The dependency analysis, however, requires the property-element structure, which can be obtained through NLP [55].

VII. Conclusion

In this paper, we explored possible assumptions regarding dependencies of CRs and inconsistencies and how they can improve the consistency maintenance process. Our approach defines a metamodel used to express these dependencies. The consistency maintenance and dependency analysis proposed are used to detect dependencies and provide them for repair generation. During the repair generation, the approach can obtain valuable information about inconsistencies, aiding practitioners in repairing the model by considering conflicting and overlapping repairs. To evaluate our approach, we conducted an empirical evaluation showing how the approach can be applied in a variety of UML models with a large set of CRs (on a satisfactory time). The results also show that dependencies can be used to group inconsistencies, proving a mechanism to identify a large number of overlapping and conflicting repairs. These results can then be used to minimize the impact on the model by reducing the repairing locations. Future directions of our research include exploring the use of filters to define subsets of dependencies based on the CRs, model elements, and user preferences. In addition, we plan to conduct a case study with human participants to evaluate the approach from their perspective.

Data Availability

Additional information about the approach, the evaluation's artifacts and results are available in an online appendix [39].

Acknowledgements

This research has been funded by the Austrian Science Fund (FWF, P31989-N31), and by the FFG-COMET-K1 Center "Pro^{2}Future", (881844).

REFERENCES

[1] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated consistency checking of requirements specifications," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 3, p. 231–261, jul 1996.

[2] I. David, K. Aslam, I. Malavolta, and P. Lago, "Collaborative model-driven software engineering — a systematic survey of practices and needs in industry," *Journal of Systems and Software*, vol. 199, p. 111626, 5 2023.

[3] W. Torres, M. G. Van den Brand, and A. Serebrenik, "A systematic literature review of cross-domain model consistency checking by model management tools," *Software and Systems Modeling*, pp. 1–20, 2020.

[4] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein, "xlinkit: a consistency checking and smart link generation service." *ACM Trans. Internet Techn.*, vol. 2, no. 2, pp. 151–185, 2002.

[5] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency management with repair actions," in *25th International Conference on Software Engineering, 2003. Proceedings.*, 2003, pp. 455–464.

[6] A. Egyed, "Fixing Inconsistencies in UML Design Models," in *ICSE '07: 29th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 292–301.

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

[8] I. Pete and D. Balasubramaniam, "Handling the differential evolution of software artefacts: A framework for consistency management," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 599–600.

[9] L. Marchezan, W. K. G. Assunção, G. Michelon, E. Herac, and A. Egyed, "Code smell analysis in cloned java variants: The apogames case study," in *International Systems and Software Product Line Conference - Volume A*, ser. SPLC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 250–254.

[10] S. Maro, A. Anjorin, R. Wohlrab, and J.-P. Steghöfer, "Traceability maintenance: Factors and guidelines," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 414–425. [Online]. Available: https://doi.org/10.1145/2970276.2970314

[11] S. Herold, M. English, J. Buckley, S. Counsell, and M. Cinnéide, "Detection of violation causes in reflexion models," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 565–569.

[12] H. König and Z. Diskin, "Efficient consistency checking of interrelated models," in *13th European Conference Modelling Foundations and Applications (ECMFA)*. Springer, 2017, pp. 161–178.

[13] D. Torre, M. Genero, Y. Labiche, and M. Elaasar, "How consistency is handled in model driven software engineering and uml: a survey of experts in academia and industry," Carleton University, Tech. Rep., 2018.

[14] L. Marchezan, W. K. G. Assunção, E. Herac, F. Keplinger, A. Egyed, and C. Lauwerys, "Fulfilling industrial needs for consistency among engineering artifacts," in *45th International Conference on Software Engineering (ICSE) - Software Engineering in Practice*, 2023, pp. 1–12.

[15] D. Torre, M. Genero, Y. Labiche, and M. Elaasar, "How consistency is handled in model-driven software engineering and uml: an expert opinion survey," *Software Quality Journal*, pp. 1–54, 2022.

[16] R. Jongeling, F. Ciccozzi, J. Carlson, and A. Cicchetti, "Consistency management in industrial continuous model-based development settings: A reality check," *Software and Systems Modeling*, vol. 21, no. 4, p. 1511–1530, aug 2022.

[17] F. J. Lucas, F. Molina, and A. Toval, "A systematic review of UML model consistency management," *Information and Software Technology*, vol. 51, no. 12, pp. 1631–1645, 2009, quality of UML Models.

[18] N. Macedo, T. Jorge, and A. Cunha, "A feature-based classification of model repair approaches," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 615–640, 2017.

[19] L. Marchezan, R. Kretschmer, W. K. Assunção, A. Reder, and A. Egyed, "Generating repairs for inconsistent models," *Software and Systems Modeling*, pp. 1–33, 2022.

[20] A. Reder and A. Egyed, "Incremental consistency checking for complex design rules and larger model changes," in *15th International Conference Model Driven Engineering Languages and Systems (MODELS)*. Springer, 2012, pp. 202–218.

[21] 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 *Asia-Pacific Software Engineering Conference*, 2017, pp. 456–465.

[22] M. Ohrndorf, C. Pietsch, U. Kelter, and T. Kehrer, "ReVision: A Tool for History-Based Model Repair Recommendations," in *40th International Conference on Software Engineering (ICSE)*. ACM, 2018, pp. 105–108.

[23] N. Nassar, H. Radke, and T. Arendt, "Rule-based repair of emf models: An automated interactive approach," in *Theory and Practice of Model Transformation*. Springer, 2017, pp. 171–181.

[24] A. Nöhrer and A. Egyed, "Utilizing the relationships between inconsistencies for more effective inconsistency resolution." in *Workshop on Living with Inconsistencies in Software Development*, 2010, pp. 39–43.

[25] A. Nöhrer, A. Reder, and A. Egyed, "Positive effects of utilizing relationships between inconsistencies for more effective inconsistency resolution," in *33rd International Conference on Software Engineering (ICSE)*. ACM, 2011, p. 864–867.

[26] D. E. Khelladi, R. Kretschmer, and A. Egyed, "Detecting and Exploring Side Effects When Repairing Model Inconsistencies," in *12th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 2019, pp. 113–126.

[27] M. Ohrndorf, C. Pietsch, U. Kelter, L. Grunske, and T. Kehrer, "History-Based Model Repair Recommendations," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, Jan. 2021.

[28] A. Barriga, A. Rutle, and R. Heldal, "Improving model repair through experience sharing." *J. Object Technol.*, vol. 19, no. 2, pp. 13–1, 2020.

[29] R. Kretschmer, D. E. Khelladi, and A. Egyed, "Transforming abstract to concrete repairs with a generative approach of repair values," *Journal of Systems and Software*, vol. 175, p. 110889, 2021.

[30] D. Torre, Y. Labiche, M. Genero, M. Elaasar, and C. Menghi, "Uml consistency rules: A case study with open-source uml models," in *8th International Conference on Formal Methods in Software Engineering*. ACM, 2020, p. 130–140.

[31] L. Marchezan, W. K. G. Assuncao, R. Kretschmer, and A. Egyed, "Change-oriented repair propagation," in *International Conference on Software and System Processes and International Conference on Global Software Engineering*. ACM, 2022, p. 82–92.

[32] OMG, "OCL Specification," http://www.omg.org/spec/OCL/, 2014.

[33] D. Torre, Y. Labiche, and M. Genero, "Uml consistency rules: A systematic mapping study," in *International Conference on Evaluation and Assessment in Software Engineering (EASE)*. ACM, 2014.

[34] OMG, "UML 2.5.1 Specification," https://www.omg.org/spec/UML/, 2017.

[35] H. Wu, "Maxuse: A tool for finding achievable constraints and conflicts for inconsistent uml class diagrams," in *Integrated Formal Methods*. Cham: Springer International Publishing, 2017, pp. 348–356.

[36] G. Soltana, M. Sabetzadeh, and L. C. Briand, "Practical constraint solving for generating system test data," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 2, 2020.

[37] A. Egyed, "Instant Consistency Checking for the UML," in *28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 381–390.

[38] A. Barriga, R. Heldal, L. Iovino, M. Marthinsen, and A. Rutle, "An extensible framework for customizable model repair," in *23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 24–34.

[39] L. Marchezan, W. K. G. Assunção, E. Herac, S. Shafiq, and A. Egyed, "Exploring Dependencies Among Inconsistencies to Enhance Models Consistency Maintenance (Online Appendix)," 2023. [Online]. Available: https://sites.google.com/view/crdependenciessaner2023/

[40] E. Herac, W. Assunção, L. Marchezan, R. Haas, and A. Egyed, "A flexible operation-based infrastructure for collaborative model-driven engineering," vol. 22, no. 2, Jul. 2023, pp. 2:1–14, the 19th European Conference on Modelling Foundations and Applications (ECMFA 2023).

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

[42] S. Easterbrook, "Empirical research methods for software engineering," in *Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 574.

[43] L. Marchezan, W. K. G. Assunção, G. K. Michelon, and A. Egyed, "Do developers benefit from recommendations when repairing inconsistent design models? a controlled experiment," in *27th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2023, pp. 1–10.

[44] A. Reder and A. Egyed, "Determining the Cause of a Design Model Inconsistency," *Transaction on Software Engineering (TSE)*, 2013.
[45] R. Kretschmer, D. E. Khelladi, R. E. Lopez-Herrejon, and A. Egyed, "Consistent change propagation within models," *Software and Systems Modeling*, pp. 1–17, 2020.
[46] D. Torre, Y. Labiche, M. Genero, and M. Elaasar, "A systematic identification of consistency rules for uml diagrams," *Journal of Systems and Software*, vol. 144, pp. 121–142, 2018.
[47] C. Nentwich, W. Emmerich, and A. Finkelsteiin, "Consistency management with repair actions," in *International Conference on Software Engineering*, ser. ICSE '03. IEEE, 2003, pp. 455–464.
[48] M. A. Tröls, L. Marchezan, A. Mashkoor, and A. Egyed, "Instant and global consistency checking during collaborative engineering," *Software and Systems Modeling*, pp. 1–27, 2022.
[49] H. Wu, "Qmaxuse: A new tool for verifying uml class diagrams and ocl invariants," *Science of Computer Programming*, p. 102955, 2023.
[50] R. Clarisó, L. Burgueño, and J. Cabot, "Managing design-time uncertainty in ocl expressions," *Journal of Object Technology*, vol. 21, no. 4, pp. 4:1–10, Oct. 2022.
[51] R. Yan, C.-H. Cheng, and Y. Chai, "Formal consistency checking over specifications in natural languages," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015, pp. 1677–1682.
[52] V. Bertram, M. Boß, E. Kusmenko, I. H. Nachmann, B. Rumpe, D. Trotta, and L. Wachtmeister, "Neural language models and few shot learning for systematic requirements processing in mdse," in *15th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 2022, p. 260–265.
[53] I. Buzhinsky, "Formalization of natural language requirements into temporal logics: a survey," in *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, vol. 1, 2019, pp. 400–406.
[54] S. Salemi, A. Selamat, and M. Penhaker, "A model transformation framework to increase ocl usability," *Journal of King Saud University - Computer and Information Sciences*, vol. 28, no. 1, pp. 13–26, 2016.
[55] J. Cabot, D. Delgado, and L. Burgueño, "Combining OCL and Natural Language: A Call for a Community Effort," in *25th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. ACM, 2022, p. 908–912.