Detecting and Exploring Side Effects when Repairing Model Inconsistencies

Djamel Eddine Khelladi, Roland Kretschmer, Alexander Egyed

Detecting and Exploring Side Effects when Repairing Model Inconsistencies

Djamel Eddine Khelladi
CNRS UMR 6074, IRISA
35000, Rennes, France
djamel-eddine.khelladi@irisa.fr

Roland Kretschmer
Johannes Kepler University Linz, ISSE
4040 Linz, Austria
roland.kretschmer@jku.at

Alexander Egyed
Johannes Kepler University Linz, ISSE
4040 Linz, Austria
alexander.egyed@jku.at

Abstract

When software models change, developers often fail in keeping them consistent. Automated support in repairing inconsistencies is widely addressed. Yet, merely enumerating repairs for developers is not enough. A repair can as a side effect cause new unexpected inconsistencies (negative) or even fix other inconsistencies as well (positive). To make matters worse, repairing negative side effects can in turn cause further side effects. Current approaches do not detect and track such side effects in depth, which can increase developers' effort and time spent in repairing inconsistencies. This paper presents an automated approach for detecting and tracking the consequences of repairs, i.e. side effects. It recursively explores in depth positive and negative side effects and identifies paths and cycles of repairs. This paper further ranks repairs based on side effect knowledge so that developers may quickly find the relevant ones. Our approach and its tool implementation have been empirically assessed on 14 case studies from industry, academia, and GitHub. Results show that both positive and negative side effects occur frequently. A comparison with three versioned models showed the usefulness of our ranking strategy based on side effects. It showed that our approach's top prioritized repairs are those that developers would indeed choose. A controlled experiment with 24 participants further highlights the significant influence of side effects and of our ranking of repairs on developers. Developers who received side effect knowledge chose far more repairs with positive side effects and far less with negative side effects, while being 12.3% faster, in contrast to developers who did not receive side effect knowledge.

CCS Concepts • **Software and its engineering → Model-driven software engineering**; **Software maintenance tools**.

Keywords Model Inconsistencies, Repairs, Consequences, Side effects

1 Introduction

Model-Driven Engineering (MDE) has shown to be effective in the development and maintenance of large scale and embedded systems [25, 43]. *MDE* typically puts models as a central artifact in the various phases of the development process [24, 57]. Indeed, models are used in all development stages, from specifying the customer's requirements, design, all the way to source code, with the benefits of increased productivity and reduced time to market [1, 6, 64]. These benefits, however, hinge on the assumption that models remain consistent during development which is a problem when changes happen. Changes often do cause inconsistencies. If these inconsistencies are not recognized in a timely manner, then they cause subsequent errors. Moreover, if models are inconsistent, all automation (e.g., analysis [5] or model transformation [44]) using them is untrustworthy and likely causes even more errors. Therefore, inconsistencies must not only be detected but ultimately be repaired [12, 15, 64].

Detecting and repairing model inconsistencies is widely addressed in the literature (e.g., [22, 38, 45, 47, 52, 53, 62, 66], see related work section 6.). The existing approaches typically compute a set of repairs for each of the detected inconsistencies. Studies reported on the adoption of inconsistency detection and repair in various industries, such as by Thales a company in aeronautic, transport, and security [42], and Van Hoecke Automation a company in the areas of production automation and processing [12]. In this paper, we argue that merely proposing repairs is not yet sufficient, and consequences of repairs must be handled too. In addition to fixing a given inconsistency, repairs can also cause *side effects* that software developers are not aware of (empirical evidence is given later in this paper). A side effect of a given repair either can repair other inconsistencies (referred to as a *positive side effect*) or can cause new inconsistencies (referred to as a *negative side effect*). For example, with two conflicting simple consistency rules (imagine they are about the number of lectures a student can enroll in) $CR1[l > 5]$ and $CR2[l < 15]$. Repairing one inconsistency, e.g., of $CR1$ because $l = 2$, by changing l to $l = 20$, would cause another inconsistency (of $CR2$) as a negative side effect. To make matters worse, repairing a negative side effect can in turn cause additional side effects, e.g., $CR2$ by changing $l = 20$ to $l = 4$ which causes again $CR1$. This can lead to a path and a cycle of repairs causing successive negative side effects. A *path* is a sequence of repairs fixing their arising negative side effects. A *cycle* occurs when repairing a negative side effect causes a previously repaired inconsistency (i.e., a closed path), such as between $CR1$ and $CR2$. Not considering side effects can unfortunately increase developers' effort and time spent in repairing inconsistencies.

Furthermore, when repairs are computed they are typically not ranked w.r.t. to the likelihood of usefulness to developers. To the best of our knowledge, no existing approach (e.g., [22, 38, 45, 47, 52, 53, 62, 66]) explored side effects in depth nor ranked repairs based on the gained side effect knowledge when repairing model inconsistencies. Thus, leaving the burden of understanding and tracking the repairs consequences to the developers. Our novel approach fills this gap. Repairs causing side effects should not be filtered but rather considered and must be explored incrementally. Thus, guiding developers until the models are consistent.

Providing developers with information about side effects is crucial. On the one hand, developers can prioritize repairs with positive side effects, which reduces the repairing effort (i.e., 1 repair for n inconsistencies). On the other hand, developers can avoid, and most importantly be aware of, repairs with negative side effects and their possible paths and cycles. Ultimately, we aim to guide developers for a more efficient repairing process of inconsistencies. The novel contributions of this paper w.r.t. the state of the art are as follows:

- First, we identify positive side effects for each computed repair. This allows us to highlight to developers repairs that fix many inconsistencies at once.
- Second, we identify negative side effects for each computed repair. This allows us to warn developers about repairs that cause additional inconsistencies. We also explore the negative side effects recursively and thus guiding developers step by step in repairing them.
- Third, we identify paths and cycles of repairs caused by negative side effects. In so doing, we aim to prevent developers from being trapped in a cycle of repairs unknowingly.
- Fourth, we propose a ranking strategy of repairs based on the gained knowledge of side effects. This aims to rank repairs w.r.t. their likelihood of usefulness to developers.

We have empirically evaluated the performance, feasibility, and usefulness of our approach on 14 case studies. The evaluation results show that side effects not only exist but are frequent, where 398 repairs caused 12166 side effects and 151 cycles. Our evaluation also showed the usefulness of our ranking strategy as it correctly prioritized repairs that developers actually applied on our versioned models. Furthermore, we conducted a controlled experiment with 24 participants that highlighted a significant influence on developers when provided with side effect knowledge. Developers with side effect knowledge applied far more repairs with positive side effects (30) and far less with negative side effects (8), while being 12.3% faster. In contrast to developers without side effect knowledge who chose far less repairs with positive side effects (4) and far more with negative side effects (30).

2 Challenges and Motivation

To illustrate the issue of side effects when repairing model inconsistencies, we reuse the simple example of a *video on demand* (VOD) system from Egyed et al. [13] which is based on a client-server architecture. This example consists of three diagrams modeled with the Unified Modeling Language (UML) [51]. Figure 1 depicts the three UML diagrams of this system: class diagram, sequence diagram, and state chart diagram.

The class diagram represents the structure of a movie player. A `User` initiates the process of selecting and displaying a movie, a `Display` is used for visualizing movies and receiving user input and a `Streamer` is used for decoding movies. The sequence diagram describes the process of selecting and playing a movie, while showing the interaction between the instances of the classes `User`, `Display`, and `Streamer`. Here a user starts by selecting a movie (i.e., call of the operation `select`), and then the display starts streaming the movie (i.e., call of the operation `stream`). The `Streamer` object then sends frames to the `Display` instance via message `draw`. The streaming of the movie can also be paused with the message `pause` sent from `Display` to `Streamer`. The state chart diagram finally shows the states of the class `Streamer` in the VOD system, i.e., `wait` and `stream`.

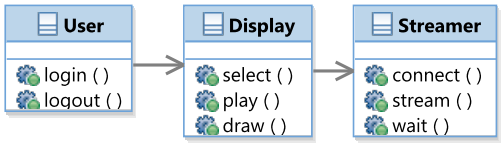
Inconsistencies can be repaired if and only if they are detected [55]. An established practice for detecting inconsistencies is to express consistency rules on models, which are the foundation to understand their failure, i.e., presence of an inconsistency. The consistency rules can be specified with the well-established standard Object Constraint Language (OCL) [49]. For the sake of simplicity, let us give an informal description of three structural consistency rules from the UML specification [50]:

CR1 Every transition in a state chart diagram has to have a corresponding message in a sequence diagram.
CR2 Every message in a sequence diagram has to have a corresponding operation in the class diagram.
CR3 Every transition in a state chart diagram has to have a corresponding operation in the class diagram.

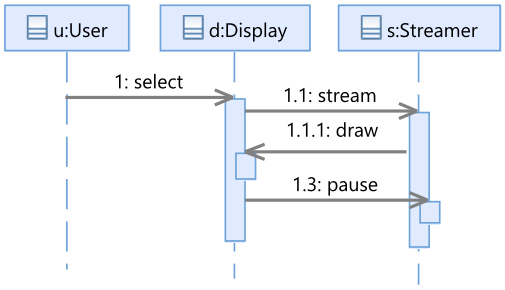
These consistency rules ensure that there are no undeclared transitions and messages. After checking the above consistency rules, two inconsistencies are detected, respectively:

VI Violation I of CR1 Transition `wait` has no corresponding message in lifeline `s:Streamer`.
VII Violation II of CR2 Message `pause` has no corresponding operation in class `Streamer`.

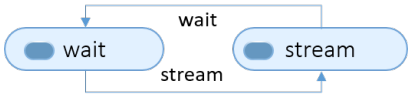
Alternative repairs can be proposed for the two inconsistencies `VI` and `VII` meeting different developers needs. For instance, adding a message `wait` would repair `VI`. However, among those possible repairs, some may have either positive or negative side effects. For example, to repair `VI` one could rename the message `pause` to `wait` so that the transition `wait` has a corresponding message. This repairs `VI`



(a) Class diagram.



(b) Sequence diagram.



(c) State chart diagram of the class Streamer.

Figure 1. Side effects in repairing model inconsistencies: An example.

but, as a positive side effect, would also repair VII since now all messages in the sequence diagram have corresponding operations in the class Streamer.

Another possible repair of VI could be to rename the transition wait to pause, so that the transition would have a corresponding message. This repair has an undesired negative side effect that causes a new inconsistency VIII (see below) which must be repaired as well. This leads to a path of two subsequent repairs, the first one fixing VI and the second one fixing its negative side effect VIII. To make matters worst, repairing VIII can in turn cause a negative side effect. For example, repairing VIII by renaming the transition pause to wait would again cause the inconsistency VI. This situation might then lead to a cycle of repairs that would be continuously executed. The developer would be trapped in a cycle that would repair VI and VIII where repairing one causes the other. In such a simple case, cycles could be easily handled. However, cycles of multiple repairs are an issue in particular when the models are worked on by multiple developers that are unaware of each others' changes.

VIII Violation III of CR3 Transition pause has no corresponding operation in class Streamer.

The example depicted in Figure 1 already shows non trivial side effects occurring with few simple consistency rules. These phenomena are amplified the more consistency rules are defined and the more inconsistencies are detected. In particular, when consistency rules are interrelated where the same model elements are used by multiple consistency rules, e.g., the model element *Transition* in CR1 and CR3. This is indeed very common in practice as was observed by Nöhrer et al. [48]. There, it was found that relationships among the consistency rules, and thus among inconsistencies, not only exist but are common [48].

Therefore, it becomes crucial to not simply list repairs but also to detect their side effects, paths, and cycles to better support developers when repairing inconsistencies with relevant feedback and guidance. This paper aims to fill this gap.

3 Background

This section provides definitions and examples of the most important terms for a prompt understanding of this paper.

Definition 1. *A model $\mathbb{M}$ consists of elements ($e \in \mathbb{M}$) having properties accessible with the dot (.) operator, e.g., "e.p".*

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

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

To detect inconsistencies, a given consistency rule is instantiated for each model element as the context of the consistency rules. For example, the consistency rule CR1 is instantiated for every transition wait and stream in the state machine diagram in Figure 1. We refer to such instances as validation trees which are defined as follows.

Definition 4. *A validation tree consists of a set of hierarchical ordered (tree-based) expressions and represents the Abstract Syntax Tree of a consistency rule for a specific model element. An expression consists of its operation (op) (e.g, logical AND ($\wedge$), equals (=)), one or more children (children) which are also expressions, exactly one parent expression (parent) and values to be validated (se2v).*

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

An example of a validation tree is shown in Figure 2 on the transition wait for the consistency rule CR1. Every validation tree validates to a boolean value in its root expression. Only those that validate to false are inconsistencies for which repairs are computed, e.g., the case transition wait in Figure 2. Note that the instances returning true are ignored when computing repairs but they are not removed because a repair could affect them as a side effect. The validation trees are used as a basis in the current approach to explore side effects. This will further be explained in the next Section 4.

Definition 5. *A repair action (ra) is a change to a model element property that resolves an inconsistency in part or full*

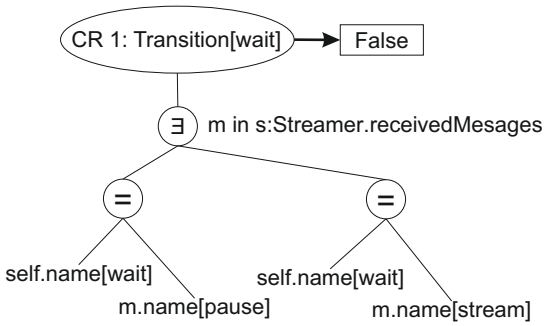


Figure 2. Example of validation tree of the transition `wait` for the Consistency rule `CR1`.

*(often multiple repair actions are needed to resolve an inconsistency). A repair action contains the model element (*e*), the property (*p*) that is affected by the change, the type of change (*ch*), and a value (*v*), which can be a model element $v \in \mathbb{M}$, or a primitive value $v \in \mathbb{V}$. The following types of changes are possible: $\oplus$ adds, $\ominus$ deletes, and $\odot$ updates.*

$$ra := \langle e, p, ch, v \rangle, ch \in \{\oplus, \ominus, \odot\},$$

Note that it might be necessary to change multiple scope elements to fix an inconsistency. For that purpose, we define a group of repair actions as follows.

Definition 6. *A repair is a non empty collection of repair actions that is guaranteed to fix an inconsistency (i).*

$$\langle i \in \mathbb{I}, ra \subseteq \mathbb{RA}_i \rangle$$

As repairs may have side effects, we define them as follows.

Definition 7. *A repair with a positive side effect (pse) is a repair that fixes more than one (≥ 2) inconsistency.*

Definition 8. *A repair with a negative side effect (nse) is a repair that causes at least one (≥ 1) new inconsistency.*

Note that a side effect can be caused by repairs from different inconsistencies. As repairing negative side effects can cause side effects, we define a graph of side effects.

Definition 9. *A side effect graph is a directed graph $G = (V, E)$ where V is the set of all vertices and E the set of all directed edges. Vertices represent repairs and edges represent side effects (positive(+) or negative(-)) a repair has on another inconsistency, and thus on its repairs. We define an edge as the triple $(v_1, v_2, +|-)$, where v_1 is the source vertex, v_2 is the target vertex and $+|-$ is the type of side effect (either positive (+) or negative(−)). Figure 3 shows an example of a constructed side effect graph.*

Definition 10. *A path of repairs causing negative side effects is a sequence of consecutive edges $((v_1, v_2, -), (v_2, v_3, -), \ldots, (v_x, v_y, -))$ which only consists of edges with negative side effects (−). The start edge $(v_1, v_2, -)$ and the end edge $(v_x, v_y, -)$ must not be the same ($(v_1, v_2) \neq (v_x, v_y)$, no cycles allowed).*

Definition 11. *A cycle of repairs causing negative side effects is a path $((v_1, v_2, -), \ldots, (v_x, v_y, -))$ where the start and end edge is the same ($(v_1, v_2) = (v_x, v_y)$).*

3.1 How do we Detect Inconsistencies?

In the validation tree, the root expression is expected to validate to `true` (i.e., consistent) and so its children expressions. For example, in Figure 2 the ∃ and = expressions 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.

3.2 How do we Generate Repair Actions?

Before to compute the repairs, we first need to identify the *cause* of an inconsistency. The *cause* is all scope elements that are part of a violated expression, i.e., where validation result differs from (≠) expected result. The *cause* in the example of Figure 2 is the transition `wait`, messages `pause` and `stream` and the lifeline `Streamer`.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 parent expression is validated.

4 Overall Approach

This section presents our approach that explores side effects when repairing inconsistencies, before to focus on evaluating it. First, it computes the repairs for the model inconsistencies. For that we use our previous work[1]. However, our current approach is designed to utilize repairs computed by any of the existing related approaches (in section VI). After that we identify the side effects for each computed repair and we explore the side effect recursively to identify paths and cycles of repairs. Finally, we propose to rank the repairs based on the gained side effect knowledge to better support developers in deciding how to repair their model inconsistencies.

4.1 Identifying Positive and Negative Side Effects

As introduced in Section 3, the validation trees are instantiations of consistency rules. Those that validate to false are treated as inconsistencies for which repairs are computed. For example, the instance of the consistency rule `CR1` (from section 2) on the transition *wait* validates to false since it has no corresponding message. Whereas, the instance of the consistency rule `CR3` on the transition *wait* validates to true since it has a corresponding operation.

Algorithm 1 presents our detection and exploration algorithm of side effects. It first computes a finite set of repairs

[1]We omit references to comply with the double blind rules.

Algorithm 1 Side effect detection

```
1:  function DETECTSIDEEFFECTS(Set[Inconsistency] si)
2:      sr ← si.generateRepairs4AllInconsistencies()
3:      for all repair ∈ sr do
4:          simulate(repair)
5:          for all vt ∈ AllOtherValidationTrees do
6:              if (vt.preResult() = false
7:                  ∧ vt.Result() = true) then
8:                  ▷ It is a positive side effect
9:                  repair.pse ← vt
10:             end if
11:             if (vt.preResult() = true
12:                 ∧ vt.Result() = false) then
13:                 ▷ It is a negative side effect
14:                 repair.nse ← vt
15:                 DetectSideEffects(repair.nse)
16:             end if
17:             jGraphT.updateGraph(repair)
18:         end for
19:     end for
20: end function
```

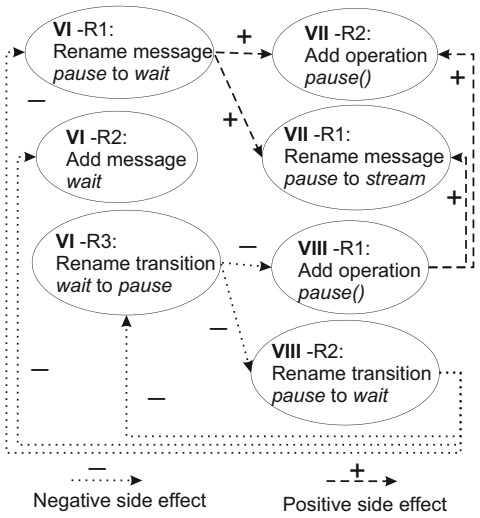


Figure 3. Excerpt of graph construction of side effects for the inconsistency VI in section 2.

for the inconsistencies, and then simulates on an internal copy of the model each repair on its inconsistency (lines 2-4). After that we monitor which other validation trees validate to a different boolean value than before (lines 5-17). If so, a side effect is detected and further investigated. Otherwise, the repair is considered as a side effect free. Depending on how the validation of an instance changes (*from true to false* or *from false to true*), either a positive or a negative side effect is detected (lines 4-8).

4.1.1 Positive Side Effects

In case the validation results of other validation trees change from false to true (after simulating a given repair), this is then identified as a positive side effect (lines 6-10). In the example shown in section 2, repairing the inconsistency VI by renaming the message pause to wait also repairs the inconsistency VII. This repair is then listed as causing a positive side effect along with its affected inconsistencies (line 9).

4.1.2 Negative Side Effects

In case the validation results of other validation trees change from true to false after simulating a given repair, this is then identified as a negative side effect (lines 11-16). Again, on the example shown in section 2, repairing the inconsistency VI by renaming the message wait to pause also causes the inconsistency VIII. This repair is then listed as causing a negative side effect along with its affected inconsistencies (line 14). Note that when a negative side effect is identified, i.e., a new inconsistency is caused, we compute its repairs and in turn we explore recursively their side effects (line 15). This is crucial in particular when exploring the negative side effects in depth and also identifying paths and cycles of repairs that the developer might be trapped in.

4.2 Paths and Cycles Detection

Already while detecting the side effects for each repair, we incrementally construct a graph that links the different repairs from the different inconsistencies (line 17). The graph consists of *nodes* that are repairs and *edges* linking repairs pairwise. An edge can be of type positive side effect or of type negative side effect, as defined in Definition 9. Figure 3 gives an excerpt example of a graph based on the inconsistencies and repairs from section 2. The first repair for the inconsistency VI has a positive side effect on the repairs for VII. Whereas the third repair for VI has a negative side effect on the repairs for VIII which in turn has positive and negative side effects on respectively VII and VI. Here, even though a repair has several arrows to other repairs from the same inconsistency, such as VI-R1 in Figure 3, we do not count each edge as a different side effect. Instead we count them as one side effect on the inconsistency, e.g., the repair VI-R1 as one positive side effect on VII in Figure 3.

Note that we also represent the repairs for the negative side effects, i.e., new inconsistencies that would arise after the execution of a given repair (not present in the initial state of the model). This can be seen as a projection of the future states of the model after executing a given repair.

This graph now is used as a basis to detect paths and cycles of repairs due to negative side effects. To do so, we use the *simple paths and cycles detection algorithms* from the open source library JGraphT (http://jgrapht.org/).

4.3 Ranking the Repairs Based on Side Effect Knowledge

In our approach, we further propose to sort the computed repairs based on the gained knowledge of side effects. Similar idea was applied by Cuadrado et al. [10] and showed to be useful. The rationale behind our ranking mechanism is: *i)* to reduce the developer effort and time when repairing inconsistencies by prioritizing repairs with positive side effects, *ii)* warn developers about the presence of negative side effects, paths and cycles, and *iii)* guide developers in repairing all inconsistencies, in particular repairing the negative side effects. Thus, we propose to rank the repairs *for each inconsistency* as the following:

1. The repairs with only positive side effects are prioritized. This category of repairs receive the priority *"very high"*. Note that herein the repairs are ranked from those with the most positive side effects to the least ones.
2. The repairs without side effects are then listed to the developer. This category of repairs receive the priority *"high"*. We rank first the repairs with the least repair actions (lowest effort) up to those with the most repair actions (highest effort).
3. After that, we list the repairs that have both positive and negative side effects. This category of repairs receive the priority *"medium"*. We compute the difference between the number of positive side effects and the number of negative side effects ($diff = \#pse - \#nse$). Then we rank the repairs based on their $diff$ value with a decreasing order (e.g., 10, 6, 2, -1, -3 ...). Thus, prioritizing repairs which cause more positive than negative side effects.
4. Finally, the repairs with only negative side effects are the least prioritized. This category of repairs receive the priority *"low"*. First, we rank repairs from those with the least to the most number of negative side effects. Second, we list the repairs that trigger a path/cycle of negative side effects. We rank them from the smallest (with the least number of repairs) to the longest paths/cycles.

Repairs would thus be sorted and presented per inconsistency for developers. An example of our ranking strategy on the three repairs for the inconsistency `VI` in Figure 1 is as follows:

▷ Inconsistency `VI`
#$R1$ Rename the message `pause` to `wait` – causes a positive side effect on `VII`.
#$R2$ Add a message `wait` – no side effects.
#$R3$ Rename the transition `wait` to `pause` – causes a negative side effect on `VIII`.

Note that in the end, only developers decide which repairs to apply. Repairs with negative side effects can be useful as well for developers. In that case, our approach ensures that developers track the side effects and repairs them later on.

5 Evaluation

This section evaluates our approach performance, feasibility and usefulness. Our evaluation consists of two parts.

At first, we assess the feasibility of our approach on 14 UML models (class, sequence, state chart, deployment, use case diagrams) taken from three different sources: *academia (VOD), industry (eBullition, MVC, Micro, ATM, Course, Planner, Dice3, Home, Robot, Vacation) and GitHub (Pro11, FullAdder, ActMgr)* [21]. The domains of the models range from control of a micro wave oven and an inventory storage management system to a Vacation and Sick Leave System. The model sizes range from 284 to 4485 model elements (e.g., class, lifeline, message, state, transitions, their properties, etc.). To detect inconsistencies, we used 20 complex consistency rules taken from the UML specification [50]. The number of inconsistencies ranges from 7 to 78, and the number of all repairs from 9 to 26682. Table 1 details our data set used in the evaluation, i.e., the number of model elements, number of inconsistencies, number of repairs, and their sources. Three of these models are from GitHub and have two versions each, where version one had inconsistencies that had been fixed in version two by a developer. This further allows us to assess the usefulness of our ranking strategy based on side effects, i.e. whether the manually applied repairs by the developers were among the prioritized ones with our ranking strategy or with the least prioritized. The time performance was also measured when detecting side effects in depth.

After that, we further assess with a controlled experiment the influence of the side effects that are detected by our approach on 24 participants divided in two groups. The first group receives side effect knowledge and ranked repairs (with our ranking strategy in section 4.3), and the second group not.

Note that our implementation has a compilation module integrated to check the syntactical correctness of the OCL consistency rules. The used dataset, consistency rules, and constructed side effects graphs can be found on our anonymized companion web page[2].

5.1 Research Questions

We formulate the following research questions:

RQ1 To what extent do positive side effects occur? This aims to investigate the frequency of positive side effects.
RQ2 To what extent do negative side effects occur? And what are the path lengths of negative side effects that occur recursively? This also aims to investigate the frequency of negative side effects and their consequences.

[2]https://figshare.com/s/e2c8c67bc94583897786

Table 1. Model information.

Model Name	#Model Elements	#Inconsistencies	#Repairs	Source
ActMgr	1185	7	9	GitHub
Pro11	284	7	18	GitHub
FullAdder	992	12	65	GitHub
Micro	2346	10	34	Industry
VOD	467	9	46	Academia
ATM	313	13	763	Industry
MVC	1410	58	3530	Industry
eBullition	1346	54	26682	Industry
Course	1620	42	318	Industry
Planner	868	17	73	Industry
Dice3	4485	71	131	Industry
Home	1882	60	578	Industry
Robot	1471	24	48	Industry
Vacation	1805	78	92	Industry

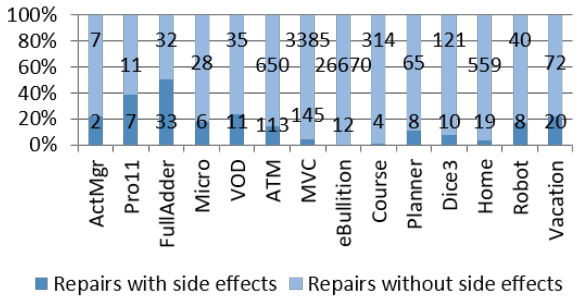


Figure 4. Number of repairs that do have side effects and that do not have side effects.

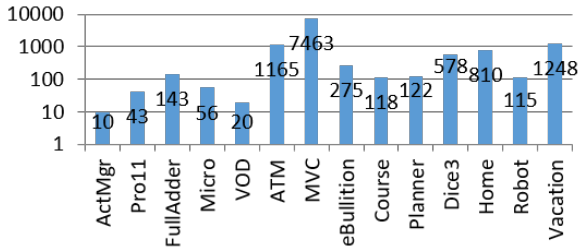


Figure 5. Number of all caused side effects.

RQ3 To what extent do cycles of repairs occur in our case studies? What is the amount of repairs in a cycle? This aims to investigate the frequency and nature of cycles.

RQ4 To what extent is our ranking strategy helpful for developers? This aims to assess the usefulness of our ranking strategy w.r.t. to the likelihood of usefulness to developers.

RQ5 How fast can we detect side effects, paths, and cycles? This aims to assess the time performances and scalability.

RQ6 To what extent does side effect knowledge influence developers when repairing inconsistencies? This aims to determine whether developers behave differently when side effects knowledge is available and when it is not. This also aims to further assess our approach's usefulness.

5.2 Results

We first report on the initialization phase. Detecting inconsistencies (i.e., creation of validation trees) took milliseconds per inconsistency. After computing repairs, we created the side effects graphs, which took minutes to generate (<1 minute for *VOD* to 10 minutes for *eBullition*). Now we report on the obtained results when tracking side effects in depth.

Figure 4 gives the number of repairs that do have side effects and repairs without side effects. The amount of repairs causing side effects varied from 2 (*ActivityManager* model) to 145 (*MVC* model) resulting in a total of 398 repairs with side effects in our case studies. Figure 5 further shows the numbers of all side effects for each case study[3]. The number of all side effects varied from 10 (*ActivityManager* model) to 7463 (*MVC* model) resulting in a total of 12166 side effects caused by 398 repairs. It is worth noting from Figures 4 and 5 that only few repairs in each case study tend to result in a large number of side effects. For example, in the *MVC* model only 145 repairs resulted in 7463 side effects.

5.2.1 RQ1

The number of repairs causing positive side effects varied from 2 (*ActivityManager* model) to 145 (*MVC* model) in our case studies, as shown in Figure 6. Figure 7 further shows the number of identified positive side effects in all case studies. The number of positive side effects varied from 4 (*VOD* model) to 7460 (*MVC* model). Moreover, columns 2 and 3 of Table 2 gives the minimum and maximum number of positive side effects caused by a single repair in each case study, which was a minimum of 1 (*MVC* model) and a maximum of 74 (*Vacation* model). This shows that positive side effects are frequent and could benefit developers, in reducing time and effort, if they would be prioritized over the rest of repairs.

[3]The number of side effects refers to the number of validation trees that are affected by a given repair, and not the graph edges.

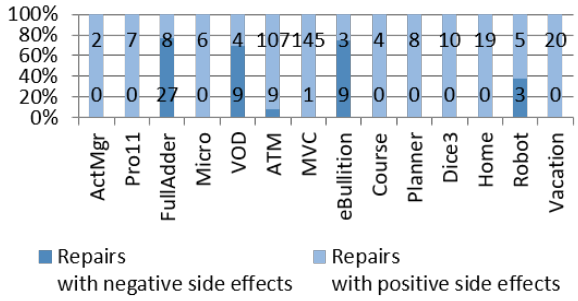


Figure 6. Number of repairs causing positive and negative side effects.

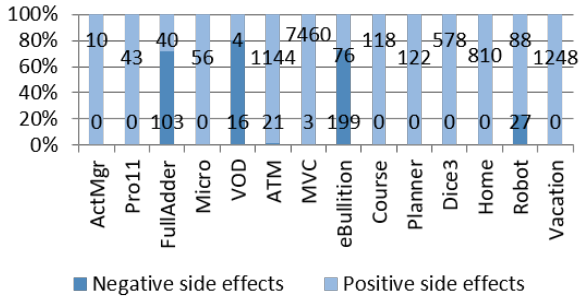


Figure 7. Ratio of positive and negative side effects.

5.2.2 RQ2

The number of repairs that caused negative side effects varied from 1 (*MVC* model) to 27 (*FullAdder* model) in our case studies, as shown in Figure 6. Among our case studies the eight models *ActivityManager*, *Pro11*, *Micro*, *Course, Planner, Dice3, Home, Vacation* did not have negative side effects. Among the rest of the case studies, the number of the identified negative side effects varied from 3 (*MVC* model) to 199 (*eBullition* model), as shown in Figure 7. We also observed the number of negative side effects caused by a single repair in each case study which was a minimum of 1 (e.g., *ATM* model) and a maximum of 32 (*eBullition* model), as in columns 4 and 5 in Table 2.

Moreover, for each negative side effect we computed their repairs and explored their side effects recursively. Some repairs in turn resulted in negative side effects themselves, and thus resulting in a path of successive repairs. The minimum and maximum path length of negative side effects were respectively 1 and 8 (*VOD* model), as shown in columns 6 and 7 in Table 2. This confirms that successive negative side effects occur in practice and thus it is crucial to highlight them to developers.

It is worth noting that we encountered repairs that caused both positive and negative side effects. In particular, *MVC*, *FullAdder*, *VOD*, and *ATM* respectively had one, two, two, and three repairs with both positive and negative side effects.

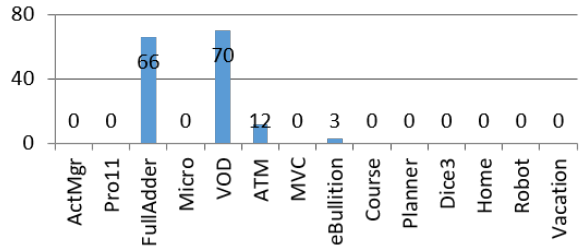


Figure 8. Cycles of negative side effects per case study.

5.2.3 RQ3

In addition to the identified paths of negative side effects, we detected 151 cycles in total varying from 3 (*eBullition* model) to 70 (*VOD* model), as shown in Figure 8. Similarly with the path length of negative side effects, we calculated the length of cycles which varied from 2 (minimum possible length of a cycle) to 6 (*VOD* model) repairs included in a cycle, as shown in columns 8 and 9 in Table 2. The results confirm that cycles are frequent, and hence, it is essential to detect them so that developers would not be trapped in them.

5.2.4 RQ4

To investigate the usefulness of our ranking strategy, we applied it to three versioned models (`ActMgr`, `Pro11`, and `FullAdder`) retrieved from GitHub. Every model contains inconsistencies in version 1 that the developer had manually fixed in version 2 (since the inconsistencies are not present anymore).We were able to extract the repairs performed by the developers. We thus could investigate if our approach would have prioritized those repairs with a high ranking.

We found that the applied repairs in `ActMgr`, `Pro11`, and `FullAdder` were all ranked in both the categories *very high* (with only positive side effects) and *high* (without side effects). In particular, 1) in `ActMgr` two repairs were applied where one caused positive side effects and one without side effects, 2) in `Pro11` three repairs without side effects were applied (herein not all inconsistencies were repaired in the second version), and 3) in `FullAdder` three repairs were applied where two caused positive side effects (category *high*) and one without side effects. This shows that all three developers applied only repairs with positive side effects and without side effects. Our ranking strategy thus would have met the developers needs by indeed prioritizing the manually applied repairs among all computed alternative repairs in our three versioned models. This shows evidence of the usefulness of our performed ranking. However, for statistical evidence, more versioned models are needed for further evaluation.

Table 2. Number of positive/negative side effects per repair, length of paths and cycles, and time performances.

1) Case studies	2) Max number of positive side effects per repair	3) Min number of positive side effects per repair	4) Max number of negative side effects per repair	5) Min number of negative side effects per repair	6) Max path length of negative side effects	7) Min path length of negative side effects	8) Max cycle length	9) Min cycle length	10) Time detection of paths /cycle [ms]
ActMgr	6	4	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Pro11	6	5	n/a	n/a	n/a	n/a	n/a	n/a	n/a
FullAdder	7	6	7	1	5	1	4	2	1 ms
Micro	9	6	n/a	n/a	n/a	n/a	n/a	n/a	n/a
VOD	2	2	3	1	8	1	6	2	3,9 ms
ATM	11	11	6	1	6	1	4	2	2,1 ms
MVC	54	1	3	3	1	1	0	0	0,3 ms
eBullition	30	16	32	1	4	1	2	2	64 ms
Course	34	20	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Planner	17	9	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Dice3	64	44	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Home	56	31	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Robot	19	16	16	1	2	1	0	0	4,5 ms
Vacation	74	44	n/a	n/a	n/a	n/a	n/a	n/a	n/a

5.2.5 RQ5

During evaluation we recorded time performances[4] while detecting side effects, paths, and cycles. We ran our evaluation on a desktop PC with an intel core i7-6700 3.4GHz Processor and 32GB of RAM on Windows 10. Each side effect took us less than 0,1 milliseconds to detect, even in the biggest models. As shown in column 10 of Table 2, detecting the paths and cycles due to successive negative side effects took us less than 65 milliseconds in the worst case. Hence, showing that our approach scales on large models with a large number of repairs and side effects.

5.3 RQ6 – Controlled Experiment

In *RQ4* we saw that the side effect knowledge showed to be useful in prioritizing the repairs that developers actually applied manually. However, that does not investigate the influence of side effects on developers when repairing inconsistencies. To gain more evidence we conducted a controlled experiment.

5.3.1 Experiment Set Up

Subjects selection. The controlled experiment was run with 24 participants (8 females and 16 males), all master students in computer science at JKU university. All participants had knowledge in the field of modeling and consistency checking. Five had no professional programming experience and 19 had between 1 and 14 years of programming experience. Overall, they had an average of 2 years and 4 months of experience as developers.

Experiment Design. Our participants were randomly assigned to a control and an experimental groups of 12 each. Hence, alleviating the threat of having all best participants in one group. Then, we checked that both groups were balanced w.r.t. to: *i)* professional programming experience, which was respectively 2.2 and 2.6 years, and *ii)* modeling experience which was respectively 2.4 and 2.5 years.

Experiment Task. We used a medium sized UML model consisting of Class, Sequence, and State chart diagrams, fitting for the limited time allocated for the experiment. This model is simple and similar to the VOD, yet exhibiting constructions present in our industry models. Each of the 24 participants had to repair a set of six inconsistencies from the three consistency rules shown in Section 2. Those inconsistencies are realistic as they occurred in our models from industry and their consistency rules are from the UML specification [50]. For each inconsistency, a set of repairs was provided (a minimum of 8 repairs and a maximum of 14 repairs) from which one had to be chosen. A total of 68 repairs is given for the six inconsistencies. Among the 68 repairs, 12 have positive side effects, 24 have negative side effects, and 32 have no side effects. Thus, for the 24 participants and six inconsistencies, we had to analyze 144 chosen

[4]Not including the initialization phase, i.e., after generating the validation trees, repairs and side effects graphs (10min for the largest model). Nonetheless, the whole process of detecting inconsistencies, computing repairs with their side effects, summed together, remains a matter of couple of minutes for the largest models.

Table 3. Mean time spent to repair the inconsistencies.

Groups	Repair Time	Difference
With side effect knowledge	13 min 40 sec	1 min 55 sec
Without side effect knowledge	15 min 35 sec	

repairs. The experimental group had side effect knowledge with ranked repairs (as described in section 4.3). The control group had not. The experiment thus set the initial condition to explore whether side effect knowledge can make a difference when repairing inconsistencies. At the end we gave a questionnaire about the complexity of tasks, and feedback on usefulness.

Variables. The experiment was aiming to measure the influence on subjects resolving inconsistencies when side effect knowledge was provided in contrast to when it was not. This was the *independent variable* we controlled, i.e., presence or not of side effects knowledge. To measure its effect, we observed two *dependent variables*, namely: the *time* developers took to repair the inconsistencies and the *types* of the chosen repairs, i.e., repairs with positive side effects, without side effects, and with negative side effects.

5.3.2 Experiment Results

We first checked for normality using the Shapiro-Wilk test [58]. The recorded times passed the normality test. Table 3 shows the recorded mean time it took developers from both groups to repair all given inconsistencies. We observed that developers with side effect knowledge performed faster than those without, with a difference of roughly 2 minutes, i.e., 12.3% faster. This shows that relying on side effect knowledge seems to accelerate the decision time when repairing inconsistencies.

Moreover, from the 144 selected repairs we analyzed the number of repairs for the three categories, namely: with positive, with negative, and without side effects, as shown in Figure 9. On the one hand, developers with side effect knowledge favored 30 repairs with positive side effects and only selected 8 repairs with negative side effects, but repaired the new inconsistencies afterward. All of the 12 developers here applied at least 2 (and some applied at most 4) repairs with positive side effects. Only eight developers knowingly applied exactly 1 repair with negative side effects. On the other hand, developers without side effect knowledge chose only 4 repairs with positive side effects and selected 30 repairs with negative side effects, but did not repair the new inconsistencies since they were not even aware of them. Here all of the 12 developers applied at least 2 (and some applied at most 3) repairs with negative side effects. Only two developers unknowingly applied exactly 2 repairs with positive side effects. The number of repairs without side effect was relatively the same in both groups (34 vs 38). We also checked

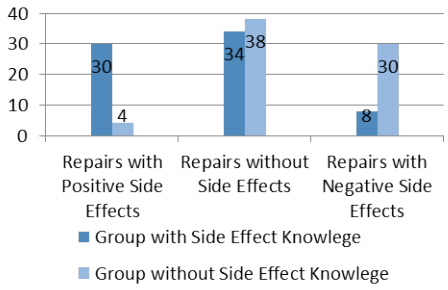


Figure 9. Number of selected repairs with positive, negative and without side effects in both groups.

the position of the selected repairs. We observed that they were from different positions, and were not simply in the first or second position, which indicate a non-random choice of repairs. These results highlight the significant benefit and influence of side effect knowledge to developers, where the number of repairs with positive and negative side effects are proportionally inversed in both groups.

We further gave a questionnaire to the participants. Let us highlight the main observations. In the group that did not have side effect knowledge, (to the question *'Did you consider side effects when repairing the inconsistencies?'*) seven out of the 12 developers answered that they have considered possible side effects when selecting a repair. However, as shown in Figure 9, they nonetheless applied few repairs with positive side effects and many repairs with negative side effects. They failed in correctly tracking side effects, and hence, this emphasizes the necessity of an automatic detection and tracking approach of side effects. In the other group that did have side effect knowledge, (to the question *'How useful was the ranking of repairs based on side effect knowledge to choose a repair?'*[5]) six out of the 12 developers graded our ranking strategy as 'useful', four graded it as 'very useful', and two graded it as 'extremely useful'. This further emphasizes the usefulness (discussed in *RQ4*) of ranking repairs based on side effect knowledge to developers. From our controlled experiment, we can conclude that the knowledge of side effects seems to positively influence the way developers repairs their inconsistencies, in both *time* and *type* of chosen repairs.

5.4 Threats to Validity

In this section we discuss internal, external and conclusion threats to validity according to Wohlin et. al. [65].

5.4.1 Internal Validity

In our evaluation, the internal threats to validity are centered on the used consistency rules and the amount of repairs. The number of detected side effects depends on the used

[5]Between 'useless – little useful– useful– very useful– extremely useful'.

consistency rules and to what extent they are interrelated. We provided our used consistency rules for reproducibility purposes. The number of side effects also depends on the number of considered repairs. However, considering more repairs would only increase the number of side effects, and not the opposite. In this paper, we aimed to show that side effects are frequent and that we are able to efficiently explore them in depth. This threat is acceptable here. Moreover, our approach depends on the validation trees which are instances of OCL consistency rules specified by the user. We only check their syntactic correctness, but not their semantic correctness/completeness since only the user knows its intent. This might affect the detection of side effects. However, we took our consistency rules from the UML specification [50]. In addition, by systematically simulating each repair and monitoring how the results of validation trees changes, we ensure the detection of all existing side effects.

5.4.2 External Validity

We implemented and evaluated our approach on UML/OCL and our previous work of repair computation. We cannot generalize our results to other modeling/constraint languages and other approaches of repair computation. However, the only requirement to reuse our approach is to have access to instances of consistency rules with an equivalence to the validation trees. Other constraint languages than OCL can be used. Moreover, as existing approaches computes similar repairs to ours (see definition 5). Reusing our detection algorithms of side effects, paths/cycles, and the ranking heuristic would be equally good as in the current approach. Nonetheless, further experimentation is necessary. In our experiment, we selected master students as participants. Recent studies [56, 61] showed that students can be valid and well representative subjects for experiments and development tasks. Nonetheless, to further reduce this threat, we selected master students in their final year. They had an average of 2 Years and 4 months professional programming experience, which is near to a junior developer experience.

5.4.3 Conclusion Validity

Our evaluation gave promising results (quantitatively and qualitatively), demonstrating that our detection algorithms of side effects and paths/cycles is very fast and efficiently explores repairs' side effects. The evaluation results also showed evidence that our ranking strategy showed to be useful in prioritizing the developers applied repairs. These results are further emphasized by our controlled experiment that showed the benefit in giving side effect knowledge to developers. However, we only evaluated on 14 models (3 versioned ones) with a total of 32387 repairs in all our case studies, and only 24 students in our experiment. To have more statistical evidence, we plan to evaluate on more (versioned) models and reproduce our experiment with more participants.

6 Related Work

This section focuses on works most related to the topic of detecting and repairing model inconsistencies and exploring side effect. Many approaches proposed to co-evolve models [7, 16, 23, 28, 29, 63], constraints [2, 8, 32, 35, 40], and model transformations [17, 18, 27, 36, 37, 41]. However, co-evolution approaches repair a particular type of inconsistencies w.r.t. metamodels conformance [19, 20]. Here we focus on model repair. All existing approaches to detect inconsistencies use various techniques. Briand et al. [3] proposed an approach to check UML consistency and by applying an impact analysis to identify consistencies in UML models. Konig et al. [39] proposed an algorithm for consistency checking on inter-related models to reduce cost of inconsistency detection due to model merging. There is also approaches that relies on formal methods to detect inconsistencies (e.g., [4, 30, 59]). However, those approaches do not propose repairs.

Nentwich et al. [47] provided an incremental approach for checking the consistency of distributed heterogeneous documents including models. They presented a method for generating repairs from the consistency rules. Xiong et al. [66] presented an approach that detects inconsistencies and allows users to predefine repair actions associated with each consistency rule. The consistency rules are defined with their proposed language Beanbag that defines a consistency relation between model elements similarly as in OCL. Kolovos et al. [38] proposed to address inconsistencies across heterogeneous models and provide repair strategies for those constraints. They established a classification of the different types of relationships that exist between heterogeneous models and to identify the types of inconsistencies each relationship suffers from. Da Silva et al. [11] propose to compute repairs to resolve inconsistencies in their UML models. However, valid repairs that do have side effects are filtered. We argue that all valid repairs must be considered and side effects of a chosen repair by the developer should be repaired incrementally. Thus, guiding the engineer until the models are consistent once again. Hegedues et al. [22] defined in a graphical notation consistency rules and uses a Constraint Satisfaction Problem solver to suggest repairs for inconsistencies for Domain Specific Modeling Languages. Egyed et al. [14] proposed an automated approach for detecting and tracking inconsistencies in design models in real time. They do this by observing the behavior of consistency rules to understand how they are affected by model changes. Reder et al. [54] proposed an incremental consistency checking approach. Reder et al. [53] further generate repairs for inconsistencies while pinpointing exactly which parts of an inconsistency must be repaired. Macedo et al. [45] rely on `Alloy` [26] to generate minimal repairs for inter-related models. Puissant et al. [52] 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. Taentzer et al. [62] proposed to repair inconsistent models w.r.t. their metamodels. They do not explore all possible repairs by relying on the model change history which helps in reducing the amount of repairs.

Although, the above existing approaches [22, 38, 45, 47, 52, 53, 62, 66] provide repairs to developers, no approach proposed to explore in depth the side effects of the computed model repairs, as it is achieved in this paper. Briand et al. [3], used the term of side effect. However, only in the sense of identifying inconsistencies caused by model changes. Reder et al. [53] identify positive side effects by computing the intersection between repairs of different inconsistencies. However, they did not explore negative side effects. To the best of our knowledge, no related work in model repair addressed explicitly and completely the issue of exploring side effects in depth and detecting paths and cycles. Developers are left with the burden of understanding and tracking the repairs consequences (what side-effects? where? which impacted elements?) without even being aware of them.

However, existing works on repairing programs and transformations do explore side effects. Muslu et al. [46] proposed to detect consequences of the code quick fixes but without exploring paths and cycles. Steimann et al. [60] proposed to repair malformd programs. They explored the consequences (side effects) of the repairs successively, i.e., compute in advance paths of repairs. However, they do not explore nor detect the cycles of repairs. Both Muslu et al. [46] and Steimann et al. [60] also do not rank the repairs as we do in this paper. Cuadrado et al. [9] proposed to compute quick fixes for ATL transformations. They also proposed in [10] to detect positive and negative side effects for each quick fix. The main difference with our work is they do not explore negative side effect in depth and do not detect paths and cycles. Moreover, they further rank the quick fixes in a similar way as our strategy, but without distinguishing fixes with negative side effects that lead to paths and cycles from those that do not. To the best our of knowledge all existing approaches of model repair compute repairs without knowing their side effects and without ranking them w.r.t the likelihood of usefulness to developers, as we do in this work.

7 Conclusion

In this paper, we presented an approach for detecting and tracking positive and negative side effects. Then, based on the successive negative side effects, we detect paths and cycles of repairs to better guide and support developers in repairing model inconsistencies. Finally, we rank the repairs w.r.t. their usefulness to developers. Ultimately, to guide developers for a more efficient repairing process of inconsistencies.

In our evaluation, we applied 20 consistency rules to 14 models. The results show that side effects not only exist but they tend to be frequent. In our case studies, 398 repairs caused 12166 side effects and 151 cycles in total. Furthermore, to check the usefulness of our ranking strategy, we used 3 versioned models retrieved from GitHub and found that the repairs manually applied by the developers were indeed among the highest prioritized. Thus showing the usefulness of our ranking strategy. A conducted controlled experiment highlighted the significant influence and useful value of providing developers with side effect knowledge and ranking repairs based on it. Developers with side effect knowledge preferred far more repairs with positive side effects (30 vs 4) and far less with negative side effects (8 vs 30) in contrast to developers without side effect knowledge. While they also required less effort and time to repair the given inconsistencies (12.3% faster).

For future work, we plan to propose heuristics that would break cycles. We also plan to propose other alternative ranking strategies that would meet different developers needs. Finally, we plan to investigate propagation-based repair of inconsistencies. With our detection approaches [31, 33, 34] of the models changes causing inconsistencies, we could propagate them to compute relevant repairs only. For example, it would allow us to not compute undo repairs.

Acknowledgment

We would like to thank all participants of the experiment study and our colleagues for the constructive feedback. The research leading to these results has received funding from the CNRS PEPS, and from the AIS Rennes Métropole under grant no. 190270. It is also funded by Pro2Future, a COMET K1-Centre of the Austrian Research Promotion Agency (FFG), grant no. 854184, the LIT Secure and Correct System Lab funded by the State of Upper Austria, and the Austrian Science Fund (FWF), grand no. P31989.

References

[1] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. 2007. UML2Alloy: A challenging model transformation. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 436–450.

[2] Edouard Batot, Wael Kessentini, Houari Sahraoui, and Michalis Famelis. 2017. Heuristic-Based Recommendation for Metamodel—OCL Coevolution. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 210–220.

[3] Lionel C Briand, Yvan Labiche, Leeshawn O'Sullivan, and Michal M Sówka. 2006. Automated impact analysis of UML models. *Journal of Systems and Software* 79, 3 (2006), 339–352.

[4] Jordi Cabot, Robert Claris, Daniel Riera, et al. 2008. Verification of UML/OCL class diagrams using constraint programming. In *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*. IEEE, 73–80.

[5] Jordi Cabot and Ernest Teniente. 2009. Incremental integrity checking of UML/OCL conceptual schemas. *Journal of Systems and Software* 82, 9 (2009), 1459–1478.

[6] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. 2008. Automating co-evolution in model-driven engineering. In *Enterprise Distributed Object Computing Conference, 2008. EDOC'08. 12th International IEEE*. IEEE, 222–231.

[7] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. 2008. Automating co-evolution in model-driven engineering. In *Enterprise Distributed Object Computing Conference, 2008. EDOC'08. 12th International IEEE*. IEEE, 222–231.

[8] Alexandre Correa and Cláudia Werner. 2007. Refactoring object constraint language specifications. *Software & Systems Modeling* 6, 2 (2007), 113–138.

[9] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2015. Quick fixing ATL model transformations. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 146–155.

[10] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2018. Quick fixing ATL transformations with speculative analysis. *Software & Systems Modeling* (2018), 1–35.

[11] Marcos Aurélio Almeida da Silva, Alix Mougenot, Xavier Blanc, and Reda Bendraou. 2010. Towards Automated Inconsistency Handling in Design Models. In *CAiSE*. 348–362. https://doi.org/10.1007/978-3-642-13094-6_28

[12] Andreas Demuth, Roland Kretschmer, Alexander Egyed, and Davy Maes. 2016. 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*. IEEE, 529–538.

[13] Alexander Egyed. 2006. Instant consistency checking for the UML. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*. 381–390. https://doi.org/10.1145/1134339

[14] Alexander Egyed. 2011. Automatically Detecting and Tracking Inconsistencies in Software Design Models. *IEEE Trans. Software Eng.* 37, 2 (2011), 188–204. https://doi.org/10.1109/TSE.2010.38

[15] William B Frakes and Kyo Kang. 2005. Software reuse research: Status and future. *IEEE transactions on Software Engineering* 31, 7 (2005), 529–536.

[16] Kelly Garcés, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. 2009. Managing model adaptation by precise detection of metamodel changes. In *Model Driven Architecture-Foundations and Applications*. Springer, 34–49.

[17] Kelly Garcés, Juan M Vara, Frédéric Jouault, and Esperanza Marcos. 2014. Adapting transformations to metamodel changes via external transformation composition. *Software & Systems Modeling* 13, 2 (2014), 789–806.

[18] Jokin García, Oscar Diaz, and Maider Azanza. 2013. Model transformation co-evolution: A semi-automatic approach. *SLE* 7745 (2013), 144–163.

[19] Regina Hebig, Djamel Eddine Khelladi, and Reda Bendraou. 2015. Surveying the corpus of model resolution strategies for metamodel evolution. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 135–142.

[20] Regina Hebig, Djamel Eddine Khelladi, and Reda Bendraou. 2016. Approaches to co-evolution of metamodels and models: A survey. *IEEE Transactions on Software Engineering* 43, 5 (2016), 396–414.

[21] Regina Hebig, Truong Ho Quang, Michel RV Chaudron, Gregorio Robles, and Miguel Angel Fernandez. 2016. The quest for open source projects that use UML: mining GitHub. ACM, 173–183.

[22] Ábel Hegedüs, Ákos Horváth, István Ráth, Moisés Castelo Branco, and Dániel Varró. 2011. Quick fix generation for DSMLs. In *VL/HCC*. 17–24. https://doi.org/10.1109/VLHCC.2011.6070373

[23] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. 2009. COPE-automating coupled evolution of metamodels and models. In *ECOOP 2009–Object-Oriented Programming*. Springer, 52–76.

[24] John Hutchinson, Mark Rouncefield, and Jon Whittle. 2011. Model-driven engineering practices in industry. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 633–642.

[25] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. 2011. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 471–480.

[26] Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11, 2 (2002), 256–290.

[27] Wael Kessentini, Houari Sahraoui, and Manuel Wimmer. 2018. Automated Co-evolution of Metamodels and Transformation Rules: A Search-Based Approach. In *International Symposium on Search Based Software Engineering*. Springer, 229–245.

[28] Wael Kessentini, Houari Sahraoui, and Manuel Wimmer. 2019. Automated metamodel/model co-evolution: A search-based approach. *Information and Software Technology* 106 (2019), 49–67.

[29] Wael Kessentini, Manuel Wimmer, and Houari Sahraoui. 2018. Integrating the Designer in-the-loop for Metamodel/Model Co-Evolution via Interactive Computational Search. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. ACM, 101–111.

[30] Djamel-Eddine Khelladi, Reda Bendraou, Souheib Baarir, Yoann Laurent, and Marie-Pierre Gervais. 2015. A framework to formally verify conformance of a software process to a software method. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, 1518–1525.

[31] Djamel Eddine Khelladi, Reda Bendraou, and Marie-Pierre Gervais. 2016. Ad-room: a tool for automatic detection of refactorings in object-oriented models. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 617–620.

[32] Djamel Eddine Khelladi, Reda Bendraou, Regina Hebig, and Marie-Pierre Gervais. 2017. A semi-automatic maintenance and co-evolution of OCL constraints with (meta) model evolution. *Journal of Systems and Software* 134 (2017), 242–260.

[33] Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou, Jacques Robin, and Marie-Pierre Gervais. 2015. Detecting complex changes during metamodel evolution. In *International Conference on Advanced Information Systems Engineering*. Springer, 263–278.

[34] Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou, Jacques Robin, and Marie-Pierre Gervais. 2016. Detecting complex changes and refactorings during (meta) model evolution. *Information Systems* 62 (2016), 220–241.

[35] Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou, Jacques Robin, and Marie-Pierre Gervais. 2016. Metamodel and constraints co-evolution: A semi automatic maintenance of ocl constraints. In *International Conference on Software Reuse*. Springer, 333–349.

[36] Djamel Eddine Khelladi, Roland Kretschmer, and Alexander Egyed. 2018. Change Propagation-based and Composition-based Co-evolution of Transformations with Evolving Metamodels. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. ACM, 404–414.

[37] Djamel Eddine Khelladi, Horacio Hoyos Rodriguez, Roland Kretschmer, and Alexander Egyed. 2017. An Exploratory Experiment on Metamodel-Transformation Co-Evolution. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 576–581.

[38] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. 2008. Detecting and Repairing Inconsistencies across Heterogeneous Models. In *ICST*. 356–364. https://doi.org/10.1109/ICST.2008.23

[39] Harald König and Zinovy Diskin. 2017. Efficient Consistency Checking of Interrelated Models. In *European Conference on Modelling Foundations and Applications*. Springer, 161–178.

[40] Angelika Kusel, Juergen Etzlstorfer, Elisabeth Kapsammer, Werner Retschitzegger, Johannes Schoenboeck, Wieland Schwinger, and Manuel Wimmer. 2015. Systematic Co-Evolution of OCL Expressions. In *11th APCCM 2015*, Vol. 27. 30.

[41] Angelika Kusel, Jurgen Etzlstorfer, Elisabeth Kapsammer, Werner Retschitzegger, Wieland Schwinger, and Johannes Schonbock. 2015. Consistent co-evolution of models and transformations. In *ACM/IEEE 18th MODELS*. 116–125.

[42] Jerome Le Noir, Olivier Delande, Daniel Exertier, Marcos Aurélio Almeida da Silva, and Xavier Blanc. 2011. Operation based model representation: experiences on inconsistency detection. In *European Conference on Modelling Foundations and Applications*. Springer, 85–96.

[43] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. 2014. Assessing the state-of-practice of model-based engineering in the embedded systems domain. In *Model-Driven Engineering Languages and Systems*. Springer, 166–182.

[44] Levi Lúcio, Moussa Amrani, Jürgen Dingel, Leen Lambers, Rick Salay, Gehan MK Selim, Eugene Syriani, and Manuel Wimmer. 2016. Model transformation intents and their properties. *Software & systems modeling* 15, 3 (2016), 647–684.

[45] Nuno Macedo, Tiago Guimaraes, and Alcino Cunha. 2013. Model repair and transformation with Echo. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 694–697.

[46] Kıvanç Muşlu, Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. 2012. Speculative analysis of integrated development environment recommendations. *ACM SIGPLAN Notices* 47, 10 (2012), 669–682.

[47] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. 2003. Consistency Management with Repair Actions. In *ICSE*. 455–464. http://computer.org/proceedings/icse/1877/18770455abs.htm

[48] Alexander Nöhrer, Alexander Reder, and Alexander Egyed. 2011. Positive effects of utilizing relationships between inconsistencies for more effective inconsistency resolution (NIER track). In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 864–867.

[49] OMG. 2014. Object Constraint Language. http://www.omg.org/spec/OCL/2.4

[50] OMG. 2015. Object Management Group. Unified Modeling Language (UML). http://www.omg.org/spec/UML/.

[51] OMG. 2015. Unified Modeling Language. http://www.omg.org/spec/UML/2.5

[52] Jorge Pinna Puissant, Ragnhild Van Der Straeten, and Tom Mens. 2015. Resolving model inconsistencies using automated regression planning. *Software & Systems Modeling* 14, 1 (2015), 461–481.

[53] Alexander Reder and Alexander Egyed. 2012. Computing repair trees for resolving inconsistencies in design models. In *ASE*. 220–229. https://doi.org/10.1145/2351676.2351707

[54] Alexander Reder and Alexander Egyed. 2012. Incremental Consistency Checking for Complex Design Rules and Larger Model Changes. In *MODELS*. 202–218. https://doi.org/10.1007/978-3-642-33666-9_14

[55] Alexander Reder and Alexander Egyed. 2013. Determining the Cause of a Design Model Inconsistency. *IEEE Trans. Software Eng.* 39, 11 (2013), 1531–1548. https://doi.org/10.1109/TSE.2013.30

[56] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. 2015. Are students representatives of professionals in software engineering experiments?. In *ICSE-Volume 1*. IEEE Press, 666–676.

[57] Douglas C Schmidt. 2006. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-* 39, 2 (2006), 25.

[58] Samuel Sanford Shapiro and Martin B Wilk. 1965. An analysis of variance test for normality (complete samples). *Biometrika* 52, 3/4 (1965), 591–611.

[59] Colin Snook and Michael Butler. 2006. UML-B: Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 15, 1 (2006), 92–122.

[60] Friedrich Steimann, Jörg Hagemann, and Bastian Ulke. 2016. Computing repair alternatives for malformed programs using constraint attribute grammars. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 711–730.

[61] Mikael Svahnberg, Aybüke Aurum, and Claes Wohlin. 2008. Using students as subjects-an empirical evaluation. In *2nd ESEM*. ACM, 288–290.

[62] Gabriele Taentzer, Manuel Ohrndorf, Yngve Lamo, and Adrian Rutle. 2017. Change-Preserving Model Repair. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 283–299.

[63] Guido Wachsmuth. 2007. Metamodel adaptation and model co-adaptation. In *ECOOP*. Springer, 600–624.

[64] Jon Whittle, John Hutchinson, and Mark Rouncefield. 2014. The state of practice in model-driven engineering. *IEEE software* 31, 3 (2014), 79–85.

[65] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[66] Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Hui Song, Masato Takeichi, and Hong Mei. 2009. Supporting automatic model inconsistency fixing. In *ESEC FSE*. 315–324. https://doi.org/10.1145/1595696.1595757