

Positive Effects of Utilizing Relationships Between Inconsistencies for more Effective Inconsistency Resolution (NIER Track)

Alexander Nöhrer
Johannes Kepler University
Linz, Austria
alexander.noehrer@jku.at

Alexander Reder
Johannes Kepler University
Linz, Austria
alexander.reder@jku.at

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

ABSTRACT

State-of-the-art modeling tools can help detect inconsistencies in software models. Some can even generate fixing actions for these inconsistencies. However such approaches handle inconsistencies individually, assuming that each single inconsistency is a manifestation of an individual defect. We believe that inconsistencies are merely expressions of defects. That is, inconsistencies highlight situations under which defects are observable. However, a single defect in a software model may result in many inconsistencies and a single inconsistency may be the result of multiple defects. Inconsistencies may thus be related to other inconsistencies and we believe that during fixing, one should consider clusters of such related inconsistencies. This paper provides first evidence and emerging results that several inconsistencies can be linked to a single defect and show that with such knowledge only a subset of fixes need to be considered during inconsistency resolution.

Categories and Subject Descriptors: I.6.4 Simulation and Modeling: Model Validation and Analysis

General Terms: Algorithms, Human Factors, Verification.

Keywords: User Guidance, Clustering, Inconsistencies

1. INTRODUCTION

Model inconsistencies are violations of design rules and other constraints. Many technologies exist for automatically detecting inconsistencies [9, 4]. However, today fixing those inconsistencies is a non-trivial task that requires extensive human-intervention (i. e., software engineers). To reduce the complexity of such fixing tasks on the engineer's side we previously proposed [11] to look at inconsistencies as symptoms of defects rather than defects themselves. The idea was to utilize relationships between inconsistencies for a more effective inconsistency resolution. This basic idea is not entirely new and fairly established in the compiler community dating back as early as 1982 when Johnson and Runciman wrote:

“It is important to distinguish between an error diagnosis and error reporting. Correct error diagnosis must rely upon the programmer as it may depend upon intentions that are not expressed in his program. The compiler's job is correct error reporting using a form and content of reports most likely to help the programmer in error diagnosis. We can compare error reports to the symptoms of a sick patient: the location at which the error is detected is not necessarily its source.” [8].

Inconsistencies in the modeling world are quite analogous to compile-time errors in source code. In that spirit, inconsistencies are the observable symptoms caused by defects. Fixing design models should thus not be seen as the fixing of inconsistencies (i. e., curing the symptoms) but rather identifying and fixing the cause(s) for the inconsistencies (i. e., the causes being the defects; perhaps caused by different stakeholder opinions [3]). Since a defect may cause multiple inconsistencies (or none if not observable), it is not sufficient for the engineer to identify the defects by exploring the fixes of individual inconsistencies – though one of these fixes (if computed correctly) must inevitably also involve the defect.

Software models typically contain many defects but current state-of-the-art focuses on detecting and correcting inconsistencies. The issue that inconsistencies are not self-contained is largely ignored in the literature but is of essential importance. It is far more important to fix the cause of an inconsistency than just the symptoms. After all, the goal of an engineer is not just to resolve one inconsistency at a time but in the end to get a consistent model with all defects having been identified and resolved. While inconsistencies can only be resolved by fixing the underlying defects, we must recognize that those defects may also have caused additional inconsistencies at other locations. In certain situations this could be reversed, meaning that several defects cause the same inconsistency. An example for such a situation would be a requirement change in an already consistent model. This requirement change should lead to several model changes where initial changes are likely to conflict with existing parts of the model. As a consequence the first change(s) introduce inconsistencies, though these changes would not be defects because they are the initial steps of a larger requirements change. It follows that other model elements need to be changed. Note that in this context, the term defect may be misleading because propagating a change is not the same thing as fixing a defect; however, the same principles apply and we consider them quite analogous.

Our working assumption is that the number of fixes that can be generated to resolve interrelated inconsistencies should

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA
Copyright 2011 ACM 978-1-4503-0445-0/11/05 ...\$10.00.

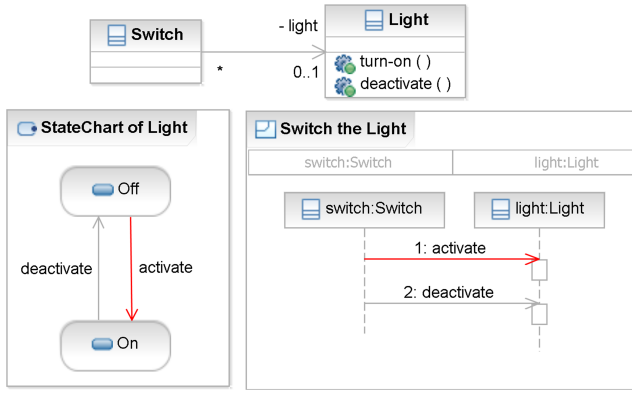


Figure 1: Two related Inconsistencies in a UML model of a Light and a Switch.

decrease with the number of inconsistencies involved because each inconsistency constrains the set of possible fixes. Considering the effects of multiple, interrelated inconsistencies should make it simpler for engineers to identify and fix defects in models or propagate changes. In this paper we presents emerging results concerning the feasibility of this proposed work:

1. How often do interrelated inconsistencies occur in real world examples?
2. How many choices for fixing an inconsistency can be excluded considering these interrelations? How strong is this reduction?

We provide evidence that inconsistencies are related rather with the help of fixing locations, which are the basis for every fix generation approach, than fixes themselves. We are not claiming that related inconsistencies always must have single defects, but we show that the number of fixing locations decreases under the assumption that one defect causes inconsistencies related through overlapping fixing locations. Even though the example and the emerging results provided in this work focus on UML models, we also have made similar experience with fixes in another domain – configuration scenarios in product lines.

This paper is structured as follows: In Section 2 we give a brief description of the scenario and problem we address. This is followed by the emerging results we have so far in Section 3. In Section 4 we discuss the state-of-the-art and related work. Finally we draw a conclusion and give an outlook to future work in Section 5.

2. SCENARIO AND PROBLEM

Inconsistencies can be resolved through different fixing actions where each fixing action involves one or more changes to model elements, we refer to those model elements as *fixing locations*. Figure 1 shows a simple UML model describing a **Light** with a **Switch** containing two inconsistencies. Those inconsistencies are the result of the two violated constraints stating that a state-chart transition must be defined as an operation in the owner’s class and that a message must also be defined as an operation in the receiver’s class – violated by both the **activate** message and transition because the class **Light** does not contain an operation of that name. Since

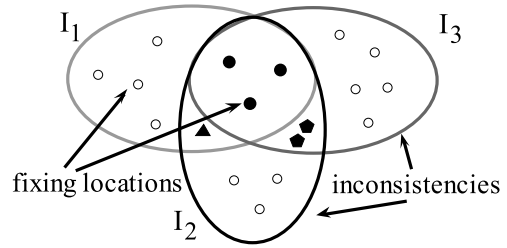


Figure 2: Fixing Locations for three Inconsistencies.

the state-chart describes the behavior of the class **Light** and the **activate** message is an invocation on an object of type **Light**, the consistency of both constraints is affected by the class **Light** and its operations – indicating an overlap between those inconsistencies. Examples resolving both inconsistencies based on the mentioned fixing location are: adding the operation **activate** to the class **Light**, or changing the name of the existing operation **turn-on** to **activate**.

Obviously the common fix may not be the correct one, depending on the engineer’s intentions, it could also be that changing the names of the **activate** message and transition to **turn-on** is the solution the engineer is looking for. How to distinguish such situations and provide the right guidance at the right time is not the focus of this paper, although the results will have an impact on how to proceed with further research into this area. Instead we present emerging results concerning the number of fixing locations to start out with, assuming that the correct fix is an overlapping one.

Figure 2 shows fixing locations (dots, triangle, and pentagons) belonging to three different inconsistencies (ellipses marked I_1 , I_2 , and I_3). Each inconsistency has a set of fixing locations and their overlap is defined by the intersection of those sets. Applying this simple set operator, depending on the knowledge which inconsistencies should be fixed or not, the fixing locations to start out with can be derived easily. For instance if inconsistencies I_2 and I_3 should be resolved but not I_1 , the fixing locations to start out with are represented by the pentagons in Figure 2.

The following emerging results try to answer the two research question, how often do interrelated inconsistencies occur in real world examples and how many choices for fixing an inconsistency can be excluded considering these interrelations as well as how strong this reduction is.

3. EMERGING RESULTS

Our emerging results are based on a set of four models from industrial partners. Due to proprietary information we are not allowed to present any specific details of the models. The model sizes range from 1,200 to 33,000 model elements containing several types of diagrams like class diagrams, sequence diagrams and state-charts as well as use-case diagrams. The used design rules check generic aspects of the underlying meta model such as the definition of messages and transitions as operations in the corresponding class of the class diagram, as well as the direction of the associations in the class diagram regarding the messages of the sequence diagram, and if the connected classifiers of the association ends are included in the namespace of the association. These rules are derived from larger rules used by these industrial partners (see [6] for a complete list).

Model	#Elements	#Incons.	#Overlapping	%
A	1282	36	29	80.5
B	2809	365	363	99.5
C	16255	489	487	99.6
D	33347	1271	1246	98.0

Table 1: Overview of Analyzed Models.

The results are based on the analysis of the accessed model elements and their properties, using Egyed’s approach [4]. Each evaluation of a design rule inspects various properties of different model elements and we call a single element of this set a fixing location (Egyed refers to them as scope elements in [4]). An evaluated design rule is either consistent or inconsistent. In our results only inconsistent design rules are considered. Side effects to other design rules, consistent ones included, are disregarded at this time and not subject of this paper, although of importance for further questions proposed for future work.

Table 1 provides some background about the models used for our evaluation. It shows the number of elements, how many inconsistencies were detected, how many of these inconsistencies have at least one fixing location in common with another inconsistency, and the percentage of such overlaps in inconsistencies. These results in Table 1 indicate that most of the inconsistencies found in these real world models have common fixing locations.

We conducted a detailed analysis of the overlapping inconsistencies for each model, where we counted the number of occurrences of different sized overlaps between inconsistencies. For instance counting these areas in Figure 2 would result in two occurrences of a size two overlap ($\{(I_1 \cap I_2) \setminus (I_1 \cap I_2 \cap I_3), (I_2 \cap I_3) \setminus (I_1 \cap I_2 \cap I_3)\}$) and one occurrence of a size three overlap ($\{I_1 \cap I_2 \cap I_3\}$). The size of an overlap is determined by the number of inconsistencies contributing to an overlapping area. Although the percentage of overlaps was very high in the analyzed models (see Table 1), the composition clearly showed that most of the overlaps were of size two, especially with bigger models. These results thus answer our first question of how often interrelated inconsistencies in real world examples exist.

Having established that overlaps among inconsistencies are common, in the following we will try to answer the second question of how many choices for fixing an inconsistency can be eliminated by considering the effects of interrelated inconsistencies. We will try to answer this question indirectly, again with the help of fixing locations since choices for fixing an inconsistency are based on fixing locations. By focusing on fixing locations, we try to avoid any bias from different fix generation methods – particularly because at present state-of-the-art is only able to compute fixing locations that cause inconsistencies completely but not necessary all locations affected by fixes as this requires human intervention [10].

Figure 3 shows the average number of common fixing locations for the different overlap groups. Each group has several bars indicating the impact on the number of fixing locations by considering additional inconsistencies of the overlap. For overlaps of size two, there are two bars: the first accounting for the average number of fixing locations if both inconsistencies are considered separately (no overlap effect).

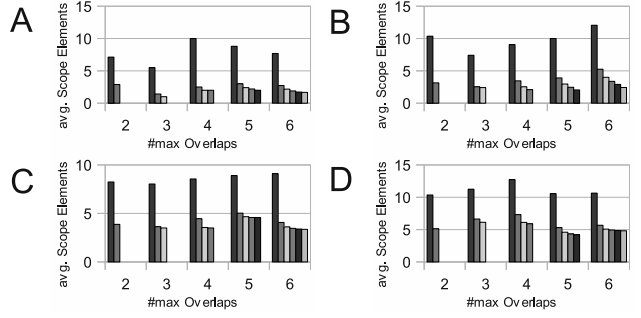


Figure 3: Reduction of Fixing Locations based on Overlap Size.

The second bar accounting for the average number of fixing locations if the overlap among these two inconsistencies is considered. We thus see the effect of no overlap vs. pairwise overlap next to each other. The same is valid for overlaps of bigger sizes though additional bars are used to indicate additional overlaps. For example, the fixing locations of three overlapping inconsistencies can be viewed individually (first bar), pairwise as in any two overlapping inconsistencies out of the three (second bar), and altogether (third bar). This is simple to calculate. For example, if we wanted to calculate these results for Figure 2, they would be the following:

- *Overlaps size two:* the first bar would be the result of $Average(|I_1|, |I_2|, |I_3|)$, the second bar the result of $Average(|I_1 \cap I_2|, |I_1 \cap I_3|, |I_2 \cap I_3|)$.
- *Overlaps size three:* since the same inconsistencies are involved in the overlap size three the first two bars would be the same, the third however would be the result of $Average(|I_1 \cap I_2 \cap I_3|)$.

These diagrams show consistently that the number of fixing locations is approximately reduced by half considering overlaps among two inconsistencies, but it also shows that this effect diminishes in larger groups – the limited subsequent effect may be due to the small group of inconsistencies considered.

In summary, we observe that clustering inconsistencies significantly reduces the number of fixing locations to consider. In all four models, we found that a clustering of two inconsistencies already reduces the set of possible fixing locations by half. By reducing the basis (the locations) on which fixing actions are being calculated, the number of actual fixing actions may even be reduced more drastically. For further evaluation it is necessary to include more design rules, perhaps also considering application or domain specific design rules as they are likely to further reduce the set of possible fixing locations. Note again, that this work presumes that we know of the existence of clusters of related inconsistencies to benefit from the reduction of fixing locations. What we have shown here is that there is a strong benefit in understanding clusters to reduce the complexity of fixing inconsistencies. Now that we know that research in this direction is useful, our next step is to devise methods for identifying such clusters. Furthermore, we plan on considering the side effects that a fix of an inconsistency might have onto other inconsistencies, as this will likely provide additional information on how to resolve an inconsistency.

4. RELATED WORK

Consistency management, especially inconsistency resolution, has received considerable attention in the last two decades. In this section we give a brief overview of work that has been done in this research area.

First of all, to resolve inconsistencies they have to be detected. However, the knowledge if the whole model or a single constraint is consistent, is not enough to produce fixes. As Nentwich et. al. for example stated in [9], it is important that trace links from the inconsistency to the model element(s) in question exist. In their work they propose to use first-order logic to express consistency rules and are able to provide trace links between inconsistent elements. Performance also is an issue when checking for consistency and approaches like the incremental consistency checking approach by Egyed [4] addresses this issue.

After being detected, it is beneficial to temporarily live with them [1], e. g. during the implementation of a requirements change. However, at some point those inconsistencies have to be resolved, preferably with the support of automated techniques.

For generating fixing or repair actions several approaches exist. On the one hand, Xiong et. al. propose writing additional “fixing procedures” for each constraint, in order to produce fixes when needed [13]. On the other hand Nentwich et. al. describe [10] a method for generating interactive repairs from first order logic formulae – the same formulae that they already used to detect inconsistencies [9]. Another approach described by Egyed et. al. [7] shows how to generate choices for fixing an inconsistency without having to understand such formulae which can be complex in case consistency rules are written in programming languages. These approaches look at other model elements already defined in the model and use them as choices. The generated choices are then reduced by looking at the impact of each choice [5, 2] and removing those that would cause additional inconsistencies. This can be problematic as already mentioned in the introduction because during fixing it could be necessary to introduce new inconsistencies temporarily (refactoring) [1]. Correspondingly, dismissing fixing actions because they cause new inconsistencies could be counterproductive. This is where our approach for reducing the fixing locations (model elements) via overlaps could help.

Despite the considerable progress on research for fixing inconsistencies, to the best of our knowledge no approach looks at more than one inconsistency at a time. However, the need for a more “global” approach during consistency checking itself is demonstrated by Sabetzadeh et. al. in [12] but not used for fixing yet. Additionally Nentwich et. al. already stated in their work [10], that one of the biggest challenges is not to look at one single inconsistency but to look at inconsistencies from a more “global” point of view. This notion is also in accordance with our vision that a more “global” view should be beneficial for fixing inconsistencies.

5. CONCLUSIONS AND FUTURE WORK

In this work, we presented emerging results on the positive effects of resolving clusters of inconsistencies at once. We see that by not looking at inconsistencies individually but in clusters of related inconsistencies we can reduce the number of possible fixing locations by at least half. We also showed that overlaps of fixing locations among inconsistencies not

only exist in real world examples but are the norm. This work will be used as a basis for future research in generating concrete fixes and more advanced concepts to improve the usability of modeling tools in general. It also raises questions regarding the circumstances when generating fixes based on related inconsistencies makes sense and is in the best interest of engineers.

Acknowledgments

We gratefully acknowledge the Austrian FWF (P21321-N15) and IBM (SRG-CAS-2010-04) for funding this work.

6. REFERENCES

- [1] R. Balzer. Tolerating Inconsistency. In *13th ICSE, Austin, Texas, USA*, pages 158–165, 1991.
- [2] L. C. Briand, Y. Labiche, and L. O’Sullivan. Impact Analysis and Change Management of UML Models. In *19th ICSM, Amsterdam, The Netherlands*, pages 256–265, 2003.
- [3] S. Easterbrook and B. Nuseibeh. Managing Inconsistencies in an Evolving Specification. In *2nd IEEE International Symposium on Requirements Engineering*, pages 48 – 55, 1995.
- [4] A. Egyed. Instant consistency checking for the UML. In *28th ICSE, Shanghai, China*, pages 381–390, 2006.
- [5] A. Egyed. Fixing Inconsistencies in UML Design Models. In *29th ICSE, Minneapolis, USA*, pages 292–301, 2007.
- [6] A. Egyed. Automatically Detecting and Tracking Inconsistencies in Software Design Models. *IEEE Transactions on Software Engineering*, 99(RapidPosts), 2010.
- [7] A. Egyed, E. Letier, and A. Finkelstein. Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In *23rd ASE, L’Aquila, Italy*, pages 99–108, 2008.
- [8] C. W. Johnson and C. Runciman. Semantic Errors - Diagnosis and Repair. In *SIGPLAN Symposium on Compiler Construction*, pages 88–97, 1982.
- [9] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Trans. Internet Techn.*, 2(2):151–185, 2002.
- [10] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency Management with Repair Actions. In *25th ICSE, Portland, Oregon, USA*, pages 455–464, 2003.
- [11] A. Nöhner and A. Egyed. Utilizing the Relationships Between Inconsistencies for more Effective Inconsistency Resolution. In *3rd Workshop on Living with Inconsistencies in Software Development, Colocated with ASE, Antwerp, Belgium*, pages 39–43. CEUR Workshop Proceedings Vol-661, 2010.
- [12] M. Sabetzadeh, S. Nejati, S. M. Easterbrook, and M. Chechik. Global consistency checking of distributed models with TReMer+. In *30th ICSE, Leipzig, Germany*, pages 815–818, 2008.
- [13] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei. Supporting Automatic Model Inconsistency Fixing. In *7th ESEC/FSE, Amsterdam, The Netherlands*, pages 315–324, 2009.