Utilizing the Relationships Between Inconsistencies for more Effective Inconsistency Resolution

Alexander Nöhrer
Institute for Systems Engineering and Automation
Johannes Kepler University Linz, Austria
alexander.noehrer@jku.at

Alexander Egyed
Institute for Systems Engineering and Automation
Johannes Kepler University Linz, Austria
alexander.egyed@jku.at

ABSTRACT

During software modeling, engineers are prone to making mistakes. State-of-the-art tool support can help detect these mistakes and point to inconsistencies in the model. They even can generate fixing actions for these inconsistencies. However state-of-the-art approaches process inconsistencies individually, assuming that each single inconsistency is a manifestation of an individual defect. This paper presents our vision of the next steps in inconsistency resolution. We believe that inconsistencies are merely expression 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 thus believe that during fixing, one should consider the clusters of such related inconsistencies. The main benefit of clustering inconsistencies is that it becomes easier to detect the defect the bigger the cluster. This paper discusses the idea in principle, provides some qualitative aspects of its benefit, and gives an outlook on how we plan to realize our vision.

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

General Terms: Algorithms, Human Factors, Verification.

Keywords: User Guidance, Grouping and Clustering, Inconsistencies

1. INTRODUCTION

C. W. Johnson and C. Runciman already wrote in 1982 [6] "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."

This is analogous to the modeling world, where inconsistencies are the symptoms (rules and/or constraints that are violated) which are caused by defects (the sources of symptoms that need fixing) in the model. It is thus the job of the designer to identify the defects by exploring the choices for fixing the inconsistencies – one of these choices (if complete) for each inconsistency inevitably must involve fixing a defect. Thus, inconsistencies are the sheer symptoms of a defect, but usually involve other model elements that when changed could also resolve the inconsistency. In accordance, fixing inconsistencies individually could mean fixing the symptoms only but not the defects (i. e., much like temperature-lowering medication merely "fixes" the symptom – the fever – but not the cause – an infection). Much like a good doctor attempts to identify all symptoms about a sickness to then hypothesize about the cause, a good software modeler should identify relationships among inconsistencies to reason about the cause(s) (defects) for these inconsistencies. Perhaps a key difference here: software models typically contain many defects.

A lot of research has been conducted to avoid and help detect and correct inconsistencies. The issue that inconsistencies are not self-contained is largely ignored in literature but of essential importance, it is far more important to fix the cause than just the symptoms. After all, the goal of engineers is not just to resolve one inconsistency at a time but in the end to get a consistent model. In order to get a consistent model, all defects have to be resolved. Of course some inconsistencies can only be resolved by fixing the underlying defects, but those defects often cause additional inconsistencies at other locations. In certain situations this even could be reversed, meaning that several defects cause the same inconsistency. An example for such an situation would be a requirement change in a already consistent model. This requirement change could require a set of model changes conflicting with the present requirements. As a consequence the first change would introduce an inconsistency without being the defect itself, instead the other model elements that are required to be changed are the defects. This also relates to the need of tolerating inconsistencies [1], since preventing them in this case would significantly change the typical work flow. So the challenge lies in determining where the defects are located and how to fix them, not just their symptoms, whilst not being too concerned with not causing new inconsistencies since they could be required to achieve the engineer's goals.

We present our vision and proposed approach of how to exploit interrelations between inconsistencies for resolving them in this paper. It is our believe that using this information will result in fewer, concrete fixes and provide guidance to engineers for locating the defects. Additionally we address open issues and discuss steps that in our opinion have to be taken to provide some insights in the qualitative aspects of this approach. However we are ignoring techniques for including semantic analysis of constraints and generelly information on how the inconsistencies were created for the

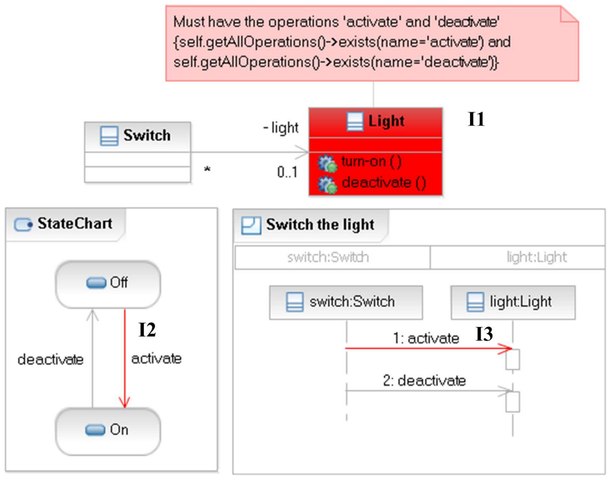

Figure 1: Several Inconsistencies in an UML model of a Light and a Switch

time being, at this point we just plan to investigate if inconsistencies are related and how this fact can be used to our advantage before combining it with other technologies.

This paper is structured as follows: In Section 2 we describe the scenario and problem we address. This is followed by the vision of how we want to tackle the problem in Section 3. In Section 5 we discuss the state-of-the-art and related work. In Section 4 we describe in detail how we plan to realize our vision. Finally we draw a conclusion and give an outlook to future work in Section 6.

2. SCENARIO AND PROBLEM

During modeling, engineers are prone to making mistakes. State-of-the-art tool support can help to detect these mistakes and point to inconsistencies in the model. For example, Figure 1 shows a simple UML model describing a `Light` with a `Switch` containing three inconsistencies:

I1 A violated model constraint (*C1*) that states that the `Light` class has to have at least two operations named `activate` and `deactivate`.

I2 A violated meta-model constraint (*C2*) that states that state-chart actions must be defined as an operation in the owner's class. In this case the owner of the state-chart class is `Light` and `Light` has no `activate` operation defined.

I3 A violated meta-model constraint (*C3*) that states that collaboration message actions must be defined as an operation in receiver's class. In this case the receiver's class is `Light` which has no `activate` operation defined.

Typical tool support for fixing these inconsistencies will only look at each inconsistency individually and generate fixing actions for each of them [4, 5, 8, 7]. Current state of the art has produced interesting solutions for suggesting fixes to inconsistencies. There is the pioneering work of Nentwich et al. [8] that demonstrated how to generate fixing actions for inconsistencies (one inconsistency at a time). They distinguished abstract and concrete fixes where abstract fixes in essence identified the locations where to fix (e.g., change the name of the `turn-on` method) and concrete actions in addition identify how to fix that location (e.g., change the name of the `turn-on` method to `activate`). However, the fixing of inconsistencies also has side effects onto other design constraints: negative side effects if the fixing causes new inconsistencies or positive side effects if the fixing of an inconsistency also fixes other inconsistencies. While the existence of these side effects has been widely published, to date they have not been exploited much. Existing state of the art, either attempts to minimize negative side effects (a heuristic that is not at all guaranteed to be the right strategy) or focuses only on visualizing them. If the goal is to avoid negative side effects (i. e., avoid additional inconsistencies) then possible fixes for the above model would be:

I1 The first inconsistency could be fixed by adding the operation `activate` to the class `Light` (*F1*), or changing its operation `turn-on` to `activate` (*F2*). Of course the model constraint *C1* could be changed to fit the model (*F3*).

I2 The second inconsistency would also be fixed with fixes *F1* and *F2*. Additional fixes would be to change the inconsistent action in the state-chart to `turn-on` (*F4*) or to `deactivate` (*F5*), or simply remove it (*F6*). And again also the meta-model constraint *C2* could be changed (*F7*).

I3 The third inconsistency again could be fixed with fixes *F1* and *F2*. Additional fixes would be to change the inconsistent message in the collaboration diagram to `turn-on` (*F8*) or to `deactivate` (*F9*), or simply remove it (*F10*). And of course the meta-model constraint *C3* could be changed (*F11*).

Looking at these inconsistencies individually results in several equally viable fixing actions. To choose one of those fixing actions for each individual inconsistency automatically is not reasonable as all are viable. Clearly, the decision on how to fix an individual inconsistency should be made by the software engineer. Since the example given is a fairly small and comprehensible model, it should be no challenge for an engineer to figure out that all inconsistencies can be resolved by either choosing fixing action *F1* or *F2*. In larger, more complex models, such "ideal" fixing actions cannot be identified manually as easily, and as a consequence automated support is needed. Nonetheless, even the "ideal" fixing actions with the least number of changes and/or the fewest negative side effects are not necessarily the fixes the software engineers intends. For example, if a model is fully consistent but a requirement change requires a set of model changes, then the first change likely causes inconsistencies because the set of model changes are not yet finished. The fix with the least number of changes and fewest negative side effects is then often a simple undo to restore the initial, correct state – a minimal, consistent solution, however, clearly an incorrect one with respect to the intended requirements change.

Even though this paperfocuses on modeling with the UML , we previously also investigated the concept of tolerating conflicts in other domains. In [9] we discussed different fixing strategies in the domain of decision-making and product-line engineering especially. From our experiences in these domains, the same basic principles discussed in this paper are valid. If decision makers pick conflicting decisions, those

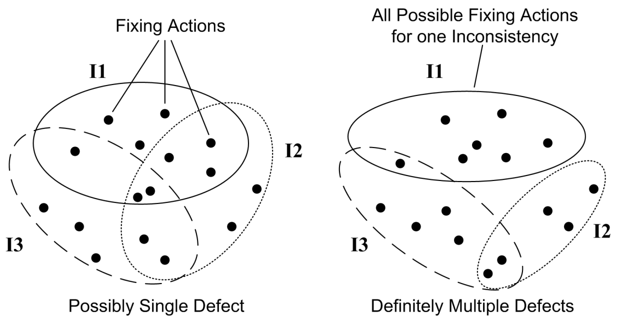
Fixing Actions
All Possible Fixing Actions for one Inconsistency
I1
I2
I3
I1
I2
I3
Possibly Single Defect
Definitely Multiple Defects

Figure 2: Examples of different Interrelations between Inconsistencies

decisions are often involved in several related rule violations in the decision model (symptoms).

3. VISION

As mentioned in the introduction, fixing inconsistencies should not focus on fixing them individually because they are only the symptoms of defects. Software modelers should identify relationships among inconsistencies to reason about the cause(s) (defects) for these inconsistencies.

Looking at a group of inconsistencies instead of a single one provides more information about the defect and should prove to be useful in reasoning about possible fixes: 1) by reducing the number of possible fixes and thus more easily identifying the defect(s) at hand and 2) by understanding how many defects are involved and what combination of changes are necessary fix them. The latter aspect in particular is challenging because fixes for defects do not necessarily involve single changes to a model but may require sets of changes. While the set of changes is a subset of the changes of the individual inconsistencies, the combinatorial explosion in what combination of changes of the individual inconsistencies to consider would be hard to decide manually. A fixing action in this scenario thus involves all the changes to fix the defect(s) involved and thus all its/their inconsistencies. Depending on the number of inconsistencies that are investigated at a time, this could mean a significant scalability improvement when only searching for fixing actions with a number of changes considerably smaller than the number of inconsistencies. In addition, the search for fixes that involve single changes only, could be used to determine if inconsistencies are related and if one or more defects caused the inconsistencies under investigation respectively.

Figure 2 shows two sets of inconsistencies that are different in terms of their relationships. An ellipse represents the fixing actions of a single inconsistency which are depicted as black dots. Fixing actions that are located in the overlapping areas of the ellipses are fixing actions shared among several inconsistencies. These fixing actions could fix all inconsistencies involved – however, it is not necessarily true that there is a single concrete fix for every fixing action in case the fixing action is abstract. On the left hand side of Figure 2, a scenario with three inconsistencies and an overlap among all of them is shown. In this case, there exist two possible fixing actions to resolve all three inconsistencies (overlap among all three ellipses). This implies that those inconsistencies are related, however it does not necessarily imply that these fixes are indeed the only correct fixes the software engineer should consider. In other words the cause for those three inconsistencies is possibly a single defect in which case one of the two fixing actions must be taken. However, if multiple defects cause the inconsistencies then other combinations of fixing actions are also possible. On the right hand side a different scenario is shown. We can see that both **I1** and **I2** are related to **I3** but they are in no relation to each other. In this scenario it is safe to say that there are at least two defects as there exists no single fixing action that could resolve all three inconsistencies at hand.

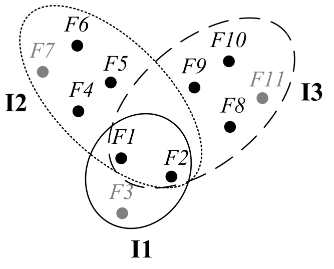

Figure 3: Overview of Fixes for the Example from Figure 1

In the example described in Section 2, all three described inconsistencies are related, they all can be fixed by either choosing fixing actions *F1* or *F2* as is apparent in Figure 3 (assuming the constraints are correct and therefore fixes *F3*, *F7*, and *F11* are irrelevant and grayed out). In this example, we even have the special situation that fixing **I1** in any case resolves all three inconsistencies. Additionally if the designer decides that there are multiple defects, the number of combined fixing actions is reduced by the fact that *F1* and *F2* respectively always have to be part of the solution. Furthermore, choosing *F2* to change the operation name from `turn-on` to `activate` in the class `Light`, excludes fixes *F4* and *F8* as they would require the operation `turn-on` to be present in the class `Light`.

To summarize we think that calculating fixes for more than one inconsistency at a time, especially if they are interrelated, is highly beneficial. Possible positive effects are:

1. A reduction in the number of possible fixing actions, through reasoning with more facts in the knowledge base (notice that knowledge about relationships among inconsistencies reduces the number of fixing actions).
2. As a consequence an increase in scalability since the calculation can be cut-off as soon as a combined solution is not achievable any more.
3. More precise fixing actions since the impact of the changes onto a larger amount of model elements is already considered.
4. Supporting designers by unburden them of having to know about interrelations when choosing a fixing action.

4. PROPOSED APPROACH

We propose to realize our vision stepwise. First of all we will use the choice generation technique described by Egyed et. al. [5] to better characterize fixes for individual inconsistencies (i. e., to compute concrete fixes for given abstract fixes). As a second step, we will investigate overlaps among inconsistencies. Initially, we will require the software engineer to identify relationships among inconsistencies; however, we will also develop heuristics to help the engineer. Questions we plan to answer are:

- How often occur interrelated inconsistencies in real world examples?
- How many choices for fixing an inconsistency can be excluded considering these interrelations? How strong is this reduction?
- Can abstract fixes become constrained fixes (constrained fixes being abstract fixes with some restrictions, for example the location is known and some possibilities of how to fix the inconsistency at that location have been excluded) or constrained fixes concrete fixes respectively? If yes, how often does this happen?

If the above mentioned qualitative aspects prove to be useful, as a next step we plan to investigate how the choice generation can be improved and sped up by considering the interplay among related inconsistencies already during the choice generation. For that we will rely on concepts and algorithms from CSPs (Constrained Satisfaction Problems). We also think some sort of grouping of actions the user can take will be necessary to reduce the amount of information we have to deal with. This grouping for example could be according to the effect on other inconsistencies or consistency rules in general.

As a final step we are planning to automatically determine which inconsistencies are related. On the one hand for certain consistency rules it could easily be defined on the meta-level, on the other hand for situational and not so obvious relations an online determination could be necessary. Online meaning in this case that the determination would occur while engineers are using the modeling tool. Our basic idea for this task until now is:

1. Get all model elements of one inconsistency that are involved during the evaluation of the consistency rule.
2. Look for other inconsistencies that share model elements with the one under investigation and repeat this step for those inconsistencies.
3. Do a pairwise search for fixes with a cardinality of one. If there are none handle those inconsistencies as none-related. If there are fixes search for overlaps with other pairwise search results to form inconsistency clusters.

Additionally we want to incorporate the concept of trust into our reasoning, this concept was already described in [9]. The concept basically states that some pieces of information observed through user behavior can be trusted. Such pieces of information are not evident from the model itself. An example would be that design decisions that introduce an inconsistency and are not undone must be important to the user and therefore can be assumed to be correct. Of course this assumption could only hold if it would be apparent to the user that an inconsistency was just created through tool support like instant consistency checking as described in [3]. This sort of trust would for example benefit engineers in including new requirements into consistent models. As already mentioned the first change of a series of model changes could cause several inconsistencies. Even with considering all those inconsistencies while searching for fixes the result will probably be an undo, since it is the simplest fix. However combined with trusting this first change the search would continue and hopefully come up with changes that are required by the requirements change anyway.

5. RELATED WORK

The problem of resolving inconsistencies has received considerable attention in the last two centuries. In this section we give a brief overview of work that has been done in this research area. On the one hand, in order to resolve inconsistencies, they have to be detected and tolerated. Almost 20 years ago, Balzer argued that inconsistencies should be detected and communicated to the developers; however, developers should not be hindered in continuing their work despite the presence of inconsistencies [1]. However at some point those inconsistencies have to be resolved, preferably with the support of automated techniques.

First off 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 [7], 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 A. Egyed [3] addresses this issue.

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 [11]. On the other hand Nentwich et. al. describe in their work [8] a method for generating interactive repairs from first order logic formulae - the same formulae that they already used to detect inconsistencies [7]. Another approach described by Egyed et. al. in their paper [5] shows how to generate choices for fixing an inconsistency without having to understand such formulae which can be complex in case conistency rules are written in programming languages. These approaches look at other model elements already defined in the model and use them as choices. This generated choices are then reduced by looking at the impact of each choice [4, 2] and removing those that would cause additional inconsistencies.

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 [10] but not used for fixing yet. Additionally Nentwich et. al. already stated in their work [8], 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.

6. CONCLUSIONS AND FUTURE WORK

In this work, we presented our vision of the next steps in inconsistency resolving, namely to not look at them individually but in clusters of related inconsistencies. We described the potential we think this approach can have for resolving inconsistencies and hope that we can substantiate it in the near future. As a result we hope we can further improve the user guidance during modeling. Once we have evaluated and validated the qualitative properties of searching for fixes for several inconsistencies at once, we are planning to explore the scalability properties.

Open questions that also would be interesting to investigate are: Are there different relationships between inconsistencies? Can more conceptual parallels be found to the compiler community and used? To what degree can clustering algorithms be applied? From a user guidance point of view it would be interesting to investigate how important qualitative aspects can be utilized. For example if the user tells the system that certain inconsistencies are related, but no common fix can be found. As stated before this would imply that those inconsistencies are not related. How is this piece of information useful to the user and can it be used to guide the user to a satisfying solution?

Acknowledgments

This research was funded by the Austrian FWF under agreement P21321-N15.

7. REFERENCES

[1] R. Balzer. Tolerating Inconsistency. In *ICSE*, pages 158–165, 1991.

[2] L. C. Briand, Y. Labiche, and L. O'Sullivan. Impact Analysis and Change Management of UML Models. In *ICSM*, pages 256–265. IEEE Computer Society, 2003.

[3] A. Egyed. Instant consistency checking for the UML. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *ICSE*, pages 381–390. ACM, 2006.

[4] A. Egyed. Fixing Inconsistencies in UML Design Models. In *ICSE*, pages 292–301. IEEE Computer Society, 2007.

[5] A. Egyed, E. Letier, and A. Finkelstein. Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In *ASE*, pages 99–108. IEEE, 2008.

[6] C. W. Johnson and C. Runciman. Semantic Errors - Diagnosis and Repair. In *SIGPLAN Symposium on Compiler Construction*, pages 88–97, 1982.

[7] 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.

[8] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency Management with Repair Actions. In *ICSE*, pages 455–464. IEEE Computer Society, 2003.

[9] A. Nöhrer and A. Egyed. Conflict Resolution Strategies during Product Configuration. In D. Benavides, D. Batory, and P. Grünbacher, editors, *VaMoS*, volume 37 of *ICB Research Report*, pages 107–114. Universität Duisburg-Essen, 2010.

[10] M. Sabetzadeh, S. Nejati, S. M. Easterbrook, and M. Chechik. Global consistency checking of distributed models with TReMer+. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *ICSE*, pages 815–818. ACM, 2008.

[11] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei. Supporting Automatic Model Inconsistency Fixing. In H. van Vliet and V. Issarny, editors, *ESEC/SIGSOFT FSE*, pages 315–324. ACM, 2009.