Fixing Inconsistencies in UML Design Models

Alexander Egyed
Teknowledge Corp.
4640 Admiralty Way, Suite 1010
Marina Del Rey, CA 90292, USA
aegyed@teknowledge.com

Abstract

Changes are inevitable during software development and so are their unintentional side effects. The focus of this paper is on UML design models, where unintentional side effects lead to inconsistencies. We demonstrate that a tool can assist the designer in discovering unintentional side effects, locating choices for fixing inconsistencies, and then in changing the design model. Our techniques are "on-line," applied as the designer works, and non-intrusive, without overwhelming the designer. This is a significant improvement over the state-of-the-art. Our tool is fully integrated with the design tool IBM Rational Rose™. It was empirically evaluated on 48 case studies.

1. Introduction

The very essence of iterative software development is change – and the designer's ability to make changes wherever and whenever necessary [2]. Changes have desired effects. They add features or fix bugs. However, changes also have undesired side effects [4,11]. They introduce new bugs.

This paper examines the impact of changes on UML design models [15]. There, negative side effects are observable if they violate known design rules (e.g., consistency and well-formedness rules; best practices). This paper explores how to discover the negative side effects of design changes, how to locate choices for fixing them, and how to predict the positive and negative side effects of these choices. Understanding about the impact of design changes is a fundamental best practice of the software engineering process [4,11,14].

Change impact analysis for design models is a complex problem because not every change causes inconsistencies, not every inconsistency is undesirable (living with inconsistencies [1,9]), and not every undesirable inconsistency is fixable without causing new inconsistencies. In fact, it is infeasible and impractical to enumerate all ways of fixing a given inconsistency as there are too many [11]. It would certainly overwhelm the designer if more than a handful of choices were presented. And it is beneficial to know what caused an inconsistency to decide how to fix it (some are caused by incorrect changes; others are caused by correct albeit incomplete changes).

We do not believe that a tool can automatically resolve inconsistencies because a tool cannot know whether an inconsistency is tolerable or why it was caused. However, a tool can be an assistant that provides the facts the designer must consider [16]. This work demonstrates that it is feasible to *locate all choices for fixing inconsistencies* and to *predict their positive and negative side effects.* However, inconsistencies are not independent events [4,11]. If a choice for fixing one inconsistency inadvertently affects how to fix another one then the designer should know about this dependency. This work thus also demonstrates how to *identify dependencies among inconsistencies*.

No existing work is able to identify all choices for fixing inconsistencies. Also, to the best of our knowledge, no existing work is able to identify dependencies and predict side effects. Our approach is also unique in that it does not analyze rules but rather *observes their behavior during evaluation* (i.e., model profiling [6]) – thus treating rules as black-box entities. Our approach is fully automated, tool supported, and integrated with the modeling tool IBM Rationale Rose™. Its correctness, completeness, and scalability are demonstrated and supported by the empirical evaluation of 48 small-to-large-scale UML models covering a total of 250,000 model elements and over 400,000 separate rule evaluations.

2. Related Work

Historically, impact analysis originated from database systems where data changes needed to be propagated to affected views and/or distributed locations. In

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 IEEE

software development, most work on impact analysis focused on source code. Some of these techniques emphasized on static or dynamic program slicing [13]. Other techniques emphasized on traceability. Bohner-Arnold [3] discuss many of these approaches. The goal is typically to narrow down what part of the system to change and/or what part to reanalyze/retest after the change. These approaches are very powerful but do not readily apply to (UML) design models.

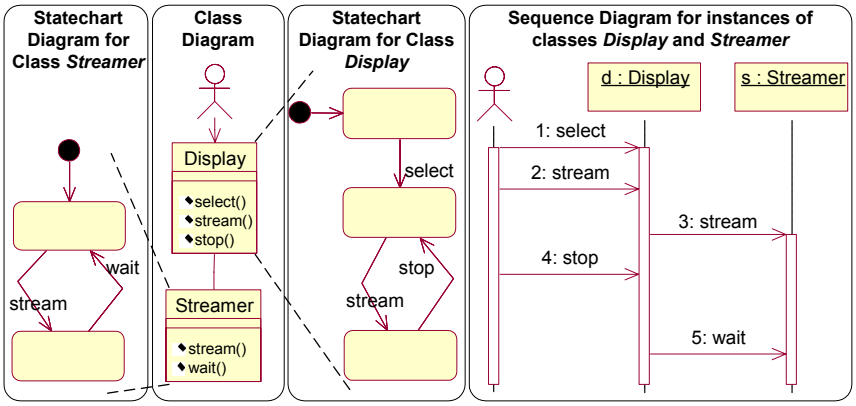

Figure 1. UML Model Illustration of a Video-On-Demand System

The work of Nentwich et al. [11] does apply to UML design models. It detects repair actions for fixing inconsistencies by analyzing consistency rules expressed in first-order logic and models expressed in xLinkIt [12]. Their repair actions are correct but not guaranteed to be complete (i.e., the repair actions identified depend in part on how the rules are written). Furthermore, their work does not identify dependencies among inconsistencies and potential side effects for fixing them – thus treating repair actions as independent events. However, their work is also fundamental to our work because they propose *abstract repair actions* as a way to reasonably enumerate the otherwise large number of concrete ways of fixing inconsistencies.

The work of Briand et al. [4] also computes change actions for UML models but takes an alternative approach. It identifies specific change propagation rules for all types of changes. This is problematic because there is no guarantee of correctness or completeness associated with these rules. Indeed, it is very hard to enumerate all kinds of changes and all their effects [6].

While it is important to know about inconsistencies, it is often too distracting to resolve them right away. The notion of "living with inconsistencies" [1,9,10] advocates that there is a benefit in allowing inconsistencies in design models on a temporary basis. While our approach provides choices for fixing inconsistencies instantly, it does not require the designer to fix them instantly. Our approach lets the designer explore inconsistencies according to their interests in the model.

3. Background, Illustration, and Problem

We previously introduced an approach for instant consistency checking of UML models [6]. This approach was fully automated and correctly decided what consistency rules to re-evaluate when a model changed. This approach was unique in that it did not require consistency rules with special annotations but rather treated consistency rules as black-box entities. It was based on a key enabling technology – *a model profiler*. Much like a source code profiler observes the execution of the source code during runtime, our model profiler observed the evaluation of a model during consistency checking (i.e., it knows what fields of what model elements are accessed when and how often). In our previous work, we used profiling data to establish a correlation among model elements and consistency rules to decide what consistency rules to re-evaluate with changes. This paper builds on this technology.

3.1. Illustration and Two Sample Rules

The illustration in Figure 1 depicts four UML diagrams [15] created with the modeling tool IBM Rational Rose™. The given model represents a design-time snapshot of a real, albeit simplified video-on-demand (VOD) system [5]. The class diagram represents the structure of the VOD system: a *Display* used for visualizing movies and receiving user input and a *Streamer* for downloading and decoding movie streams. The two statechart diagrams describe the behavior of the two classes. For example, the behavior of the *Streamer* simply toggles between the waiting and the streaming state depending on whether it receives the *wait* or *stream* commands. Finally, the sequence diagram contains interactions among objects of the *Streamer* and *Display* classes. The interaction depicts a user invoking the *select* and *stream* messages on object *d* (an instance of *Display*), which, in turn, invokes another *stream* message on object *s* (an instance of *Streamer*). The sequence diagram also shows that invoking *stop* on *Display* causes *wait* on *Streamer*.

Consistency rules for UML describe conditions that all UML models must satisfy for them to be considered valid (e.g., syntactic well-formedness, coherence be-

tween different diagrams, and even best practices). The empirical study in this paper involves 34 such consistency rules covering class, sequence, and statechart diagrams. For example:

Rule 1: message name must match class method

Rule 1 states that the name of a message must match a method in the receiver's class. If this rule is evaluated on message *select* in the sequence diagram then the condition first identifies the receiver object *d* of the message (*message.receiver*) (see arrowhead of message), followed by the declared base class *Display* (*receiver.base*), and the methods {*select()*, *stream()*, *stop()*} owned by the class *Display* (*base.methods*). The condition then returns true because the set of method names contains the message name *select*.

Rule 2: message sequence must match behavior

Rule 2 states that the sequence of incoming messages in an object of a sequence diagram must match the allowed behavior of the statechart diagram of the object's class. For example, object *d* receives the messages *select*, *stream*, and *wait* – in this order. The statechart diagram of class *Display* (the base of object *d*) allows this behavior because it is a valid sequence.

3.2. Understanding Change

Since consistency rules are conditions on a model, their truth values change only if the model changes. Instant consistency checking thus requires an understanding when, where, and how the model changes. For this purpose, our UML/Analyzer tool relies on the UML Interface Wrapper component – an infrastructure we previously developed and integrated with IBM Rational Rose and other COTS modeling tools [8]. This infrastructure exposes the modeling data of the COTS modeling tool in an UML-compliant fashion. It also employs a sophisticated change detection mechanism. The latter is particularly important because it notifies our tool of changes to the UML model in real time while the designer uses the modeling tool. The consistency rules themselves are hard coded into the logic of the UML/Analyzer tool. No explicit rule language was used. However given that the UML Interface Wrapper does not observe the internals of our tool but rather profiles its interaction with the Rose modeling tool, it should be quite simple to replace the consistency checker component and rule language of our tool. More details on the tool's architecture are in [7].

3.3. Problem Statement Revisited

Note that a rule may be instantiated and evaluated many times. The 34 rules used in this study were instantiated over 400,000 times on the 48 UML models we evaluated. Every *rule instance* represented a separate consistency statement. All rule instances had to be consistent for the model to be consistent. Thus, the designer had to understand and manage the impact of a design change onto *all* rule instances. This was no easy task. Fortunately, a design change did not randomly affect rule instances. Rejlich [14] and Briand et al. [4] argued that change impact analysis only had to consider a limited degree of neighboring model elements.

Our empirical evaluation (Section 6) showed that 90% of the rule instances accessed at most 11 model elements spanning up to five degrees of neighbors. While this number appears small, one must consider that the number of accessible model elements increases exponentially with the degrees of neighbors involved. We evaluated this increase on the 48 UML models and found that five degrees of neighbors involve between several hundred and several thousand model elements depending on the model (Figure 2). *It was not practical for a designer to consider these many model elements to understand the impact of a single design change.*

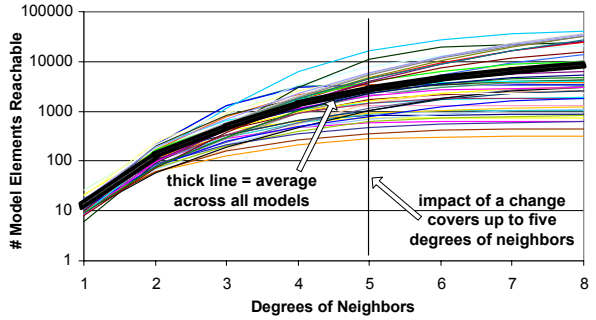

Figure 2. Exponential Increase of Number of Model Elements Reachable by Degrees of Neighbors

4. Approach

4.1. Discovering Effects of Design Changes

The only correct way of detecting the negative effects of a design change is for the designer to execute the design change and observing its effects. In [6], we demonstrated that it is possible to re-evaluate the impact of design changes instantly for many kinds of consistency and well-formed rules. A design change may cause one or more of the following situations:

A design change may cause the instantiation of a rule, which, once evaluated, is either consistent (new C) or inconsistent (new I). A design change may also require the re-evaluation of a rule instance. This rule instance may remain consistent (C->C) or inconsistent (I->I) or it may change from consistent to inconsistent

(C->I) or from inconsistent to consistent (I->C). A design change may also cause a rule instance to become obsolete, usually when a model element is deleted or its evaluation path is broken (removed C or removed I).

One may think of the negative side effects of a design change as the ones that lead to inconsistencies as in *new I* or *C->I*. If a design change failed to correct a previously inconsistent rule (I->I) then this may also be considered an undesired side effect.

Continuing with the illustration in Figure 1, consider the simple problem of renaming the method *stream* of class *Display* to *play* to avoid the confusing dual use of the term *stream*. Renaming the class method alone is bound to cause inconsistencies because other diagrams might refer to it by name. Also, a global search and replace is useless because the term *stream* is also used in the class *Streamer* to mean something else. Let us assume that the designer, after inspecting the model, renames the class method, the same-named state transition in *Streamer's* statechart diagram, and the message "*3: stream*" in the sequence diagram – three changes.

To identify whether these three design changes are sufficient to change the naming convention of *stream*, we use the UML/Analyzer tool. The tool determines that the four previously consistent rule instances R13, R14, R22, and R23 have become inconsistent (C->I). In [6], we demonstrated that the feedback generated by the UML/Analyzer tool is complete and correct.

R13	Rule 1 evaluated on link message "2:stream"
R14	Rule 1 evaluated on link message "3:stream"
R22	Rule 2 evaluated on object "s:Streamer"
R23	Rule 2 evaluated on object "d:Display"

While a tool can identify inconsistencies caused by changes, it cannot decide on whether they are desirable or not. It is safe to assume that consistencies are always desirable. Yet, it is invalid to claim that inconsistencies are always undesirable. There are certainly situations where designers do tolerate inconsistencies [9] (usually temporarily).

Since a tool cannot decide whether the effects of a change are desirable or not, the designer must make this decision. If the designer is ok with the effects of a design change then there is no problem. But if the effects are undesired then the designer may want to fix them. The following demonstrates how to locate the choices for fixing inconsistencies.

4.2. Locating Incorrect Model Elements

If an inconsistency is undesired then this implies that a design change was not planned and/or executed correctly. There usually are multiple choices on what model elements to change in order to fix an inconsistency but it is typically very hard to identify them all [11]. In the following, we will show how our approach locates all choices automatically. If the designer should also know why an inconsistency happened then our approach narrows down the list of choices.

Locating Choices for Fixing an Inconsistency

We use model profiling [6] to determine which model elements and fields are accessed during the evaluation of a rule. Take, for example, rule instance R14 – one of the four inconsistent rules identified in 4.1. R14 evaluates the consistency of the message "*2:stream*" with respect to Rule 1. Table 1 depicts the list of model elements/fields accessed during the evaluation of the rule. We see that the rule first looked at the name of message *stream* and then at its receiver (object *d*), the receiver's base class (*Display*), and all of the class' methods. The rule was inconsistent because the name of the message did not match the name of one of the methods.

Table 1. Scope Elements for Rule R14

R14	message *stream*[name]	I-only
	message *stream*[receiver]	I-only
	object *d*[base]	C&I
	class *Display*[methods]	C-only
	method *select*[name]	C-only
	method *play*[name]	C-only
	method *stop*[name]	C-only

We define a *scope element* to denote a model element/field pair – the smallest kind of element in UML models. From [6] we know that the scope determined through model profiling is complete which means that it includes *all* scope elements that affect the consistency of a rule. Consequently, at least one scope element must change to fix a given inconsistency. There are several choices for fixing R14:

1) renaming message *stream* to *play*
2) changing the receiver of message *play* to object *s*
3) adding a new method *stream* to the class *Display*
4) changing the ownership of object *d* to *Streamer*
5) renaming method *select* to *stream*
6) renaming method *play* to *stream*
7) renaming method *stop* to *stream*
8) deleting message *stream* (makes R14 obsolete)

And there are additional choices. For example, instead of adding a new method *stream* to *Display*, we could move the existing method *stream* from *Streamer* to *Display*. With all its variations, one could come up with a very large number of *concrete fixes* for this in-

consistency. However, all of these fixes involve at least one of the rule's scope elements.

It is impossible to enumerate all concrete fixes for a given inconsistency [11]. However, it is possible to identify the *starting points* for fixing a given inconsistency – its scope elements. There are seven scope elements in R14 (from six model elements) and as such there are seven locations from where to start fixing this inconsistency. We will show later that our list of choices is typically small enough not to overwhelm the designer. And we will show that it is correct and complete (no false positives/negatives). For the designer, knowing the starting point is often sufficient (i.e., Nentwich et al.'s *abstract change actions* [11]).

Locating Choices for an Erroneous Design Change

While the list of choices is small already, it can be reduced further if the reason for the inconsistency is known. If the designer believes that an inconsistency was caused due to an erroneous design change then the erroneous design change must be *located* and *changed*. Locating an erroneous design change is difficult if more than one change was involved (typical). Our approach identifies the erroneous design change by intersecting the set of design changes with the scope elements of a given inconsistency.

Table 2. Some of the Scope Elements for Rule R22

R22	~~object s[incomingMessages]~~	~~C&I~~
	message play[name]	I-Only
	~~message wait[name]~~	~~C-Only~~
	~~class Streamer[statemachine]~~	~~Unused~~
	~~object s[base]~~	~~C&I~~
	~~transition play[name][target]~~	~~Unused~~
	~~transition wait[name][target]~~	~~Unused~~

Revisiting the illustration, it does not take much reasoning to see that inconsistency R22 is misplaced. R22 talks about the inconsistency of some messages with respect to the statechart of *Streamer*. Our change was supposed to only affect class *Display* and those who use this class. Investigating the choices for fixing R22, we find that only one of the three design changes contributed to its inconsistency: the name change to message "*3:stream*", now *play*. This change must have been incorrect and only one choice remains out of originally seven possible choices (see Table 2; note, the list of scope elements was abbreviated).

Once located, the easiest way to fix an erroneous design change is to undo it. If the rule was consistent before the change then undoing the change will make it consistent again. Alternatively, the erroneous design chance could be fixed by doing it differently. For rule R22, the name change was wrong and should be undone because we changed the name of an outgoing message of object *d* instead of an incoming message. Surprisingly, undoing the name change also fixes inconsistency R13. This is desirable but also unexpected. We will discuss these kinds of side effects later.

Locating Choices for a Missing Design Change

Inconsistencies may happen even if the design changes are correct. In our illustration, two inconsistencies (R14 and R21) remain after the undo. Yet, on closer inspection, the two design changes are correct. It follows that additional design changes are needed to regain consistency and the choices for fixing such inconsistencies must involve the scope elements *minus* the previous design changes. Since the name change to message *play* (originally *stream*) is in the scope of inconsistency R14 (recall Table 1), it should not be considered as a choice for fixing this inconsistency.

4.3. Fixing an Inconsistency

A tool cannot decide on the best choice for fixing a given inconsistency because there are factors other than consistency that contribute to a good design (e.g., taste, gut feeling, experience, knowledge on the future evolution of the design). Since these factors are typically not formalizable, it is the designer who must pick the best choice. Also recall that the choices identified are merely the starting points for fixing inconsistencies because there are many concrete fixes for any given choice [11]. So, it is the designer who must execute a fix even if the starting point (choice) is obvious.

4.4. Dependencies among Inconsistencies

Existing work identifies choices for fixing individual inconsistencies but it does not consider the side effects of these choices onto other inconsistencies [4]. Inconsistencies are not independent events – they typically come in clusters (see 6.2): they either share erroneous model elements or the erroneous model elements are located in close proximity in the model. Therefore, if a choice resolves one inconsistency but in doing so affects the choices of another inconsistency then the designer should know about this.

Our tool cannot decide whether fixes for multiple inconsistencies are contradictory [11] but our tool can tell the designer which inconsistencies are related. This solves a hard problem because larger models contain thousands of inconsistencies (the worst model of the 48 ones evaluated contained 10,465 inconsistencies!) and *a designer would find it impossible to identify related inconsistencies*.

A *dependency among inconsistencies* exists if these inconsistencies share one or more choices for fixing them. We already observed a dependency among two inconsistencies earlier when we undid the erroneous design change (renaming of message *stream* to *play*). There, the undo fix had the intentional effect of fixing inconsistency R22 and it had the unintentional, though positive, side effect of fixing inconsistency R13.

However, it turns out that it is quite possible that two rules have a common choice although their scopes do not intersect. This problem is caused by scope elements that refer to other UML model elements.

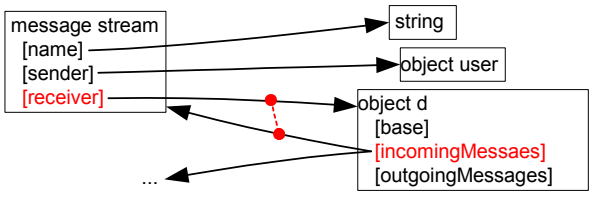

Figure 3. Two Scope Elements but One Choice

Consider the two remaining inconsistencies R14 and R21. Looking at their scopes, we find that they have three common scope elements and consequently three common choices for fixing them both. However, this list is incomplete. Missing from the overlap are two distinct scope elements whose changes are linked. Figure 3 depicts the message *stream* as a data element and shows that its *receiver* field references object *d*. Yet, object *d* is aware of this reference because it contains a back-pointing field, called *incomingMessages*, which lists all messages that have object *d* as their receiver.

This example illustrates one out of many cases where some fields of distinct scope elements are tied. In almost all cases, the tie is between two scope elements but there could be more. Fortunately, it is possible to compute the back-pointing scope elements because UML has a well-defined meta model that identifies all bi-directional relationships and derived fields. We thus generated the complete list of back-pointing scope elements for the fields of UML model element (there are 3487 such back-pointing fields for UML 1.3[1]). And, we located the specific model element(s) that hold the back pointers introspectively. This has a small overhead cost but this computation is only necessary for changing model elements – a rather small number over time. The *extended scope* of a rule thus includes the original scope elements and their back-pointing scope elements. The extended scope is 1.6 times larger.

[1] Our approach is based on a UML 1.3 because of the needs of our industrial partners. The newer UML 2.0 is not commonly supported in industry, in part, because of legacy models and tools.

Dependencies among inconsistencies reveal the existence of common choices for fixing multiple inconsistencies. Dependencies, however, cannot guarantee that there is indeed a single concrete fix for a common choice that would satisfy all the inconsistencies. Thus, a dependency implies an opportunity for a single design change to fix multiple inconsistencies. In reverse, if there is no dependency among inconsistencies then there cannot be a common choice for fixing them. No dependency implies the need for multiple design changes. Of the four inconsistencies caused by the original three design changes, the choices for R14 and R21 did not overlap with those of R13 and R22. The undo fix earlier was thus a necessary and separate fix.

4.5. Side Effects of Fixing Inconsistencies

Fixes for inconsistencies are simply additional design changes. We already know that design changes may have positive effects and negative effects – but it is generally not possible to predict these side effects in advance. The only way to correctly compute the side effects is to explore each choice (execute its concrete fix) and observe its effects. Doing so for every concrete fix is not scalable and generally not of interest to the designer. To the best of our knowledge, there exists no work today that is able to predict the side effects of changes with any claim of accuracy. Nentwich et al. [11] even argued that this problem is undecideable if one attempts to analyze the rule logic. Our approach is also limited in its ability to predict such side effects but it does provide a useful annotation for every choice:

Unused Choices do not have Side Effects

A choice marked *Unused* is guaranteed to not affect another rule. It is determined by the fact that its scope element occurs in only one scope. The proof is omitted but can be inferred from the completeness/correctness property of the scope. From the over 73,000 inconsistencies across the 48 UML models, we found that roughly 13% had unused choices.

I-Only Choices have Positive Side Effects Only

A choice marked *I-only* affects multiple inconsistencies but it is guaranteed to never affect consistent rules. These choices do not have negative side effects. We found that nearly 66% of all inconsistencies had *I-Only* choices.

C-Only Choices have Negative Side Effects Only

A choice marked *C-only* may affect consistent rules. These choices are problematic because they may cause inconsistencies (C->I). However, our approach cannot guarantee the correctness of *C-only*. We already know that the scopes of rules are small but they are not minimal. The scopes of consistent rules do include

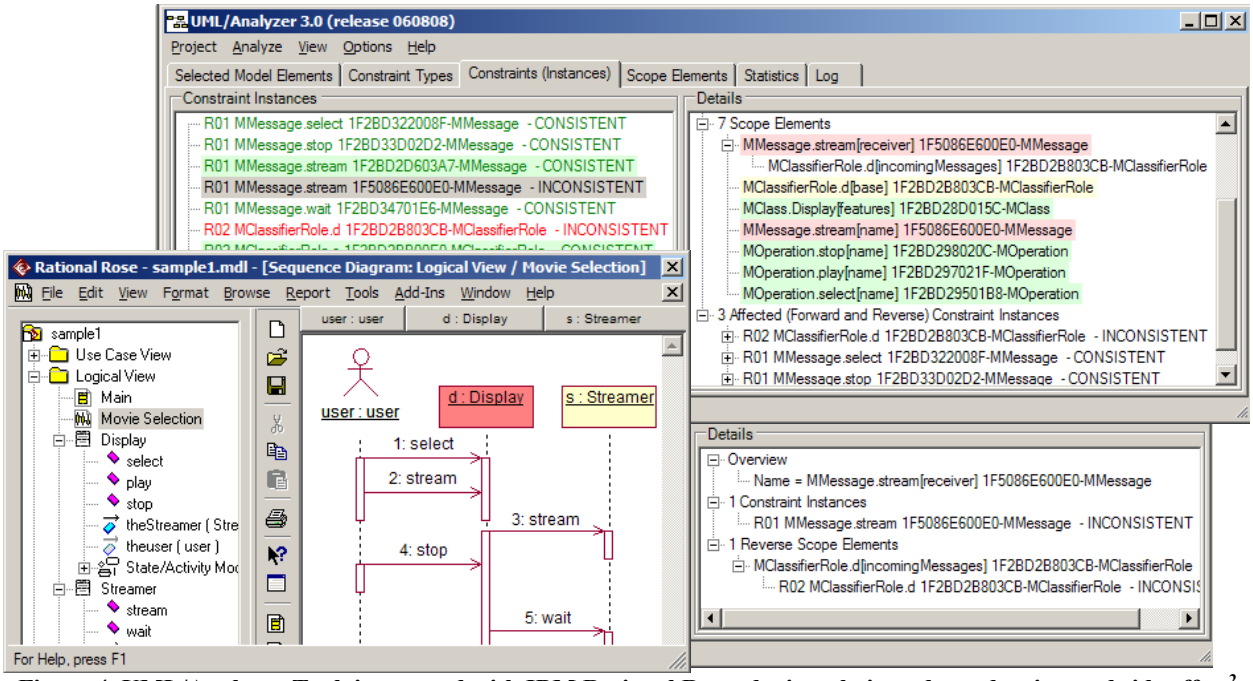

Figure 4. UML/Analyzer Tool, integrated with IBM Rational Rose, depicts choices, dependencies, and side effect[2]

scope elements that do not affect their consistency – though we believe this error to be small! Fortunately, *C-only* is conservative. If a model element is *not* in the scope of a consistent rule then it truly does not affect the consistency of that rule. Over 52% of all inconsistencies had *C-Only* choices.

C&I Choices have Positive & Negative Side Effects

A choice marked *C&I* affects both inconsistent and consistent rules. Such a choice may have positive and negative side effects. As with *C-Only*, the negative side effects cannot be guaranteed (false positives), however, the positive ones can. Over 82% of all rule instances had C&I scope elements. Table 1 and 2 above showed these annotations for inconsistencies R14 and R22.

5. UML/Analyzer Tool

The UML/Analyzer tool [6] was extended in this work to help the designer locate choices for fixing inconsistencies, identify dependencies among rule instances, locate common choices for fixing related inconsistencies, and annotate choices with their potential side effects. The tool provides all this information to the designer on demand and in a non-intrusive manner.

The tool was also integrated with the design tool IBM Rational Rose™ for ease of use and it was used for the empirical evaluation discussed in Section 6. Figure 4 depicts a few screen snapshots of the tool. The bottom-left depicts IBM Rational Rose™. A choice for fixing inconsistency R14 is highlighted. This inconsistency is also highlighted in the UML/Analyzer tool on the top. The top right depicts the scope elements for R14[2] and all its dependencies to other rules. The color coding used in the tool indicates the annotations *Unused*, *I-Only*, *C-Only*, and *C&I*. The bottom right depicts another screen snapshot of the UML/Analyzer tool. The scope element for message *stream*[receiver] is highlighted and we see that this scope element is used in R14 and that there is a back-pointing scope element object *d* [incomingMessages] which is used in R21[2].

[2] The tool displays a more descriptive name for rule instances instead of the abbreviated names used in this paper.

6. Evaluation

The following discusses the correctness, completeness, usability, and scalability of our approach.

6.1. Correctness and Completeness

The located choices for fixing inconsistencies are guaranteed to be correct and complete: *(1) for every choice there is an actual concrete fix that resolves a given inconsistency* and *(2) every concrete fix involves at least one of the choices*.

This observation should come as a surprise given that the scope of rules contains false positives [6]. The scopes of rules are not guaranteed to be minimal be-

cause a rule typically iterates over a set of model elements in search for an invalid condition. In doing so, some of the model elements accessed in search for this condition are irrelevant. For example, rule R15, which is consistent, iterated over all class methods of *Display* until it found one that matches the name *stop*. It found such a method after the third iteration (after methods *select* and *play*). Since the first two methods did not have the desired name, they were irrelevant to the consistency of this rule. The evaluation backtracked until it found the desired method. Unfortunately, this backtracking was invisible to model profiling and thus all three methods were in the scope although only the last one mattered.

During the evaluation of an inconsistency the same backtracking occurs however all investigated model elements have the *potential* of making the rule consistent – or otherwise why would a rule care to access the model element? For example, inconsistency R14 also accessed the three methods *select*, *play* and *stop* and it was inconsistent because none of the methods matched the message name. While we could speculate which one of the three methods should be changed to fix this inconsistency, it is entirely feasible to rename any one of the three methods. Renaming *select* would fix this inconsistency as well as renaming *play*.[3]

The scope for inconsistent rules is thus complete and minimal. Since our approach only locates choices for fixing *inconsistencies*, we do not care that the scope for consistencies is not minimal. Dependencies and side effects of choices are computed based on the *extended scope* of an inconsistency. Since our approach is correctly able to compute the extended scope (based on code generated from the UML meta model), we found that the extended scope is also correct and minimal for *inconsistencies*. Only the *C-only* and *C&I* annotations have false positives because they require the extended scopes for consistent rules.

6.2. Usable but Useful?

We did not perform usability studies on human designers. We thus could not evaluate whether the information provided by our tool would indeed simplify the design process. This is future work.

However, we empirically evaluated 48 UML models to determine whether our tool would not overwhelm its user with too much information (i.e., too many choices). These models were very diverse in domain and size. Figure 5 depicts the sizes of the models which cover the entire spectrum from small to very large models of up to 36,000 model elements. Most of the larger models originated from industry, some were reverse engineered from software systems, and yet others were obtained indirectly through colleagues. In terms of domain, the models covered avionics systems, medical systems, data-centric systems, and command-and-control types of systems.

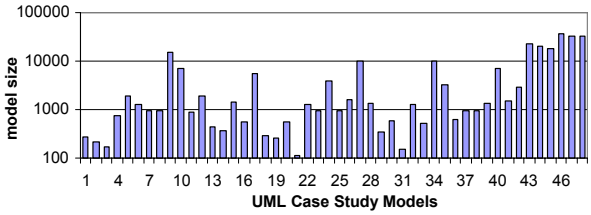

Figure 5. Sizes of the 48 UML Models

The empirical evaluation focused on 34 types of rules covering UML class, sequence, and statechart diagrams – the most widely used UML diagrams. The rules were selected because of the needs of our industrial partners. They do cover consistency, well-formedness, and best practices. These rules had to be instantiated over 400,000 times for all models combined (the largest model evaluated 58,000 instances).

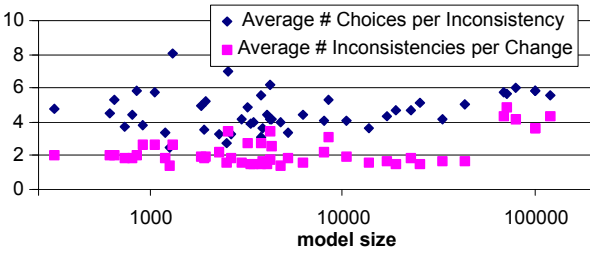

Figure 6. Number of Choices per Inconsistency and Number of Inconsistencies per Design Change

The choices for fixing an inconsistency comprises of a fix for every scope element: *#choices = #scope elements*. We empirically evaluated whether the number of choices is manageable and does not overwhelm the designer. We found that the number of choices was in average 5.4 (between 2.5 and 8 in average) – a reasonably small number. But more importantly, we found that the number of choices did not increase with the size of the model. Figure 6 demonstrates significant variations in the number of choices across the 48 UML models, however, the average number of choices do not increase with size.

However, a design change may cause multiple inconsistencies and thus the choices multiply accordingly: #choices = *#inconsistencies/change* * *#choices/inconsistency*. The average *#inconsistencies/change* was very small. However, it was skewed by the fact that we did not care about changes that caused consistencies. We thus empirically evaluated *#inconsisten-*

[3] While both changes fix the inconsistency, renaming *select* causes new inconsistencies while renaming *stream* does not!

cies/change for changes that caused inconsistencies and it was 2.8 in average (Figure 6). Note that we did this by randomly seeding changes because we could not observe the designers. This number obviously depended on how inconsistent a model was and we observed that 90% of scope elements affected fewer than 9 rules in total – also a reasonable worst case.

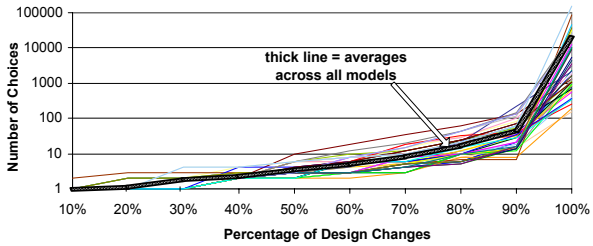

Figure 7. Most Design Changes have few Choices

Yet, there were some inconsistencies that had hundreds of choices for fixing them. Figure 7 depicts the #choices for fixing all inconsistencies caused by single design changes. We observed a diseconomy of scale in that 50% of all design changes generated up to 10 choices; 70% generated up to 36 choices; and 90% generated up to 200 choices. Clearly, it was not feasible for a designer to manually investigate all choices for all changes. However, this figure also showed that most of the diseconomy of scale occured for a rather small percentage of all changes. A designer could handle 60-70% of all changes for all models, and 80% for most models. And up to 90% for the majority of models. However, it is obvious that there is a need to eliminate choices automatically for the <20% of changes. This could be done through heuristics [11] or by prioritizing the choices (i.e., based on our annotations). We will investigate this issue in future work.

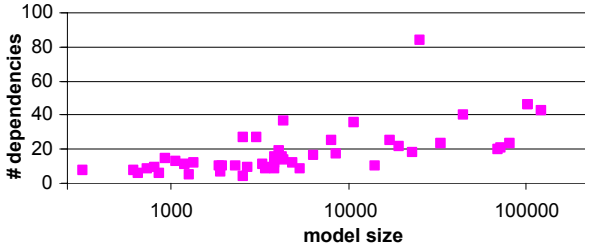

Figure 8. Number of Dependencies per Inconsistency

The number of dependencies among inconsistencies would appear to be quite large. First, it is based on the extended scopes for inconsistencies which are 1.6 times larger than the regular scopes. Second, only a single scope element must overlap for there to be a dependency. This could average to the squared size of the extended-scope – an unreasonable number. Fortunately, we observed 18 dependencies or less for most inconsistencies (Figure 8) – this is quite reasonable given that there is a similar diseconomy of scale associated with dependencies (Figure 9) and thus most inconsistencies had many fewer dependencies. This number also confirmed that inconsistencies were not random occurrences. They clustered among common model elements (hence, the need for dependencies in the first place) and thus if there was a dependency among two inconsistencies then it usually involved multiple scope elements. We did observe a slight increase in the number of dependencies and the model size. This is partially explained by the fact that the larger models were somewhat more inconsistent than the smaller models. However, do note that the x-axis of Figure 8 is exponential. The increase is thus very small.

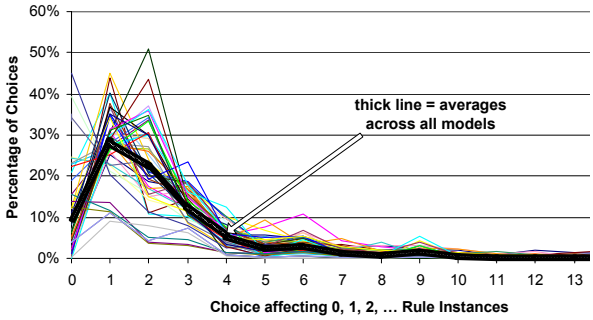

Figure 9. Most Choices affect few Rules

Finally, our approach annotates choices with *Unused*, *I-only*, *C-only*, and *C&I*. Since there is exactly one annotation per choice, this annotation will not overwhelm the designer. In section 4.5 we already presented empirical evidence that all four annotations are very likely – though *C&I* is the strongest of all. These annotations are a window into the most complex part of fixing inconsistencies – avoiding new inconsistencies. Figure 9 demonstrates that most choices (>90%) affect multiple rules – thus most changes need to consider the side effects of change onto other rules. However, most choices do not affect more than a handful of rules. Thus, the side effect is manageable.

6.3. Scalability

Our approach is linearly complex (both memory and computation cost) and thus quite scalable. The overhead of locating choices, identifying dependencies, and annotating choices is negligible. The worst case computation was less than 20 milliseconds and was often too short to even be measured on an Intel Centrino 1.7GHz. This cost is in addition to the cost of instant consistency checking [6] which was in average 9ms per change with a worst case of 2 seconds.

6.4. Threats to Validity

Internal validity: We investigated 34 rules in the context of 48 third-party UML models. Since our approach performed well for *all* these models, we believe that there is little threat to the internal validity of the measured data. However, we were not able to directly observe the designers in their use of our approach and cannot provide heuristics on likely changes. Future work is thus needed to augment the usability study.

External validity: The more serious issue of our study hinges on rules. It is quite feasible to construct rules that are inherently inscalable. While the large number of rules used in this study suggests that most rules are scalable we must exclude non-scalable rule should they occur. Furthermore, while the quantity of rule types has no implication on the number of choices, it does affect the number of dependencies and the distribution of *Unused*, *I-only*, *C-only*, and *C&I*. The more rules, the more crowded the model and thus the more likely the undesired side effects. This is an inherent problem of modeling and not just a limitation of our approach. In terms of rule quality, our approach assumes that rules are implemented correctly and do not access model elements they do not need.

There is also the issue of abstract choices versus concrete choices. It is infeasible to enumerate all concrete choices of fixing an inconsistency. Nentwich et al. [11] thus proposed *abstract change actions*, a concept we borrowed in this work. Our abstract choices identified the starting points for fixes but not all model elements involved in fixing a model. Consequently, the dependencies and annotations identified by our approach were limited to these starting points.

7. Conclusion

Generally, a tool cannot repair models automatically; however, a tool can provide choices and it can predict side effects. This paper demonstrated that it is quite feasible to correctly detect all choices for fixing inconsistencies and doing so does not overwhelm the designer in most situations. However, the more serious issue of working with inconsistencies is deciding on choices that affect multiple inconsistencies. Inconsistencies typically come in clusters and it is important to avoid making repairs that inadvertently and adversely affect how to fix other, related inconsistencies. We demonstrated that these situations are the norm and not the exception (i.e., most inconsistencies have dependencies on other inconsistencies). We also demonstrated that our approach is able to detect these dependencies among inconsistencies and identify the common choices for fixing them.

8. References

1. Balzer, R.: "Tolerating Inconsistency," *Proceedings of 13th International Conference on Software Engineering (ICSE-13)*, May 1991, pp.158-165.
2. Boehm B. W.: A Spiral Model of Software Development and Enhancement. *IEEE Computer* 21(5), 1988, pp.61-72.
3. Bohner, S.A., Arnold, R. S.: Software Change Impact Analysis. IEEE Computer Society Press, 1991.
4. Briand, L. C. , Labiche, Y., and O'Sullivan, L.: "Impact Analysis and Change Management of UML Models ," *Proc. of the International Conference on Software Maintenance*, Amsterdam, The Netherlands, 2003, pp.256.
5. Dohyung, K.: "Java MPEG Player," *http://peace.snu.ac.kr/dhkim/java/MPEG/*, 1999.
6. Egyed, A.: "Instant Consistency Checking for the UML," *Proc. of the 28th International Conference on Software Engineering (ICSE)*, Shanghai, China, 2005.
7. Egyed, A.: "UML/Analyzer: A Tool for the Instant Consistency Checking of UML Models," *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, Minneapolis, USA, May 2006.
8. Egyed A. and Balzer B.: Integrating COTS Software into Systems through Instrumentation and Reasoning. *International Journal of Automated Software Engineering (JASE)* 13(1), 2006, pp.41-64.
9. Fickas, S., Feather, M., Kramer, J.: Proceedings of ICSE-97 Workshop on Living with Inconsistency. Boston, USA, 1997.
10. Hunter A. and Nuseibeh B.: Managing Inconsistent Specifications: Reasoning, Analysis, and Action. *ACM Transactions on Software Engineering and Methodology* 7(4), 1998, pp.335-367.
11. Nentwich, C., Emmerich, W., and Finkelstein, A.: "Consistency Management with Repair Actions," *Proc. of the 25th International Conference on Software Engineering (ICSE)*, Portland, USA, 2003, pp.455-464.
12. Nentwich C., Capra L., Emmerich W., and Finkelstein A.: xlinkit: a consistency checking and smart link generation service. *ACM Transactions on Internet Technology (TOIT)* 2(2), 2002, pp.151-185.
13. Orso, A., Apiwattanapong, T., Law, J., and Rothermel, G.: "An Empirical Comparison of Dynamic Impact Analysis Algorithms," *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, Edinburgh, United Kingdom, May 2004, pp.491-500.
14. Rajlich V. and Gosavi P.: Incremental Change in Object-Oriented Programming. *IEEE Software* 21(4), 2004, pp.62-69.
15. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison Wesley, 1999.
16. van Der Straeten, R., Mens, T., Simmonds, J., and Jonckers, V.: "Using Description Logic to Maintain Consistency between UML Models," *Proceedings of 6th International Conference on the Unified Modeling Language (UML 2003)*, October 2003.