Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models

Alexander Egyed [1,2], Emmanuel Letier [2], and Anthony Finkelstein [2]

Johannes Kepler University [1]
Institute for Systems Eng. and Automation
Linz, Austria
ae@sea.uni-linz.ac.at

University College London [2]
Department of Computer Science
London, UK
{e.letier, a.finkelstein}@cs.ucl.ac.uk

Abstract

Our objective is to provide automated support for assisting designers in fixing inconsistencies in UML models. We have previously developed techniques for efficiently detecting inconsistencies in such models and identifying where changes need to occur in order to fix problems detected by these means. This paper extends previous work by describing a technique for automatically generating a set of concrete changes for fixing inconsistencies and providing information about the impact of each change on all consistency rules. The approach is integrated with the design tool IBM Rational Rose™. We demonstrate the computational scalability and usability of the approach through the empirical evaluation of 39 UML models of sizes up to 120,000 elements.

1. Introduction

Integrated development environments (IDE) provide rich automated features for detecting and fixing errors at the code level. For example, Eclipse has support for so-called "Quick Fixes" that generate potential resolutions for resolving Java errors. Programmers find these resolution mechanisms extremely useful. Some design tools have started to develop similar features by presenting choices for fixing design problems. Figure 1 shows a capability of the UML tool IBM Rational Rose™, which suggests the set of choices *stream()*, *wait()*, and *connect()* to fix the incorrect message name *play* in a sequence diagram. However, the support currently provided at the model level is very limited.

For modeling languages, such as UML [1], constraints are typically defined in form of *consistency rules* [2]. Automated resolution mechanisms then require tool developers to manually implement separate *fixing rules* for every location where a violation of a consistency rule could occur. When the number of consistency rules increases, this represents a significant effort. Furthermore, developers of such resolution mechanisms must take care that their fixes are correct in the sense that they resolve the inconsistencies without introducing new ones. For multi-paradigm modeling languages, such as UML, this is extremely difficult to achieve because of the numerous interactions among consistency rules. An additional problem when developing tool support for resolution mechanisms in UML models is that different development teams often use different consistency rules and different set of consistency rules may be used at different stages of the project lifecycle. This *make a 'hard coded' resolution mechanism defined for a fixed set of consistency rules almost useless.*

This paper suggests an alternative approach that eliminates the need for creating fixing rules altogether. The approach first generates a set of values a model element may take. The values generated depend on the

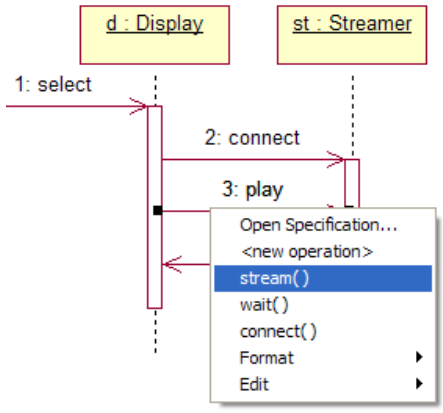


Figure 1. IBM Rational Rose™ Suggesting a Fix for an incorrect Message Name

model element expected type and on the set of values of that type already present in the model. This set may contain valid and invalid choices with respect to the consistency rules. The approach then prunes the set of values trough incremental consistency checking. The result is a set of choices on how to fix an inconsistency that is guaranteed not to violate any known consistency rule (i.e., impact of a change [3]). The critical step of our technique is to discover and re-evaluate the correct and complete set of consistency rules that are affected by a fix. This paper also presents an empirical scalability analysis showing that this approach maintains instantaneous response time to user queries even when the size of the models increases.

For modelers (the end users of modeling tools), our approach is fully automated. Manual overhead is required for the tool builders responsible for implementing the value generation functions. However, even tool builders save effort with respect to the alternative approach of developing resolution functions that are dependent on the violated consistency rules because the development of the choice generators is much simpler than that of resolution rules. An important additional benefit of our approach is that the value generation functions do not need to be changed when consistency rules change (an extremely frequent situation with UML modeling). Our generate and prune approach also has the advantage that it provides rich feedback to the designer by presenting not only valid choices satisfying all consistency rules, but also why invalid ones could not. We believe this feature is particularly useful because fixing inconsistencies in design models often involves making several changes that do not immediately resolve all inconsistencies [4].

2. Background and Illustration

Our technique builds on previous work for automatically and efficiently detecting inconsistencies in design models [5, 6], understanding trade-offs among design choices [7], and providing support for the resolution of such inconsistencies [8, 9].

Figure 2 introduces a simple illustration used in the remainder of this paper. The figure depicts the model of a video-on-demand system (VOD) created with the UML modeling tool IBM Rational Rose™. The model contains a class, a statechart, and a sequence diagram. The class diagram (top) represents the two main components of the VOD system with *Display* handling the user IO (visualizing movies and receiving user input) and *Streamer* handling the server interaction from the client perspective. The sequence diagram (middle) describes the user scenario of selecting a movie (user invoking *select* on instance of class *Display*). We see that this user invocation then triggers a communication between *Display* and *Streamer* where *Display* first requests *Streamer* to connect to the server and then requests *Streamer* to start playing the movie. The Streamer then draws picture frames in periodic intervals (only one such *draw* message is shown). Finally, the model contains a statechart diagram, depicting the behavior of the class *Streamer* (bottom). The behavior is quite simple. Once instantiated, the *Steamer* first establishes a connection to the server and then toggles between the *waiting* and *streaming* mode depending on the *wait* or *stream* events.

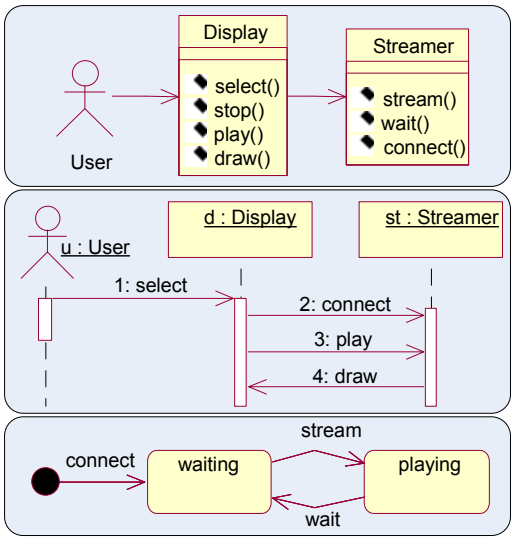


Figure 2. Model of Video-On-Demand System

C1: operations=message.receiver.base.operations
return (operations->name->contains(message.name))

Inconsistency 1: Message *play* is not defined as a method in Class *Streamer*

C2: startingPoints = find state transitions equal first message name
startingPoints->exists(object sequence equal reachable sequence from startingPoint)

Inconsistency 2: Behavior of class *Streamer* (statechart) does not allow for message *play* to follow message *connect*

C3: in=message.receiver.base.incomingAssociations;?
out=message.sender.base.outgoingAssociations;?
return (in.intersectedWith(out)<>{})

Inconsistency 3: Calling direction of Message *draw* does not match association direction between classes *Display* and *Streamer*

Figure 3. Consistency Rules and Inconsistencies

While the UML specification primarily defines basic well-formedness rules, researchers have come up with a wide range of consistency rules to define the correctness of UML diagrams. A consistency rule is a formal condition that evaluates a portion of a model. If a rule is violated then we speak of an *inconsistency*. Some of these consistency rules are widely applicable while others might only apply in certain domains.

Figure 3 depicts three consistency rules. Rule C1

ensures that a message in a sequence diagram is declared as a method in the receiver's class. Rule C2 ensures that the behavior of a sequence of messages is allowed by a state machine. And rule C3 ensures that the calling direction of a message is allowed by the calling direction among classes. It must be noted that consistency rules are written from the perspective of the meta model. For example, the top of Figure 4 depicts the (simplified) portion of the UML meta model used by rule C1. Since this rule is instantiated once for every message, we speak of four ***consistency rule instances*** of rule C1 for the four messages in Figure 2. One of these four rule instances is inconsistent (Inconsistency 1).

Inconsistency 1 violates consistency rule C1 because the class *Streamer* never declared a method with the name *play*. The formal definition of C1 says that the rule first computes operations = message.receiver.base.operations. We must look at the actual UML model, to understand what this instruction means. The bottom of Figure 4 depicts the portion of the UML model that is accessed by Inconsistency 1. There we see that message.receiver is the object *st*, receiver.base is the class *Streamer*, and base.operations are the methods {*connect*(), *stream*(), *wait*()}. The condition returns false because the set of method names (operations->name) does not contain the message name *play*.

In [5], we showed that it was possible to detect inconsistencies quickly. We did so by building a *model profiler* for observing which model elements a consistency checker accessed during the evaluation of consistency rule instances. For example, when consistency rule C1 was evaluated on the message *play* (resulting in Inconsistency 1) then the profiler observed the following accesses to model elements (Figure 4): Message.play [name], Message.play [receiver], ClassifierRole.st [base], Class.Streamer [features], Operation.stream [name], Operation.wait [name], Operation.connect [name].

In [5], this profiling data formed the basis for deciding *when* to re-evaluate *what* consistency rule instance. In essence, a consistency rule instance only then had to be re-evaluated if a part of the model changed that it previously accessed. For example, if the designer changes the method name *connect()* then rule C1 must be re-evaluated on message *play* because the method name was accessed during that rule's evaluation. Note that in the remainder of this paper, we identify consistency rule instances through the consistency rule and model element it was applied on (e.g., C1 Message.play).

In [9], we demonstrated that an expanded version of this profiling data was beneficial in understanding *where* to fix inconsistencies (i.e., identifying all the model elements that *potentially* fix an inconsistency). Here we must also distinguish between the ***location type*** in the meta model and the ***location*** in the model. While it is straightforward to determine the location type, identifying the actual location in the model is much harder. The xlinkit framework [8] did this determination in part through white-box analysis of consistency rules while [9] relied on the model profiler. Both approaches were able to reveal *where* to fix inconsistencies (=locations) but not *how* to fix them (=values or choices). ***Deciding how to fix inconsistencies is the focus of this paper***.

3. Problem

The problem addressed in this paper is on identifying all concrete choices for fixing a model element such that it resolves a given inconsistency and does not cause new ones.

Consider again Inconsistency 1, where the receiver's class did not declare a method that matched message *play*. In [9], we demonstrated how to identify all potential locations where to fix this inconsistency. One obvious location is the message name itself but we could also change the name of one of the methods or the message receiver. However, identifying the location where to fix an inconsistency is only half the solution. The other half is identifying how to change that location. Not every change is valid. Consider, for example, the message name. There are an infinite number of string choices available for changing a name. To date there exists no automated approach for determining valid choices. The only approach known today, to manually code *fixing rules* as supplements to consistency rules, is not only problematic because of the manual overhead involved but also because it is inadequate in considering the side effects of multiple inconsistencies onto a single location [3]. For example, for changing a message name, that violated consistency rule C1, a fixing rule

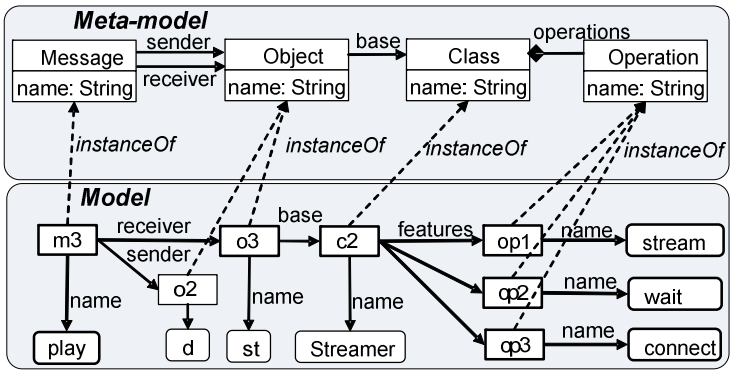


Figure 4. Meta-Model and Model for Inconsistency 1 (simplified)

would simply returns all the method names in the message receiver's class: the strings *connect*, *stream*, and *wait* (as did IBM Rational Rose in Figure 1). While this fixing rule approach is simple, we must consider the following problems:

Problem 1: Fixing Rules for all Locations

The tool developer must re-write the fixing rule (manually!) for every location where the inconsistency could be fixed. For example, changing the method name rather than the message name requires another fixing rule that is simply Method.name = Message.name. Alternatively, changing the receiver of the message *play* requires yet another fixing rule that searches for any object in the model whose base.operations contains the desired name. Indeed, object *d* (instance of *Display*) would be a suitable object because it has a method with the name *play*. One problem with fixing rules is thus its repetition for every possible location where the inconsistency could be resolved. Our empirical evaluation on 39 models and 24 consistency rules showed that the number of locations is not too large (typically around 5-15) but consequently requires one fixing rule per location type and consistency rule (roughly 24 * 5-15 rules) which is labor intensive and a source of errors.

Problem 2: Fixing Rule for all Consistency Rules

A more severe problem is the interplay among multiple consistency rule instances. For example, the above fixing rule for message name *play* identified all method names of the class *Streamer* as choices for fixing the incorrect message name. Indeed, all of these names resolve inconsistency 1, however, these choices also affect consistency rule C2 and its Inconsistency 2.

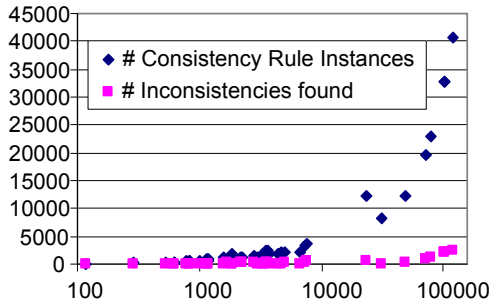


Figure 5. Number of Consistency Rules Instances in UML Models and Number of Inconsistencies Found

Inconsistency 2 was about the wrong sequence of messages in the statechart diagram. Considering the second inconsistency, we find that neither message *wait* nor message *connect* are allowed to follow message *connect* (notice that the *connect* event in the statechart diagram leads to the *waiting* state from which only the transition *stream* is allowed). Consequently, only the message name *stream* fixes both inconsistencies.

Considering simultaneously all consistency rules that affect a model element is particularly hard to do manually. Design models typically require the evaluation of a large number of consistency rule instances. Figure 5 demonstrates the magnitude of the challenge. There we see that the number of rule instances (y-axis) increases with the model size (x-axis). This data is based on the evaluation 39 small-to-large scale UML models on 24 kinds of consistency rules. We see that some of the larger models required tens of thousands of rule instances and often contained thousands of inconsistencies. It is impossible for a human user to know which ones of these many rule instances affect the choices of a particular fix.

Even our small illustration in Figure 2 required 11 consistency rule instances of which only three returned inconsistencies. *Fixing inconsistencies must thus not only be aware of other inconsistencies but also of consistencies.* So, *the true challenge of fixing inconsistencies is about understanding the side effects of a change on all affected consistency rules* – whether they be consistent or inconsistent. The fixing rules identified above failed in this regard.

Problem 3: Consistency Rules differ among Users

Finally, an additional problem when developing tool support for managing inconsistencies in UML models is that different designers often use different consistency rules which may make the resolution rules defined for one designer useless for another. *A truly useful approach in helping designers fix inconsistencies must not be 'hard coded' to a specific set of consistency rules.*

4. Approach

Our approach and tool solves the problem described above by systematically exploring all choices in a trial-and-error exploration. To illustrate this basic idea, consider a set of choices on how to change a model element. Each choice is a potential value for fixing a given inconsistency. Our approach then determines the validity of these choices by trying them on the model – one by one – and re-evaluating the consistency of the model for each trial. A choice is valid if it fixes the consistency rule and does not cause other inconsistencies. Otherwise, a choice is invalid.

As input, our approach takes a location where to fix an inconsistency and as output it generates a list of choices on how to fix the inconsistency (Figure 6)

without causing any new inconsistencies. Our current approach considers choices from the pool of existing model elements only. This approach thus ignores the creation of new model elements as choices for fixing inconsistencies. For the modellers (end user), our approach is fully automated and tool supported (it is also integrated with the design tool IBM Rational Rose™) – even if the modellers change the consistency rules (add/remove/modify). For the tool builders, our approach does require manual effort in form of writing the choice generators. However, we will demonstrate below that writing such generators is much less effort than writing fixing rules.

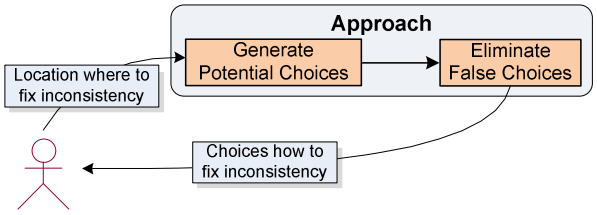


Figure 6. Choice Generation and Elimination

While our approach appears simple at first glance, we must address three important issues:

- How do we generate the initial set of choices?
- How do we know what rules to re-evaluate to eliminate false choices?
- Is this approach correct and does it scale?

The remainder of the paper will explore these issues.

4.1. Choice Generation

Choice generation functions generate possible values that specific fields of a model element may take. They only consider values that are already present in the model so that the generation and evaluation of choices can be done fully automatically without user intervention. However, a brute force generation of choices is not scalable. Figure 7 shows that larger models would yield thousands of choices per location.

To counter the scalability problem, our choice generation functions are manually custom-tailored to syntactical constraints of the modeling language – in particular to its well-formedness criteria. For example, we defined the choice generation function for fixing the receiver of a message as including objects of the same sequence diagram only. This function reflects the fact that, in UML, messages may not refer to objects outside a sequence diagram (row 1 in Table 1).

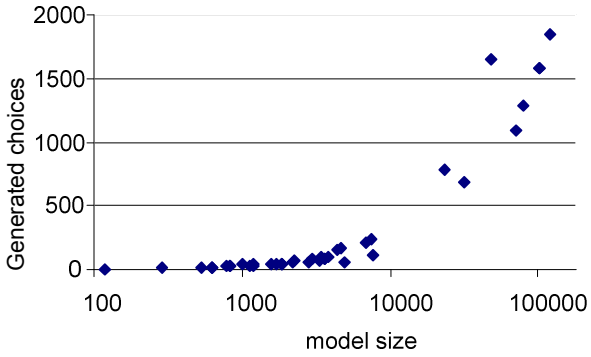


Figure 7. Brute Force Choice Generation does not Scale with the Model Size

Table 1. Example of four manually customized Choice Generator Functions

1	m:Message.receiver: choices = m.interaction.classifierRoles
2	m:Message.name: choices = {} foreach (method in m.receiver.base.methods) choices.insert(method.name)
3	ae:AssociationEnd.multipliciy choices = {1, 0..1, 1..n, 0..n}
4	c:Class.namespace choices = {} foreach (a in c.associations) foreach (oc in a.classifiers) choices.insert(oc.namespace)

The choice generation functions are mostly independent from the consistency rules so as to ease their implementation and make them more usable across projects that use different consistency rules. At the present time, the functions must be coded manually. The difference to fixing rules discussed earlier is that we only require one function per location type. This significantly reduces the implementation overhead required for coding:

O(#consistency rules * #location types) fixing rules
versus
O(#location types) choice generator functions.

Moreover, the choice generation functions need not consider the impact of multiple consistency rules. This eliminates much of the complexity of writing them. It should be noted that in a few cases we based choice generation functions on consistency rules. As an example, consider the choices for fixing a message name which is not adequately constrained through well-formedness rules. To limit the near-infinite number of name strings, we devised a choice generation function based on consistency rule C1. The

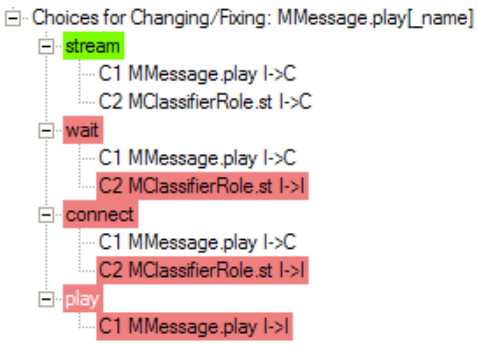
choice generator thus simply returns the names of methods in the message receiver's class (row 2 in Table 1). We previously introduced this rule as a fixing rule for C1. Basing choices on consistency rules is not a problem unless the consistency rule is considered irrelevant by a designer.

It is important to note that *the choice generator function must only be written once per modeling language and is reusable across domains.* Ideally, the tool builder creates it. The modeller (end user of our approach) does not have to customize the choice generator *unless* she changes the modeling language.

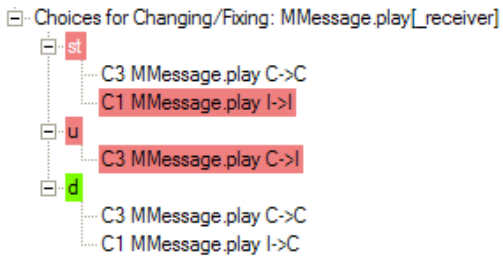
4.2. Choice Reduction

Choice generator functions are simple but may produce false choices. The choice reduction step then eliminates all those choices that cannot satisfy all consistency rules. This problem is very hard because there are many instances of consistency rules (typically tens of thousands) and every choice may affect different consistency rule instances. Again, a brute force approach of simply re-evaluating *all* consistency rule instances would not scale (recall Figure 5). Our approach instead relies on its ability to perform instant, incremental consistency checking.

To illustrate this, let us again try to fix Inconsistency 1 by changing the name of message *play*. In this case, the choice generator suggests the following *potential* choices: *connect*, *stream*, and *wait*.



The figure above depicts the result of our tool's exploration of these choices. The original value, string *play*, is also included for comparison. We see that *play* is an invalid name because it cannot satisfy the rule *C1 Message.play* – i.e., *play* is not declared as a method in the receiver class). While the other three values all satisfy rule C1, not all of them are valid because a second consistency rule interferes. The second rule, *C2 ClassifierRole.st*, ensures that the sequence of message names is allowed in the statechart diagram. Choices *wait* and *connect* are invalid because neither is allowed to follow the message *connect*. Choice *stream,* on the other hand, is allowed to follow *connect*.



Another way of fixing Inconsistency 1 is to change the receiver of message *play* (see figure above). Currently, the receiver of that message is object *st* and we could change the receiver to either the instance of *User* (object *u*) or the instance of *Display* (object *d*). While our tool's exploration appears to be similar to the exploration of the message name choices above, do note that we are no longer dealing with the exact same consistency rule instances. While fixing the message name required the re-evaluation of ruleinstances of C1 and C2, fixing the message receiver requires the re-evaluation of rule instances of C1 and C3. Rule *C3 MMessage.play* is concerned with the calling direction of a message.

The original message receiver (object *st*) did not violate that consistency rule because the calling direction of message *play* was consistent with the association direction in the class diagram (i.e., note that the class *Display* is allowed to call the class *Streamer*). However, if we change the receiver of message *play* to object *u*, an instance of *User*, then we trigger an inconsistency because *User* is not allowed to call methods of *Streamer*. Object *d* is however a valid choice because *Display* is allowed to call itself (implicit in the rule). As a result, choice *u* is eliminated. The two remaining choices are re-explored in context of rule C1 where object *st* is eliminated.

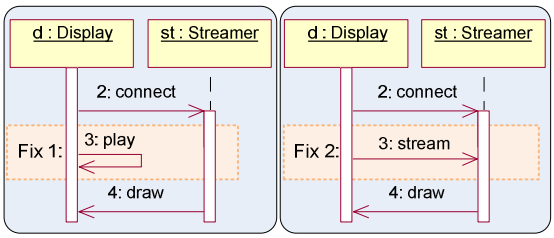


Figure 8. Two Choices for Fixing Inconsistency 1

Therefore, the first inconsistency can be fixed in at least two ways as depicted in Figure 8:

1) changing the receiver of the message to object *d*
2) changing the name of the message to *stream*

It is important to note that we do not decide automatically on what fix to choose. The designer will make this decision [10].

4.3. Impact of a Change

Obviously, this approach is only then correct if it can identify *all* consistency rule instances affected by a fix (i.e., all of them must be re-evaluated). In previous work [5], we demonstrated how to identify all affected consistency rules instances under the assumption that we know all model elements that have changed. However, the challenge here is that a single design change often modifies multiple model elements. That is, the change suggested by one of our fix choices affect not only the element being fixed but also other elements that reference it. Consider the example of the message *draw*. The receiver of message *draw* refers to object *d*. Since the internal data structure of UML uses classical forward and backward chaining, the reference from the message *receiver* implies a backward reference from object's *incomingMessages* field. Much of the meta model of the UML is composed of such bi-directional references. The single fix of message receiver thus changes three model elements: 1) message *receiver*, 2) object *d*'s *incomingMessages* (it looses a message), and 3) object *st*'s *incomingMessages* (it gains a message).

Our approach must thus identify all model elements affected by a choice in order to determine all consistency rule instances that are affected by that choice. This problem can be solved easily by maintaining records of (back)pointers. Like the choice generator functions, these data structures are generic and need only be built once per modeling language.

5. Scalability and Correctness Usability

We evaluated our approach on 39 UML models with model sizes between 100-120,000 elements. For measuring the model sizes, we only counted those elements that were used during the consistency checking of the 24 types of consistency rules included in this study. Depending on the model size, consistency rules were instantiated many times. In total 223,000 consistency rule instances were evaluated.

The 39 models were rather diverse. Most models originated from industry, some were reverse engineered, and yet others were obtained through colleagues. In terms of domains, the models covered avionics systems, medical systems, data-centric systems, and closed-loop types of systems. All of these models were built with the modeling tool IBM Rational Rose™. Their level of consistency was also diverse – between 2-26% with an average of 8.4%.

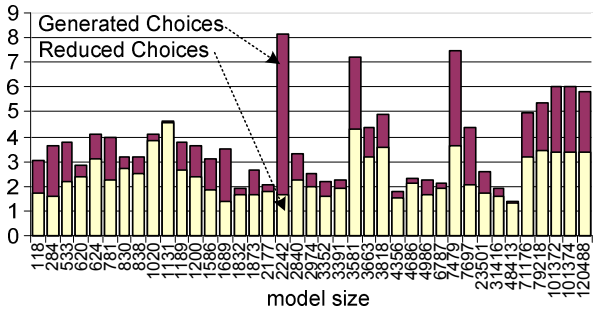


Figure 9. Generated Choices and Reduced Choices

In this study, we looked at 14 types of locations. These 14 types of locations adequately covered many interesting places for fixing inconsistencies though this study cannot be considered exhaustive because the UML meta model is much larger. We then defined 14 choice generator functions to cover the 14 types of locations. In the 39 models, these 14 types of locations occurred 65,379 times. We then proceeded in exhaustingly computing the choices for fixing all these locations. Figure 9 depicts the number of choices generated by our approach. It was surprising to find that in average only 2.4 valid choices (with a worst case of 69 choices) were found for fixing a location. Moreover, the number of choices for fixing inconsistencies did not increase with the size of the model. Both observations are beneficial for usability because the user is not overwhelmed with a large number of valid choices. Our approach also proved to be highly scalable in terms of performance. In average, the choice generation and reduction required only 11ms per location on a 2.2GHz Pentium Processor.

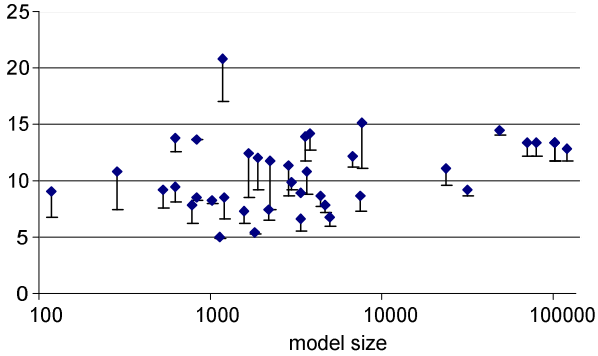


Figure 10. Number of Locations for Fixing Inconsistencies (with/without false ones)

Through our previous work [9], we already knew that there are in average few locations were to fix any given inconsistency. For the 39 models and 24 consistency rules included in this study, there were in average 10.4 such locations per inconsistency (Figure 10). Therefore, the total number of choices for fixing

an inconsistency was in average 10.4 locations * 2.4 choices per location ≈ 25 choices.

While [9] was able to identify all locations, the list did contain false positives. A false location is a location for which no valid choice exists. Such false locations are not uncommon as indicated in Figure 10 through a vertical error bar. In average, 11.2% of all locations suggested in [9] were false. This fact was recognized in that paper. This paper also eliminates this drawback. By exploring choices, we are now able to automatically detect these false locations. Of the remaining, valid locations, roughly 48% had multiple valid choices whereas 40% had only a single valid choice. This observation might prove useful in future work since locations with single choices suggest the possibility of auto-correction.

Our approach does not suggest false choices (i.e., false positives) because we are able to identify all consistency rules affected by a fix automatically. However, since writing the choice generator involves manual overhead, our approach may miss valid choices if the choice generator does not find them. It is therefore important to design the choice generator well. Fortunately, choice generator functions are generic and can be used across domains and applications. Moreover, the cost of writing them is small in comparison to fixing rules. Note that instead of manually writing O(#type of consistency rules * #types of locations) fixing rules, our approach only requires O(#types of locations) choice generate rules – a significant savings.

6. Threads to Validity

This study was based on 39 small-to-large UML models covering a wide range of domains and originating from a diverse set of designers. The 24 types of consistency rules were representative of consistency rules found in industry. The 14 types of locations were typical locations for fixing inconsistencies. Since the 24 types of consistency rules were instantiated over 223,000 times and the 14 types of location occurred over 65,000 times in the 39 UML models, we are confident that our findings are accurate with respect to the rules and locations used.

However, there are many other types of consistency rules and many more types of locations. Only 17% of all relevant locations were evaluated in this case study. Of the remaining locations, 4% were deemed unchangeable (i.e., we believe that they should never be changed) and the rest were simply undefined. This leaves a large pool of uncertainty and thus we cannot generalize that every location will be scalable or useable. Yet, given that our findings are highly encouraging, we believe that our approach would scale with respect to many other consistency rules and locations as well.

Another limitation of this work is its restriction to single changes. While this work allows the designer to explore the different choices for fixing inconsistencies, even among multiple locations, these choices are mutually exclusive (i.e., every choice is considered separate). Yet, there are cases where adequately fixing an inconsistency may involve several *concurrent* changes – where each change may not resolve all inconsistencies or may even temporarily introduce new ones, before reaching a consistent state. So, this work cannot be considered a complete solution to the problem of fixing inconsistencies.

Finally, this work generated choices solely from the pool of existing model elements. This approach thus did not consider the creation of new model elements to fixing inconsistencies. This problem is in fact related to the earlier point of restricting to single changes since the creation of new model elements would typically involve multiple concurrent changes.

7. Related Work

This work in essence explores trade-offs among design decisions. In Section 2, we already outlined the differences to our extensive previous work [5-9]. In the following, we discuss other relevant work.

A very significant problem is reasoning about inconsistencies that are the result of conflicts among multiple stakeholders – often referred to as viewpoints. For example, [11] and [12] define techniques for reasoning formally over multiple models despite the presence of logical inconsistencies. [13] defines a technique for merging conceptual models and detecting inconsistencies over the merged model. These techniques focus on handling inconsistencies between multiple models expressed in a single language, either conceptual models or state machine models, and do not explicitly support the generation of concrete ways to resolve inconsistencies. In contrast, the techniques describe in this paper deal with multi-paradigm descriptions and aims at providing support for transforming the models so as to fix inconsistencies. We do not however support the merging of different models coming from different sources and formal reasoning about the behavioral properties of our models. Most of our consistency rules are the equivalent of the static semantic rules of programming languages.

While it is important to know about inconsistencies, it is often too distracting to resolve them right away. The notion of "living with inconsistencies" [4] advocates that there is a benefit in allowing

inconsistencies in design models on a temporary basis. While our approach provides fixes for inconsistencies instantly, it does not require the engineers to fix inconsistencies when they first occur (although it could be used that way). Our approach tracks all presently-known inconsistencies and lets the engineers decide *when* to resolve *what* inconsistencies.

Our work is loosely related to the constraint satisfaction problem (CSP). CSP deals with the combinatorial problem of what choices best satisfy a given set of constraints. Since this problem is computationally expensive, certain optimizations have been developed. In particular, the AC3 optimization [14] defines a mapping between choices and the constraints they affect. Constraints are only then re-evaluated if their choices change. We borrowed this concept in our use of scopes. A key difference is that CSP uses "white-box constraints." It is thus known, in advance, what choices a constraint will encounter. Consistency rules in UML typically are black-box constraints. This is the why our approach relies on model profiling. Since model checkers, such as Alloy [15], are built in part on CSP-like technology, they are also quite capable of solving the problem in this paper. However, since our approach is specifically tailored to the problem of fixing inconsistencies in design models, it does not suffer from the scalability problems.

The work of Briand et al. [3] is also relevant to our approach because it computes change actions for UML models. However, 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. The work by Robins et al. [16] is similar in that it introduces wizards which are defined manually. However, it is very hard to enumerate all kinds of changes and all their effects [5]. Our approach does not require such annotations.

An area that seems related to fixing inconsistencies is that of repairing data structures in databases or code. In particular, the assertion-based approach in [17] appears similar since there faults in data structures are repaired through constraint-based reasoning which is not unlike consistency rules. However, their approach applies to code and does not take under consideration multi-paradigm modeling. Also, their approach applies only to the fixing of faults that are simple enough for automated approaches to not only identify the choices for fixing it but also selecting the most suitable one.

It must be emphasized that dealing with choices in design models is about more than just reducing infeasible choices. Outside the scope of this paper were thus other aspects of fixing inconsistencies, such as keeping a formal history of the choices [18] for later maintenance, using version control and rollback mechanisms [19], recording why each alternative is rejected [20], managing the problem-domain dependencies among the decisions [21], decentralized consistency [22] and consistency checking among different languages [23]. However, it is easy to see that all these areas affect the fixing of inconsistencies and future work will explore these and other issues.

8. Conclusions

We have developed tool support allowing UML modelers to systematically explore alternative ways of fixing inconsistencies at different locations in the model (some of which may not be obvious) and anticipate the impact of such changes on all consistency rules simultaneously. Our technique can only generate fixes to inconsistencies whose resolution does not require the designer to introduce new model elements or new names. The latter case could however be partially handled by defining default name generation mechanisms in the value generator for some location type (for example, generating default names for roles in an association).

More importantly, our technique can only generate resolutions that involve *change in a single location at a time*. Frequently, resolving an inconsistency involves *changing multiple model elements simultaneously*. For example, fixing an inconsistent message name in a message sequence chart may involve changing a method name in a class diagram and several transitions in a sequence diagram. We plan on addressing this problem by developing an adequate set of higher-level model evolution operators [24] that aggregates the application of the elementary changes we have considered so far and correspond to frequently needed model transformation steps.

The approach described here on UML models can be transferred to other meta-models and consistency rules. In future work, we wish to develop a generic meta-tool that would allow tool developers to automatically instantiate our techniques for identifying and resolving inconsistencies to any meta-model. In particular, we envision enriching the XML-based techniques of xlinkit to MOF-based meta-model descriptions.

9. Acknowledgement

We gratefully acknowledge support from EPSRC through Standard Research Grant EP/F032110/1.

10. References

[1] J. Rumbaugh, J. Ivar, and B. Grady, *The Unified Modeling Language Reference Manual*: Addison Wesley, 1999.

[2] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency Handling in Multi-Perspective Specifications," *Transactions on Softw. Eng.,* vol. 20, pp. 569-578, 1994.

[3] L. C. Briand, Y. Labiche, and L. O'Sullivan, "Impact Analysis and Change Management of UML Models," Proceedings of the International Conference on Software Maintenance (ICSM), Amsterdam, The Netherlands, 2003.

[4] R. Balzer, "Tolerating Inconsistency," Proceedings of 13th International Conference on Software Engineering (ICSE), 1991.

[5] A. Egyed, "Instant Consistency Checking for the UML," Proceedings of the International Conference on Software Engineering, 2006.

[6] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein, "xlinkit: a consistency checking and smart link generation service," *ACM Transactions on Internet Technology,*vol. 2, pp. 151-185, 2002.

[7] A. Egyed and D. S. Wile, "Support for Managing Design-Time Decisions," *IEEE Transactions on Software Engineering* vol. 32, pp. 299-314, 2006.

[8] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency Management with Repair Actions," Proceedings of the 25th International Conference on Software Engineering (ICSE), Portland, Oregon, USA, 2003.

[9] A. Egyed, "Fixing Inconsistencies in UML Design Models," Proceedings of the International Conference on Software Engineering 2007.

[10] R. van Der Straeten, T. Mens, J. Simmonds, and V. Jonckers, "Using Description Logic to Maintain Consistency between UML Models," Proceedings of 6th International Conference on the Unified Modeling Language (UML), 2003.

[11] A. Hunter and B. Nuseibeh, "Analysing Inconsistent Specifications," presented at Proceedings of 3rd International Symposium on Requirements Engineering (RE), 1997.

[12] S. Easterbrook and M. Chechik, "A Framework for Multi-Valued Reasoning over Inconsistent Viewpoints," Proceedings of the 23rd International Conference on Software Engineering, 2001.

[13] M. Sabetzadeh, S. Nejati, S. Liaskos, S. Easterbrook, and M. Chechik, "Consistency Checking of Conceptual Models via Model Merging," Proceedings of the 15th IEEE International Requirements Engineering Conference, New Delhi, India, 2007.

[14] A. K. Mackworth, "Consistency in Networks of Relations," *Journal of Artificial Intelligence* vol. 8, pp. 99-118, 1977.

[15] D. Jackson, "Alloy: A lightweight object modelling notation," *ACM Transactions on Software Engineering Methodology* vol. 11, 2002.

[16] J. Robbins and D. Redmiles, "Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML," International Conference on Construction of Software Engineering Tools, Los Angeles, CA, 1999.

[17] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid, "Assertion-based repair of complex data structures," Proceedings of the 22nd International Conference on Automated Software Engineering, Atlanta, GA, 2007.

[18] A. G. Cass and L. J. Osterweil, "Requirements-Based Design Guidance: A Process-Centered Consistency Management Approach," in *Department of Computer Science, University of Massachusetts, Amherst, MA 01003 (UM-CS-2002-024)*, 2002.

[19] A. van der Hoek and E. Dashofy, "xADL 2.0," in http://www.isr.uci.edu/projects/xarchuci/.

[20] D. Wile, "Program Developments: Formal Explanations of Implementations," *Communications of the ACM* vol. 26, 1983.

[21] M. L. Begeman and J. Conklin, "The right tool for the job," *Byte* vol. 13, pp. 255-266, 1988.

[22] S. M. Kaplan and G. E. Kaiser, "Incremental attribute evaluation in distributed language-based environments," Proceedings of the 5th Annual Symposium on Principles of Distributed Computing, Calgary, Canada, 1986.

[23] R. N. Taylor, R. W. Selby, M. Young, F. C. Belz, L. A. Clarce, J. C. Wileden, L. Osterweil, and A. L. Wolf, "Foundations of the Arcadia Environment Architecture," Proceedings of the 4th Symposium on Software Development Environments, Irvine, CA, 1998.

[24] W. L. Johnson and M. Feather, "Building an Evolution Transformation Library," Proceedings of the 12th International Conference on Software Engineering Nice, France, 1990.