Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models

Alexander Egyed, Emmanuel Letier, and Anthony Finkelstein

Presented by Mohammad Hossein Danesh
Consistency and Inconsistency

- A consistency rule is a
  - Formal condition that evaluates a portion of a model.
  - Modeling constraints in UML are defined as *consistency rules*
- If a consistency rule is violated → *inconsistency*.

- Consistency rules
  - Might be widely applicable or domain specific
  - are written from the perspective of the meta model.
Current Automation in IDEs

Limited to "Quick Fixes" that generate potential resolutions for resolving Java errors.

IBM Rational Rose™, which suggests the set of choices `stream()`, `wait()`, and `connect()` to fix the incorrect message name `play` in a sequence diagram.
Challenges of Current Inconsistency Resolution Approaches

- Implement fixing rules for potential violation locations
  - Requires significant effort as the number of inconsistencies increase

- Must resolve inconsistency without creating a new one
  - Hard for multi paradigm modeling languages such as UML

- Different teams use different rules
- Different set rules may be used at different stages
  - Therefore a ‘hard coded’ resolution mechanism is almost useless.
Video On Demand System

- The Display class handles user IO
  - visualizing movies
  - receiving user input
- Streamer handles server interaction
Samples of Consistency Rules

Rule C1 ensures that a message in a sequence diagram is declared as a method in the receiver’s class.

Operations = Message.Reciever.base.operations

return(operation->name->contains(message.name))

Rule C2 ensures that the behavior of a sequence of messages is allowed by a state machine

Rule C3 ensures that the calling direction of a message is allowed by the calling direction among classes.
Inconsistency 1:
Message *play* is not defined as a method in the class diagram for *Streamer*.

Inconsistency 2:
The state chart does not allow message *play* to follow message *connect*.

Inconsistency 3:
Calling direction of message *draw* does not match association direction between classes *Display* and *Streamer*. 
Tracking Consistency Rules

- This Profiling data is used as a basis for deciding
  - when to re-evaluate what consistency rule instance
  - In essence a rule has to be re-evaluated if a part of the model changed that it previously accessed.
Resolution Objectives

1. Identify all concrete choices for fixing a model element
   A fix should resolve a inconsistency and not cause new ones

2. Assuming we know how to find the location where to fix an inconsistency identify how to make changes

3. Not every change is valid. Identify an automated approach for determining valid choices.

4. Consider the side effects of multiple inconsistencies onto a single location simultaneously
Consideration for Suggesting a Fix to an Inconsistency

1. Fixing Rules for all Locations
   - Tool developer must re-write the fixing rule (manually!) for every location where the inconsistency could be fixed
   - Message Name, Method Name, Receiver
   - A study on 39 models and 24 consistency rules showed:
     - The number of locations is typically around 5-15
     - Require one fixing rule per location type and consistency rule (roughly 24 * 5-15 rules)
   - Writing these functions is intensive and a source of errors.
Consideration for Suggesting a fix to an inconsistency

2. **Fixing Rule for all Consistency Rules** (interplay among multiple consistency rule instances)

- Manually considering all consistency rules for a model element is hard
- Design models typically require the evaluation of a large number of consistency rule instances.
- The small sample had 11 consistency rule instances of which only three returned inconsistencies.

Therefore 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 approach fails in this regard
Consideration for Suggesting a fix to an inconsistency

3. **Consistency Rules differ among Users**
   - 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.
Approach – High level

- Generate Potential Choices
- Eliminate False Choices

Location where to fix inconsistency

Choices how to fix inconsistency
Considerations

1. How do we generate the initial set of choices?

2. How do we know what rules to re-evaluate to eliminate false choices?

3. Is this approach correct and does it scale?
Choice Generation Challenge

- Scalability when using brute force choice generation

- To counter this problem:
  manually custom-tailored functions to syntactical constraints of the modeling language – in particular to its well-formedness criteria
Benefits of Choice Generation

- The difference with fixing rules approach discussed
  - It only require one function per location type

- The functions doesn’t need to consider the impact of multiple consistency rules.
  - Eliminates much of the complexity of writing them.

- Note: In a few cases the functions were based on consistency rules.
### Samples of choice Generation function

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>m:Message.receiver: choices = m.interaction.classifierRoles</td>
<td>m:Message.receiver: choices = m.interaction.classifierRoles</td>
</tr>
<tr>
<td>2</td>
<td>m:Message.name: choices = {} foreach (method in m.receiver.base.methods)</td>
<td>m:Message.name: choices = {} foreach (method in m.receiver.base.methods) choices.insert(method.name)</td>
</tr>
<tr>
<td>3</td>
<td>ae:AssociationEnd.multiplicity choices = {1, 0..1, 1..n, 0..n}</td>
<td>ae:AssociationEnd.multiplicity choices = {1, 0..1, 1..n, 0..n}</td>
</tr>
<tr>
<td>4</td>
<td>c:Class.namespace choices = {} foreach (a in c.associations) foreach (oc in a.classifiers) choices.insert(oc.namespace)</td>
<td>c:Class.namespace choices = {} foreach (a in c.associations) foreach (oc in a.classifiers) choices.insert(oc.namespace)</td>
</tr>
</tbody>
</table>
Choice Reduction
Challenges

- Many instances of consistency rules (typically tens of thousands) and
- Every choice may affect different rule instances
- Therefore re-evaluating all consistency rule instances would not scale

Solution: Relies on performing instant, incremental consistency checking
Inconsistency 1:
Message *play* is not defined as a method in the class diagram for *Streamer*.
Choice Reduction for Inconsistency 1

Changing Message Name
Need to check C1 & C2

Changing Message Receiver
Need to check C1 & C#
Final Choices

Final decision on a choice is done by the designer
Impact of The Change

Can it identify all consistency rule instances affected by a fix?
- In previous work, authors have demonstrated how to identify all affected rule instances when changed model elements are recognized.
- The challenge here is that a change often modifies multiple elements.
- How to analyze the affect on other elements that reference the change.
Trace Impact of Change

The approach must identify

1. All model elements affected by a choice
2. Determine all consistency rule instances affected by that choice

This is accomplished by maintaining records of (back)pointers

These data structures are defined generic and are built once per modeling language.
Scalability & Correctness
Usability Study

- Evaluated on 39 models – size between 100 to 120,000 elements
  - Obtained from industry, reverse engineered or colleagues
  - avionics systems, medical systems, data-centric systems, and closed-loop types
- 24 Type of consistencies
  - In total 223,000 consistency rule instances were evaluated
  - Their level of consistency was also diverse – between 2-26% with an average of 8.4%.
- 14 Types of location were considered
  - In total they occurred 65,379 times.
Results of The Study

- In average only 2.4 valid choices
  - worst case of 69 choices
  - Number of choices did not increase with model size
- Highly scalable in terms of performance.
  - Average 11ms per location
  - 2.2GHz Pentium Processor.
- In average 10.4 locations per inconsistency
  - Total number of choices for fixing an inconsistency:
    10.4 locations * 2.4 choices ≈ 25 choices per location
Results of The Study

- The approach was able to identify false positives
  - False Positives: A false location is a location for which no valid choice exists.
- In average, 11.2% false positions existed
- Among valid locations,
  - 48% had multiple valid choices
  - 40% had only a single valid choice.
- Writing a good choice generator is critical in order to produce valid choices
- Cost of writing fixing rules:
  - $O(\text{#type of consistency rules} \times \text{#types of locations})$ fixing rules
- Cost of writing choice generation functions:
  - $O(\text{#types of locations})$ choice generate rules
Threads to Validity

Location coverage:
- Only 17% relevant locations
- 4% were deemed unchangeable
- The rest were simple undefined.

Every choice is considered separately
- Fixing inconsistency may require several concurrent changes

Did not consider the creation new elements
Most approaches focus on handling inconsistencies between multiple models expressed in a single language.

This paper dealt with multi-paradigm descriptions and its consistency rules are similar to static semantic rules of programming languages.

“living with inconsistencies”
   Benefit in allowing temporary inconsistencies

Fix inconsistencies whenever the designer wants
Related Work analysis

Similarity to Constraint satisfaction problem (CSP)
- Deals with what choices best satisfy a given set of constraints

Difference
- CSP uses “white-box constraints.” - thus known in advance
- Consistency rules in UML are typically black-box constraints.

Similar to repairing data structures in databases or code
- since faults in data structures are repaired through constraint-based reasoning

Difference
- Their approach applies to code
- This approach deals with multi-paradigm modeling.
Conclusion & Future Work

- Systematically explore alternative ways of fixing inconsistencies at different locations in the model and anticipate the impact of such changes on all consistency rules simultaneously.

- Plan to address the need to change multiple model elements simultaneously by developing an adequate set of higher-level model evolution operators that aggregates the application of the elementary changes.
Questions

Writing choice generation cost is less because the author has assumed writing them has the same complexity as fixing rules?? Is this a valid assumption??

The fact that all these comparisons are done on 24 consistency but as consistencies get more the impact of change on multiple elements and tracking that will affect the scalability or not??