Designing a Change-Driven Language for Model Consistency Repair Routines

Advisors: Max Kramer, Michael Langhammer
Reviewers: Prof. Dr. Ralf Reussner, Jun.-Prof. Dr.-Ing. Anne Koziolek

SOFTWARE DESIGN AND QUALITY GROUP
INSTITUTE FOR PROGRAM STRUCTURES AND DATA ORGANIZATION, FACULTY OF INFORMATICS
Motivation

Software-System

PCM

Java

UML

Tests

Motivation & Foundations  ➔  Model Consistency & Repair  ➔  The Response Language  ➔  Evaluation  ➔  Conclusion
Motivation

**UML class diagram**

```java
public interface Account {
    int getBalance();
}
```

**Java code**

```java
public interface Account {
    int getBalance();
}
```

**response**: UpdateInterfaceName
**trigger**: replace value `uml.Interface[name]`
**retrieve required element**: `java.Interface as javaInterf` corresponding to `change.affectedObject`
**execute**: `javaInterf.name = change.newValue`

Motivation & Foundations  ➤  Model Consistency & Repair  ➤  The Response Language  ➤  Evaluation  ➤  Conclusion
Change-Driven MIR Language Family

- **Mapping language:**
  declarative, bidirectional consistency repair

- **Invariant language:**
  consistency constraints specification

- **Response language:**
  imperative, unidirectional consistency repair

Virtual single underlying model
Model Consistency

Consistency Overlap

A relation between model elements that contain dependent information.

Motivation & Foundations  ➤  Model Consistency & Repair  ➤  The Response Language  ➤  Evaluation  ➤  Conclusion
Model Consistency

```java
public interface BankAccount {
    int getBalance();
}

public interface Transaction {
    int getAmount();
}
```
Model Consistency

A constraint that describes an information dependency between instances of two or more metaclasses.
Model Consistency

Consistency Overlap Type

A type of a possible relation between model elements described by a set of consistency constraints.
Model Consistency

Two models are consistent if all their consistency overlaps have fulfilled constraints.

When do elements share a consistency overlap?

Correspondence model

Correspondence

A correspondence consists of two sets of elements of different models that share a consistency overlap.
Change-Driven Consistency Repair

<<interface>>

BankAccount

getBalance(): int

Notify about change

Consistency mechanism

Retrieve corresponding Java interface

Find responsible routines

Consistency-restoring routine

public interface BankAccount {
    int getBalance();
}

Restore consistency

Motivation & Foundations  Model Consistency & Repair  The Response Language  Evaluation  Conclusion
Change-Driven Consistency Repair

- Restriction to atomic changes

**Atomic Model Change**

A model change is atomic if it can be unambiguously described by the old and new value of a single model element or property.

- Possible types of atomic changes are predetermined by the meta-metamodel
Change-Driven Consistency Repair

- A consistency repair routine is responsible for
  - a certain way of repair of
  - a certain consistency overlap type

Structure of a consistency repair routine:

<table>
<thead>
<tr>
<th>Trigger</th>
<th>Check change precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matcher</td>
<td>Check model state precondition</td>
</tr>
<tr>
<td>Effect</td>
<td>Restore consistency</td>
</tr>
</tbody>
</table>
The Response Language: Trigger

trigger: replace value uml.Interface[name]

Can a consistency overlap of the repaired type be affected... …in a way the response repairs it?

- Primarily declarative
- Additional preconditions in imperative code block
The Response Language: Matcher

```
trigger: replace value uml.Interface[name]
retrieve required element: java.Interface as javaInterf
corresponding to change.affectedObject
```

Does the affected object share a consistency overlap?

- Retrieve elements for repair
  - Require their existence/non-existence
  - Restrict with filter function and tags
- Imperative check blocks
The Response Language: Effect

trigger: replace value uml.Interface[name]
retrieve required element: java.Interface as javaInterf corresponding to change.affectedObject
execute: javaInterf.name = change.newValue

- Specify constraints repair: imperative, Turing-complete code block
- Additional language constructs:
  - Create/delete element
  - Add/remove correspondence
The Response Language

Reusable repair routines

```
repair routine: CreateJavaClass
input: pcm.NamedElement as pcmElement
create element: java.Class as javaClass
add correspondence between javaClass and pcmElement
execute: javaClass.name = pcmElement.entityName
```

User interaction

Persistence of models
Evaluation

1. Functionality
   - Reaction to all possible changes
   - Arbitrary repair logic

2. Applicability
   - Case study: PCM Repository to Java
   - Integration tests

3. Benefits
   - Conciseness
   - Relevance of language constructs
Applicability

- Case study: PCM Repository to Java

<table>
<thead>
<tr>
<th>Required responses</th>
<th>Implemented responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>38*</td>
</tr>
</tbody>
</table>

- Language constructs for correspondence model interaction
  - Almost sufficient
  - Limitation: handling of dynamically sized overlaps

- 52 of 53 integration tests successful
Trigger Conciseness

Response language:

```
trigger: insert in list pcm.Repository[components]
```

Xtend implementation:

```
if (change instanceof InsertRootEObject) {
    val typedChange = change as InsertRootEObject;
    if (change.affectedObject instanceof pcm.Repository &&
        change.affectedFeature.name == "components") {
        // Repair routine specification
    }
}
```
Relevance of Constructs

Reasons for providing repair routines

Reasons for calling repair routines

Used language constructs

- Retrieve required element: 59
- Response: 42
- Create element: 21
- Add correspondence: 21
- Delete element: 13
- Retrieve optional element: 5
- Require element absence: 1
- Remove correspondence: 0

For reuse: 19
For chaining: 2
For iterating: 2

Motivation & Foundations ➔ Model Consistency & Repair ➔ The Response Language ➔ Evaluation ➔ Conclusion
Related Work

Problem-Specific Consistency Repair
- Consistency between certain metamodels
- UML-Code (Fujaba, Enterprise Architect)

Declarative Consistency Repair
- Describe consistency constraints, derive repair
- TGGs, QVT-R: Echo, Answer Set Programming

Imperative Consistency Repair
- Specify consistency repair operations
- Event-driven grammars, VIATRA
Conclusion

- The response language
  - Change-driven, imperative consistency repair
  - Reusable repair routines

Future work
- Additional language constructs
- Transitivity of response execution
- Integration of MIR languages
- Evaluation: applicability in further domains, usability
BACKUP
Change-Driven Consistency Repair

Generic Structure of a repair routine:

Step 1: Change precondition
Identify if the change can
• affect a consistency overlap of the repaired type
• in the way the routine repairs it

Step 2: Model state precondition
Identify if the affected element shares a consistency overlap of the repaired type

Step 3: Repair
Perform modifications to repair the consistency overlap
Implementation

- Language implementation with Xtext
- Code blocks
  - Reuse Xbase expression blocks
  - Mapped to Java methods
- Effect constructs mapped to predefined methods
Implementation

- Language implementation with Xtext
- Code blocks are mapped to Java methods
- Effect constructs mapped to methods of state objects

```java
interface ResponseElementState {
    void preprocess();
    void postprocess();
    void addCorrespondingElement(EObject newCorrespondingElement, String tag);
    void removeCorrespondingElement(EObject oldCorrespondingElement);
    void delete();
}
```
Implementation: Example Environment

```
class RepairRoutinesFacade {
    void callExplicitRepairRoutine(...) {
        ...
    }
}

class ImplicitRepairRoutine {
    private extension RepairRoutinesFacade _facade

    void applyRoutine() {
        ...
        callExplicitRepairRoutine(...)
        ...
    }
}
```

Change2CommandTransforming | 1..* | ResponseExecutor | 1..* | Response
Evaluation: Functionality

Reaction to possible changes

- Minimal metamodel
- Responses transfer atomic changes to second model instance
- Unit tests perform possible changes and compare models
  - Correct responses are executed if triggers are defined correctly

Arbitrary repair logic

- Turing complete due to code blocks
Response Example

response: CreatedInnerDeclaration
trigger: insert in list pcm.CompositeDataType[innerDeclaration_CompositeDataType]
match:
  retrieve optional element: java.Class as nonPrimitiveInnerDataTypeClass
corresponding to change.newValue.datatype_InnerDeclaration
effect:
  execute: {
    val innerDataTypeReference =
      createTypeReference(change.newValue.datatype_InnerDeclaration, nonPrimitiveInnerDataTypeClass);
    val compositeDataType = change.newAffectedEObject as CompositeDataType;
    val innerDeclaration = change.newValue;
    callAddInnerDeclarationToCompositeDataType(compositeDataType, innerDeclaration, innerDataTypeReference);
  }
Response Example

routine: AddInnerDeclarationToCompositeDataType
input: pcm.CompositeDataType as compositeDataType,
      pcm.InnerDeclaration as innerDeclaration, java.TypeReference as dataTypeReference
match:
  retrieve required element: java.Class as dataTypeClass
  corresponding to compositeDataType
effect:
  create element: java.Field as innerDataTypeField
  create element: java.ClassMethod as getterMethod
  create element: java.ClassMethod as setterMethod
  add correspondence: innerDataTypeField, innerDeclaration
  add correspondence: getterMethod, innerDeclaration tag with "getter"
  add correspondence: setterMethod, innerDeclaration tag with "setter"
execute: {
  createPrivateField(innerDataTypeField, EcoreUtil.copy(dataTypeReference),
                      innerDeclaration.entityName);
  createSetter(innerDataTypeField, setterMethod);
  createGetter(innerDataTypeField, getterMethod);
  dataTypeClass.members += innerDataTypeField;
  dataTypeClass.members += getterMethod;
  dataTypeClass.members += setterMethod;
  sortMembers(dataTypeClass.members);
}