Graph-Transformation-Based Support for Model Evolution

SegraVis Advanced School on Visual Modeling Techniques

Tom Mens
Institut d’Informatique
Université de Mons-Hainaut
Belgium
Challenge: Model-driven evolution

Observation

Model-driven software engineering approaches do not adequately address software evolution problems.

Need better support (tools, formalisms) for:

- Change propagation
- Impact analysis
- Traceability management
- Inconsistency management
- Version control
- Model refactoring
- Reverse engineering
- Code generation
Goal of this talk

Show how graph transformation theory can help to

- gain better understanding of ...
- improve tool support for ...

... the following activities:

- inconsistency management
- model refactoring
Model Inconsistency Management through Graph Transformation

An Experiment
Goal

Specify model inconsistencies and their resolution strategies as graph transformation rules
Use this formalisation to support the inconsistency management process
- Automate the detection of inconsistencies
- Interactively support the resolution of inconsistencies

Analyse transformation dependencies to optimise this process
- Detect sequential dependencies between resolution rules
- Detect parallel conflicts between resolution rules that cannot be applied together
- Remove redundancy between resolution rules
- Provide “optimal” resolution strategies
Model inconsistency management

Iterative inconsistency resolution process
Example of inconsistency resolution

Illustration of the ripple effect

Let’s start with a simple motivating example of an inconsistent model

Concrete syntax

abstract syntax

C:Class «concrete»

op:Operation «abstract»

instance of

contains

behaviour

dangling operation reference

myStateMachine

op

obj

C

« behaviour »

« instance of »

o:Object

Statemachine

Transition

© Tom Mens, Invited Lecture, SegraVis Summer School, September 2006
Example of inconsistency resolution

Illustration of the ripple effect
Resolution leads to a new inconsistency

Concrete syntax:
- C
  - op
  - « instance of »
  - « behaviour »
  - myStateMachine

Abstract syntax:
- C:Class
  - « concrete »
  - o:Object
  - instance of
  - contains
  - behaviour
  - contains
  - abstract operation
  - myStateMachine
  - op:Operation
  - « abstract »
  - ref:Transition
Example of inconsistency resolution

Illustration of the ripple effect
Resolution leads to 2 new inconsistencies
Example of inconsistency resolution

Illustration of the ripple effect
Resolution removes 1 inconsistency

c:Class «abstract»

op:Operation «abstract»

myStateMachine

C

obj

« behaviour »

op

D:Class «concrete»

o:Object

Statemachine

Transition

contains

Abstract statemachine

contains

refers to

gen

instance of

behaviour

« instance of »

© Tom Mens, Invited Lecture, SegraVis Summer School, September 2006
Example of inconsistency resolution

Illustration of the ripple effect
Resolution finally removes last remaining inconsistency

concrete syntax

abstract syntax

C

« instance of »

obj

myStateMachine

« behaviour »

op

D:Class

«concrete»

o:Object

contains

gen

instance of

Statemachine

contains

refers to

op:Operation

«abstract»

op

© Tom Mens, Invited Lecture, SegraVis Summer School, September 2006
Tool support

**SIRP tool**

*“Simple Iterative Resolution Process”*

An interactive tool for selecting and resolving model inconsistencies
Tool support

SIRP tool in action

Before detecting any inconsistency
Tool support

SIRP tool in action

After detecting all inconsistencies
Tool support

**SIRP tool in action**

- After resolving "**duplicate class name**"
  - Two occurrences of same inconsistency removed
  - Class renamed from “A” to “C”
Tool support

SIRP tool in action

After resolving "classless instance"

* One occurrence of “classless instance” removed
* One occurrence of “instanceless class” removed
Tool support

SIRP tool in action

After resolving "abstract object" (first try)
- One occurrence of "abstract object" removed
- One occurrence of "abstract operation" added!
Tool support

**SIRP tool in action**

- After resolving "abstract object" (second try)
  - One occurrence of “abstract object” removed
  - One occurrence of “nameless instance” removed
  - One occurrence of “instanceless class” added
Tool support

**SIRP tool in action**

- After disabling the "instanceless class" rule
- Two occurrences of "instanceless class" ignored
Tool support

- SIRP tool in action
  - After resolving "abstract operation"
  - One occurrence of "abstract operation" removed
Tool support

SIRP tool in action

After resolving "undefined parameter type"
One occurrence of "undefined parameter type" removed
Tool support

*SIRP* tool in action

After resolving “nameless instance”
- One occurrence of “nameless instance” removed
- No more remaining inconsistencies!
This tool relies on the underlying mechanism of graph transformation for detecting inconsistencies, for proposing resolution rules, and for analysing which of the proposed resolution rules is most appropriate.

But how does this all work?
Tool support

The tool has been implemented on top of the AGG engine (version 1.4)
- AGG is a general-purpose graph transformation tool

We used AGG in the following way
- specify the UML metamodel as a type graph
- specify the models as graphs
- detect and resolve model inconsistencies by means of graph transformation rules
- analyse mutual exclusion relationships and sequential dependencies between inconsistency resolutions by means of critical pair analysis
Step 1: Specify the metamodel

AGG type graph
Specify the UML model

Example of an inconsistent UML model

statemachine diagram

Dangling Operation Reference "selectReverse" is used in the statemachine but is not defined in class AutomaticGear or any of its ancestors

AutomaticGear
- boolean driveSelected
- selectDrive()
- reach2ndSpeed()
- reach3rdSpeed()
- dropBelow2ndSpeed()
- dropBelow3rdSpeed()

Gear
- int noOfGears = 3
- selectNeutral
- selectFirst
- selectSecond
Represent the UML model as a graph

Automatic generation of the corresponding graph representation for the UML state machine

This graph conforms to the type graph specified before
## Step 2: Classify model inconsistencies

<table>
<thead>
<tr>
<th>Class Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dangling Type Reference</td>
<td>An operation has one or more parameters whose types are not specified</td>
</tr>
<tr>
<td>Classless Instance</td>
<td>A model contains an instance specification that is not linked to a class</td>
</tr>
<tr>
<td>Abstract Object</td>
<td>A model contains an instance specification that is an instance of an abstract class that does not have any concrete subclasses.</td>
</tr>
<tr>
<td>Abstract State Machine</td>
<td>An abstract operation is defined in a concrete class.</td>
</tr>
<tr>
<td>Abstract State Machine</td>
<td>A state machine expresses the behaviour of an abstract class that does not have any concrete subclasses.</td>
</tr>
<tr>
<td>Cyclic Composition</td>
<td>A class contains at least one instance of its subclasses through a composition relationship that may lead to an infinite containment of instances of the affected classes.</td>
</tr>
<tr>
<td>Dangling Operation Reference</td>
<td>A state machine contains a transition that refers to an operation that does not belong to any class (or that belongs to a different class than the one whose behaviour is expressed by the state machine).</td>
</tr>
<tr>
<td>Transition Without Operation</td>
<td>A transition does not have a referred operation attached to it.</td>
</tr>
</tbody>
</table>
Example of a model inconsistency

Dangling operation reference

Using graph representation
Step 3: Specify model inconsistencies

Using graph transformation

Example: Dangling operation reference

negative application condition (NAC)

left-hand side (LHS)

right-hand side (RHS)
### Step 4: Identify inconsistency resolutions

<table>
<thead>
<tr>
<th>Dangling Operation Ref.</th>
<th>Res1</th>
<th>Res2</th>
<th>Res3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add the operation to the class (or one of its ancestor classes) whose behaviour is described by the state machine.</td>
<td>Let the transition refer to an existing operation belonging to the class (or one of its ancestors) whose behaviour is described by the state machine.</td>
<td>Remove the reference from the transition to the operation.</td>
</tr>
<tr>
<td></td>
<td>Remove the reference from the transition to the operation.</td>
<td>Remove the transition.</td>
<td>Remove the transition.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Classless Instance</th>
<th>Res1</th>
<th>Res2</th>
<th>Res3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove the instance specification.</td>
<td>Link the instance specification to an existing class.</td>
<td>Link the instance specification to a new class.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abstract Object</th>
<th>Res1</th>
<th>Res2</th>
<th>Res3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change the abstract class into a concrete one.</td>
<td>Add a concrete descendant of the abstract class, and redirect the outgoing instance-of relation of the instance specification to this concrete descendant.</td>
<td>Remove the instance specification.</td>
<td></td>
</tr>
</tbody>
</table>
Step 5: Specify inconsistency resolutions

Using graph transformation

Example: Dangling Operation Reference
Step 6: Detect mutually conflicting resolution rules

Some resolution rules cannot be jointly applied (parallel conflict!)

Conflict graph can be generated by means of critical pair analysis
Informal definition (parallel conflict)

$T_1$ and $T_2$ form a critical pair if

- they can both be applied to the same initial graph $G$
- applying $T_1$ prohibits application of $T_2$
Step 6: Detect mutually conflicting resolution rules

Example of a critical pair detecting a parallel conflict between resolution rules. The resolution rules are not jointly applicable.
Step 7: Detect / analyse sequential dependencies

Some resolution rules may give rise to opportunities for applying other resolution rules

Graph of sequential dependencies generated by AGG
Informal definition (sequential dependency)

$T_2$ sequentially depends on $T_1$ if

- $T_1$ can be applied to $G$ but $T_2$ cannot
- applying $T_1$ triggers application of $T_2$
Step 7: Detect / analyse sequential dependencies

Example of a sequential dependency representing an induced inconsistency. A resolution rule gives rise to a new inconsistency.
Step 7: Detect / analyse sequential dependencies

Some resolution rules may give rise to new model inconsistencies.

- Can be detected by analysing the dependency graph.
Step 7: Detect / analyse sequential dependencies

We can use the dependency graph to detect potential cycles in the resolution process.

Cycles should be avoided, since this implies that the resolution process may continue forever…
Step 8: Tool support revisited

- Open graph and apply all detection rules
- List all found inconsistencies (Conflict nodes in the graph)
- List all resolution rules for selected inconsistency
- Apply selected resolution rule to the graph
- Display resolution history
Step 8: Tool support revisited

Detection or resolution rules may be disabled by the user.
Step 8: Tool support revisited

🚀 A resolution rule may be parametrised
   e.g. DanglingTypeRef-Res3(n,a)

➡️ User needs to provide necessary input values
Step 8: Tool support revisited

A rule may have several possible matches

![Diagram showing possible matches and conflict]

Several matches are available for applying the transformation. Please, make the appropriate choice in the list below:

<table>
<thead>
<tr>
<th>Artifacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class C, InstanceSpecification (unnamed)</td>
</tr>
<tr>
<td>Class B, InstanceSpecification (unnamed)</td>
</tr>
<tr>
<td>Class A, InstanceSpecification (unnamed)</td>
</tr>
</tbody>
</table>

- **Class**: name = “A”
- **Class**: name = “B”
- **Class**: name = “C”
Step 8: Tool support revisited

Future work

- Direct integration of interactive support in a UML modeling environment
- Direct generation of type graph from the UML metamodel
- Direct generation of inconsistency detection rules from given metamodel
- User-friendly specification of resolution rules using some UML representation (Fujaba-like?)
- Support for more complex (composite) resolution rules
Discussion topics

- Optimality
- Expressiveness
- Completeness
- Compositionality
- Termination
Discussion topics

Optimality

- Find an “optimal way” to resolve model inconsistencies
- How can we define “optimality”?  
- Use heuristics and resolution strategies, locally as well as globally
  - Avoid resolution rules that introduce too many new inconsistencies
  - Prefer resolution rules that add new model elements over rules that remove model elements
  - Take into account, and “learn”, resolution rules preferred by the user
  - Other strategies?
Discussion topics

Expressiveness

- Can all model inconsistencies be expressed?
  - Which types of model inconsistency can be expressed?
  - Graphs are well-suited for detecting structural problems
  - Behavioural problems are more difficult to express
  - Possible solution: use other formalisms (e.g. based on description logics) to detect behavioural problems
    – Integrate both formalisms in a common tool for model inconsistency management

- Can all resolution rules be expressed?
Discussion topics

Completeness

- Are all possible model inconsistencies expressed?
  - Can they be generated automatically?
- Are all possible ways to resolve a particular inconsistency covered by the resolution rules?
  - Can they be generated automatically?
Discussion topics

Compositionality

How to define resolution strategies as a composition of primitive resolution rules?
- Using sequencing, branching, looping constructs
- How does this affect the conflict and dependency analysis?
Discussion topics

Termination

Given a set of resolution rules, can we prove that it will resolve all detected inconsistencies and terminate in a finite amount of time?

Rely on termination results of graph transformation
Conclusion

Graph transformation seems to be a viable option to support certain activities in model-driven software engineering.

But it is no “silver bullet”

Alternative mechanisms and formalisms are also needed.
Conclusion

There is still a long way to go ...

Questions?