Transformational Model Evolution with Tests as Observations

Bernhard Rumpe
Software & Systems Engineering
Computer Science
Munich University of Technology
New International Journal on Software and Systems Modeling

Published by Springer
Editors-In-Chief:
Robert France,
Bernhard Rumpe

First issue: published recently.

Content:
Architectures
Testing
Realtime
Model-based development
Formal Methods
Tools
Transformation
Foundations
UML
etc.

Subscription and article submission www.SoSyM.org
Contents

• Some comments on the UML
• Transformations as model evolution
• Example: Refinement of Broy’ components
• Putting transformations into practice: Automated tests
• Example: Adapting an OO data structure
Why UML is relevant

• Common language reduces the effort need to understand each other
• People like to think visually
• Standards help in tool support (and reuse of tools)
• ...
• UML is a perfect vehicle to transfer scientific results to practitioners
• It is a myth that diagrams are necessarily informal!
• Formal use of diagrams is still is a research topic on its own

• But certain kinds of people cannot or don’t want to think in diagrams for a variety of reasons
  – Edsger Dijkstra, Bertrand Meyer, Kent Beck
• So UML is useful for many purposes in a variety of projects, but not everywhere.
Problems with the UML

• UML suffers from a precise semantics, Why?
• UML is large
• Characteristics: Formal language \(\cong\) UML \(\cong\) natural language
• Two dimensions of the problem:
  – a) „Internal“ Semantics: Mapping UML to a formal domain
  – b) „Real-World“ connection: How UML concepts map to real-world concepts?
• Formality is not a technical, but a social/political problem
  – Stakeholders are tool vendors, academics, users in different domains, ...

• But, more important are useful analysis and generation techniques
  – Context conditions
  – Code generation
  – David Harel’s Play-In/Play-Out-Approach
  – Model Checking
  – Model transformations
Future of UML

• UML is being standardized
• This means most of its current flaws will be standardized

• Solution: Alternative „European Set of Modeling Languages (EML)“?

• Some basic questions to be solved:
  – How small/focused/general shall EML be?
  – How integrated (syntactically or methodically) should EML notations be?
  – How much does research have to care of current background of average developers?
  – Should EML be based on a new paradigm (components/services)?
Contents

• Some comments on the UML
• Transformations as model evolution
• Example: Refinement of Broy’ components
• Putting transformations into practice: Automated tests
• Example: Adapting an OO data structure
Software Evolution

• “Software evolution is the key problem in software development.” Nierstrasz

• Requirements change
• Platforms and system contexts evolve
• Bugs needs to be fixed
• Time and space optimisations are desired

• ? Existing software needs to be evolved
• ? Code as well as models need to be adapted to keep them consistent
Approach presented here

• Evolution in the small supports evolution in the large

• Evolution in the small:
  – Transformation rules = set of small, manageable and systematic steps

• Goals of transformations:
  – reasoning,
  – deriving implementation oriented artefacts,
  – building abstractions e.g. for reengineering,
  – evolutionary improvement

• Transformation calculi can serve as technical basis for an evolutionary approach to software development
Examples for Transformational Development

• Mathematical calculi for reasoning

• State machine transformations for error completion, determinism, …

• Stepwise refinement of programs (Bauer, Partsch) for software development

• Hoare calculus for reasoning

• Refactoring (Opdyke, Fowler) for evolution

• …. 
Contents

• Some comments on the UML
• Transformations as model evolution
  • Example: Refinement of Broy’ components
• Putting transformations into practice: Automated tests
• Example: Adapting an OO data structure
Streams & Behaviors (This part is based on the talk by Manfred Broy, but UML capsule diagrams are very similar)

- Communication histories over channels are modeled by **streams**: 
  - **Timed** streams \( M^\mathbb{R} = \langle 1, 2, a, \emptyset, b, 3, c, \ldots \rangle \)

- Channel valuations assign streams to channel names: \( \overrightarrow{C} = C \rightarrow M^\mathbb{R} \)

- An I/O behavior relates input and output channel valuations: \( \beta : \overrightarrow{I} \rightarrow \mathcal{P}(\overrightarrow{O}) \)

Composition of behaviors can be modeled graphically:
Kinds of Transformations

• Behavioral Refinement:
  – A behavior $\beta'$ is a refinement of a behavior $\beta$

\[
\forall x : \beta'(x) \subseteq \beta(x)
\]

• Structural Refinement (Decomposition)

• Evolution of architecture (Refactoring)
Semantics of Transformations

Transformation rules:
- Add or remove components
- Add or remove channels
- Refine component behavior
- Fold and unfold subsystems

Blackbox behavior

Refinement relation =, ?

Blackbox behavior
Example: Communication System

- Data (Consisting of key and value) is accepted via „In“
- and transmitted to the „Remote Data Base“ (RDB)
- Upon sending a key, the requested value is sent

- Problem:
  – Transmission from Stub to RDB shall be encrypted
- Solution:
  – We evolve the part of the system, we are currently focusing on
Example: Communication System

• Step 0:
  – Decide what the „observed behavior“ will be that shall not be changed.
  – Here, we group the observed channels into a component
Example: Communication System

- Step 1:
  - Add encryption and decryption components
  - No connection to the rest of the system: Nothing bad can happen
Example: Communication System

- Step 2:
  - Define signature and behavior of new components (may be we reuse of the shelf components?)
  - Still no connection to the rest of the system: Nothing bad can happen
Example: Communication System

- Step 3:
  - connect Input and output channels
  - RDB now has an additional input channel, but doesn't use it yet
  - Still nothing bad can happen
Example: Communication System

- Step 4:
  - establish invariant between channels:
    - CData = encrypt* (Data)
    - Data’ = Data (modulo time)
  - RDB‘ now can use Data‘ instead of Data
Example: Communication System

- Step 5:
  - Remove unused channel Data
Example: Communication System

- Step 6:
  - Fold new parts into subcomponents
Contents

• Some comments on the UML
• Transformations as model evolution
• Example: Refinement of Broy’ components
• Putting transformations into practice: Automated tests
• Example: Adapting an OO data structure
Context conditions of transformations

• Transformations have context conditions
• Most transformation conditions are syntactical and can be resolved by CASE tools.
• For the behavioral refinement rules invariants about channels resp. component behaviors are necessary
  – For dataflow systems: inductive reasoning
  – Or a pragmatic approach: testing ...
• But tests need executability:
• Therefore the modeling language should be executable or have a strong connection to a programming language
Model-based development

- Models as central notation

- An adapted version of the UML serves as central notation for the development
- UML is programming, test and modelling language at the same time

Diagrams:
- Models
- static analysis
- rapid prototyping
- code generation
- automated tests
- refactoring/transformation
- documentation
Model-based “programming”

- Two kinds of models are used: the system and executable tests

UML models check mutual conformance
Automated testing

• Tests must be
  – automatic to allow frequent repetition
  – deterministic with determined result
  – free of side effects
  – need effective execution
• Tests are a demonstration of quality
  – Tests (almost) ensure correctness of transformations
• Unit tests are developed together with the system
• Acceptance tests are provided by the customer
• Tests are developed through
  – discussion with customer, test coverage metrics, boundary analyses, etc.
Typical infrastructure of an automated test

- Principle: use
  - “source” object diagram (OD) for test data
  - “expected” OD and OCL as oracle
  - sequence diagram (SD) or Java as test driver

expected result and/or OCL-contract as test oracle
Modelling invariants with object diagrams and OCL

- OCL and object diagrams can be mutually integrated
- E.g. object diagrams used as predicates within OCL statements

\[
\text{inv:}
\forall \text{Auction } a: \text{OD.Open} \implies \exists \text{TextMessage } w: \text{OD.Welcome}
\]

\[
\begin{array}{c}
\text{a:Auction} \\
\text{OD.Open} \\
\text{OD.Welcome}
\end{array}
\]

\[
\begin{array}{c}
\text{a:Auction} \\
\text{TimingPolicy} \\
\text{status} == \text{OPEN}
\end{array}
\]

\[
\begin{array}{c}
\text{a:Auction} \\
\text{TextMessage} \\
\text{content} == \text{“Welcome.”}
\end{array}
\]

\[
\text{describes type of diagram and gives it a name}
\]
Sequence diagram: test driver and interaction description

- linear structure of an exemplaric system run
- + OCL for property description

```
copper912: Auction
  «trigger» handleBid(bid)
  validateBid(bid)
  return OK
  newCurrentClosingTime(copper912, bid)
  return t

bidPol: BiddingPolicy

getTime(bid)
  return ok

timePol: TimingPolicy

getTime(bid)
  return ok

  t.time == bid.time + extensionTime
```

OCL constraints describe properties during the test run
Test as observation for transformation

- Both structure and behaviour are observed by tests

\[
\text{test} = \text{driver and “observer”}
\]

setup & call

<table>
<thead>
<tr>
<th>snapshots of the test run</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>observe creation</th>
</tr>
</thead>
<tbody>
<tr>
<td>observe interaction</td>
</tr>
<tr>
<td>check property</td>
</tr>
<tr>
<td>compare with expected result</td>
</tr>
</tbody>
</table>

time axis
Validation of transformations

- Observation remains invariant under transformations

\[\text{test} = \text{driver and “observer”}\]

- But: structure is likely to change through transformation
- So: use proper **abstractions** and **published interfaces** for **acceptance tests**
Tests as abstract observation

- **Unit tests** check methods, single classes and small collaborations:
  - they need not be abstract
  - but may become obsolete or need to be changed as well when tested element is transformed

- **Acceptance tests** capture user input/observation
  - Tests should be as abstract as possible
  - **Query-methods** instead of direct attribute access
    - more stable when data structure is changed
  - **OCL property definitions** instead of exact description of result
    - descriptive properties allow a range of results
  - Ignore uninteresting objects and attribute values
  - Observations of interesting interactions only
    - sequence diagram need not show all interactions of the test
  - Explicitly published interfaces
Contents

• Some comments on the UML
• Transformations as model evolution
• Example: Refinement of Broy’ components
• Putting transformations into practice: Automated tests
• Example: Adapting an OO data structure
Practical example: changing OO data structures

A series of steps to apply:

1. Identify old data structure:
   here: long to be replaced by Money
   
<table>
<thead>
<tr>
<th>Auction</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>currentBidInCent</td>
</tr>
</tbody>
</table>

2. Add new datastructure + queries + compile
   
<table>
<thead>
<tr>
<th>Auction</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>currentBidInCent</td>
</tr>
<tr>
<td>Money</td>
<td>bestBid</td>
</tr>
</tbody>
</table>

3. Identify invariants to relate both
   
   ```
   context Auction inv M:
   currentBidInCent == bestBid.valueInCent()
   ```

4. Add code for new data structure & invariants wherever old data structure is changed + compile & run tests
   
   ```
   currentBidInCent = ... 
   bestBid.setValue... 
   assert M
   ```

5. Modify places where old data structure was used + compile & run tests
   
   ```
   = ... currentBidInCent ... 
   ? 
   = ... bestBid.valueInCent() ... 
   ```

6. Simplify + compile & run tests

7. Remove old data structure + compile & run tests
   
<table>
<thead>
<tr>
<th>Auction</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Money</td>
<td>bestBid</td>
</tr>
</tbody>
</table>
Conclusion

• Transformational techniques assist software evolution

• Whatever models and code used: they need to evolve consistently

• Intelligent use of models

• Model-based software development is one of the next big issues in software engineering
  – analysis and transformational techniques are the key
  – but: a practical (syntactic and methodological) flavour is necessary