PAPYRUS ROUND-TRIP ENGINEERING

Van Cam Pham, Shuai Li, Ansgar Radermacher, Cédric Dumoulin, Chokri Mraidha, Rémi Schnekenburger, François Le Fèvre, Sébastien Gérard

{first_name}.{last_name}@cea.fr
What is round-trip engineering?

“The ability to automatically maintain the consistency of multiple, changing software artifacts, in software development environments/tools, is commonly referred to as round-trip engineering”

- Related to traditional software engineering disciplines:
  - Forward engineering: creating software from specifications
  - Reverse engineering: creating specifications from existing software

- Round-trip engineering adds synchronization of existing artifacts that evolved **concurrently** by **incrementally** updating each artifact to reflect **changes** made to the other artifact

- Round-trip generalizes both forward and reverse engineering

INTRODUCTION

A real need for round-trip engineering

- Previously, UML-RT runtime code was manually reversed to a Papyrus UML model and code was generated
- The task was “long” and “tedious”
- In the mean time model and code evolved concurrently

A real problem of synchronization

Papyrus offer for round-trip engineering

- UML modeler and C++ code generator are released…
- …but reverse and incremental update tool are still in development

Propose a IDE for C++ round-trip engineering
CONTENT

- Introduction
- Round-trip Use-cases and Scenarios
- Implementation Technologies
- Demo Video
- Conclusion and Future Work
A developer who works mostly with models

User of the Papyrus Round-trip IDE

A developer who works mostly with code

Existing feature in Papyrus

New feature in Papyrus

Code change detection framework

CDT model listener  File watcher

Code comparator

CDT editor  Non-Eclipse editor
Initialization step

1. Reverse to new model + validation

2. Code generation

3. Modification

4. Reverse (Overwrite)

(Manual) Validation is needed because we use some patterns to transform pure C code (not C++) to an object-oriented model (e.g. file scope functions, extern variables)

Scenario 1: only code is modified
ROUND-TRIP USE-CASES AND SCENARIOS

Initialization step

1. Reverse to new model + validation

Model

2. Code generation

Modified model

3. Modification

Generated code

Generated code

4. Code generation

Scenario 2: only model is modified
ROUND-TRIP USE-CASES AND SCENARIOS

Initialization step

1. Reverse to new model + validation

Model

Code generation

- Several possible strategies
- We propose one based on comparing manually modified code with "image of model"
- Image of model = representation of model as code

Modified model

2. Code generation

Generated code

3. Modification

Modified code

4. Merge

Scenario 3: model and code are both modified

Legacy code

Modification

- Several possible strategies
- We propose one based on comparing manually modified code with "image of model"
- Image of model = representation of model as code
Scenario 3.1: merge by creating an image of the model as code

1. Code generation
2. Code compare
3. Reverse (merge)

“Reverse (merge)” is an incremental update of the model (preserve non-affected elements)

Scenario 3.2: merge by creating an image of the code as model

1. Reverse (overwrite)
2. Model compare
3. Code generation (optional: incremental)
Scenario 3.1: merge by creating an image of the model as code

1. Code generation
2. Code sync.
3. Reverse (merge)

Scenario 3.2: merge by creating an image of the code as model

2. Model sync.
1. Reverse (overwrite)
3. Code generation (optional: incremental)
## Technology comparison

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Merge using code</th>
<th>Merge using model</th>
<th>Winner</th>
</tr>
</thead>
<tbody>
<tr>
<td>UI usability</td>
<td>Suited for algorithmic changes</td>
<td>Suited for architectural changes</td>
<td>Draw</td>
</tr>
<tr>
<td>Change detector</td>
<td>CDT listener is unreliable and not fine-grained</td>
<td>EMF IncQuery is reliable and fine-grained</td>
<td>Merge using model</td>
</tr>
<tr>
<td>Comparison tool</td>
<td>Many mature comparators (but basic can be messy)</td>
<td>EMF Compare comparison fine-grained in Papyrus, needs some UI work outside of Papyrus, xmi:id limitation needs to be leveraged</td>
<td>Merge using code</td>
</tr>
<tr>
<td>Overall robustness</td>
<td>“Reverse (merge)” operation is dependent on code changes detector and handling of changes, which in the current state is error-prone</td>
<td>Incremental code generation is based on more reliable EMF IncQuery AND it’s optional</td>
<td>Merge using model</td>
</tr>
<tr>
<td>Technology dependency</td>
<td>CDT</td>
<td>EMF technologies</td>
<td>Draw</td>
</tr>
</tbody>
</table>
Summary

- Papyrus round-trip engineering proof of concept
- Most limitations are on the implementation side due to technology issues

Current and future work

- Unitary testing and debugging of reverse tool for robustness
- Improvement of code generator
- Test on several real case studies:
  - UML-RT runtime
  - Diversity
  - Embedded applications developed by CEA in C++
  - LEGO EV3 C++ library
- Java version (integration of Cédric Dumoulin’s work)
- Research work (thesis of Van Cam Pham): incremental code generation

Roadmap and project

- Target Eclipse Neon M6 milestone (end Q1 2016) for release of round-trip features
- European project on this subject being proposed?