The Aspectix Transformation Process Language
Detailed Transformation for Middleware-Based Software

Andreas I. Schmied
(andreas.schmied@uni-ulm.de)

Distributed Systems Lab, Ulm University, Germany
Agenda

• Requirements on Transformations

• System Architecture and Concepts

• Language Features by Example

• Prospectus: Composition Issues
Motivation

Distributed applications need support by middleware

• Generation: interface $\mapsto$ proxy code, servant adapters, ...
Motivation

Distributed applications need support by middleware

- Generation: interface → proxy code, servant adapters, ...

- Example: Fault-tolerance by replication

Client/Server Model

Replicated Servants
Motivation

Distributed applications need support by middleware

- Generation: interface $\rightarrow$ proxy code, servant adapters, . . .
- Not sufficient for Aspectix programming model

- Example: Fault-tolerance by replication

Client/Server Model

Fragmented Object Model

Replicated Servants
Motivation

Distributed applications need support by middleware

- Generation: interface → proxy code, servant adapters, ...
  - Not sufficient for Aspectix programming model

- Example: Fault-tolerance by replication

Client/Server Model
  Replicated Servants

Fragmented Object Model
  Replicated Fragments
Requirements

Automatically derive feature/middleware-enabled applications

- Adapt unprepared applications
  - Fine-grained manipulation of implementation sourcecode

AOP may help...
Requirements

Automatically derive feature/middleware-enabled applications

- Adapt unprepared applications
  - Fine-grained manipulation of implementation sourcecode
    - External metadata, annotated interface descriptions
    - Several input sources of different languages
    - Specialised application variants
    - Multiple output targets
Requirements

Automatically derive feature/middleware-enabled applications

- Adapt unprepared applications
  - Fine-grained manipulation of implementation sourcecode
    - External metadata, annotated interface descriptions
    - Several input sources of different languages
    - Specialised application variants
    - Multiple output targets

- AOP may help...
  - Granularity too coarse-grained
  - No statement/expression level access
Requirements

Multiple non-functional concerns at once

- Unforeseen, due to separate development teams
  - Need over-all composite transformation
Requirements

Multiple non-functional concerns at once

• Unforeseen, due to separate development teams
  ➔ Need over-all composite transformation

Objectives

• Comprehend semantics of composite transformations
• Controlled deviation from original application
• Find collisions and give rich diagnostic aid
• Solve collisions
Requirements

Multiple non-functional concerns at once

- Unforeseen, due to separate development teams
  
  ➔ Need over-all composite transformation

Objectives

- Comprehend semantics of composite transformations
- Controlled deviation from original application
- Find collisions and give rich diagnostic aid
- Solve collisions

➔ Own transformation language: *Aspectix TPL*
System Architecture

- AST/MOF-based with few metamodel constraints
- Purely syntactic and semantic models
- Multi-stage, multi-model transformations
Main Concept: Transformation Process (TP)

- Self-contained, reusable transformation task
- No pattern-based mapping
- No high-level aspect language
System Architecture

Main Concept: Transformation Process (TP)

- Self-contained, reusable transformation task
- No pattern-based mapping
- No high-level aspect language

→ Basic transformation “assembly language”
- Primitive operators on model graphs
- List-based queries, expressions
Example: Replication + Synchronization

Determinism of threads is crucial for replication

➡ Replace Java VM synchronization with own monitor logic
  • Substitute synchronized modifier, wait, notify, . . .

```java
synchronized void m() {
    sth();
}

void m() {
    PV pv = getObjectMonitor();
    try {
        pv.lock();
        sth();
    } finally {
        pv.unlock();
    }
}(Listings shortened for clarity)
```
Example: Replication + Synchronization

Determinism of threads is crucial for replication
  ➔ Replace Java VM synchronization with own monitor logic
    • Substitute synchronized modifier, wait, notify, . . .

Demonstration:

```java
synchronized void m() {
  sth();
}
```

```java
void m() {
  PV pv = getObjectMonitor();
  try {
    pv.lock();
    sth();
  } finally {
    pv.unlock();
  }
}
```

(Listings shortened for clarity)
Example: Replication + Synchronization

(module adk.repl.sync.test

(model tree "java ./src-in ./src-out")

(process main

(parse 'BlockStatement "PV pv = getObjectMonitor();" )=INIT
(parse 'BlockStatement "pv.lock();" )=LOCK
(parse 'BlockStatement "pv.unlock();" )=UNLOCK


SYMS.(replaceSyncMethods INIT LOCK UNLOCK)

)

(process replaceSyncMethods ...) ; next slide)
Example: Replication + Synchronization

```plaintext
(process replaceSyncMethods

(_1=INIT _2=LOCK _3=UNLOCK)

(parse 'Statement "try {} finally {}")=NewTRY

(NewTRY.Block.append LOCK _.Block.?)
(NewTRY.finally_.Block.append UNLOCK)

(Modifiers.remove 'synchronized_)
(Block.clear)
(Block.append INIT NewTRY)
)
```
Language Features

- Few elementary operators on graphs
- Model traversal with queries, predicates, dynamic typing
- Implicit iteration in context-bound paths
- Completely list-based expressions
- Labels and sophisticated referencing
- Multi-model access in varying metamodels

```
(model i:tree "idl ...")
(model m:mof "javaml ...")
(model n:jmod)

(i ".[Module].Interface=I.( 
  (m:'somewhere'.remove m:'Classifier I.name+"_Stub" ...) 
  (new n:'Class I.name+"FragIfc" ...) 
))
```
Prospectus: Composition Issues

• Objectives (briefly)
  • Comprehend semantics of single TPs
  • Yield sound composite TP, or
  • Refuse with rich diagnostics
Prospectus: Composition Issues

- Objectives (briefly)
  - Comprehend semantics of single TPs
  - Yield sound composite TP, or
  - Refuse with rich diagnostics

- Challenges
  - Superfluous operations: create+delete
  - Repeated or contradictory operations: double move
  - Cyclic dependencies: “$TP_A < TP_B < TP_C < TP_A$”
  - Unstable qualifiers: “At the beginning”
  - Unstable quantifiers: “For all types”
Prospectus: Approach

- Reason about operator graph
  - Basic semantics of operators known
  - Metamodels introduce semantics on operator $\times$ target

$\text{Modifiers.remove} \neq \text{Methods.remove}$

- Calculate effective range of operators
  - Overlapping targets may yield collisions
Prospectus: Approach

- Break strict serialisation of TP parts
Prospectus: Approach

- Break strict serialisation of TP parts
- Annotations relate parts for composition
Prospectus: Approach

- Break strict serialisation of TP parts
- Annotations relate parts for composition
- Consolidate TP parts
Prospectus: Approach

• Break strict serialisation of TP parts
• Annotations relate parts for composition
• Consolidate TP parts
Prospectus: Approach

- Break strict serialisation of TP parts
- Annotations relate parts for composition
- Consolidate TP parts
- Enumerate alternative serialisations

→ TPs must be prepared for composition

2 alternative serialisations

C → E → D → B
B → C → E → D
Conclusion

- Middleware-features for unprepared applications
  - Multi-model/-concern transformations

- Lowlevel transformation language TPL
  - Prototype in Java for ANTLR/JMI-based models

- Ideas how to cope with composition issues
  - Some early experiments successful

Thank you for listening!
(andreas.schmied@uni-ulm.de)