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

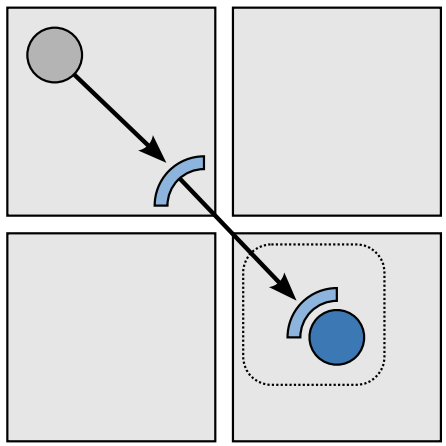
Agenda

- Requirements on Transformations
- System Architecture and Concepts
- Language Features by Example
- Prospectus: Composition Issues

Motivation

Distributed applications need support by middleware

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



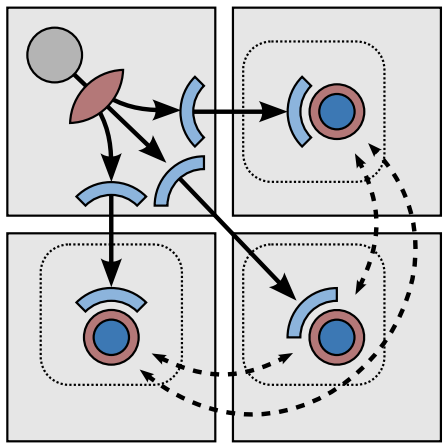
Client/Server Model

Motivation

Distributed applications need support by middleware

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

- Example: Fault-tolerance by replication



Client/Server Model

Replicated Servants

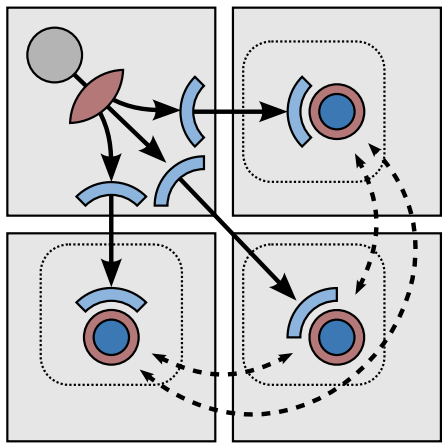
Motivation

Distributed applications need support by middleware

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

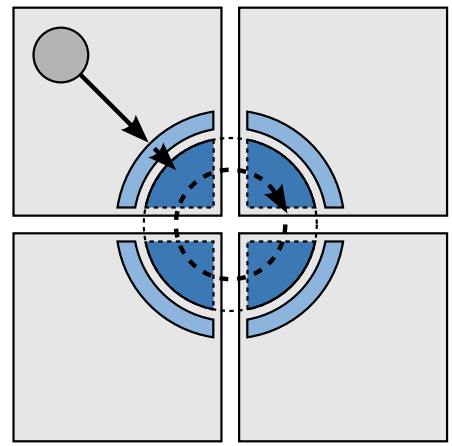
➜ Not sufficient for Aspectix programming model

- Example: Fault-tolerance by replication



Client/Server Model

Replicated Servants



Fragmented Object Model

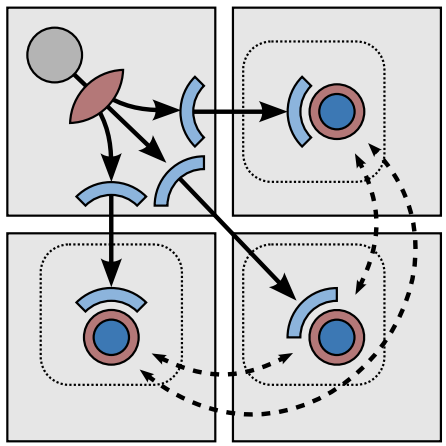
Motivation

Distributed applications need support by middleware

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

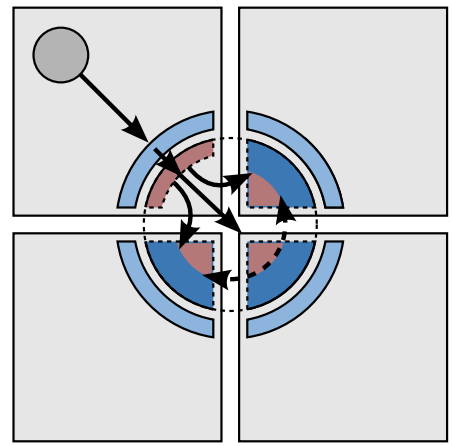
➜ Not sufficient for Aspectix programming model

- 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

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

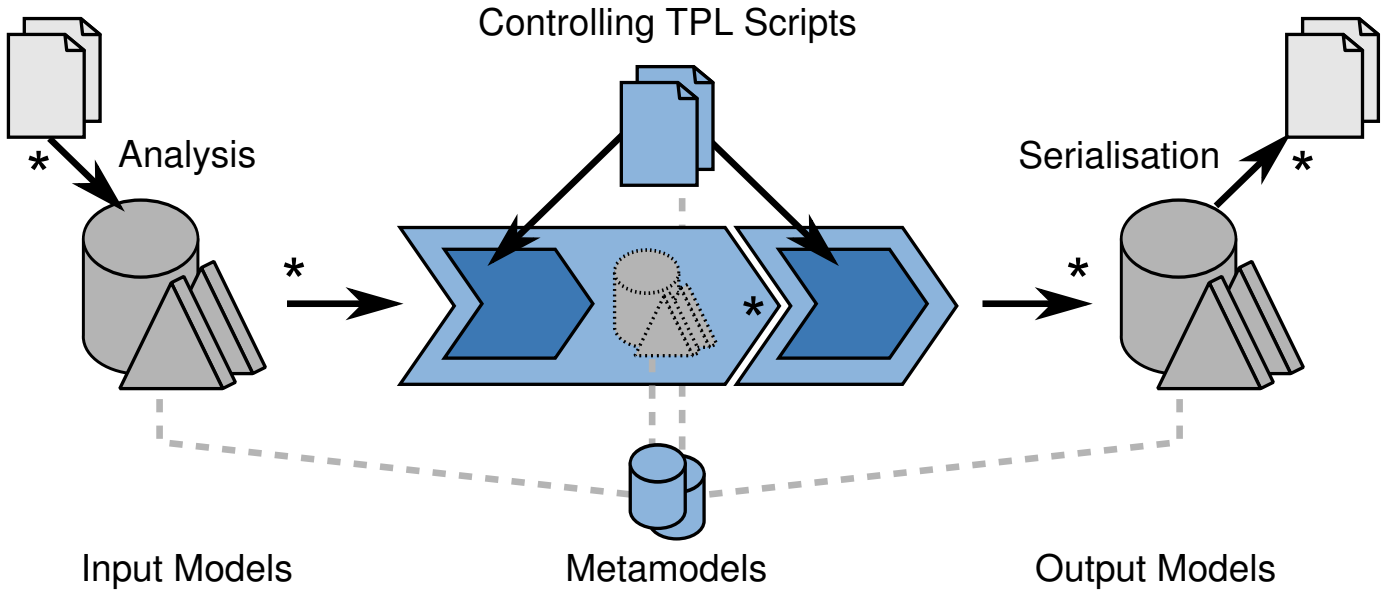
➜ Need over-all composite transformation

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

System Architecture

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, ...

Example: Replication + Synchronization

Determinism of threads is crucial for replication

➜ Replace Java VM synchronization with own monitor logic

- Substitute synchronized modifier, wait, notify, ...

Demonstration:

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

➜

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

(Listings shortened for clarity)

Example: Replication + Synchronization

```
1   (module adk.repl.sync.test
2
3     (model tree "java ./src-in ./src-out")
4
5     (process main
6
7         (parse 'BlockStatement "PV pv = getObjectMonitor();" )=INIT
8         (parse 'BlockStatement "pv.lock();" )=LOCK
9         (parse 'BlockStatement "pv.unlock();" )=UNLOCK
10
11        (`org.aspectix`.[Package].Classifier
12          .Block.Method[Modifiers.? |= 'synchronized_] )=SYMS
13
14        SYMS.(replaceSyncMethods INIT LOCK UNLOCK)
15    )
16
17    (process replaceSyncMethods ...) ; next slide
18  )
```

Example: Replication + Synchronization

```
1   (process replaceSyncMethods
2
3     (_1=INIT _2=LOCK _3=UNLOCK)
4
5     (parse 'Statement "try {} finally {}")=NewTRY
6
7     (NewTRY.Block.append LOCK _.Block.?)
8     (NewTRY.finally_.Block.append UNLOCK)
9
10    (Modifiers.remove 'synchronized_)
11    (Block.clear)
12    (Block.append INIT NewTRY)
13  )
```

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

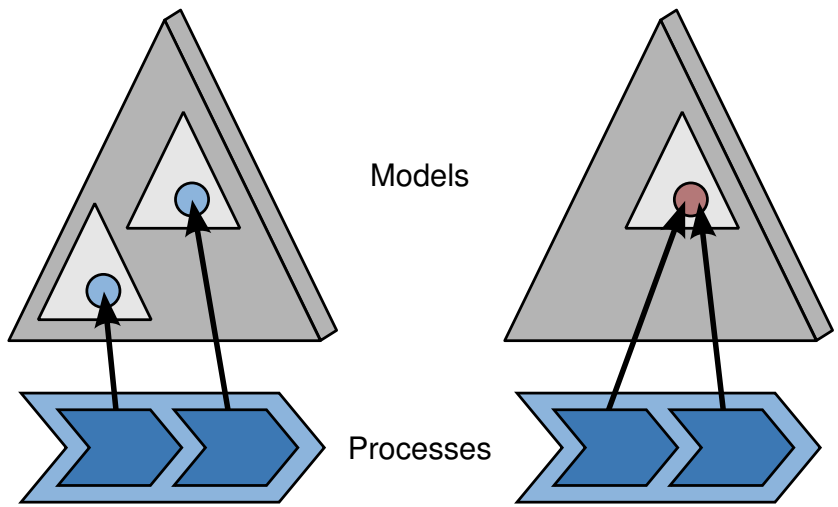
- Challenges
  - Superfluous operations: create+delete
  - Repeated or contradictory operations: double move
  - Cyclic dependencies: "$TP_A < TP_B < TP_C < TP_A$"
  - Unstable qualifiers: "At the beginning"
  - Unstable quantifiers: "For all types"

Prospectus: Approach

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

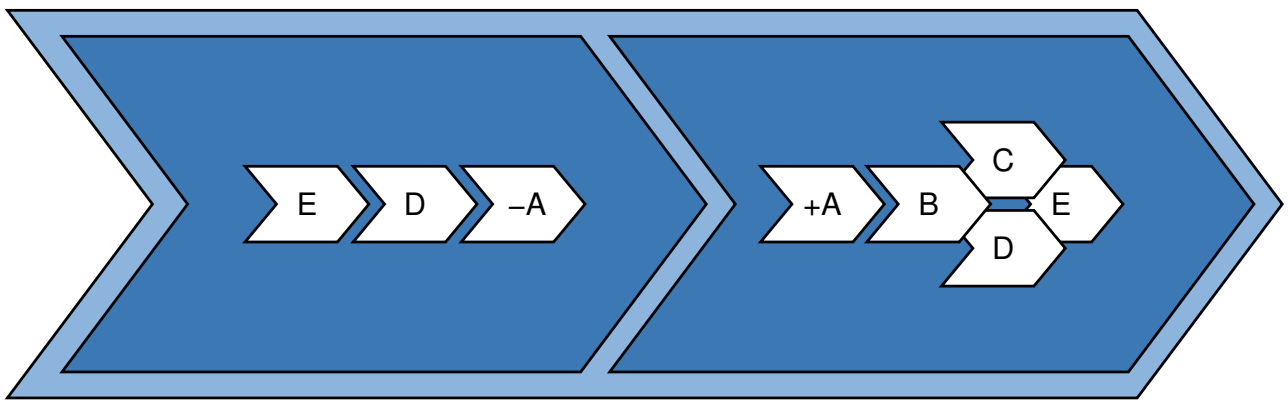
    Modifiers.remove $\neq$ Methods.remove

- Calculate effective range of operators
  - Overlapping targets may yield collisions



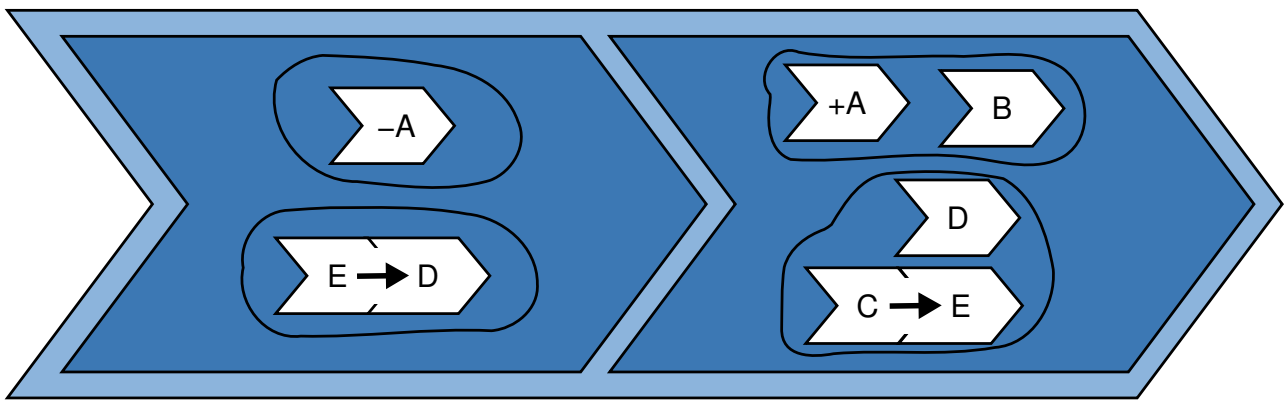
Prospectus: Approach

- Break strict serialisation of TP parts



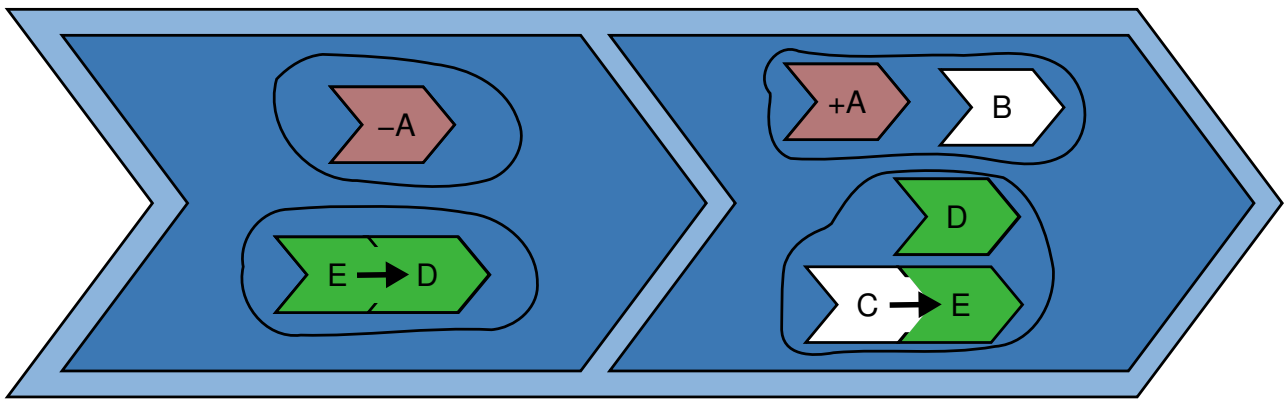
Prospectus: Approach

- Break strict serialisation of TP parts
- Annotations relate parts for composition



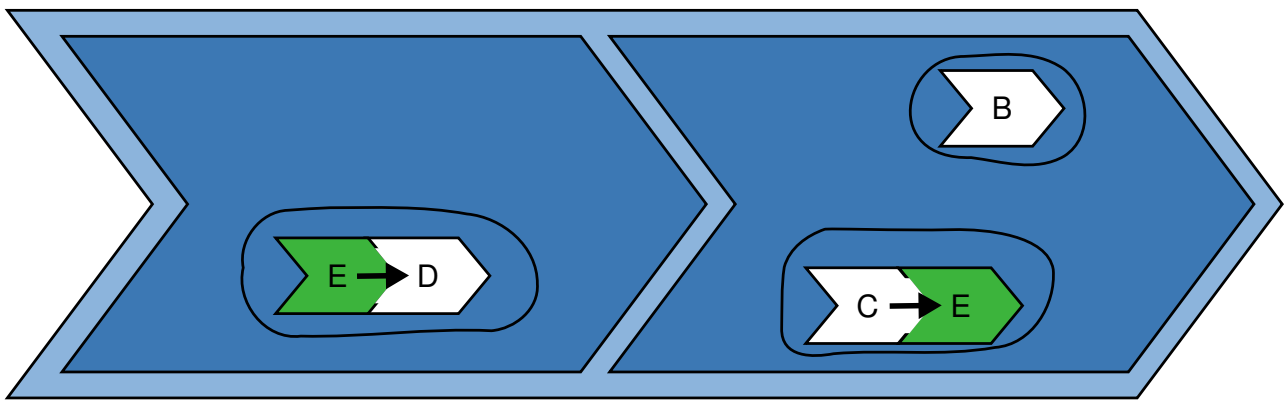
Prospectus: Approach

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



Prospectus: Approach

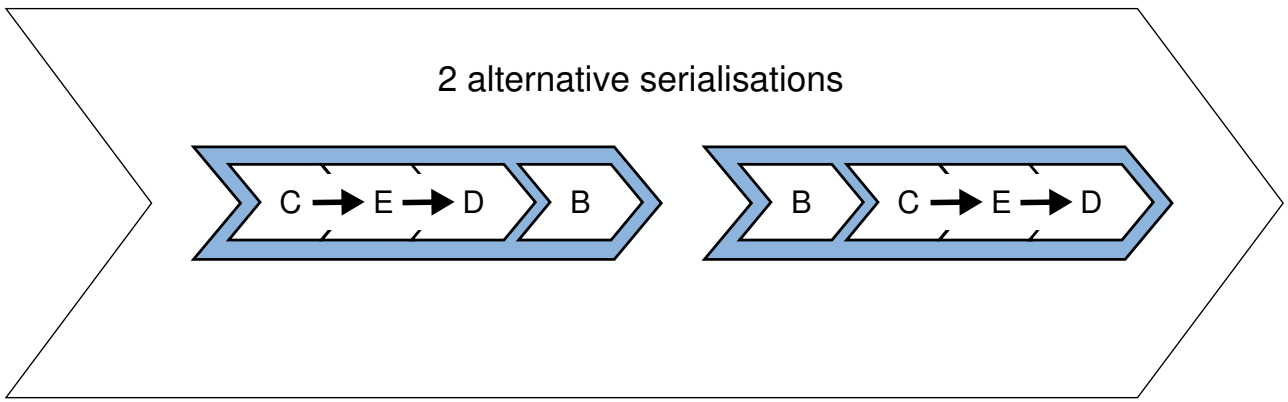
- Break strict serialisation of TP parts
- Annotations relate parts for composition
- Consolidate TP parts



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



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)