

# Trace Analysis for Aspect Application

**Maximilian Störzer, Jens Krinke, Silvia Breu**  
**University of Passau**



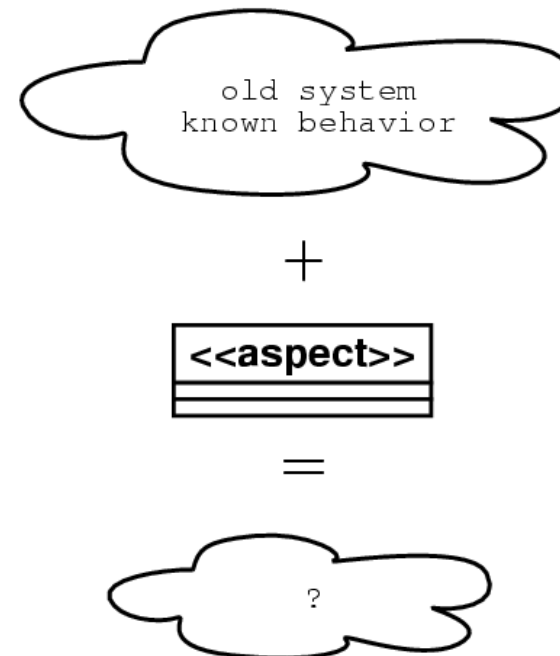


## Motivation, Scenario

Consider the following common scenario:

- We have an existing system
- We have new requirements
- An aspect could avoid invasive changes of source

Genuine strength of AOP,  
so everything is fine, but . . .



→ **What does the aspect really do with our system?**



## How to find out

Base technique: **regression testing**

→ run regression test suite on new woven program.

Problems:

- Testing can **never prove** the absence of bugs, only show their presence.
  - Aspect influence might **not** be **observable** directly, especially for non-functional requirements.
- Better use static analysis to *guarantee* noninterference:
- ✗ Sufficient analysis techniques are **not available** (yet)



# A different approach: Tracing

## Idea:

run regression tests,  
trace programs.

## Traces reveal

- **internal** program flow
- show otherwise non-observable behavior

## Problem: Traces are huge

→ programmers get lost

Telecom example: 160 lines.

```
...
--> void telecom.AbstractSimulation.run(): telecom....
--> telecom.Timing(): telecom.Timing
<-- telecom.Timing(): telecom.Timing
--> telecom.Customer(String, int): telecom.Customer
<-- telecom.Customer(String, int): telecom.Customer
--> telecom.Customer(String, int): telecom.Customer
<-- telecom.Customer(String, int): telecom.Customer
--> telecom.Customer(String, int): telecom.Customer
<-- telecom.Customer(String, int): telecom.Customer
jim calls mik...
--> Call telecom.Customer.call(Customer): telecom....
--> telecom.Call(Customer, Customer): telecom.Call
--> boolean telecom.Customer.localTo(Customer): ...
<-- boolean telecom.Customer.localTo(Customer): ...
--> telecom.Timer(): telecom.Timer
<-- telecom.Timer(): telecom.Timer
--> telecom.Connection(Customer, Customer): ...
<-- telecom.Connection(Customer, Customer): ...
--> telecom.Local(Customer, Customer): ...
[new local connection from Jim(650) to Mik(650)]
<-- telecom.Local(Customer, Customer): ...
<-- telecom.Call(Customer, Customer): telecom.Call
--> void telecom.Customer.addCall(Call): ...
<-- void telecom.Customer.addCall(Call): ...
<-- Call telecom.Customer.call(Customer): ...
mik accepts...
...
```



## Reduce information overload

Use **trace deltas** to reduce the amount of information.

- Run regression tests on both program versions (with/without aspect).
- Run “diff” on the traces.

### Results:

- Any trace line not reflecting aspect impact is filtered out.
- Deltas contain only relevant information.



## Results

Trace Deltas show ...

**Primary effects:** explicit advice executions

- calls to advice (via `adviceexec`)
- additional calls (before, after, inits)
- replaced calls (around advice)

**Secondary effects:** differences in traces due to changed system state

- no explicit advice execution, but diverging traces

*If the Aspect is orthogonal, no secondary effects should appear.*



## Example - Telecom Base & Telecom Timing

```
connection completed
<-- void telecom.Connection.: ...
<-- void telecom.Connection.complete(): telecom.
> --> execution(ADVICE: void telecom.Timing.ajc$after
> --> Timer telecom.Timing.getTimer(Connection):
> <-- Timer telecom.Timing.getTimer(Connection):
> --> void telecom.Timer.start(): telecom.Timer
> <-- void telecom.Timer.start(): telecom.Timer
> --> telecom.TimerLog(): telecom.TimerLog
> <-- telecom.TimerLog(): telecom.TimerLog
> --> execution(ADVICE: void telecom.TimerLog.ajc$
> Timer started: 1049378141427
> <-- execution(ADVICE: voi telecom.TimerLog.ajc$
> <-- execution(ADVICE: void telecom.Timing.ajc$
<-- void telecom.Call.: ...
<-- void telecom.Call.pickup(): telecom.Call
```

### Differences:

- 49 Lines (1/3 of original trace)
- only primary effects, no secondary effects



## No restrictions

### Multi-Threading:

1. Map each trace line to its thread
2. Separate different thread traces
3. Compare matching threads of different versions

### Several Aspects:

Which one is responsible for a secondary effect?

→ Apply delta debugging algorithm.



## Future Work

**Scalability:** “on-the-fly” comparison (save time & space).

**Delta-Generation:** more sophisticated trace comparison/analysis

- Filter repetitions
- Quality feedback (coverage analysis)
- Recognize wrapping (around-advice) or other common patterns

→ Further reduction of output complexity

### **Vision:**

Automate configurable trace generation process,  
craft simple-to-use testing tool (usable similar to JUnit).



## Summary

Our approach discovers aspect influence using trace deltas.

It shows:

- explicitly changed program flow (primary effects, advice calls)
- implicit behavioral changes (secondary effects, state change)
- unexpected side effects  
(*e.g. inter type declarations may change lookup*)

This analysis is a **flexible, low-cost, low-tech** approach, easy to use, more fine grained and powerful than traditional testing.

Our approach can be generally used for impact analysis.

## Questions?