Interface-based aspect-oriented programming in Compose*: its language independency, semantic point-cuts and aspect-interface detection possibilities

Mehmet Aksit
Chair Software Engineering (TRESE)
Department of Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede, The Netherlands
aksit@ewi.utwente.nl
trese.cs.utwente.nl/

Table of contents

- Key concepts of languages
- Composition mechanisms
- Invasive AOP
- Language independence
- Verification of semantic interference
- Conclusions and future work
Where are we now in programming?

Von Neumann machine

Procedures & control statements

Abstract data types

Transitive hierarchical composition (Inheritance)

Non-hierarchical composition

Universal abstraction mechanisms:
Classification, Composition-decomposition, Generalization-specialization

This looks great, BUT!

What are the key concepts of programming languages?

(for programming in the large!)
What are programming languages good for?

I guess, Problem solving by delivering the right program

If all is about problem solving..

Requirements

Problems

verification

solution_1 + solution_n

Problem analysis

solution_1

Solution domain_1

synthesis

Solution domain_2

Solution domain analysis
It must be about separation of concerns, composition of concerns, and semantics!!

But, then, .. what is the right composition mechanism?
Once upon a time..

Delegation, inheritance & reflection debate

Delegation is better
Delegation is inheritance
I can do anything with reflection
Non-invasive generalization approaches (1)

The Sina language

- point
- History point
- Bounded point
- Bh_point
- Solid_line
- Dashed-line
- Protection

Reuse mechanisms
“Dispatch Control”

The general idea behind the data abstraction model of Sina/st is that, starting from a simple object-based model, one can simulate various forms of abstractions without committing to a fixed number of alternative abstraction techniques such as delegation, relations or inheritance.

Non-invasive generalization approaches (2)

The law-governed system

- point
- History point

Law-governed System
In Prolog

“Dispatch Control”
But, what about the semantics?

Then some time has passed..
Synchronization inheritance anomalies

This is an anomaly, I have no reuse! Inheritance is useless

MIT Press: Research directions In concurrent OOP

Bergmans PhD 94
Research on synchronization inheritance anomalies attracted some attention

Real-time inheritance anomalies
Obstacles in object-oriented programming included seven “crosscutting-like” problems:

- Coordinated behavior: tangled & scattered
- Inheritance & delegation: dispatching aspect
- Synchronization inheritance: synchron. dispatch
- Multiple views: conditional dispatch & logical order
- Arbitrary inheritance: Any dispatch
- Associative inheritance: predicate dispatch
- Atomic delegation: Atomic dispatch & logical order

Composition-filters without superimposition:

- IEEE PDS 97 Distr sync
- IEEE Software 91 Transactions
- MCSEAI 98 Sync.
- ECOOP92 Multiple views
- OBDS93 Coordination
- ECOOP94 Realtime
- JPDC 96 Realtime & Sync
Interface-based aspect-oriented programming in Compose*:

Aspect-identification and aspect-based re-engineering example (from OBDS'93)

Then some more time has passed..
Interface-based aspect-oriented programming in Compose*:

Invasive approaches

Examples of AOP languages: AspectJ, HyperJ and framework-based approaches
Interface-based aspect-oriented programming in Compose*: 

AspectJ

OO languages

Reflection (Smith 81)

MOP (1985)

CLOS-MOP

Adaptive programming (1996)

Crosscutting aspects (1996)

AspectJ (1997)

Domain specific aspects

AspectJ (2001)

General purpose aspects

AspectJ like models

Clearing House

Account1

Account2

advice

Pointcut designator

transfer

withdraw

deposit

Call join point

execution join point

© Mehmet Aksit Interface-based programming in Compose*: Foundations of Aspect-Oriented Languages 2009
Interface-based aspect-oriented programming in Compose*

HyperJ

- Multiple views concerns
- OO languages
- Combination of inheritance hierarchies (ECOOP 1992)
- Subject-oriented programming (1993)
- Adapting interfaces according to the context
- Composition of Classes via relational operators

HyperJ (continued)

- Class1
  - a1
  - a2
  - m1
  - m2

- Class2
  - ax
  - ay
  - mx
  - my

- Class3
  - a1
  - a2
  - ax
  - ay
  - M1
  - m2
  - mx
  - my

[Diagram of class relationships and methods]

[References: OOPSLA'93, Kluwer01]
Framework-based aspect-oriented approaches

- Supported within a platform with a number of libraries & tools
- Provides aspect weaving mechanisms as a tool
- Becoming more and more popular
- Mostly implement Composition Filters like mechanisms (using proxies/interceptors, etc.)

AOP languages have some concerns

Being able to compose crosscutting concerns is an additional benefit but, crosscutting concern mechanisms are “tangled with language semantics” (language dependent);

Extending existing languages with AOP constructs makes the languages too complex;

Verifying semantic correctness of compositions is still difficult;

Joinpoint level of composition is too low-level.
How to address these problems?

Step 1:
Separation of composition operators from concerns
Composition operators as dedicated interface extensions

Step 2: Composing composition operators with concerns
Composition of composition operators using “quantifiers”

The Composition-filters model

- Uniform model (everything is a “concern”)
- Supports strong encapsulation (“interface programming”)

Filters are modular and composable extensions & manipulate messages
First two claims of composition filters:

*language independence*

*domain specific aspects as filters*

Simulated demonstration of the proof of these two claims
What we will show in the demo

- Composing a *word counting* ‘feature’ with the *book shelf* application as a modular concern
- Composing a *cache* optimization concern with the *word counting* concern.
- Language/platform independence

Example: Bookshelf

```
Book
#title : String
#author : String
#chapters[] : Chapter
+getText() : String
+getTitle() : String
+getAuthor() : String
+addChapter(in cha : Chapter)
+getChapter(in index : int) : Chapter
```

```
Chapter
#title : String
#paragraphs[] : Paragraph
+getText() : String
+getTitle() : String
+addParagraph(in par : Paragraph)
+getParagraph(in index : int) : Paragraph
```

```
Paragraph
#text : String
+getText() : String
```
Counting words and sentences

Design using Composition Filters
Interface-based aspect-oriented programming in Compose*:

Result of using Composition Filters

myBook.countWords();
myChapter.countWords();
myPar.countWords();

Adding Caching

myBook.countWords();
myChapter.countWords();
myPar.countWords();
Conclusion of the first demonstration

- Composing a word counting ‘feature’ with the bookshelf application
- Composing the caching optimization concern as a high level concern with the word counting concern
- Composition Filters tools work on C/.NET / Java
  - BookShelf + WordCounting + Caching in Java
  - Fibonacci + Caching in C
  - Same concern code (Caching) reused!
- Efficiency: filters can be in-lined -> low overhead

The third claim of the Composition-Filters

Should be easier to verify the semantic correctness of filter compositions since filters are modular extensions;
CF’s: Should be easier to verify

- Controlling the order of aspect superimpositions using partial, advice-based priority specifications
- Resource-model based aspect-interference analysis approach
- Graph-based aspect-interference analysis approach

Correct composition

Controlling the order of aspect superimpositions using partial, advice-based priority specifications

Composing Aspects at Shared Join Points

István Nagy, Lodewijk Bergmans and Mehmet Aksit
TRIIDE group, Dept. of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
+31-53-4895482, 44271, 2638
{magist, bergmans, aksit}@cs.utwente.nl

Abstract. Aspect-oriented languages provide means to superimpose aspectual behavior on a given set of join points. It is possible that not just a single, but several units of aspectual behavior need to be superimposed on the same join point. Aspects that specify the superimpositions of these units are said to “share” the same join point. Such shared join points may give rise to issues such as determining the exact execution order and the dependencies among the aspects. In this paper, we present a detailed analysis of the problem, and identify a set of requirements upon mechanisms for composing aspects at shared join points. To address the identified issues, we propose a general and declarative model for defining constraints upon the possible compositions of aspects at a shared join point. Finally, by using an extended notion of join points, we show how concrete aspect-oriented programming languages, particularly AspectJ and Compose*, can adopt the proposed model.
Resource-model based aspect-interference analysis approach

Serialization-based aspect-interference approach (continued)

From the set of filters (from multiple filtermodules), per resource a sequence of operations is compiled, e.g.:

NB: not for all combinations of accept & reject a corresponding message exists: this is filtered out

These sequences are matched with conflict patterns (defined per resource), e.g. "R*WR"

So: RR and RRRW don’t match, but RWR does match → conflict
Serialization-based aspect-interference approach (continued)

The process:

Graph-based aspect-interference analysis
Graph-based aspect-interface analysis approach

Detect aspect-interference on a shared joinpoints without:

– (formally) specifying the aspect
– (formally) specifying the base program
– specifying the base language semantics

“Only analyse the aspects, not the base program.”

Graph-based aspect-interface approach (continued)

generate abstract syntax graph
substitute filter-types with filter-actions
construct control flow graph
create runtime state (method call)

simulation: generate state space
– Fixed production system for operational semantics

verification: analyse state space
Graph-based aspect-interface approach (continued)

Two kinds of non-determinism:
- Assignment of unknown runtime values (conditions)
- Selection of filter module order

Conclusions about the semantic interference detection

- **Prioritization** of compositions is sometimes necessary. Prioritization must be based on partial specifications. This is integrated in filters.

- **Well-defined** interfaces of concerns (filters) and/or concerns with well-defined semantics (domain specific concerns) make it easier to analyze & verify semantic interference among concerns (for example by using graph-based verification).

- In case of large state spaces, concerns can be abstracted as **semantic signatures** (related to a resource model) and be analyzed based on the references to the resource-model.
Our future work

Define larger set of filters (filter library) identified from practical needs

Work further on semantic composition also for not shared joinpoints

Composition patterns

Higher-level compositions

Larger set of explicit composition operator

(a) Formatter
   Software Module 1

(b) OR composer
   Software Module 1
   Software Module 2

(c) Selector
   Strategy
   Software Module 1
   Software Module 2

(d) coordinator
   Coordination protocol
   Software Module 1
   Software Module 2

© Mehmet Aksit
Interface-based aspect-oriented programming in Compose*: Foundations of Aspect-Oriented Languages 2009
Conclusions

- **Non-invasive AOP** has some advantages; **language independence, semantic verification**, etc.

- **Non-invasive AOP** must be supported with a high-level compositional language.

- **Invasive AOP** is probably good for applications like runtime verification kind of applications, but the ideal level of detail of the joinpoints is difficult to determine. Open joinpoint models can be the answer!