Aspects, Processes, and Components

PL 2008 - Punta Arenas
Aspects, Processes, and Components

Jacques Noyé
OBASCO - Ecole des Mines de Nantes/INRIA, LINA
Jacques.Noye@emn.fr

12 November 2008
The basic idea

- There is usually no specific support for concurrency in AOP languages.
- Both (regular) **stateful aspects** and processes can be represented as automata.
- What about modelling both the base program and aspects as automata and combine stateful aspects and concurrency (between base and aspects as well as between aspects).
- May such a model be used to synthesize aspects and facilitate reuse?
Aspects, Processes, and Components

Introduction

Stateful Aspects [DFS02, DFS04]

- Standard aspects are **stateless**, they deal with a unique atomic action (a *join point*):

  \[
  \text{if } \text{pointcut}(\text{join point}) \text{ eval}(\text{advice});
  \]

- **Stateful** aspects may affect the execution of the base program, depending on the *state* of the program, ie depending on the previous execution.

  \[
  \text{if } \text{pointcut1}(\text{join point}) \text{ eval}(\text{advice1});
  \]

  \[
  \text{if } \text{pointcut2}(\text{join point}) \text{ eval}(\text{advice2});
  \]
Aspects, Processes, and Components

A Naive Example

Base Model

Server =
    ( login -> Session
     | update -> Server
    ),

Session =
    ( checkout -> Server
     | update -> Session
     | browse -> Session
    ).
Event-Based AOP (EAOP) [DFS02].

\[ \mu a. (\text{login}; \mu a'. ((\underline{\text{update}} \triangleright \text{skip}; \log; a') \square (\text{checkout}; a))) \]

Consistency =
\[ (\text{login} \rightarrow \text{Session} )\]

Session =
\[ (\text{update} \rightarrow \text{skip} \rightarrow \log \rightarrow \text{Session} \]
\[ | \text{checkout} \rightarrow \text{Consistency} \]

Aspects, Processes, and Components

A Naive Example

Woven Model

$ S = (\text{Server} \parallel \text{Consistency}) $.

Problems:

- We want to execute update out of a session
- We don’t want to update within a session
Instrumentation of Base

- We are interested in the event “an update is about to take place” in order to execute a before advice.
- When “an update is about to take place” an aspect interested in updates should be able to decide whether the action should take place or not.

Server =

( login -> Session
 | bUpdate -> ( skip -> Server
 | proceed -> update -> Server
 )
 ),

Session =

( checkout -> Server
 | bUpdate -> ( skip -> Session
 | proceed -> update -> Session
 )
 )
All the actions **shared** between the base and the aspect: \( \{ \text{login}, \text{bUpdate}, \text{checkout} \} \) must be dealt with in each aspect state. Some actions are **skippable** (the aspect may decide not to execute them), others are **not skippable**.

Consistency =

\[
( \text{login} \rightarrow \text{Session}
| \text{bUpdate} \rightarrow \text{proceed} \rightarrow \text{Consistency}
| \text{checkout} \rightarrow \text{Consistency}
),
\]

Session =

\[
( \text{bUpdate} \rightarrow \text{skip} \rightarrow \text{log} \rightarrow \text{Session}
| \text{checkout} \rightarrow \text{Consistency}
| \text{login} \rightarrow \text{Session}
).
\]
Aspects, Processes, and Components

Solution

Woven Model
Controlling Concurrency Between Base and Aspect

We introduce pairs of instrumentation events (beginEvent, endEvent):

\[
\text{bUpdate} \rightarrow \\
( \text{bSkip} \rightarrow \text{eSkip} \rightarrow \text{eUpdate} \rightarrow \text{Server} \\
| \text{bProceed} \rightarrow \text{update} \rightarrow \text{eProceed} \rightarrow \text{eUpdate} \rightarrow \text{Server} )
\]
Controlling Concurrency Between Base and Aspect (2)

\[ \text{Concurrent Consistency} = (\text{Consistency}) \setminus \{e\text{Update}\}. \]
Summary

- **Input:**
  - a base program modelled as an FSP $B$
  - a stateful aspect $A$ expressed in an extended version of FSP:

  \[
  \text{Consistency} = \langle \text{login} \rightarrow \text{Session} \rangle, \\
  \text{Session} = \langle \text{update} > \text{skip}, \text{log} \rightarrow \text{Session} \rangle, \\
  \text{| checkout} \rightarrow \text{Consistency} \rangle.
  \]

- **Output (the weaving of $A$ into $B$):**
  \[
  \text{BaseTransf}(B) \parallel \text{hiding(AspectTransf}(A))
  \]

- The transformations are independent from the composition.

- Hiding controls concurrency between the base and the aspect.
Composing Aspects - Basic Idea

- Abstract point of view: the aspects are composed via operators
- Example: Fun(Consistency, Safety) with
  \[
  \text{Safety} \triangleq \mu a''.(\text{update} \triangleright \text{rehash proceed backup}; a'')
  \]
- An operator is modelled as the composition of a specific FSP with a proper renaming.
The Fun Operator

Fun(Aspect1, Aspect2) is the “functional” sequential composition (used in AspectJ) of Aspect1 and Aspect2.
The Fun Operator - Simplified Structural View

Fun = (skip1 -> skip -> Fun l skip2 -> skip -> Fun).
The Fun Operator - Simplified Structural View

Fun = (skip1 -> skip -> Fun | skip2 -> skip -> Fun).

Base -> proceed, skip
Aspect1 -> skip, skip1
Aspect2 -> skip, skip2

The Fun Operator - Simplified Structural View

\[ \text{Fun} = (\text{skip1} \rightarrow \text{skip} \rightarrow \text{Fun} \mid \text{skip2} \rightarrow \text{skip} \rightarrow \text{Fun}) \]
The ParAnd Operator - Simplified Structural View

ParAnd = (skip1 -> (skip2 -> skip -> ParAnd
                             | proceed2 -> skip -> ParAnd
                             | proceed1 -> (skip2 -> skip -> ParAnd
                             | proceed2 -> proceed -> ParAnd)).

Base
  proceed
  skip
  bEvent

Aspect1
  proceed
  skip
  skip1
  bEvent proceed

Aspect2
  skip
  skip2
  bEvent proceed

ParAnd Operator Diagram with nodes and edges representing the aspect compositions and control flow.
Prototype: Baton [NN07a]

- The base program is instrumented with AspectJ-like pointcuts describing the actions of interest (using Reflex [TTPN08]).
- The previous transformations are used to generate the aspects (as active objects) from a concrete syntax close to FSP (using Metaborg/SDF).
- Calls to a global monitor are used to synchronize the shared actions:
  - two synchronization barriers per transition!
  - naive but guarantees correction wrt the model
aspect Consistency {
    public void log(Client client, Admin admin) {
        System.out.println(admin + " skipped:
                        + client + " is connected."));
    }
    behaviour {
        Server = ( login(Client client) -> InSession(client) ),
        InSession(client) =
            ( update(Admin admin) > skip, log(client,admin)
                -> InSession(client)
            | checkout(client) -> Server ).
    }
}
connector ClientConnector{
    connect login(Client c):
        execution(* Client.login(..)) && this(c);
    connect checkout(Client c):
        execution(* Client.checkout(..)) && this(c);
}
main Ecommerce{
    Aspect aspect = new ParAnd(new Consistency(), new Safety());
    Client client = new Client();
    Admin admin = new Admin();
    Connector clientCon = new ClientConnector();
    Connector adminCon = new AdminConnector();
    Baton.connect(aspect, clientCon, client);
    Baton.connect(aspect, adminCon, admin);
    Baton.start();
}
The base program is structured as components with interfaces specifying the provided and required services, as well as the published events (these are kinds of *open modules* [Ald05]).

Published events look very much like required services, but their connection is optional.

The aspect protocols are also associated to interfaces specifying the expected events (and their property skippable or not) as well as the required services.

An application composed of components and aspects is transformed/compiled into a component-based application.
Concurrent Event-Based AOP (CEAOP)

- A formal model of concurrent stateful aspects [DLBNS06].
  - Transformation semantics (translation into pure FSP).
  - The base as well as the aspects can be concurrent.
  - Composition operators are used to coordinate the aspects and the base program.

- Prototype implementations (extensions of Java).

- The aspects can be reused in various compositions.

- Clarifies the relationship between stateful aspects and process calculi.
Current work

These ideas are currently integrated into a new version of CaesarJ [AGMO06]:

- Extended advice language
- Processes are class members and can be redefined or extended in superclasses, and composed using mixin composition
Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann.
An overview of CaesarJ.
In *Transactions on Aspect-Oriented Software Development I*,
volume 3880 of *Lecture Notes in Computer Science*, pages

Jonathan Aldrich.
Open modules: Modular reasoning about advice.
In Andrew P. Black, editor, *ECOOP 2005 - Object-Oriented
Programming, 19th European Conference*, volume 3586 of
*Lecture Notes in Computer Science*, pages 144–168, Glasgow,

Rémi Douence, Pascal Fradet, and Mario Südholt.
A framework for the detection and resolution of aspect
interactions.

Rémi Douence, Pascal Fradet, and Mario Südholt.
Composition, reuse and interaction analysis of stateful aspects.


Rémi Douence, Didier Le Botlan, Jacques Noyé, and Mario Südholt.
Concurrent aspects.

In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*
Angel Núñez and Jacques Noyé.
A domain-specific language for coordinating concurrent aspects in java.

Angel Núñez and Jacques Noyé.
A seamless extension of components with aspects using protocols.
Éric Tanter, Rodolfo Toledo, Guillaume Pothier, and Jacques Noyé.

Flexible metaprogramming and AOP in Java.


Special issue on Experimental Software and Toolkits.