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?

Stateful Aspects [DFS02, DFS04]

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

  `if` *pointcut*`(`*join point*`)` `eval(`*advice*`);`

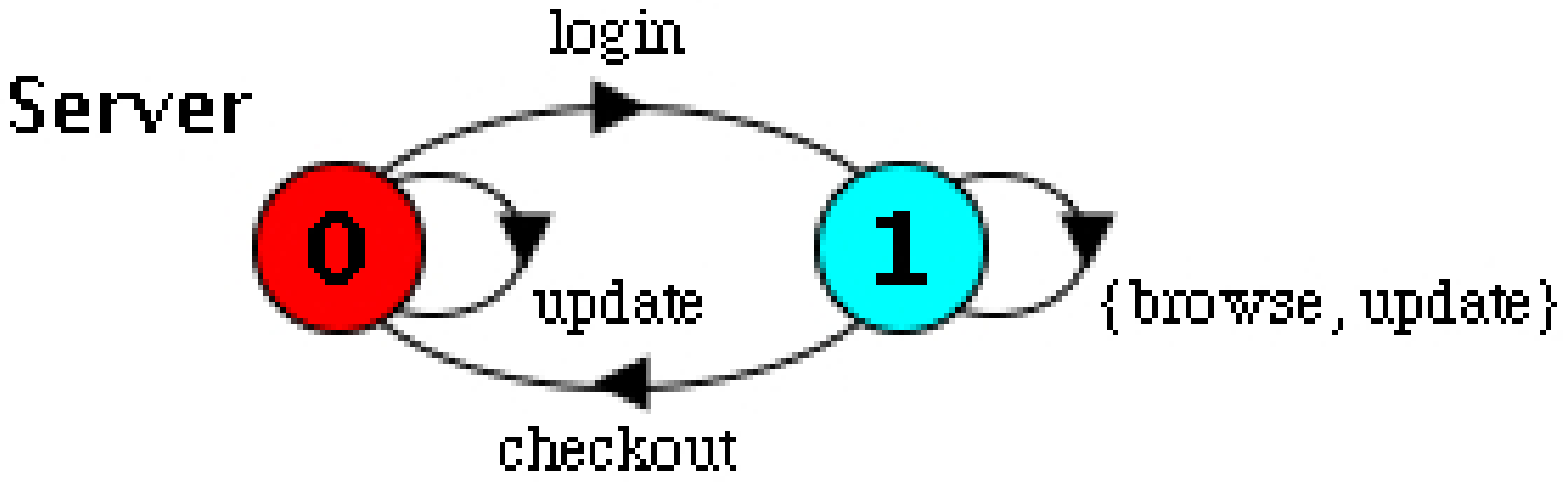
- Stateful aspects may affect the execution of the base program, depending on the *state* of the program, ie depending on the previous execution.

  `if` *pointcut*1`(`*join point*`)` `eval(`*advice*1`);`
  `if` *pointcut*2`(`*join point*`)` `eval(`*advice*2`);`

Base Model

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

```
Session =
  ( checkout -> Server
  | update -> Session
  | browse -> Session
  ).
```



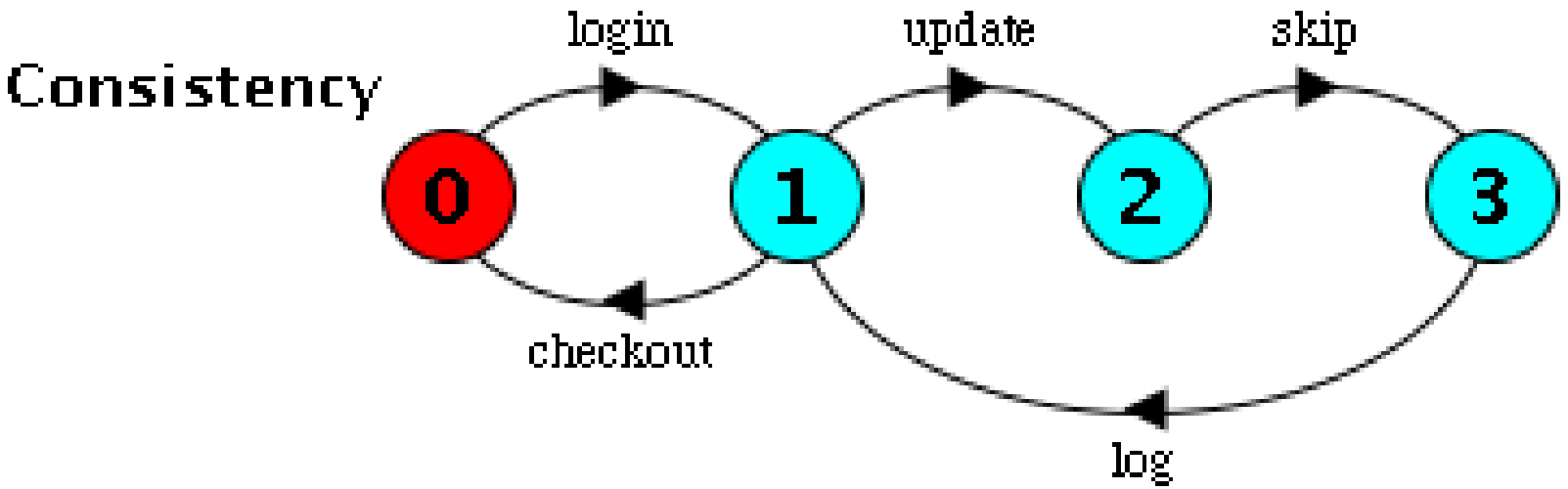
Aspect

Event-Based AOP (EAOP) [DFS02].

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

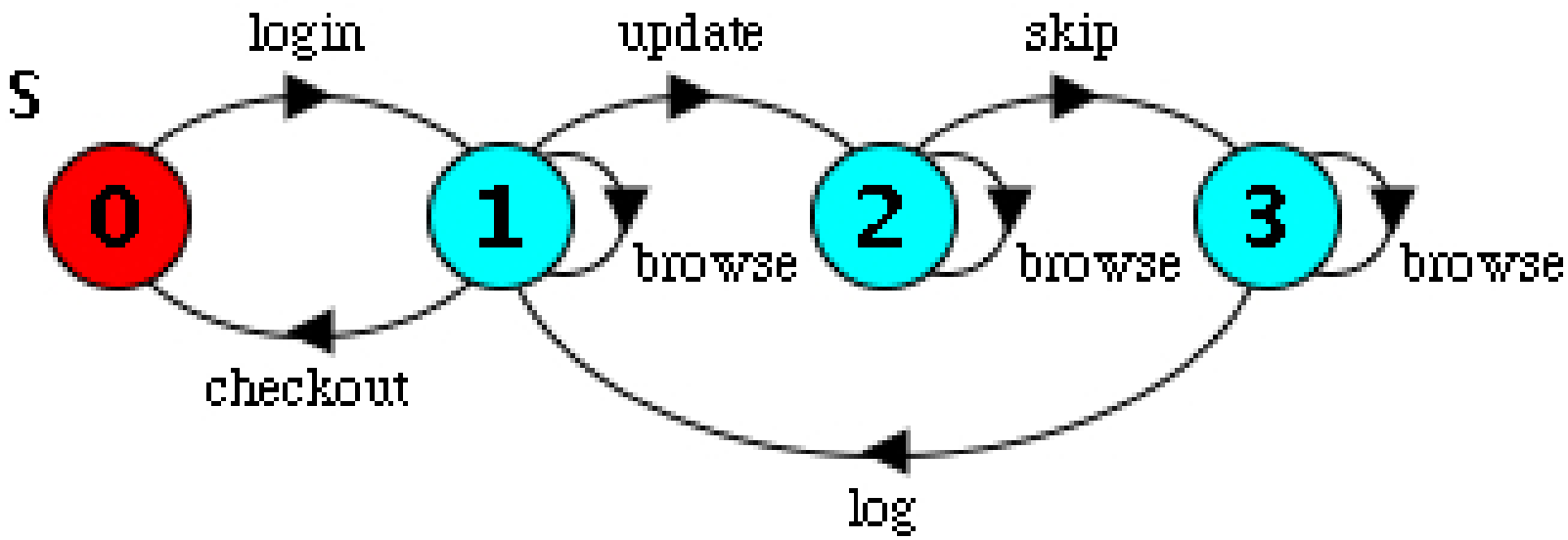
```
Consistency =
  ( login -> Session
  ),
```

```
Session =
  ( update -> skip -> log -> Session
  | checkout -> Consistency
  ).
```



Woven Model

```
||S = (Server || 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.

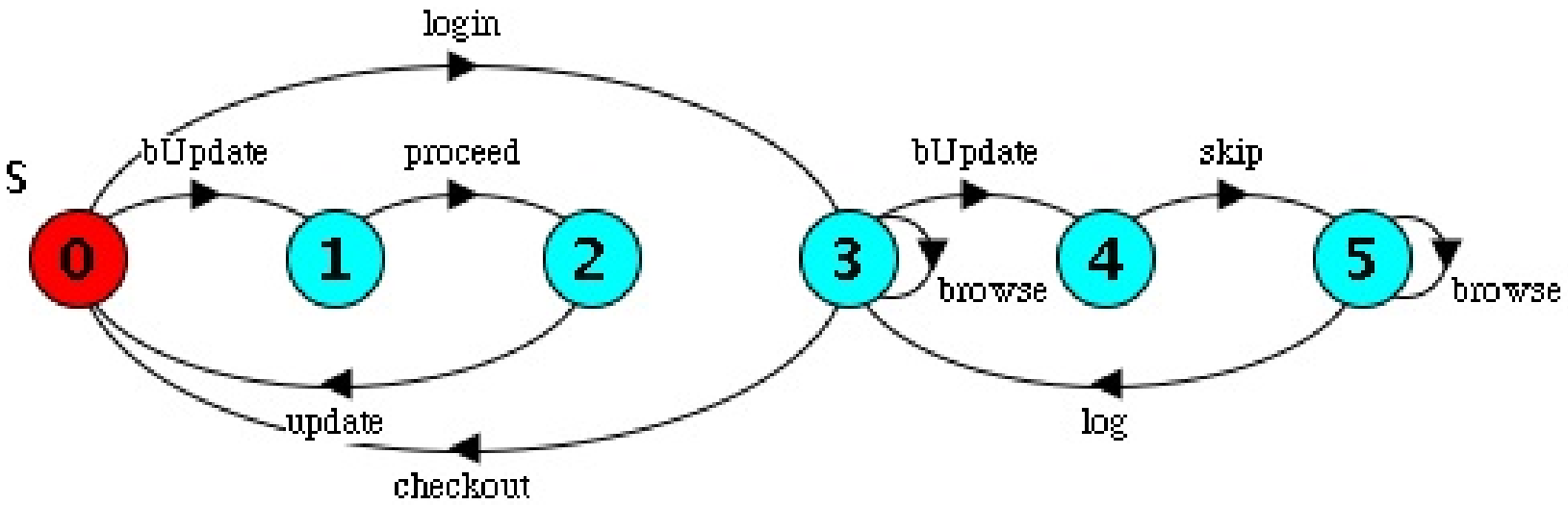
```
Server =
  ( login -> Session
  |  bUpdate -> ( skip -> Server
                | proceed -> update -> Server
                )
  ),
Session =
  ( checkout -> Server
  |  bUpdate -> ( skip -> Session
                | proceed -> update -> Session
                )
```

Adding Waiting Loops to Aspects

- All the actions shared between the base and the aspect: $\{\texttt{login}, \texttt{bUpdate}, \texttt{checkout}\}$ 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 =
  ( login -> Session
  | bUpdate -> proceed -> Consistency
  | checkout -> Consistency
  ),
Session =
  ( bUpdate -> skip -> log -> Session
  | checkout -> Consistency
  | login -> Session
  ).
```

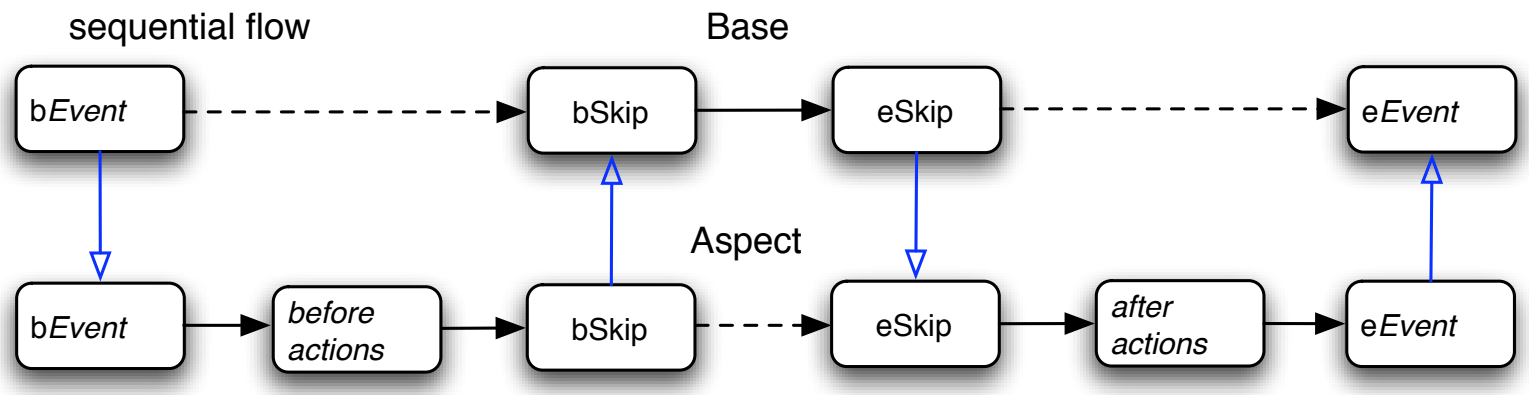
Woven Model


login
bUpdate
proceed
bUpdate
skip
S
0
1
2
3
4
5
browse
browse
update
log
checkout

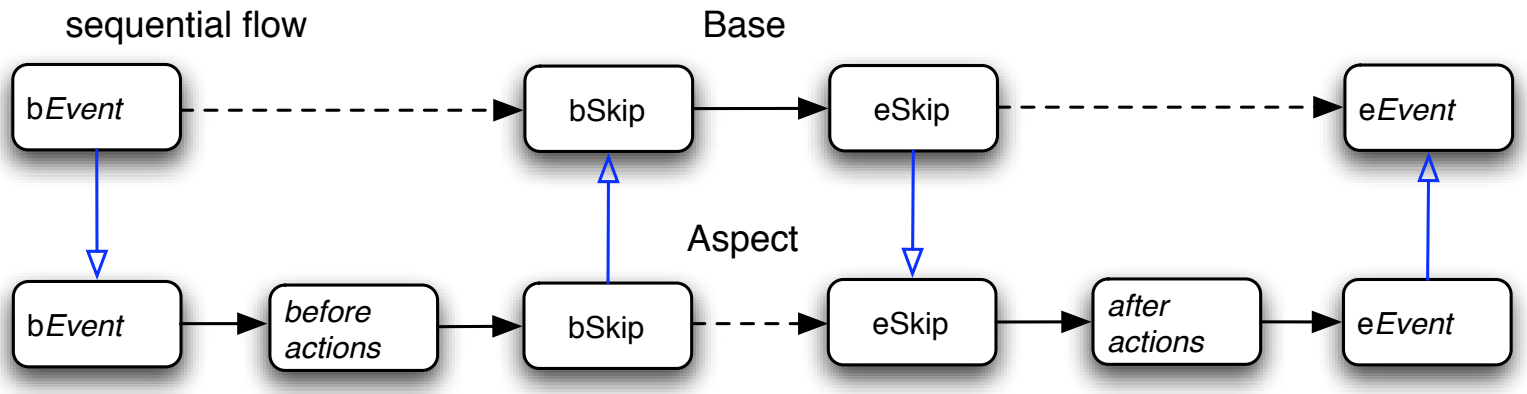
Controlling Concurrency Between Base and Aspect

We introduce pairs of instrumentation events (begin*Event*, end*Event*):

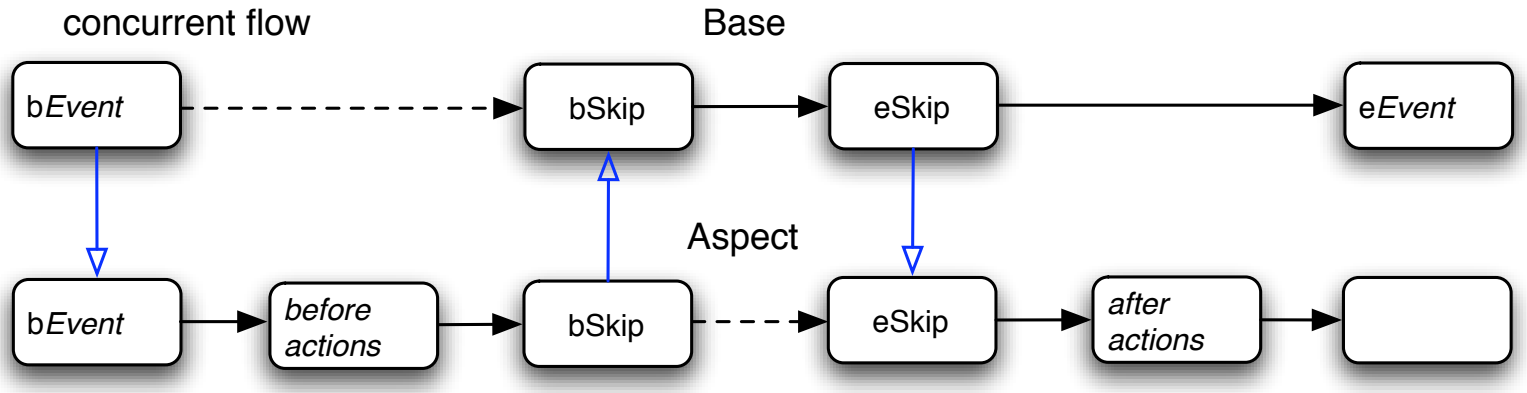
```
bUpdate ->
  ( bSkip -> eSkip -> eUpdate -> Server
  | bProceed -> update -> eProceed -> eUpdate -> Server
  )
```



Controlling Concurrency Between Base and Aspect (2)




```
||ConcurrentConsistency = (Consistency)\{eUpdate}.
```



Summary

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

```
Consistency =
  ( login -> Session
  ),
```

```
Session =
  ( update > skip , log -> Session
  | checkout -> Consistency
  ).
```

- Output (the *weaving* of `A` into `B`):
  `BaseTransf(B) || hiding(AspectTransf(A))`
- The transformations are independent from the composition.
- Hiding controls concurrency between the base and the aspect.

Composing Aspects - Basic Idea

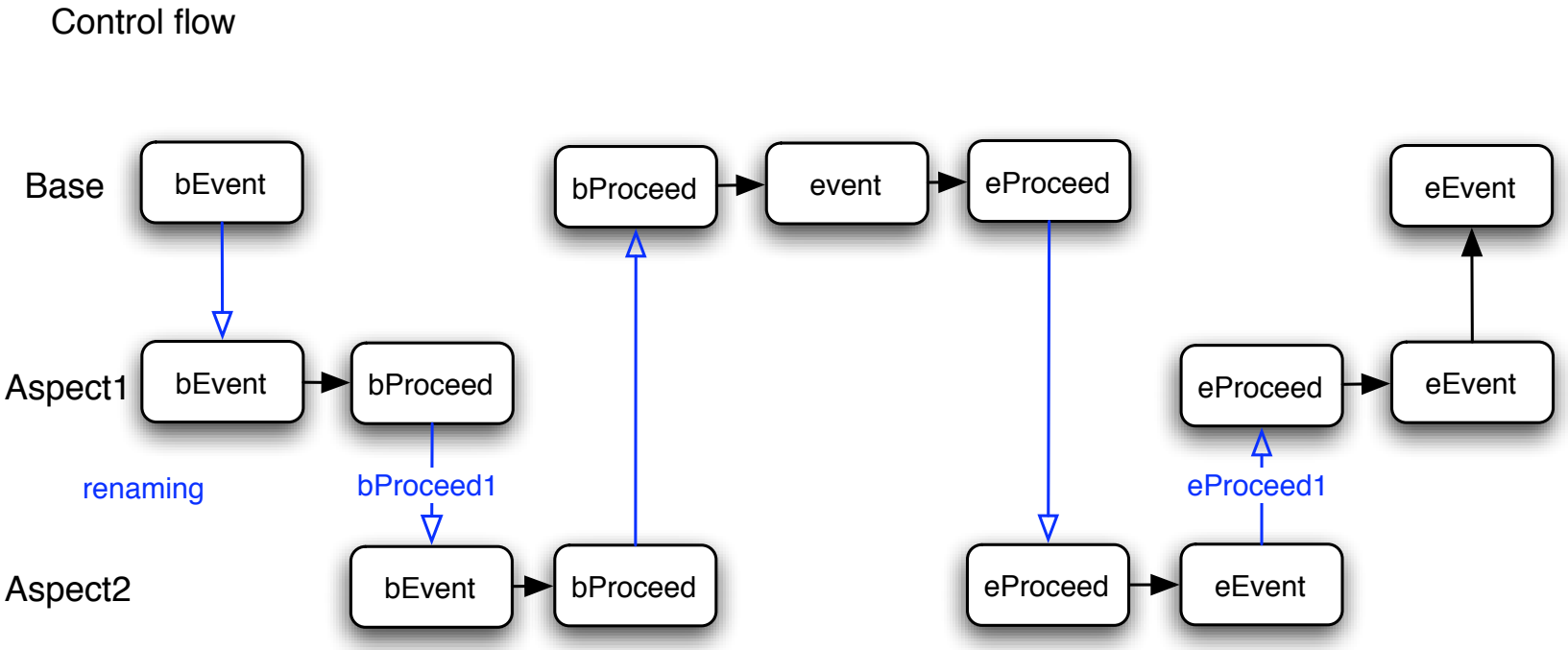
- Abstract point of view: the aspects are composed via operators
- Example: `Fun(Consistency, Safety)` with

$$\text{Safety} \triangleq \mu a''.(\text{update} \triangleright \text{rehash proceed backup}; a'')$$

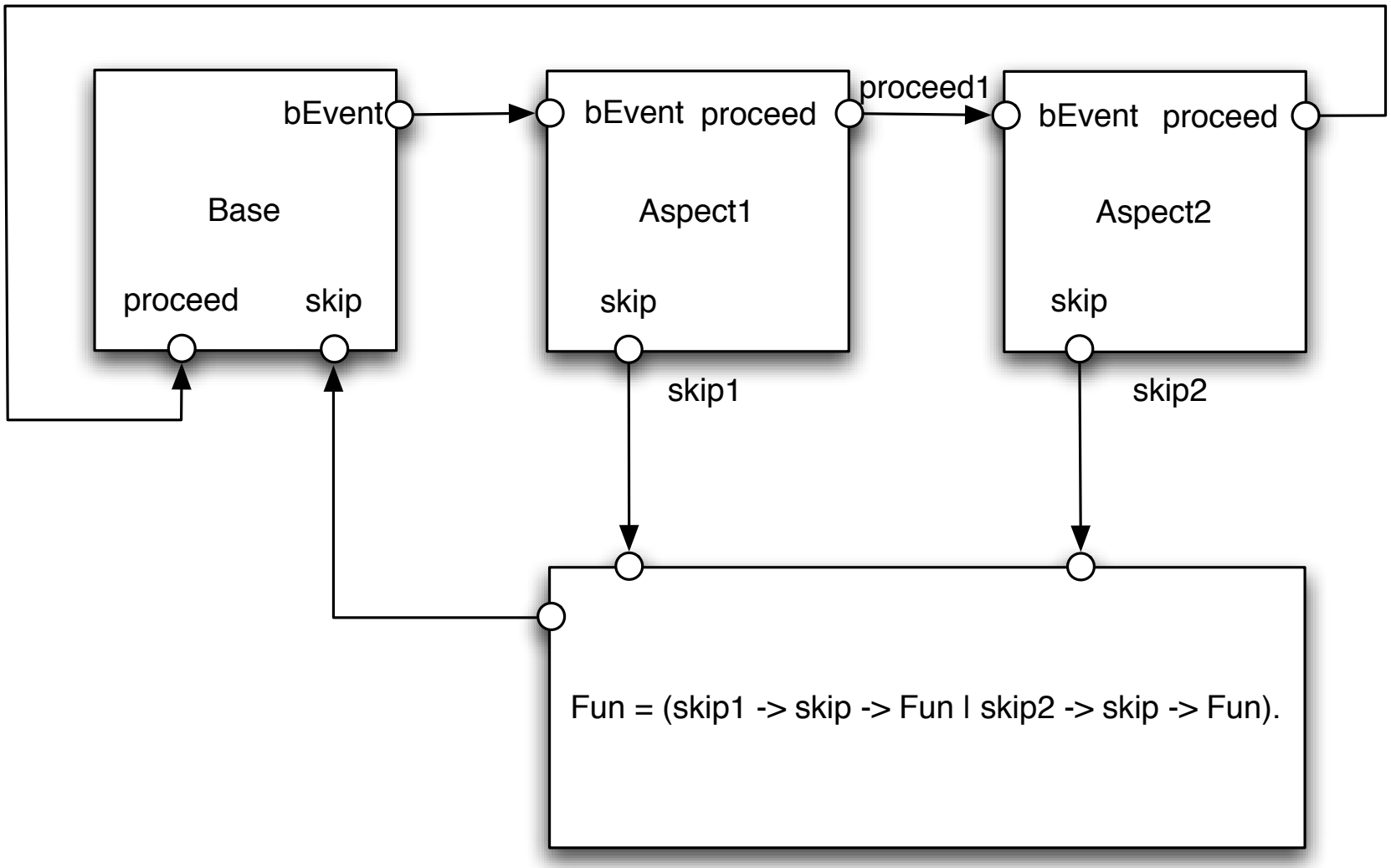
- An operator is modelled as the composition of a specific FSP with a proper renaming.

The Fun Operator

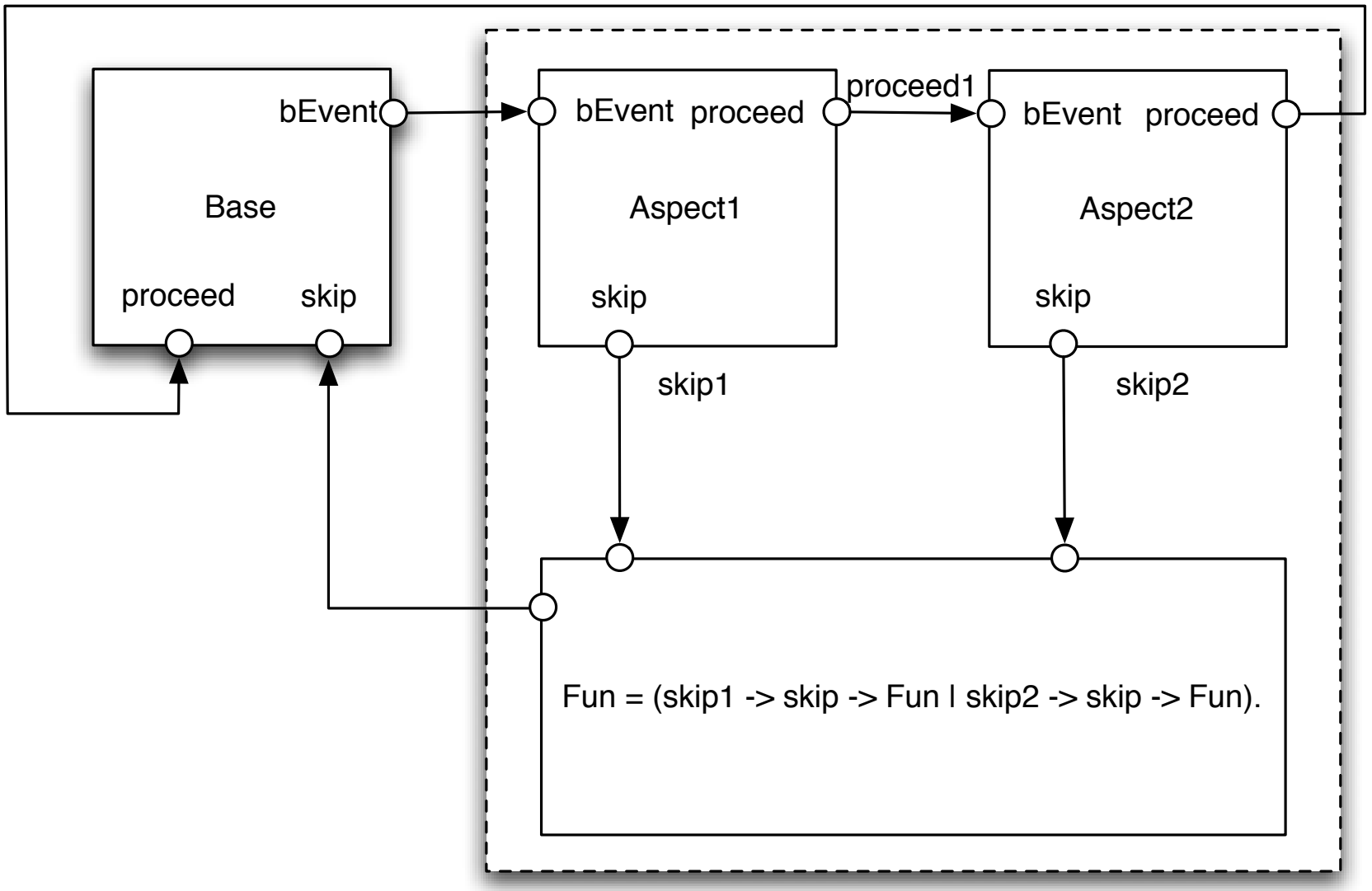
`Fun(Aspect1, Aspect2)` is the "functional" sequential composition (used in AspectJ) of `Aspect1` and `Aspect2`.



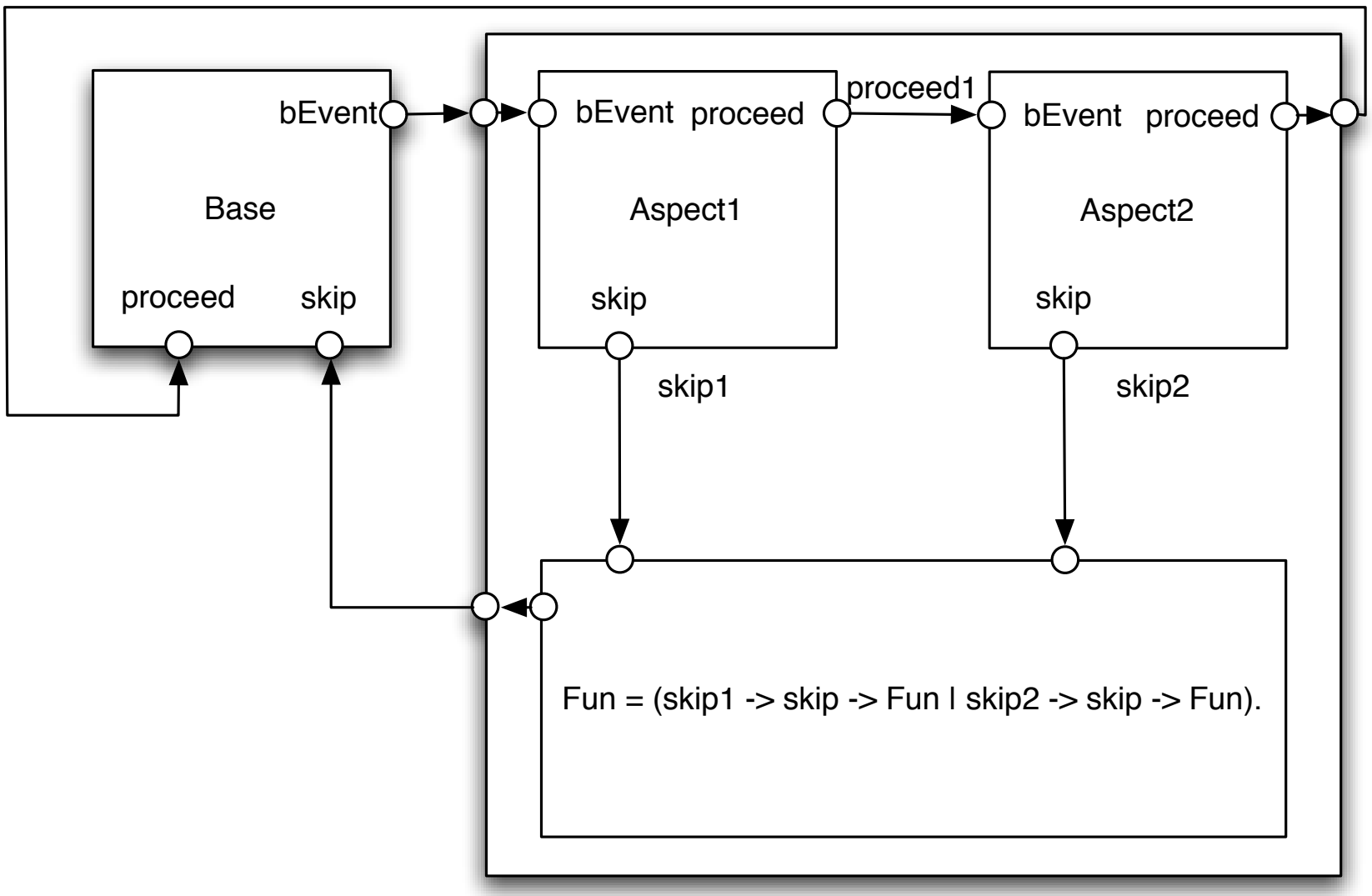
The Fun Operator - Simplified Structural View



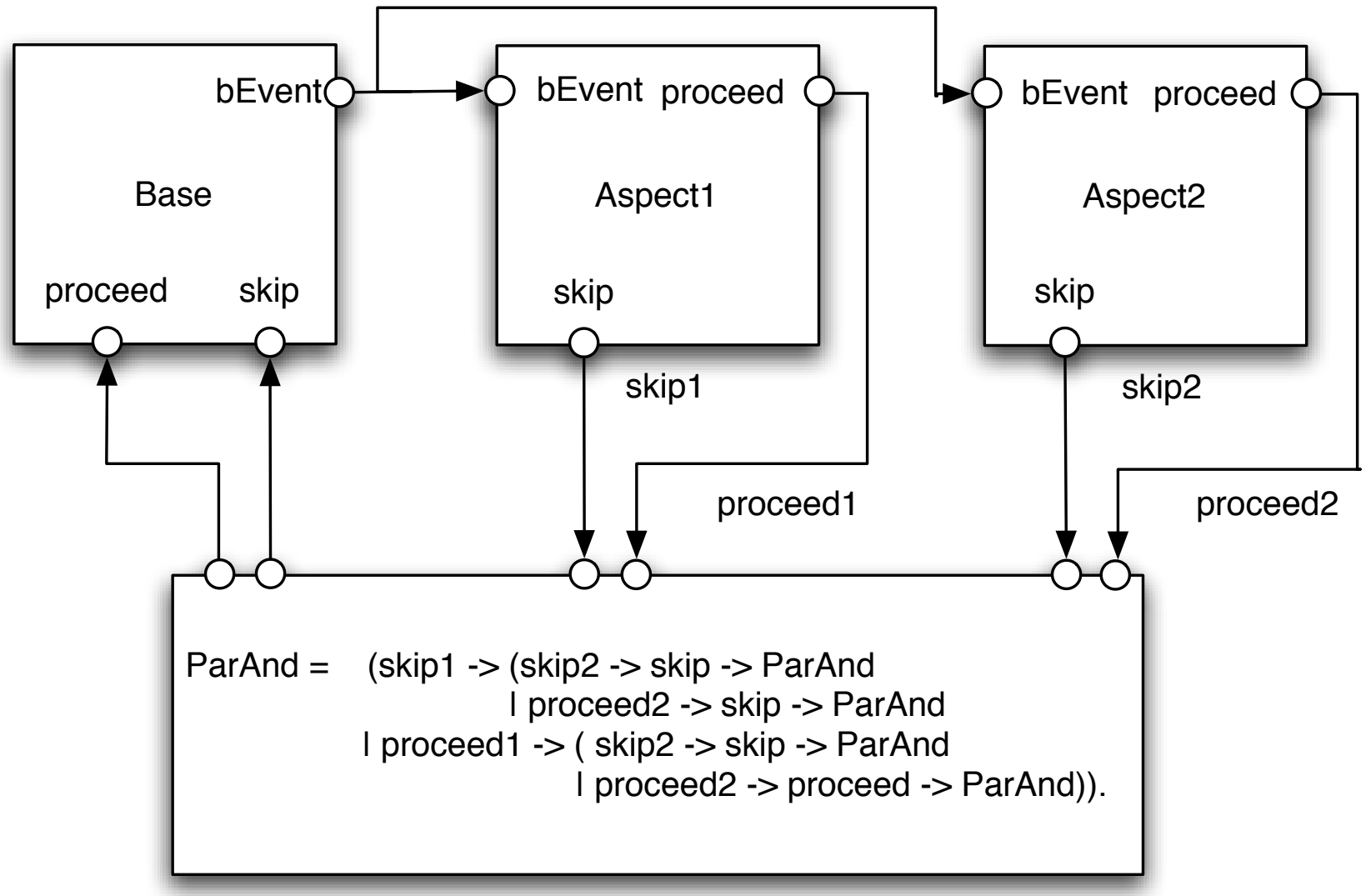
The Fun Operator - Simplified Structural View



The Fun Operator - Simplified Structural View



The ParAnd Operator - Simplified Structural View



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

```
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

```
connector ClientConnector{
  connect login(Client c) :
    execution(* Client.login(..)) && this(c);
  connect checkout(Client c) :
    execution(* Client.checkout(..)) && this(c);
}
```

Main Program

```
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();
}
```

Prototype Components/Aspects [NN07b]

- 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 135–173. Springer-Verlag, February 2006.


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, UK, July 2005. Springer-Verlag.


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

In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference, GPCE 2002 - Proceedings*, volume 2487 of *Lecture Notes in Computer Science*, pages 173–188, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

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

In Karl Lieberherr, editor, *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 141–150, Lancaster, UK, March 2004. ACM Press.

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*

(GPCE'06), pages 79–88, Portland, USA, October 2006. ACM Press.

Angel Núñez and Jacques Noyé.
A domain-specific language for coordinating concurrent aspects in java.
In Rémi Douence et Pascal Fradet, editor, *3ème Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA 2007)*, Toulouse, France, March 2007.

Angel Núñez and Jacques Noyé.
A seamless extension of components with aspects using protocols.
In Ralf Reussner, Clemens Szyperski, and Wolfgang Weck, editors, *WCOP 2007 - Components beyond Reuse - 12th International ECOOP Workshop on Component-Oriented Programming*, Berlin, Germany, July 2007.


Éric Tanter, Rodolfo Toledo, Guillaume Pothier, and Jacques Noyé.
Flexible metaprogramming and AOP in Java.
Science of Computer Programming, 72(1-2):22–30, 2008.
Special issue on Experimental Software and Toolkits.