Concurrent Event-Based AOP Protocols: the missing link between components and aspects?

Rémi Douence, Didier Le Botlan, **Jacques Noyé**
Angel Núñez, Mario Südholt
OBASCO

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
centre de recherche RENNES - BRETAGNE ATLANTIQUE
INRIA
ECOLE DES MINES DE NANTES
lina LABORATOIRE D'INFORMATIQUE DE NANTES ATLANTIQUE

May 13, 2008

Context

- EAOP [DFS02]: sequential semantics, prototype in Java using coroutines.
- There is usually no specific support for concurrency in "standard" AOP.
- Event-based aspects as well as processes can be represented as Labelled Transition Systems (LTSs).
- What about modelling both the base program and the aspects as LTSs and combining event-based aspects and concurrency?
- Can such a model be used to synthesize aspects and facilitate reuse?

Tools

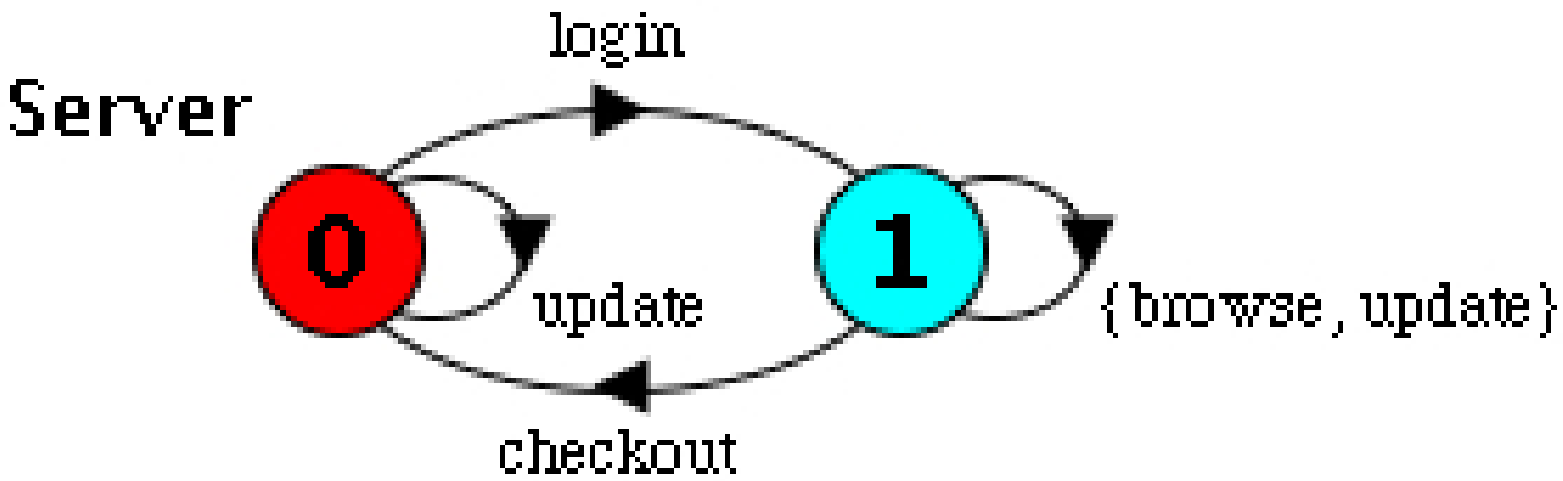
LTSA (*Labelled Transition System Analyzer*) [MK06].

- Models finite state machines with a dual representation:
  - Graphical: LTS (*Labelled Transition Systems*).
  - Textual: FSP (*Finite State Processes*).
  - Synchronisation through shared actions.
- Animation, checking safety and progress properties.

Base Model

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

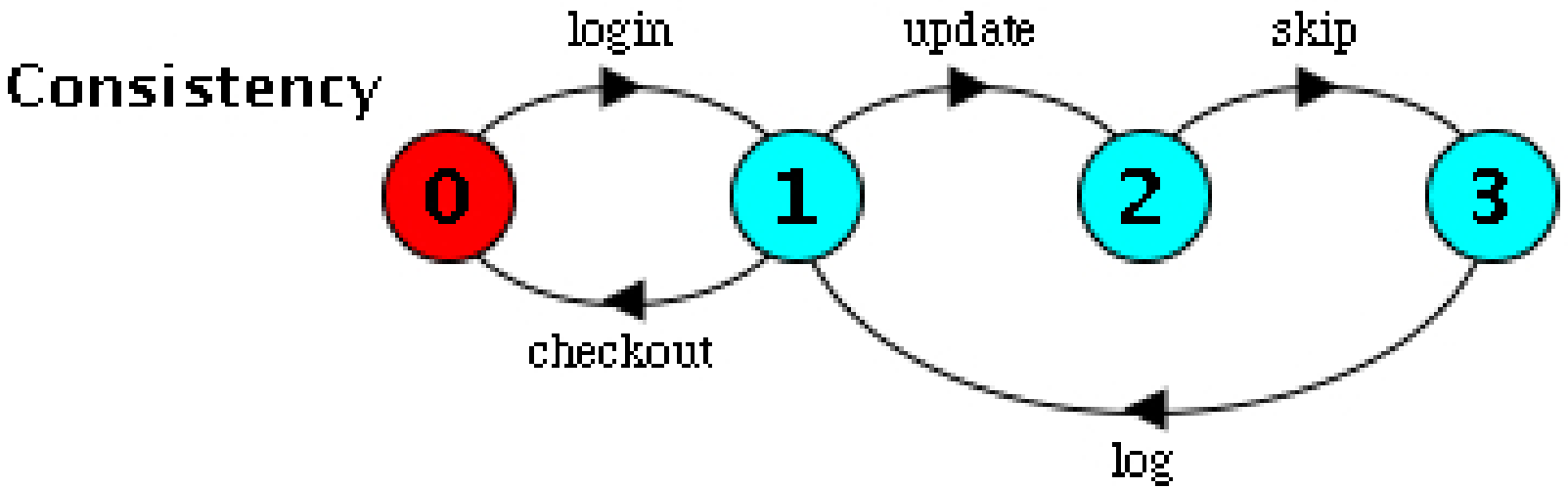
```
Session =
  ( checkout -> Server
  | update -> Session
  | browse -> Session
  ).
```



An Event-based Aspect in Pseudo-FSP

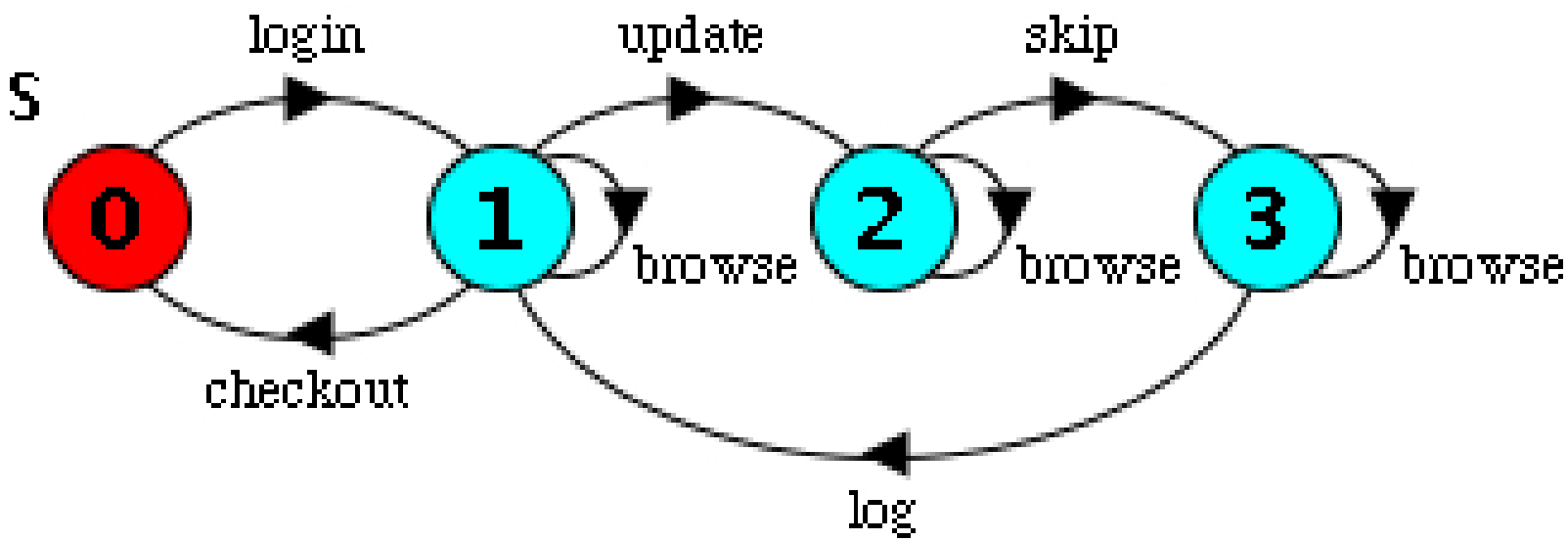
```
Consistency =
  ( login -> Session
  ),
```

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



An Attempt at Weaving using Process Composition

```
||S = (Server || Consistency).
```



Issues:

- The action `update` should still be executed outside of a session.
- It should not occur within a session.

Base Model Instrumentation

- The event of interest is not the update but the fact that the update is about to happen.
- An aspect can then decide whether the update should be skipped.

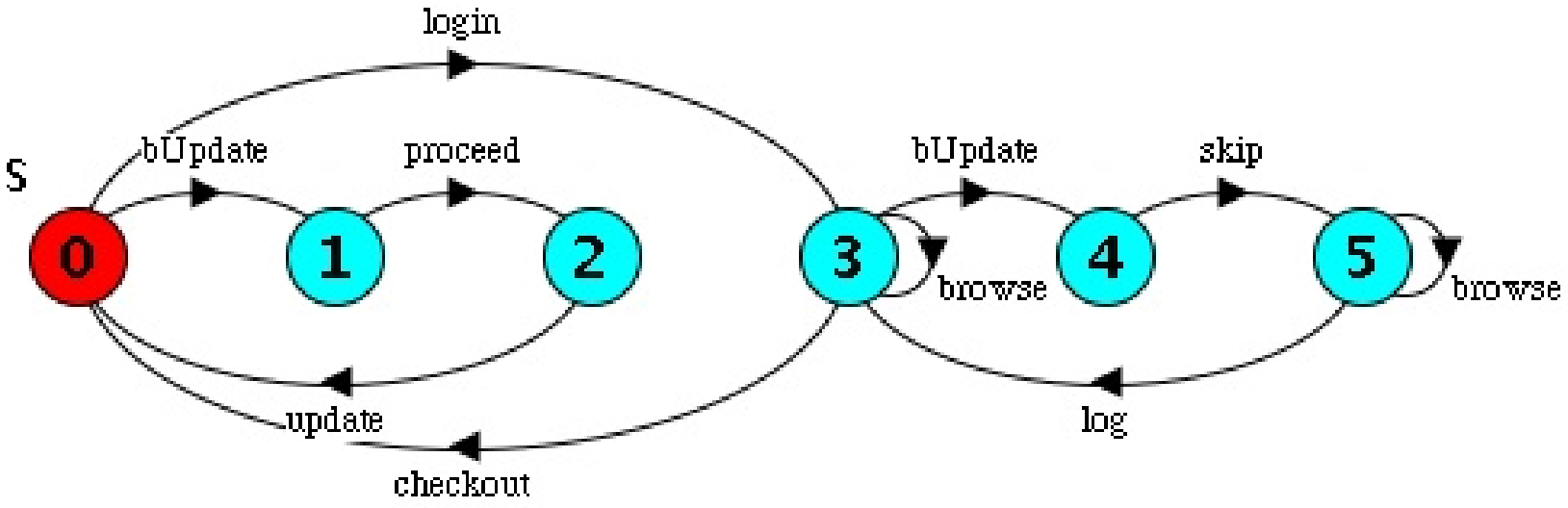
```
Server =
  ( login -> Session
  |  bUpdate -> ( skip -> Server
                | proceed -> update -> Server
                ) ),
Session =
  ( checkout -> Server
  |  bUpdate -> ( skip -> Session
                | proceed -> update -> Session
                )
  | browse -> Session).
```

From Pseudo-FSP to FSP

- The action `update` is replaced by `bUpdate` and `skip` is considered as a standard label.
- Pseudo FSP Completion: in each state, all the *shared* events, skippable or not skippable, have to be taken into account.

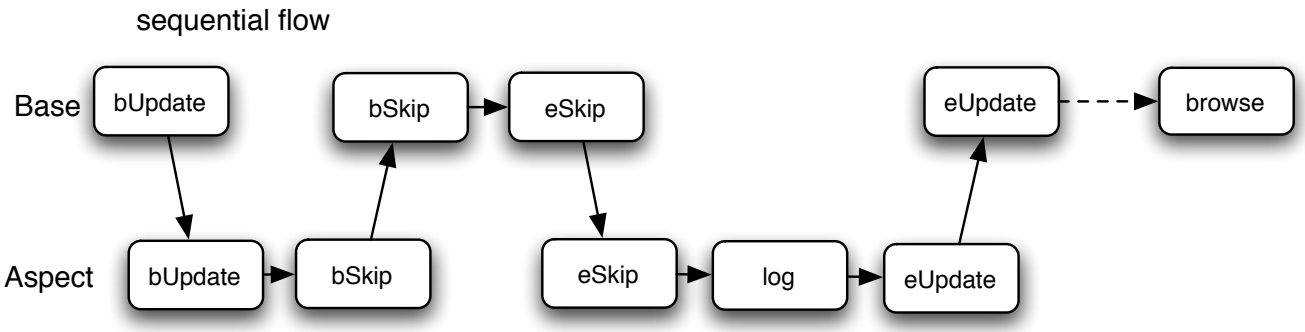
```
Consistency =
  ( login -> Session
  | bUpdate -> proceed -> Consistency
  | checkout -> Consistency
  ),
Session =
  ( bUpdate -> skip -> log -> Session
  | checkout -> Consistency
  | login -> Session
  ).
```

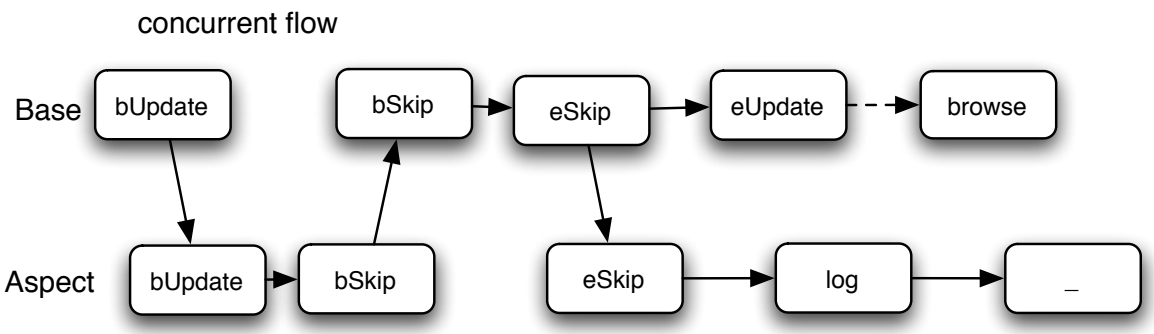
Woven LTS


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

Controlling concurrency between base and aspects

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





Summary

- Input:
  - the base model: an LTS `B`
  - the aspect model: `A`
- Output (a model of the *woven* application): `hiding(BaseTransf(B)) || hiding(AspectTransf(A))`
- The transformations are independent from the specific composition.
- Hiding makes it possible to control concurrency between aspect and base.

A Base Model with Several Clients

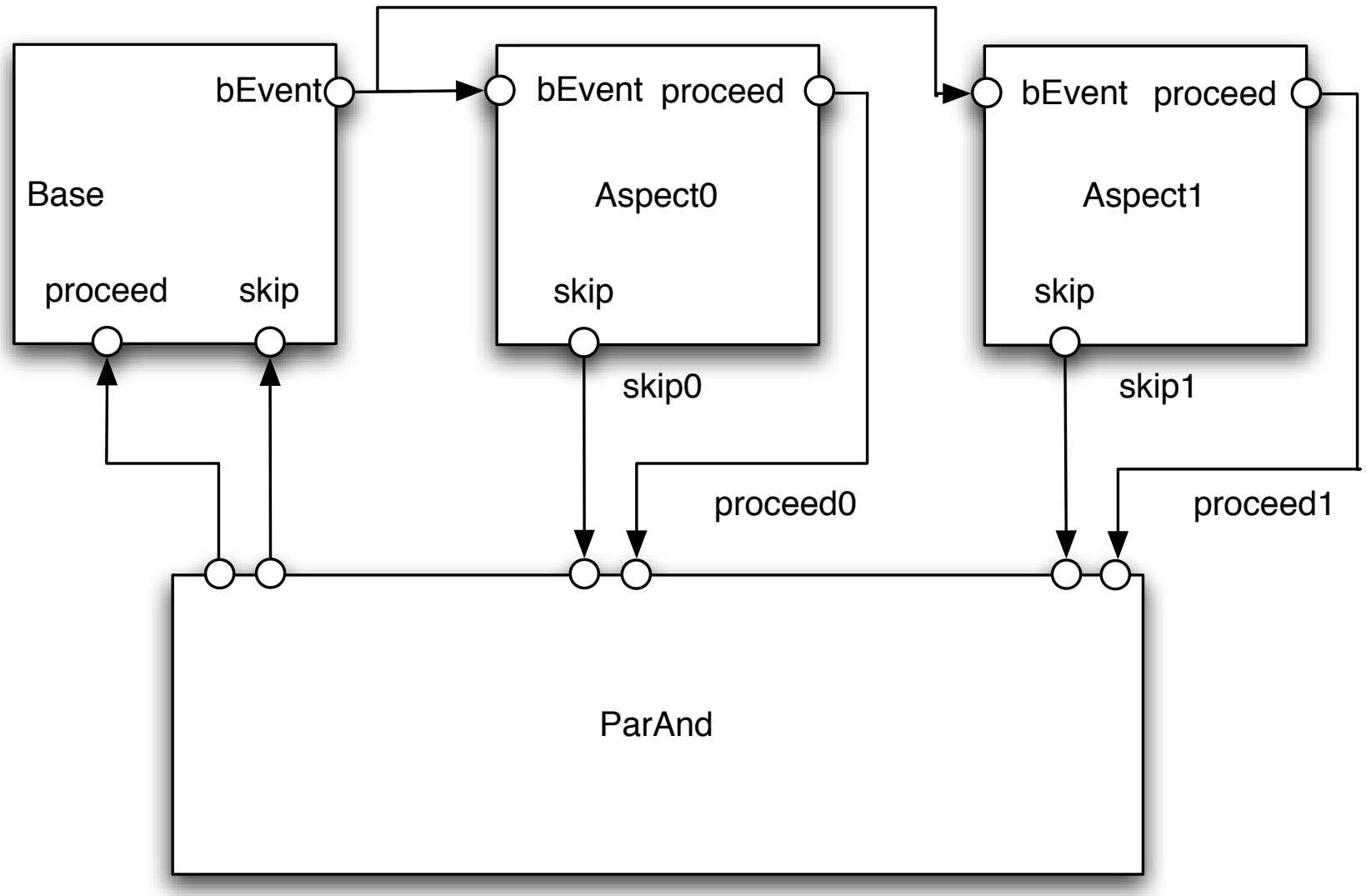
```
Server =
  ( login -> Session ),
Session =
  ( checkout -> Server
  | browse -> Session
  ).

Admin =
  ( bUpdate -> ( skip -> Admin
               | proceed -> update -> Admin
               )
  ).

||Base = (c[i:0..1]:Server || Admin).
```

Composing Aspects - Basic Idea

We need one aspect per server process. In case of an update, if the update is within one session, it should be skipped.

```
ParAnd = ( skip[0] -> ( skip[1] -> skip -> ParAnd
                      | proceed[1] -> skip -> ParAnd )
         | proceed[0] -> ( skip[1] -> skip -> ParAnd
                         | proceed[1] -> proceed -> ParAnd )
         | skip[1] -> ( skip[0] -> skip -> ParAnd
                      | proceed[0] -> skip -> ParAnd )
         | proceed[1] -> ( skip[0] -> skip -> ParAnd
                         | proceed[0] -> proceed -> ParAnd )
         ).
```

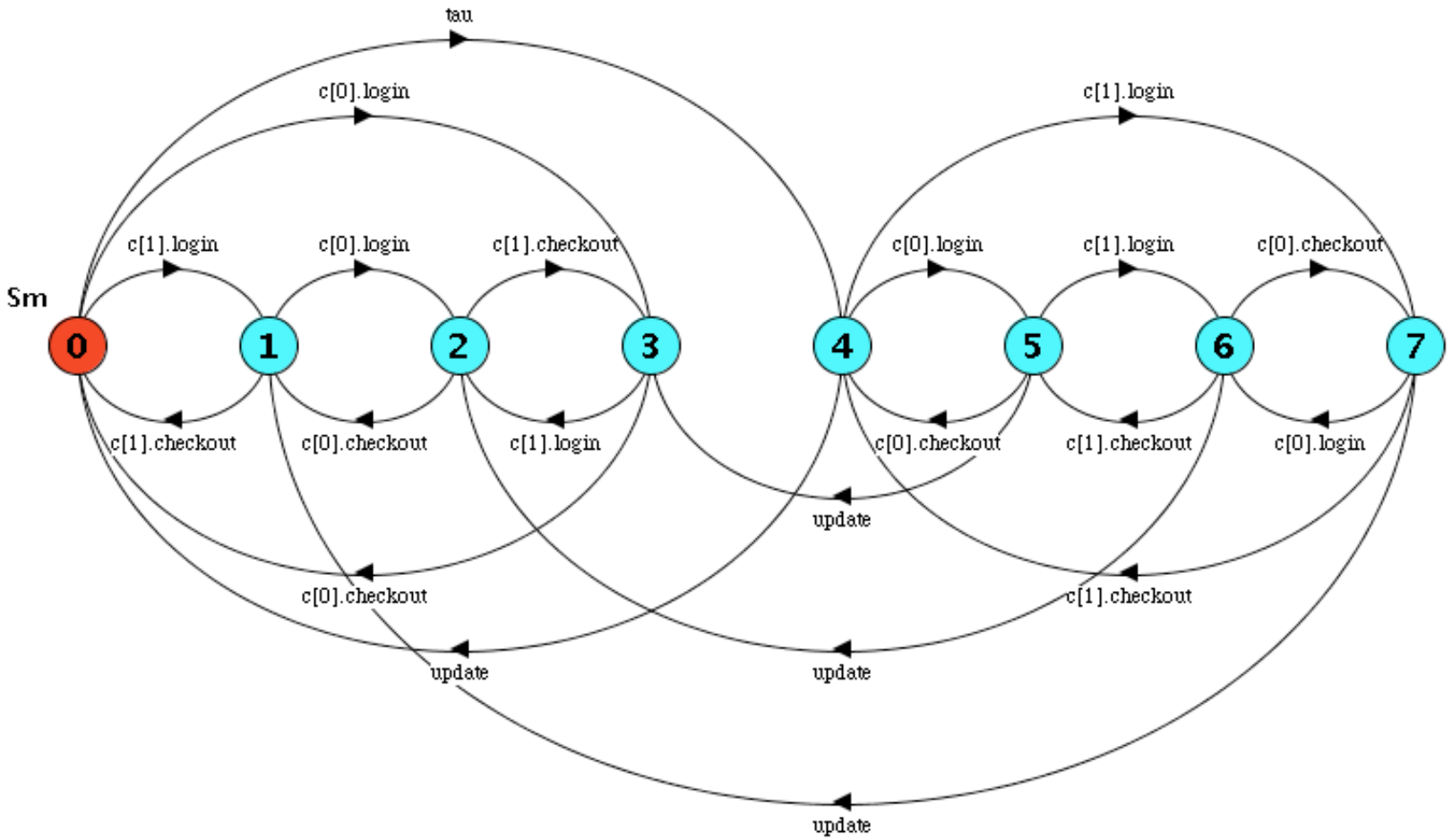
A Structural View



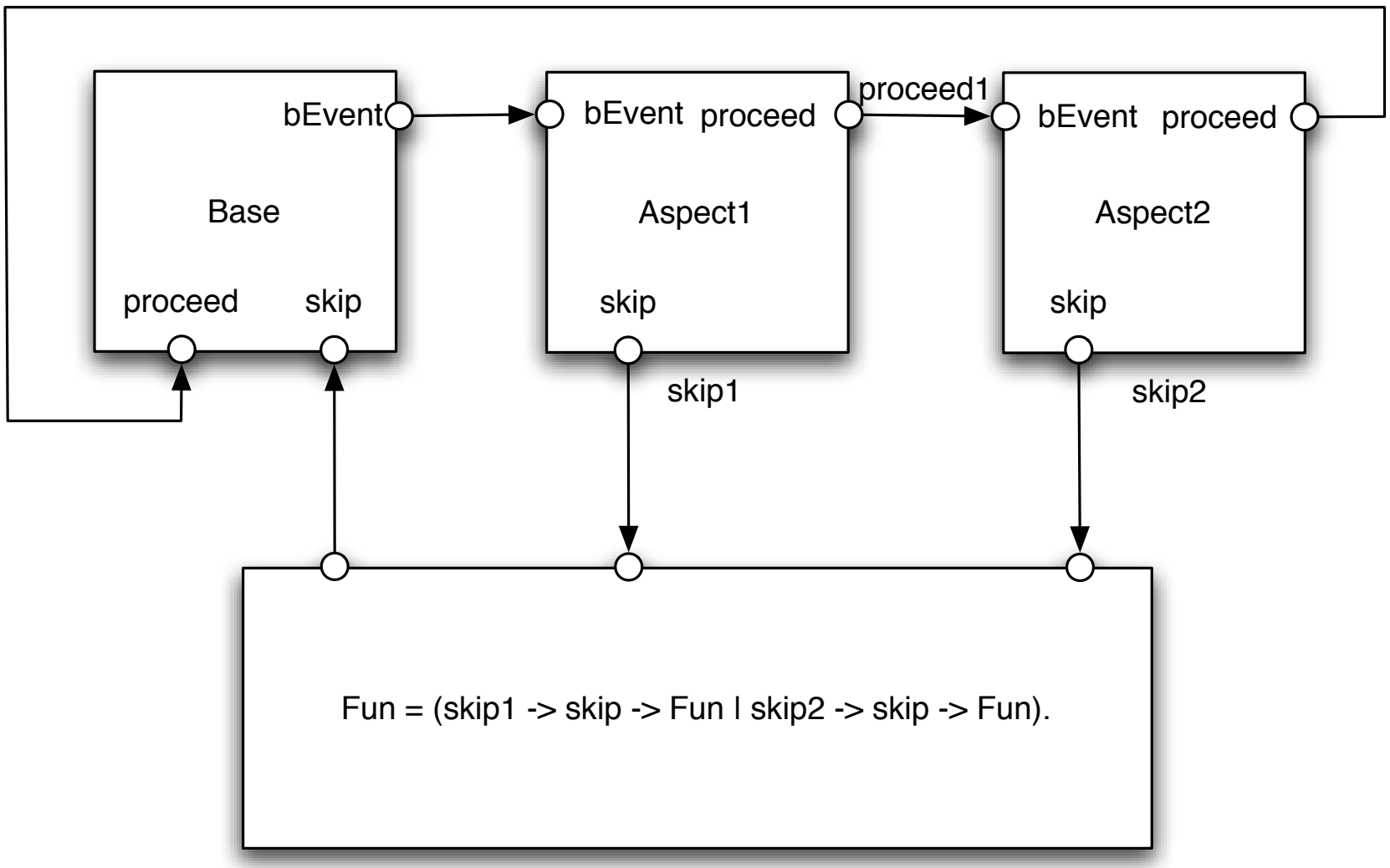
Wiring through Renaming

```
||S = (ParAnd || Base || c[i:0..1]:Consistency)
/{forall[i:0..1]{
    proceed[i]/c[i].proceed,
    skip[i]/c[i].skip,
            bUpdate/c[i].bUpdate,
            log[i]/c[i].log
            }
        }.

||Sm = S@{c[0].login, c[1].login,
          c[0].checkout, c[1].checkout,
          update}.
```

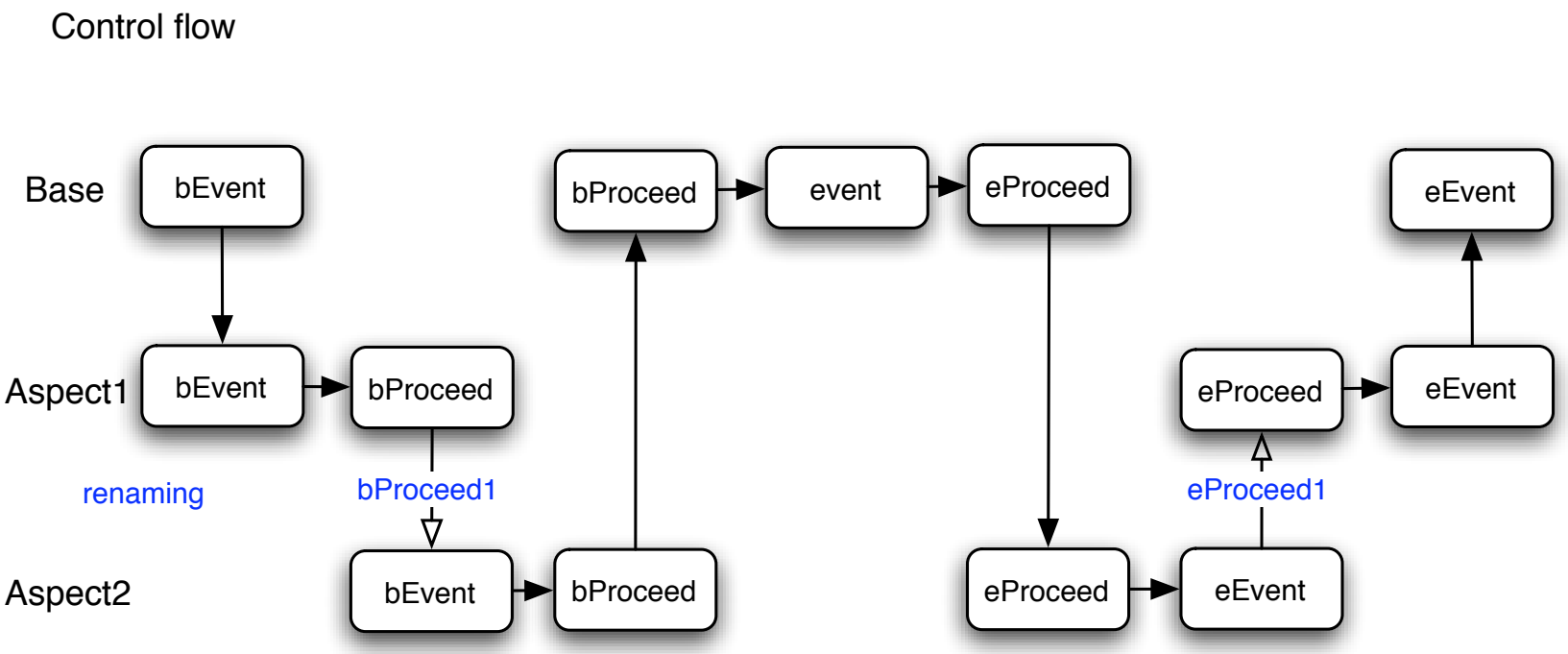
Woven LTS


tau
c[0].login
c[1].login
c[1].login
c[0].login
c[1].checkout
c[0].login
c[1].login
c[0].checkout
Sm
0
1
2
3
4
5
6
7
c[1].checkout
c[0].checkout
c[1].login
c[0].checkout
c[1].checkout
c[0].login
update
c[0].checkout
c[1].checkout
update
update
update

Fun - (Simplified) Structural View



Fun - Control Flow



Composing Aspects - Summary

- The aspect models are composed with the base model via operators.
- An operator is implemented as an LTS with an appropriate renaming.

Prototype in Java: Baton [NN07a]

- Each process is implemented as an active object.
- The base program is instrumented with pointcuts describing the events of interest.
- The transformations are used to synthesize aspects described in an FSP-like concrete syntax.
- Calls to a global monitor are used to synchronize shared actions (naïve but guarantees correctness 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);
}
```

Prototype mixing components and aspects [NN07b]

- The base program is structured as components with interfaces describing the *required* and *provided* services, as well as the *published* events (this is related to *open modules*).
- Aspect interfaces describe the events of interest, which may be *skippable*, as well as *required* and *provided* services.
- An application composed of aspects and components is turned into a composition of components.

Modelling Context-Aware Applications [NN08]

Context:

```
@InRoom     = ( enter:in -> leave:out -> InRoom ).
@Connected  = ( acquire:in -> release:out -> Connected ).
```

Rules:

```
+PlayDef(Cxt)  = ( in => play -> out => stop -> PlayDef(Cxt) ).
+ConnDef(Cxt)  = (  in => acquire ->
                       out => release -> ConnDef(Cxt),
                    | in => out -> ConnDef(Cxt) ).
+ConnRule      = ConnDef(InRoom).
+PlayRule      = PlayDef(Ready).
```

Achievements

- A formal model of concurrent event-based aspects defined using a transformation-based semantics [DLBNS06].
  - The base application as well as the aspects can be concurrent.
  - Composition operators are used to coordinate aspects and base.
- The aspects can be reused in various assemblies.
- Links between (event-based) aspects, components, and processes.
- Path to concrete languages (including support at the architectural level).

Perspectives

- Play with more applications.
- Improve the model (supported by an appropriate modelling language).
  - Using LTSs.
  - Using Visibly Pushdown Automata or Symbolic Transition Systems.
- Design a language (languages, DS(A)Ls) based on this model.

  - Integration with a redesign of CaesarJ (applying mixin inheritance to states, optimizations).
  - Integration with AWED.


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, 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)*, Portland, USA, October 2006. ACM Press.


J. Magee and J. Kramer.
Concurrency: State Models and Java.

Wiley, 2nd edition, 2006.


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.


Angel Núñez and Jacques Noyé.

An event-based coordination model for context-aware applications.
In *10th International Conference on Coordination Models and Languages (Coordination'08)*, Oslo, Norway, June 2008.