Distributed programming with aspect languages

Mario Südholt
http://www.emn.fr/sudholt

ASCOLA research group
École des Mines de Nantes-INRIA, LINA

AOSD-Europe Summer School, 23 July 2008

Outline

1. Introduction
2. Crosscutting in non-sequential applications
3. Language support for distributed and concurrent aspects
   - Aspects with explicit distribution (AWED)
     - Language
     - Prototype implementation
   - Aspects for concurrency
4. Application: architectural programming with invasive patterns
   - Pattern language
   - Ex.: JBoss Cache refactored
5. Perspectives and future work

Beyond sequential programs: problems

- Improve **execution speed** of resource-intensive calculations
  E.g.: weather simulations
- Improve **reactivity** of interactive programs
  E.g.: graphical user interfaces
- Improve **resource sharing**
  E.g.: sharing of physically distributed printers

Beyond sequential programs: programming paradigms

- **Massively parallel programs**: perform similar calculations on large number of similar (equal) processors
- **Concurrent programs**: multiple executions in an interleaved fashion on one machine
- **Distributed programs**: coordinate calculations on a, possibly heterogeneous, network linking, possibly different, machines

Basic goals of distributed programming

Heart of the story: **hiding** of

- **Access** to distributed data
- **Locations** of distributed entities
- **Migration** and **relocation** of distributed entities
- **Concurrent** execution
- Distribution-related **failures**

Complete hiding of distribution issues is unreasonable:
remote method calls, e.g., inherently different from local ones

$\Rightarrow$ **Explicitely distributed programming**

Basic goals of distributed programming

Heart of the story: **hiding** of

- **Access** to distributed data
- **Locations** of distributed entities
- **Migration** and **relocation** of distributed entities
- **Concurrent** execution
- Distribution-related **failures**

Complete hiding of distribution issues is unreasonable:
remote method calls, e.g., inherently different from local ones

$\Rightarrow$ **Explicitely distributed programming**

Basic goals of distributed programming

Heart of the story: **hiding** of

- **Access** to distributed data
- **Locations** of distributed entities
- **Migration** and **relocation** of distributed entities
- **Concurrent** execution
- Distribution-related **failures**

Complete hiding of distribution issues is unreasonable:
remote method calls, e.g., inherently different from local ones

$\Rightarrow$ **Explicitely distributed programming**

Basic goals of distributed programming

Heart of the story: **hiding** of

- **Access** to distributed data
- **Locations** of distributed entities
- **Migration** and **relocation** of distributed entities
- **Concurrent** execution
- Distribution-related **failures**

Complete hiding of distribution issues is unreasonable:
remote method calls, e.g., inherently different from local ones

$\Rightarrow$ **Explicitely distributed programming**

Concrete issues

- **Sequential execution** vs. **concurrency on one machine** vs. **parallel execution on several processors/machines**
- Execute activities on **multiple processors/machines**
  - Access to **distributed state**
- **Synchronous** (blocking) vs. **asynchronous** (non-blocking) **communication**
- **Error handling** of distribution, communication and concurrency failures

Distributed and concurrent applications

Large applications

- Graphical user interfaces
  $\longrightarrow$ typically concurrent applications
- Peer-to-peer systems
  $\longrightarrow$ concurrent and distributed
- Web-based/enterprise information systems
  $\longrightarrow$ concurrent and distributed

Distributed applications typically subject to concurrency issues

Distributed and concurrent applications

Large applications

- Graphical user interfaces
  $\longrightarrow$ typically concurrent applications
- Peer-to-peer systems
  $\longrightarrow$ concurrent and distributed
- Web-based/enterprise information systems
  $\longrightarrow$ concurrent and distributed

Distributed applications typically subject to concurrency issues

Languages vs. library-based approaches

Languages (this part)

- Provide language mechanisms for the expression of distributed aspects
- \+ Concise expression
- \+ Better readability and understandability
- – Potentially difficult integration with existing languages
- – Learning curve

Libraries, frameworks, APIs (second part: Prof. Wouter Joosen)

- Design library (OO framework) representing and implementing distributed aspects
- – More verbose and expression
- – Readability and understandability
- \+ Facilitates integration with existing languages
- \+ Learning curve

Languages vs. library-based approaches

Languages (this part)

- Provide language mechanisms for the expression of distributed aspects
- \+ Concise expression
- \+ Better readability and understandability
- – Potentially difficult integration with existing languages
- – Learning curve

Libraries, frameworks, APIs (second part: Prof. Wouter Joosen)

- Design library (OO framework) representing and implementing distributed aspects
- – More verbose and expression
- – Readability and understandability
- \+ Facilitates integration with existing languages
- \+ Learning curve

Outline

1 Introduction

2 Crosscutting in non-sequential applications

3 Language support for distributed and concurrent aspects
- Aspects with explicit distribution (AWED)
  - Language
  - Prototype implementation
- Aspects for concurrency

4 Application: architectural programming with invasive patterns
- Pattern language
- Ex.: JBoss Cache refactored

5 Perspectives and future work

Crosscutting and non-sequential applications

- **Are there crosscutting concerns in non-sequential applications**?

- YES! **Large number of different concerns**
  - Distribution, synchronization
  - Security
  - Transactions
  - Persistence
  - . . .

Crosscutting and non-sequential applications

- **Are there crosscutting concerns in non-sequential applications**?
- YES! **Large number of different concerns**
  - Distribution, synchronization
  - Security
  - Transactions
  - Persistence
  - . . .

Ex.: EJB-based application servers

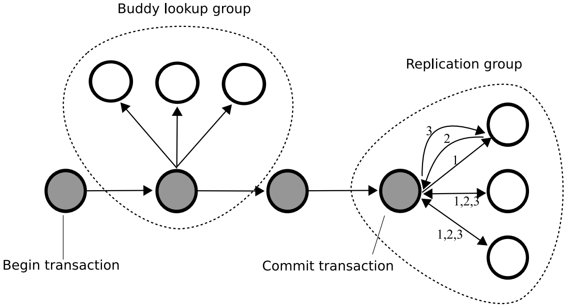
- EJB: **industrial (distributed) component model** defined by Sun Microsystems (`http://java.sun.com/products/ejb`)
  - **3-tier** distributed systems: distributed database, business logic, user-level presentation layer
  - 3 crosscutting functionalities: **persistance**, **transactional behavior**, **security**
  - Current specification: ˜550 pages (150: components, 400: crosscutting functionalities)
- **Application servers**: development and execution platform on top of such component models
  - Ex. **JBoss** application server (`jboss.com`)

Ex.: JBoss Cache

- **Data replication under transactional control**
- **Replication**: provide copies of a set of data on all machines
- **Transactions**: manage (concurrent) changes on each machine
  1. Do local changes (from begin to end of transaction)
  2. Globally coherent changes: success (transaction committed)
     incoherent changes: failure (transaction aborted)

  - All or nothing model: either commit all local changes or abort all changes and return to previous state

(Abstract) runtime architecture

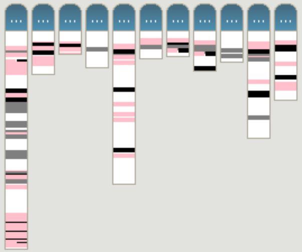


(Simple) composition of 3 standard parallel patterns

1. Transaction (pipe pattern)
2. Get data during transaction (farm)
3. Transaction (2-phase) commit protocol (farm and gather)

Crosscutting in JBoss Cache [Benavides et al., AOSD'06]

- Representation of part of codebase (JBoss Cache 1.2)
  - Class `TreeCache`: main data structure
  - `interceptor` package: modularization of replication and transaction code



- Crosscutting concerns:

  replication (black),
  transactions (dark gray),
  interception filters (light gray)

- Class `TreeCache`: 1741 lines of code (comments incl.)
- Replication code: $\geq$ 196 LOC
- Transactions: $\geq$ 228 LOC

Resulting problems

- What about
  - **nicely modularizing** transactions and replication code?
  - **adding** new replication policies?
- Requires **different modifications at multiple places** among the 424 LOC concerning replication and transactions
- Involves modifications involving **different distributed entities**, in particular machines

Outline

1. Introduction
2. Crosscutting in non-sequential applications
3. Language support for distributed and concurrent aspects
   - Aspects with explicit distribution (AWED)
     - Language
     - Prototype implementation
   - Aspects for concurrency
4. Application: architectural programming with invasive patterns
   - Pattern language
   - Ex.: JBoss Cache refactored
5. Perspectives and future work

Overview of approaches for distributed AOP

- Surprisingly few work
- Most approaches: **apply sequential AOP to distributed infrastructures**
  - Java RMI and AspectJ [Soares et al., OOPSLA'02]
  - Manipulation of IBM's EJB-based application server Websphere using AspectJ [Colyer, Clement, AOSD'04]
  - Frameworks for sequential AOP over distributed infrastructures: small language-level support
    - Spring AOP
    - JBoss AOP
    - see Prof. Wouter Joosen's part
  - ...

Sequential AOP and distributed infrastructures

Ex.: AspectJ and RMI [Soares et al., OOPSLA'02]

Server-side distribution

```
aspect ServerSideHWDistribution {
  public static void
      HWFacade.main(String[] args) {
    try {
      HWFacade f = HWFacade.getInstance();
      UnicastRemoteObject.exportObject(f);
      java.rmi.Naming.rebind("/HW",f);
    } catch (Exception rmiEx) { ... }
  }
  ...
}
```

Client-side distribution

```
aspect ClientSideHWDistribution {
  pointcut facadeCalls(HWFacade f):
    target(f) && call(* *(..))
      && !call(static * *(..))
      && this(HttpServlet);

  int around(Complaint c) throws /*...*/:
      facadeCalls() && args(c) &&
      call(int registerComplaint(Complaint)) {
    return remoteHW.registerComplaint(c);
  }
}
```

- Register server with naming service
- Redirect client facade calls to server
- No modular definition of distributed functionalities

Challenges

- No explicit support for **concurrent** and **distributed** abstractions
- Each aspect only treats parts of crosscutting functionality of non-sequential applications
  - Ex.: separate aspects for client and server sides in client-server architectures
    [Soares et al., OOPSLA'02]
  - Ex.: multiple aspects for handling of distributed transactions

Approaches for AOP with explicit distribution

(Almost) exhaustive list of approaches:

- **Riddle** [Lopes'97]: argument passing modes
- **JAC** [Pawlak et al., Reflection'01]: Java framework (no dedicated language) with features for matching of distributed events
- **Djcutter** [Nishizawa et al., AOSD'04]: `host`, distributed `cflow`
- **ReflexD** [E. Tanter et al., DAIS'06]: a reflective kernel for distributed AOP
- **AWED** [Benavides et al., AOSD'06]: detailed later

Expressive aspect languages

- Make **explicit relationships** between execution events
  Eliminate use of non-local state in aspects
- Means
  - **"Stateful aspects"** [Douence et al., GPCE'02]
    Synonyms: history-based or trace-based aspects
  - **Richer pointcut languages**
    Regular aspects, temporal logic-based aspects, logic pointcuts, etc.
  - **Domain-specific** sublanguages in pointcuts and advice
    Especially for distribution and concurrency
- Goals
  - **Enhance expressivity**: enable expression of complex aspects
  - Support definition and **reasoning** of aspectual properties

Expressive aspect languages

- Make **explicit relationships** between execution events
  Eliminate use of non-local state in aspects
- Means
  - **“Stateful aspects”** [Douence et al., GPCE’02]
    Synonyms: history-based or trace-based aspects
  - **Richer pointcut languages**
    Regular aspects, temporal logic-based aspects, logic pointcuts, etc.
  - **Domain-specific** sublanguages in pointcuts and advice
    Especially for distribution and concurrency
- Goals
  - **Enhance expressivity**: enable expression of complex aspects
  - Support definition and **reasoning** of aspectual properties

Expressive aspect languages

- Make **explicit relationships** between execution events
  Eliminate use of non-local state in aspects
- Means
  - **“Stateful aspects”** [Douence et al., GPCE’02]
    Synonyms: history-based or trace-based aspects
  - **Richer pointcut languages**
    Regular aspects, temporal logic-based aspects, logic pointcuts, etc.
  - **Domain-specific** sublanguages in pointcuts and advice
    Especially for distribution and concurrency
- Goals
  - **Enhance expressivity**: enable expression of complex aspects
  - Support definition and **reasoning** of aspectual properties

Outline

1 Introduction

2 Crosscutting in non-sequential applications

3 Language support for distributed and concurrent aspects
- Aspects with explicit distribution (AWED)
  - Language
  - Prototype implementation
- Aspects for concurrency

4 Application: architectural programming with invasive patterns
- Pattern language
- Ex.: JBoss Cache refactored

5 Perspectives and future work

Motivation: transactional replicated caches

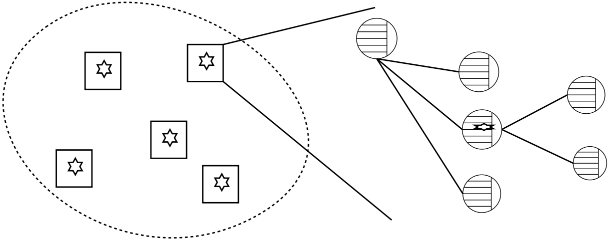
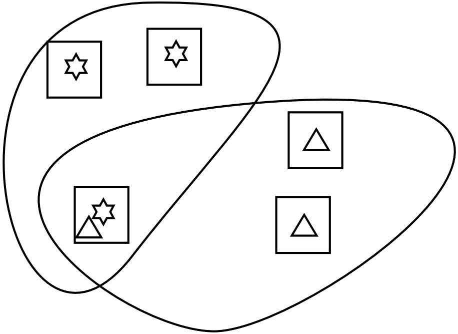


Figure: a) Replicated Caches b) Zoom of Data structure

- Cache **data structure** deployed on each node
- Data **replication** under control of **transactions**
- JBoss Cache (version 1.2):
  replicate item in `TreeCache` structure on all hosts of a cache

Ex.: support cache evolution



- New replication policies
  - **Don’t replicate unnecessarily** huge objects
  - Replicate only in case of **interest**

Modularization of distribution concerns

Requirements for distribution-specific aspect abstractions:

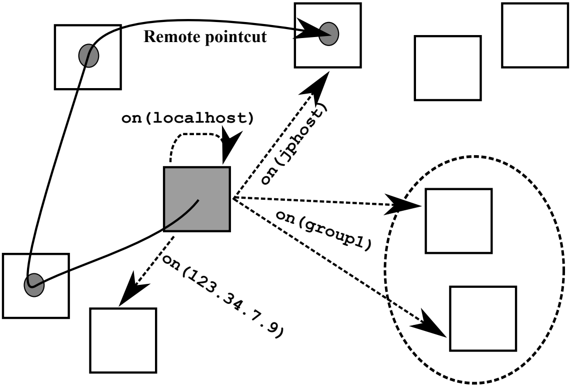
- Detection of **remote events**
- **Remote execution** of code
- Support for **distributed state**
- **Distributed deployment** of aspects

Aspects With Explicit Distribution (AWED)

- **Remote pointcuts**
  - References to remote hosts: `host`, on, host groups
  - Sequence pointcuts: `seq`, `step`
- **Remote advice**
  - Asynchronous and synchronous execution: `syncex`
  - Synchronization between interacting advice using futures
- **Distributed aspects**
  - Deployment: `single` and `all`
  - Instantiation: *e.g.*, `perthread`, `perbinding`...
  - State sharing: *e.g.*, `global`, `group(Group)`

[Benavides et al., AOSD'06]

Remote pointcuts and advice



Remote pointcuts examples

Replication pointcuts for a replicated cache application:

- Using the `host` pointcut:

  ```
  call(* Cache.put(Object,Object))
      && !host(localhost)
  ```

- Using the on pointcut:

  ```
  call(* Cache.put(Object,Object))
      && !on(jphost)
  ```

AWED sequence examples

Replication protocol for a lazy replicated cache (delimit via start/stop)

```
pointcut replPolicy(String key, Object o):
  replS: seq(s1: startCache() → s3 || s2,
             s2: cachePut(key, o) → s3 || s2,
             s3: stopCache() → s1)

pointcut putVal(String key, Object o):
   step(replS, s2) && args(key, o)
```

- Essentially (non-deterministic) finite-state automata

Remote Advice

- 2 synchronization **modes**: a/synchronous
  Access to result managed using **futures**
- Management of **groups**:
  dynamically add hosts to or remove from groups
- Ex: replication advice:

```
before(String k, Object o):
    localCachePut(k, o){
        addGroup(k);
        proceed();
}
```

Ex.: lazy replication aspect

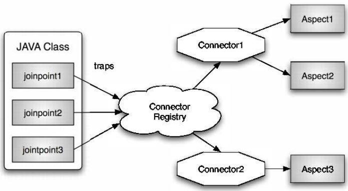
```
all aspect CacheReplication{
  pointcut cachePcut(Object key, Object o):
      call(* Cache.put(Object, Object))
      && args(key,o) && !on(jphost)
      && !within(CacheReplication);

  before(String k, Object o):
      cachePcut(k, o) && on(interestedHosts) {
          Cache.getInstance().put(k, o);
  }
}
```

- Aspect deployed on all hosts
- Matches `put` method in all remote hosts in the group of interest of value k
- Replicate call only if interest in that value

Implementation basis: JAsCo Infrastructure

- Tool from SSEL group at Free University of Brussels
- **Dynamic** aspect weaver for **Java**
- Supports **sequence pointcuts**
- **Connector registry**
  Connectors
  - control distribution of execution events that represent matched join points and
  - manage aspect compositions and resolve interactions



Distributed JAsCo Infrastructure

- One connector registry per host
- Aspects/Connectors registered in connector registries
- **Joinpoints** propagated among connector registries

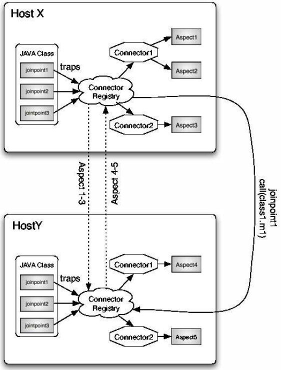


Figure: DJAsCo architecture.

Overview of implementation

- Remote pointcuts
  - JP propagation using **JGroups** (a popular Java library for multicast communication)
  - Cflow: **customized sockets** (extension mechanism of Java RMI)
  - Remote Sequences: extension of **JAsCo sequences**
- Remote advice
  - Based on **activation of deployed aspects**
  - Synchronization by futures implemented using **AWED aspects**
- Distributed aspects
  - **Connector distribution** using JGroups
  - Aspect state sharing, parameter passing: **AWED aspects**

Example for **integration of diverse implementation techniques/platforms**

Overview of implementation

- Remote pointcuts
  - JP propagation using **JGroups** (a popular Java library for multicast communication)
  - Cflow: **customized sockets** (extension mechanism of Java RMI)
  - Remote Sequences: extension of **JAsCo sequences**
- Remote advice
  - Based on **activation of deployed aspects**
  - Synchronization by futures implemented using **AWED aspects**
- Distributed aspects
  - **Connector distribution** using JGroups
  - Aspect state sharing, parameter passing: **AWED aspects**

Example for **integration of diverse implementation techniques/platforms**

Overview of implementation

- Remote pointcuts
  - JP propagation using **JGroups** (a popular Java library for multicast communication)
  - Cflow: **customized sockets** (extension mechanism of Java RMI)
  - Remote Sequences: extension of **JAsCo sequences**
- Remote advice
  - Based on **activation of deployed aspects**
  - Synchronization by futures implemented using **AWED aspects**
- Distributed aspects
  - **Connector distribution** using JGroups
  - Aspect state sharing, parameter passing: **AWED aspects**

Example for **integration of diverse implementation techniques/platforms**

Overview of implementation

- Remote pointcuts
  - JP propagation using **JGroups** (a popular Java library for multicast communication)
  - Cflow: **customized sockets** (extension mechanism of Java RMI)
  - Remote Sequences: extension of **JAsCo sequences**
- Remote advice
  - Based on **activation of deployed aspects**
  - Synchronization by futures implemented using **AWED aspects**
- Distributed aspects
  - **Connector distribution** using JGroups
  - Aspect state sharing, parameter passing: **AWED aspects**

Example for **integration of diverse implementation techniques/platforms**

Outline

1 Introduction

2 Crosscutting in non-sequential applications

3 Language support for distributed and concurrent aspects
- Aspects with explicit distribution (AWED)
  - Language
  - Prototype implementation
- Aspects for concurrency

4 Application: architectural programming with invasive patterns
- Pattern language
- Ex.: JBoss Cache refactored

5 Perspectives and future work

Aspects for concurrency: overview of approaches

- Few approaches, even fewer than for distributed aspects
- Use of **sequential AO systems** applied to concurrency libraries
  - Aspect-based implementation of concurrency patterns using the Java concurrency library [Cunha et al., AOSD'06]
- Approaches with **explicit AO-abstractions** for concurrency
  - COOL [Lopes'97]
    Aspects define mutual exclusion relationships on base methods
  - Aspects as concurrent processes [Andrews, Reflection'01]
    CSP-based aspect language, incomplete weaver definition
  - Concurrent Event-based AOP [Douence et al., GPCE'06]

Aspects for concurrency: overview of approaches

- Few approaches, even fewer than for distributed aspects
- Use of **sequential AO systems** applied to concurrency libraries
  - Aspect-based implementation of concurrency patterns using the Java concurrency library [Cunha et al., AOSD'06]
- Approaches with **explicit AO-abstractions** for concurrency
  - COOL [Lopes'97]
    Aspects define mutual exclusion relationships on base methods
  - Aspects as concurrent processes [Andrews, Reflection'01]
    CSP-based aspect language, incomplete weaver definition
  - Concurrent Event-based AOP [Douence et al., GPCE'06]

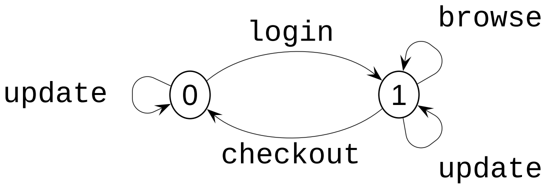
Concurrent Event-based AOP (CEAOP)

- Goals
  - Definition of base and aspects based on **process calculus**
  - **Synchronization of base program and aspects** in terms of the aspect structure
  - Explicit**composition operators** for flexible synchronization
  - **Formal definition** of aspects and base programs and support for property verification
  - **Java-based implementation**

[Douence et al., GPCE’06]

Definition of base programs

- Definition of base and aspects as **Finite State Processes** (FSP)
  Ex.: e-commerce base program

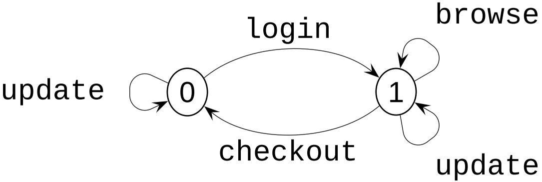


(0)	Server	=	login $\rightarrow$ InSession
		$\mid$	update $\rightarrow$ Server,
(1)	InSession	=	checkout $\rightarrow$ Server
		$\mid$	update $\rightarrow$ InSession,
		$\mid$	browse $\rightarrow$ InSession.

Aspects

- Finite-state pointcuts with interleaved actions as advice
  Advice must contain `proceed` or `skip` to execute or not the matched base functionality
- Ex. consistency aspect: suppress updates (price changes) in sessions

$$\mu a.\left(\texttt{login}; \mu a'.\left(\left(\texttt{update} \triangleright \texttt{skip}\ \texttt{log}; a'\right) \,\square\, \left(\texttt{checkout}; a\right)\right)\right)$$



- Ex. safety aspect: rehash and backup views after updates

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

Aspect composition

- **Composition: synchronize parts of advice applying at the same execution points**
  - Form of advice: $(\mathit{before}, \mathit{control}, \mathit{after})$ where $\mathit{control} \in \{\texttt{proceed}, \texttt{skip}\}$
  - Synchronize different advice: execute different *before* (*after*) parts sequential/concurrently
  - Generalization of standard AspectJ-style aspect composition: sequentialized advice execution
- Ex.: operator $\mathbf{ParAnd}(\mathbf{A_1}, \mathbf{A_2})$ executes *before* and *after* in parallel, proceeds to base functionality iff both aspects proceeds
- Resolve **interactions on common execution events**
  Ex.: $\mathbf{ParAnd}(\mathbf{Consistency}, \mathbf{Safety})$ backs up where necessary and in parallel to base functionality

Aspect weaving

- Instrument aspects and base program with **synchronization events** that delimit parts of advice (*before*, *control*, *after*)
- Goal: translate aspects and base program into **plain FSP expressions** and use **standard parallel composition** (as used in FSP)
- Ex.: Instrumentation of consistency aspect

a = (login → a'
| eventB_update → proceedB_update → proceedE_update → eventE_update → a
| checkout → a | browse → a),

a' = (eventB_update → skipB_update → skipE_update → log → eventE_update → a'
| checkout → a
| browse → a' | login → a').

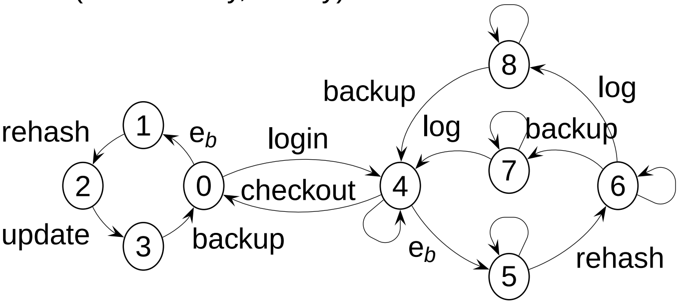
Weaving of composition operator **ParAnd**

- Translated as well into **plain FSP expressions**

(skipB_e1 → (skipB_e2 → skipB_e → skipE_e → skipE_e1 → skipE_e2 → ParAnd
| proceedB_e2 → skipB_e → skipE_e → skipE_e1 → proceedE_e2 → ParAnd)

| proceedB_e1 → (skipB_e2 → skipB_e → skipE_e → skipE_e2 → proceedE_e1 → ParAnd
| proceedB_e2 → proceedB_e → proceedE_e →
proceedE_e1 → proceedE_e2 → ParAnd)).

- Ex.: ParAnd(Consistency, Safety)



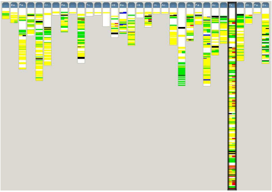
Property analysis

- Properties of interest
  - **Safety** properties ("sth bad never happens")
    Ex.: the price of products does not change once it has been put in the caddy
  - **Liveness** properties ("sth good eventually happens")
    Ex.: absence of deadlock, skipping of `updates` does not eliminate all price updates
  - General **properties of composition operators**
    Ex.: associativity
- Tool **LTSA**: visualization, simulation and automatic property analysis using model checking techniques

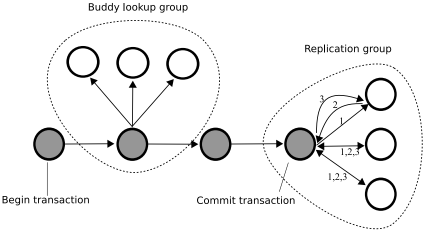
Outline

1. Introduction
2. Crosscutting in non-sequential applications
3. Language support for distributed and concurrent aspects
   - Aspects with explicit distribution (AWED)
     - Language
     - Prototype implementation
   - Aspects for concurrency
4. Application: architectural programming with invasive patterns
   - Pattern language
   - Ex.: JBoss Cache refactored
5. Perspectives and future work

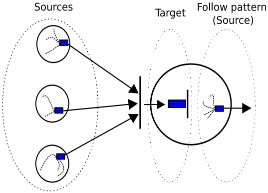
Ex.: “architectural programming” of JBoss Cache



Implementation (~16 KLOC)



Runtime architecture



Invasive pattern `gatherI`

```
pipeI( ...
    farmI(...) ...;
    farmI(...
        gatherI(...
            farmI(...)));
```

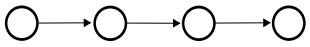
Program

Requirements for suitable notion of patterns

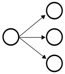
- **Modularization of similar (micro) pattern applications in different contexts**
    - Unlike standard distributed patterns
- **Remote executions and multicast communication**
    - Similar to standard distributed patterns
- **Heterogenous synchronization requirements**
    - Unlike standard distributed patterns

Approach: invasive distributed patterns

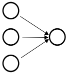
- **Extend standard computation and communication patterns**
  - Source side: context-dependent pattern applications
    - Extraction of pattern-enabling state information
    - Preparation of data and communication
  - Target side: remote executions
  - multicast communication and heterogenous synchronization requirements: cooperation of both sides
- **Invasive distributed patterns** integrate these features
- Implementation by means of **stateful distributed aspects**



a) pipe



b) farm



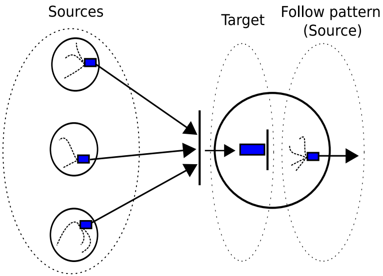
c) gather

Outline

1. Introduction
2. Crosscutting in non-sequential applications
3. Language support for distributed and concurrent aspects
    - Aspects with explicit distribution (AWED)
        - Language
        - Prototype implementation
    - Aspects for concurrency
4. Application: architectural programming with invasive patterns
    - Pattern language
    - Ex.: JBoss Cache refactored
5. Perspectives and future work

Ex.: invasive (gather) pattern

- **n-to-1** communication and data integration
- Source and target **groups** of computing units
- **Sequences over remote events** to quantify over different pattern-enabling contexts
- **Data prepared** on source units and **integrated** in target units

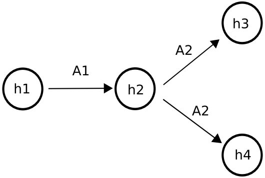


- **Asynchronous execution of source computations** and communications
- **Synchronization** between sources and host as well as target integration and follow pattern

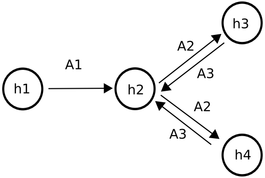
Pattern language and compositions

P	::=	`patternSeq` $G_1\ A_1\ G_2\ A_2 \ldots G_n$
G	::=	$H\ G \mid P\ G \mid \epsilon$
A	::=	`aspect { around`((H, Id*)*)`:` *PCD* *SourceAdvice* [sync] *TargetAdvice* }
PCD	::=	`call`(*MSig*) | `target`(*Id*) | `args`(*Id*+)
	|	*PCD* && *PCD* | *PCD* || *PCD* | ! *PCD*
	|	*Seq*

- **`patternSeq`** $G_1\ A_1\ G_2\ A_2 \ldots G_n$
  - $G_i\ A_i\ G_{i+1}$: pattern application
    triple of source group, distributed aspect, target group
- Examples:



patternSeq(h1,A1,h2,A2,(h3,h4))



patternSeq(h1,A1,h2,A2,(h3,h4),A3,h2)

Outline

1. Introduction
2. Crosscutting in non-sequential applications
3. Language support for distributed and concurrent aspects
   - Aspects with explicit distribution (AWED)
     - Language
     - Prototype implementation
   - Aspects for concurrency
4. Application: architectural programming with invasive patterns
   - Pattern language
   - Ex.: JBoss Cache refactored
5. Perspectives and future work

Pattern-based definition of JBoss Cache

```
gCaches = {h1, h2, h3}
pipe([h], // h is current host
    Atransac,
    farm(
      gather(
        farm([h], Aprepare, sync gCaches-[h]),
        Apresp,
        [h]),
      Acommit,
      gCaches-[h])
   );
```

- Composition of **4 pattern applications**
- **3 aspects** used to modularize crosscutting pattern-enabling state accesses
- **Directly implements overall runtime architecture** of replicated caching

Ex. Aprepare: aspectized prepare phase

```
aspect Aprepare {
  org.jboss.cache.TreeCache tc = CacheRegistry.getInstance(

  around(DataStorage d, String txId):
    call(* PrepareHelper.send(..)) && args(d,s) &&
    !cflow(call(TransactionManager.prepare(..)))

    // Source advice
    { proceed(); }

    // Target advice
    { TransactionManager tm = TransactionManager.getInstance
      PrepareHelper ph = new PrepareHelper();
      try{
      tm.prepare(d, txId, tc);
        ph.respAgree(txId);
      } catch(Exception e) {
        ph.respNotAgree(txId);
```

Evaluation

- **Significant reduction in code size and complexity**
  - Original JBoss Code: 2674 LOC in 17 classes with intricate scattering
  - Refactored functionally-equivalent solution: 532 LOC **($\sim$ 80% reduction)** in **11 well-modularized aspects and classes**
- **Execution overhead**
  - Amortized over application typically very low
  - Benchmark on cache use only: **currently factor 3**
  - Dynamic aspect application using aspect execution infrastructure on top of J2SE
  - Use of JGroups library for multicast communication

Much optimization potential: compile patterns, optimize communication

No reason not to match original performance:
use static aspect system

Outline

1. Introduction
2. Crosscutting in non-sequential applications
3. Language support for distributed and concurrent aspects
   - Aspects with explicit distribution (AWED)
     - Language
     - Prototype implementation
   - Aspects for concurrency
4. Application: architectural programming with invasive patterns
   - Pattern language
   - Ex.: JBoss Cache refactored
5. Perspectives and future work

Conclusion

- Distributed and concurrent applications are subject to numerous crosscutting concerns.
- Currently, few approaches for non-sequential AOP: no domain-specific support, correctness difficult to evaluate
- Expressive aspects as means to tackle drawbacks of AOP
- Distributed AOP with AWED:
  Remote pointcuts, remote advice, distributed aspects
- Concurrent AOP:
  Synchronization among aspects and base, composition operators, tool-based property support

Future work

- Aspects for distributed programming
  - Composition operators
  - Formal semantics, property support (p.ex. correctness, QoS)
  - Introduce causality guarantees for distributed aspects
- Concurrent aspects
  - Larger set of composition operators
  - Efficient implementation in mainstream languages
- Application to grid algorithms (forthcoming publication in Oct.'08)

References (1)

[Akşit et al., Addison-Wesley'04] Mehmet Akşit, Siobhán Clarke, Tzilla Elrad, Robert E. Filman (eds.): *"Aspect-Oriented Software Development"*, Addison-Wesley Professional, Sep. 2004.

[Kiczales, CIOO'96] Gregor Kiczales: *"Aspect Oriented Programming"*, Proc. of the Int. Workshop on Composability Issues in Object-Orientation (CIOO'96) at ECOOP, July 1996.

[Tarr et al., ICSE'99] Peri Tarr, Harold Ossher, William Harrison, Stanley M. Sutton, Jr: *"N Degrees of Separation: Multi-dimensional Separation of Concerns"*, Proceedings of ICSE, 1999.

[Soares et al., OOPSLA'02] Sergio Soares, Eduardo Laureano, Paulo Borba: *"Implementing distribution and persistence aspects with AspectJ"*, Proc. of OOPSLA, Nov. 2002.

[Colyer, Clement, AOSD'04] Adrian Colyer, Andrew Clement: *"Large-scale AOSD for Middleware"*, Proc. of AOSD, Mar. 2004.

[Benavides et al., AOSD'06] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, Davy Suvée: *"Explicitly distributed AOP using AWED"*, Proc. of AOSD, Mar. 2006.

[Douence et al., GPCE'02] Rémi Douence, Pascal Fradet, Mario Südholt: *"A framework for the detection and resolution of aspect interactions"*, Proc. of GPCE, Oct. 2002.

References (2)

[Lopes'97] Cristina Videira Lopes: *"D: A Language Framework for Distributed Programming"*, PhD thesis, Northeastern University, 1997.

[Pawlak et al., Reflection'01] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gérard Florin: *"JAC: A Flexible Solution for Aspect-Oriented Programming in Java"*, Proc. of Reflection, Sep. 2001.

[Nishizawa et al., AOSD'04] Muga Nishizawa, Shigeru Chiba, Michiaki Tatsubori: *"Remote Pointcut - A Language Construct for Distributed AOP"*, Proc. of AOSD, Mar. 2004.

[E. Tanter et al., DAIS'06] É. Tanter, R. Toledo: *"A versatile kernel for distributed AOP"*, Proc. of DAIS, June 2006.

[Andrews, Reflection'01] James Andrews: *"Process-Algebraic Foundations of Aspect-Oriented Programming"*, Proc. of Reflection, Sep. 2001.

[Cunha et al., AOSD'06] Cunha, Carlos A. and Sobral, João L. and Monteiro, Miguel P. : *"Reusable Aspect-Oriented Implementations of Concurrency Patterns and Mechanisms"*, Proc. of AOSD, Mar. 2006.

[Douence et al., GPCE'06] Rémi Douence, Didier Le Botlan, Jacques Noyé, Mario Südholt: *"Concurrent Aspects"*, Proc. of GPCE, Oct. 2006.

[Benavides et al., DOA'07] D. Benavides, M. Südholt, R. Douence, J.-M. Menaud: *"Invasive patterns for distributed applications"*, Proc. of DOA, Nov. 2007.