From Modeling to Deployment of Active Objects - A ProActive backend for ABS

Ludovic Henrio, Justine Rochas

With the contribution of: Fabrice Huet, Zsolt István

Active Object Workshop, Sep 2015
Agenda

I. Active Object Programming models

II. Multi-active Objects: Principles

III. Scheduling in Multi-active Objects

IV. A ProActive backend for ABS

V. Conclusion and Future Works
ASP/ProActive

- No race condition: each object manipulated by a single thread
- Active and Passive objects
- Asynchronous method calls; request queue
- With implicit transparent futures

ASP/ProActive

- No race condition: each object manipulated by a single thread
- Active and Passive objects
- Asynchronous method calls; request queue
- With implicit transparent futures
ASP/ProActive

• No race condition: each object manipulated by a single thread
• Active and Passive objects
• Asynchronous method calls; request queue
• With implicit transparent futures
ASP/ProActive

- No race condition: each object manipulated by a single thread
- Active and Passive objects
- Asynchronous method calls; request queue
- With implicit transparent futures
ASP/ProActive

- No race condition: each object manipulated by a single thread
- Active and Passive objects
- Asynchronous method calls; request queue
- With implicit transparent futures

```java
A beta = new Active("A", ...);
V result = beta.foo(b);
...
result.getval();
```

ProActive is a Java library
ASP is a “calculus”

Active objects are the unit of distribution and concurrency (one thread per AO / no data shared)

Futures are transmitted between AOs and accessed transparently
- COGs (set of objects)
- Each object is active (can be invoked remotely)
- Cooperative scheduling
- Explicit syntax for asynchronous call and future access

```
foo(A a) {
    Fut<V> vFut = a!bar(p);
    await vFut?;
    V v = vFut.get;
}
```

```java
A a = new cog A();
B b = new cog B();
b!foo(a); b!foo(a);
```
ABS versus ProActive

Creol, ABS, JCoBox
- Explicit asynchronous calls
  object.method() // synchronous
  object!method() // asynchronous
- Explicit futures
  Fut<T> future = object!method();
  T t = future.get; // blocks

ASP/ProActive
- Transparent asynchronous calls
  object.method() // synchronous or asynchronous
- Transparent first class futures
  T future = object.method();

Single-threaded
- ASP/ProActive
  - Creol
  - JCoBox
  - ABS

Cooperative
  - Creol
  - JCoBox
  - ABS

Multi-threaded
  - Multi-active Objects: MultiASP/new ProActive

Local Concurrency
I. Active Object Programming models

II. Multi-active Objects: Principles

III. Scheduling in Multi-active Objects

IV. A ProActive backend for ABS

V. Conclusion and Future Works
Multi-active objects

- A programming model that mixes local parallelism and distribution with high-level programming constructs
- Execute several requests in parallel but in a *controlled manner* (compatibility notion)

Provided add, add and monitor are *compatible*

Note: monitor is compatible with join
Scheduling Requests

• An « optimal » request policy that « maximizes parallelism »:
  ➔ Schedule a new request as soon as possible (when it is compatible with all the served ones)
  ➔ Serve it in parallel with the others
  ➔ Serves
    ● Either the first request
    ● Or the second if it is compatible with the first one
    ● Or the third one…

Compatibility = requests can execute at the same time and can be re-ordered
Declarative concurrency by annotating methods

```java
@DefineGroups(
    @Group(name="join", selfCompatible=false)
    @Group(name="routing", selfCompatible=true)
)
```

```java
@DefineRules(
    @Compatible({"join", "monitoring"})
    @Compatible({"routing", "monitoring"})
)
```

```java
class Peer {
    ...
    @MemberOf("join")
    public JoinResponse join(Peer other) { ... }
    @MemberOf("routing")
    public void add(Key k, Serializable value) { ... }
    @MemberOf("routing")
    public Serializable lookup(Key k) { ... }
}
```

- **Groups** (Collection of related methods)
- **Rules** (Compatibility relationships between groups)
- **Memberships** (To which group each method belongs)
Hypotheses and programming methodology

• **We trust the programmer**: annotations supposed correct static analysis or dynamic checks should be applied in the future

• Without annotations, a multi-active object runs like an active object

• If more parallelism is required:
  1. Add annotations for non-conflicting methods
  2. Declare dynamic compatibility (depending on runtime values, e.g. request parameters)
  3. Protect some memory access (e.g. by locks) and add new annotations

More parallelism ➔ More complex code / better performance
Agenda

I. Active Object Programming models

II. Multi-active Objects: Principles

III. Scheduling in Multi-active Objects

IV. A ProActive backend for ABS

V. Conclusion and Future Works
Thread Limitation per Multi-active Object

- Too many threads harm:
  - memory consumption,
  - too much concurrency wrt number of cores

```java
@DefineThreadConfig(threadPoolSize=1, hardLimit=true)
public class MyObject {
    ...
}
```

```
x = b.m1()
x.doSmth()  // WBN!!
```

```
y = a.m2()
y.doSmt()
```
Multi-active Object Scheduling - Workflow

1. Receive request
2. Apply compatibilities
3. Apply priorities
4. Apply threading policies
5. Execute request

FILTER

REORDER

FILTER AGAIN
Priority Specification Mechanism

@Group(name="G1", selfCompatible=true)
...
@PriorityOrder(
    @Set(groupNames = {"G1"}),
    @Set(groupNames = {"G2"}),
    @Set(groupNames = {"G5","G4"})
),
@PriorityOrder(
    @Set(groupNames = {"G3"}),
    @Set(groupNames = {"G2"})
)
public class MyObject {
    ...
}
Thread Limitation per Group

```
@Group(name="routing ", minThreads=2, maxThreads=5)
public class MyObject {
    ...
}
```

Threads never used by the **routing group**

Threads never used by other **groups**

Thread pool
Summary of the MAO programming model

Key notions

• Groups of requests
• Compatibility
  − possibly decided dynamically
  − between groups
  − between requests from the same group

Key features

• Global thread limit (soft or hard)
• Upper and lower bounds per group
• Priorities among compatible requests
Agenda

I. Active Object Programming models

II. Multi-active Objects: Principles

III. Scheduling in Multi-active Objects

IV. A ProActive backend for ABS

V. Conclusion and Future Works
Motivation

**ProActive – Multi-active Objects**
Development and deployment of distributed applications ✔

**ABS – Abstract Behavioral Spec. Language**
Modeling of distributed applications ✔
Verification tools ✔
Java Translator ✔
No support for distribution ✗

OBJECTIVE
Provide distributed deployment to ABS using ProActive
## From ABS to ProActive: Challenges

<table>
<thead>
<tr>
<th></th>
<th>ABS</th>
<th>MultiASP/ProActive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active object model</td>
<td>Object group</td>
<td>Non uniform</td>
</tr>
<tr>
<td>Asynchronous method calls and futures model</td>
<td>Explicit</td>
<td>Transparent</td>
</tr>
<tr>
<td>Threading model</td>
<td>Cooperative</td>
<td>Multi-threaded</td>
</tr>
</tbody>
</table>
Towards translation of ABS in ProActive

• Select active objects
  - ABS object group (COG) = ProActive active object
  - Entry point to the local memory space

• Hierarchical indexing of objects

① Global index via the network

| Object URL | @COG1 | @COG2 | ... | ...
|------------|-------|-------|-----|-----
| Object     | cog1  | cog2  | ... | ...

② Local index via shared memory

| Object ID  | ID1  | ID2  | ... | ...
|------------|------|------|-----|-----
| Object ref | o1   | o2   | ... | ...

| Object ID  | ID3  | ID4  | ... | ...
|------------|------|------|-----|-----
| Object ref | o3   | o4   | ... | ...

Translation of a *new cog* statement

**ABS code:**

```
Server server = new cog Server()
```

**Translation in ProActive:**

```
Server server = new Server()
COG cog = newActive(COG.class, {}, node2)
cog.registerObject(server)
```

---

**Diagram:**

- **node1**
  - **mainCog**
  - **server**
  - **cog**
    - **(2) cog (proxy)**

- **node2**
  - **(3) remote server**
  - **(2) cog**

**Relevant notes:**

1. Server creation
2. COG creation
3. Registering the server

**Runs:**

- Server runs
- COG runs
- Remote server run
## From ABS to ProActive: Challenges

<table>
<thead>
<tr>
<th></th>
<th>ABS</th>
<th>MultiASP/ProActive</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Active object model</strong></td>
<td>Object group</td>
<td>Non uniform</td>
</tr>
<tr>
<td><strong>Asynchronous method calls</strong></td>
<td>Explicit</td>
<td>Transparent</td>
</tr>
<tr>
<td>and futures model**</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Threading model</strong></td>
<td>Cooperative</td>
<td>Multi-threaded</td>
</tr>
</tbody>
</table>
Translation of an Asynchronous Method Call

ABS code:

\[
\text{server!start()}
\]

Translation in ProActive:

\[
\text{server.getCog().execute("start", {}, server.getId())}
\]
Asynchronous Method Call with Parameters

ABS code:

server!start(param1, param2)

Translation in ProActive:

server.getCog().execute("start", {param1, param2}, server1.getID())
From ABS to ProActive: Challenges

<table>
<thead>
<tr>
<th></th>
<th>ABS</th>
<th>MultiASP/ProActive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active object model</td>
<td>Object group</td>
<td>Non uniform</td>
</tr>
<tr>
<td>Asynchronous method calls and futures model</td>
<td>Explicit</td>
<td>Transparent</td>
</tr>
<tr>
<td>Threading model</td>
<td>Cooperative</td>
<td>Multi-threaded</td>
</tr>
</tbody>
</table>
Translation of an *await* statement

**ABS code:**

```java
Fut<Bool> readyFut = server!start() (1)
await readyFut? (2)
```

**Translation in ProActive:**

```java
PAFuture.getFutureValue(readyFut) (2)
```

```java
@DefineGroups(
    @Group(name="scheduling", selfCompatible=true)
)
@DefineThreadConfig(threadPoolSize=1, hardLimit=false)
public class COG {
    @MemberOf("scheduling")
    public ABSType execute(...) {
    }
}
```
Translation of a *get* statement

ABS code:

```java
Fut<Bool> readyFut = server!run();
Bool ready = readyFut.get;
```

Translation in ProActive:

```java
this.getCOG().switchHardLimit(true);
Boolean ready = PAFuture.getFutureValue(readyFut);
this.getCOG().switchHardLimit(false);
```

@DefineThreadConfig(threadPoolSize=1, hardLimit=false)
public class COG {
  ...
}

Change temporarily

The default value
Direct Modifications for Distribution

- Serialization
  - Most classes implement now "Serializable"
  - Optimization: "transient" fields

- Deployment
  - **Node specification** added in the ABS language

ABS code:
```java
Server server = new cog "slaves" Server();
```
Experimental evaluation: DNA matching algorithm
The ProActive backend: A Fully Working Tool

• Automated compilation & deployment of ABS programs

• Significant speedup from local programs

• Overhead < 10% from a native ProActive application
Formalization: Correctness of translation

• Operational semantics of MultiASP/ProActive
• Translational semantics: ABS $\rightarrow$ MultiASP/ProActive

• Proven:
  - Translation simulates any ABS execution ✔
  - Translation corresponds to a possible ABS execution

• Restrictions:
  - FIFO request service
  - Causally ordered communications
Conclusion

- Multi-active objects implemented on top of ProActive
  - A programming model for local concurrent and global distributed objects
- Many case studies & benchmarks
  - NAS, Content Addressable Network, GCM components, ProActive backend for ABS
- Visual *post-mortem* debugger of Multi-active Object app.
- Formal proofs (based on semantics)
  - « maximal parallelism »
  - Correctness of ABS translation
- Next steps:
  - Prove stronger properties, ongoing formalisation in Isabelle/HOL
  - Find deadlocks using behavioural types (ongoing PhD)
  - Design a recovery protocol for MAO
Questions?

Related publications:

1. Multi-threaded Active Objects. Ludovic Henrio, Fabrice Huet, and Zsolt István - In COORDINATION 2013.
2. Declarative Scheduling for Active Objects paper. Ludovic Henrio and Justine Rochas - SAC 2014.