Distributed JAVA

Eric Madelaine
INRIA Sophia-Antipolis, Oasis team

• Aims and Principles
• The ProActive library
• Models of behaviours
• Generation of finite (parameterized) models
Distributed JAVA : ProActive

http://www-sop.inria.fr/oasis/ProActive

• Aims:
  Ease the development of distributed applications, with mobility and security features.

• Distributed = Network + many machines
  (Grids, WANs, LANs, P2P, PDAs, ...)

• Library for distributed JAVA active objects
  – Communication:
    Asynchronous remote methods calls
    Non blocking futures (return values)
  – Control:
    Explicit programming of object activities
    Transparent distribution / migration
**ProActive PDC**

- Most of the time, activities and distribution are not known at the beginning, and change over time.
- Seamless implies reuse, smooth and incremental transitions.
**ProActive : model**

- **Active objects**: coarse-grained structuring entities (subsystems)
- Each active object:
  - possibly owns many **passive objects**
  - has exactly one **thread**.
- **No shared** passive objects -- Parameters are passed by **deep-copy**
- **Asynchronous** Communication between active objects
- Future objects and **wait-by-necessity**.
- Full control to **serve** incoming requests
Call between Objects

\[ b \to \text{foo}(x) \]
**ProActive**: Active object

An active object is composed of several objects:

- The object itself (1)
- The body: handles synchronization and the service of requests (2)
- The queue of pending requests (3)
An object created with $A a = new A (\text{obj, 7});$ can be turned into an active and remote object:

- **Instantiation-based:**
  
  $A a = (A)\text{newActive(«A», params, node)};$

  The most general case.

- **Class-based: a static method as a factory**

  To get a non-FIFO behavior:

  ```java
class pA extends A implements RunActive {
    ...
  }
```

- **Object-based:**

  $A a = new A (\text{obj, 7});$

  $\ldots$

  $a = (A)\text{turnActive (a, node)};$
ProActive : Reuse and seamless

- **Polymorphism** between standard and active objects
  - Type compatibility for classes (and not only interfaces)
  - Needed and done for the future objects also
  - Dynamic mechanism (dynamically achieved if needed)

```java
foo (A a) {
    a.g (...);
    v = a.f (...);
    ...
    v.bar (...);
}
```

- **Wait-by-necessity**: inter-object synchronization
  - Systematic, implicit and transparent futures
    Ease the programming of synchronizations, and the reuse of routines
**ProActive** : Reuse and seamless

- **Polymorphism** between standard and active objects
  - Type compatibility for classes (and not only interfaces)
  - Needed and done for the future objects also
  - Dynamic mechanism (dynamically achieved if needed)

```java
foo (A a) {
  a.g();
  v = a.f(...);
  ... v.bar(...);
}
```

- **Wait-by-necessity**: inter-object synchronization
  - Systematic, implicit and transparent futures
    
    Ease the programming of synchronizations, and the reuse of routines

```
O.foo(a) : a.g() and a.f() are « local »
O.foo(p_a): a.g() and a.f() are « remote + Async »
```
**ProActive**: Intra-object synchronization

**Explicit control:**

**Library of service routines:**
- Non-blocking services,...
  
  `serveOldest ();`
  `serveOldest (f);`
- Blocking services, timed, etc.
  
  `serveOldestBl ();`
  `serveOldestTm (ms);`
- Waiting primitives
  
  `waitARequest();`
  etc.

**Implicit (declarative) control:** library classes

```java
class BoundedBuffer extends FixedBuffer
    implements Active
{
    void runActivity (Body myBody)
    {
        while (...)
        {
            if (this.isFull())
                myBody.serveOldest("get");
            else if (this.isEmpty())
                myBody.serveOldest("put");
            else myBody.serveOldest();
            // Non-active wait
            myBody.waitARequest();
        }
    }
}
```

```java
myBody.forbid ("put", "isFull");
```
Example: Dining Philosophers

- Very classical toy example for distributed system analysis:
  
  Both Philosophers and Forks are here implemented as distributed active objects, synchronised by ProActive messages (remote method calls).
public class Philosopher implements Active {

protected int id;
protected int rightForkIndex;
protected int State;
protected Forks Fork[];
public Philosopher (int id, Forks forks[]) {
    this.id = id;
    this.Fork = forks;
    this.State = 0;
    if (id + 1 == 5) rightForkIndex = 0;
    else rightForkIndex = id + 1;
}

/../.
public void runActivity (Body myBody) {
    while (true) {
        switch (State) {
            case 0: think(); break;
            case 1: getForks(); break;
            case 2: eat(); break;
            case 3: putForks(); break;
        }
    }
}

public void getForks() {
    ProActive.waitFor(Fork[rightForkIndex].take());
    ProActive.waitFor(Fork[leftForkIndex].take());
    State=2;
}
..../..
public class Forks implements Active {

    protected int id;
    protected boolean FreeFork;
    protected int State;

    public void ProActive. runActivity(Body myBody) {
        while (true) {
            switch (State) {
                case 0: myBody.getService().serveOldestWithoutBlocking("take"); break;
                case 1: myBody.getService().serveOldestWithoutBlocking("leave"); break;
            }
        }
    
    ..../.
}
Philosophers.java : initialization

// Creates the fork active objects

Fks= new Forks[5];
Params = new Object[1]; // holds the fork ID
for (int n = 0; n < 5; n++) {
    Params[0] = new Integer(n); // parameters are Objects
    try {
        if (url == null)
            Fks[n] = (Forks) newActive ("Fork", Params, null);
        else
            Fks[n] = (Forks) newActive ("Fork", Params, NodeFactory.getNode(url));
    } catch (Exception e) {
        e.printStackTrace();
    }
}

..../

Mastère RSD - TC4  2005/2006  15
Distributed JAVA

Eric Madelaine
INRIA Sophia-Antipolis, Oasis team

• Aims and Principles
• The ProActive library

• Models of behaviours
• Generation of finite (parameterized) models
**Objectives:**

- Behavioural model (Labelled Transition Systems), built in a compositional (structural) manner: One LTS per active object.
- Synchronisation based on ProActive semantics
- Usable for Model-checking => finite / small
Principles (2)

- Define a **behavioural model**: networks of parameterized LTSs
- Implement using:
  - abstraction of source code (slicing, data abstraction),
  - analysis of method call graphs.
- Build **parameterized models**, then **instantiate** to obtain a finite structure.
- Build **compositional models**, use minimisation by bisimulation.
- Use **equivalence-checker** to prove equivalence of a component with its specification, **model-checker** to prove satisfiability of temporal logic formulas.
Communication model

- Active objects communicate through by Remote Method Invocation (requests, responses).

- Each active object:
  - has a Request queue (always accepting incoming requests)
  - has a body specifying its behaviour (local state and computation, service of requests, submission of requests)
  - manages the « wait by necessity » of responses (futures)
Remote requests

- A ag = newActive ("A", [...], VirtualNode)
- V v1 = ag.foo (param);
- V v2 = ag.bar (param);
- ...
- v1.bar(); //Wait-By-Necessity

Wait-By-Necessity is a Dataflow Synchronization
Method Calls: informal modelisation

- method call
  - request arriving in the queue
  - request served (executed and removed)
  - response received
  - response received

Local object

Remote object

!ro.Q_m(f,args)

?Q_m(f,args)

Serv_Q(A)

!lo.R_m(f,val)

?R_m(f,val)

!ro.Q_m(f,args)

?Q_m(f,args)

Serv_Q(A)

!lo.R_m(f,val)

?R_m(f,val)
Example (cont.)

(1) Build the network topology:

Static code analysis for identification of:

- ProActive API primitives
- References to remote objects
- Variables carrying future values

```java
public void runActivity (Body myBody) {
    while (true) {
        switch (State) {
            case 0: think(); break;
            case 1: getForks(); break;
            case 2: eat(); break;
            case 3: putForks(); break;
        }
    }
}

public void getForks() {
    ProActive.waitFor(Fork[rightForkIndex].take());
    ProActive.waitFor(Fork[leftForkIndex].take());
    State=2;
}
```
Example (cont.)

Or better: using parameterized networks and actions:

\[ !F(n).Q\_Take(f\_j) \]
\[ ?R\_Take(f\_j) \]
\[ !F(n).Q\_Drop() \]

\[ !F(n+1).Q\_Take(f\_i) \]
\[ ?R\_Take(f\_j) \]
\[ !F(n+1).Q\_Drop() \]

\[ ?Q\_Drop() \]

\[ ?Q\_Take(f\_i) \]
\[ !P(m).R\_Take(f\_i) \]
Exercice: Draw the (body) Behaviour of a philosopher, using a parameterized LTS

public class Philosopher implements Active {
    protected int id;
    ...
    public void runActivity (Body myBody) {
        while (true) {
            switch (State) {
            case 0: think(); break;
            case 1: getForks(); break;
            case 2: eat(); break;
            case 3: putForks(); break;
            }
        }
        public void getForks() {
            ProActive.waitFor(Fork[rightForkIndex].take());
            ProActive.waitFor(Fork[leftForkIndex].take);
            State=2;
        }
    }
Exercice:  Same exercice for the Fork !
Server Side : models for the queues

• General case :
  – Infinite structure (unbounded queue)
  – In practice the implementation uses bounded data structures
  – Approximation : (small) bounded queues
  – Operations : Add, Remove, Choose (filter on method name and args)

• Optimisation :
  – Most programs filter on method names : partition the queue.
  – Use specific properties to find a bound to the queue length
Example (cont.)

```java
public void ProActive. runActivity(Body myBody){
    while(true){
        switch (State){
            case 0: myBody.getService().serveOldestWithoutBlocking("take");
                break;
            case 1: myBody.getService().serveOldestWithoutBlocking("drop");
                break;
        }
    }
}
```

Fork: A queue for Take requests

Fork: body LTSs
Active object model: Full structure

\[ \text{Proxy} \quad \text{Use}_m(\text{val}) \quad \text{Serve}_m(f,\text{args}) \]

\[ \text{Body} \quad \text{Queue} \]

\[ A1 \quad A2 \]

\[ !A2.Q_m(\text{args}) \quad !A2.Q_m(f,\text{args}) \quad !f.R_m(\text{val}) \]
Verification: Properties

1) Deadlock
   - It is well-known that this system can deadlock. How do the tools express the deadlock property?

   - Trace of actions:
     
     sequence of (visible) transitions of the global system, from the initial state to the deadlock state.

     Decomposition of the actions (and states) on the components.

   - Correction of the philosopher problem:
     
     Left as an exercise.
Next courses

3) Distributed Components
   – Fractive : main concepts
   – Black-box reasoning
   – Deployment, management, transformations

www-sop.inria.fr/oasis/Eric.Madelaine/Teaching