Semantic Formalisms 3: Distributed Applications

Eric Madelaine
eric.madelaine@sophia.inria.fr

INRIA Sophia-Antipolis
Oasis team

- **Formal Methods**
  Operational Semantics:
  CCS, Bisimulations
- **Software Components**
  Fractal: hierarchical components
  Deployment, transformations
  Specification of components
- **Distributed applications**
  Active object and distributed components
  Behaviour models
  “Realistic” Case-study

UNICE – EdStic
Mastère Réseaux et Systèmes Distribués
TC4
3: Models of Distributed Applications

• Active object and distributed components
  – Example: philosophers
• Behaviour models
• “Realistic” Case-study : wifi network
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, clusters, LANs, P2P desktops, 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: Seamless distribution

- 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->foo(x)

Copy
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
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

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

can be turned into an active and remote object:

- **Instantiation-based:**
  \[ A \ a = \text{(A)newActive} (\text{«A», params, node}); \]
  
The most general case.

- **Class-based: a static method as a factory**
  To get a non-FIFO behavior:
  \[
  \text{class pA extends A implements RunActive \{ ... \}}
  \]

- **Object-based:**
  \[ A \ a = \text{new} \ A \ (\text{obj}, \ 7); \]
  \[ \ldots \]
  \[ a = \text{(A)turnActive} (a, \ \text{node}); \]
**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)

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

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

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

Implicit (declarative) control: library classes
  e.g. : myBody.forbid("put", "isFull");

```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.waitForRequest();
        }
    }
```
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();
    }
}
3: Models of Distributed Applications

- Active object and distributed components
  - Example: philosophers
- Generation of finite (parameterized) models
- “Realistic” Case-study: wifi network
Principles (1)

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 \(\Rightarrow\) finite / small

2 views:

- Theoretical: give the semantics of ProActive code
- Tooling: build a model from static analysis of the code.
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)
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)
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:

![Diagram](image)
Exercice: Draw the (body) Behaviour of a philosopher, using a parameterized LTS

```java
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
Asynchronous Membrane
Full model of a composite component

- Producer
- Consumer
- Buffer
- Queue
- LF

Events:
- ?Requests
- !Response
- ?Serve
- start/stop
- !bind/unbind
- !fut.call(M, args)
- ?Serve(M, fut, args)
- !Request
- !start/stop
- !bind/unbind
Verification : Properties

• 1) Deadlock (ex Philosophers)
  – 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.
3: Models of Distributed Applications

• Active object and distributed components
  – Example: philosophers
• Generation of finite (parameterized) models
• “Realistic” Case-study: wifi network
Fractal case-study:  
(FT + Charles Un., Prague)

Public Wifi Network system for an Airport Hotspot
Model generation
Model generation

• Branching minimisation, all upper level events visible

• Instantiation
  – Simplification: 1 single user
  – Abstraction: 3 web pages, 2 tickets, 2 databases

• Sizes
  – global system – 17 visible labels
    • [non-minimised] 2152 states, 6553 transitions
    • [minimised] 57 states, 114 transitions
  – biggest primitive component
    • 5266 states, 27300 transitions
Mastering the complexity

• Smaller representations
  – partial orders, symmetries

• Reduce the number of visible events

• Use advanced verification tools
  – Distributed space generation
  – On-the-fly tools

• Reason at component level
  – Equivalence / Compliance with a specification
Proving Properties

• Deadlock: our initial specification has one.
  – Diagnostic:
    • <initial state>
    • ""loginWithFlyTicketId(IAccess)(0,1,1)"
    • ""loginWithFlyTicketId(ILogin)(0,1,1)"
    • ""loginWithFlyTicketId(IAccess)(0,1,1)"
    • ""CreateToken_req(IFTAuth)(1,1)"
    • ""GetFlyTicketValidity_req(IFTAuth)(1,1)"
    • ""GetFlyTicketValidity_resp(IFTAuth)(1,1)"
    • ""CreateToken_resp(IFTAuth)(1)"
    • <deadlock>
Deadlock explanation

loginWithFlyTicketId(IAccess)(0,1,1)
Deadlock explanation

`loginWithFlyTicketIdd(ILogin)(0,1,1)`
Deadlock explanation
Deadlock explanation

loginWithFlyTicketId(IAccess)(0,1,1)
Deadlock explanation

CreateToken_req(IFTAuth)(1,1)
Deadlock explanation

GetFlyTicketValidity_req(IFTAuth)(1,1)
Deadlock explanation

GetFlyTicketValidity_resp(IFTAuth)(1,1)
Deadlock explanation

CreateToken_resp(IFTAuth)(1)
Deadlock explanation

deadlock
Deadlock Interpretation

- Fractal synchronous implementation, with mono-threaded components.
- Solution with multi-threaded servers: Behaviour analysis becomes much more difficult.
- ProActive solution: request queues and asynchronous computations. Analysis easier, but finite representation of the queues are a problem.
References – previous work

- **pNets model**

- **Hierarchical components**

- **Asynchronous hierarchical components**
References - general

• Fractal

• Abstraction

• Properties patterns
References - general

• CADP model-checker

• WiFi airport Case-Study

• Behavior protocols