Programming and composing safely distributed applications with Active objects

Introduction: programming languages, distributed systems
Active object languages
Software components
Verification

Ludovic Henrio – CNRS/I3S – Scale project (Sophia Antipolis -- France)
ludovic.henrio@cnrs.fr

ENS Lyon– Apr 2018
Introduction (I): why distributed computing?

• Use several computers to handle a task
  - To go faster
  - To handle bigger datasets
  - Because some problems are by nature distributed
    Involve entities from different places
      web, online commerce …
Introduction: Why is it difficult to program (distributed systems)?

Programming requires:
• defining precise algorithms (procedures to follow)
• Planning what can happen when the program runs
• Consider all possible situations

When considering distributed systems, it is even more difficult:
• Some computers/tasks can go faster or slower (depending on varying parameters)
• Some tasks can even fail

Event not expected
Expected event never occurs
How can programming languages help?

A programming language should be:

- **Simple:** write programs easily
  - easy and fast programming; less bugs
    e.g. high-level programming (python)
    powerful synchronisations (algorithmic skeletons)
- **Expressive:** Write complex programs
  - efficient programs; complex applications
    e.g. expressive complex languages (C++)
- **Restrictive:** Avoid error-prone way of programming
  - limit bugs
    e.g. typing (Caml)

All that is of course contradictory

In the Scale team: Java

- There are many programming models
- Here: actors and active objects
My general approach / contributions

- Design efficient and correct distributed systems
  - ProActive middleware

- Find bugs in programs
  - Vercors verification tool suite

- Formalised programming model
  - ASP calculus

- Implementation
  - Correctness & Optimizations

- Verification and tools
  - Generic properties

- Formal Models for Programming and Composing Correct Distributed Systems

- Habilitation thesis (HDR) 2012

- Increase the confidence people have in the languages and programs
- Help my colleagues implement correct (and efficient) middlewares
- Help the programmer write, compose, and run correct and efficient distributed programs
- Using formal methods (theorem prover, model checking)
Agenda

I. Introduction: Programming languages

II. Active Objects languages
   Principles
   Classification
   Focus on ASP

III. Software components

IV. Verification
An actor does one thing at a time. No parallelism inside an actor.
Principles: actor communication

1. ........
2. ..... 
3. ..... 
4. ..... 
5. ..... 

1. ........
2. ..... 
3. ..... 
4. ..... 
5. .....
**Principles: requests and replies: *futures***

1. ....
2. ....
3. Send message to B
4. ....
5. Get the result

I will reply later

Here is the result

Message
Criteria 1: Which Objects are Active?
Have a « thread »?
Can be accessed from any object?
1 – ALL objects are active (uniform model)

- Creol, Rebeca, ...
- Actors do not share memory
- Used in modelling languages / verification tools
- A lot of parallelism

+ Very convenient abstraction
- Scaling might be an issue (non-trivial implementation)
- Data localisation?
2 - Some objects are active, other passive (non-uniform)

- ProActive, Joelle, (Encore)
- No data shared

+ Closed to real implementation, programmer can control granularity, convenient for distribution (RMI)

- Consistency issues, useless copies can be inefficient

\[
\text{result} = \text{beta.foo}(b)
\]
3 – Object groups (COGs / Cobox / …)

- Jcobox, ABS
- All objects are accessible from any objects,
- They receive asynchronous invocations
- All objects in the same Cobox/COG share a single thread
  a single object is active at a time (in a cog)
+ Convenient abstraction, easy reasoning
+ no problem of consistency
- Scaling might be an issue (non-trivial implementation)
- Distribution is difficult (addressing)
Criteria 2: What happens inside an activity?
1 – One thread at a time, non-interrupted

- ProActive, Rebeca, actors
- The handling of a message runs until completion
  + Easy to reason about (no interleaving),
  + The local behaviour is sequential
- Can deadlock if any other form of synchronization (e.g. futures)
2 - Cooperative multithreading

Creol, ABS, Jcobox, Encore

- All requests served at the same time
- But only one thread active at a time
- Explicit release points in the code (await for a future)

+ Control of scheduling and execution
+ More or less deadlock-free if programmed “correctly”
  - Possible interleaving of thread (no data-race but local race-conditions), especially if not programmed “correctly”
  - More difficult to program: less transparency
IDEA: partially remove the single-threaded constraint of the actor model:

- Controlled and local parallelism

- Two directions
  - Parallel service of requests
  - Intra-request parallelism
3 – Local multithreading I: Multi-active objects

- Mixes local parallelism and distribution
- Execute several requests in parallel but in a *controlled manner* (compatibility)
  
  + Efficient local parallelism (less copy)
  + Less deadlocks (blocked threads)
  - Data races possible, if wrong compatibility

The programmer defines compatibility annotations

Provided add, add and monitor are *compatible*
3 – Local multithreading II: Parallel combinators in Encore

- Collection of asynchronous operations and values
- With synchronisation and parallel primitives to compose them (pipeline, combine and compose parallel workflows)
- Some local parallelism restricted to synchronisation aspects (~skeletons); inside a single message handling
  + Add rich synchronization patterns
  + Well integrated with active objects: futures, asynchronous method calls
- Multithreading is more restricted than multiactive objects

```python
class Mtx
...

def computeFastestDiagonal(): Par Mtx
    let mtx = this.getMtx()
    fLU = async luFact(mtx)
    ffChlsk = async choleskyFact(mtx)
    par = (lift fLU) || (lift fChlsk)

    getDiagonalMtx << (par >>= mtvInv)
```
Criteria 3: Operations available on Futures
Futures in actors?

- Originally, in actors, there is no “future”
- Communication is message sending without result
- Sending result is triggered by an explicit callback
  + No deadlock (no synchronization)
  - Inversion of control (programming more difficult, no synchronization)
  - Need to encode the callback
- Often futures are added to actor frameworks (e.g. Akka)
Explicit synchronous futures

- `await`: check + release thread? (cooperative multithreading)
- `get`: access the future value

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

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

• In Encore:
  - Future chaining
  - « ~> » registers a callback anonymous function that will be executed when the future is available

• In ambientTalk and Akka(*)
  No synchronization: a call on a future is an asynchronous invocation / an asynchronous continuation
  + No deadlock (no synchronization)
    Kind of automatic callback
  - Inversion of control (less intuitive, no synchronization)

(*) Synchronous future access also exist in Akka
Implicit futures with data-flow synchronisation

- In ProActive
  - Future created automatically upon asynchronous invocation -- method call on an active object.
  - No future type
  - No explicit future operation
  - Wait-by-necessity

+ Transparent
+ Data-flow synchronization: efficient and less deadlocks
- Difficult to know where futures are
  ➜ to find deadlocks
A word on transparency

Asynchronous method calls and futures transparent in ASP:

- No specific syntax
  - No future type; No active object type; No explicit future operation
- Program looks like it is not distributed
  - Easy to program simple applications
  - Data-flow synchronization: efficient and less deadlocks
- Difficult to reason about if there is a bug
- More difficult to program complex systems

Fut<A> f = o!m() ; await f? ; x=f.get ; x.foo()

A x=o.m(); x.foo()
ASP properties

- The only source of non-determinism is concurrent request sending
  - Implement different future update strategies (transparent futures)
  - Implement effective fault-tolerance protocol
  - Flexibility in the design of static analyses
- Compilation from ABS to MultiASP (cf conclusion)
  - No partial confluence in MultiASP but richer semantics
  - Proved correctness of the translation
II. Active Objects languages
   - Principles
   - Classification
   - Focus on ASP

III. Software components

IV. Verification
What is a component? / Why components?

• Piece of code (+data) encapsulated with well defined interfaces [Szyperski 2002]

• Very interesting for reasoning on programs (and for formal methods) because:
  - components encapsulate isolated code
    ➔ compositional approach (verification, …)
  - interaction (only) through interfaces
    ➔ well identified interaction
    ➔ easy and safe composition

➔ Reasoning and programming is easier and compositional

➔ Less dynamic than objects:
  
  dynamic aspects = reconfiguration / adaptation (constrained)
What are (Fractal/GCM) Components?

Server / input

Business code

Primitive component

Client / output
What are (Fractal/GCM) Components?

Grid Component Model (GCM)
An extension of Fractal for Distributed computing

Primitive components communicate by asynchronous requests on interfaces
Components abstract away distribution and concurrency
Business code: A primitive component is an active object
Composition and interaction: A composite component is a predefined active object
Agenda

I. Introduction: Programming languages

II. Active Objects languages
   Principles
   Classification
   Focus on ASP

III. Software components

IV. Verification
Software component verification in VerCors

Design and static analysis

Verification

true*. R_S2_ComputeChildren ? x:Nat . ‘Scenario_S1.foo’
<('.*Serve_S1.foo.*') {x+1}> true

Integrated environment for verifying and running distributed components. Ludovic Henrio, Oleksandra Kulankhina, Siqi Li, Eric Madelaine. FASE’16
A Workflow example in VerCors

Generated the state-space:

- 4 million states

Checked the correctness of the flow of futures:

<’iQ_runWorkflow’ . (not ’R_Task1.*’)* . ’Q_Task2.*’> true
Underlying model: pNets

- LTS with explicit data handling (value-passing) with 1\textsuperscript{st} order types
- Parallelism and hierarchy using extended synchronization vectors, with parameterized topology.

$pNet$ Node

$Hole =$ process parameter

$S \quad S'$

$SV$

Synchronization Vectors

Action algebra in pLTS:

\[ a(\ ?x \ , \ y+3 \ , \ ?z \ , \ t) \]

pLTS =

\[ \langle S, s_0, \rightarrow \rangle \]

with labels:

\[ \langle \alpha, e_b, (x_j := e_j)_{j \in J} \rangle \]
Conclusion: Some recent results and a few research directions
Deadlock analysis for transparent futures (with Uni Bologna)

- Behavioural types allows detecting deadlock in ABS
- Extension to transparent first-class futures is not trivial
- Because of the **data-flow** nature: an **unbound** number of method behaviours may have to be **unfolded** at the synchronization point
- We exhibit an analysis for transparent futures
  - Harder than for explicit futures
  - Even more useful as deadlocks are more difficult to find manually
From Modelling to Systematic Deployment of Distributed Active Objects

- Systematic translation of cooperative active objects into multi-threaded active objects
  - Instantiation on ABS and ProActive specifically
  - Faithful simulation
- Show the expressiveness of multiactive objects
- Show the differences between active object languages

≠ semantics for future access

**SHALLOW TRANSFORMATION**

- ABS requests → MultiASP requests
- ABS COGs → MultiASP activities
- ABS objects → MultiASP objects + copies
- ABS futures → MultiASP futures

---

Multiactive objects and their applications. L Henrio, J Rochas. Logical Methods in Computer Science, 2017
Futures: Dataflow synchronization ≠ explicit (control-flow) synchronization

- Restriction for equivalence Theorem (The translation simulates all possible ABS executions)
  - A future value cannot be a future (too strong restriction)
  - Not observable in MultiASP
  - Simulation is impossible

- In ABS one can observe the end of a method execution, in ASP one can only observe the availability of some data
A Few Hot Topics

• A new kind of futures (submitted)
  Maintain the data-driven synchronisation of transparent futures while being more explicit (future type)

• Verification of annotations for multi-active objects: for the moment we trust the programmer (with E Lozes – I3S)
  – Ensure absence of data-races… and of other race-conditions?

• Theoretical foundations of pNets (with E Madelaine – INRIA Sophia, R Boulifa -- Eurecom)

• Verification of Phaser Programs (with A Rezine – Univ of Linköping)

• Specification and Testing of VM schedulers [Socc’17] with F Hermenier – Nutanix
Thank you – Some selected publications

- **A Survey of Active Object Languages.** F De Boer, V Serbanescu, R Hähnle, L Henrio, J Rochas, C Din, E Broch Johnsen, M Sirjani, E Khamespanah, K Fernandez-Reyes, A Mingkun Yang. ACM Computing Surveys (CSUR) 2017
- **A Theory of Distributed Objects.** D Caromel, L Henrio - Springer 2004
- **Asynchronous and Deterministic Objects.** D. Caromel, L. Henrio, B. Serpette - POPL 04
- **Multi-threaded Active Objects.** L Henrio, F Huet, and Z István - COORDINATION 2013
- **Multiactive objects and their applications.** L Henrio, J Rochas. Logical Methods in Computer Science, 2017
  - Programming distributed and adaptable autonomous components - the GCM/ProActive framework. F Baude, L Henrio, C Ruz - Software: Practice and Experience–2014
  - Integrated environment for verifying and running distributed components. L Henrio, O Kulankkhina, S Li, E Madelaine. FASE’16
  - A mechanized model for CAN protocols. F. Bongiovanni and L. Henrio. FASE’13
  - Trustable Virtual Machine Scheduling in a Cloud. F Hermenier, L Henrio - SoCC 2017