Orchestration Evolution & Dataflow Concepts:
Introducing Unanticipated Loops Inside a Legacy Workflow

Sébastien Mosser *†  Mireille Blay–Fornarino *†
Johan Montagnat †
mosser@polytech.unice.fr

CNRS, I3S Laboratory, Modalis Team (†)
University of Nice – Sophia Antipolis (*)
http://modalis.i3s.unice.fr

4th Int. Conference on Internet, Web App. & Services
Thursday, May 28th 2009
Venice ~ Mestre, Italy
Agenda


2. ADORE: Activity moDel supPor ting oRche stration Evolution

3. Introducing Dataflow concepts inside ADORE

4. The “Set Evolution” algorithm: \( v \mapsto v^* \)

5. Conclusions & Perspectives
Agenda


2. ADORE: Activity moDel suppOrting oRchestration Evolution

3. Introducing Dataflow concepts inside ADORE

4. The “Set Evolution” algorithm: $v \mapsto v^*$

5. Conclusions & Perspectives
SOA are based on Services

- Specialized (i.e. elementary) code
- Atomic (& stateless) functionalities
  ⇒ written by developers
  ⇒ Implemented as Web Services [JAX-WS]

SOA rely on Business Processes (aka Workflows)

- Value–added assemblies of services calls
- Business–driven, high reactivity, ...
  ⇒ written by business experts
  ⇒ Implemented as Orchestrations [BPEL]
Objectives: Broadcast relevant information to school users

Principles

1. Sources of information can deliver relevant informations
   - e.g. bus schedules, alerts, restaurant menu
2. Providers retrieve and assemble these informations
   - conforming with broadcasting policies
3. Broadcasting devices deliver information to users
   - e.g. public plasma screens, PDA, cellphone

Implementation

- Sources: Web Services implemented by engineer students
- Providers: Orchestration implemented by SOA teachers
- Broadcasters: Javascript code inside kiosks browser
Orchestration Example: InfoProvider

receive(login, group, loc)

tok := Token::Get(login)

!tok

tt := TimeTable::Get(group)

throw BAD_TOKEN

fcast := Weather::Get(loc, tok)

ftt := Filter::Run(login, tt)

r := concatenate(ftt, fcast)

reply(r)
Going back to business experts reality ... Ouch!

```xml
<scope name="TvShows_Feed">
  <sequence name="internal">
    <if name="shouldAdd">
      <condition>count($tvShows/ns0:item) &gt; 0</condition>
      <sequence name="Sequence1">
        <assign name="feed">
          <copy>
            <from variable="tvShows"/>
            <to>$result/ns0:item/ns0:ShowSet</to>
          </copy>
          <copy>
            <from>'ShowSet'</from>
            <to>$result/ns0:item/@kind</to>
          </copy>
        </assign>
        <assign name="increment">
          <copy>
            <from>$globalIndex + 1</from>
            <to variable="globalIndex"/>
          </copy>
        </assign>
      </sequence>
    </if>
    <else>
      <empty name="nop"/>
    </else>
  </if>
</sequence>
</scope>
```
Going back to business experts reality ... Ouch!
Motivations: A *Design–By–Composition* approach

Business Experts cannot handle complete BPEL processes …
- Too development-oriented, too technical, …
- Too complex, …even for developers!

… But they can focus on small parts …
- How to add a new source inside jSeduite?
- How to add a cache mechanism in front of a source?

…And let a framework perform the composition

1. Merging multiple processes fragments [ECSA’08]
2. Introducing loops over datasets [ICIW’09]
   \[ \Rightarrow \text{Auto-magically handling a set of } \textit{groups} \text{ instead of a single one.} \]
Agenda

2. ADORE: Activity moDel supporing oRchestration Evolution
3. Introducing Dataflow concepts inside ADORE
4. The “Set Evolution” algorithm: $v \mapsto v^*$
5. Conclusions & Perspectives
Example: A mathematical SOA

Legacy Infrastructure (small is beautiful)
- A Memory service
  - Bind a mnemonic name to an integer value
- An Adder process
  - Written by our mathematical business expert

adder : String \times String \rightarrow Integer

\begin{align*}
x \times y & \mapsto x_{int} + y_{int}
\end{align*}
Adder(x,y) process: Perform $x_{int} + y_{int}$

1. receive(x,y)
2. $x_{int} := Memory::Get(x)$
3. $y_{int} := Memory::Get(y)$
4. $r := integerAdd(x_{int}, y_{int})$
5. reply(r)
Adder^x*: ∀x ∈ x^*, Adder(x, y)

receive(x^*, y)

yInt := Memory::Get(y)

x := feed(x^*)

xInt := Memory::Get(x)

r := integerAdd(xInt, yInt)

r* := swell(r)

reply(r*)
Summarizing ADORE in just one slide

An abstract reasoning model

- An orchestration $O$ is defined as:
  - A set of activities $\mathcal{A}^*$
  - A partial order $\prec^*$ between those activities

- An activity is defined as:
  - An unique identifier $Id$, a kind $K$ (BPEL subset)
  - A set of inputs variables $Ins$, and an output one $Out$

$$O \equiv (\mathcal{A}^*, \prec^*)$$

\[ \forall a \in \mathcal{A}^*, a = (Id, K, Ins, Out) \]

\[ (a_1 \prec a_2) \in \prec^* \equiv \text{start}(a_2) \Rightarrow \text{end}(a_1) \]
Expressing the Adder process inside ADORE

\[ \text{adder} \equiv (\{a_1, a_{20}, a_{21}, a_3, a_4\}, \{a_1 \prec a_{20}, a_1 \prec a_{21}, a_{20} \prec a_3, \ldots\}) \]

\[ a_1 \equiv (a_1, \text{receive}, \{x, y\}, \emptyset) \]

\[ a_{20} \equiv (a_{20}, \text{invoke}(\text{memory}, \text{get}), \{x\}, \{x_{\text{int}}\}) \]

\[ a_{21} \equiv (a_{21}, \text{invoke}(\text{memory}, \text{get}), \{y\}, \{y_{\text{int}}\}) \]

\[ a_3 \equiv (a_3, \text{assign}(\text{integerAdd}), \{x_{\text{int}}, y_{\text{int}}\}, \{r\}) \]

\[ a_4 \equiv (a_4, \text{reply}, \{r\}, \emptyset) \]
ADORE evolution capabilities

**Behavioral evolution:**

- Expresses *orchestration & evolutions*
  - ⇒ Small fragments, business–driven
- Use a merge algorithm to compose it

**Datasets evolutions:**

- Define (or reuse) a process handling a variable \( \nu \)
  - ⇒ More simple than directly working with sets
- Enhance it to iterate over a dataset \( \nu^* \equiv \{ \nu_1, \ldots, \nu_n \} \)

Merge: \( O \times E^* \rightarrow O \)

Enhance: \( \nu \mapsto \nu^* \)

www.adore-design.org

2. ADORE: Activity model supporting orchestration evolution

3. Introducing dataflow concepts inside ADORE

4. The “Set Evolution” algorithm: \( \nu \mapsto \nu^* \)

5. Conclusions & Perspectives
Dataflow expressiveness inside ADORE

Defined as a set of functions

- **Dataflow**: \( D_F \)
  - Transitivity retrieves activities using a given variable

- **Firsts & Lasts**: \( F, L \)
  - Retrieves activities without any predecessors / successors in \( \prec^* \)

- **Interface**: \( Interface \)
  - Identify interface activities (receive, reply or throw)

- **Core**: \( C \)
  - Built over the previous functions
  - Identify the body of the expected loop.
Illustrating $D_F$

```
receive(x,y)

xInt := Memory::Get(x)

yInt := Memory::Get(y)

r := integerAdd(xInt,yInt)

reply(r)
```
Illustrating $D_F$
Illustrating $D_F$
Illustrating $D_F$

$D_F(A^*, x)$

1. `receive(x, y)`
2. `xInt := Memory::Get(x)`
3. `yInt := Memory::Get(y)`
4. `r := integerAdd(xInt, yInt)`
5. `reply(r)`
Illustrating $\mathcal{F}$ & $\mathcal{L}$

1. receive($x,y$)
2. $xInt := $ Memory:::Get($x$)
3. $yInt := $ Memory:::Get($y$)
4. $r := $ integerAdd($xInt,yInt$)
5. reply($r$)
Illustrating $\mathcal{F}$ & $\mathcal{L}$

$\mathcal{F}(A^*, \prec^*)$

1. $\text{receive}(x,y)$
2. $\text{xInt} := \text{Memory::Get}(x)$
3. $\text{yInt} := \text{Memory::Get}(y)$
4. $r := \text{integerAdd}(\text{xInt}, \text{yInt})$
5. $\text{reply}(r)$
Illustrating $F$ & $L$

$L(A^*, \prec^*)$

```
receive(x, y)

xInt := Memory::Get(x)

yInt := Memory::Get(y)

r := integerAdd(xInt, yInt)

reply(result)
```
Illustrating Interface

receive($x,y$)

$xInt := Memory::Get(x)$

$yInt := Memory::Get(y)$

$r := integerAdd(xInt,yInt)$

$reply(r)$
Illustrating *Interface*

\[ \text{Interface}(A^*, x) \]

```
receive(x, y)

xInt := Memory:::Get(x)

yInt := Memory:::Get(y)

r := integerAdd(xInt, yInt)

reply(r)
```
Illustrating $C$

```
receive(x, y)

xInt := Memory::Get(x)

yInt := Memory::Get(y)

r := integerAdd(xInt, yInt)

reply(r)
```
Illustrating $C$

$D_F(A^*, x)$

```
receive(x, y)

xInt := Memory::Get(x)

yInt := Memory::Get(y)

r := integerAdd(xInt, yInt)

reply(r)
```
Illustrating $C$

```
receive(x,y)

xInt := Memory:::Get(x)

yInt := Memory:::Get(y)

r := integerAdd(xInt,yInt)

reply(r)
```
Illustrating $C$

\[ C(A^*, \prec^*, x) \]

1. **receive**($x, y$)
2. $x\text{Int} := \text{Memory::Get}(x)$
3. $y\text{Int} := \text{Memory::Get}(y)$
4. $r := \text{integerAdd}(x\text{Int}, y\text{Int})$
5. **reply**($r$)
Agenda


2. ADORE: Activity moDel suppOrting oRchestratIon Evolution

3. Introducing Dataflow concepts inside ADORE

4. The “Set Evolution” algorithm: $\nu \mapsto \nu^*$

5. Conclusions & Perspectives
Generic algorithm : $SetEnhance(o, v)$

Handling $v^* \equiv \{v_1, \ldots, v_n\}$ instead of $v$ in $o$

1. Identify the iteration body
   $$\Rightarrow \text{body} \equiv C(Activities(o), Order(o), v)$$

2. Substitute loops input & output variables
   $$\Rightarrow \text{Stickel symbolic substitution } \sigma(a \rightarrow b)$$

3. Add a loop activity
   $$\Rightarrow \text{Contains the core activities}$$
   - And 2 special activities to feed (resp. swell) the loop.

4. Reorder external activities
   $$\Rightarrow \text{To avoid inconsistent orders}$$

Sébastien Mosser
CNRS – I3S / University of Nice

Orchestration Evolution & Dataflow Concepts:
Running $\text{SetEnhance}(o, x)$:

$D_F(\text{Activities}(o), x)$

1. receive($x, y$)
2. $x\text{Int} := \text{Memory:}:\text{Get}(x)$
3. $y\text{Int} := \text{Memory:}:\text{Get}(y)$
4. $r := \text{integerAdd}(x\text{Int}, y\text{Int})$
5. reply($r$)
Running $\text{SetEnhance}(o, x)$:

1. $\text{receive}(x, y)$
2. $x\text{Int} := \text{Memory::Get}(x)$
3. $y\text{Int} := \text{Memory::Get}(y)$
4. $r := \text{integerAdd}(x\text{Int}, y\text{Int})$
5. $\text{reply}(r)$

Loop body
Running $\text{SetEnhance}(o, x)$:

$\sigma(x \rightarrow x^\star), \sigma(r \rightarrow r^\star)$

1. `receive(x, y)`
2. $x\text{Int} := \text{Memory::Get}(x)$
3. $y\text{Int} := \text{Memory::Get}(y)$
4. $r := \text{integerAdd}(x\text{Int}, y\text{Int})$
5. `reply(r)`
Running $\text{SetEnhance}(o, x)$:

1. $\text{receive}(x*, y)$
2. $y\text{Int} := \text{Memory}::\text{Get}(y)$
3. $x := \text{feed}(x*)$
4. $x\text{Int} := \text{Memory}::\text{Get}(x)$
5. $r := \text{integerAdd}(x\text{Int}, y\text{Int})$
6. $r* := \text{swell}(r)$
7. $\text{reply}(r*)$
Running $\text{SetEnhance}(o, x)$: Reordering inconsistent orders

receive($x^*, y$)

$y_{\text{Int}} := \text{Memory}::\text{Get}(y)$

$x := \text{feed}(x^*)$

$x_{\text{Int}} := \text{Memory}::\text{Get}(x)$

$r := \text{integerAdd}(x_{\text{Int}}, y_{\text{Int}})$

$r^* := \text{swell}(r)$

reply($r^*$)
Running $\text{SetEnhance}(o, x)$:

1. $\text{receive}(x^*, y)$
2. $y_{\text{Int}} := \text{Memory}\:::\text{Get}(y)$
3. $x := \text{feed}(x^*)$
4. $x_{\text{Int}} := \text{Memory}\:::\text{Get}(x)$
5. $r := \text{integerAdd}(x_{\text{Int}}, y_{\text{Int}})$
6. $r^* := \text{swell}(r)$
7. $\text{reply}(r^*)$
Implementation: Prolog language

Logical definition $\Rightarrow$ logical language

- An orchestration is defined as facts
- The algorithm is defined as Prolog rules
- It computes a set of actions to enhance a process

?- [seduite, enhance].
?- setEnhance(informationProvider, lectureGroup, ToDo).
  ToDo = [addActivity(l1, loop([l1_feeder, a20, a21, a4, l1_sweller]), [], []),
  delOrder(a1, a20), addOrder(a1, l1),
  sigma(..., ...) | ...]
Idea: $SetEnhance(\mathcal{O}, v)$ as a generic algorithm

- The algorithm builds a Cartesian product
- This choice doesn’t fit particular cases
  $\Rightarrow$ It’s not a bug, it’s a functionality!

Need: Loop “make-up” capability

1. Let the algorithm perform the (nasty) loop introduction task  
   $\Rightarrow$ It’s its job!
2. Customize the Cartesian product to achieve your goal  
   $\Rightarrow$ It’s your job!

jSeduite case study: $\text{group} \rightarrow \text{group}^*$

1. Invoke $SetEnhance(\text{infoProvider}, \text{group})$
2. Reuse a “concatenate instead of cross” ADORE evolution  
   $\Rightarrow$ job’s done.
Agenda


2. ADORE: Activity moDel suppOrting oRchestration Evolution

3. Introducing Dataflow concepts inside ADORE

4. The “Set Evolution” algorithm: $v \mapsto v^*$

5. Conclusions & Perspectives
Summary

Done
- ADORE, a tool to reason on orchestrations
- A set of functions to handle dataflows concepts
- A loop introduction algorithm based on these functions

To do …
- Enhancing the BPEL generation process
- Larger experiments than the jSeduite one, metrics, …
Perspectives

Formal methods & process algebra (e.g. \( \pi \)-calculus)
- Validating built processes
  - Property preservation (e.g. execution order)
  - Model checking (e.g. determinism, dead-lock, user properties)
- Enhancing the conflict detection mechanisms

Composing loops
- Expressing data composition policies
- Composing datasets using *grid–computing* operators
Thanks for your attention!

Any questions?

http://www.jseduite.org
http://www.adore-design.org