Extension of Behaviours with Formal Data Types: Integration and Coordination

Pascal Poizat

Laboratoire de Méthodes Informatiques (LaMI)
UMR 8042 CNRS - University of Évry, GENOPOLE

Invited Lecture, Universities of Málaga and Extremadura
1 Introduction

2 Integration

3 Coordination

4 Conclusions
1. Introduction
2. Integration
3. Coordination
4. Conclusions
1. Introduction

2. Integration

3. Coordination

4. Conclusions
The problem: complex systems

- expressive structuring needed (modules $\leadsto$ objects $\leadsto$ components $\leadsto$ aspects)
- encapsulated datatypes
- behaviours, communication, value-passing
- verification
The problem: complex systems

- expressive structuring needed
  (modules $\leadsto$ objects $\leadsto$ components $\leadsto$ aspects)
- encapsulated datatypes
- behaviours, communication, value-passing
- verification

(Possible) pieces of a solution

- trusted components, ADL: interface, ports, ... concepts
The problem: complex systems

- expressive structuring needed
  (modules $\leadsto$ objects $\leadsto$ components $\leadsto$ aspects)
- encapsulated datatypes
- behaviours, communication, value-passing
- verification

(Possible) pieces of a solution

- trusted components, ADL: interface, ports, ... concepts
- mixed specifications: behaviours + datatypes
The problem: complex systems

- expressive structuring needed
  (modules $\leadsto$ objects $\leadsto$ components $\leadsto$ aspects)
- encapsulated datatypes
- behaviours, communication, value-passing
- verification

(Possible) pieces of a solution

- trusted components, ADL: interface, ports, ... concepts
- mixed specifications: behaviours + datatypes
- formality
The problem: complex systems

- expressive structuring needed
  (modules $\leadsto$ objects $\leadsto$ components $\leadsto$ aspects)
- encapsulated datatypes
- behaviours, communication, value-passing
- verification

Our framework

- formal components with BIDL
- expressive gluing mechanisms
- mixity: behaviours + abstract datatypes = STS
- analysis techniques for STS
LTS in everyday life

Labelled Transition Systems

Usual models for behaviours are LTS \( < S, s_0, A, T > \) with \( s_0 \in S \) and \( T \subseteq S \times A \times S \), often, \( A = A^{\text{in}} \sqcup A^{\text{out}} \) (IOLTS)

Example (Coffee Machine)

Poizat

Extension of Behaviours with Formal Data Types
Labelled Transition Systems

Usual models for behaviours are LTS $< S, s_0, A, T >$ with $s_0 \in S$ and $T \subseteq S \times A \times S$, often, $A = A^{in} \cup A^{out}$ (IOLTS)

Example (Coffee Machine)

Poizat Extension of Behaviours with Formal Data Types
State explosion

In presence of data ...

The computation of an LTS from a specification may explode!

Example (Buffer)
State explosion

In presence of data ...

The computation of an LTS from a specification may explode!

Example (Buffer)

Buffer<> = in ?a: Nat . Buffer<a>
Buffer<b.X> = in ?a: Nat . Buffer<b.X.a>
+ out !b . Buffer<X>
State explosion

In presence of data ...

The computation of an LTS from a specification may explode!

Example (Buffer)

horizontal explosion (encapsulation)

vertical explosion (receptions)
The STS Solution

Symbolic Transition Systems

STS abstract the data on states and transitions

[HL-HandbookPA,STS4LOTOS], [CPR00]

e.g., \(< D, (\Sigma, Ax), S, s_0, v_0, T >\) [MPR04]

with elements of \(T\) of the form:
The STS Solution

Symbolic Transition Systems

STS abstract the data on states and transitions

[HL-HandbookPA,STS4LOTOS], [CPR00]

e.g., \(< D, (\Sigma, Ax), S, s_0, v_0, T >\) [MPR04]

with elements of \(T\) of the form:

\[ S \xrightarrow{\text{guard}(\text{self}, x_1, \ldots, x_n) \text{ event}\?x_1\ldots\?x_n \!t_1\ldots\!t_m /\text{action}(\text{self}, x_1, \ldots, x_n)} \] \( s' \)
Symbolic Transition Systems

STS abstract the data on states and transitions

[HL-HandbookPA,STS4LOTOS], [CPR00]

e.g., \(< D, (\Sigma, Ax), S, s_0, v_0, T >\) [MPR04]

with elements of \(T\) of the form:

\[ S \xrightarrow{\text{[guard}(\text{self},x_1,...,x_n)]} \text{event}\?x_1...?x_n \!t_1...!t_m /\text{action}(\text{self},x_1,...,x_n) \rightarrow S' \]
The STS Solution

Symbolic Transition Systems

STS abstract the data on states and transitions

[HL-HandbookPA,STS4LOTOS], [CPR00]

e.g., \(< D, (\Sigma, Ax), S, s_0, v_0, T >\) [MPR04]

with elements of \(T\) of the form:

\[
S \xrightarrow{\text{[guard\( (self,x_1,\ldots,x_n)\)] event?x_1\ldots?x_n !t_1\ldots!t_m / action\( (self,x_1,\ldots,x_n)\) }} S'
\]
Symbolic Transition Systems

STS abstract the data on states and transitions

[HL-HandbookPA,STS4LOTOS], [CPR00]

\[ e.g., \langle D, (\Sigma, Ax), S, s_0, v_0, T \rangle \] [MPR04]

with elements of \( T \) of the form:

\[ S \xrightarrow{\text{guard}(self, x_1, \ldots, x_n)} \text{event}\_?x_1\ldots?x_n \!t_1\ldots!t_m \text{ /action}(self, x_1, \ldots, x_n) \rightarrow S' \]
The STS Solution

Symbolic Transition Systems

STS abstract the data on states and transitions

[HL-HandbookPA,STS4LOTOS], [CPR00]

*e.g.*, \(< D, (\Sigma, Ax), S, s_0, v_0, T >\) [MPR04]

with elements of \(T\) of the form:

\[
S \xrightarrow{\text{guard}\(x_1, ..., x_n\)}\text{event}\?x_1\ldots\!x_n \text{!}t_1\ldots\!t_m /\text{action}\(x_1, ..., x_n\)} \rightarrow S'
\]
The STS Solution

Symbolic Transition Systems

STS abstract the data on states and transitions

[HL-HandbookPA,STS4LOTOS], [CPR00]

e.g., \( < D, (\Sigma, Ax), S, s_0, v_0, T > \) [MPR04]

with elements of \( T \) of the form:

\[
S \xrightarrow{\text{guard}(x_1,\ldots,x_n) \text{ event?} x_1\ldots?x_n \text{ !}t_1\ldots!t_m} S'
\]
Question

Are BIDL needed anyway?

Yes
Question

Are BIDL needed anyway?

Yes

what does this component?

FOOBAR

result

read
Are BIDL needed anyway?

Yes
it reads (something) and then outputs a result
Question

Are data needed anyway?
Yes
Are data needed anyway?

Yes

What does this component?
Are data needed anyway?

Yes, it reads (at a time) two integers and then outputs the result of the \texttt{div} operation applied to the integers.

```
read ?a:Nat ?b:Nat
FOOBAR
result !div(a,b)
```
Are data needed anyway?

Yes

hidden underlying static type (with usual static signatures)
What does formality bring in?

Some information in [PRS04]:

- abstract, expressive descriptions for BIDL
- animation
- equivalence checking, deadlock freedom, adaptors
Lots of mixed specification languages

<table>
<thead>
<tr>
<th>kind</th>
<th>dynamic</th>
<th>static</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heterogeneous</td>
<td>P. Alg.</td>
<td>model</td>
<td>ObjectZ-CSP, CSP-OZ, ZCCS, ZCSP, TCOZ</td>
</tr>
<tr>
<td></td>
<td>P. Alg.</td>
<td>alg.</td>
<td>LOTOS, PSF</td>
</tr>
<tr>
<td></td>
<td>T/S</td>
<td>model</td>
<td>$\mu$SZ, MaC, Event Calculus</td>
</tr>
<tr>
<td></td>
<td>T/S</td>
<td>alg.</td>
<td>Korrigan, SDL, CASLChart, TAG</td>
</tr>
<tr>
<td></td>
<td>T/S</td>
<td>– spec. –</td>
<td>Estelle, UML, Argos, BDL</td>
</tr>
<tr>
<td></td>
<td>Petri</td>
<td>alg.</td>
<td>OBJSA, Clown, CO-OPN/2</td>
</tr>
<tr>
<td></td>
<td>Petri</td>
<td>– spec. –</td>
<td>CO, OPN</td>
</tr>
<tr>
<td>Homogeneous</td>
<td>Algebraic</td>
<td>LTL</td>
<td>LTL, Rewriting Logic, ASM</td>
</tr>
<tr>
<td></td>
<td>Logical</td>
<td>TLA</td>
<td>TLA, Unity, TRIO, OSL</td>
</tr>
<tr>
<td></td>
<td>Proc. Alg.</td>
<td>CCS+value</td>
<td>CCS+value, CSP, $\pi$-calcul</td>
</tr>
</tbody>
</table>

Poizat

Extension of Behaviours with Formal Data Types
Formal + Semi-Formal

semi-formal
- graph. notations, readability, expressiveness, structuring
  - UML (formal ?)
  - tools, consistency ?
    - ArgoUML, SMW, UMLAut, ...

formal
- abstraction
  - what not how
- semantics
  - tools, verification
- not easy to learn and use

Poizat
Extension of Behaviours with Formal Data Types
Formal + Semi-Formal

semi-formal

+ graph. notations, readability, expressiveness, structuring
  • UML (formal ?)
- tools, consistency ?
  • ArgoUML, SMW, UMLAut, ...

formal

+ abstraction
  • what not how
+ semantics
  • tools, verification
- not easy to learn and use

Poizat  Extension of Behaviours with Formal Data Types
Syntactic Extensions

```
IMPORT Larch-spec Module1
IMPORT Z-spec Module2
IMPORT B-spec Module3
x : Module1.Type1
y : Type2
...
```

<table>
<thead>
<tr>
<th>transition part</th>
<th>interaction kind</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVENT</td>
<td>reception</td>
<td><code>evt-name(x_1 : T_1, \ldots, x_n : T_n)</code></td>
</tr>
<tr>
<td>GUARD</td>
<td>guard guard</td>
<td>predicate</td>
</tr>
<tr>
<td>ACTION</td>
<td>emission</td>
<td><code>receiver ^ evt-name(t_1, \ldots, t_n)</code></td>
</tr>
<tr>
<td>ACTION</td>
<td>assignment</td>
<td><code>x := t</code></td>
</tr>
</tbody>
</table>

Poizat: Extension of Behaviours with Formal Data Types
Typical Use: Case Study

The Gas Station

- furnishes different gas
- three pumps, three tanks
- credit card payment
Typical Use: Analysis

Static part
- booleans ($Z$)
- integers, real numbers (Larch)
- gases, pumps, tanks ($Z$)

Dynamic part
- card manager
- pump manager
- tank manager
Typical Use: Analysis

Static part
- booleans \((Z)\)
- integers, real numbers \((Larch)\)
- gases, pumps, tanks \((Z)\)

Dynamic part
- card manager
- pump manager
- tank manager
Typical Use: Analysis

Static part
- booleans (Z)
- integers, real numbers (Larch)
- gases, pumps, tanks (Z)

Dynamic part
- card manager
- pump manager 3 Extended State Diagrams
- tank manager

Typical Use: Analysis

Static part
- booleans (Z)
- integers, real numbers (Larch)
- gases, pumps, tanks (Z)

Dynamic part
- card manager
- pump manager 3 Extended State Diagrams
- tank manager

Poizat
Extension of Behaviours with Formal Data Types
Poizat

Extension of Behaviours with Formal Data Types
Poizat

Extension of Behaviours with Formal Data Types
Operational Semantics – sos

On operational semantics …

• can be used for Transition Systems and Process Algebras
• well suited for animation and equivalence checking
  
  are the interfaces of C(lient) and S(erver) compatible?

• refinement
  
  does the C implementation do what is required in its interface?

• compositionality
  
  if I prove that C and C’ are compatible, may I replace C with C’ in any system?

• adequacy wrt temporal logic
  
  if C and C’ are equal, may I prove properties on the simplest one?
On operational semantics ...

- can be used for Transition Systems and Process Algebras
- *are the interfaces of C(lient) and S(erver) compatible?*
  \[ \text{Interf}(C) = \text{Interf}(S), \text{with } = \in \{=T, \sim, \approx, \ldots\} \]
- *does the C implementation do what is required in its interface?*
  \[ \text{Interf}(C) \subseteq \text{Interf}(\text{Impl}(C)), \text{with } \subseteq \in \{\subseteq_T, \subseteq_F, \ldots\} \]
- *if I prove that C and C’ are compatible, may I replace C with C’ in any system?*
  \[ C \sim C' \Rightarrow (\forall S[.].S[C] \sim S[C']) \]
- *if C and C’ are equal, may I prove properties on the simplest one?*
  \[ C \sim C' \Leftrightarrow (\forall \phi \in \Phi_{\text{HML}}. C \models \phi \Leftrightarrow C' \models \phi) \]
The [APS03] Semantics

- based on experience with several mixed languages (Korrigan, CCS+ADT, TAG, MaC, ...)
- representative for the definition of a generic approach to integrate static formal specifications (SFS) into dynamic formal specification (DFS)
  - builds on a first proposal for UML state diagrams + SFS + synchronous communication
  - generalizing and asynchronous communication
Principle

Formal rules in 4 layers

- meta-typing
- static evolution
- dynamic evolution and open-systems
- composition
Principle

\[
\{ (\text{INIT}, \text{STATE}, \exists x \text{ TRANS})_i \} \\
\text{(a)} \quad \exists x \quad (\forall i) \\
\{ D_i = (\text{INIT}, \text{STATE}, \text{TRANS})_i \} \\
\text{(1)} \quad \text{dynamic rules (\forall i)} \\
\{ (\text{INIT}, \text{STATE}, \exists x \text{ TRANS})_i \} \\
\text{(b)} \quad \parallel \cdot \parallel x \quad (\forall i) \\
\{ (\text{INIT}, \text{STATE}, \text{TRANS})_i \} \\
\text{(c)} \quad \text{used in (\forall i)} \\
\{ (\text{INIT}^{\text{open}}, \text{STATE}^{\text{open}}, \text{TRANS}^{\text{open}})_i \} \\
\text{(2)} \quad \text{open systems rules (\forall i)} \\
\{ (\text{INIT}, \text{STATE}, \text{TRANS}) \} \\
\text{(3)} \quad \text{global system rules}
\]
Remarks

- lots of dynamic semantics
- use of generic elements, e.g., $\text{event} \in Q_{in}$

Constraints

- $||D||_{sos} = \text{LTS} (\text{INIT}, \text{STATE}, \text{TRANS}) \Rightarrow \text{OK}$ !

Notation

- $\mathcal{D}, D = (\text{INIT}, \text{STATE}, \text{TRANS}, \text{DeclImp}, \text{DeclVar}) \in \mathcal{D}$
- $\text{EVENT} = \text{EVENT}^? \cup \text{EVENT}^!$, $\text{DeclVar} = \text{DeclVar}^? \cup \text{DeclVar}^!$
- $S \subseteq \text{STATE} \times \varepsilon \times Q[\text{EVENT}^?] \times Q[\text{EVENT}^!]$
Remarks

- lots of dynamic semantics
- use of generic elements, e.g., $\text{event} \in Q_{in}$

Constraints

- $||D||_{sos} = \text{LTS } (\text{INIT}, \text{STATE}, \text{TRANS}) \Rightarrow \text{OK}$

Notation

- $D, D = (\text{INIT}, \text{STATE}, \text{TRANS}, \text{DeclImp}, \text{DeclVar}) \in D$
- $\text{EVENT} = \text{EVENT}? \cup \text{EVENT}^!, \text{DeclVar} = \text{DeclVar}? \cup \text{DeclVar}^!$
- $S \subseteq \text{STATE} \times \mathcal{E} \times Q[\text{EVENT}?] \times Q[\text{EVENT}!]$
Remarks

- lots of dynamic semantics
  - use of generic elements, e.g., $event \in Q_{in}$

Constraints

- $||D||_{sos} = LTS (INIT, STATE, TRANS) \Rightarrow \text{OK}$

Notation

- $\mathcal{D}, D = (INIT, STATE, TRANS, DeclImp, DeclVar) \in \mathcal{D}$
- $EVENT = EVENT^? \cup EVENT^!, DeclVar = DeclVar^? \cup DeclVar^!$
- $S \subseteq STATE \times \mathcal{E} \times Q[EVENT^?] \times Q[EVENT^!]$
∀i ∈ 1..n . ∃Xᵢ . tᵢ ::₃ Xᵢ
∃vᵢ . E ⊢ tᵢ ⊢ Xᵢ vᵢ

act−eval(rec^e(t₁, . . . , tₙ), < Γ, E, Q_{in}, Q_{out} >, D) =
< Γ, E, Q_{in}, Q_{out} ⊔ \{ rec^e(v₁, . . . , vₙ) \} >

∃X . t ::₃ X
∃v . E ⊢ t ⊢ X v

act−eval(x := t, < Γ, E, Q_{in}, Q_{out} >, D) =< Γ, E{x ↦ v}, Q_{in}, Q_{out} >
Static Evolution

Term evaluation, $\triangleright_X$

- $\triangleright_{Alg}$: rewriting (+ tools: Larch Prover, ELAN)
- $\triangleright_Z, \triangleright_B$: LTS construction (+ tools: Z-Eves)
- $\triangleright$ : classes formelles, Z
Dynamic Evolution

Notation

\[ \text{EVENT}^{?+} = \text{EVENT}^? \cup \{\varepsilon\} \]

\[ \|D\|_{sos} = \text{LTS}((\text{INIT, STATE, TRANS}) \text{ with:} \]

- \( \text{STATE} \subseteq S \)
- \( \text{INIT} \subseteq \text{STATE} \)
- \( \text{TRANS} \subseteq \text{STATE} \times \text{EVENT}^{?+} \times \text{STATE} \)
Dynamic Evolution

Event [Guard] / Actions

Poizat

Extension of Behaviours with Formal Data Types
Dynamic Evolution

Outline
Introduction
Integration
Coordination
Conclusions

Motivations
Overview
Semantics
Tool

Poizat
Extension of Behaviours with Formal Data Types
Open Systems

Notation

$$||D||_{SOS}^{open} = LTS(INIT^{open}, STATE^{open}, TRANS^{open})$$ with:

- $$INIT^{open} \subseteq INIT$$
- $$TRANS^{open} \subseteq TRANS \times Q[EVENT?] \times Q[EVENT!]$$
- $$STATE^{open} \subseteq SOURCE(TRANS^{open}) \cup TARGET(TRANS^{open})$$
Open Systems

Event? → Event?*, Event!* → Event [ Guard ] / Actions

Poizat
Extension of Behaviours with Formal Data Types
Compositions

Notation

\[ \| \bigcup_{i=1}^{n} D_i \|_{oper}^{open} = \text{LTS}(\overline{\text{INIT}}^{open}(\bigcup_{i=1}^{n} D_i), \overline{\text{STATE}}^{open}(\bigcup_{i=1}^{n} D_i), \overline{\text{TRANS}}^{open}(\bigcup_{i=1}^{n} D_i)) \]

with:

- \( \overline{\text{INIT}}(\bigcup_{i=1}^{n} D_i) \subseteq \prod_i \overline{\text{INIT}}^{open}(D_i) \)
- \( \overline{\text{TRANS}}(\bigcup_{i=1}^{n} D_i) \subseteq \{ t \in \prod_i \overline{\text{TRANS}}^{open}(D_i) | \text{CC}(t) \} \)
- \( \overline{\text{STATE}}(\bigcup_{i=1}^{n} D_i) \subseteq \overline{\text{INIT}}(\bigcup_{i=1}^{n} D_i) \cup \text{TARGET}(\overline{\text{TRANS}}(\bigcup_{i=1}^{n} D_i)) \)
Compositions

Idea

whenever

something addressed to $D_j$

is taken out of a given $D_k$ output queue

then

it is put, at the same time, within the $D_j$ input queue
Formally ...

\[ CC(S_1 \xrightarrow{I_1} E_{in_1}, E_{out_1}, S'_1, \ldots, S_n \xrightarrow{I_n} E_{in_n}, E_{out_n}, S'_n) \iff \]

\[ \forall k \in 1 \ldots n. \forall D_j \hat{e} \in E_{out_k}. D_j \in \bigcup_{i \in 1 \ldots n} D_i \implies e \in E_{in_j} \]
xCLAP - Architecture

- Motivations
- Overview
- Semantics
- Tool

xCLAP - Architecture

- SMW
- Translator: smw2xclap
- State diagrams (textual format)
- Parsing: Spark
- Automaton instances
- Animation
- Class hierarchy
- Data tools

Poizat: Extension of Behaviours with Formal Data Types
xCLAP - Designing

Poizat  Extension of Behaviours with Formal Data Types
xCLAP - Translation

(a) state diagram (graphical format)

(b) state diagram (textual format)

```
D1
/n:=n+x

esdstate
EsdState s1 {'initial':1, 'final':0}
EsdState s2 {'initial':0, 'final':0}

declare D1

import LarchSpec Nat: nat.lp
n: Nat

esdtrans
EsdTrans s1 {'event':"tick(x:nat)", 'guard':"x>succ(0)", 'action':""}.
EsdTrans s2 {'event':"", 'guard':"", 'action':"n:=n+x"}.

IMPORT LarchSpec Nat: nat.lp

translator

smw2xclap

Poizat
Extension of Behaviours with Formal Data Types
xCLAP - Configuration

Poizat

Extension of Behaviours with Formal Data Types
xCLAP - Animation

Poizat  Extension of Behaviours with Formal Data Types
What do we model?

Distributed Entities

- viewed through *interfaces* (black-box foundation)
- interfaces have to take into account *behavioural information* (BIDL)
- goal: quick survey and comparison of *formal material* to describe coordination/interaction among entities
- remember?
  formal means enable one to use existing *verification tools* to ensure correctness of interactions
- applications: web services, genetic regulatory networks
A Simple Formal Model: LTS

- here: simple yet general **formal model** of entities: a nondeterministic LTS $\langle L, S, I, F, T \rangle$
- labels may be emissions $e!$ or receptions $r?$
- **data information** is discarded for simplicity
- running example: one store and several suppliers

---

**How?**

**A Simple Formal Model: LTS**

- here: simple yet general **formal model** of entities: a nondeterministic LTS $\langle L, S, I, F, T \rangle$
- labels may be emissions $e!$ or receptions $r?$
- **data information** is discarded for simplicity
- running example: one store and several suppliers

---

**Store**

- nok?
- buy!
- ok?

**Supplier**

- request?
- refuse!
- accept!

---

Poizat

**Extension of Behaviours with Formal Data Types**

- [YellinEtAl-TOPLAS’97]
- [deAlfaroHenzinger-ESEC’01]
- [ArbabEtAl-FOCLASA’02]
Communication Model

- depends on the means used to compose entities
  - **implicit means**: semantic rules (first part)
  - **explicit means**

*Formal Model*

**Coordination Means**

- **Communication Model**
  - depends on the means used to compose entities
  - *implicit means*: semantic rules (first part)
  - *explicit means*

**Entity**

- synchronized products
- process algebra
- interaction diagrams
- temporal logic
- LTS

*Poizat Extension of Behaviours with Formal Data Types*
Semantics

- basic idea: redefine the CC constraint of part I
Semantics

- basic idea: redefine the CC constraint of part I

\[ CC(S_1 \xrightarrow{l_1} E_{in_1}, E_{out_1}, S'_1, \ldots, S_n \xrightarrow{l_n} E_{in_n}, E_{out_n}, S'_n) \iff \]

\[ \forall k \in 1 \ldots n \ . \ \forall D_j \in E_{out_k} \ . \ D_j \in \bigcup_{i \in 1 \ldots n} D_i \implies e \in E_{in_j} \]
Semantics

- basic idea: redefine the CC constraint of part I

\[ \text{CC}(S_1 \xrightarrow{l_1} S_1', \ldots, S_n \xrightarrow{l_n} S_n', \text{Coord}) \iff \] ???

- see [SP04], [JUCS, 2005, submitted]
- here: examples
Process Algebra

- **parallel composition operators** are way to match inputs and outputs
- may be used as an **explicit 1\textsuperscript{st} class coordinator language** to take into account more complex coordination protocols
Process Algebra

- **parallel composition operators** are a way to match inputs and outputs.
- May be used as an **explicit 1st class coordinator language** to take into account more complex coordination protocols.

Example (with processes)

\[ S = \text{Supplier}[\text{req1/request, ref1/refuse, ..}] \mid \text{Supplier}[\text{req2/request, ref2/refuse, ..}] \mid \text{Store} \mid \text{Coord} \]

\[ \text{Coord} = \text{buy}.(\text{’req1.Wait1} + \text{’req2.Wait2}) \]

\[ \text{Wait1} = \text{acc1.’ok .0} + \text{ref1.’nok.Cord} \]

\[ \text{Wait2} = \text{acc2.’ok.0} + \text{ref2.’nok.Cord} \]
Process Algebra

- **parallel composition operators** are a way to match inputs and outputs
- may be used as an **explicit 1st class coordinator language** to take into account more complex coordination protocols

**Example (with processes)**

\[
S = \text{Supplier}[\text{req1/request, ref1/refuse, ...}] \\
| \text{Supplier}[\text{req2/request, ref2/refuse, ...}] \\
| \text{Store} \\
| \text{Coord}
\]

\[
\text{Coord} = \text{buy.('req1.Wait1 + 'req2.Wait2)} \\
\text{Wait1} = \text{acc1.'ok .0 + ref1.'nok.Coord} \\
\text{Wait2} = \text{acc2.'ok.0 + ref2.'nok.Coord}
\]

[SalaünEtAl-IJBPIIM]
Synchronized Products

- simple and readable means to define interactions among entities [Arnold94,ArnoldEtAl-FI04]
- extended synchronization vectors [SP04]
Synchronized Products

- simple and readable means to define interactions among entities [Arnold94, ArnoldEtAl-FI04]
- extended synchronization vectors [SP04]

- synchronous, one to many: \(<a!,\varepsilon,b?,\varepsilon,c?>\)
- synchronous, matching: \(<a!,\varepsilon,b!,\varepsilon,c!>\)
- synchronous, generation: \(<a?,\varepsilon,b?,\varepsilon,c?>\)
- asynchronous, one to many: \([a!,\varepsilon,b?,\varepsilon,c?]\)
Synchronized Products

- simple and readable means to define interactions among entities [Arnold94, ArnoldEtAl-FI04]
- extended synchronization vectors [SP04]

Example (with vectors)

<buy!, request?, request?>
<nok?, ε, refuse!>
<nok?, refuse!, ε>
<ok?, ε, accept!>
<ok?, accept!, ε>
Interaction Diagrams

- coordination may be described using interaction diagrams: MSC, or UML sequence and collaboration diagrams
- many formalisations proposed so far

[ITU-MSC’96, MauwReniers-MSC’96, KrügerEtAl-SFEDL’02]
Interaction Diagrams

- coordination may be described using interaction diagrams: MSC, or UML sequence and collaboration diagrams
- many formalisations proposed so far

[ITU-MSC’96, MauwReniers-MSC’96, KrügerEtAl-SFEDL’02]
Temporal Logic

- numerous: LTL, CTL/CTL*, ACTL, TLA, \( \mu \)-calculus, ...
- expressive means to coordinate entities, e.g. in formal ADLs [JUCS, 2005, submitted]
  - first, being able to describe the properties of objects that are to be glued (states and transitions)
  - indexed formulas, then lift the properties of the subcomponents of a composition up to the composition
  - the logic also takes into account coordination using logical conjunction

Example (with logic)

\[
\begin{align*}
\text{Store.buy!} & \iff \text{ALL}(\{i : [1..N]\text{Supplier}_i\}).\text{request?} \\
\lor \text{Store.ok?} & \iff \text{ONE}(\{i : [1..N]\text{Supplier}_i\}).\text{accept!} \\
\lor \text{Store.nok?} & \iff \text{ONE}(\{i : [1..N]\text{Supplier}_i\}).\text{refuse!}
\end{align*}
\]
Temporal Logic

- numerous: LTL, CTL/CTL*, ACTL, TLA, $\mu$-calculus, ...
- expressive means to coordinate entities, e.g. in formal ADLs [JUCS, 2005, submitted]
  - first, being able to describe the properties of objects that are to be glued (states and transitions)
  - indexed formulas, then lift the properties of the subcomponents of a composition up to the composition
  - the logic also takes into account coordination using logical conjunction

Example (with logic)

\[
\begin{align*}
\text{Store.buy!} & \iff \text{ALL}({i : [1..N]Supplier_i}).\text{request?} \\
\lor \quad \text{Store.ok?} & \iff \text{ONE}({i : [1..N]Supplier_i}).\text{accept!} \\
\lor \quad \text{Store.nok?} & \iff \text{ONE}({i : [1..N]Supplier_i}).\text{refuse!}
\end{align*}
\]
## A First Comparison

<table>
<thead>
<tr>
<th>Communication Expressiveness</th>
<th>Process Algebras</th>
<th>Vectors</th>
<th>Interaction Diagrams</th>
<th>Logics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 to 1</td>
<td>yes</td>
<td>+</td>
<td>yes</td>
<td>++</td>
</tr>
<tr>
<td>1 to N</td>
<td>yes</td>
<td></td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>1 to M in N</td>
<td>extension</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Name matching | yes | + | yes | + | yes |
| Order | yes | + | extension | yes | |
| Data | yes | + | no | - | yes |

| Tools | animation equivalence checking | + | + | + | + |
| Executability | model–checking | yes | no | no | no |

| Graphical notations | no | - | no | - | yes | ++ | no | -- |

Poizat: Extension of Behaviours with Formal Data Types
Conclusions

Overview
- semantics for STS: operational (here), denotational
- partially tool-equipped: animating (xCLAP), PVS embedding
- semantics for different coordination means

Perspectives
- framework for STS (Eclipse)
- implement coordination means
- better verification means
- relations wrt code / code generation
Conclusions

Overview
- semantics for STS: operational (here), denotational
- partially tool-equipped: animating (xCLAP), PVS embedding
- semantics for different coordination means

Perspectives
- framework for STS (Eclipse)
- implement coordination means
- better verification means
- relations wrt code / code generation

Poizat
Extension of Behaviours with Formal Data Types
Any questions?

Pascal.Poizat@lami.univ-evry.fr
http://www.lami.univ-evry.fr/~poizat


Olivier Maréchal, Pascal Poizat, and Jean-Claude Royer. Checking Asynchronously Communicating Components using Symbolic Transition Systems.
Pascal Poizat, Jean-Claude Royer, and Gwen Salaün. Formal methods for component description, coordination and adaptation (organizer position paper).

In Carlos Canal, Juan Manuel Murillo, and Pascal Poizat, editors, Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT’04), pages 89–100, 2004.

Gwen Salaün and Pascal Poizat. Interacting Extended State Diagrams.