Programming interaction with Types

Marco Carbone (Queen Mary)
Kohei Honda (Queen Mary)
Nobuko Yoshida (Imperial College)

W3C WS-CDL WG London F2F

June 14, 2005

Plan

1. A Recap on the π-Calculus
2. Web Service as a Programming Paradigm
3. Two Calculi for Structured Interaction (1)
4. Two Calculi for Structured Interaction (2)
5. Two Calculi for Structured Interaction (3)
6. Further Topics

1. A Recap on the π-Calculus

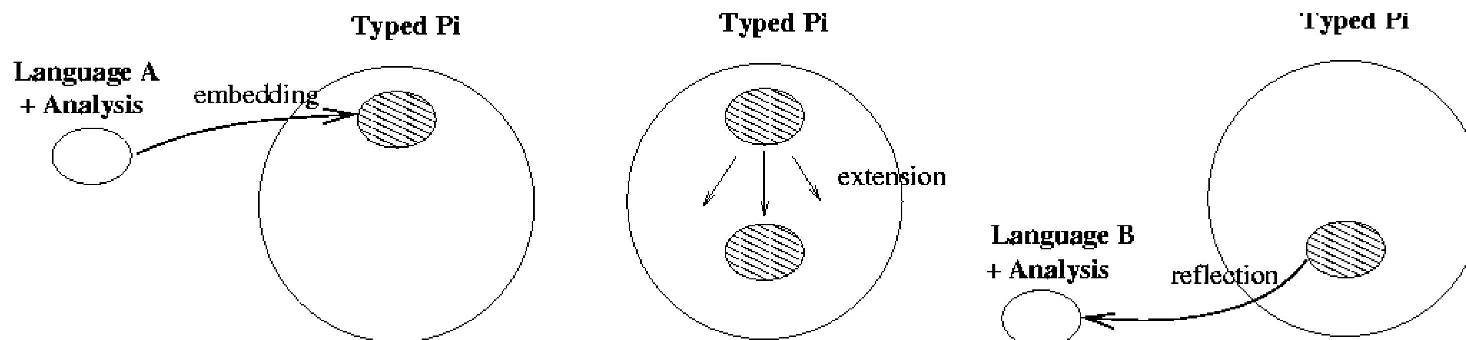
π-Calculus as Foundation (1)

- Offers a precise embedding of programming languages of all paradigms (functional, OOP, parallel, ...);
- Offers a huge design space for diverse paradigms as a stratified mathematical universe.
- Offers rich theories (type disciplines, equivalences, static analysis, program logics, ..) that are reflected from, and onto, concrete languages.

π-Calculus as Foundation (2)

Importance of embeddings:

- Can directly reflect analysis in the π-calculus onto target languages.
- Can transfer ideas/results in an object language directly into the π-calculus.



π-Calculus as Foundation (3)

In the context of CDL, the π-calculus and its theories offer:

- General mathematical framework for representing interaction behaviours.
- A rigorous and fertile basis for diverse tools for software development.
- Broad and precise linkage with existing programming models and infrastructure (cf. Java, .Net, C++, etc.).

Today's talk intends to present a (small) step towards realising this potential.

Essence of CDL

Clarification of the role of the π-calculus in the present enterprise also clarifies the huge potential of CDL in WS technologies.

2. Web Service as a Programming Paradigm.

Why WS? A Recap (1)

WS's infrastructural basis:

- *Naming via URI (hence DNS):* simple scheme, natural correspondence with administrative structures.
- *Routing/messaging via HTTP:* convenient, ubiquitous.
- *Types as XML schemas:* standardised, versatile, extendible.

Further assisted by service standards such as WSDL.

Why WS? A Recap (2)

WS's holy grails *(cf. DS in general and CORBA in particlar)*:

- *Interoperable, integratable:* connect businesses, connect clients, integrate applications.
- *Distributed, shared and robust:* share resources, use existing services, no single point of failure.
- *Evolvable:* long-running applications, incremental updates as a norm, tension with interoperability.

A Coin has Two Sides

The WS has two faces:

1. An advanced form of programming and software development in the web (cf. SOAP).
2. A tool and infrastructure for Business Process Management (cf. BPEL4WS).

Observation:

> *If key software components in businesses all get WS interface, "integration" means integration of WS's.*

That is, two sides do not contradict, they *promote* each other.

WS as a Programming Paradigm (1)

- What does WS mean for programming principles?

  Communication-centric programming in the main stream software development.

- A huge challenge: in fact the current programming community has sparse (if any!) experience in this domain.

WS as a Programming Paradigm (2)

The central features of Web Service:

- Main factor of (end-point) software:
  Communication behaviour.
- Main concern:
  Interactions among end-points proceed properly.
- Other new elements:
  Complex interface (cf. rpc/rmi);
  Open, evolving, intra/inter-organisational;
  Whole concurrency issues (e.g. deadlock, interference,..).

New Principles Needed!

- *How to describe interaction*: elegantly, generally, in a well-structured fashion?
- *Types for interaction:* basic structuring principles for interaction?
- *Specification/verification for interaction:* How can we guarantee e.g. lack of deadlock? Tools based on static analyses, dynamic monitoring, program logics, ...?

Experience in Sequential Programming

Goto (jump) (60's):

(1) Expressive, primitive; (2) Reasoning/analysis is very hard.

Structured Programming (70'–80's):

(1) Expressive; (2) Reasoning/analysis is OK;

(3) Hard to write exceptions clearly and economically.

Structured Programming + **Structured Exceptions** (90's)

(1) Expressive; (2) Reasoning/analysis is OK;

(3) Clear, economical description of exceptions.

Types for Programming (1)

Question:

We have two programs/specifications, P_1 and P_2.
Does $P_1|P_2$ behave all right?

Answer without Types.

...well let's run $P_1|P_2$ and see what happens.

Answer with Types. We put a small tag t_1 to P_1, t_2 to P_2.

Let's check $t_1|t_2$ makes sense. If it does, $P_1|P_2$ behaves well, and has a new tag $t_1|t_2$.

NB. *Checking consistency of tags is very efficient.*

Types for Programming (2)

Types also offer an essential basis for:

- Specification/verification methods (e.g. Hoare Logic).
- Security (e.g. JVM/CLR).

This is because types can articulate a deep structure of language dynamics (Landin, Milner, . . .).

Can we do the same for communication-centric concurrent programming?

3. Two Calculi for Structured Interaction (1)

Technical Elements of CDL

Three bullet points for CDL:

- Interaction scenarios naturally come out globally: *hence CDL.*
- But these scenarios are realised as communication among local behaviour: *hence end-point projection.*
- Inter-organisational nature and subtlety in concurrent programming demand rigorous validation: *hence type checking and other analysis tools*.

Global vs. Local Description (1)

Many buyers, one seller, the same protocol (consider Amazon.com and its customers).

- How do you describe this situation in the π-calculus?
- How do you describe this situation in CDL?

So we conclude ...

Global vs. Local Description (1)

Many buyers, one seller, the same protocol (consider Amazon.com and its customers).

- How do you describe this situation in the π-calculus?
- How do you describe this situation in CDL?

So we conclude ...

We need both!

Global vs. Local Description (2)

But if so, which comes first?

- We often envision interaction globally.
- We often start from existing web services, refine them, increment them, ... to realise a target service.

So we conclude ..

Global vs. Local Description (2)

But if so, which comes first?

- We often envision interaction globally.
- We often start from existing web services, refine them, increment them, ... to realise a target service.

So we conclude ..

Not only do we need two: two forms of descriptions should be inter-related through and through.

Types for Interaction

Drawing on an accumulated study on types for the π-calculus, we single out three basic demands on types for interaction.

- We need to structure interactions: *session types.*
- We need to constrain sharing: *linear types.*
- We need to constrain behaviour: *type disciplines for deadlock/livelock freedom.*

We may further consider stronger behavioural constraints on the top of these type disciplines.

Two Calculi for Interactions (1)

Two typed small languages (calculi):

- A simplification of CDL for global description.
- An applied π-calculus for local description.

Both calculi are strongly typed:

1. A basic typing for well-structured interaction.
2. A refined typing for deadlock-free interaction.
3. A further refinement for livelock-freedom.

Two Calculi for Interactions (2)

And they are related through and through...

- The Global Calculus with session types $\Rightarrow$ The Local Calculus with session types
- The Global Calculus with session types $\Leftarrow$ The Local Calculus with session types

NB. *Two maps are mutually inverse.*

Two Calculi for Interactions (2)

And they are related through and through...

- The Global Calculus with DLF types $\stackrel{?}{\Rightarrow}$ The Local Calculus with DLF types
- The Global Calculus with DLF types $\stackrel{?}{\Leftarrow}$ The Local Calculus with DLF types

NB. *Two maps are mutually inverse.*

Two Calculi for Interactions (2)

And they are related through and through...

- The Global Calculus with LLF types $\stackrel{?}{\Rightarrow}$ The Local Calculus with LLF types
- The Global Calculus with LLF types $\stackrel{?}{\Leftarrow}$ The Local Calculus with LLF types

NB. *Two maps are mutually inverse.*

Two Calculi for Interactions (3)

Engineering implications of two-way embeddings:

- *Write globally*, without any concern on their distributed execution, and with a formal guarantee that its static analysis is reflected on the real behaviour.
- *Refine your code locally*, then obtain its global picture from a vantage point. Or start from the existing service and design global behaviours of interest.
- The two-way embeddings give CDL a *high-bandwidth linkage to and from the π-calculus and its theories*. Behaviours involved in a CDL description can now be rigorously analysed through translation.

Caveat Emptor...

Disclaimer.

1. Restriction by **types** is essential for obtaining two-way embeddability.
2. CDL as of now is *not* (explicitly) based on **session**, though it may be realisable using corelation.
3. **Extensions** to time-out, priotised choices, multi-casting, roll-backs, etc. need be fully examined.

In spite of these reservations, we are optimistic about the general applicability of the presented results to CDL.

Programming vs. description

The calculi we are going to present have two aspects in one: description and programming.

- A high-level descriptive language should enable accurate description of relevant software behaviours at a high level of abstraction — the same thing as programming at a high level of abstraction.
- Descriptive nature of CDL is retained by having non-deterministic sums, by which we expect multiple possible behaviour of interacting parties (can be taken off from its PL instantiation).

4. Two Calculi for Structured Interaction (2)

Global Calculus

- Syntax and examples.
- Reduction (operational semantics)
- Basic ideas of typing:
  1. Session types
  2. Deadlock-free types
  3. Livelock-free types (with termination proofs)

A Brief History

To our knowledge, the first general language for describing global interaction flow is CDL, whose original form was conceived by Nicholas Kavantzas at Oracle and which has been developed by (past and present) members of W3C CDL WG, chaired by Steve Ross-Talbot and Martin Chapman. The CDL calculus in its present form, including formal operational semantics, is due to Marco Carbone, in collaboration with Kohei Honda (a similar calculus without formal semantics was discussed by Kavantzas in 2004: Carbone's key contribution is introduction of distributed local states which enabled formulation of precise operational semantics). The session/deadlock-free typing is based on the ideas of Nobuko Yoshida, and is being developed by her, Carbone and Honda.

A CDL Calculus: Syntax (1)

$\mathsf{A} \longrightarrow \mathsf{B} : [ch(S), \mathsf{op}\langle\vec{e}\rangle, (\vec{y})]$	initial interaction
$\mathsf{A} \longrightarrow \mathsf{B} : [S, \mathsf{op}\langle\vec{e}\rangle, (\vec{y})]$	interaction
$I_1 \; ; \; I_2$	sequencing
$I_1 \mid I_2$	parallel
$I_1 \; + \; I_2$	choice
if $e@\mathsf{A}$ then I_1 else I_2	conditional
while $e@\mathsf{A}$ do I	loop
$\mathbf{0}$	inaction/skip
$x := e \, @ \, \mathsf{A}$	assignment
mutex$\{ \; I \; \} \, @ \, \mathsf{A}$	lock-unlock

A CDL Calculus: Syntax (2)

Comments on syntax (1):

- $\mathsf{A}, \mathsf{B}, \ldots$ are *participants* (or *principals*).
- $ch, ch', \ldots$ are *channels*.
- $e, e', \ldots$ are *expressions* (incl. numeric and boolean constants/functions).
- $S, S', \ldots$ are *session contexts* (communicated initially and used throughout interactions in that session).
- A predicate-driven conditional:

$$\mathsf{when}\ e@\mathsf{A}\ \mathsf{do}\ I \stackrel{\text{def}}{=} \mathsf{while}\ \neg e@\mathsf{A}\ \mathsf{do}\ \mathbf{0}\ ;\ I.$$

A CDL Calculus: Syntax (3)

Comments on syntax (2):

- By introducing assignment and locks, the calculus has a full expressive power as a concurrent language.
- We omit request-reply which a session can express.
- A session context may be realised by a pair of channels with a common co-relation.
- Sequencing/loop can be replaced by prefix/recursion.
- We may add: *local time-out*, *priorities*, *co-relations*, *refined locks*, *multi-casting*, etc.

A CDL Calculus: Example (1)

$\text{Buyer} \longrightarrow \text{Seller} : [s(BS), \text{QuoteReq}\langle prod\rangle, (prod)]$;

$\text{Seller} \longrightarrow \text{Buyer} : [BS, \text{QuoteRes}\langle quote\rangle, (quote)]$;

$\text{Buyer} \longrightarrow \text{Seller} : [BS, \text{QuoteAcc}\langle cred, adr\rangle, (cred, adr)]$;

$\text{Seller} \longrightarrow \text{Shipper} : [p(SP), \text{ShipReq}\langle prod, adr\rangle, (prod, adr)]$;

$\text{Shipper} \longrightarrow \text{Seller} : [SP, \text{ShipConf}]$;

$\text{Seller} \longrightarrow \text{Buyer} : [BS, \text{OrderConf}]$

A CDL Calculus: Example (2)

$\mathsf{Buyer} \longrightarrow \mathsf{Seller} : [s(BS), \mathsf{QuoteReq}\langle prod\rangle, (prod)]$;

$\mathsf{Seller} \longrightarrow \mathsf{Buyer} : [BS, \mathsf{QuoteRes}\langle quote\rangle, (quote)]$;

choice $\{$

$\mathsf{Buyer} \longrightarrow \mathsf{Seller} : [BS, \mathsf{QuoteAcc}\langle cred, adr\rangle, (cred, adr)]$;

$\mathsf{Seller} \longrightarrow \mathsf{Shipper} : [p(SP), \mathsf{ShipReq}\langle prod, adr\rangle, (prod, adr)]$;

$\cdots$(as before)$\cdots$

$+$

$\mathsf{Buyer} \longrightarrow \mathsf{Seller} : [BS, \mathsf{QuoteNoGood}]$

$\}$

A CDL Calculus: Example (3)

$\mathsf{Buyer} \rightarrow \mathsf{Seller} : [s(BS), \mathsf{QuoteReq}\langle prod \rangle, (prod)]$;

$\mathsf{Seller} \rightarrow \mathsf{Buyer} : [BS, \mathsf{QuoteRes}\langle quote \rangle, (quote)]$;

if $(\mathsf{reasonable}\langle quote \rangle)@Buyer$ **then**

$\quad \mathsf{Buyer} \rightarrow \mathsf{Seller} : [BS, \mathsf{QuoteAcc}\langle cred, adr \rangle, (cred, adr)]$;

$\quad \mathsf{Seller} \rightarrow \mathsf{Shipper} : [p(SP), \mathsf{ShipReq}\langle prod, adr \rangle, (prod, adr)]$;

$\quad \ldots$ (as before) $\cdots$

else

$\quad \mathsf{Buyer} \rightarrow \mathsf{Seller} : [BS, \mathsf{QuoteNoGood}]$

end

A CDL Calculus: Example (4)

$\mathsf{Buyer} \longrightarrow \mathsf{Seller} : [s(BS), \mathsf{QuoteReq}\langle prod \rangle, (prod)]$;
$\mathsf{Seller} \longrightarrow \mathsf{Buyer} : [BS, \mathsf{QuoteRes}\langle quote \rangle, (quote)]$;
if $(\mathsf{reasonable}\langle quote \rangle)@Buyer$ **then**
  $\mathsf{Buyer} \longrightarrow \mathsf{Seller} : [BS, \mathsf{QuoteAcc}\langle cred, adr \rangle, (cred, adr)]$;
  $\mathsf{Seller} \longrightarrow \mathsf{CCA} : [a(SC), \mathsf{Check!}\langle cred \rangle, (cred)]$;
  if $(\mathsf{cleared}\langle cred \rangle)@CCA$ **then**
    $\mathsf{CCA} \longrightarrow \mathsf{Seller} : [SC, \mathsf{credOK}]$;
    $\mathsf{Seller} \longrightarrow \mathsf{Shipper} : [p(SP), \mathsf{ShipReq}\langle prod, adr \rangle, (prod, adr)]$;
  else
    $\mathsf{CCA} \longrightarrow \mathsf{Seller} : [SC, \mathsf{credNotOK}]$;
    $\mathsf{Seller} \longrightarrow \mathsf{Buyer} : [BS, \mathsf{OrderRefused}]$
  end
else
  $\mathsf{Buyer} \longrightarrow \mathsf{Seller} : [BS, \mathsf{QuoteNoGood}]$
end

A CDL Calculus: Example (5)

$\mathsf{Buyer} \longrightarrow \mathsf{Seller} : [s(BS), \mathsf{QuoteReq}\langle prod \rangle, (prod)]$;

$\mathsf{Seller} \longrightarrow \mathsf{Buyer} : [BS, \mathsf{QuoteRes}\langle quote \rangle, (quote)]$;

while $(\mathsf{expensive}\langle quote \rangle)@Buyer$ **do**

$\quad \mathsf{Buyer} \longrightarrow \mathsf{Seller} : [BS, \mathsf{QuoteNoGood}]$;

$\quad quote := quote - 10$ @ Seller ;

$\quad \mathsf{Seller} \longrightarrow \mathsf{Buyer} : [BS, \mathsf{QuoteRes}\langle quote \rangle, (quote)]$

end ;

$\mathsf{Buyer} \longrightarrow \mathsf{Seller} : [BS, \mathsf{QuoteAcc}\langle cred,\ adr \rangle, (cred,\ adr)]$;

$\cdots$(as before)$\cdots$

A CDL Calculus: Example (6)

$\mathsf{Buyer} \longrightarrow \mathsf{Seller} : [s(BS), \mathsf{QuoteReq}\langle prod \rangle, (prod)] \; ;$

$\textbf{mutex} \; \{$

$\quad \mathsf{Seller} \longrightarrow \mathsf{Buyer} : [BS, \mathsf{QuoteRes}\langle quote \rangle, (quote)] \; ;$

$\quad \mathsf{Buyer} \longrightarrow \mathsf{Seller} : [BS, \mathsf{QuoteAcc}\langle cred,\ adr \rangle, (cred,\ adr)] \; ;$

$\quad \mathsf{Seller} \longrightarrow \mathsf{Shipper} : [p(SP), \mathsf{ShipReq}\langle prod,\ adr \rangle, (prod,\ adr)] \; ;$

$\quad \mathsf{Shipper} \longrightarrow \mathsf{Seller} : [SP, \mathsf{ShipConf}] \; ;$

$\quad \mathsf{Seller} \longrightarrow \mathsf{Buyer} : [BS, \mathsf{OrderConf}]$

$\} \; @ \; \mathsf{Seller}$

A CDL Calculus: Example (7)

Deadlock!

$\mathsf{Buyer} \longrightarrow \mathsf{Seller} : [s(BS), \mathsf{QuoteReq}\langle prod\rangle, (prod)]$;

mutex $\{$

$\quad \mathsf{Seller} \longrightarrow \mathsf{Buyer} : [BS, \mathsf{QuoteRes}\langle quote\rangle, (quote)]$;

$\quad \mathsf{Buyer} \longrightarrow \mathsf{Seller} : [BS, \mathsf{QuoteAcc}\langle cred, adr\rangle, (cred, adr)]$;

$\quad \mathsf{Seller} \longrightarrow \mathsf{Shipper} : [p(SP), \mathsf{ShipReq}\langle prod, adr\rangle, (prod, adr)]$;

$\quad \mathsf{Shipper} \longrightarrow \mathsf{Seller} : [s(PS), \mathsf{Question!}]$;

$\quad \mathsf{Seller} \longrightarrow \mathsf{Buyer} : [BS, \mathsf{OrderConf}]$

$\}$ @ Seller

A CDL Calculus: Examples (8)

Concurrently compose:

$\mathsf{Buyer} \longrightarrow \mathsf{SellerA} : [s_a, \mathsf{QuoteReq}\langle prod \rangle, (prod)]$;

$\mathsf{SellerA} \longrightarrow \mathsf{Buyer} : [b, \mathsf{QuoteRes}\langle quote, \mathsf{T} \rangle, (quoteA, A\text{-}ok)]$;

and

$\mathsf{Buyer} \longrightarrow \mathsf{SellerB} : [s_b, \mathsf{QuoteReq}\langle prod \rangle, (prod)]$;

$\mathsf{SellerB} \longrightarrow \mathsf{Buyer} : [b, \mathsf{QuoteRes}\langle quote, \mathsf{T} \rangle, (quoteB, B\text{-}ok)]$;

and

when $(A\text{-}ok$ **or** $B\text{-}ok)$ **do**

{ use one of them }

A CDL Calculus: Semantics (1)

The reduction has the form:

$$(I, \sigma) \quad \longrightarrow \quad (I', \sigma')$$

which we read:

> *An interaction I in the initial state σ one-step reduces to the resulting interaction I' and state σ'.*

σ is a collection of *distributed states*: for each participant (principal) A_i, there is its state $\sigma@A_i$ in σ, which maps variables in A_i to their values.

A CDL Calculus: Semantics (2)

As an example, let:

$$I_1 \stackrel{\text{def}}{=} \mathsf{Buyer} \rightarrow \mathsf{Seller} : [s(BS), \mathsf{QuoteReq}\langle prod \rangle, (prod)] \ ;$$

$$I_2 \stackrel{\text{def}}{=} \mathsf{Seller} \rightarrow \mathsf{Buyer} : [BS, \mathsf{QuoteRes}\langle quote \rangle, (quote)]$$

Further let:

$$\sigma \stackrel{\text{def}}{=} [\mathsf{prod} : \text{``}book'', \mathsf{quote} : ?]@\mathsf{Buyer}, \quad [\mathsf{prod} : \text{``}book'', \mathsf{quote} : 30]@\mathsf{Seller}$$

Then we have:

$$(I_1; I_2, \ \sigma) \longrightarrow (\mathsf{Seller} \rightarrow \mathsf{Buyer} : [BS, \mathsf{QuoteRes}\langle quote \rangle, (quote)], \ \sigma)$$

$$\longrightarrow (\mathbf{0}, \quad \sigma[\mathsf{quote} := 30@\mathsf{Buyer}]$$

A CDL Calculus: Semantics (3)

As another example, let:

$$\sigma \stackrel{\text{def}}{=} [\mathsf{quote} : 30]@\mathsf{Buyer}, [\mathsf{quote} : 20]@\mathsf{Seller}$$

Then we have:

$$(\mathsf{if}\ (\mathsf{quote} \leq 20)@\mathsf{Buyer}\ \mathsf{then}\ I_1\ \mathsf{else}\ I_2,\ \sigma) \longrightarrow (I_2,\ \sigma)$$

NB: *The quote value looked up is that of Buyer.*

A CDL Calculus: Semantics (4)

An example of mutex. Let:

$$I_i \stackrel{\text{def}}{=} \mathsf{A_i} \longrightarrow \mathsf{B} : [b(S_i), \mathsf{dec}]$$

$$J_i \stackrel{\text{def}}{=} \textbf{mutex}\, x := x - 1 \ @\ B \ ;\ \mathsf{B} \longrightarrow \mathsf{A_i} : [S_i, \mathsf{ack}] \ @\ B$$

Then we have:

$$\begin{array}{rcl} (I_1;J_1|I_2;J_2,\ [x:30]@B) & \longrightarrow & (J_1|I_2;J_2,\ [x:30]@B) \\ & \longrightarrow & (J_1|J_2,\ [x:30]@B) \\ & \longrightarrow^2 & (J_2,\ [x:29]@B) \\ & \longrightarrow^2 & (\mathbf{0},\ [x:28]@B) \end{array}$$

NB: *J_1 and J_2 do not interleave thanks to mutex.*

A CDL Calculus: Semantics (5)

As an example of a choice, let:

$$\begin{array}{rcl} I_1 & \stackrel{\text{def}}{=} & \mathsf{B} \longrightarrow \mathsf{S} : [S, \mathsf{OK}\langle cred \rangle, (cred)]; \\ I_2 & \stackrel{\text{def}}{=} & \mathsf{S} \longrightarrow \mathsf{B} : [S, \mathsf{Conf}] \\ J & \stackrel{\text{def}}{=} & \mathsf{B} \longrightarrow \mathsf{S} : [S, \mathsf{NotOK}] \end{array}$$

Then we have, with σ to be any state, either:

$$((I_1; I_2) + J,\ \sigma) \longrightarrow (I_1; I_2,\ \sigma) \longrightarrow (I_2,\ \sigma) \longrightarrow (\mathbf{0},\ \sigma)$$

or

$$((I_1; I_2) + J,\ \sigma) \longrightarrow (J,\ \sigma) \longrightarrow (\mathbf{0},\ \sigma)$$

Connectedness (1)

A fundamental principle for the global calculus, originally observed by Steve Ross-Talbot in the context of CDL:

Causality in actions should always be local.

For example,

$$\mathsf{B} \to \mathsf{S} : [s(S), \mathsf{ReqQuote}] \ ; \ \mathsf{S} \to \mathsf{B} : [S, \mathsf{ResQuote}]$$

is connected, but

$$\mathsf{B} \to \mathsf{S} : [s(S), \mathsf{ReqQuote}] \ ; \ \mathsf{P} \to \mathsf{C} : [S, \mathsf{CreditCheck}]$$

is not.

Connectedness (2)

More examples: with

$$I_1 \stackrel{\text{def}}{=} \mathsf{S} \longrightarrow \mathsf{B} : [b, \mathsf{Quote}] \qquad I_4 \stackrel{\text{def}}{=} \mathsf{P} \longrightarrow \mathsf{E} : [e, \mathsf{Deliver}]$$

$$I_2 \stackrel{\text{def}}{=} \mathsf{B} \longrightarrow \mathsf{S} : [s, \mathsf{Conf}] \qquad I_5 \stackrel{\text{def}}{=} \mathsf{E} \longrightarrow \mathsf{B} : [b, \mathsf{HereItIs}]$$

$$I_3 \stackrel{\text{def}}{=} \mathsf{S} \longrightarrow \mathsf{P} : [p, \mathsf{ReqShip}]$$

Then:

- $I_1;I_2$, $I_2;I_3$, $I_3;I_4$ and $I_4;I_5$ are all connected.
- $I_1;I_4$ is not connected. Neither is $I_1;I_5$, since I_1 involves S and B, whereas I_5 is (actively) initiated only by E.

Connectedness (3)

An example of a loop:

$$I' \stackrel{\text{def}}{=} \mathsf{while}\ e@A\ \mathsf{do}\ I$$

$$I \stackrel{\text{def}}{=} \mathsf{A} \longrightarrow \mathsf{B} : [b, \mathsf{run}]\ ;\ \mathsf{B} \longrightarrow \mathsf{C} : [c, \mathsf{run}].$$

I' starts from evaluation by A, does communication from A to B, followed by another from B to C. By unfolding:

$$I' \equiv \mathsf{if}\ \ e@\mathsf{A}\ \ \mathsf{then}\ \ \underline{I\,;I'}\ \ \mathsf{else}\ \ \mathbf{0}$$

Focussing on the underlined part, we see I ends with an action by B and C while I' starts from an action by A, so that I' as a whole is not connected.

Connectedness (4)

Definition. I is *connected* iff:

1. In each $I_1;I_2$, we have $\mathsf{final}(I_1) \cap \mathsf{init}(I_2) \neq \emptyset$.
2. In each $\mathsf{if}\ e@A\ \mathsf{then}\ I_1\ \mathsf{else}\ I_2$, we have $A \in \mathsf{init}(I_1) \cap \mathsf{init}(I_2)$.
3. In each $\mathsf{while}\ e@A\ \mathsf{do}\ I'$, we have $A \in \mathsf{init}(I') \cap \mathsf{final}(I')$.

where $\mathsf{init}(I)$ (resp. $\mathsf{final}(I)$) is the set of initial active (resp. final) participants of I, both defined in the next slide.

Connectedness (5)

- $\mathsf{init}(I)$ is the set of participants involved in any initiating action of I except as a receiver.
- $\mathsf{final}(I)$ is the set of participants involved in any finalising action of I.
- Special cases:

$$\mathsf{init}(\mathsf{if}\ e@A\ \mathsf{then}\ I_1\ \mathsf{else}\ I_2)\ =\ \{A\}.$$

$$\mathsf{final}(\mathsf{if}\ e@A\ \mathsf{then}\ I_1\ \mathsf{else}\ I_2)\ =\ \mathsf{final}(I_1) \cap \mathsf{final}(I_2)$$

$$\mathsf{init}(\mathsf{while}\ e@A\ \mathsf{do}\ I)\ =\ \mathsf{final}(\mathsf{while}\ e@A\ \mathsf{do}\ I) = \{A\}.$$

Connectedness (6)

Proposition.
If I is connected and $(I, \sigma) \longrightarrow (I', \sigma')$, then I' is also connected.

NB. This holds intuitively because connectedness is a postfix-closed property.

Session Types (1)

Types for the simple Buyer-Seller:

$$s\,@\,\mathbf{Buyer}: \quad \uparrow \mathsf{QuoteReq}(\mathit{qreq.xsd})\ ;$$
$$\downarrow \mathsf{QuoteRes}(\mathit{quote.xsd})\ ;$$
$$\uparrow \mathsf{QuoteAcc}(\mathit{address.xsd})$$

$$s\,@\,\mathbf{Seller}: \quad \downarrow \mathsf{QuoteReq}(\mathit{areq.xsd})\ ;$$
$$\uparrow \mathsf{QuoteRes}(\mathit{quote.xsd})\ ;$$
$$\downarrow \mathsf{QuoteAcc}(\mathit{address.xsd})$$

Session Types (2)

Types for the Buyer-Seller with choice/conditional.

$$s\,@\,\textbf{Buyer}: \quad \uparrow\mathsf{QuoteReq}(\mathit{qreq.xsd})\ ;$$
$$\downarrow\mathsf{QuoteRes}(\mathit{quote.xsd})\ ;$$
$$\uparrow\mathsf{QuoteAcc}(\mathit{address.xsd}) \oplus \uparrow\mathsf{QuoteRef}(\,)$$

$$s\,@\,\textbf{Seller}: \quad \downarrow\mathsf{QuoteReq}(\mathit{qreq.xsd})\ ;$$
$$\uparrow\mathsf{QuoteRes}(\mathit{quote.xsd})\ ;$$
$$\downarrow\mathsf{QuoteAcc}(\mathit{address.xsd})\ \&\ \downarrow\mathsf{QuoteRef}(\,)$$

Session Types (3)

Types for the Buyer-Seller with loop.

$$\mathbf{s}\,@\,\mathbf{Buyer}: \quad \uparrow\mathsf{QReq}(\mathit{qreq.xsd})\ ;\ \mu X.\ \downarrow\mathsf{QuoteRes}(\mathit{quote.xsd})\ ;\ \begin{array}{l} \uparrow\mathsf{QAcc}(\mathit{address.xsd}) \ \oplus \\ \uparrow\mathsf{QAnother}(\mathit{qreq.xsd})\ ;\ X \ \oplus \\ \uparrow\mathsf{QRefuse}(\,) \end{array}$$

$$\mathbf{s}\,@\,\mathbf{Seller}: \quad \downarrow\mathsf{QReq}(\mathit{qreq.xsd})\ ;\ \mu X.\ \uparrow\mathsf{QuoteRes}(\mathit{quote.xsd})\ ;\ \begin{array}{l} \downarrow\mathsf{QAcc}(\mathit{address.xsd}) \ \& \\ \downarrow\mathsf{QAnother}(\mathit{qreq.xsd})\ ;\ X \ \& \\ \downarrow\mathsf{QRefuse}(\,) \end{array}$$

Types for Deadlock-Freedom (1)

Types for Seller with mutex (no deadlock).

$$
\begin{array}{ll}
\textbf{Seller}: & s(BS) :\downarrow \mathsf{QuoteReq}(\mathit{qreq.xsd}) \; ; \\
& \textbf{mutex} \; \langle \\
& \quad BS :\uparrow \mathsf{QuoteRes}(\mathit{quote.xsd}) \; ; \\
& \quad BS :\downarrow \mathsf{QuoteAcc}(\mathit{address.xsd}) \; ; \\
& \quad p(SP) :\uparrow \mathsf{ShipReq}(\mathit{shipping.xsd}) \; ; \\
& \quad SP :\downarrow \mathsf{ShipConf}(\,) \; ; \\
& \quad BS :\uparrow \mathsf{OrderConf}(\,) \\
& \rangle \; @ \; \mathsf{Seller}
\end{array}
$$

***NB:** All consecutive actions are included in the type.*

Types for Deadlock-Freedom (2)

Types for Seller with mutex (with deadlock).

$$
\begin{array}{ll}
\textbf{Seller}: & s(BS) : \downarrow \mathsf{QuoteReq}(\mathit{qreq.xsd})\ ; \\
& \textbf{mutex}\ \langle \\
& \quad BS : \uparrow \mathsf{QuoteRes}(\mathit{quote.xsd})\ ; \\
& \quad BS : \downarrow \mathsf{QuoteAcc}(\mathit{address.xsd})\ ; \\
& \quad p(SP) : \downarrow \mathsf{ShipReq}(\mathit{shipping.xsd})\ ; \\
& \quad s(PS) : \downarrow \mathsf{Question!}(\ )\ ; \\
& \ldots \\
& \rangle\ @\ \mathsf{Seller}
\end{array}
$$

NB: *Circularity occurs because s is Seller's channel.*

Types for Livelock Freedom

To prevent livelock, we verify lack of behavioural circularity on the basis of deadlock freedom. The lack of circularity can be checked statically in many interesting cases.

Tips for Session-based Description (1)

- In session-based communication, crucial information is distinction between session-initiating communication and ordinary ones.
- This information leads to effective detection of deadlock/livelock in interactions.
- Usually an interaction scenario naturally divides into distinct sessions (e.g. travel agents, brokerage, . . .).

Tips for Session-based Description (2)

In WS, a session may be realised by:

- A pair of channels of conversants (e.g. Buyer's channel and Seller's channel in a relationship), and:
- A session co-relation freshly generated at each instance of a session.

How does this translate to CDL syntax?

Tips for Session-based Description (3)

A possible machinery to realise session in CDL (we do not consider session passing):

- Declare a special session co-relation (freshly generated at each run) with an associated pair of channels.
- Each Interaction belonging to that session (which should use associated channels) should mention this co-relation.
- Mark the initiating Interaction(s) of each session.

Adding explicit support for session greatly expands the potential for static analyses on CDL.

5. Two Calculi for Structured Interaction (3)

End-Point Calculus: Syntax (1)

Syntax:

$$
\begin{array}{rcl}
P & ::= & x(w) \triangleright [\&_i\, l_i(\vec{y}_i).P_i] \;\mid\; \overline{x}(w) \triangleleft l\langle\vec{y}\rangle.P \;\mid \\
 & & x \triangleright [\&_i\, l_i(\vec{y}_i).P_i] \;\mid\; \overline{x} \triangleleft l\langle\vec{y}\rangle.P \;\mid \\
 & & \mathbf{mBegin}.P \;\mid\; \mathbf{mEnd}.P \;\mid \\
 & & (\nu\, x)P \;\mid\; P|Q \;\mid\; P \oplus Q \;\mid\; !P \;\mid\; \mathbf{0} \\
N & ::= & A[P]_\sigma \;\mid\; (\nu\, x)N \;\mid\; N_1 \| N_2 \;\mid\; \mathbf{0}
\end{array}
$$

End-Point Calculus: Syntax (2)

Comments on syntax.

- $P, Q, \ldots$ are processes, $A, B, \ldots$ are participant names, $A[P]_\sigma$ is a participant (where σ is a local store of A, a finite map from variables to their values), and $N, \ldots$ denote interacting participants.
- $x(w) \triangleright [\&_i\ l_i(\vec{y}_i).P_i]$ and $\overline{x}(w) \triangleleft l\langle\vec{y}\rangle.P$ are for initial interaction (which establish session). In input, (w) binds, while $(\vec{y})$ do not. In examples we often ajoin (w) with $(\vec{y}_i)$ for brevity.
- $x \triangleright [\&_i\ l_i(\vec{y}_i).P_i]$ and $\overline{x} \triangleleft l\langle\vec{y}\rangle.P$ are for communication inside a session. Again in input $(\vec{y}_i)$ do not bind.

End-Point Calculus: Semantics

Dynamics of the end-point calculus is standard except for an update of local states as a result of communication.

$$\mathbf{A}[x(w) \triangleright [\&_i\ l_i(\vec{y}_i).P_i] | R]_\sigma \parallel \mathbf{B}[\overline{x}(w) \triangleleft l_i \langle \vec{v} \rangle . Q | R']_{\sigma'}$$
$$\longrightarrow (\nu\, w)(\mathbf{A}[P_i | R]_{\sigma[\vec{y}_i := \vec{v}]} \parallel \mathbf{B}[Q | R']_{\sigma'})$$

$$\mathbf{A}[x \triangleright [\&_i\ l_i(\vec{y}_i).P_i] | R]_\sigma \parallel \mathbf{B}[\overline{x} \triangleleft l_i \langle \vec{v} \rangle . Q | R']_{\sigma'}$$
$$\longrightarrow \mathbf{A}[P_i | R]_{\sigma[\vec{y}_i := \vec{v}]} \parallel \mathbf{B}[Q | R']_{\sigma'}$$

We also use the standard structural equality $\equiv$.

End-Point Calculus: Semantics

Dynamics of the end-point calculus is standard except an update of local states as a result of communication.

$$\mathbf{A}[x(w) \triangleright [\&_i\, l_i(\vec{y}_i).P_i] | R]_\sigma \parallel \mathbf{B}[\overline{x}(w) \triangleleft l_i\langle\vec{v}\rangle.Q | R']_{\sigma'}$$
$$\longrightarrow (\nu\, w)(\mathbf{A}[P_i | R]_{\sigma[\vec{y}_i := \vec{v}]} \parallel \mathbf{B}[Q | R']_{\sigma'})$$

$$\mathbf{A}[x \triangleright [\&_i\, l_i(\vec{y}_i).P_i] | R]_\sigma \parallel \mathbf{B}[\overline{x} \triangleleft l_i\langle\vec{v}\rangle.Q | R']_{\sigma'}$$
$$\longrightarrow \mathbf{A}[P_i | R]_{\sigma[\vec{y}_i := \vec{v}]} \parallel \mathbf{B}[Q | R']_{\sigma'}$$

We also use the standard structural equality $\equiv$.

End-Point Calculus: Semantics

Dynamics of the end-point calculus is standard except an update of local states as a result of communication.

$$\mathbf{A}[x(w) \triangleright [\&_i\, l_i(\vec{y}_i).P_i] | R]_\sigma \parallel \mathbf{B}[\overline{x}(w) \triangleleft l_i\langle\vec{v}\rangle.Q | R']_{\sigma'} \longrightarrow (\nu\, w)(\mathbf{A}[P_i | R]_{\sigma[\vec{y}_i := \vec{v}]} \parallel \mathbf{B}[Q | R']_{\sigma'})$$

$$\mathbf{A}[x \triangleright [\&_i\, l_i(\vec{y}_i).P_i] | R]_\sigma \parallel \mathbf{B}[\overline{x} \triangleleft l_i\langle\vec{v}\rangle.Q | R']_{\sigma'} \longrightarrow \mathbf{A}[P_i | R]_{\sigma[\vec{y}_i := \vec{v}]} \parallel \mathbf{B}[Q | R']_{\sigma'}$$

We also use the standard structural equality $\equiv$.

End-Point Calculus: Semantics

Dynamics of the end-point calculus is standard except an update of local states as a result of communication.

$$\mathbf{A}[x(w) \triangleright [\&_i\, l_i(\vec{y}_i).P_i] | R]_\sigma \parallel \mathbf{B}[\overline{x}(w) \triangleleft l_i\langle \vec{v} \rangle.Q | R']_{\sigma'}$$
$$\longrightarrow (\nu\, w)(\mathbf{A}[P_i | R]_{\sigma[\vec{y}_i := \vec{v}]} \parallel \mathbf{B}[Q | R']_{\sigma'})$$

$$\mathbf{A}[x \triangleright [\&_i\, l_i(\vec{y}_i).P_i] | R]_\sigma \parallel \mathbf{B}[\overline{x} \triangleleft l_i\langle \vec{v} \rangle.Q | R']_{\sigma'}$$
$$\longrightarrow \mathbf{A}[P_i | R]_{\sigma[\vec{y}_i := \vec{v}]} \parallel \mathbf{B}[Q | R']_{\sigma'}$$

We also use the standard structural equality $\equiv$.

End-Point Calculus: Semantics

Dynamics of the end-point calculus is standard except an update of local states as a result of communication.

$$\mathbf{A}[x(w) \triangleright [\&_i\, l_i(\vec{y}_i).P_i] | R]_\sigma \parallel \mathbf{B}[\overline{x}(w) \triangleleft l_i \langle \vec{v} \rangle .Q | R']_{\sigma'}$$
$$\longrightarrow (\nu\, w)(\mathbf{A}[P_i | R]_{\sigma[\vec{y}_i := \vec{v}]} \parallel \mathbf{B}[Q | R']_{\sigma'})$$

$$\mathbf{A}[x \triangleright [\&_i\, l_i(\vec{y}_i).P_i] | R]_\sigma \parallel \mathbf{B}[\overline{x} \triangleleft l_i \langle \vec{v} \rangle .Q | R']_{\sigma'}$$
$$\longrightarrow \mathbf{A}[P_i | R]_{\sigma[\vec{y}_i := \vec{v}]} \parallel \mathbf{B}[Q | R']_{\sigma'}$$

We also use the standard structural equality $\equiv$.

Process Representation: Buyer (1)

A concrete Buyer.

$\mathsf{Buyer}[$

$(\nu\, c)\overline{s} \triangleleft \mathtt{qReq}\langle c, prod\rangle.c \triangleright \mathtt{qRes}(x, quote).$

if $quote < 1000$ then

$\quad \overline{c} \triangleleft \mathtt{qAccept}\langle cred, adr\rangle.adr \triangleright \mathtt{receive}(y).P$

else

$\quad \overline{c} \triangleleft \mathtt{qRefuse}\langle\rangle$

end $]$

Process Representation: Buyer (2)

Generic Buyer.

$$
\begin{array}{l}
\mathsf{Buyer}[\\
(\nu\, c)\overline{s} \triangleleft \mathtt{qReq}\langle c, prod\rangle . c \triangleright \mathtt{qRes}(x, quote). \\
\{ \\
\qquad \overline{c} \triangleleft \mathtt{qAccept}\langle cred, adr\rangle . adr \triangleright \mathtt{receive}(y).P \\
\qquad\qquad \bigoplus \\
\qquad \overline{c} \triangleleft \mathtt{qRefuse}\langle\rangle \\
\}\,]
\end{array}
$$

Process Representation: Seller (1)

$\mathsf{Seller}[$

$!s \triangleright \mathtt{qReq}(c, \mathit{prod}). \cdots \text{(find price)} \cdots .\overline{c} \triangleleft \mathtt{qRes}\langle \mathit{quote} \rangle.$

$\quad c \triangleright [\, \mathtt{qAccept}(\mathit{cred}, \mathit{adr}).\overline{\mathit{shipper}} \triangleleft \mathtt{ship}\langle \mathit{prod}, \mathit{adr} \rangle.\mathbf{0}$

$\qquad \& \, \mathtt{qRefuse}().\mathbf{0} \,]$

$]$

Process Representation: Seller (2)

$$
\begin{array}{l}
\mathsf{Seller}[\\
!s \triangleright \mathtt{qReq}(c, prod). \cdots (\text{find price}) \cdots .\overline{c} \triangleleft \mathtt{qRes}\langle quote \rangle.\\
\quad c \triangleright [\mathtt{qAccept}(cred, adr).\\
\quad\quad (\nu\, e)\overline{cca} \triangleleft \mathtt{check}\langle cred, e \rangle.\\
\quad\quad\quad e \triangleright [\mathtt{credOk}.\overline{c} \triangleleft \mathtt{orderConf}\langle\rangle.\\
\quad\quad\quad\quad \overline{shipper} \triangleleft \mathtt{ship}\langle prod, adr \rangle\\
\quad\quad\quad \& \mathtt{credNotOk}().\overline{c} \triangleleft \mathtt{orderRefused}\langle\rangle\ ]\\
\quad \&\ \mathtt{qRefuse}().\mathbf{0}]\\
]
\end{array}
$$

Process Representation: CCA

$$!cca \triangleright \texttt{check}(cred, e).$$

$$\text{if } Ok(cred) \text{then}$$

$$\overline{e} \triangleleft \texttt{credOk}$$

$$\text{else}$$

$$\overline{e} \triangleleft \texttt{credNotOk}\langle\rangle.\mathbf{0}.$$

"CCA" stands for **C**redit**C**heck**A**gency.

Process Representation: Shipper

$$!shipper(prod, adr).\overline{adr} \triangleleft \mathtt{deliver}\langle Goods(prod)\rangle$$

Session Types (1): Buyer and Seller

…haven't we seen these types before?

$$s \,@\, \mathbf{Buyer} : \quad \uparrow\mathsf{QuoteReq}(\mathit{qreq.xsd})\ ;$$
$$\downarrow\mathsf{QuoteRes}(\mathit{quote.xsd})\ ;$$
$$\uparrow\mathsf{QuoteAcc}(\mathit{address.xsd}) \oplus \uparrow\mathsf{QuoteRef}(\,)$$

$$s \,@\, \mathbf{Seller} : \quad \downarrow\mathsf{QuoteReq}(\mathit{qreq.xsd})\ ;$$
$$\uparrow\mathsf{QuoteRes}(\mathit{quote.xsd})\ ;$$
$$\downarrow\mathsf{QuoteAcc}(\mathit{address.xsd})\ \&\ \downarrow\mathsf{QuoteRef}(\,)$$

Session Types (2): Seller and CCA

…well these look new.

$$cca \,@\, \textbf{CCA} : \quad \downarrow\mathsf{check}(\mathit{credit.xsd})\ ;\ \uparrow\mathsf{credOk}() \oplus \uparrow\mathsf{credNotOk}()$$

$$cca \,@\, \textbf{Seller} : \quad \uparrow\mathsf{check}(\mathit{credit.xsd})\ ;\ \uparrow\mathsf{credOk}() \,\&\, \uparrow\mathsf{credNotOk}()$$

Session Types (3): The Rest

For Shipper and Seller:

$$\textit{shipper} \,@\, \mathbf{Shipper} : \quad \downarrow \mathsf{ship}(\textit{shipping.xsd})$$

$$\textit{shipper} \,@\, \mathbf{Seller} : \quad \uparrow \mathsf{ship}(\textit{shipping.xsd})$$

For Shipper and Buyer:

$$\textit{address} \,@\, \mathbf{Shipper} : \quad \uparrow \mathsf{deliver}(\textit{DVD.xsd})$$

$$\textit{address} \,@\, \mathbf{Buyer} : \quad \downarrow \mathsf{deliver}(\textit{DVD.xsd})$$

Session Types: summary

- Session types offer clean abstraction of communication behaviour of processes, and prevents communication error ("well-typed processes do not go wrong").
- They become a basis of more refined behavioural typing, such as deadlock freedom.
- Session types can also be incorporated into other languages: for example, we recently studied session types in Java [Dezani et al. 2005].

Types for Deadlock-Freedom (1)

A Seller with mutex.

$$\mathsf{Seller}[$$

$$!s \triangleright \mathtt{qReq}(c, prod).\mathbf{mBegin}.\cdots(\text{find price})\cdots.\overline{c} \triangleleft \mathtt{qRes}\langle quote \rangle.$$

$$c \triangleright [\mathtt{qAccept}(cred, adr).$$

$$(\nu\, e)\overline{cca} \triangleleft \mathtt{check}\langle cred, e \rangle.$$

$$e \triangleright [\mathtt{credOk}.\overline{c} \triangleleft \mathtt{orderConf}\langle\rangle.$$

$$\overline{shipper} \triangleleft \mathtt{ship}\langle prod, adr \rangle.\mathbf{mEnd}.\mathbf{0}$$

$$\& \mathtt{credNotOk}().\overline{c} \triangleleft \mathtt{orderRefused}\langle\rangle.\mathbf{mEnd}.\mathbf{0}\ ]$$

$$\&\ \mathtt{qRefuse}().\mathbf{mEnd}.\mathbf{0}] \quad]$$

Types for Deadlock-Freedom (2)

The following CCA lets the system deadlock:

$$!cca \triangleright \mathtt{check}(cred, e).$$

$$\text{if } Ok(cred) \text{then}$$

$$\overline{e} \triangleleft \mathtt{credOk}$$

$$\text{else}$$

$$\overline{s} \triangleleft \mathtt{quoteReq}\langle prod \rangle \ldots \overline{e} \triangleleft \mathtt{credNotOk}\langle\rangle.\mathbf{0}$$

because, when s is invoked, Seller's state is locked.

Types for Deadlock-Freedom (3)

This deadlock can be detected by composing:

$$
\begin{array}{ll}
\textbf{Seller}: & s(c) :\downarrow \mathsf{QuoteReq}(qreq.xsd)\ ; \\
& \textbf{mutex}\ \langle\ BS :\uparrow \mathsf{QuoteRes}(quote.xsd)\ ; \\
& \qquad BS :\downarrow \mathsf{QuoteAcc}(address.xsd)\ ; \\
& \qquad cca(e) :\downarrow \mathsf{check}(creddata.xsd)\ ; \\
& \qquad \ldots \\
& \rangle\ @\ \mathsf{Seller} \\
\textbf{CCA}: & \quad cca(e) :\downarrow \mathsf{check}(creddata.xsd)\ ; \\
& \quad s(c) :\uparrow \mathsf{QuoteReq}(qreq.xsd)\ ; \\
& \quad \ldots
\end{array}
$$

NB: *The red part induces circularity within the mutex block.*

Guarantee of Livelock-Freedom

Prevention of livelock is built on the basis of deadlock-free typing. While general verification demands a termination proof, we can often use simpler circularity detection as for the deadlock freedom.

Two-Way Embeddings (1)

We now have two calculi, which can be related by:

- *End-point projection (EPP)*, which maps a global description to communicating processes.
- *Global-view extraction (GVE)*, which maps processes in the end-point calculus to a global description.

Details of technical results will be published later: here we discuss general aspects of our ongoing study on EPP and GVE.

Two-Way Embeddings (2)

A crucial requirement for effective engineering:

- EPP and GVE should be, under a suitable constraint, inverse to each other.
  1. If you EPP a global description to get local processes and then GVE them, you get the original description.
  2. If you GVE local processes to get a global view and apply EPP to them, you get the original processes.
- Another related engineering requirement: both EPP and GVE should result in concise, understandable, and effective descriptions.

Two-Way Embeddings (3)

A tension between global and local descriptions:

- In global description, one may wish to be as neutral as possible (e.g. as to the choice of branches).
- In contrast, executable end-point processes demand specialised interaction scenarios (e.g. your application may not use all possible options a service provides).
- Which engineering settings prefer which methods (e.g. monitoring)? How can we unify different views? Is there a principled way to reach a conceivably most general interaction scenario?

Two-Way Embeddings (4)

Scenes which use the EPP-GVE pair in different ways (1).

- Describing/programming individual services from scratch, both a service and a (prototype) client.
- Using existing services to realise a global scenario.
- Provide a template for generic service descriptions, perhaps in different executable languages.
- Validation: check your particular interaction scenario against a given general one (conformance).
- Creating monitoring tools, both a uniform one and a tailor-made one.

Two-Way Embeddings (5)

Scenes which use the EPP-GVE pair in different ways (2).

- Combine different requirements/designs/scenarios into a most global design.
- Stipulate legitimate interaction scenarios within/between organisations as a set of protocols (this can even be part of a law!).
- Refine/evolve existing application scenarios to conform to change in technologies, demands, etc. How can we ensure compatibility? Is there any way to "gracefully" break compatibility?

Two-Way Embeddings (6)

Integration with security concerns is important in real-world engineering. Possibly useful sources are:

- Guttman's Strand Space offers a promising general and pragmatic method which will work well with the present framework.
- Secrecy (confidentiality) is studied in detail by Myers, Yoshida, Honda, Kobayashi and others. Access control is studied by Yoshida and others.
- Studies on an applied version of the π-calculus by Abadi, Gordon and Fornet also offer a useful source.

Another Remark

CDL or its future version can have an end-point protocol description language which precisely corresponds to the global language in the same way the end-point calculus corresponds to the CDL calculus.

And this pair of descriptive languages can become a basic programming tool for the web...

Related Work

- Kavantzas's calculus.
- Lucci et al.'s calculus.
- Strand Space (two-way embeddings).
- All work on communication-centric programming languages, including Occam, Pict and Occam-Pi.
- All work on types (ML, Java, ..) and, in particular, deadlock/livelock-free typing.
- All work on concurrent programming.

6. Further Topics

π-Calculus and CDL

- CDL offers an expressive language for describing global interactions: π-calculus offers a rigorous notion of behaviour, which can be reasoned, composed and controlled on the basis of its rich theories.
- We can analyse CDL and CDL descriptions through π-calculus theories; We may also directly reflect π-calculus theories onto CDL and its tools.
- Exploring rich possibilities of this relationship will lead to distinguishing competitive edges of CDL and associated tools, services, and other products.

A CDL Workflow

1. Understand requirement. [$\longrightarrow 2$].

2. Describe/edit (+static checking) [$\longrightarrow 1, 3$].

3-a. Fill end-point business logic (+verification).

3-b. Build executable end-points (+verification). [$\longrightarrow 1, 2, 4$].

4. Validate/test compatibility, conformance, etc.
[$\longrightarrow 1, 2, 3, 5$].

5. Ship/install/bind/compose/run (+monitoring).
[$\longrightarrow 1, 2, 3, 4$].

Theories and CDL Workflow

1. *Requirement:* graphical/textual syntax for describing organisational interaction.

2. *Editing:* IDE with graphical/syntactic editors, internal representation, type/model checking.

3. *End-point logic:* various verification tools (in combination with tools in other layers).

4. *Validation/testing:* type/model checking, verification engines, assertions, ...

5. *Deployment/execution:* monitoring, managerial documentation, error handling, ...

Futher Notes on the Use of Theories

- The presented theory is intended to be a basis upon which known static/dynamic checking techniques can be effectively exploited.
- Inter-organisational nature of web services makes their theoretical underpinning all the more essential.
- Another possible uses of theories include the understanding on (often unexpected) effects of combination of diverse language constructs;

Action Plan

Under the auspice of W3C, CDL's formal analysis phase will start soon. Anticipating that phase, we shall embark on full examination of the potential applicability of the presented theory and its ramifications. Activities in this phase will involve not only theoretical inquiries but also understanding on requirement for software tools for CDL, and their development and experiment. We hope we can join force in making this language and its future versions a true foundation for high-level description of present and future web-services.