Orchestration vs Choreography

- In many cases, there is **no unique point** of invocation for the services
  - In these cases, we say that the system is a **choreography**
  - Let starts with an example:
    - Consider the popular OpenID **protocol** for distributed authentication via third party services
OpenID

1. User sends username
OpenID

1. User sends username

2. Username is forwarded
OpenID

1. User sends username

2. Username is forwarded

3. User sends password
OpenID

1. User sends username
2. Username is forwarded
3. User sends password
4. Authorized/Not authorized
How to model choreographies?

Several approaches are usually adopted to model (and program) choreographies

- MSC (Message Sequence Charts)
- Cryptographic protocol notation
- Open workflow nets
- Collaboration diagrams
- Process calculi-like languages
- …
Message Sequence Chart for OpenID

1. **End-user**
   - **log me in**

2. **Relying Party**
   - **authentication request**
   - **authentication and authorization request**
   - **authenticate & authorize**
   - **authentication response**
   - **Userinfo request**

3. **OpenID Provider**
   - **Userinfo response**

Message:
- Hi. You're logged in with {Userinfo.name}
1. User sends username
1. User sends username

2. Username is forwarded
Message Sequence Chart for OpenID

1. User sends username
2. Username is forwarded
3. User sends password
Message Sequence Chart for OpenID

1. User sends username
2. Username is forwarded
3. User sends password
4. Authorized/Not authorized
Chryptographic protocol notation

- (Part of the) Needham-Schroeder authentication with public key:

\[
\begin{align*}
A &\rightarrow B: \{ n_A, A \}_{kB} \\
B &\rightarrow A: \{ n_A, n_B \}_{kA} \\
A &\rightarrow B: \{ n_B \}_{kB}
\end{align*}
\]
Cryptographic protocol notation

- A man-in-the-middle attack at the Needham-Schroeder protocol:

  A → I: \(\{n_A, A\}_{k_I}\)

  I → B: \(\{n_A, A\}_{k_B}\)

  B → I: \(\{n_A, n_B\}_{k_A}\)

  I → A: \(\{n_A, n_B\}_{k_A}\)

  A → I: \(\{n_B\}_{k_I}\)

  I → B: \(\{n_B\}_{k_B}\)
Chryptographic protocol notation

- The **patched** Needham-Schroeder-Lowe protocol:

  A → B: \{n_A, A\}_{kB}

  B → A: \{n_A, n_B, B\}_{kA}

  A → B: \{n_B\}_{kB}
Open workflow nets

- Each partner is modeled by a Petri net with input/output places
  - The partners are combined by merging their input/output places

Models and Languages for Service-Oriented and Cloud Computing

Bertinoro 7-11/3/2016
Collaboration diagram
Two kinds of dependencies among events:

- **Local dependencies:** total ordering of the events on the same channel (A2 follows A1)
- **Causal dependencies:** A2/C2 indicates that C2 causally depends on A2
The process-calculus approach

The process-calculus approach:
- Define the basic **actions** ...
- ... define operators to combine actions thus obtaining **processes**

Approach **adopted** in the most popular choreography language in the context of Web Services:
- **WS-CDL**: Web Service Choreography Description Language
WS-CDL approach

- Global view of service interactions

- Buyer
- Seller
- Bank
WS-CDL approach

- Global view of service interactions

Request

Buyer

Bank

Seller
WS-CDL approach

- Global view of service interactions

![Diagram showing interactions between Buyer, Seller, and Bank]

- Request
- Offer
- PayDescr

Models and Languages for Service-Oriented and Cloud Computing

Bertinoro 7-11/3/2016
WS-CDL approach

- Global view of service interactions
WS-CDL approach

- Global view of service interactions

Models and Languages for Service-Oriented and Cloud Computing
Bertinoro 7-11/3/2016
WS-CDL approach

Request$_{Buyer\rightarrow Seller}$;
(Offer$_{Seller\rightarrow Buyer}$ |
PayDescr$_{Seller\rightarrow Bank}$ );
Payment$_{Buyer\rightarrow Bank}$;
(Confirm$_{Bank\rightarrow Seller}$ |
Receipt$_{Bank\rightarrow Buyer}$ )
WS-CDL approach

Request \(\text{Buyer} \rightarrow \text{Seller}\);
\((\text{Offer}_{\text{Seller} \rightarrow \text{Buyer}} \mid \text{PayDescr}_{\text{Seller} \rightarrow \text{Bank}})\);
Payment \(\text{Buyer} \rightarrow \text{Bank}\);
\((\text{Confirm}_{\text{Bank} \rightarrow \text{Seller}} \mid \text{Receipt}_{\text{Bank} \rightarrow \text{Buyer}})\)

Basic Action
Buyer sends a message “Request” to Seller
WS-CDL approach

Request_{Buyer\rightarrow Seller};
(Offer_{Seller\rightarrow Buyer} | PayDescr_{Seller\rightarrow Bank});
Payment_{Buyer\rightarrow Bank};
(O Confirm_{Bank\rightarrow Seller} | Receipt_{Bank\rightarrow Buyer})
Basic Action
Buyer sends a message “Request” to Seller

Sequential composition

Parallel composition

WS-CDL approach

Request Buyer→Seller ;
(Offer Seller→Buyer | PayDescr Seller→Bank ) ;
Payment Buyer→Bank ;
(Confirm Bank→Seller | Receipt Bank→Buyer )
Interaction Oriented Approach

- In the WS-CDL approach the basic **action** corresponds to
  - *a sender that sends a message on an operation exposed by a receiver*
  - this is the so-called **global view** approach
- Easy to formalise ... we will see:

We have defined a choreography process calculus.
Choreography operational semantics

\[ a_r \rightarrow_s a_r \rightarrow^s 1 \]

\[
\begin{align*}
H & \xrightarrow{\eta} H' \\
H + L & \xrightarrow{\eta} H'
\end{align*}
\]

\[
\begin{align*}
H & \xrightarrow{\eta} H' & \eta \neq \checkmark \\
H;L & \xrightarrow{\eta} H';L
\end{align*}
\]

\[
\begin{align*}
H & \xrightarrow{\checkmark} H' & L & \xrightarrow{\checkmark} L' \\
H | L & \xrightarrow{\checkmark} H'|L'
\end{align*}
\]

\[
\begin{align*}
H & \xrightarrow{\eta} H' & \eta \neq \checkmark \\
H | L & \xrightarrow{\eta} H'|L
\end{align*}
\]

\[
H^* \xrightarrow{\eta} H'; H^* \xrightarrow{\eta} H'
\]

Models and Languages for Service-Oriented and Cloud Computing

Bertinoro 7-11-3/2016
A simple choreography (1)

- A **client** sends a reservation request to a **travel agency**, which contacts an **airplane company** and a **hotel**, and finally notify the client by confirming or cancelling the reservation:

```
Reservation_{Client → TravelAgency};
((Reserve_{TravelAgency → AirCompany}; ConfirmFlight_{AirCompany → TravelAgency}) |
(Reserve_{TravelAgency → Hotel}; ConfirmRoom_{Hotel → TravelAgency}) );
Confirmation_{TravelAgency → Client} + Cancellation_{TravelAgency → Client}
```
A simple choreography (2)

- A **client** negotiates with a **shop** the price for a product she initially indicates

\[
ProdID_{\text{client} \rightarrow \text{shop}}; \\
(Price_{\text{shop} \rightarrow \text{client}}; Offer_{\text{client} \rightarrow \text{shop}})^*; \\
(Confirm_{\text{shop} \rightarrow \text{client}} + Decline_{\text{shop} \rightarrow \text{client}})
\]
Subtleties in choreographies

- Suppose the decision to **Decline** is taken by the client:

\[
\text{ProdID}_{\text{client}} \rightarrow \text{shop}; \\
(\text{Price}_{\text{shop}} \rightarrow \text{client}; \text{Offer}_{\text{client}} \rightarrow \text{shop})^*; \\
(\text{Confirm}_{\text{shop}} \rightarrow \text{client} + \text{Decline}_{\text{client}} \rightarrow \text{shop})
\]

- Are there possible **problems** with this choreography?
Possible problems

\[ \text{ProdID}_{\text{client} \rightarrow \text{shop}}; \]
\[ (\text{Price}_{\text{shop} \rightarrow \text{client}}; \text{Offer}_{\text{client} \rightarrow \text{shop}})^*; \]
\[ (\text{Confirm}_{\text{shop} \rightarrow \text{client}} + \text{Decline}_{\text{client} \rightarrow \text{shop}}) \]

- **Intuition:** problems could depend on the kind of communication
  - **Synchronous:** (no problem) client and shop synchronise on the branch to select
  - **Asynchronous:** (problem) client and shop could select different branches (by emitting inconsistent messages)
Choreography correctness

- We have performed a **formal** study of choreography correctness
  - A calculus for **choreography** (similar to the previous one, without repetition)
  - Canonical **projection**:
    - syntactic extraction from the choreography of the behaviour of the single roles
  - Check that the parallel composition of the projections **behaves** like the choreography
We have defined an Interaction Oriented Choreography (IOC) calculus.

\[ \mathcal{I} ::= a \xrightarrow{o} b \mid 1 \mid 0 \mid \mathcal{I}; \mathcal{I}' \mid \mathcal{I} \parallel \mathcal{I}' \mid \mathcal{I} + \mathcal{I}' \]

- **Basic Action**: \( a \) sends a message \( o \) to \( b \)
- **Sequence**
- **Parallel**
- **Choice**

Models and Languages for Service-Oriented and Cloud Computing

Bertinoro 7-11/3/2016
IOC: Interaction Oriented Choreography calculus

- We have defined an Interaction Oriented Choreography (IOC) calculus

\[ \mathcal{I} ::= a \rightarrow b \mid 1 \mid 0 \mid \mathcal{I}; \mathcal{I}' \mid \mathcal{I} || \mathcal{I}' \mid \mathcal{I} + \mathcal{I}' \]

Success

Failure
IOC: operational semantics

\[(\text{Interaction})\]
\[a \xrightarrow{\sigma} b \quad a \xrightarrow{\sigma b} \rightarrow 1\]

\[(\text{End})\]
\[1 \xrightarrow{\sqrt{\sigma}} 0\]

\[(\text{Par-End})\]
\[I \xrightarrow{\sqrt{\sigma}} I' \quad J \xrightarrow{\sigma} J'\]
\[I; J \xrightarrow{\sigma} J'\]

\[(\text{Sequence})\]
\[I \xrightarrow{\sigma} I' \quad \sigma \neq \sqrt{\sigma}\]
\[I; J \xrightarrow{\sigma} I'; J\]

\[(\text{Parallel})\]
\[I \xrightarrow{\sigma} I' \quad \sigma \neq \sqrt{\sigma}\]
\[I \parallel J \xrightarrow{\sigma} I'; J\]

\[(\text{Choice})\]
\[I \xrightarrow{\sigma} I'\]
\[I + J \xrightarrow{\sigma} I'\]
POC: Process Oriented Choreography calculus

- We have defined a Process Oriented Choreography (POC) calculus

Processes:

\[ P ::= o \mid \overline{o} \mid 1 \mid 0 \mid P; P' \mid P \mid P' \mid P + P' \]

Choreography:

\[ S ::= (P)_a \mid S \parallel S' \]

Process \( P \) playing role \( a \)
POC: operational semantics

-Processes-

\[ (\text{IN}) \quad o \xrightarrow{o} 1 \]

\[ (\text{SYNC-OUT}) \quad \overline{o} \xrightarrow{\langle o \rangle} 1 \]

\[ (\text{ONE}) \quad 1 \xrightarrow{\sqrt{\cdot}} 0 \]

\[ (\text{SEQUENCE}) \quad P \xrightarrow{\gamma} P' \quad \gamma \neq \sqrt{\cdot} \]

\[ P; Q \xrightarrow{\gamma} P'; Q \]

\[ (\text{CHOICE}) \quad P \xrightarrow{\gamma} P' \]

\[ P + Q \xrightarrow{\gamma} P' \]

\[ (\text{INNER PARALLEL}) \quad P \xrightarrow{\gamma} P' \quad Q \xrightarrow{\gamma} Q' \]

\[ P \mid Q \xrightarrow{\gamma} P' \mid Q \]

\[ (\text{SEQ-END}) \quad \]

\[ P \xrightarrow{\sqrt{\cdot}} P' \quad Q \xrightarrow{\sqrt{\cdot}} Q' \]

\[ P ; Q \xrightarrow{\sqrt{\cdot}} Q' \]

\[ (\text{INNER PAR-END}) \quad \]

\[ P \xrightarrow{\sqrt{\cdot}} P' \quad Q \xrightarrow{\sqrt{\cdot}} Q' \]

\[ P \mid Q \xrightarrow{\sqrt{\cdot}} P' \mid Q' \]
POC: operational semantics
-Choreographies-

(I N N E R)

\[ P \xrightarrow{\gamma} P' \quad \gamma \neq \overline{0}, \sqrt{\cdot} \]

\[(P)_{a} \xrightarrow{\gamma:a} (P')_{a}\]

(SYNCHRO)

\[ S \xrightarrow{\langle o \rangle:a} S' \quad S'' \xrightarrow{o:b} S''' \]

\[ S \parallel S'' \xrightarrow{a \overrightarrow{b}} S' \parallel S''' \]

(EXT PARALLEL)

\[ S \xrightarrow{\gamma} S' \]

\[ S \parallel S'' \xrightarrow{\gamma} S' \parallel S'' \]
POC: the client-shop example

- Here is the behaviour of the **client** and the **shop** in the first of the two already considered choreographies:

\[
\left( \text{ProdID}; (\text{Price}; \text{Offer})^*; (\text{Confirm} + \text{Decline}) \right)_{\text{client}} \parallel \\
\left( \text{ProdID}; (\text{Price}; \text{Offer})^*; (\text{Confirm} + \text{Decline}) \right)_{\text{shop}}
\]
POC: the client-shop example

- Here is the alternative version where the client sends the **decline** message:

\[
(\frac{\text{ProdID}}{\text{ProdID}}; (\frac{\text{Price}}{\text{Price}}; \frac{\text{Offer}}{\text{Offer}})^*; (\frac{\text{Confirm}}{\text{Confirm}} + \frac{\text{Decline}}{\text{Decline}}) )_{\text{client}} \parallel (\frac{\text{ProdID}}{\text{ProdID}}; (\frac{\text{Price}}{\text{Price}}; \frac{\text{Offer}}{\text{Offer}})^*; (\frac{\text{Confirm}}{\text{Confirm}} + \frac{\text{Decline}}{\text{Decline}}) )_{\text{shop}}
\]

- **No problem:** there are three possible choices (*Price*, *Confirm*, and *Decline*) and the client / shop **synchronize** in taking this decision!
Formal correspondence

- The IOC and the POC:

\[
\text{ProdID}_{\text{client}} \rightarrow \text{shop};
(\text{Price}_{\text{shop}} \rightarrow \text{client}; \text{Offer}_{\text{client}} \rightarrow \text{shop})^*;
(\text{Confirm}_{\text{shop}} \rightarrow \text{client} + \text{Decline}_{\text{client}} \rightarrow \text{shop})
\]

(ProdID; (Price; Offer)^*; (Confirm + Decline))_{\text{client}} ||
(ProdID; (Price; Offer)^*; (Confirm + Decline))_{\text{shop}}

have exactly the same \textbf{traces} of transitions in the operational semantics
Canonical projection

- In the previous example, the behaviour of the roles in the POC is obtained from the IOC by means of projection:

\[
\begin{align*}
\text{proj}(a \overset{o}{\to} b, a) &= \overline{o} \\
\text{proj}(a \overset{o}{\to} b, c) &= 1 \text{ if } c \neq a, b \\
\text{proj}(1, a) &= 1 \\
\text{proj}(\mathcal{I}; \mathcal{I}', a) &= \text{proj}(\mathcal{I}, a); \text{proj}(\mathcal{I}', a) \\
\text{proj}(\mathcal{I} \parallel \mathcal{I}', a) &= \text{proj}(\mathcal{I}, a) \mid \text{proj}(\mathcal{I}', a) \\
\text{proj}(\mathcal{I} + \mathcal{I}', a) &= \text{proj}(\mathcal{I}, a) + \text{proj}(\mathcal{I}', a)
\end{align*}
\]
The OpenID Example

- The OpenId **IOC:**

\[
\begin{align*}
\text{Username}_{user} & \rightarrow \text{relyingParty;} \\
\text{Username}_{relyingParty} & \rightarrow \text{IdProvider;} \\
\text{Password}_{user} & \rightarrow \text{IdProvider;} \\
(\text{Ok}_{IdProvider} & \rightarrow \text{relyingParty} + \text{Fail}_{IdProvider} \rightarrow \text{relyingParty})
\end{align*}
\]

and the projected **POC:**

\[
\begin{align*}
(\overline{\text{Username}; \overline{\text{Password}}})_{user} & \ || \\
(\text{Username}; \overline{\text{Username}}; (\text{Ok} + \text{Fail}))_{relyingParty} & \ || \\
(\text{Username}; \text{Password}; (\overline{\text{Ok}} + \overline{\text{Fail}}))_{IdProvider}
\end{align*}
\]

- **No Problem! Same traces!**

Models and Languages for Service-Oriented and Cloud Computing

Bertinoro 7-11/3/2016
An alternative end of OpenID

Consider the following **alternative** end:

\[
\text{Password}_{\text{user}} \rightarrow \text{IdProvider};
\]

\[
( \text{Ok}_{\text{IdProvider}} \rightarrow \text{relyingParty}; \text{Notify}_{\text{IdProvider}} \rightarrow \text{user}; \\
\text{Connect}_{\text{user}} \rightarrow \text{relyingParty} )
\]

\[+ ( \text{Fail}_{\text{IdProvider}} \rightarrow \text{relyingParty}; \text{Notify}_{\text{IdProvider}} \rightarrow \text{user} ) \]

and its **projection**:  

\[
( \overline{\text{Password}}; (\overline{\text{Notify}}; \overline{\text{Connect}}) + (\overline{\text{Notify}}) )_{\text{user}} \parallel
\]

\[
( \text{Ok}; \text{Connect} + \text{Fail} )_{\text{relyingParty}} \parallel
\]

\[
( \text{Password}; (\overline{\text{Ok}}; \overline{\text{Notify}}) + (\overline{\text{Fail}}; \overline{\text{Notify}}) )_{\text{IdProvider}}
\]
An alternative end of OpenID

Consider the following **alternative** end:

\[
\text{Password}_{\text{user}} \rightarrow \text{IdProvider}; \\
(\text{Ok}_{\text{IdProvider}} \rightarrow \text{relyingParty}; \text{Notify}_{\text{IdProvider}} \rightarrow \text{user}; \\
\text{Connect}_{\text{user}} \rightarrow \text{relyingParty}) \\
+ (\text{Fail}_{\text{IdProvider}} \rightarrow \text{relyingParty}; \text{Notify}_{\text{IdProvider}} \rightarrow \text{user})
\]

and its **projection:**

\[
(\overline{\text{Password}}; (\overline{\text{Notify}}; \overline{\text{Connect}}) + (\overline{\text{Notify}}))_{\text{user}} \parallel \\
((\overline{\text{Ok}}; \overline{\text{Connect}}) + \overline{\text{Fail}})_{\text{relyingParty}} \parallel \\
(\overline{\text{Password}}; (\overline{\text{Ok}}; \overline{\text{Notify}}) + (\overline{\text{Fail}}; \overline{\text{Notify}}))_{\text{IdProvider}}
\]
Choreography correctness

- Is it possible to check the correctness of a choreography?
  - **Yes**: by controlling three independent conditions
    - Connectedness for **sequence**
    - Unique point of **choice**
    - **Causality** safety
Connectedness for sequence

**Intuition:**
- Given the choreography $C;D$ the rhs $D$ should **start** only when the lhs $C$ has **completed**.
- We check this condition by controlling that among the roles involved in the **completion** of $C$ there is at least one role involved in the **starting** of $D$. 
Initial and final transitions

- The functions $\text{transI}$ and $\text{transF}$ return the interactions in the **starting** and **completion** of a choreography.

\[
\begin{align*}
\text{transI}(a \rightarrow b) &= \text{transF}(a \rightarrow b) = \{a \rightarrow b\} \\
\text{transI}(1) &= \text{transI}(0) = \text{transF}(1) = \text{transF}(0) = \emptyset \\
\text{transI}(I \parallel I') &= \text{transI}(I + I') = \text{transI}(I) \cup \text{transI}(I') \\
\text{transF}(I \parallel I') &= \text{transF}(I) \cup \text{transF}(I') \\
\text{transF}(I + I') &= \text{transF}(I) \cup \text{transF}(I') \\
\text{transI}(I; I') &= \text{transI}(I') \text{ if } I \rightarrow, \text{ transI}(I) \text{ otherwise} \\
\text{transF}(I; I') &= \text{transF}(I) \text{ if } I' \rightarrow, \text{ transF}(I') \text{ otherwise}
\end{align*}
\]
Connectedness for sequence: formally

**Definition 4.2** (Synchronous connectedness for sequence). An IOC $\mathcal{I}$ is synchronous connected for sequence if for each subterm of the form $\mathcal{I};\mathcal{J}$ we have $
exists a \xrightarrow{o} b \in \text{transF}(\mathcal{I}), \forall c \xrightarrow{o'} d \in \text{transI}(\mathcal{J}), \{a, b\} \cap \{c, d\} \neq \emptyset$.

- The last action before a sequential composition will surely **occur before** every possible initial action after
  - guaranteed by the involvement of at least **one role** in both actions
Connectedness for sequence

- Here is a trivial example of an **incorrect** choreography:

\[ a \xrightarrow{o} b; \; c \xrightarrow{o'} d \]

having the following **projection**:

\[ (\overline{o})_a \parallel (o)_b \parallel (\overline{o'})_c \parallel (o')_d \]
Unique point of choice

Intuition:

- Given the choreography $C+D$ all the involved roles should agree on the branch to select:
  - When an initial action in one branch occur, all the initial actions in the other one should be disabled
  - The selected branch must be communicated to all partners, hence the partners involved in $C$ and in $D$ must be the same
Unique point of choice: formally

**Definition 4.3** (Synchronous unique point of choice). An IOC $\mathcal{I}$ has synchronous unique points of choice if for each subterm of the form $\mathcal{I} + \mathcal{J}$ we have $\forall a \xrightarrow{\sigma} b \in \text{transI}(\mathcal{I}), \forall c \xrightarrow{\sigma'} d \in \text{transI}(\mathcal{J}), \{a, b\} \cap \{c, d\} \neq \emptyset$. Furthermore $\text{roles}(\mathcal{I}) = \text{roles}(\mathcal{J})$.

- Every pair of initial actions in the two branches must include a **common** role:
  - this role will be the **unique** point of choice
  - moreover, the selected branch must be communicated to **all involved** roles
Unique point of choice

- Here is a trivial example of an incorrect choreography:

\[
(a \xrightarrow{o} b + a \xrightarrow{o'} c); a \xrightarrow{o} b
\]

having the following projection:

\[
\left( (\overline{o} + \overline{o'}); \overline{o} \right)_a \parallel \left( (o + 1); o \right)_b \parallel \left( (1 + o'); 1 \right)_c
\]
Causality safety

- **Intuition:**
  - If an operation occurs **twice** in a choreography, the two interactions should be **not active contemporaneously**
  - Consider the following **choreography**:

\[
(a \xrightarrow{o} b; b \xrightarrow{o'} c; c \xrightarrow{o} b) + (a \xrightarrow{o'} c; c \xrightarrow{o'} b)
\]

with the **projection**:

\[
(\overline{o} + \overline{o'})_a \parallel ((o; \overline{o'}; o) + o')_b \parallel ((\overline{o'}; \overline{o}) + (\overline{o'}; \overline{o'}))_c
\]
Causality safety: formally

Definition 4.5 (Synchronous causality-safety). An IOC is synchronous causality-safe iff for each pair of interactions $i$ and $j$ using the same operation, either $s_i \leq_s r_j \land r_i \leq_s s_j$ or $s_j \leq_s r_i \land r_j \leq_s s_i$.

- The relation $\leq_s$ formalises causal dependencies
  - hence if $x \leq_s y$ the two events $x$ and $y$ cannot be both ready to be executed
Causality safety: formally

**sequentiality:** for each $I, I'$, if $i$ is an interaction in $I$, $j$ is an interaction in $I'$, and $e_i$ and $e_j$ are events in the same role then $e_i \leq_s e_j$;

**synchronization:** for each $i, j$ if $e_i \leq_s e_j$ then $\overline{e_i} \leq_s e_j$.

- The relation $\leq_s$ formalises **causal dependencies**
  - hence if $x \leq_s y$ the two events $x$ and $y$ cannot be both ready to be executed
Exercise

- Which condition is not satisfied in the **wrong** alternative OpenID end?

\[
\text{Password}_{user} \rightarrow \text{IdProvider}; \\
(\text{Ok}_{IdProvider} \rightarrow \text{relyingParty}; \text{Notify}_{IdProvider} \rightarrow \text{user}; \\
\text{Connect}_{user} \rightarrow \text{relyingParty} ) \\
+ (\text{Fail}_{IdProvider} \rightarrow \text{relyingParty}; \text{Notify}_{IdProvider} \rightarrow \text{user} )
\]
Exercise

Which condition is not satisfied in the **wrong** alternative OpenID end?

\[
Password_{user\rightarrow IdProvider};
(\text{Ok}_{IdProvider\rightarrow relyingParty}; \text{Notify}_{IdProvider\rightarrow user};
\text{Connect}_{user\rightarrow relyingParty})
+
(\text{Fail}_{IdProvider\rightarrow relyingParty}; \text{Notify}_{IdProvider\rightarrow user})
\]

Connectedness for sequence: ?
Exercise

- Which condition is not satisfied in the **wrong** alternative OpenID end?

```
Password_{user \rightarrow IdProvider};
( Ok_{IdProvider \rightarrow relyingParty}; Notify_{IdProvider \rightarrow user};
Connect_{user \rightarrow relyingParty} )
+( Fail_{IdProvider \rightarrow relyingParty}; Notify_{IdProvider \rightarrow user} )
```

Connectedness for sequence: ✔
Exercise

Which condition is not satisfied in the **wrong** alternative OpenID end?

\[
\text{Password}_{\text{user}} \rightarrow \text{IdProvider}; \\
( \text{Ok}_{\text{IdProvider}} \rightarrow \text{relyingParty}; \text{Notify}_{\text{IdProvider}} \rightarrow \text{user}; \\
\text{Connect}_{\text{user}} \rightarrow \text{relyingParty} ) \\
+( \text{Fail}_{\text{IdProvider}} \rightarrow \text{relyingParty}; \text{Notify}_{\text{IdProvider}} \rightarrow \text{user} )
\]

Unique point of choice: ?
Exercise

Which condition is not satisfied in the **wrong** alternative OpenID end?

```
Password_{user \to IdProvider};
( Ok_{IdProvider \to relyingParty}; Notify_{IdProvider \to user};
  Connect_{user \to relyingParty} )
+( Fail_{IdProvider \to relyingParty}; Notify_{IdProvider \to user} )
```

Unique point of choice: ✔
Exercise

Which condition is not satisfied in the wrong alternative OpenID end?

\[
\text{Password}_{\text{user}} \rightarrow \text{IdProvider};
\]
\[
(\text{Ok}_{\text{IdProvider}} \rightarrow \text{relyingParty}; \text{Notify}_{\text{IdProvider}} \rightarrow \text{user};
\text{Connect}_{\text{user}} \rightarrow \text{relyingParty} )
\]
\[
+(\text{Fail}_{\text{IdProvider}} \rightarrow \text{relyingParty}; \text{Notify}_{\text{IdProvider}} \rightarrow \text{user} )
\]

Causality safety: ?
Exercise

- Which condition is not satisfied in the **wrong** alternative OpenID end?

\[
\begin{align*}
\text{Password}_{user \rightarrow IdProvider} ; \\
\text{( Ok}_{IdProvider \rightarrow relyingParty} ; \\
\text{Connect}_{user \rightarrow relyingParty} ) \\
+ \text{( Fail}_{IdProvider \rightarrow relyingParty} ; \\
\text{Notify}_{IdProvider \rightarrow user} \\
\text{Notify}_{IdProvider \rightarrow user} )
\end{align*}
\]

Causality safety: ×
Let’s go back to the client-shop example

\[
ProdID_{client \rightarrow shop}; \\
(Price_{shop \rightarrow client}; Offer_{client \rightarrow shop})^*; \\
(Confirm_{shop \rightarrow client} + Decline_{client \rightarrow shop})
\]

\[
\left(ProdID; (Price; Offer)^*; (Confirm + Decline)\right)_{client} || \\
\left(ProdID; (Price; Offer)^*; (Confirm + Decline)\right)_{shop}
\]

- Remember: possible **problems** if the client sends *Decline* while the shop *Price*
  - This could happen in **asynchronously communicating systems**
Asynchronous choreography correctness

- We have defined **asynchronous communication** for POCs
  - We have defined the **correspondence** between IOCs and their asynchronous execution
    - More complex because **send** and **receive** events are **disjoint** in POCs
  - We have revisited the three **correctness** conditions that guarantee correspondence
POC: Asynchronous semantics

\[ (\text{IN}) \quad o \xrightarrow{o} 1 \quad (\text{OUT}) \quad \overline{o} \xrightarrow{\overline{o}} 1 \quad (\text{SYNC-OUT}) \quad \langle o \rangle \xrightarrow{\langle o \rangle} 1 \quad (\text{ONE}) \quad 1 \xrightarrow{\sqrt{\cdot}} 0 \]

\[ (\text{SEQUENCE}) \quad P \xrightarrow{\gamma} P' \quad \gamma \neq \sqrt{\cdot} \]

\[ P ; Q \xrightarrow{\gamma} P' ; Q \]

\[ (\text{CHOICE}) \quad P \xrightarrow{\gamma} P' \]

\[ P + Q \xrightarrow{\gamma} P' \]

\[ (\text{INNER PARALLEL}) \quad P \xrightarrow{\gamma} P' \]

\[ P \parallel Q \xrightarrow{\gamma} P' \parallel Q \]

\[ (\text{SEQ-END}) \quad \]

\[ P \xrightarrow{\sqrt{\cdot}} P' \quad Q \xrightarrow{\sqrt{\cdot}} Q' \]

\[ P ; Q \xrightarrow{\sqrt{\cdot}} Q' \]

\[ (\text{INNER PAR-END}) \quad P \parallel Q \xrightarrow{\sqrt{\cdot}} P' \parallel Q' \]
POC: Asynchronous semantics

(INNER)

\[
P \xrightarrow{\gamma} P' \quad \gamma \neq \overline{o}, \sqrt{\gamma}
\]

\[
(P)_a \xrightarrow{\gamma:a} (P')_a
\]

(SYNCHRO)

\[
S \xrightarrow{\langle o \rangle:a} S' \quad S'' \xrightarrow{\overline{o}:b} S'''
\]

\[
S \parallel S'' \xrightarrow{a \overline{o} b} S' \parallel S'''
\]

(MSG)

\[
P \xrightarrow{\overline{o}} P'
\]

\[
(P)_a \xrightarrow{\overline{o}:a} (P' \mid \langle o \rangle)_a
\]

(EXT PARALLEL)

\[
S \xrightarrow{\gamma} S'
\]

\[
S \parallel S'' \xrightarrow{\gamma} S' \parallel S''
\]
Correspondence relation

- At the level of POC the interaction coincides with the **receive event**
  - Hence, interaction correspondence guarantees correspondence of receive (**send** could not correspond)
  - See the choreography: \((a \xrightarrow{\sigma} b; c \xrightarrow{\sigma'} b)\)

  with projection: \((\overline{o})_a \parallel (o; o')_b \parallel (\overline{o'})_c\)
Several correspondence relations

- The previous correspondence is named **receiver** correspondence
  - We have considered a **lattice** of correspondence relations
  - In the literature, receiver seems the **mainly considered** correspondence
Receiver connectedness for sequence

- There are two ways to preserve the receive events in a sequential composition $C ; D$
  - the receiver in $C$ is the emitter in $D$
    
    \[(a \overset{o}{\rightarrow} b; \ b \overset{o'}{\rightarrow} c)\]
  - the receiver in $C$ is also the receiver in $D$
    
    \[(a \overset{o}{\rightarrow} b; \ c \overset{o'}{\rightarrow} b)\]
Receiver connectedness for sequence

Definition 5.7 (Receiver connectedness for sequence). An IOC $\mathcal{I}$ is receiver connected for sequence if for each subterm of the form $\mathcal{I}; \mathcal{J}$ we have $\forall a \xrightarrow{\circ} b \in \text{transF}(\mathcal{I}), \forall c \xrightarrow{\circ'} d \in \text{transI}(\mathcal{J}). b = c \lor b = d$.

- the **receiver** in $\mathcal{C}$ is the **emitter** in $\mathcal{D}$

\[(a \xrightarrow{\circ} b; b \xrightarrow{\circ'} c)\]

- the **receiver** in $\mathcal{C}$ is also the **receiver** in $\mathcal{D}$

\[(a \xrightarrow{\circ} b; c \xrightarrow{\circ'} b)\]
Asynchronous unique point of choice

**Definition 5.1** (Asynchronous unique point of choice). An IOC $\mathcal{I}$ has asynchronous unique points of choice if for each subterm of the form $\mathcal{I} + \mathcal{J}$ we have $\forall a \xrightarrow{o} b \in \text{transI}(\mathcal{I}), \forall c \xrightarrow{o'} d \in \text{transI}(\mathcal{J}). a = c$. Furthermore $\text{roles}(\mathcal{I}) = \text{roles}(\mathcal{J})$.

- In an asynchronous setting, the **emitter** can locally decide to send a message
- In a choice only the emitter **decide** a branch
Asynchronous causality safety

Definition 4.5 (Synchronous causality-safety). An IOC is synchronous causality-safe iff for each pair of interactions \( i \) and \( j \) using the same operation, either

\[
s_i \leq_s r_j \land r_i \leq_s s_j \text{ or } s_j \leq_s r_i \land r_j \leq_s s_i.
\]

- Same definition, but with a relation \( \leq_a \) that formalises causal dependencies for the asynchronous semantics
Asynchronous causality safety

**sequentiality:** for each $\mathcal{I}, \mathcal{I}'$, if $i$ is an interaction in $\mathcal{I}$, $j$ is an interaction in $\mathcal{I}'$, and $r_i$ and $e_j$ are respectively a receive and a generic event in the same role then $r_i \leq_a e_j$;

**synchronization:** for each $i, j$ if $r_i \leq_a e_j$ then $s_i \leq_a e_j$ (here $s_i$ can be an output or a message).

- Same definition, but with a relation $\leq_a$ that formalises **causal dependencies** for the asynchronous semantics.
Exercise: the initial WS-CDL example

Request_{Buyer→Seller} ;
( Offer_{Seller→Buyer} | PayDescr_{Seller→Bank} ) ;
Payment_{Buyer→Bank} ;
( Confirm_{Bank→Seller} | Receipt_{Bank→Buyer} )
Exercise: the initial WS-CDL example

\[
\text{Request}_{\text{Buyer} \rightarrow \text{Seller}} ; \\
( \text{Offer}_{\text{Seller} \rightarrow \text{Buyer}} \mid \text{PayDescr}_{\text{Seller} \rightarrow \text{Bank}} ) ; \\
\text{Payment}_{\text{Buyer} \rightarrow \text{Bank}} ; \\
( \text{Confirm}_{\text{Bank} \rightarrow \text{Seller}} \mid \text{Receipt}_{\text{Bank} \rightarrow \text{Buyer}} )
\]

Asynchronous unique point of choice: ?
Exercise: the initial WS-CDL example

Asynchronous unique point of choice: ✔

Request_{Buyer→Seller} ;
(Offer_{Seller→Buyer} | PayDescr_{Seller→Bank} ) ;
Payment_{Buyer→Bank} ;
(O Confirm_{Bank→Seller} | Receipt_{Bank→Buyer} )
Exercise: the initial WS-CDL example

Request\textsubscript{\text{Buyer} \rightarrow \text{Seller}} ;
(Offer\textsubscript{\text{Seller} \rightarrow \text{Buyer}} |
PayDescr\textsubscript{\text{Seller} \rightarrow \text{Bank}} ) ;
Payment\textsubscript{\text{Buyer} \rightarrow \text{Bank}} ;
(O Confirm\textsubscript{\text{Bank} \rightarrow \text{Seller}} |
Receipt\textsubscript{\text{Bank} \rightarrow \text{Buyer}} )

Asynchronous causality safety: ?
Exercise: the initial WS-CDL example

Request\textsubscript{Buyer\rightarrow Seller} ;
(Offer\textsubscript{Seller\rightarrow Buyer} | PayDescr\textsubscript{Seller\rightarrow Bank}) ;
Payment\textsubscript{Buyer\rightarrow Bank} ;
(O Confirm\textsubscript{Bank\rightarrow Seller} | Receipt\textsubscript{Bank\rightarrow Buyer})

Asynchronous causality safety: ✔
Exercise: the initial WS-CDL example

Request_{Buyer \rightarrow Seller} ;
( Offer_{Seller \rightarrow Buyer} | PayDescr_{Seller \rightarrow Bank} ) ;
Payment_{Buyer \rightarrow Bank} ;
( Confirm_{Bank \rightarrow Seller} | Receipt_{Bank \rightarrow Buyer} )

Asynchronous connectedness: ?
Exercise:
the initial WS-CDL example

\[
\text{Request}_{\text{Buyer} \rightarrow \text{Seller}} \;
\left( \text{Offer}_{\text{Seller} \rightarrow \text{Buyer}} \mid \text{PayDescr}_{\text{Seller} \rightarrow \text{Bank}} \right) \;
\text{Payment}_{\text{Buyer} \rightarrow \text{Bank}} \;
\left( \text{Confirm}_{\text{Bank} \rightarrow \text{Seller}} \mid \text{Receipt}_{\text{Bank} \rightarrow \text{Buyer}} \right)
\]

Asynchronous connectedness: ✔
Exercise: try to remove the *Payment*

\[
\text{Request}_{\text{Buyer} \rightarrow \text{Seller}} ;
( \text{Offer}_{\text{Seller} \rightarrow \text{Buyer}} | \\
\text{PayDescr}_{\text{Seller} \rightarrow \text{Bank}} ) ;
( \text{Confirm}_{\text{Bank} \rightarrow \text{Seller}} | \\
\text{Receipt}_{\text{Bank} \rightarrow \text{Buyer}} )
\]
Exercise:
try to remove the Payment

\[
\text{Request}_{\text{Buyer}\rightarrow\text{Seller}}; \\
( \text{Offer}_{\text{Seller}\rightarrow\text{Buyer}} | \\
\text{PayDescr}_{\text{Seller}\rightarrow\text{Bank}} ) ; \\
( \text{Confirm}_{\text{Bank}\rightarrow\text{Seller}} | \\
\text{Receipt}_{\text{Bank}\rightarrow\text{Buyer}} )
\]

Asynchronous unique point of choice: ✔
Exercise: try to remove the *Payment*

\[
\text{Request}_\text{Buyer}\rightarrow\text{Seller} ; \\
(\text{Offer}_\text{Seller}\rightarrow\text{Buyer} \mid \text{PayDescr}_\text{Seller}\rightarrow\text{Bank}) ; \\
(\text{Confirm}_\text{Bank}\rightarrow\text{Seller} \mid \text{Receipt}_\text{Bank}\rightarrow\text{Buyer})
\]

Asynchronous causality safety: ✔
Exercise: try to remove the *Payment*

\[
\text{Request}_{\text{Buyer} \rightarrow \text{Seller}} ; \quad ( \text{Offer}_{\text{Seller} \rightarrow \text{Buyer}} \mid \text{PayDescr}_{\text{Seller} \rightarrow \text{Bank}} ) ; \\
( \text{Confirm}_{\text{Bank} \rightarrow \text{Seller}} \mid \text{Receipt}_{\text{Bank} \rightarrow \text{Buyer}} )
\]

Asynchronous connectedness: ?
Exercise: try to remove the *Payment*

Request_{Buyer\to Seller}; (Offer_{Seller\to Buyer} | PayDescr_{Seller\to Bank}); (Confirm_{Bank\to Seller} | Receipt_{Bank\to Buyer})

Asynchronous connectedness: ✗

In an asynchronous setting there is no ordering guarantee (while it is guaranteed for synchronous communication!)
What to do with choreographies?

- You are now among the greatest worldwide experts in choreographies!
  - so what? ...
- Don’t worry... you didn’t lose your time
  - There is at least one programming language that you can use to realise your distributed systems using choreographies

Choreography programming

- A IOC-like language is used to specify the possible communication protocols used in the system.
- An enriched choreography language (variables, if-then-else, functions, ..) is used to specify the overall system.
- **End-point** programs are obtained by means of projection (written in Jolie).
The approach: graphically
Chor: basic ideas

- Describe the behaviour of processes in a *system*:
  - Each process has a *local state*
  - Processes take part to *multiparty conversations*, tracked as sessions
  - **Sessions** are started through public channels (e.g., URLs).
  - Both sessions and processes can be **dynamically created**
Alice and Bob buy a book together

- Protocol
- Choreography
Alice and Bob buy a book together

<table>
<thead>
<tr>
<th>Buyer</th>
<th>Seller : string; // Ask the price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seller</td>
<td>Buyer : int; // Get the price</td>
</tr>
<tr>
<td>Buyer</td>
<td>Helper : int; // Contribution</td>
</tr>
<tr>
<td>Helper</td>
<td>Seller : { // Choice</td>
</tr>
<tr>
<td></td>
<td>ok: Seller → Helper: string;</td>
</tr>
<tr>
<td></td>
<td>end,</td>
</tr>
<tr>
<td></td>
<td>ko: end</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
Alice and Bob buy a book together

Buyer → Seller : string; // Ask the price
Seller → Buyer : int; // Get the price
Buyer → Helper : int; // Contribution
Helper → Seller : {
    // Choice
    ok: Seller → Helper: string;
    end,
    ko: end
}

- Let a be a public URL (e.g., www.amazon.com) with the protocol above.

alice[Buyer], bob[Helper] start amazon[Seller]: a(k)
Alice and Bob buy a book together

<table>
<thead>
<tr>
<th>Buyer</th>
<th>Seller</th>
<th>string;</th>
<th>// Ask the price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seller</td>
<td>Buyer</td>
<td>int;</td>
<td>// Get the price</td>
</tr>
<tr>
<td>Buyer</td>
<td>Helper</td>
<td>int;</td>
<td>// Contribution</td>
</tr>
<tr>
<td>Helper</td>
<td>Seller</td>
<td>{</td>
<td>// Choice</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ok:</td>
<td>Seller → Helper: string;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>end,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ko:</td>
<td>end</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

```
alice[Buyer], bob[Helper] start amazon[Seller]: a(k);
alice."rabbits" → amazon.book : k
```
Alice and Bob buy a book together

<table>
<thead>
<tr>
<th>Source</th>
<th>Target</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buyer</td>
<td>Seller</td>
<td>string;</td>
<td>Ask the price</td>
</tr>
<tr>
<td>Seller</td>
<td>Buyer</td>
<td>int;</td>
<td>Get the price</td>
</tr>
<tr>
<td>Buyer</td>
<td>Helper</td>
<td>int;</td>
<td>Contribution</td>
</tr>
<tr>
<td>Helper</td>
<td>Seller</td>
<td>{</td>
<td>Choice</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ok:</td>
<td>Seller → Helper: string;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>end,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ko:</td>
<td>end</td>
</tr>
</tbody>
</table>

- Let a be a public URL (e.g., www.amazon.com) with the protocol above.

```
alice[Buyer], bob[Helper] start amazon[Seller]: a(k)
```
Alice and Bob buy a book together

Buyer → Seller : string;  // Ask the price
Seller → Buyer : int;    // Get the price
Buyer → Helper : int;    // Contribution
Helper → Seller : {
    // Choice
    ok: Seller → Helper: string;
    end,
    ko: end
}

alice[Buyer], bob[Helper] start amazon[Seller]: a(k) ;
alice."rabbits" → amazon.book : k ;
amazon.price(book) → alice.price : k
Alice and Bob buy a book together

Buyer → Seller : string; // Ask the price
Seller → Buyer : int; // Get the price
Buyer → Helper : int; // Contribution
Helper → Seller : {
    // Choice
    ok: Seller → Helper: string;
    end,
    ko: end
}

alice[Buyer], bob[Helper] start amazon[Seller] : a(k);
alice."rabbits" → amazon.book : k;
amazon.price(book) → alice.price : k;
alice.(price/2) → bob.contrib : k
Alice and Bob buy a book together

Buyer → Seller : string;  // Ask the price
Seller → Buyer : int;   // Get the price
Buyer → Helper : int;   // Contribution
Helper → Seller :
   {                          // Choice
     ok: Seller → Helper: string;
     end,
     ko: end
   }

alice[Buyer], bob[Helper] start amazon[Seller]: a(k);
alice."rabbits" → amazon.book : k;
amazon.price(book) → alice.price : k;
alice.(price/2) → bob.contrib : k;
if (contrib < 100$) @bob

Condition

Evaluator
Alice and Bob buy a book together

Buyer → Seller : string; // Ask the price
Seller → Buyer : int; // Get the price
Buyer → Helper : int; // Contribution
Helper → Seller : {
    ok: Seller → Helper: string;
    end,
    ko: end
}

alice[Buyer], bob[Helper] start amazon[Seller] : a(k);
alice."rabbits" → amazon.book : k;
amazon.price(book) → alice.price : k;
alice.(price/2) → bob.contrib : k;
if (contrib < 100)$@bob

bob → amazon : k[ok]
Alice and Bob buy a book together

<table>
<thead>
<tr>
<th>Buyer</th>
<th>Seller</th>
<th>string;</th>
<th>// Ask the price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seller</td>
<td>Buyer</td>
<td>int;</td>
<td>// Get the price</td>
</tr>
<tr>
<td>Buyer</td>
<td>Helper</td>
<td>int;</td>
<td>// Contribution</td>
</tr>
<tr>
<td>Helper</td>
<td>Seller</td>
<td>{</td>
<td>// Choice</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ok:</td>
<td>Seller → Helper: string;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ko:</td>
<td>end</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

alice[Buyer], bob[Helper] start amazon[Seller]: a(k);
alice."rabbits" → amazon.book : k;
amazon.price(book) → alice.price : k;
alice.(price/2) → bob.contrib : k;
if (contrib < 100$)@bob
   bob → amazon : k[ok] ;
amazon.text(book) → bob.text : k
Alice and Bob buy a book together

Buyer → Seller : string; // Ask the price
Seller → Buyer : int; // Get the price
Buyer → Helper : int; // Contribution
Helper → Seller : {
    // Choice
    ok: Seller → Helper: string;
    end,
    ko: end
}

alice[Buyer], bob[Helper] start amazon[Seller]: a(k);
alice."rabbits" → amazon.book : k;
amazon.price(book) → alice.price : k;
alice.(price/2) → bob.contrib : k;
if (contrib < 100$)@bob
    bob → amazon : k[ok];
amazon.text(book) → bob.text : k
else
Alice and Bob buy a book together

Buyer → Seller : string; // Ask the price
Seller  → Buyer  : int;  // Get the price
Buyer  → Helper : int;  // Contribution
Helper → Seller : {
    // Choice
  ok: Seller → Helper: string;
    end,
  ko: end
}

alice[Buyer], bob[Helper] start amazon[Seller]: a(k);
a["rabbits"] → amazon.book : k;
amazon.price(book) → alice.price : k;
alice.(price/2) → bob.contrib : k;
if (contrib < 100$)@bob
  bob → amazon : k[ok];
amazon.text(book) → bob.text : k
else
  bob → amazon : k[ko]