Services: when specification meets implementation

Luís Cruz-Filipe
(joint work with A. Lopes)

LaSIGE and
Department of Informatics
FCUL, Lisbon, Portugal

GLOSS seminar
April 1, 2009

Background

- Sensoria, (web) services and service-oriented computing
- SRML: very graphical, funny symbols, rich logic with intuitive semantics
- Conversation Calculus: same intuitive concepts, simple ideas
- A mathematician's view: the same, at the "right" level of abstraction
- . . . and what is the "right" level of abstraction?

Background

- Sensoria, (web) services and service-oriented computing
- SRML: very graphical, funny symbols, rich logic with intuitive semantics
- Conversation Calculus: same intuitive concepts, simple ideas
- A mathematician's view: the same, at the "right" level of abstraction
- . . . and what is the "right" level of abstraction?

Background

- Sensoria, (web) services and service-oriented computing
- SRML: very graphical, funny symbols, rich logic with intuitive semantics
- Conversation Calculus: same intuitive concepts, simple ideas
- A mathematician's view: the same, at the "right" level of abstraction
- . . . and what is the "right" level of abstraction?

Background

- Sensoria, (web) services and service-oriented computing
- SRML: very graphical, funny symbols, rich logic with intuitive semantics
- Conversation Calculus: same intuitive concepts, simple ideas
- A mathematician's view: the same, at the "right" level of abstraction
- . . . and what is the "right" level of abstraction?

Background

- Sensoria, (web) services and service-oriented computing
- SRML: very graphical, funny symbols, rich logic with intuitive semantics
- Conversation Calculus: same intuitive concepts, simple ideas
- A mathematician's view: the same, at the "right" level of abstraction
- . . . and what is the "right" level of abstraction?

Background

- Sensoria, (web) services and service-oriented computing
- SRML: very graphical, funny symbols, rich logic with intuitive semantics
- Conversation Calculus: same intuitive concepts, simple ideas
- A mathematician's view: the same, at the "right" level of abstraction
- . . . and what is the "right" level of abstraction?

Background

- Sensoria, (web) services and service-oriented computing
- SRML: very graphical, funny symbols, rich logic with intuitive semantics
- Conversation Calculus: same intuitive concepts, simple ideas
- A mathematician's view: the same, at the "right" level of abstraction
- ...and what is the "right" level of abstraction?

Background

- Sensoria, (web) services and service-oriented computing
- SRML: very graphical, funny symbols, rich logic with intuitive semantics
- Conversation Calculus: same intuitive concepts, simple ideas
- A mathematician's view: the same, at the "right" level of abstraction
- ...and what is the "right" level of abstraction?

Background

- Sensoria, (web) services and service-oriented computing
- SRML: very graphical, funny symbols, rich logic with intuitive semantics
- Conversation Calculus: same intuitive concepts, simple ideas
- A mathematician's view: the same, at the "right" level of abstraction
- . . . and what is the "right" level of abstraction?

Background

- Sensoria, (web) services and service-oriented computing
- SRML: very graphical, funny symbols, rich logic with intuitive semantics
- Conversation Calculus: same intuitive concepts, simple ideas
- A mathematician's view: the same, at the “right” level of abstraction
- . . . and what is the “right” level of abstraction?

Background

- Sensoria, (web) services and service-oriented computing
- SRML: very graphical, funny symbols, rich logic with intuitive semantics
- Conversation Calculus: same intuitive concepts, simple ideas
- A mathematician's view: the same, at the "right" level of abstraction
- ... and what is the "right" level of abstraction?

Motivation

Goal

Establish a formal correspondence between SRML and the Conversation Calculus.

We don't want a mapping, translation, or even to give semantics of one into the other. Just find that "right" level of abstraction.

Several concepts (on either side) do not have correspondence. We'll just restrict ourselves to the intersection of both systems.

Goal (revised)

Given a concrete specification, establish guidelines to build an implementation that will be sound by construction.

Motivation

Goal

Establish a formal correspondence between SRML and the Conversation Calculus.

We don't want a mapping, translation, or even to give semantics of one into the other. Just find that "right" level of abstraction.

Several concepts (on either side) do not have correspondence. We'll just restrict ourselves to the intersection of both systems.

Goal (revised)

Given a concrete specification, establish guidelines to build an implementation that will be sound by construction.

Motivation

Goal

Establish a formal correspondence between SRML and the Conversation Calculus.

We don't want a mapping, translation, or even to give semantics of one into the other. Just find that "right" level of abstraction.

Several concepts (on either side) do not have correspondence. We'll just restrict ourselves to the intersection of both systems.

Goal (revised)

Given a concrete specification, establish guidelines to build an implementation that will be sound by construction.

Motivation

Goal

Establish a formal correspondence between SRML and the Conversation Calculus.

We don't want a mapping, translation, or even to give semantics of one into the other. Just find that "right" level of abstraction.

Several concepts (on either side) do not have correspondence. We'll just restrict ourselves to the intersection of both systems.

Goal (revised)

Given a concrete specification, establish guidelines to build an implementation that will be sound by construction.

Motivation

Goal

Establish a formal correspondence between SRML and the Conversation Calculus.

We don't want a mapping, translation, or even to give semantics of one into the other. Just find that "right" level of abstraction.

Several concepts (on either side) do not have correspondence. We'll just restrict ourselves to the intersection of both systems.

Goal (revised)

Given a concrete specification, establish guidelines to build an implementation that will be sound by construction.

1. Basics
2. A formal specification in SRML
3. Putting everything together
4. Conclusions

Main idea

Common knowledge

A picture is worth a thousand words.

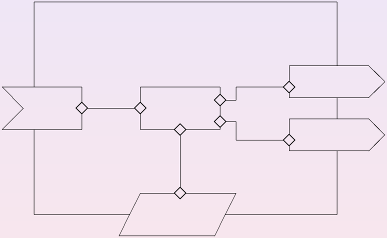
Main idea

Common knowledge

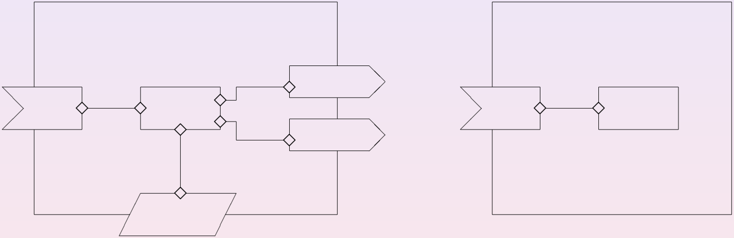
A picture is worth a thousand words.

SRML

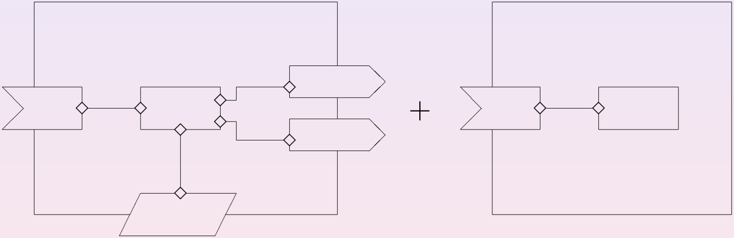
SRML



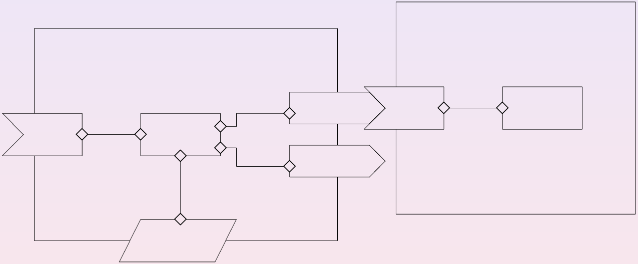
SRML



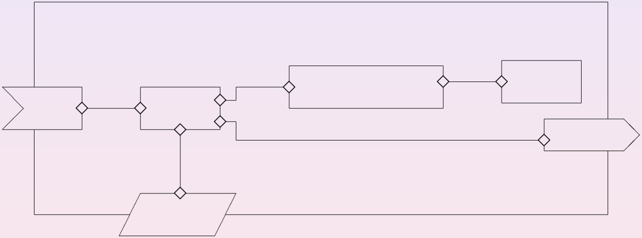
SRML



SRML

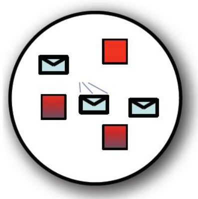


SRML

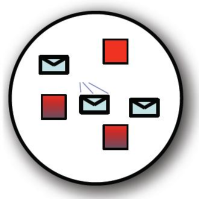


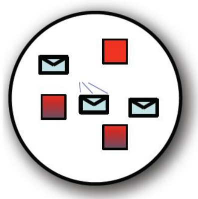
The Conversation Calculus

The Conversation Calculus

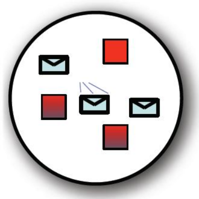


The Conversation Calculus

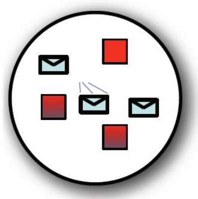




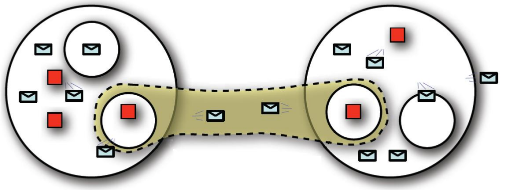
The Conversation Calculus



+



The Conversation Calculus



Message passing

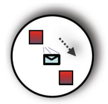
- within the same context ("here")
- to the other endpoint of a session ("there")
- to the enclosing context ("up")

Message passing

- within the same context (“here”)
- to the other endpoint of a session (“there”)
- to the enclosing context (“up”)

Message passing

- within the same context (“here”)

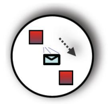


- to the other endpoint of a session (“there”)

- to the enclosing context (“up”)

Message passing

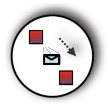
- within the same context ("here")



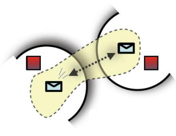
- to the other endpoint of a session ("there")
- to the enclosing context ("up")

Message passing

- within the same context ("here")



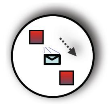
- to the other endpoint of a session ("there")



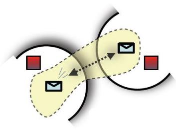
- to the enclosing context ("up")

Message passing

- within the same context ("here")



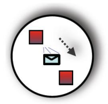
- to the other endpoint of a session ("there")



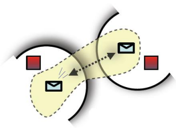
- to the enclosing context ("up")

Message passing

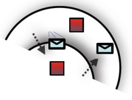
- within the same context (“here”)



- to the other endpoint of a session (“there”)



- to the enclosing context (“up”)



Case study

Consider the following example from the list of SENSORIA case studies.

Example

A travel agent provides a booking service that, upon receiving a request for a flight from a customer, executes the following steps:

Case study

Consider the following example from the list of SENSORIA case studies.

Example

A travel agent provides a booking service that, upon receiving a request for a flight from a customer, executes the following steps:

Case study

Consider the following example from the list of SENSORIA case studies.

Example

A travel agent provides a booking service that, upon receiving a request for a flight from a customer, executes the following steps:

1. contact two different airlines and ask them for prices for the flight;
2. book the cheapest flight;
3. return the flight data to the customer.

Case study

Consider the following example from the list of SENSORIA case studies.

Example

A travel agent provides a booking service that, upon receiving a request for a flight from a customer, executes the following steps:

1. contact two different airlines and ask them for prices for the flight;
2. book the cheapest flight;
3. return the flight data to the customer.

Case study

Consider the following example from the list of SENSORIA case studies.

Example

A travel agent provides a booking service that, upon receiving a request for a flight from a customer, executes the following steps:

1. contact two different airlines and ask them for prices for the flight;
2. book the cheapest flight;
3. return the flight data to the customer.

Case study

Consider the following example from the list of SENSORIA case studies.

Example

A travel agent provides a booking service that, upon receiving a request for a flight from a customer, executes the following steps:

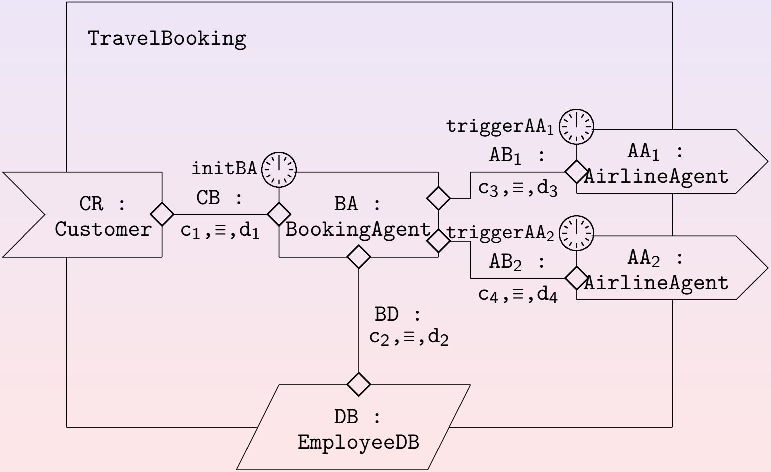
1. contact two different airlines and ask them for prices for the flight;
2. book the cheapest flight;
3. return the flight data to the customer.

Naïve implementation

def travelApp ⇒ (
  instance alphaAir ▷ flightAvails ⇐ (
    in ↑ flightRequestAA(flightData,travelClass).
    out ← flightDetails(flightData,travelClass).
    in ← flightTicket(response,price).
    out ↑ flightResponseAA(response,price).
    (**in** ↑ bookAA().**out** ← bookFlight().
      +**in** ↑ cancelAA().**out** ← cancelFlight())
) | . . . |
  in ← travelRequest(employee,flightData).
  out ↑ employeeTStatusRequest(employee).
  in ↑ employeeTStatusResponse(travelClass).
  out ↓ flightRequestAA(flightAA,travelClass).**out** ↓ flightRequestDA(flightDA,travelClass).
  ((**in** ↓ flightResponseAA(priceAA,flightAA).**out** ↓ Done)|
    (**in** ↓ flightResponseDA(priceDA,flightDA).**out** ↓ Done)|
    (**in** ↓ Done.**in** ↓ Done.
    if (priceAA<priceDA) **then**
      (**out** ← travelResponse(flightAA).**out** ↓ bookAA().**out** ↓ cancelDA())
      else (**out** ← travelResponse(flightDA).**out** ↓ bookDA().**out** ↓ cancelAA())
    )))

Specification: diagram

Specification: diagram



Insight #1

An implementation will consist of several subprocesses running in parallel.

COMPONENTS

BA: BookingAgent

initBA **init**: s=INIT ∧ rec_1=false ∧ rec_2=false

initBA **term**: s=DONE

PROVIDES

CR: Customer

REQUIRES

AA_1: AirlineAgent

$triggerAA_1$ **trigger**: BA.$Flight_1$?

AA_2: AirlineAgent

$triggerAA_2$ **trigger**: BA.$Flight_2$?

USES

DB: EmployeeDB

COMPONENTS

BA: BookingAgent

initBA **init**: $s=INIT \wedge rec_1=false \wedge rec_2=false$

initBA **term**: $s=DONE$

PROVIDES

CR: Customer

REQUIRES

AA_1: AirlineAgent

$triggerAA_1$ **trigger**: $BA.Flight_1$?

AA_2: AirlineAgent

$triggerAA_2$ **trigger**: $BA.Flight_2$?

USES

DB: EmployeeDB

COMPONENTS

BA: BookingAgent

initBA **init**: s=INIT $\wedge$ rec_1=false $\wedge$ rec_2=false

initBA **term**: s=DONE

PROVIDES

CR: Customer

REQUIRES

AA_1: AirlineAgent

$triggerAA_1$ **trigger**: $BA.Flight_1$?

AA_2: AirlineAgent

$triggerAA_2$ **trigger**: $BA.Flight_2$?

USES

DB: EmployeeDB

COMPONENTS

BA: BookingAgent

initBA **init**: s=INIT $\wedge$ rec_1=false $\wedge$ rec_2=false

initBA **term**: s=DONE

PROVIDES

CR: Customer

REQUIRES

AA_1: AirlineAgent

triggerAA_1 **trigger**: BA.$Flight_1$?

AA_2: AirlineAgent

triggerAA_2 **trigger**: BA.$Flight_2$?

USES

DB: EmployeeDB

BUSINESS ROLE `BookingAgent` **is**

```
INTERACTION
  rcv Travel
      emp:  employee
      fl:  flightData
  s&r EmployeeTStatus
      emp:  employee
      cl:  travelClass
  (...)
```

```
ORCHESTRATION
  local s:  [INIT, DBQUERY, WAIT, DONE]
    e:employee, f:flightData, tc:travelClass
    p1:price, rec1:boolean, f1:flight
    p2:price, rec2:boolean, f2:flight

  transition GetData
      triggeredBy Travel🔔
      guardedBy s=INIT
      effects e=Travel.emp ∧ f=Travel.fl ∧
              s'=DBQUERY
      sends EmployeeTStatus🔔 ∧
              EmployeeTStatus.emp=e
```

transition $\mathtt{BookFlight}$

triggeredBy $\mathtt{EmployeeTStatus}$✉
guardedBy $\mathtt{s{=}DBQUERY}$
effects $\mathtt{tc{=}EmployeeTStatus.trav} \wedge \mathtt{s'{=}WAIT}$
sends $\mathtt{Flight}_1$🔔 $\wedge$ $\mathtt{Flight}_2$🔔 $\wedge$
$\mathtt{Flight}_1\mathtt{.flD{=}f} \wedge \mathtt{Flight}_1\mathtt{.cl{=}tc} \wedge$
$\mathtt{Flight}_2\mathtt{.flD{=}f} \wedge \mathtt{Flight}_2\mathtt{.cl{=}tc}$

transition $\mathtt{FlightAnswer}_i$ $(i = 1, 2)$

triggeredBy $\mathtt{Flight}_i$✉
guardedBy $\mathtt{s{=}WAIT} \wedge \neg\mathtt{rec}_i$
effects $\mathtt{rec}_i\mathtt{{=}true} \wedge \mathtt{p}_i\mathtt{{=}Flight}_i\mathtt{.pr} \wedge$
$\mathtt{f}_i\mathtt{{=}Flight}_i\mathtt{.fl}$

transition $\mathtt{ClientCallBack}_i \ (i = 1, 2)$

triggeredBy
guardedBy $\mathtt{s{=}WAIT} \wedge \mathtt{rec}_1 \wedge \mathtt{rec}_2 \wedge \mathtt{p}_i < \mathtt{p}_{3-i}$
effects $\mathtt{S{=}DONE}$
sends $\mathtt{Cancel}_{3-i}$🔔 $\wedge$ $\mathtt{ClientCallBack}$🔔 $\wedge$
$\mathtt{ClientCallBack.fl{=}f}_i \wedge \mathtt{Book}_i$🔔

Insight #2

A correct implementation of a component allows as semantics the transition system specifying its behaviour.

LAYER PROTOCOL `EmployeeDB` **is**

`INTERACTION`

  r&s `EmployeeTStatus`

    🔔 `emp:  employee`

    ✉ `cl:  travelClass`

`BEHAVIOUR`

  initiallyEnabled `EmployeeTStatus`🔔`?`

Insight #3

The system depends upon another service running in the context. This protocol specifies the type of that service.

BUSINESS PROTOCOL `Customer` **is**

```
INTERACTION
  s&r TravelRequest
      emp:  employee
      fd:   flightData
      fl:   flight
BEHAVIOUR
  initiallyEnabled TravelRequest?
```

BUSINESS PROTOCOL `AirlineAgent` is

```
INTERACTION
  r&s FlightDetails
    🔔 data:  flightData
       class:  TravelClass
    ✉ resp:  response
       pr:  price
  rcv Book
  rcv Cancel
BEHAVIOUR
  initiallyEnabled FlightDetails🔔?
  FlightCallBack🔔! enables Book🔔?  until Cancel🔔?
  FlightCallBack🔔! enables Cancel🔔?  until Book🔔?
```

Insight #4

Business protocols are implemented as session endpoints.
The type of a correct implementation should somehow be related to the behaviour specified in the protocol.

	CR: Customer	c_1	CB	d_1		BA: BookingAgent
s&r	TravelRequest	S_1		R_1	**rcv**	Travel
	from	i_1	$\equiv$	i_1		emp
	fd	i_2		i_2		fl
				S_2	**snd**	ClientCallBack
	fl	o_1	$\equiv$	o_1		fl

	BA: BookingAgent	c_2	BD	d_2		DB: EmployeeDB
s&r	EmployeeTStatus	S_1		R_1	**r&s**	EmployeeTStatus
	emp	i_1	$\equiv$	i_1		emp
	trav	o_1		o_1		cl

	CR: Customer	c_1	CB	d_1		BA: BookingAgent
s&r	TravelRequest	S_1		R_1	**rcv**	Travel
🔔	from	i_1	$\equiv$	i_1	🔔	emp
	fd	i_2		i_2		fl
				S_2	**snd**	ClientCallBack
$\boxtimes$	fl	o_1	$\equiv$	o_1	🔔	fl

	BA: BookingAgent	c_2	BD	d_2		DB: EmployeeDB
s&r	EmployeeTStatus	S_1		R_1	**r&s**	EmployeeTStatus
🔔	emp	i_1	$\equiv$	i_1	🔔	emp
$\boxtimes$	trav	o_1		o_1	$\boxtimes$	cl

AA_1: AirlineAgent	c_3	AB_1	d_3	BA: BookingAgent
r&s FlightDetails	R_1		S_1	**s&r** $Flight_1$
🔔 data	i_1		i_1	🔔 flD
class	i_2	$\equiv$	i_2	cl
✉ resp	o_1		o_1	✉ fl
pr	o_2		o_2	pr
rcv Book	R_2	$\equiv$	S_2	**snd** $Book_1$
rcv Cancel	R_3	$\equiv$	S_3	**snd** $Cancel_1$

Wire AB_2 is similar.

AA_1: AirlineAgent	c_3	AB_1	d_3	BA: BookingAgent
r&s FlightDetails	R_1		S_1	**s&r** $Flight_1$
🔔 data	i_1		i_1	🔔 flD
class	i_2	$\equiv$	i_2	cl
✉ resp	o_1		o_1	✉ fl
pr	o_2		o_2	pr
rcv Book	R_2	$\equiv$	S_2	**snd** $Book_1$
rcv Cancel	R_3	$\equiv$	S_3	**snd** $Cancel_1$

Wire $\mathtt{AB}_2$ is similar.

Simplification is the key

Simplification: assume wires do not have any computational content: they just change some names.

Idea: encode the name changes in the remaining processes, forget the wire.

Wrong insight

Wires are coded in the implementation of the remaining blocks.

There's some unpleasant arbitrariness here. . .

All wires have some computational content. . . these ones do!

Simplification is the key

Simplification: assume wires do not have any computational content: they just change some names.

Idea: encode the name changes in the remaining processes, forget the wire.

Wrong insight

Wires are coded in the implementation of the remaining blocks.

There's some unpleasant arbitrariness here...

All wires have some computational content... these ones do!

Simplification is the key

Simplification: assume wires do not have any computational content: they just change some names.

Idea: encode the name changes in the remaining processes, forget the wire.

Wrong insight

Wires are coded in the implementation of the remaining blocks.

There's some unpleasant arbitrariness here. . .

All wires have some computational content. . . these ones do!

Simplification is the key

Simplification: assume wires do not have any computational content: they just change some names.

Idea: encode the name changes in the remaining processes, forget the wire.

Wrong insight

Wires are coded in the implementation of the remaining blocks.

There's some unpleasant arbitrariness here. . .

All wires have some computational content. . . these ones do!

Simplification is **not** the key

Simplification: assume wires do not have any computational content: they just change some names.

Idea: encode the name changes in the remaining processes, forget the wire.

Wrong insight

Wires are coded in the implementation of the remaining blocks.

There's some unpleasant arbitrariness here...

All wires have some computational content... these ones do!

Simplification is **not** the key

Simplification: assume wires do not have any computational content: they just change some names.

Idea: encode the name changes in the remaining processes, forget the wire.

Wrong insight

Wires are coded in the implementation of the remaining blocks.

There's some unpleasant arbitrariness here...

All wires have some computational content... these ones do!

Simplification is **not** the key

Simplification: assume wires do not have any computational content: they just change some names.

Idea: encode the name changes in the remaining processes, forget the wire.

Wrong insight

Wires are coded in the implementation of the remaining blocks.

There's some unpleasant arbitrariness here...

All wires have some computational content... these ones do!

Simplification is **not** the key

Simplification: assume wires do not have any computational content: they just change some names.

Idea: encode the name changes in the remaining processes, forget the wire.

Wrong insight

Wires are coded in the implementation of the remaining blocks.

There's some unpleasant arbitrariness here. . .

All wires have some computational content. . . these ones do!

How about...?

Can we see a wire as a process?

A (simple) wire reads messages from one endpoint and posts them at the other endpoint.

A (simple) wire passes messages across contexts.

How about...?

Can we see a wire as a process?

A (simple) wire reads messages from one endpoint and posts them at the other endpoint.

A (simple) wire passes messages across contexts.

How about...?

Can we see a wire as a process?

A (simple) wire reads messages from one endpoint and posts them at the other endpoint.

A (simple) wire passes messages across contexts.

How about...?

Can we see a wire as a process?

A (simple) wire reads messages from one endpoint and posts them at the other endpoint.

A (simple) wire passes messages across contexts.

Insight #5

Wires are processes just like other components.

What have we learned?

- Components yield processes.
- Wires yield processes.
- Other protocols require existence of processes with specific behaviour (type).

What have we learned?

- Components yield processes.
- Wires yield processes.
- Other protocols require existence of processes with specific behaviour (type).

What have we learned?

- Components yield processes.
- Wires yield processes.
- Other protocols require existence of processes with specific behaviour (type).

What have we learned?

- Components yield processes.
- Wires yield processes.
- Other protocols require existence of processes with specific behaviour (type).

Messages and names

Definition

A renaming function is an injective mapping $\sigma : \mathcal{L}_1 \to \mathcal{L}_2$ between two sets of CC labels.

Henceforth we will assume a canonical renaming function that takes SRML event name M in module `X` to the CC label X_M.

So e.g. message $Flight_1$🔔 in module `BA` becomes label BA_Flight_1🔔.

Observe that 🔔 and ✉ are two different events assigned to the same message in SRML, but in CC they are just syntactic symbols.

Messages and names

Definition

A renaming function is an injective mapping $\sigma : \mathcal{L}_1 \to \mathcal{L}_2$ between two sets of CC labels.

Henceforth we will assume a canonical renaming function that takes SRML event name M in module `X` to the CC label X_M.

So e.g. message $Flight_1$🔔 in module `BA` becomes label BA_Flight_1🔔.

Observe that 🔔 and ✉ are two different events assigned to the same message in SRML, but in CC they are just syntactic symbols.

Messages and names

Definition

A renaming function is an injective mapping $\sigma : \mathcal{L}_1 \to \mathcal{L}_2$ between two sets of CC labels.

Henceforth we will assume a canonical renaming function that takes SRML event name M in module `X` to the CC label X_M.

So e.g. message $Flight_1$🔔 in module `BA` becomes label BA_Flight_1🔔.

Observe that 🔔 and ✉ are two different events assigned to the same message in SRML, but in CC they are just syntactic symbols.

Messages and names

Definition

A renaming function is an injective mapping $\sigma : \mathcal{L}_1 \to \mathcal{L}_2$ between two sets of CC labels.

Henceforth we will assume a canonical renaming function that takes SRML event name M in module `X` to the CC label X_M. So e.g. message Flight_1🔔 in module `BA` becomes label $\mathit{BA\_Flight}_1$🔔.

Observe that 🔔 and ✉ are two different events assigned to the same message in SRML, but in CC they are just syntactic symbols.

Messages and names

Definition

A renaming function is an injective mapping $\sigma : \mathcal{L}_1 \to \mathcal{L}_2$ between two sets of CC labels.

Henceforth we will assume a canonical renaming function that takes SRML event name M in module `X` to the CC label X_M.
So e.g. message $Flight_1$🔔 in module `BA` becomes label BA_Flight_1🔔.
Observe that 🔔 and ✉ are two different events assigned to the same message in SRML, but in CC they are just syntactic symbols.

Definition

An orchestration O and a process P are *synchronized* at state s if:

Definition

An orchestration O and a process P are *synchronized* at state s if:

1.
2.
    1.
    2.
    3.

Definition

An orchestration O and a process P are *synchronized* at state s if:

1. For every transition whose **guardedBy** condition holds in s, there exists a sequencialization $\alpha_1, \ldots, \alpha_k$ of its **sends** messages such that $P \xrightarrow{\tau}{}^{*} \xrightarrow{\alpha}{}^{?} \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_k} P'$, where α is an action triggering the transition when it exists.
2.
   1.
   2.
   3.

Definition

An orchestration O and a process P are *synchronized* at state s if:

1. For every transition whose **guardedBy** condition holds in s, $P \xrightarrow{\tau}^{*} \xrightarrow{\alpha}^{?} \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_k} P'$.
2. For every action $\alpha \neq \tau$, if $P \xrightarrow{\alpha} Q$, then:
    1.
    2.
    3.

Definition

An orchestration O and a process P are *synchronized* at state s if:

1. For every transition whose **guardedBy** condition holds in s, $P \xrightarrow{\tau}^* \xrightarrow{\alpha}? \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_k} P'$.
2. For every action $\alpha \neq \tau$, if $P \xrightarrow{\alpha} Q$, then:
    1. there exists a transition in O **triggeredBy** α whose **guardedBy** condition holds in s;
    2.
    3.

Definition

An orchestration O and a process P are *synchronized* at state s if:

1. For every transition whose **guardedBy** condition holds in s, $P \xrightarrow{\tau}{}^{*} \xrightarrow{\alpha}{}^{?} \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_k} P'$.
2. For every action $\alpha \neq \tau$, if $P \xrightarrow{\alpha} Q$, then:
    1. there exists a transition in O **triggeredBy** α whose **guardedBy** condition holds in s;
    2. for every such transition, there exist a sequencialization of its **sends** messages such that $Q \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_k} P'$;
    3.

Definition

An orchestration O and a process P are *synchronized* at state s if:

1. For every transition whose **guardedBy** condition holds in s, $P \xrightarrow{\tau}^* \xrightarrow{\alpha}^? \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_k} P'$.
2. For every action $\alpha \neq \tau$, if $P \xrightarrow{\alpha} Q$, then:
    1. there exists a transition in O **triggeredBy** α whose **guardedBy** condition holds in s;
    2. $Q \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_k} P'$
    3. if $Q \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n}$, then $\alpha_1, \ldots, \alpha_k$ are a sequencialization of the **sends** messages of such a transition for some $k \leq n$.

Definition

An orchestration O and a process P are *synchronized* at state s if:

1. For every transition whose **guardedBy** condition holds in s, $P \xrightarrow{\tau}^* \xrightarrow{\alpha}? \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_k} P'$.
2. For every action $\alpha \neq \tau$, if $P \xrightarrow{\alpha} Q$, then:
    1. there exists a transition in O **triggeredBy** α whose **guardedBy** condition holds in s;
    2. $Q \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_k} P'$
    3. if $Q \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n}$, then $\alpha_1, \ldots, \alpha_k$ are a sequencialization of the **sends** messages of such a transition for some $k \leq n$.

All messages are received/sent in the here direction.

Definition

If P and O are synchronized at state s, then the possible evolutions of s and P according to O are defined as follows.

- If $P \xrightarrow{\tau} P'$, then $\langle s, P' \rangle$ is a possible evolution of $\langle s, P \rangle$.
- For every transition whose **guardedBy** condition holds in s and every sequencialization $\alpha_1, \ldots, \alpha_k$ of its **sends** messages such that $P \xrightarrow{\tau}{}^{*} \xrightarrow{\alpha}{}^{?} \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_k} P'$ (where α is an action triggering the transition when it exists), the pair $\langle s', P' \rangle$ (where s' is obtained by applying the **effects** of the transition to s) is a possible evolution of $\langle s, P \rangle$.

Definition

If P and $\mathtt{O}$ are synchronized at state s, then the possible evolutions of s and P according to $\mathtt{O}$ are defined as follows.

- If $P \xrightarrow{\tau} P'$, then $\langle s, P' \rangle$ is a possible evolution of $\langle s, P \rangle$.
- For every transition whose **guardedBy** condition holds in s and every sequencialization $\alpha_1, \ldots, \alpha_k$ of its **sends** messages such that $P \xrightarrow{\tau}{}^{*} \xrightarrow{\alpha}{}^{?} \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_k} P'$ (where α is an action triggering the transition when it exists), the pair $\langle s', P' \rangle$ (where s' is obtained by applying the **effects** of the transition to s) is a possible evolution of $\langle s, P \rangle$.

Definition

If P and $\mathtt{O}$ are synchronized at state s, then the possible evolutions of s and P according to $\mathtt{O}$ are defined as follows.

- If $P \xrightarrow{\tau} P'$, then $\langle s, P' \rangle$ is a possible evolution of $\langle s, P \rangle$.
- For every transition whose **guardedBy** condition holds in s and every sequencialization $\alpha_1, \ldots, \alpha_k$ of its **sends** messages such that $P \xrightarrow{\tau}^* \xrightarrow{\alpha}^? \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_k} P'$ (where α is an action triggering the transition when it exists), the pair $\langle s', P' \rangle$ (where s' is obtained by applying the **effects** of the transition to s) is a possible evolution of $\langle s, P \rangle$.

Definition

- Orchestration O can simulate process P from state s if O and P are synchronized at state s and if O can simulate P' from s' for every possible evolution $\langle s', P' \rangle$ of P from s according to O.
- Process P implements orchestration O if O can simulate P from any initial state of O.
- Process $\rho_{\mathrm{B}}(P)$ implements component B with orchestration O if P implements O.

Definition

- Orchestration $\mathtt{O}$ can simulate process P from state s if $\mathtt{O}$ and P are synchronized at state s and if $\mathtt{O}$ can simulate P' from s' for every possible evolution $\langle s', P' \rangle$ of P from s according to $\mathtt{O}$.
- Process P implements orchestration $\mathtt{O}$ if $\mathtt{O}$ can simulate P from any initial state of $\mathtt{O}$.
- Process $\rho_{\mathtt{B}}(P)$ implements component $\mathtt{B}$ with orchestration $\mathtt{O}$ if P implements $\mathtt{O}$.

Definition

- Orchestration $\mathtt{O}$ can simulate process P from state s if $\mathtt{O}$ and P are synchronized at state s and if $\mathtt{O}$ can simulate P' from s' for every possible evolution $\langle s', P' \rangle$ of P from s according to $\mathtt{O}$.
- Process P implements orchestration $\mathtt{O}$ if $\mathtt{O}$ can simulate P from any initial state of $\mathtt{O}$.
- Process $\rho_{\mathtt{B}}(P)$ implements component $\mathtt{B}$ with orchestration $\mathtt{O}$ if P implements $\mathtt{O}$.

Definition

- Orchestration $\mathtt{O}$ can simulate process P from state s if $\mathtt{O}$ and P are synchronized at state s and if $\mathtt{O}$ can simulate P' from s' for every possible evolution $\langle s', P' \rangle$ of P from s according to $\mathtt{O}$.
- Process P implements orchestration $\mathtt{O}$ if $\mathtt{O}$ can simulate P from any initial state of $\mathtt{O}$.
- Process $\rho_{\mathtt{B}}(P)$ implements component $\mathtt{B}$ with orchestration $\mathtt{O}$ if P implements $\mathtt{O}$.

Example

in ↓ Travel🔔 (e,f).
out ↓ EmployeeTStatus🔔 (e).
in ↓ EmployeeTStatus✉ (tc).
out ↓ Flight$_1$🔔 (f,tc).
out ↓ Flight$_2$🔔 (f,tc).
(
(**in** ↓ Flight$_1$✉ (p$_1$,f$_1$).**out** ↓ Done)|
(**in** ↓ Flight$_2$✉ (p$_2$,f$_2$).**out** ↓ Done)|
(**in** ↓ Done.**in** ↓ Done.
if $(p_1 < p_2)$ **then**
(**out** ↓ ClientCallBack🔔 (f$_1$).**out** ↓ Book$_1$🔔 ().**out** ↓ Cancel$_2$🔔 ())
else (**out** ↓ ClientCallBack🔔 (f$_2$).**out** ↓ Book$_2$🔔 ().**out** ↓ Cancel$_1$🔔 ())
))

Types in the Conversation Calculus

SRML separates behaviour from location.

Restriction to non-recursive behavioural types:

Message types M consist of:

- a polarity !, ? or τ;
- a direction $\uparrow$, $\downarrow$ or $\leftarrow$;
- an event from the SRML specification;
- the (atomic) types of its arguments.

$$B ::= \mathbf{0} \;[]\; M.B \;[]\; B \mid B \;[]\; \oplus\{M.B; \ldots ; M.B\} \;[]\; \&\{M.B; \ldots ; M.B\}$$

Types in the Conversation Calculus

SRML separates behaviour from location.

Restriction to non-recursive behavioural types:

Message types M consist of:

- a polarity !, ? or τ;
- a direction $\uparrow$, $\downarrow$ or $\leftarrow$;
- an event from the SRML specification;
- the (atomic) types of its arguments.

$$B ::= \mathbf{0} \mid M.B \mid B \mid B \mid \oplus\{M.B; \ldots; M.B\} \mid \&\{M.B; \ldots; M.B\}$$

Types in the Conversation Calculus

SRML separates behaviour from location.

Restriction to non-recursive behavioural types:

Message types M consist of:

- a polarity !, ? or τ;
- a direction $\uparrow$, $\downarrow$ or $\leftarrow$;
- an event from the SRML specification;
- the (atomic) types of its arguments.

$$B ::= \mathbf{0} \;[\,]\; M.B \;[\,]\; B \mid B \;[\,]\; \oplus\{M.B;\ldots;M.B\} \;[\,]\; \&\{M.B;\ldots;M.B\}$$

Types in the Conversation Calculus

SRML separates behaviour from location.

Restriction to non-recursive behavioural types:

Message types M consist of:

- a polarity !, ? or τ;
- a direction $\uparrow$, $\downarrow$ or $\leftarrow$;
- an event from the SRML specification;
- the (atomic) types of its arguments.

$$B ::= \mathbf{0} \;\Box\; M.B \;\Box\; B \mid B \;\Box\; \oplus\{M.B; \ldots ; M.B\} \;\Box\; \&\{M.B; \ldots ; M.B\}$$

Types in the Conversation Calculus

SRML separates behaviour from location.

Restriction to non-recursive behavioural types:

Message types M consist of:

- a polarity !, ? or τ;
- a direction $\uparrow$, $\downarrow$ or $\leftarrow$;
- an event from the SRML specification;
- the (atomic) types of its arguments.

$$B ::= \mathbf{0} \;[\,]\; M.B \;[\,]\; B \mid B \;[\,]\; \oplus\{M.B; \ldots ; M.B\} \;[\,]\; \&\{M.B; \ldots ; M.B\}$$

Types in the Conversation Calculus

SRML separates behaviour from location.

Restriction to non-recursive behavioural types:

Message types M consist of:

- a polarity !, ? or τ;
- a direction $\uparrow$, $\downarrow$ or $\leftarrow$;
- an event from the SRML specification;
- the (atomic) types of its arguments.

$$B ::= \mathbf{0} \;[]\; M.B \;[]\; B \mid B \;[]\; \oplus\{M.B;\ldots;M.B\} \;[]\; \&\{M.B;\ldots;M.B\}$$

Types in the Conversation Calculus

SRML separates behaviour from location.

Restriction to non-recursive behavioural types:

Message types M consist of:

- a polarity !, ? or τ;
- a direction $\uparrow$, $\downarrow$ or $\leftarrow$;
- an event from the SRML specification;
- the (atomic) types of its arguments.

$$B ::= \mathbf{0} \;[]\; M.B \;[]\; B \mid B \;[]\; \oplus\{M.B; \ldots; M.B\} \;[]\; \&\{M.B; \ldots; M.B\}$$

Types in the Conversation Calculus

SRML separates behaviour from location.

Restriction to non-recursive behavioural types:

Message types M consist of:

- a polarity !, ? or τ;
- a direction $\uparrow$, $\downarrow$ or $\leftarrow$;
- an event from the SRML specification;
- the (atomic) types of its arguments.

$$B ::= \mathbf{0} \mid M.B \mid B \,|\, B \mid \oplus\{M.B; \ldots ; M.B\} \mid \&\{M.B; \ldots ; M.B\}$$

Types in the Conversation Calculus

SRML separates behaviour from location.

Restriction to non-recursive behavioural types:

Message types M consist of:

- a polarity !, ? or τ;
- a direction $\uparrow$, $\downarrow$ or $\leftarrow$;
- an event from the SRML specification;
- the (atomic) types of its arguments.

$$B ::= \mathbf{0} \mathbin{[]} M.B \mathbin{[]} B \mid B \mathbin{[]} \oplus\{M.B; \ldots; M.B\} \mathbin{[]} \&\{M.B; \ldots; M.B\}$$

Behaviour trees

A type in the Conversation Calculus generates a tree of possible traces, containing all sequences of messages allowed by that type.

A node n in that tree may satisfy the following formulas:

- event e is satisfied ($n \models e$)
- event e is enabled ($n \models en(e)$)
- event e is enabled until e' ($n \models en(e)\mathbf{U}e'$)
- event e is ensured ($n \models \Diamond e$)

Behaviour trees

A type in the Conversation Calculus generates a tree of possible traces, containing all sequences of messages allowed by that type.

A node n in that tree may satisfy the following formulas:

- event e is satisfied ($n \vDash e$)
- event e is enabled ($n \vDash en(e)$)
- event e is enabled until e' ($n \vDash en(e)\mathbf{U}e'$)
- event e is ensured ($n \vDash \Diamond e$)

Behaviour trees

A type in the Conversation Calculus generates a tree of possible traces, containing all sequences of messages allowed by that type.

A node n in that tree may satisfy the following formulas:

- event e is satisfied ($n \vDash e$)
- event e is enabled ($n \vDash en(e)$)
- event e is enabled until e' ($n \vDash en(e)\mathbf{U}e'$)
- event e is ensured ($n \vDash \Diamond e$)

Behaviour trees

A type in the Conversation Calculus generates a tree of possible traces, containing all sequences of messages allowed by that type.

A node n in that tree may satisfy the following formulas:

- event e is satisfied ($n \vDash e$)
- event e is enabled ($n \vDash en(e)$)
- event e is enabled until e' ($n \vDash en(e)\mathbf{U}e'$)
- event e is ensured ($n \vDash \Diamond e$)

Behaviour trees

A type in the Conversation Calculus generates a tree of possible traces, containing all sequences of messages allowed by that type.

A node n in that tree may satisfy the following formulas:

- event e is satisfied ($n \vDash e$)
- event e is enabled ($n \vDash en(e)$)
- event e is enabled until e' ($n \vDash en(e)\mathbf{U}e'$)
- event e is ensured ($n \vDash \Diamond e$)

Behaviour trees

A type in the Conversation Calculus generates a tree of possible traces, containing all sequences of messages allowed by that type.

A node n in that tree may satisfy the following formulas:

- event e is satisfied ($n \vDash e$)
- event e is enabled ($n \vDash en(e)$)
- event e is enabled until e' ($n \vDash en(e)\mathbf{U}e'$)
- event e is ensured ($n \vDash \Diamond e$)

Behaviour trees

A type in the Conversation Calculus generates a tree of possible traces, containing all sequences of messages allowed by that type.

A node n in that tree may satisfy the following formulas:

- event e is satisfied ($n \models e$)
- event e is enabled ($n \models en(e)$)
- event e is enabled until e' ($n \models en(e)\mathbf{U}e'$)
- event e is ensured ($n \models \Diamond e$)

Allowed behaviours in SRML

For event e, allow E to be either $e?$ or $e!$. SP stands for a sequence of $E_1, \ldots, E_k$.

$$\varphi ::= \textbf{initiallyEnabled}\ e? \mid E\ \textbf{enables}\ e? \mid E\ \textbf{enables}\ e?\ \textbf{until}\ E \mid SP\ \textbf{ensures}\ e!$$

- Discarding events is not capturable in CC.
- Comparison of terms cannot be analyzed from the type.

Allowed behaviours in SRML

For event e, allow E to be either $e?$ or $e!$. SP stands for a sequence of $E_1, \ldots, E_k$.

$$\begin{aligned}\varphi ::= &\ \textbf{initiallyEnabled}\ e? \ \square\ E\ \textbf{enables}\ e? \ \square \\ &\ \square\ E\ \textbf{enables}\ e?\ \textbf{until}\ E \ \square\ SP\ \textbf{ensures}\ e!\end{aligned}$$

- Discarding events is not capturable in CC.
- Comparison of terms cannot be analyzed from the type.

Allowed behaviours in SRML

For event e, allow E to be either $e?$ or $e!$. SP stands for a sequence of $E_1, \ldots, E_k$.

$$\varphi ::= \textbf{initiallyEnabled } e? \ [] \ E \textbf{ enables } e? \ [] \ E \textbf{ enables } e? \textbf{ until } E \ [] \ SP \textbf{ ensures } e!$$

- Discarding events is not capturable in CC.
- Comparison of terms cannot be analyzed from the type.

Allowed behaviours in SRML

For event e, allow E to be either $e?$ or $e!$. SP stands for a sequence of $E_1, \ldots, E_k$.

$$\varphi ::= \textbf{initiallyEnabled}\ e? \ [] \ E\ \textbf{enables}\ e? \ [] \ E\ \textbf{enables}\ e?\ \textbf{until}\ E \ [] \ SP\ \textbf{ensures}\ e!$$

- Discarding events is not capturable in CC.
- Comparison of terms cannot be analyzed from the type.

Allowed behaviours in SRML

For event e, allow E to be either $e?$ or $e!$. SP stands for a sequence of $E_1, \ldots, E_k$.

$$\varphi ::= \textbf{initiallyEnabled } e? \ [] \ E \textbf{ enables } e? \ [] \ E \textbf{ enables } e? \textbf{ until } E \ [] \ SP \textbf{ ensures } e!$$

- Discarding events is not capturable in CC.
- Comparison of terms cannot be analyzed from the type.

Allowed behaviours in SRML

For event e, allow E to be either $e?$ or $e!$. SP stands for a sequence of $E_1, \ldots, E_k$.

$$\varphi ::= \textbf{initiallyEnabled}\ e? \ [] \ E\ \textbf{enables}\ e? \ [] \ E\ \textbf{enables}\ e?\ \textbf{until}\ E \ [] \ SP\ \textbf{ensures}\ e!$$

- Discarding events is not capturable in CC.
- Comparison of terms cannot be analyzed from the type.

Explicit behaviours

SRML assumes implicit behaviour associated with some message types.

Definition

The explicit behaviour associated to an SRML behaviour B is obtained by adding to B the formulas:

- (e🔔? **ensures** e✉!) for every **r&s** message e
- (e🔔! **enables** e✉?) for every **s&r** message e

Explicit behaviours

SRML assumes implicit behaviour associated with some message types.

Definition

The explicit behaviour associated to an SRML behaviour B is obtained by adding to B the formulas:

- (e🔔? **ensures** e✉!) for every **r&s** message e
- (e🔔! **enables** e✉?) for every **s&r** message e

Explicit behaviours

SRML assumes implicit behaviour associated with some message types.

Definition

The explicit behaviour associated to an SRML behaviour B is obtained by adding to B the formulas:

- (e? **ensures** e✉!) for every **r&s** message e
- (e! **enables** e✉?) for every **s&r** message e

Explicit behaviours

SRML assumes implicit behaviour associated with some message types.

Definition

The explicit behaviour associated to an SRML behaviour B is obtained by adding to B the formulas:

- (e🔔? **ensures** e✉!) for every **r&s** message e
- (e🔔! **enables** e✉?) for every **s&r** message e

Explicit behaviours

SRML assumes implicit behaviour associated with some message types.

Definition

The explicit behaviour associated to an SRML behaviour B is obtained by adding to B the formulas:

- (e🔔? **ensures** e✉!) for every **r&s** message e
- (e🔔! **enables** e✉?) for every **s&r** message e

Compliance

- A behavioural type B complies with an SRML behavioural formula φ in the following situations.

$B \models$ **initiallyEnabled** $e?$

if $\varepsilon \models_{\mathcal{T}_B} en(e?)$ with ε the root of $\mathcal{T}_B$

$B \models a$ **enables** $e?$

if for all $n \in \mathcal{T}_B$, if $n \models_{\mathcal{T}_B} a$ then $n \models_{\mathcal{T}_B} en(e?)$

$B \models a$ **enables** $e?$ **until** b

if for all $n \in \mathcal{T}_B$, if $n \models_{\mathcal{T}_B} a$ then $n \models_{\mathcal{T}_B} en(e?)\mathbf{U}b$

$B \models a$ **ensures** $e!$

if for all $n \in \mathcal{T}_B$, if $n \models_{\mathcal{T}_B} a$ then $n \models_{\mathcal{T}_B} (\Diamond e!)$

- A behavioural type B complies with an SRML behaviour B if

Compliance

- A behavioural type B complies with an SRML behavioural formula φ in the following situations.

$B \vDash$ **initiallyEnabled** $e?$

if $\varepsilon \vDash_{\mathcal{T}_B} en(e?)$ with ε the root of $\mathcal{T}_B$

$B \vDash a$ **enables** $e?$

if for all $n \in \mathcal{T}_B$, if $n \vDash_{\mathcal{T}_B} a$ then $n \vDash_{\mathcal{T}_B} en(e?)$

$B \vDash a$ **enables** $e?$ **until** b

if for all $n \in \mathcal{T}_B$, if $n \vDash_{\mathcal{T}_B} a$ then $n \vDash_{\mathcal{T}_B} en(e?)\mathbf{U}b$

$B \vDash a$ **ensures** $e!$

if for all $n \in \mathcal{T}_B$, if $n \vDash_{\mathcal{T}_B} a$ then $n \vDash_{\mathcal{T}_B} (\Diamond e!)$

- A behavioural type B complies with an SRML behaviour B if

Compliance

- A behavioural type B complies with an SRML behavioural formula φ in the following situations.

$$B \vDash \textbf{initiallyEnabled } e?$$

if $\varepsilon \vDash_{\mathcal{T}_B} en(e?)$ with ε the root of $\mathcal{T}_B$

$$B \vDash a \textbf{ enables } e?$$

if for all $n \in \mathcal{T}_B$, if $n \vDash_{\mathcal{T}_B} a$ then $n \vDash_{\mathcal{T}_B} en(e?)$

$$B \vDash a \textbf{ enables } e? \textbf{ until } b$$

if for all $n \in \mathcal{T}_B$, if $n \vDash_{\mathcal{T}_B} a$ then $n \vDash_{\mathcal{T}_B} en(e?)\mathbf{U}b$

$$B \vDash a \textbf{ ensures } e!$$

if for all $n \in \mathcal{T}_B$, if $n \vDash_{\mathcal{T}_B} a$ then $n \vDash_{\mathcal{T}_B} (\Diamond e!)$

- A behavioural type B complies with an SRML behaviour $\mathtt{B}$ if

Minimal compliance

Two extra conditions:

- no spurious behaviour;
- all communication is along the right direction: “there” for PROVIDES/REQUIRES interfaces, “up” for USES.

Minimal compliance

Two extra conditions:

- no spurious behaviour;
- all communication is along the right direction: “there” for PROVIDES/REQUIRES interfaces, “up” for USES.

Minimal compliance

Two extra conditions:

- no spurious behaviour;
- all communication is along the right direction: "there" for PROVIDES/REQUIRES interfaces, "up" for USES.

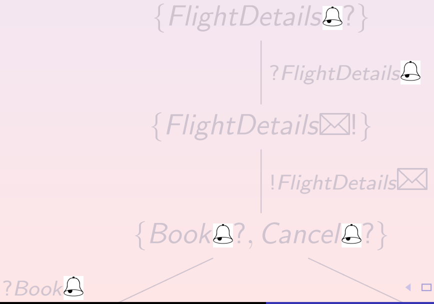
Example: airline protocol

$$B_A \triangleq ?\leftarrow FlightDetails\bell(D,C).!\leftarrow FlightDetails\Letter(R,P).$$
$$\&\{?\leftarrow Book\bell();?\leftarrow Cancel\bell()\}$$

$$\{FlightDetails\bell?\}$$
$$\Big\downarrow\ ?FlightDetails\bell$$
$$\{FlightDetails\Letter!\}$$
$$\Big\downarrow\ !FlightDetails\Letter$$
$$\{Book\bell?, Cancel\bell?\}$$
$$?Book\bell \qquad\qquad ?Cancel\bell$$

Example: airline protocol

$$B_A \triangleq ? \leftarrow \mathit{FlightDetails}\Bell(D, C).! \leftarrow \mathit{FlightDetails}\boxtimes(R, P).$$
$$\& \{? \leftarrow \mathit{Book}\Bell(); ? \leftarrow \mathit{Cancel}\Bell()\}$$



Example: airline protocol

$$\begin{aligned} B_A \triangleq\, ? \leftarrow \mathit{FlightDetails}\Bell(D, C).! \leftarrow \mathit{FlightDetails}\Letter(R, P). \\ \& \{? \leftarrow \mathit{Book}\Bell(); ? \leftarrow \mathit{Cancel}\Bell()\} \end{aligned}$$

$$\begin{tikzcd}[column sep=small]
& \{\mathit{FlightDetails}\Bell?\} \arrow[d, dash, "?\mathit{FlightDetails}\Bell"] & \\
& \{\mathit{FlightDetails}\Letter!\} \arrow[d, dash, "!\mathit{FlightDetails}\Letter"] & \\
& \{\mathit{Book}\Bell?, \mathit{Cancel}\Bell?\} \arrow[dl, dash, "?\mathit{Book}\Bell"'] \arrow[dr, dash, "?\mathit{Cancel}\Bell"] & \\
{} & & {}
\end{tikzcd}$$

Simple wires just relay messages.

Wires to the orchestrator are implemented at the other endpoint, following its protocol, and relay their messages to the orchestrator's context.

Wires between two non-orchestrators are implemented at *both* endpoints and relay their messages to the orchestrators' context, using the wire's name as identifier.

Wires between orchestrators consist simply of the parallel composition of all messages being relayed.

Simple wires just relay messages.

Wires to the orchestrator are implemented at the other endpoint, following its protocol, and relay their messages to the orchestrator's context.

Wires between two non-orchestrators are implemented at *both* endpoints and relay their messages to the orchestrators' context, using the wire's name as identifier.

Wires between orchestrators consist simply of the parallel composition of all messages being relayed.

Simple wires just relay messages.

Wires to the orchestrator are implemented at the other endpoint, following its protocol, and relay their messages to the orchestrator's context.

Wires between two non-orchestrators are implemented at *both* endpoints and relay their messages to the orchestrators' context, using the wire's name as identifier.

Wires between orchestrators consist simply of the parallel composition of all messages being relayed.

Simple wires just relay messages.

Wires to the orchestrator are implemented at the other endpoint, following its protocol, and relay their messages to the orchestrator's context.

Wires between two non-orchestrators are implemented at *both* endpoints and relay their messages to the orchestrators' context, using the wire's name as identifier.

Wires between orchestrators consist simply of the parallel composition of all messages being relayed.

Simple wires just relay messages.

Wires to the orchestrator are implemented at the other endpoint, following its protocol, and relay their messages to the orchestrator's context.

Wires between two non-orchestrators are implemented at *both* endpoints and relay their messages to the orchestrators' context, using the wire's name as identifier.

Wires between orchestrators consist simply of the parallel composition of all messages being relayed.

Simple wires just relay messages.

Wires to the orchestrator are implemented at the other endpoint, following its protocol, and relay their messages to the orchestrator's context.

Wires between two non-orchestrators are implemented at *both* endpoints and relay their messages to the orchestrators' context, using the wire's name as identifier.

Wires between orchestrators consist simply of the parallel composition of all messages being relayed.

Simple wires just relay messages.

Wires to the orchestrator are implemented at the other endpoint, following its protocol, and relay their messages to the orchestrator's context.

Wires between two non-orchestrators are implemented at *both* endpoints and relay their messages to the orchestrators' context, using the wire's name as identifier.

Wires between orchestrators consist simply of the parallel composition of all messages being relayed.

Recall the protocol at the REQUIRES interface.

$$B_A \triangleq ?\leftarrow \mathit{FlightDetails}🔔(D, C).! \leftarrow \mathit{FlightDetails}✉(R, P).$$
$$\& \{? \leftarrow \mathit{Book}🔔(); ? \leftarrow \mathit{Cancel}🔔()\}$$

Wire AB_1, connecting this interface to the orchestrator, becomes

$\mathbf{in} \uparrow$ BA_Flight$_1$🔔 (data,class).
$\mathbf{out} \leftarrow$ AA$_1$_FlightDetails🔔 (data,class).
$\mathbf{in} \leftarrow$ AA$_1$_FlightDetails✉ (resp,pr).
$\mathbf{out} \uparrow$ BA_Flight$_1$✉ (resp,pr).
(($\mathbf{in} \uparrow$ BA_Book$_1$🔔().$\mathbf{out} \leftarrow$ AA$_1$_Book🔔())
$+$
($\mathbf{in} \uparrow$ BA_Cancel$_1$🔔().$\mathbf{out} \leftarrow$ AA$_1$_Cancel🔔()))

Recall the protocol at the REQUIRES interface.

$$B_A \triangleq ? \leftarrow \mathit{FlightDetails}\Bell(D, C).! \leftarrow \mathit{FlightDetails}\Letter(R, P).$$
$$\& \{? \leftarrow \mathit{Book}\Bell(); ? \leftarrow \mathit{Cancel}\Bell()\}$$

Wire AB_1, connecting this interface to the orchestrator, becomes

$$\begin{array}{l}
\textbf{in} \uparrow \text{BA_Flight}_1\Bell\ (\text{data,class}). \\
\textbf{out} \leftarrow \text{AA}_1\text{_FlightDetails}\Bell\ (\text{data,class}). \\
\textbf{in} \leftarrow \text{AA}_1\text{_FlightDetails}\Letter\ (\text{resp,pr}). \\
\textbf{out} \uparrow \text{BA_Flight}_1\Letter\ (\text{resp,pr}). \\
((\textbf{in} \uparrow \text{BA_Book}_1\Bell().\textbf{out} \leftarrow \text{AA}_1\text{_Book}\Bell()) \\
\quad + \\
(\textbf{in} \uparrow \text{BA_Cancel}_1\Bell().\textbf{out} \leftarrow \text{AA}_1\text{_Cancel}\Bell()))
\end{array}$$

Recall the protocol at the REQUIRES interface.

$$B_A \triangleq ? \leftarrow \mathit{FlightDetails}🔔(D, C).! \leftarrow \mathit{FlightDetails}✉(R, P).$$
$$\& \{? \leftarrow \mathit{Book}🔔(); ? \leftarrow \mathit{Cancel}🔔()\}$$

Wire AB_1, connecting this interface to the orchestrator, becomes

$\textbf{in} \uparrow \mathrm{BA\_Flight}_1$🔔 (data,class).
$\textbf{out} \leftarrow \mathrm{AA}_1\mathrm{\_FlightDetails}$🔔 (data,class).
$\textbf{in} \leftarrow \mathrm{AA}_1\mathrm{\_FlightDetails}$✉ (resp,pr).
$\textbf{out} \uparrow \mathrm{BA\_Flight}_1$✉ (resp,pr).
(($\textbf{in} \uparrow \mathrm{BA\_Book}_1$🔔().$\textbf{out} \leftarrow \mathrm{AA}_1\mathrm{\_Book}$🔔())
+
($\textbf{in} \uparrow \mathrm{BA\_Cancel}_1$🔔().$\textbf{out} \leftarrow \mathrm{AA}_1\mathrm{\_Cancel}$🔔()))

Recall the protocol at the REQUIRES interface.

$$B_A \triangleq ? \leftarrow \mathit{FlightDetails}\Bell(D, C).! \leftarrow \mathit{FlightDetails}\Letter(R, P). \\ \& \,\{? \leftarrow \mathit{Book}\Bell(); ? \leftarrow \mathit{Cancel}\Bell()\}$$

Wire AB_1, connecting this interface to the orchestrator, becomes

$$\begin{array}{l}
\textbf{in} \uparrow \text{BA_Flight}_1\Bell\ (\text{data,class}). \\
\textbf{out} \leftarrow \text{AA}_1\text{_FlightDetails}\Bell\ (\text{data,class}). \\
\textbf{in} \leftarrow \text{AA}_1\text{_FlightDetails}\Letter\ (\text{resp,pr}). \\
\textbf{out} \uparrow \text{BA_Flight}_1\Letter\ (\text{resp,pr}). \\
((\textbf{in} \uparrow \text{BA_Book}_1\Bell().\textbf{out} \leftarrow \text{AA}_1\text{_Book}\Bell()) \\
\quad + \\
(\textbf{in} \uparrow \text{BA_Cancel}_1\Bell().\textbf{out} \leftarrow \text{AA}_1\text{_Cancel}\Bell()))
\end{array}$$

How everything fits

Assume:

- P implements the wire ends at the PROVIDES interface;
- C_i implement the orchestrators;
- U_i implement wire ends at each used module;
- R_i have the form **instance** $P_i \blacktriangleright S_i \Leftarrow Q_i$, where P_i provides service S_i being invoked at REQUIRES interface i with wire ends S_i.

The implementation is

$$\textbf{def}\, \mathit{Service} \Leftarrow (P \mid C_1 \mid \cdots \mid C_k \mid U_1 \mid \cdots \mid U_m \mid R_1 \mid \cdots \mid R_n)$$

How everything fits

Assume:

- P implements the wire ends at the PROVIDES interface;
- C_i implement the orchestrators;
- U_i implement wire ends at each used module;
- R_i have the form **instance** $P_i \blacktriangleright S_i \Leftarrow Q_i$, where P_i provides service S_i being invoked at REQUIRES interface i with wire ends S_i.

The implementation is

$$\mathbf{def}\, Service \Leftarrow (P \mid C_1 \mid \cdots \mid C_k \mid U_1 \mid \cdots \mid U_m \mid R_1 \mid \cdots \mid R_n)$$

How everything fits

Assume:

- P implements the wire ends at the PROVIDES interface;
- C_i implement the orchestrators;
- U_i implement wire ends at each used module;
- R_i have the form **instance** $P_i \blacktriangleright S_i \Leftarrow Q_i$, where P_i provides service S_i being invoked at REQUIRES interface i with wire ends S_i.

The implementation is

$$\mathbf{def}\,Service \Leftarrow (P \mid C_1 \mid \cdots \mid C_k \mid U_1 \mid \cdots \mid U_m \mid R_1 \mid \cdots \mid R_n)$$

How everything fits

Assume:

- P implements the wire ends at the PROVIDES interface;
- C_i implement the orchestrators;
- U_i implement wire ends at each used module;
- R_i have the form **instance** $P_i \blacktriangleright S_i \Leftarrow Q_i$, where P_i provides service S_i being invoked at REQUIRES interface i with wire ends S_i.

The implementation is

$$\textbf{def}\, Service \Leftarrow (P \mid C_1 \mid \cdots \mid C_k \mid U_1 \mid \cdots \mid U_m \mid R_1 \mid \cdots \mid R_n)$$

How everything fits

Assume:

- P implements the wire ends at the PROVIDES interface;
- C_i implement the orchestrators;
- U_i implement wire ends at each used module;
- R_i have the form **instance** $P_i \blacktriangleright S_i \Leftarrow Q_i$, where P_i provides service S_i being invoked at REQUIRES interface i with wire ends S_i.

The implementation is

$$\mathbf{def}\,\mathit{Service} \Leftarrow (P \mid C_1 \mid \cdots \mid C_k \mid U_1 \mid \cdots \mid U_m \mid R_1 \mid \cdots \mid R_n)$$

How everything fits

Assume:

- P implements the wire ends at the PROVIDES interface;
- C_i implement the orchestrators;
- U_i implement wire ends at each used module;
- R_i have the form **instance** $P_i \blacktriangleright S_i \Leftarrow Q_i$, where P_i provides service S_i being invoked at REQUIRES interface i with wire ends S_i.

The implementation is

$$\mathbf{def}\,Service \Leftarrow (P \mid C_1 \mid \cdots \mid C_k \mid U_1 \mid \cdots \mid U_m \mid R_1 \mid \cdots \mid R_n)$$

How everything fits

Assume:

- P implements the wire ends at the PROVIDES interface;
- C_i implement the orchestrators;
- U_i implement wire ends at each used module;
- R_i have the form **instance** $P_i \blacktriangleright S_i \Leftarrow Q_i$, where P_i provides service S_i being invoked at REQUIRES interface i with wire ends S_i.

The implementation is

$$\mathbf{def}\, Service \Leftarrow (P \mid C_1 \mid \cdots \mid C_k \mid U_1 \mid \cdots \mid U_m \mid R_1 \mid \cdots \mid R_n)$$

The nice part

Applying this to our example yields almost the process that had been defined directly.

Both processes are equivalent (one would hope bisimilar).

The nice part

Applying this to our example yields almost the process that had been defined directly.

Both processes are equivalent (one would hope bisimilar).

The nice part

Applying this to our example yields almost the process that had been defined directly.

Both processes are equivalent (one would hope bisimilar).

So what?

For the specification

For the implementation

So what?

For the specification

- Realizable specification
- No deadlock

For the implementation

- Soundness
- Inherits properties proved abstractly

So what?

For the specification

- Realizable specification
- No deadlock

For the implementation

- Soundness
- Inherits properties proved abstractly

So what?

For the specification

- Realizable specification
- No deadlock

For the implementation

- Soundness
- Inherits properties proved abstractly

So what?

For the specification

- Realizable specification
- No deadlock

For the implementation

- Soundness
- Inherits properties proved abstractly

So what?

For the specification

- Realizable specification
- No deadlock

For the implementation

- Soundness
- Inherits properties proved abstractly

So what?

For the specification

- Realizable specification
- No deadlock

For the implementation

- Soundness
- Inherits properties proved abstractly

Future work

- More formal proofs of some technical details
- Actually write a paper…

Future work

- More formal proofs of some technical details
- Actually write a paper. . .

Future work

- More formal proofs of some technical details
- Actually write a paper. . .