Services: when specification meets implementation

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

LaSIGE and
Department of Informatics
FCUL, Lisbon, Portugal

Brouwer Institute Seminar Series
April 7, 2009

Background

- Sensoria, (web) services and service-oriented computing
- SRML: very graphical, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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. Building the bridge
3. Conclusions

1. Basics
2. Building the bridge
3. Conclusions

Main idea

Common knowledge

A picture is worth a thousand words.

Main idea

Common knowledge

A picture is worth a thousand words.

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

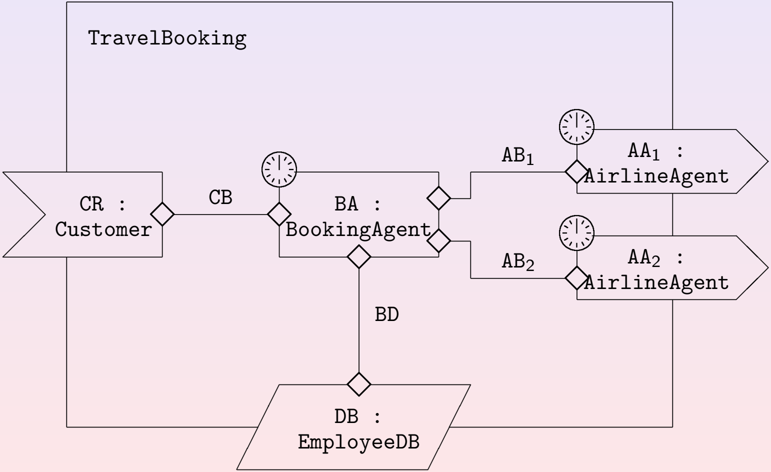
Consider the following example from the list of Sensoria case studies.

Example

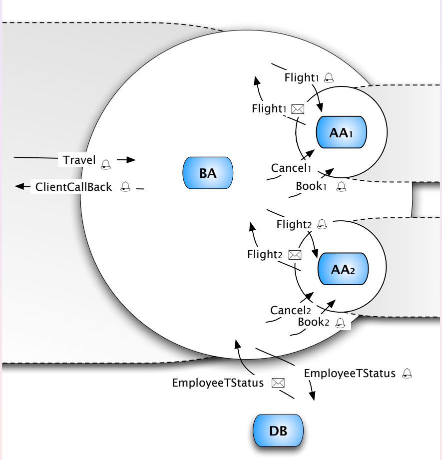
A travel agent provides a booking service that, upon receiving a request for a flight from a customer, executes the following steps:

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.

Specification: diagram



Implementation: diagram



Insight #1

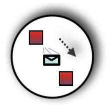
There's a clear structural correspondence between specification and implementation!

Message passing

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

Message passing

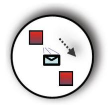
- within the same context (“here”)



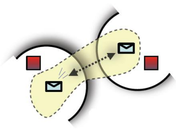
- to the other endpoint of a session (“there”)
- to the enclosing context (“up”)

Message passing

- within the same context ("here")



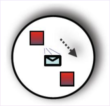
- to the other endpoint of a session ("there")



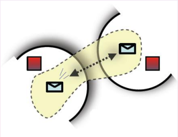
- to the enclosing context ("up")

Message passing

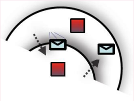
- within the same context (“here”)



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

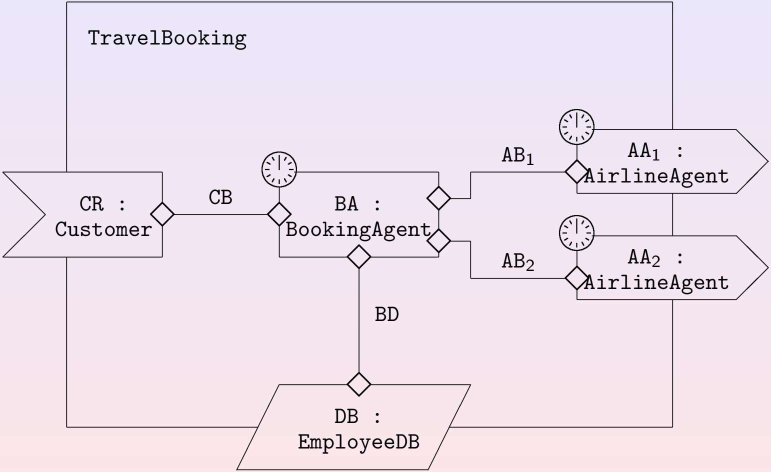


- to the enclosing context (“up”)



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())
    )))
```


TravelBooking
CR : Customer
CB
BA : BookingAgent
AB1
AA1 : AirlineAgent
AB2
AA2 : AirlineAgent
BD
DB : EmployeeDB

Insight #1 (rephrased)

An implementation will consist of several subprocesses running in parallel.

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 $\wedge$ $\mathtt{rec}_1$=false $\wedge$ $\mathtt{rec}_2$=false

initBA **term**: s=DONE

PROVIDES

CR: Customer

REQUIRES

$\mathtt{AA}_1$: AirlineAgent

$\mathtt{triggerAA}_1$ **trigger**: BA.$\mathtt{Flight}_1$?

$\mathtt{AA}_2$: AirlineAgent

$\mathtt{triggerAA}_2$ **trigger**: BA.$\mathtt{Flight}_2$?

USES

DB: EmployeeDB

COMPONENTS

BA: BookingAgent

initBA **init**: s=INIT $\land$ $\mathtt{rec}_1$=false $\land$ $\mathtt{rec}_2$=false

initBA **term**: s=DONE

PROVIDES

CR: Customer

REQUIRES

$\mathtt{AA}_1$: AirlineAgent

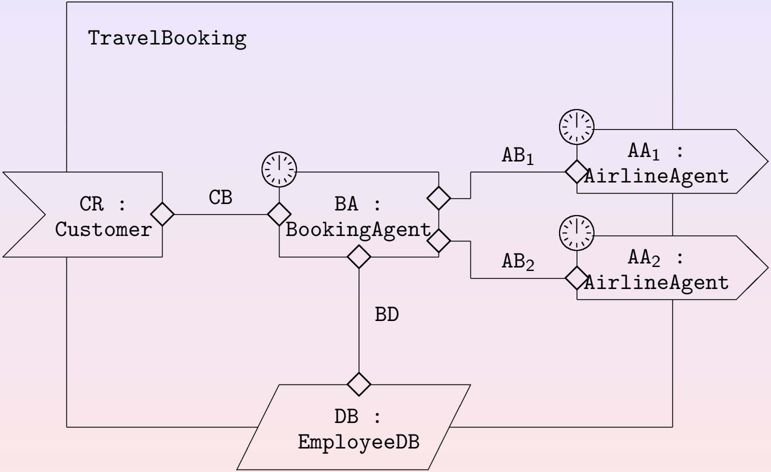
$\mathtt{triggerAA}_1$ **trigger**: $\mathtt{BA.Flight}_1$?

$\mathtt{AA}_2$: AirlineAgent

$\mathtt{triggerAA}_2$ **trigger**: $\mathtt{BA.Flight}_2$?

USES

DB: EmployeeDB


TravelBooking
CR : Customer
CB
BA : BookingAgent
AB1
AA1 : AirlineAgent
AB2
AA2 : AirlineAgent
BD
DB : EmployeeDB

LAYER PROTOCOL `EmployeeDB` **is**

```
INTERACTION
  r&s EmployeeTStatus
    🔔 emp:  employee
    ✉ cl:  travelClass
BEHAVIOUR
  initiallyEnabled EmployeeTStatus🔔?
```

Insight #2

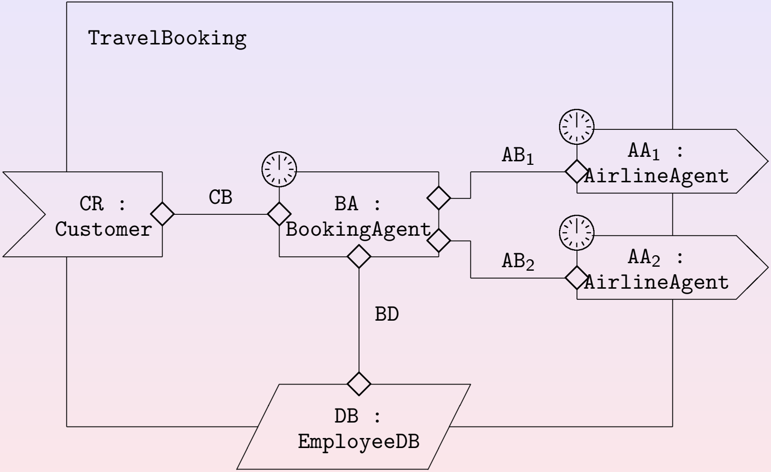
The system depends upon another service running in the context. This protocol specifies the type of that service.

This is typing information.

Insight #2

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

This is typing information.


TravelBooking
CR : Customer
CB
BA : BookingAgent
AB1
AA1 : AirlineAgent
AB2
AA2 : AirlineAgent
BD
DB : EmployeeDB

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 #3

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

This is more typing information.

Insight #3

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

This is more typing information.

More about types

SRML separates behaviour from location. . .

. . . therefore restrict 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.

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

More about types

SRML separates behaviour from location...

...therefore restrict 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.

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

More about types

SRML separates behaviour from location. . .

. . . therefore restrict 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.

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

More about types

SRML separates behaviour from location...

...therefore restrict 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.

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

More about types

SRML separates behaviour from location…

…therefore restrict 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.

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

More about types

SRML separates behaviour from location...

...therefore restrict 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.

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

More about types

SRML separates behaviour from location...

...therefore restrict 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.

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

More about types

SRML separates behaviour from location...

...therefore restrict 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.

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

More about types

SRML separates behaviour from location...

...therefore restrict 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.

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

Allowed behaviours in SRML

For event e, allow E to be either $e?$ or $e!$.

$$\varphi ::= \textbf{initiallyEnabled } e? \mid E \textbf{ enables } e? \mid$$
$$\mid E \textbf{ enables } e? \textbf{ until } E \mid E_1, \ldots, E_k \textbf{ ensures } e!$$

Comparison of terms has no counterpart in these types.

Allowed behaviours in SRML

For event e, allow E to be either $e?$ or $e!$.

$$\varphi ::= \textbf{initiallyEnabled } e? \;[]\; E \textbf{ enables } e? \;[]\; E \textbf{ enables } e? \textbf{ until } E \;[]\; E_1, \ldots, E_k \textbf{ ensures } e!$$

Comparison of terms has no counterpart in these types.

Allowed behaviours in SRML

For event e, allow E to be either $e?$ or $e!$.

$$\begin{aligned}\varphi ::= &\ \textbf{initiallyEnabled}\ e? \mid E\ \textbf{enables}\ e? \mid \\ &\mid E\ \textbf{enables}\ e?\ \textbf{until}\ E \mid E_1, \ldots, E_k\ \textbf{ensures}\ e!\end{aligned}$$

Comparison of terms has no counterpart in these types.

Allowed behaviours in SRML

For event e, allow E to be either $e?$ or $e!$.

$$\begin{aligned}\varphi ::= &\textbf{initiallyEnabled } e? \mid E \textbf{ enables } e? \mid \\ &\mid E \textbf{ enables } e? \textbf{ until } E \mid E_1, \ldots, E_k \textbf{ ensures } e!\end{aligned}$$

Comparison of terms has no counterpart in these types.

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 $\mathtt{B}$ is obtained by adding to $\mathtt{B}$ the formulas:

- $(e\Bell?$ **ensures** $e\Letter!)$ for every **r&s** message e
- $(e\Bell!$ **enables** $e\Letter?)$ for every **s&r** message e

Behaviour trees

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

This tree can be seen as providing a semantics for SRML formulas.

Two extra conditions:

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

Behaviour trees

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

This tree can be seen as providing a semantics for SRML formulas.

Two extra conditions:

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

Behaviour trees

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

This tree can be seen as providing a semantics for SRML formulas.

Two extra conditions:

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

Behaviour trees

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

This tree can be seen as providing a semantics for SRML formulas.

Two extra conditions:

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

Behaviour trees

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

This tree can be seen as providing a semantics for SRML formulas.

Two extra conditions:

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

Behaviour trees

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

This tree can be seen as providing a semantics for SRML formulas.

Two extra conditions:

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

Airline protocol

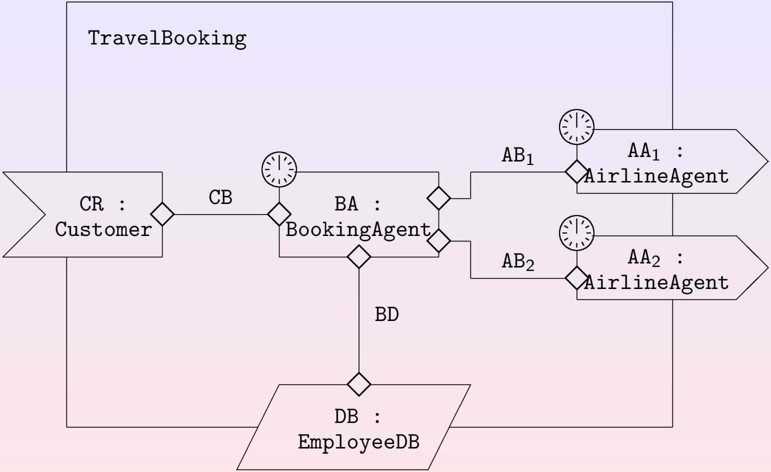
Example

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

Airline protocol

Example

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


TravelBooking
CR : Customer
CB
BA : BookingAgent
AB_1
AA_1 : AirlineAgent
AB_2
AA_2 : AirlineAgent
BD
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 BookFlight

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

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

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

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

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

Insight #4

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

More precisely: there should be a "bisimulation" between the transition system in the specification and the one induced by the implementation.

Formal definition beyond the scope of this presentation :-) besides requiring that all messages be read/written "here".

More precisely: there should be a “bisimulation” between the transition system in the specification and the one induced by the implementation.

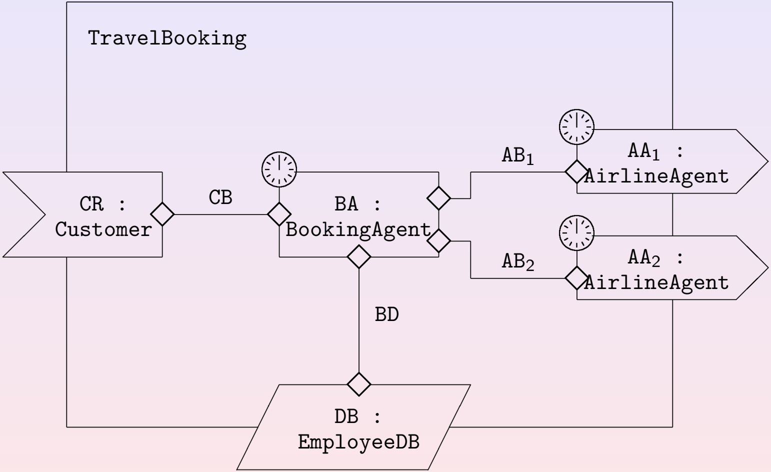
Formal definition beyond the scope of this presentation :-) besides requiring that all messages be read/written “here”.

More precisely: there should be a “bisimulation” between the transition system in the specification and the one induced by the implementation.

Formal definition beyond the scope of this presentation :-) besides requiring that all messages be read/written “here”.

Example

$\mathbf{in} \downarrow \mathrm{Travel}\Bell\ (\mathrm{e,f}).$
$\mathbf{out} \downarrow \mathrm{EmployeeTStatus}\Bell\ (\mathrm{e}).$
$\mathbf{in} \downarrow \mathrm{EmployeeTStatus}\Letter\ (\mathrm{tc}).$
$\mathbf{out} \downarrow \mathrm{Flight}_1\Bell\ (\mathrm{f,tc}).$
$\mathbf{out} \downarrow \mathrm{Flight}_2\Bell\ (\mathrm{f,tc}).$
(
$\quad(\mathbf{in} \downarrow \mathrm{Flight}_1\Letter\ (\mathrm{p}_1,\mathrm{f}_1).\mathbf{out} \downarrow \mathrm{Done})|$
$\quad(\mathbf{in} \downarrow \mathrm{Flight}_2\Letter\ (\mathrm{p}_2,\mathrm{f}_2).\mathbf{out} \downarrow \mathrm{Done})|$
$\quad(\mathbf{in} \downarrow \mathrm{Done}.\mathbf{in} \downarrow \mathrm{Done}.$
$\quad\mathbf{if}\ (p_1 < p_2)\ \mathbf{then}$
$\qquad(\mathbf{out} \downarrow \mathrm{ClientCallBack}\Bell\ (\mathrm{f}_1).\mathbf{out} \downarrow \mathrm{Book}_1\Bell\ ().\mathbf{out} \downarrow \mathrm{Cancel}_2\Bell\ ())$
$\qquad\mathbf{else}\ (\mathbf{out} \downarrow \mathrm{ClientCallBack}\Bell\ (\mathrm{f}_2).\mathbf{out} \downarrow \mathrm{Book}_2\Bell\ ().\mathbf{out} \downarrow \mathrm{Cancel}_1\Bell\ ())$
$\quad))$


TravelBooking
CR : Customer
CB
BA : BookingAgent
AB1
AA1 : AirlineAgent
AB2
AA2 : AirlineAgent
BD
DB : EmployeeDB

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

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

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.

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.

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.

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.

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.

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

Wire AB_1, connecting this interface to the orchestrator, becomes

$\mathbf{in} \uparrow \mathrm{BA\_Flight_1}\Bell\ \mathrm{(data,class)}.$
$\mathbf{out} \leftarrow \mathrm{AA_1\_FlightDetails}\Bell\ \mathrm{(data,class)}.$
$\mathbf{in} \leftarrow \mathrm{AA_1\_FlightDetails}\Letter\ \mathrm{(resp,pr)}.$
$\mathbf{out} \uparrow \mathrm{BA\_Flight_1}\Letter\ \mathrm{(resp,pr)}.$
$((\mathbf{in} \uparrow \mathrm{BA\_Book_1}\Bell().\mathbf{out} \leftarrow \mathrm{AA_1\_Book}\Bell())$
$+$
$(\mathbf{in} \uparrow \mathrm{BA\_Cancel_1}\Bell().\mathbf{out} \leftarrow \mathrm{AA_1\_Cancel}\Bell()))$

Recall the protocol at the REQUIRES interface.

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

Wire AB_1, connecting this interface to the orchestrator, becomes

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

Recall the protocol at the REQUIRES interface.

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

Wire AB_1, connecting this interface to the orchestrator, becomes

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

Recall the protocol at the REQUIRES interface.

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

Wire AB_1, connecting this interface to the orchestrator, becomes

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

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).

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

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

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

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

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

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

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. . .