Towards an agent based approach for verification of OWL-S process models

Monika Solanki

Joint work with Alessio Lomuscio

Imperial College London

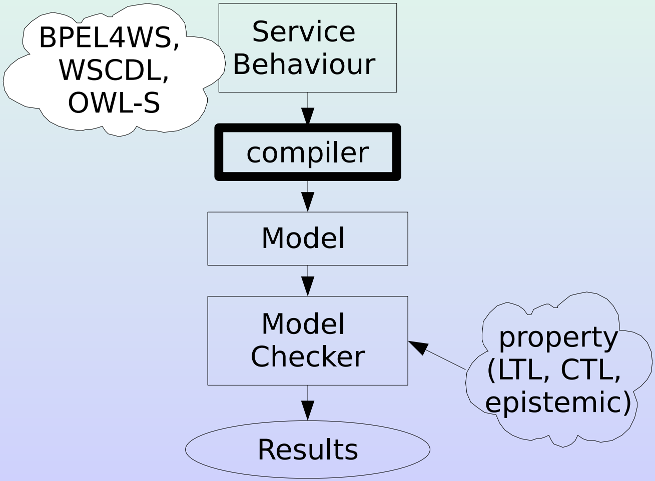
June 2, 2009

Outline

Motivation

- Web services - one of the leading paradigms underlying application integration.
- Some popular standards - WSBPEL, WSCDL, and OWL-S.
- The verification of web service behaviour and interaction protocols is now an integral aspect of several frameworks.
- A popular technique for verification - Model Checking.
- Model checkers typically use modelling languages different from those commonly used for describing services. For e.g.,
  - Promela - SPIN
  - NuSMV - SMV/NuSMV

General architecture for service verification



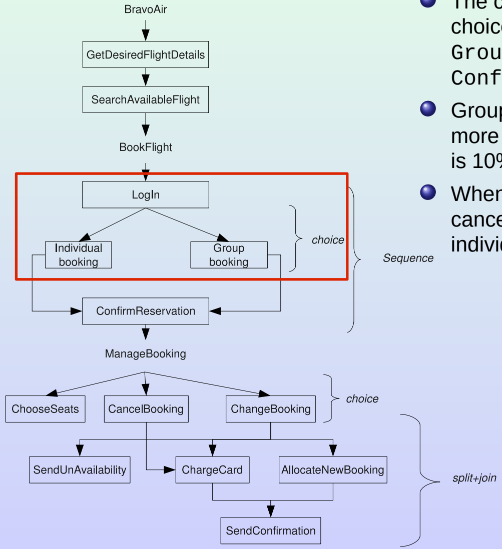
The Challenge

- Generating a well abstracted model is crucial to the verification of services.
- However, developing a tool that generates such a model is non trivial.
- Mapping rules between the languages are required to be established before any automated translation can be undertaken.
- The rules provide the basis and rationale for development of a compiler providing (semi)automatic compilations from one abstraction to the other.

In this paper,

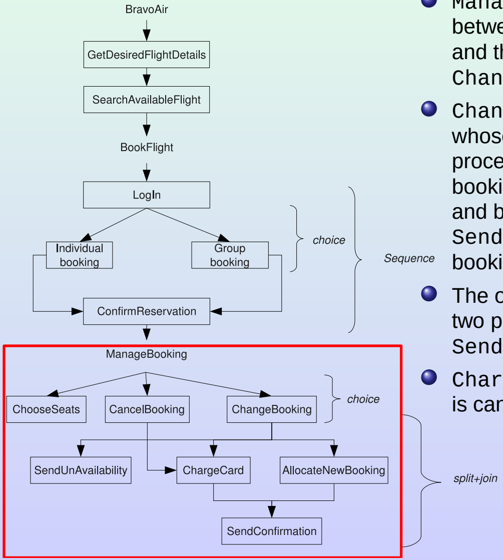
- We take the view that a web service can be modelled as an "agent".
- Multi-agent systems (MAS) serves as a useful metaphor for reasoning about the services provided by "autonomous components acting rationally to maximise their own design objectives".
- We explore the generation of mapping rules from the OWL-S process model to ISPL (Interpreted Systems Programming Language) - the input language for the model checker, MCMAS.
- Illustrative example: a flight booking and managing service - an extended version of the BravoAir process from the OWL-S suite of examples.

Case study: Extended BravoAir



- The components are `LogIn`, followed by a choice between `IndividualBookng` and `GroupBooking` and finally `ConfirmReservation`.
- Group bookings can be done for a group of more than 10 people and the discount offered is 10% of the total booking fee.
- When a group booking is cancelled, the cancellation fee is 15% rather than 10% for individual bookings.

Case study: Extended BravoAir



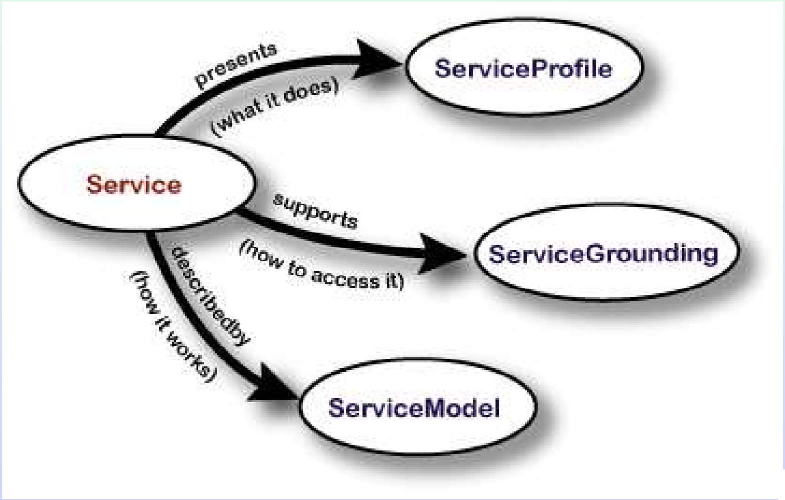
- ManageBooking is composed as a *choice* between the atomic process ChooseSeats and the composite processes, ChangeBooking and CancelBooking.
- ChangeBooking is composed of a *split+join* whose components are the three atomic processes ChargeCard- for economy bookings, AllocateNewBooking- for club and business class bookings and SendUnAvailability- when a change of booking is not possible.
- The outcome from a choice between the first two processes is composed in *sequence* with SendConfirmation.
- ChargeCard is also invoked when a booking is cancelled.

Case study: Extended BravoAir

Behavioural analysis

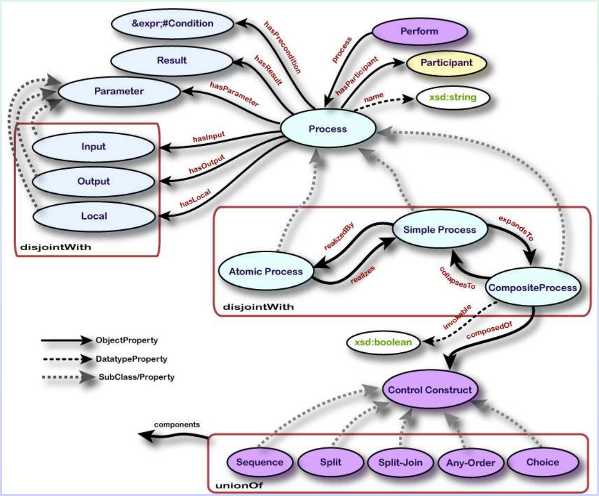
- if there is a request for confirmation of reservation, the ConfirmReservation agent knows that the booking has been successful and payment has been made.
- Whenever a booking change is requested, it will always be confirmed.
- If a card is not charged when a booking is changed, the `ChangeBooking` agent knows that the reference is a business class booking.
- if a confirmation is received, the Customer agent knows that his booking was changed.
- If a card is charged after a booking has been made, it always implies that the booking has been cancelled.

OWL-S



OWL-S Coalition

OWL-S



OWL-S Coalition

Interpreted Systems Programming Language

- MAS are described in MCMAS using a dedicated programming language derived from the formalism of **interpreted systems** - ISPL.
- The first class citizen within an ISPL program is an **“Agent”**.
- Two kinds of Agents: Standard and Environment.
- Environment agent: similar to standard agents and used to describe boundary conditions and infrastructure shared by standard agents. They are not always needed to be defined.

Interpreted Systems Programming Language

An agent in ISPL is defined in terms of:

- A set of **local states**, each of which is characterised by a set of local variables. Currently, ISPL allows three types of variables: Boolean, enumeration and bounded integer.
- A subset of each agents' local states marked as "**green**" to indicate correct functioning behaviour and "**red** " to indicate disallowed behaviour.
- A set of **actions** (for instance "sendmessage" or "open channel".
- A **protocol** - rule describing which action can be performed by an agent in a given local state.
- An **evolution function**, describing how the local states of the agents evolve based on their current local state and on other agents' actions.

Interpreted Systems Programming Language

Additionally,

- The **global evaluation function** of the system defines atomic propositions held over global states which are a combinations of local states of agents defined in the model.
- The **local initial state** for each agent in the system.
- **Specification** to be checked defined as formulae in temporal, epistemic and deontic logic and fairness formulae.

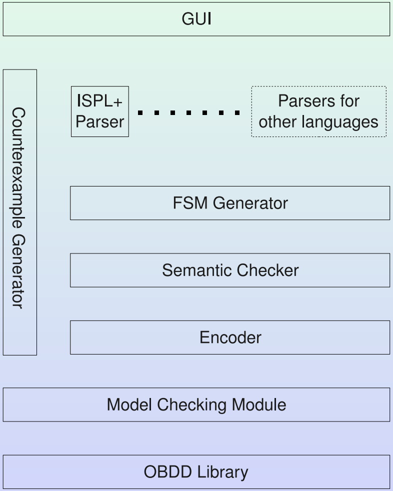
MCMAS

- MCMAS is a symbolic model checker developed particularly for multi-agent systems (MAS) to verify CTL, epistemic, deontic and ATL formulae.
- It takes as input a MAS specification and a set of formulae to be verified.
- It evaluates the truth value of these formulae using algorithms based on **OBDDs** (Ordered Binary Decision Diagrams).

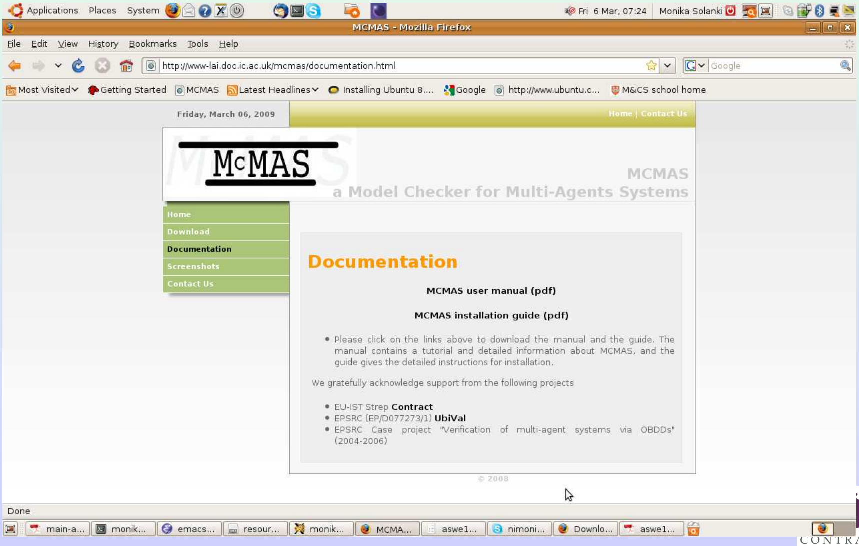
MCMAS

- Support for variables of the following types: Boolean, enumeration and bounded integer. Arithmetic operations can be performed on bounded integers.
- Counterexample/witness generation for quick and efficient display of traces falsifying/satisfying properties.
- Support for fairness constraints. This is useful in eliminating unrealistic behaviours.
- Support for interactive execution mode. This allows users to step through the execution of their model.
- A graphical interface provided as an Eclipse plug-in which includes a graphical editor with syntax recognition, a graphical simulator, and a graphical analyser for counterexamples.

Architecture of MCMAS



MCMAS Home Page


Applications Places System
Fri 6 Mar, 07:24 Monika Solanki
MCMAS - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www-lai.doc.ic.ac.uk/mcmas/documentation.html
Most Visited Getting Started MCMAS Latest Headlines Installing Ubuntu 8.... Google http://www.ubuntu.c... M&CS school home
Home | Contact Us
MCMAS
Home
Download
Documentation
Screenshots
Contact Us
Done
main-a... monik... emacs... resour... monik... MCMA... aswe1... nimoni... Downlo... aswe1...

Friday, March 06, 2009

MCMAS
a Model Checker for Multi-Agents Systems

Documentation

MCMAS user manual (pdf)

MCMAS installation guide (pdf)

- Please click on the links above to download the manual and the guide. The manual contains a tutorial and detailed information about MCMAS, and the guide gives the detailed instructions for installation.

We gratefully acknowledge support from the following projects

- EU-IST Strep **Contract**
- EPSRC (EP/D077273/1) **UbiVal**
- EPSRC Case project "Verification of multi-agent systems via OBDDs" (2004-2006)

© 2008

Mapping OWL-S to ISPL: Atomic processes

Agents:

- For every atomic process in OWL-S we define an agent qualified as `ProcessName` in ISPL.
- Recall, an agent is ISPL includes local states, a subset of local states as green states, actions, protocols and the evolution function.

Mapping OWL-S to ISPL: Atomic processes

Variables and Local states:

- The local states of an agent in ISPL are defined in terms of valuation of the local variables.
- We define the set of local variables for an agent by transforming the ontological inputs and outputs in the process model, to variables with the same identifiers and datatypes in the ISPL model.
- Bounds for integer variables are interactively assigned keeping the domain and context of the process model in perspective.

Mapping OWL-S to ISPL: Atomic processes

Variables and Valuations:

- $V_I \rightarrow Val_I$: integer variables.
- $V_B \rightarrow Val_B$: Boolean variables.
- $V_E \rightarrow Val_E$: variables of type enum.

Set of local states for an agent:

$V = V_I \cup V_B \cup V_E$

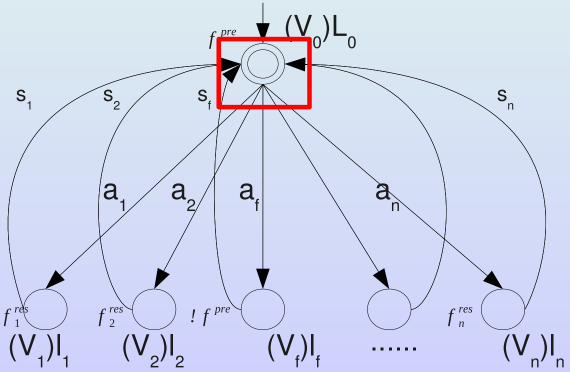
$L_{lstate} : (V_I \rightarrow Val_I) \cup (V_B \rightarrow Val_B) \cup (V_E \rightarrow Val_E)$.

Mapping OWL-S to ISPL: Atomic processes

Variables and Local states:

The set of local states for an agent (L_{lstate}) includes,

- An “Input” or initial state.
- $V_0 \subseteq V$

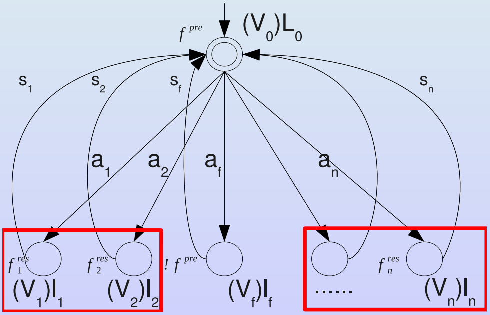


Mapping OWL-S to ISPL: Atomic processes

Variables and Local states:

The set of local states for an agent (L_{lstate}) includes,

- The set of states L_{result}, where each $l \in L_{result}$ corresponds to a non deterministic `Result` state, defined for the process.



Mapping OWL-S to ISPL: Atomic processes

Variables and Local states:

The set of local states for an agent (L_{lstate}) includes,

- The set of states $L_{result} = \{l_1, \ldots, l_n\}$, where each $l \in L_{result}$ corresponds to a non deterministic `Result`, defined for the process.

Example

A credit card validating service may produce two results:

- *ValidationSuccess* with boolean output *validated* as *true*, and,
- *ValidationnFailed* with boolean output *validated* as *false*.

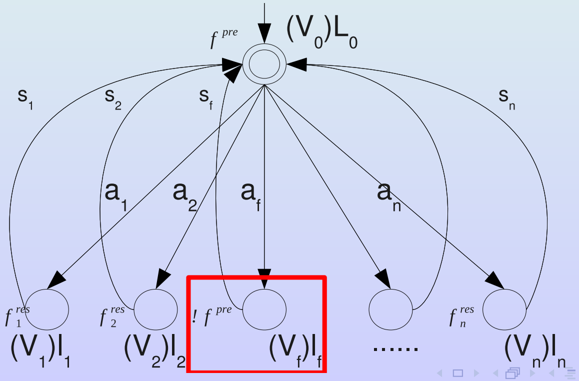
- $V_i \subseteq V, i = 1 \ldots \mid L_{result} \mid$

Mapping OWL-S to ISPL: Atomic processes

Variables and Local states:

The set of local states for an agent (L_{lstate}) includes,

- A failure state l_f which is reached when the preconditions for the process evaluate to *false*.
- $V_f \subseteq V$.



Mapping OWL-S to ISPL: Atomic processes

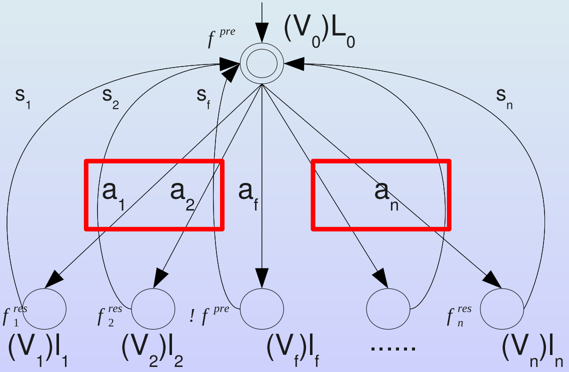
Red States:

- Typically, red states are reached when an agent performs an undesirable action.
- This feature of ISPL is most useful while encoding faults and recovery in complex systems.
- The red states of an agent are represented by a Boolean formula, f^{red}, over its local variables.

Mapping OWL-S to ISPL: Atomic processes

Actions:

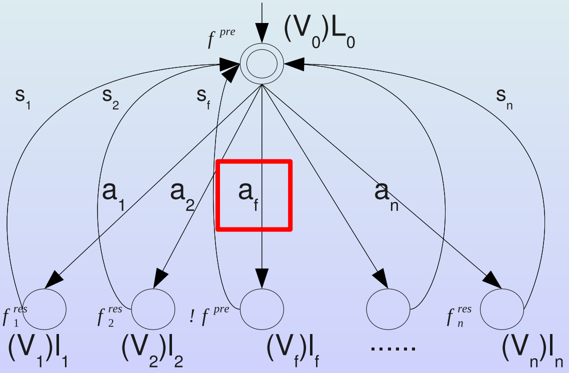
- the null action ϵ,
- the set of internal actions, $A_{int} = \{a_i | i = 1 \ldots n\}$, the agent takes at the input state to reach one of the several result states.



Mapping OWL-S to ISPL: Atomic processes

Actions:

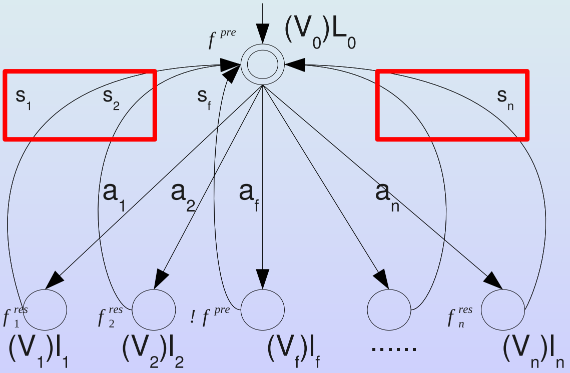
- the internal action a_f taken when the precondition fails, to reach state l_f.



Mapping OWL-S to ISPL: Atomic processes

Actions:

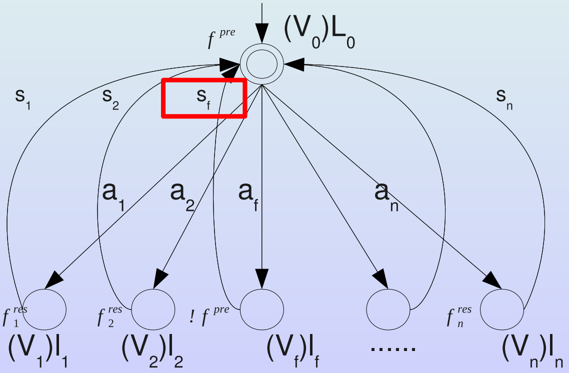
- The set of actions $A_{send} = \{s_1, \ldots, s_n\}$. The agent takes an action $s \in A_{send}$ at each $l \in l_{result}$ to send the corresponding results to the client.



Mapping OWL-S to ISPL: Atomic processes

Actions:

- The action s_f which the agent takes to send the precondition failure message at l_f.



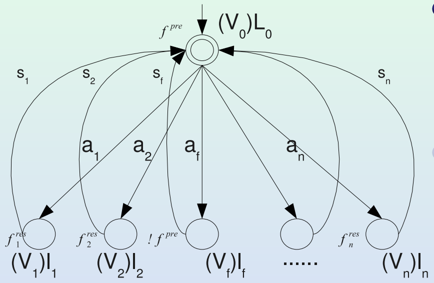
Mapping OWL-S to ISPL: Atomic processes

Protocols:

$$\begin{aligned} f^{pre} &: \{a_i \mid i = 1 \ldots \mid A_{int} \mid\} \\ !f^{pre} &: a_f \\ f_i^{res} &: \{s_i \mid i = 1 \ldots \mid A_{send} \mid\} \cup s_f \end{aligned}$$

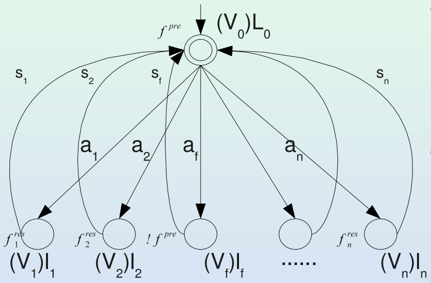
- f^{pre} (precondition), f_i^{res}, $i = 1 \ \ldots \ \mid A_{send} \mid$ (condition in results) and $!f^{pre}$ are Boolean formulae over the set of local variables at the input state, result states and the failure state respectively.
- ISPL and MCMAS allow non determinism in the specification of protocols.

Mapping OWL-S to ISPL: Atomic processes



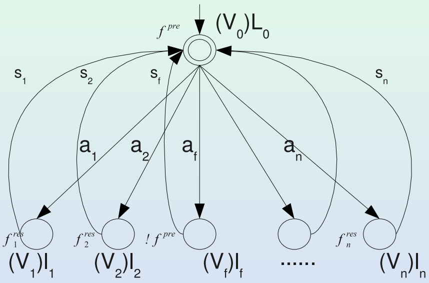
- At execution time an agent at l_0 takes an action, $a \in A_{int}$ if f^{pre}, i.e., the precondition holds and action a_f if $!f^{pre}$ holds.
- This causes a transition to one of the result states $l \in L_{result} \cup l_f$, where the conditionals from the results, f_i^{res}, $i = 1 \ldots | A_{send} |$ are required to hold.
- At l, the agent take an action, $s \in A_{send} \cup \{s_f\}$.

Mapping OWL-S to ISPL: Atomic processes



- At execution time an agent at l_0 takes an action, $a \in A_{int}$ if f^{pre}, i.e., the precondition holds and action a_f if $!f^{pre}$ holds.
- This causes a transition to one of the result states $l \in L_{result} \cup l_f$, where the conditionals from the results, f_i^{res}, $i = 1 \ldots | A_{send} |$ are required to hold.
- At l, the agent take an action, $s \in A_{send} \cup \{s_f\}$.

Mapping OWL-S to ISPL: Atomic processes



- At execution time an agent at l_0 takes an action, $a \in A_{int}$ if f^{pre}, i.e., the precondition holds and action a_f if $!f^{pre}$ holds.
- This causes a transition to one of the result states $l \in L_{result} \cup l_f$, where the conditionals from the results, f_i^{res}, $i = 1 \ldots \mid A_{send} \mid$ are required to hold.
- At l, the agent take an action, $s \in A_{send} \cup \{s_f\}$.

Mapping OWL-S to ISPL: Atomic processes

Evolutions (Transitions):

- The evolution function determines how local states evolve based on the agent's current local state and a set of actions.
- An evolution consists of a set of assignments of local variables in V and an enabling condition which is a Boolean formula, over local variables and actions of all agents.

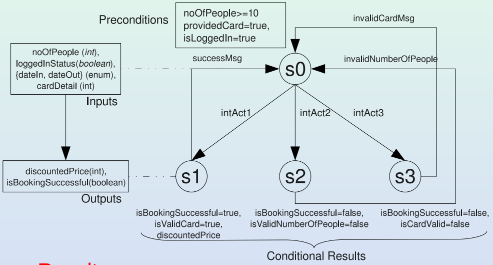
Evolution function

l_0 if f_i^{res} and ProcessName.Action $=\ s_i$ or ProcessName.Action $=\ s_f, i = 1 \ldots \mid A_{send} \mid$

l_i if f^{pre} and ProcessName.Action $= a_i, i = 1 \ldots \mid A_{int} \mid$

l_f if $!f^{pre}$ and ProcessName.Action $= a_f$

BravoAir atomic process: GroupBooking



- Inputs: *noOfPeople*, *flightDetails*, *cardDetails* and *loggedInStatus*
- Preconditions: $loggedInStatus \wedge noOfPeople >= 10 \wedge providedCard(cardDetails)$
- Outputs: *successMsg*, *invalidCardMsg*, *invalidNumMsg* and *discountedPrice*

Results:

$$isValidCard \wedge noOfPeople >= 10 \rightarrow isBookingSuccessful$$
$$!isValidCard \rightarrow invalidCardMsg$$
$$isValidNumberOfPeople \rightarrow invalidNumMsg$$

BravoAir atomic process: GroupBooking

ISPL for the GroupBooking agent:

```
Agent GroupBooking
Vars:
noOfPeople:1..20;
isValidNumberOfPeople:boolean;
loggedIn:boolean;
providedCard:boolean;
isValidCard:boolean;
isBookingSuccessful:boolean;
price:1000..200000;
discountedPrice:100..20000;
dates:{dout, din};
successMsgSent:boolean;
cardFailureMsgSent:boolean;
numberFailureMsgSent:boolean;
end Vars
RedStates:
end RedStates
Actions = {intAct1, intAct2, intAct3, intAct4,
     invalidCardMsg, invalidNumMsg, successMsg, nothing};
```

BravoAir atomic process: GroupBooking

ISPL for the GroupBooking agent:

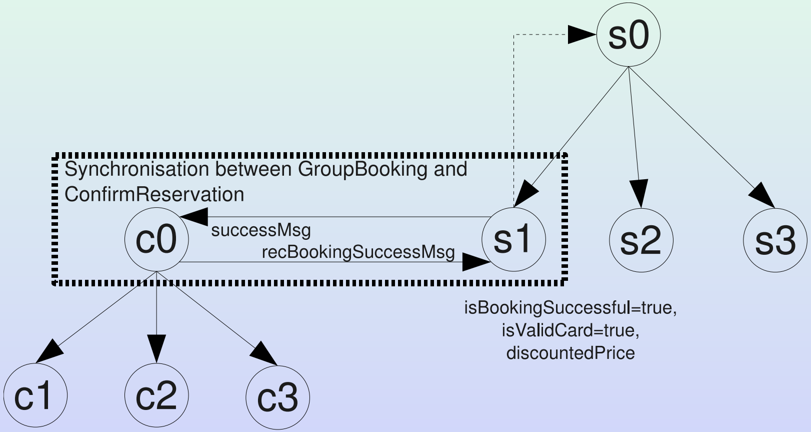
```
Protocol:
loggedIn=true and providedCard=true and noOfPeople >=10 :
        {intAct1, intAct3};
loggedIn=true and noOfPeople <10 : {intAct2};
isValidCard=false:{invalidCardMsg};
isValidNumberOfPeople=false:{invalidNumMsg};
isBookingSuccessful=true:{successMsg};
end Protocol
Evolution:
isBookingSuccessful=true and isValidCard=true and
        discountedPrice=price -(price * 1/10) if
loggedIn=true and providedCard=true and noOfPeople>=10 and
         GroupBooking.Action=intAct1;
isBookingSuccessful=false and isValidCard=false if
         providedCard=true and
         GroupBooking.Action=intAct3;
isBookingSuccessful=false if noOfPeople<=10 and
         GroupBooking.Action=intAct2;
successMsgSent=true if isBookingSuccessful=true and
         GroupBooking.Action=successMsg;
cardFailureMsgSent=true if isBookingSuccessful=false and
         GroupBooking.Action=invalidCardMsg;
end Evolution
end Agent
```

Mapping Composite Processes

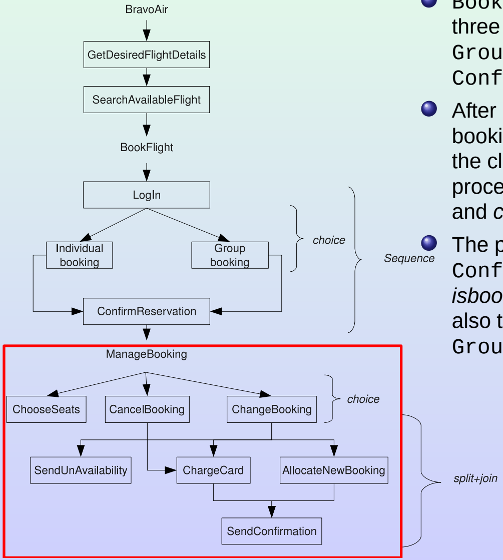
Sequence:

- The *sequence* specifies a list of processes to be executed in a certain order.
- The modelling of OWL-S sequence requires explicit synchronisation between the agents.
- In ISPL, the definition of evolution for an agent encodes this synchronisation.

BravoAir composite process: BookFlight

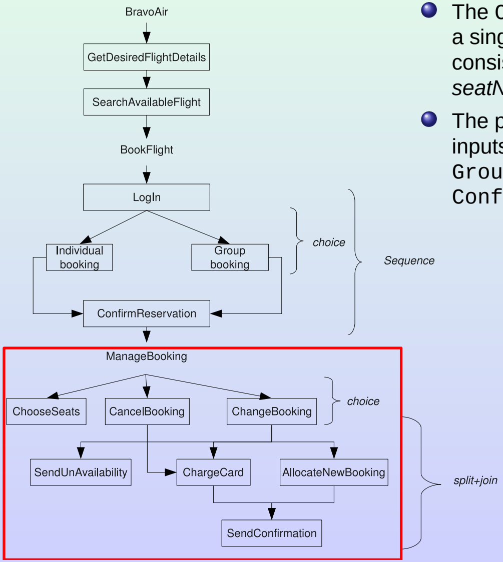

s0
Synchronisation between GroupBooking and ConfirmReservation
c0
successMsg
recBookingSuccessMsg
s1
s2
s3
isBookingSuccessful=true,
isValidCard=true,
discountedPrice
c1
c2
c3

BravoAir composite process: BookFlight



- `BookFlight` is a sequential composition of three atomic processes, `Login`, `GroupBooking` and `ConfirmReservation`.
- After receiving the result of a successful booking from the `GroupBooking` process, the client invokes the `ConfirmReservation` process with inputs *isbookingSuccessful=true* and *confirmFlight=true*.
- The precondition for the execution of `ConfirmReservation` is *isbookingSuccessful=true*. Note that this was also the result condition of the `GroupBooking` process.

BravoAir composite process: BookFlight



- The `ConfirmReservation` process returns a single result as a complex message consisting of a *reservationID* and *seatNumber*.
- The processes are synchronised for these inputs on the final state of `GroupBooking` and the initial state of `ConfirmResearvation`.

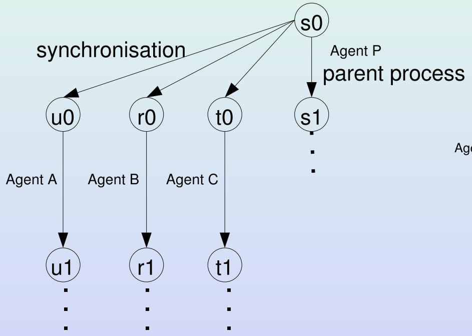
BravoAir composite process: BookFlight

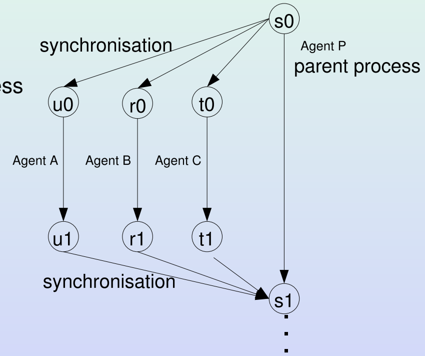
Evolution function for the “GroupBooking” agent:

```
Evolution:
isBookingSuccessful=true and isValidCard=true and
         discountedPrice=price -(price * 1/10) if
loggedIn=true and providedCard=true and noOfPeople>=10 and
          GroupBooking.Action=act1;
isBookingSuccessful=false and isValidCard=false if
 providedCard=true and GroupBooking.Action=act3;
isBookingSuccessful=false if
noOfPeople<=10 and GroupBooking.Action=act2;
sucessMsgSent=true if
isBookingSuccessful=true and GroupBooking.Action=successMsg and
ConfirmResearvation.Action=recBookingSuccessMsg;
cardFailureMsgSent=true if
isBookingSuccessful=false and GroupBooking.Action=invalidCardMsg;
numberFailureMsgSent=true if
isBookingSuccessful=false and GroupBooking.Action=invalidNumMsg;
end Evolution
```

Mapping Composite Processes

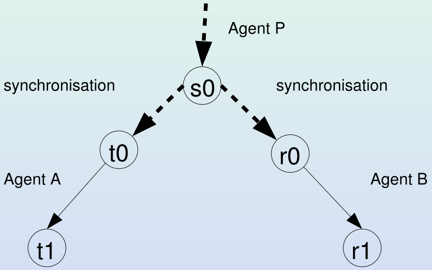
Split and Split+Join:

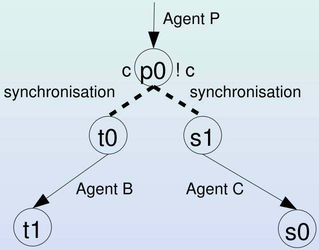




Mapping Composite Processes

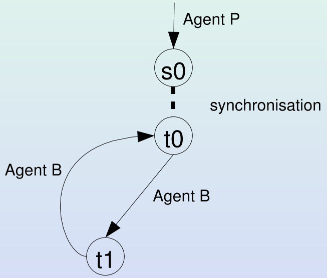
Choice and if-then-else:

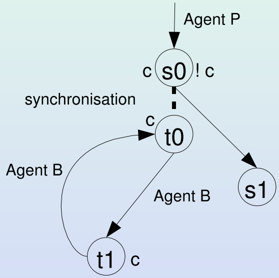




Mapping Composite Processes

Iterate, Repeat-while, Repeat-until:





A few interesting properties

- if there is a request for flight booking confirmation, the ConfirmReservation agent(*CR*) knows that the customer (*C*) is an authorised customer.

$$EF((confirmBookingRequest) \rightarrow K_{CR}(authorisedCustomer))$$

- Whenever a booking change is requested, it will eventually be confirmed.

$$EF((bookingChangeRequest) \rightarrow EF(sendConfirmation))$$

  Intuitively the property does not hold because if there are no alternative bookings available, the change will not be confirmed.

- If a card is not charged when a booking is changed, the `ChangeBooking` agent *CB* knows that the reference is a business class booking.

$$EF(bookingChanged \wedge \neg cardCharged \rightarrow K_{CB}(businessBooking))$$

AAMAS2008: Towards verifying compliance in agent-based Web service compositions

Conclusions

- MAS serves as a useful metaphor for reasoning about the services provided by "*autonomous* components acting rationally to maximise their own design objectives".
- We have proposed mapping rules from the process model of OWL-S to ISPL.
- We have shown the mapping for atomic processes and for certain control constructs used for composing them.
- Our approach provides the first steps necessary to automate the compilation from OWL-S process models to ISPL.
- A primitive semi-automatic compiler implementing the rules has been developed - takes as input the RDF serialisation of OWL-S.

Ongoing/Future work

- Formally establishing the soundness and completeness of the mapping rules.
- Enhancing and improving the existing basic compiler.
- Integrating the compiler with the MCMAS Eclipse plugin.