FLOWS:
A First-Order Logic Ontology for Web Services

June 30, 2004

D.Berardi, M.Gruninger, R.Hull, S.McIlraith

Outline

What is FLOWS? (Profile, Process Model, Surface Language)

- Representational Desiderata for a WSC ontology
- Pros/Cons of FOL

FLOWS Process Model

- The Process Specification Language (PSL)
- Issues: Relationship to OWL-S, Tractability

FLOWS Surface Language

  - FLOWS Query Language
  - FLOWS Specification Language

Issue: Implementations

Summary & Discussion

Supplementary material

What is FLOWS?

FLOWS is:

a First-order Logic Ontology for Web Services

FLOWS comprises:

- FLOWS Profile
- FLOWS Process Model
- FLOWS Surface Languages
  - FLOWS Query Language (FQL)
  - FLOWS Specification Language (FSL)

Representational Desiderata:

- Model-theoretic semantics
- Primitive and complex processes are first-class objects
- Taxonomic representation
- Leverages existing service ontologies (OWL-S)
- Embraces and integrates with existing and emerging standards and research (BPEL, W3C choreography, etc.)
- Explicit representation of messages and dataflow (cf. W3C choreography, behavioral message-based signatures, etc.)
- Captures activities, process preconditions and effects on world.
- Captures process execution history.

Some Pros/Cons of FOL

+ provides a well-understood model-theoretic semantics

+ rich expressive power (e.g., variables, quantifiers, terms, etc.)
  -- overcomes expressiveness issues that have haunted OWL-S

+ enables characterization of reasoning tasks in terms of classical notions of deduction, consistency, etc.

+ enables exploitation of off-the-shelf systems such as existing FOL reasoning engines and DB query engines.

- semi-decidable and intractable for many tasks (worst case) (tractability is not about the language, but note that many intractable tasks often prove easily solved in practice)

- syntax unsuitable for common man (surface languages under development)

+ provides a theoretical mechanism for preserving semantics and relating different SWS ontologies

+ enables (easy) mapping to lite versions of ontology

+ provides basis for blending results about SWS origins in different methodologies (e.g., automata-based, DL-based, Petri-net based, sitcalc-based, etc)

+ easily incorporate pre-existing work. Can import other ontologies relatively seamlessly

Outline

What is FLOWS? (Profile, Process Model, Surface Language)

- Representational Desiderata for a WSC ontology
- Pros/Cons of FOL

➔ FLOWS Process Model

- The Process Specification Language (PSL)
- Issues: Relationship to OWL-S, Tractability

FLOWS Surface Language

  - FLOWS Query Language
  - FLOWS Specification Language

Issue: Implementations

Summary & Discussion

Supplementary material

FLOWS Process Model

- FLOWS Process Model consists of
  - a subset of the PSL Ontology
  - extensions for service concepts

This is not new research.

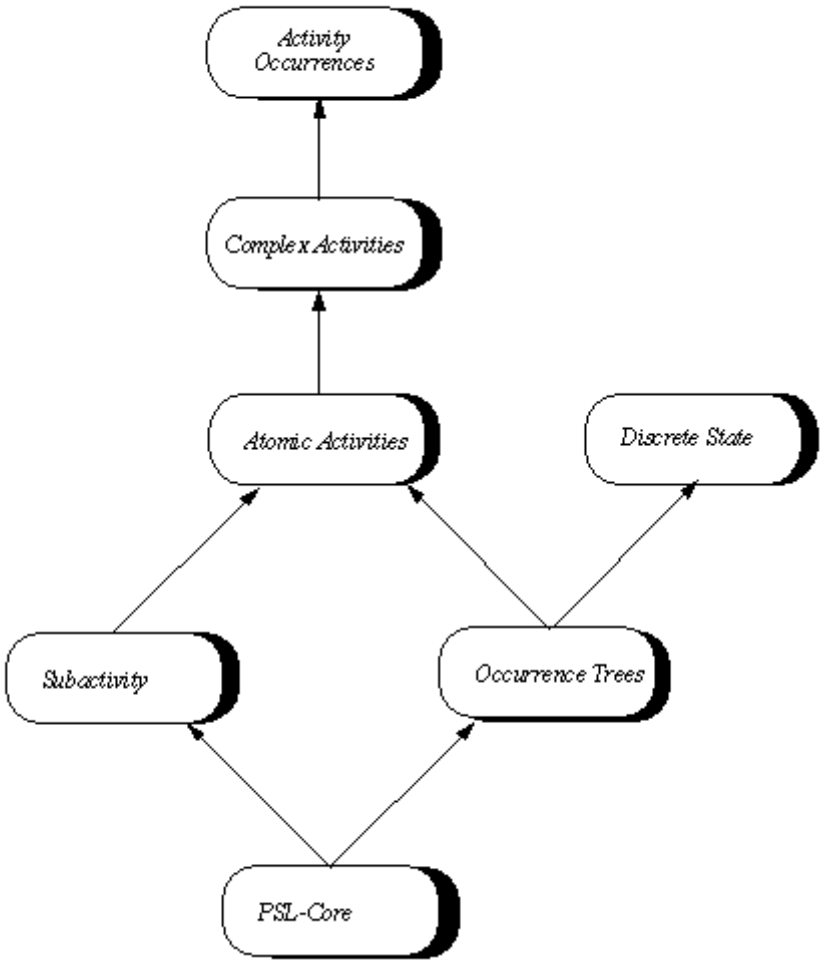
The bulk of this already exists and has been vetted.

… so here's an overview of PSL….

Process Specification Language (PSL)

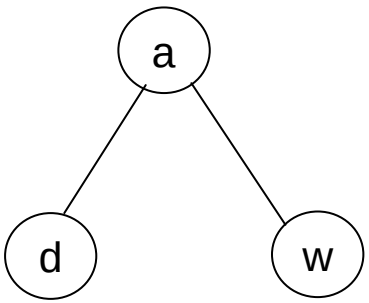
- PSL is a modular, extensible first-order logic ontology capturing concepts required for manufacturing and business process specification
  - PSL is an International Standard (ISO 18629)
  - There are currently 300 concepts across 50 extensions of a common core theory (PSL-Core), each with a set of first-order axioms written in Common Logic (ISO 24707)
  - The core theories of the PSL Ontology extend situation calculus
  - PSL is a verified ontology -- all models of the axioms are isomorphic to models that specify the intended semantics

PSL Core Theories

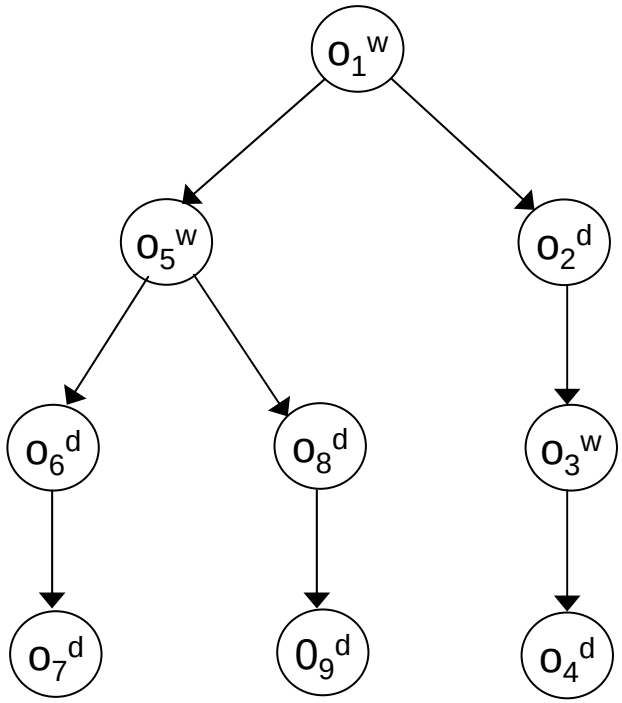


Some Structures in Models of PSL

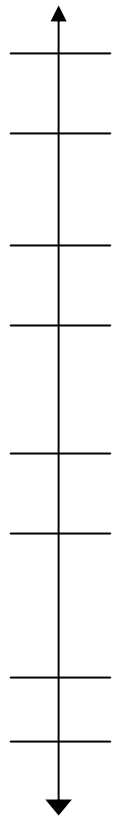
subactivity



activity tree



timeline



FLOW Process Model Concepts

The FLOWS Process Model incorporates the following concepts and features from PSL:

- Ordering and temporal constraints
  - Simple workflows
  - Iterated processes
  - Duration constraints
  - Concurrency
- Explicit representation of state and state constraints
  - World state conditions, inputs, and outputs, epistemic states of actors
  - Preconditions and effects
  - Conditional processes
- Occurrence constraints
- Composition
  - Complex activities/services are first-class objects in the domain
  - Process decomposition (e.g., subactivities)
  - Nondeterminism (e.g. alternative processes)
  - Interactions with external activities
  - Incomplete process specifications.

Pros/Cons of using PSL

+ years of development in the business process modeling arena
+ PSL is an International Standard, already proven useful as exchange language
+ extensibility of PSL
+ first-stage characterization of OWL-S process model semantics
+ PSL was designed to support interoperability and a number of different ontologies have been mapped into PSL.
+ PSL can consistently include ontologies for time and duration (e.g. DAML-Time)
- readability and writability
- ignores continuous change (though situation calculus proposals exist)
- few implementations of associated reasoners. In particular, there is no canonical implementation of PSL, since it is being used in different ways in different applications

Issue: Relationship to OWL-S

- FLOWS provides a first-order axiomatization of the intended semantics of OWL-S.
  - OWL is too weak to completely axiomatize the intended semantics of OWL-S.
  - Any implementations must resort to extralogical mechanisms if they are to conform to the OWL-S semantics, whereas implementations of FLOWS will be able to use the axioms directly.

Complementary relationship to other emerging WS standards (e.g., BPEL, WSDL). Formal characterization – future work.

Issue: Tractability

- Use case scenarios show that in general we will need to solve intractable reasoning problems.
  - Reasoning problems for semantic web services are inherently intractable -- using a different language does not make them tractable.
  - If you restrict yourself to a language that is tractable, then there will exist reasoning problems that cannot be specified in the language.
  - FLOWS enables identification and exploitation of (pragmatically) tractable subclasses, while maintaining the virtues of the full FLOWS ontology.

Outline

What is FLOWS? (Profile, Process Model, Surface Language)

- Representational Desiderata for a WSC ontology
- Pros/Cons of FOL

FLOWS Process Model

- The Process Specification Language (PSL)
- Issues: Relationship to OWL-S, Tractability

➔ FLOWS Surface Language

  - FLOWS Query Language
  - FLOWS Specification Language

Issue: Implementations

Summary & Discussion

Supplementary material

FLOWS Query Language (FQL)

- We are working on a query language proposal inspired by
  - PSL: basic objects are (atomic, composite) activities and occurrences; tests, inserts, deletes of fluents “in the world”
  - OWL-S: permit additional structure for activities, including IOPE
  - OQL: functional query language for complex objects, extended and relativized to the structures and operators in web services
- Example (simple) query in *preliminary* version of FQL

```
hotel_reservation_service =
   select h
   from h in UDDI,
        hotel,person,d1,d2 in h.inputs
   where hotel.type subclass_of Hotels and
          person.type subclass_of String and
          d1.type, d2.type subclass_of Date and
          h.precond has_element_equiv 'val(d1) < val(d2)' and
          h.precond has_element_equiv 'vacancy(val(hotel), val(d1), val(d2))' and
          h.effect has_element_equiv  '+hotel_res(val(hotel),val(person), val(d1), val(d2))'
```

- Can exploit recursive structure of query components to create intricate but natural queries, including compositions
  - Can use quantifiers, but can express many things without them

FLOWS Specification Language (FSL)

- Purpose of the specification language is to provide the “common man” with a language to describe service properties and capabilities (the FLOWS profile and process model).
- FSL will use the vocabulary defined in the profile and process ontologies to develop a surface language akin to FQL.
- This is future work, over and above what we will leverage from FQL.

Outline

What is FLOWS? (Profile, Process Model, Surface Language)

- Representational Desiderata for a WSC ontology
- Pros/Cons of FOL

FLOWS Process Model

- The Process Specification Language (PSL)
- Issues: Relationship to OWL-S, Tractability

FLOWS Surface Language

  - FLOWS Query Language
  - FLOWS Specification Language

➔ Issue: Implementations

Summary & Discussion

Supplementary material

Issue: Implementations

- FLOWS can reuse existing implementations of PSL (see attached slides)
- Golog programs are equivalent to process descriptions for restricted classes of FLOWS activities.
  - Any implementation of Golog can be used and extended for FLOWS service descriptions. (Such implementations exist in Prolog.)
- Mapping exists from finite state machine models to classes of FLOWS processes
  - Various tools available for (approximate) verification
- FQL gives framework to start working on distributed query evaluation
  - Can borrow algorithms, optimizations from DB literature

Summary of FLOWS

- Designed on rigorous theoretical foundations
- Consistent extension of OWL-S (backwards compatibility)
- Process model based on international standards (ISO 18629)
- Proposed Surface Language based on acccepted DB approaches to query languages (OQL)
- Ability to reuse existing implementations from a variety of applications

Supplementary Material

In this slide deck:

- **SWSL Case Studies**
  - Financial transaction example
  - Travel service scenario

In a separate slide deck we provided:

- **Further details on PSL**

Discussion?

Supplementary Material…

Case Studies

- Financial transaction example ✓
- Travel service scenario ✓

Financial Transactions Use Case

- Embedding in PSL involves the following:
  - Subactivities
  - Partially ordered deterministic complex activities
  - Precondition axioms
    - Conditions on fluents that must hold before an activity can occur
  - Context-sensitive effect axioms
    - Effects of an activity occurrence can vary depending on fluents
  - Classes of activities denoted by terms (with parameters)
    - This capability not in OWL
- We illustrate how selected use-case assertions can be expressed in PSL
  - We rely on quantification over complex activities

Financial Transactions: Key Building Blocks

- Activities as terms

$\forall x$ activity(buy_products(x))
$\forall x,y,z$ activity(transfer(x,y,z))
$\forall x,y$ activity(withdraw(x,y))
$\forall x,y$ activity(deposit(x,y))

- Composition relationships

$\forall a,y$ (a = buy_product(y) $\supset \exists$ x,z subactivity(transfer(x,y,z) , a))
$\forall x,y,z$ subactivity(withdraw(x,y), transfer(x,y,z))
$\forall x,y,z$ subactivity(deposit(x,z)), transfer(x,y,z))

- Process description for *buy_product*

$\forall o,x$ occurrance_of(o, buy_product(x)) $\supset$
$\exists o1,o2,y,z,w,v$ occurrence_of(o1, transfer(y,x,z)
$\wedge$ occurrence_of(o2, transfer(w,x,v))
$\wedge$ subactivity_occurrence(o1, o)
$\wedge$ subactivity_occurrence(o2, o)

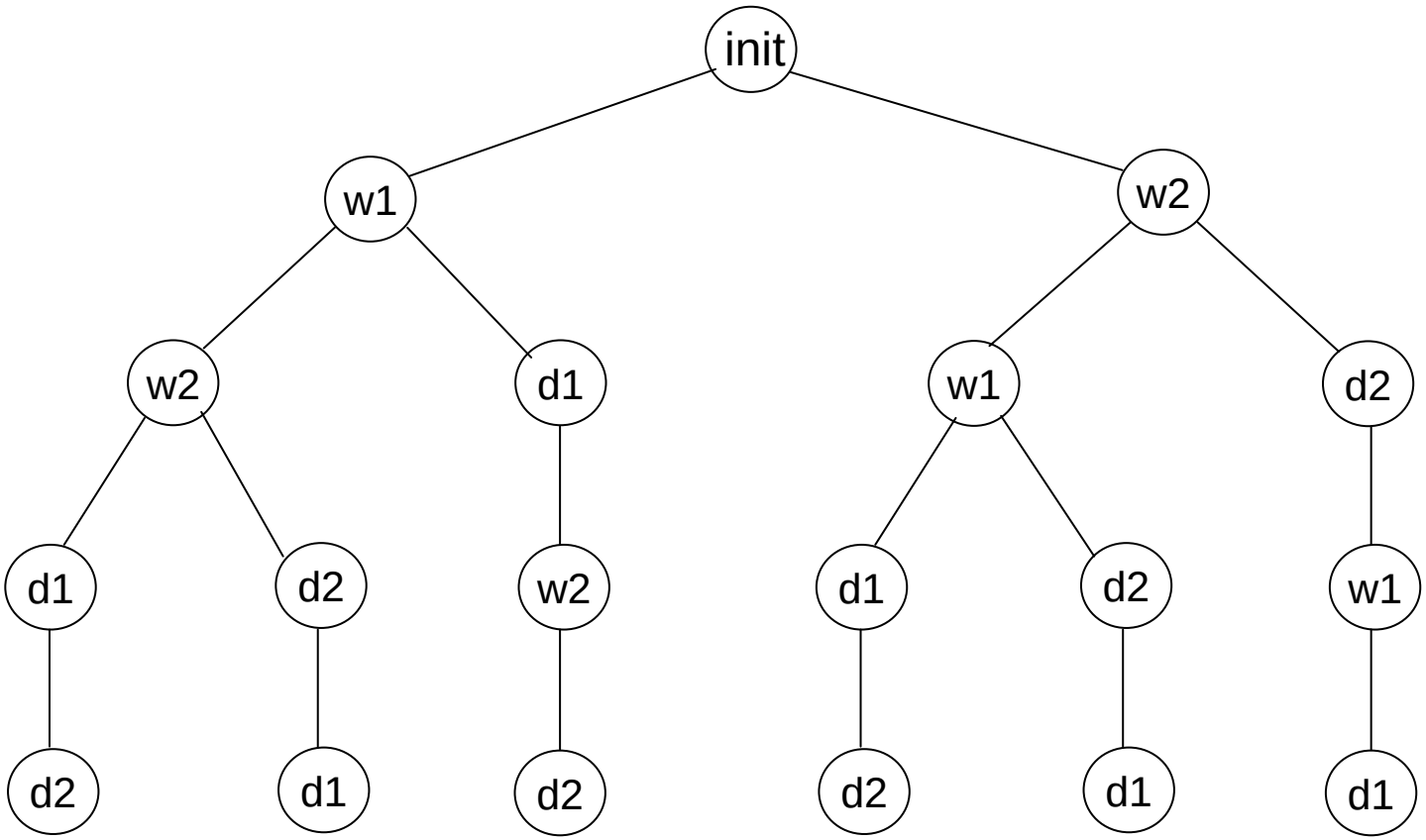
- Can represent
  - Other composite activities
  - Pre-conditions (e.g., transfers only if sufficient funds)
  - Effects (e.g., of a transfer)

Minimal activity tree

- Assume four atomic activity types

w1 = Withdraw (100, Account1)
w2 = withdraw (5, Account1)

d1 = deposit (100, Account2)
d2 = deposit (5, Account3)



Example assertion from Use Case

- Very preliminary sketch, to give basic idea
- Two transfers of X and Y are equivalent to one transfer of X+Y (between same accounts). But the fee is double.

∀ o1,o2 (
equivalent(o1,o2) iff
∀ o3, o4, buyer, seller, broker, amount1, amount2, amount3, fee1, fee2, fee3
(if occurrence_of (o1, double_transfer (buyer, seller, broker, amount1, fee1, amount2, fee2)
∧ subactivity_occurrence (o3, o1)
∧ subactivity_occurrence (o4, o1)
∧ subactivity (transfer(buyer, seller, amount1), o3)
∧ subactivity (transfer(buyer, broker, fee1), o3)
∧ subactivity (transfer(buyer, seller, amount2), o4)
∧ subactivity (transfer(buyer, broker, fee2), o4)

∧

occurrence_of (o2, merged_transfer(buyer, seller, broker, amount3, fee3)
∧ subactivity(transfer(buyer, seller, amount3), o2) and
∧ subactivity(transfer(buyer, broker, fee3)), o2)

then amount3 = plus(amount1, amount2) ∧ fee3 = plus(fee1, fee2)
)

Another assertion from Use Case

- Very preliminary sketch, to give basic idea
- Multiple international money transfers on the same account are not executed in parallel by bank B unless the costumer has a long-lasting relationship with bank B

$\forall$ o1, o2, account, account1, account2, amount1, amount2 (
  if occurrence_of (o1, transfer(account, account1, amount1))
    $\wedge$ occurrence_of (o2, transfer(account, account2, amount2))
    $\wedge$ "o1 is international"
    $\wedge$ "o2 is international"

  then precedes(o1, o2) or precedes(o2, o1)

Travel Use Case

An example of rich services and rich composition

- Atomic and non-atomic (fsa-based) “base” services
- Sequential and interleaved composition
- Activities and messages in one framework

Three services

- Different kinds of users want the services called in different orders
  - E.g., tourist wants hotel; plane; event

We illustrate how PSL can express 3 perspectives:

1. Atomic / SingleUse (cf OWL-S)
   - View each service as atomic
   - Create composite service for one use only
2. Interactive / generic re-usable (cf Roman model)
   - View each service as activity-based fsa
   - Create re-usable composite service targeted to any user
3. Blending of activity-based and message-based
   - View message send/receive as activities
   - Record message contents in predicate-based fluents
   - Can describe data flow, track history

book_hotel

Prec: hotel_booked = false

Input: hotel_city, date_arrive, date_back

Output: name_hotel, hotel_booking_id

Eff: hotel_booked = true

book_plane

Prec: plane_booked = false

Input: depature_city, date_leave, arrival_airport, date_back

Output: ticket_plane_id

Eff: plane_booked = true

register_event

Prec: event_booked = false

Input: event_name,

Output: start_attend_date, end_attend_date, registration_id, city_nearby_hotel, nearby_airport

Eff: event_booked = true

1. Atomic eService/SingleUse composition (sketch)

- Building composite activity “Maria_serv” for tourist Maria
- Specify that the three atomic services are in sequence; include simple exception handling
- (Selected) fluents: *booked_xxx*, *Success_xxx_booking*, *Fail_xxx_booking*

// establish sub-activity structure for Maria_serv

subactivity(launch, Maria_serv) ∧ subactivity(book_hotel, Maria_serv) ∧
subactivity(book_plane, Maria_serv) ∧ subactivity(register_event, Maria_serv)

// characterize all possible occurrances of Maria_serv (i.e., all paths in activity tree for Maria_serv)

∀x. occurrence_of (x, Maria_serv) ⇔

// exists a root atomic occurrance and atomic occurrance of book_hotel activity

(∃o1 occurrence_of(o1,book_hotel) ∧ subactivity_occ(o1, x) ∧ root(o0,x) ∧

(if ¬ (prior(*Precond_hotel*, o1) ∧ prior(*Input_hotel*, o1))
then (holds(*Failure_hotel_booking* , o1) ∧ leaf_occurrence(o1, x))
else (holds(*Eff_hotel*, o1) ∧ holds(*success_hotel_booking*,01) ∧

// if the book_hotel occurrance succeeded, then there is also an occurrance of book_plane

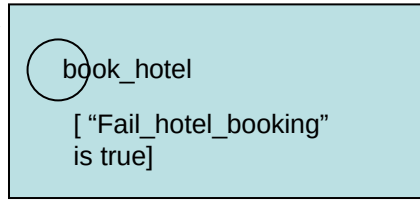
∃ o2. ocurrence_of(o2, book_plane) ∧ subactivity_occ(o2, x) ∧ next_subocc(o1, o2, x)
(if ¬ (prior(*Precond_plane*, o2) ∧ prior(*Input_hotel*, o2))
then (holds(*Failure_plane_booking*, o2) ∧ leaf_occurrence(o2, x))
else (holds(*Eff_plane*, o2) ∧ holds(*Success_plane_booking*, o2) ∧

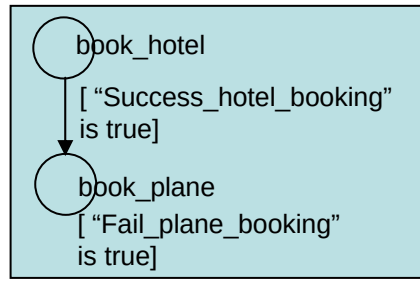
// if the book_plane occurrance succeeded, then there is also an occurrance of register_event

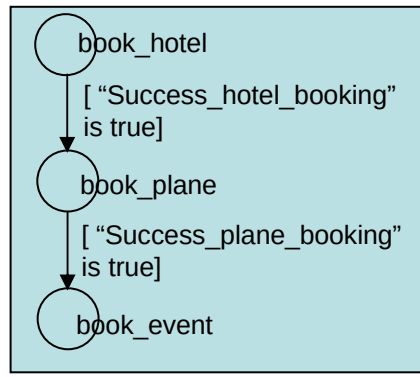
∃ o3. occurrence_of(o3, register_event) ∧ subactivity_occ(o3, x) ∧ next_subocc(o2, o3, x) ∧
(if ¬ (prior(*Precond_event*, o3) ∧ prior(*Input_event*, o3))
then (holds(*Failure_event_booking*, o3) ∧ leaf_occurrence(o3, x)
else (holds(*Eff_event*, o3) ∧ holds(*Success_event_booking*, o3) ∧ leaf_occurrence(o3, x))))))))

// some notational short-hand

Precond_hotel ⇔ ¬ booked_hotel; *Eff_hotel* ⇔ booked_hotel; ...similar for plane and event





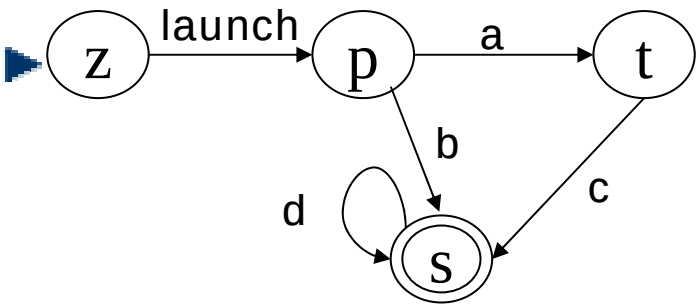


The three activity trees (up to isomorphism) corresponding to composite activity Maria_serv as defined in green box. Maria_serv can be defined in a variety of ways, leading to different (sets of) activity trees

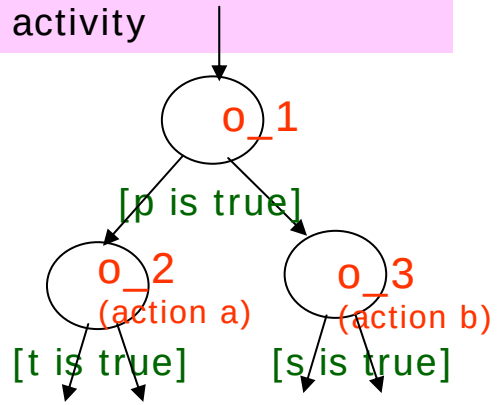
2a. Representing in PSL a complex process, whose internal structure corresponds to an activity-based FSA (sketch)

We illustrate the encoding using an abstract example

- Assume 1 fluent per state, assert that only one state-fluent can be true at a time
- We transform the fsa by adding a new start-state with "launch" activity



FSA M



Parts of (representative) "activity tree" for M

[This tree might be embedded into an "occurrence tree" which represents a family of concurrent activity occurrences]

$\varphi_M(x)$ = (

// initial situation $\exists$ o. occurrence_of(o, launch) $\wedge$ root(o,x) $\wedge$ holds(p, launch)

// for all transitions in FSA M include the following (the following example is for $\delta(p,a) = t$)

$\forall o_1, o_2$ if (subactivity_occurrence(o_1, x) $\wedge$ subactivity_occurrence(o_2, x) $\wedge$
next_subocc(o_1, o_2, x) then (holds(p, o_1) $\wedge$ occurrence_of(o_2, a) $\rightarrow$ holds(t, o_2))

// from a given atomic occurrance, there is at least one child for each transition out of the corresponding state, and no illegal transitions (the following is for atomic occurrance o_1 that corresponds to being in state p)

$\forall o_1$ if (subactivity_occurrence(o_1, x) $\wedge$ holds(p, o_1)
then $\exists$ o_2 (subactivity_occurrence(o_2, x) $\wedge$ next_subocc(o_1, o_2, x) $\wedge$ occurrence_of(o_2, a)
$\wedge$ $\exists$ o_2 (subactivity_occurrence(o_2, x) $\wedge$ next_subocc(o_1, o_2, x) $\wedge$ occurrence_of(o_2, b)
$\wedge$ $\neg$ $\exists$ o_2 (subactivity_occurrence(o_2, x) $\wedge$ next_subocc(o_1, o_2, x) $\wedge$ occurrence_of(o_2, c)

// for all final states include the following (the following example is for s in final states)

$\forall o$ (if leaf_occurrence(o, x) $\rightarrow$ holds(s, o))

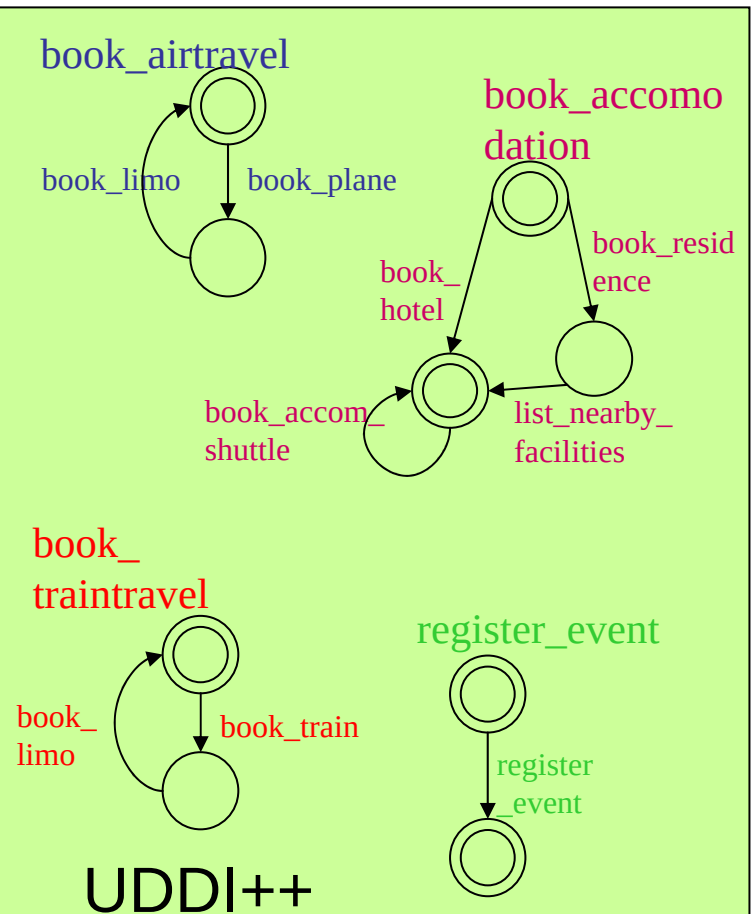
2b. Comments re embedding of FSA descrips into PSL

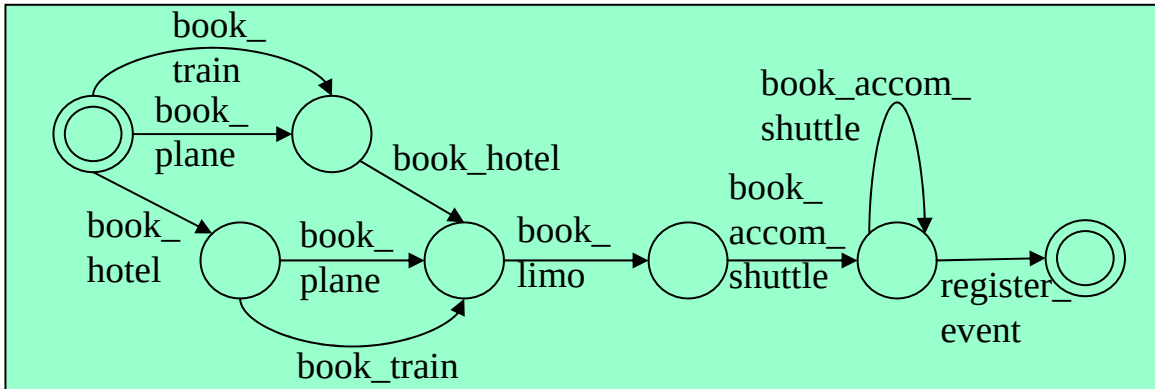
We have sketched a specific way to build up a formula $\varphi_M(.)$ as described informally on prevoius slide

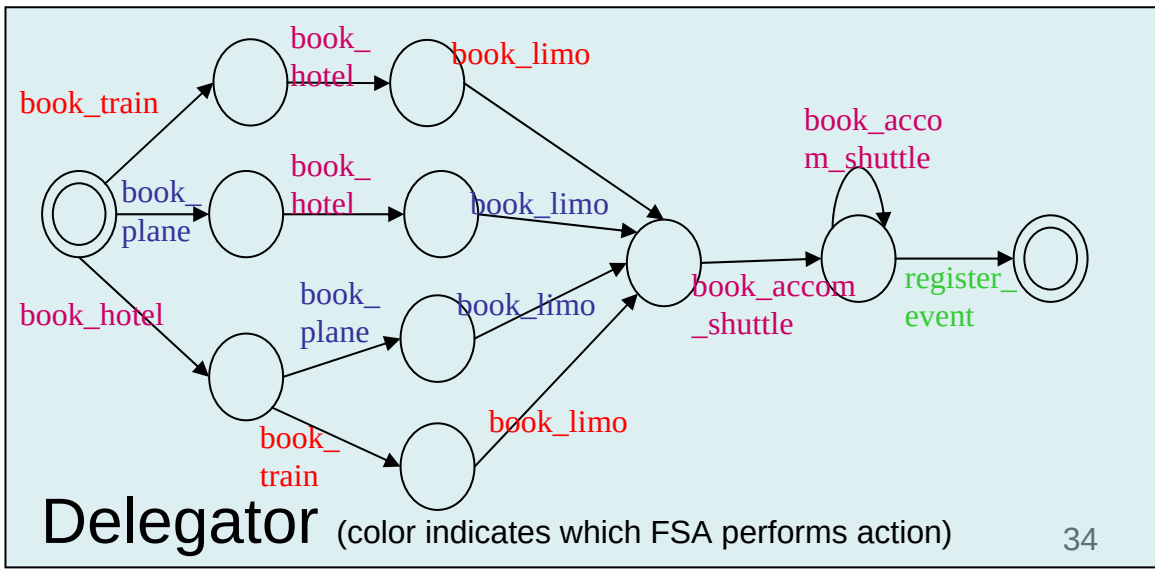
- *Conjecture ("Faithfulness"): x* satisfies formula $\varphi_M(x)$ iff *x* is an activity tree and there is a mapping between accepted words of M and finite branches of *x*.
  - For each word *w* in L(M) at least one finite branch with actions corresponding to *w*
  - For each finite branch β satisfying appropriate fluents at the end, there is a word in L(M) corresponding to β
- Can build similar formula $\chi(x)$ characterizing a single path through the activity tree for M, i.e., (finite branch) *x* satisfies $\chi(x)$ iff *x* corresponds to an accepted word of M
- Can build similar formula $\Psi_M(x,z)$ stating that *x* is the activity tree of M embedded into the occurrence tree *z*
- Given a UDDI+, can build a $\varphi_M(.)$ for each M in the UDDI+
  - Open problem: Can we reify the UDDI+ directory, and talk about member_of(*x*,*U*) ??
- Open problem (informal statement): Is there a "generic" first-order formula $\Gamma(\varphi_M(.), \varphi_N(.))$, such that for arbitrary fsa's M and N and associated formulas $\varphi_M(.)$ and $\varphi_N(.)$, we have $\Gamma(\varphi_M(x), \varphi_N(y))$ iff L(M) = L(N)
  - At a minimum, given fsa's M and N, you can by hand build a formula stating that M and N accept equiv languages

2c. Using automated composition to create re-usable, generic composition of interactive (fsa-based) services

- The base services for this example are richer than for previous example
- (We think that) we can encode multiple FSA's, and describe requirements for a composition (via delegator) to exist (in spirit of "Roman" results)

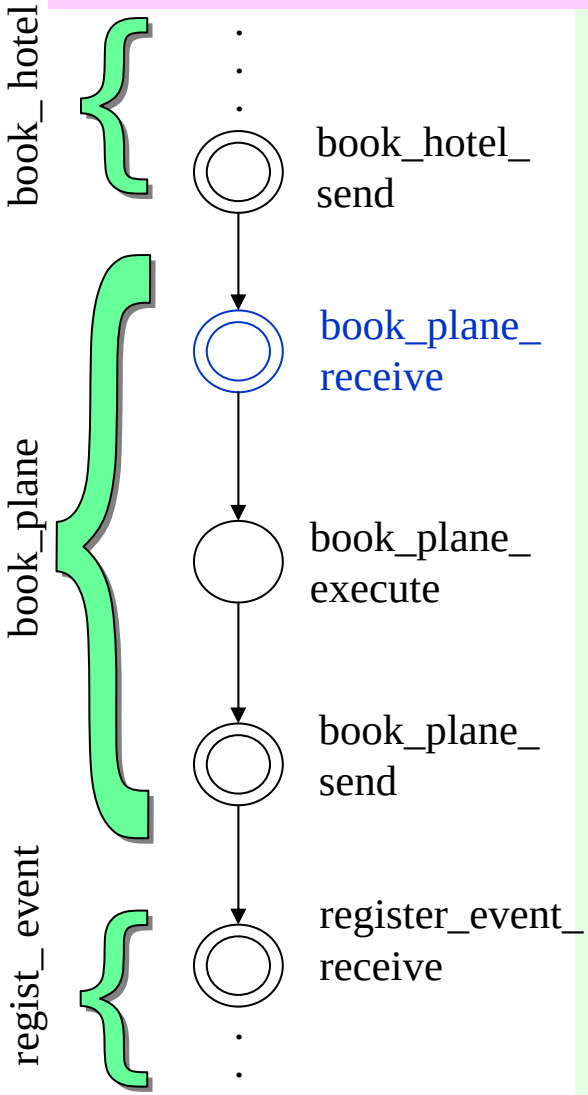






3a. Message Passing between atomic services (illustration in very simple context)

- book_plane assumed to have 3 sub-activities: _receive, _execute, _send
- Use predicate-based fluent "mess_repos(service_name, message_variable)" to hold messages being passed to a service



$\mu(x) \Leftrightarrow$

// basic structure of book_plane

occ_of(x, book_plane) $\wedge$

$\exists$ o1, o2, o3 (sub_act(o1, x) $\wedge$ sub_act(o2, x) $\wedge$ sub_act(o3, x) $\wedge$
occ_of(o1, book_plane_rec) $\wedge$
occ_of(o2, book_plane_exec) $\wedge$
occ_of(o3, book_plane_send) $\wedge$

// "glue" between book_hotel and book_plane

($\exists$o4 o5 occ_of(o5, reg_event) $\wedge$ sub_act(o4, o5) $\wedge$
occ_of(o4, reg_event_send) $\wedge$ leaf_occ(o4, o5) $\wedge$
next_subocc(o1, o4)) $\wedge$

// reading from message repository

($\exists$m', v', m'', v'', m''', v''', m'''', v''''
(prior (mess_repos(book_plane, m'), o1) $\wedge$
mess_type(m', departure_city) $\wedge$ mess_value(m', v') $\wedge$
$\neg$holds(mess_repos(book_plane, m'), o1) $\wedge$
... /* similar for m'', m''', m'''' */) $\wedge$

// execution of book_plane_execute ...

// sending messages to regist_event ...

// "glue" between book_plane and register_event

3b. Expressing Constraints on Data Flow

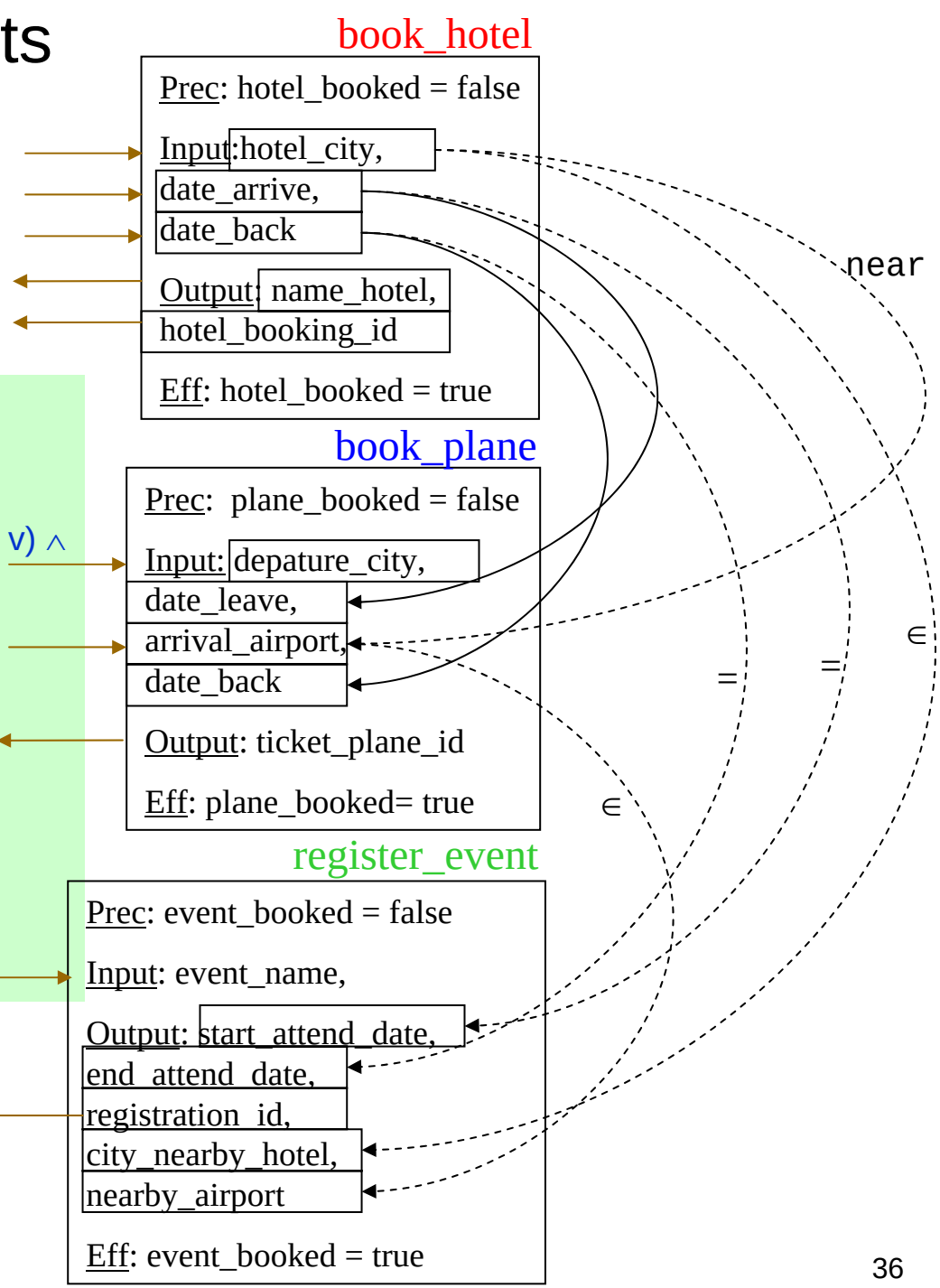
- Can express variety of data flow constraints
- Assume the 3 atomic services as on previous slide

// Values passed from book_hotel to book_plane

o is occ of composite service
o1 is occ of book_plane_receive …
∃i, m, v (input_type(i, date_arrive) ∧ input_value(i, v) ∧
mess_type(m, date_leave) ∧ mess_value(m, v) ∧
prior(mess_repos(comp_service, i), o) ∧
prior (mess_repos(book_plane, m), o1)

// Constraint between input values

o is occ of composite service
o1 is occ of book_hotel; o2 is occ of book_plane …
∃i, i', v,v' (
input_type(i, date_arrive) ∧ input_value(i, v) ∧
input_type(i', date_leave) ∧ input_value(i', v') ∧
element_of(v, v')



Legend

- data in/out of composite service
- data flow within composite service
- constraint on data flowing within composite service