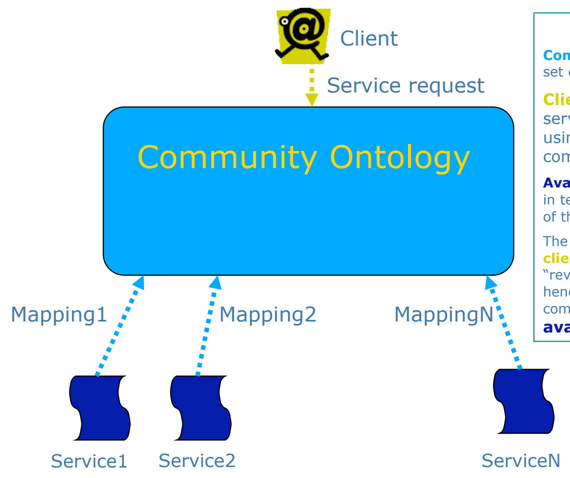
Composition: the "Roman" Approach

The Roman Approach



Client-tailored!

Community ontology: just a set of **actions**

Client formulates the service it requires as a **TS** using the **actions** of the common ontology

Available services: described in terms of a **TS** using **actions** of the community ontology

The **community** realizes the **client's target service** by "reversing" the mapping and hence using **fragments** of the computation of the the **available services**

Community of Services

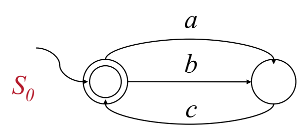
- A community of Services is
  - a set of services …
  - … that share implicitly a *common understanding* on a common set of actions (common ontology limited to the alphabet of actions)…
  - … and export their behavior using (finite) TS over this common set of actions
- A client specifies needs as a service behavior, i.e, a (finite) TS using the common set of actions of the community

(Target & Available) Service TS

- We model services as finite TS $T = (\Sigma, S, s^0, \delta, F)$ with
  - single initial state *(s^0)*
  - deterministic transitions *(i.e., δ is a partial function from $S \times \Sigma$ to S)*

Note: In this way the client entirely controls/chooses the transition to execute

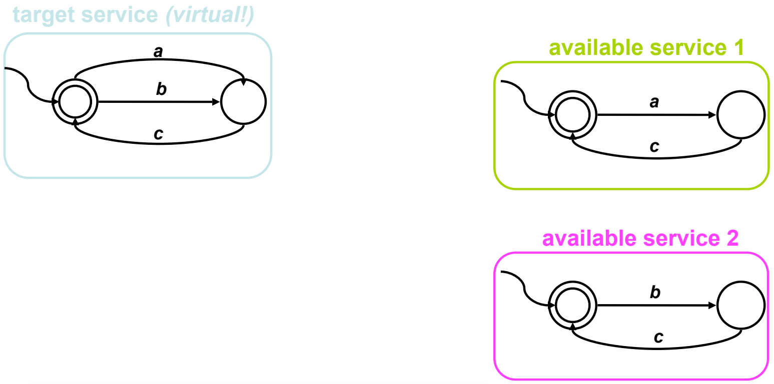
Example:



a: *"search by author (and select)"*
b: *"search by title (and select)"*
c: *"listen (the selected song)"*

Composition: an Example

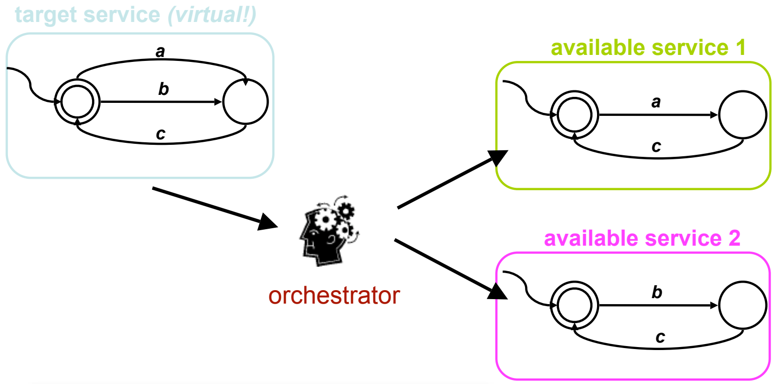
Dipartimento di Informatica e Sistemistica "Antonio Ruberti"
SAPIENZA Università di Roma



Lets get some intuition of what a composition is through an example

Composition: an Example

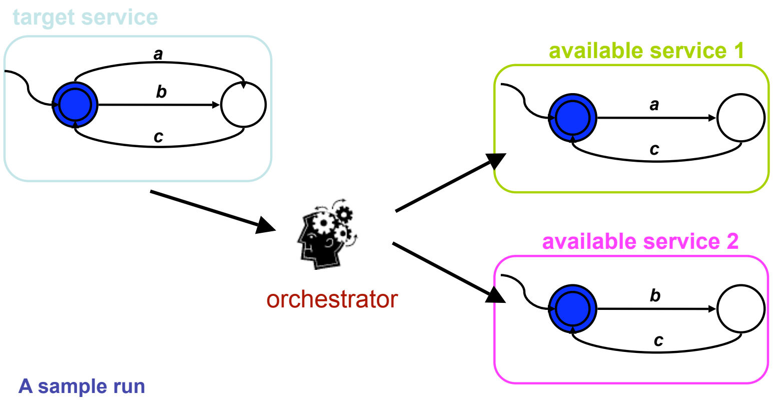
Dipartimento di Informatica e Sistemistica "Antonio Ruberti"
SAPIENZA Università di Roma



Lets get some intuition of what a composition is through an example

Composition: an Example

Dipartimento di Informatica e Sistemistica "Antonio Ruberti"
SAPIENZA Università di Roma



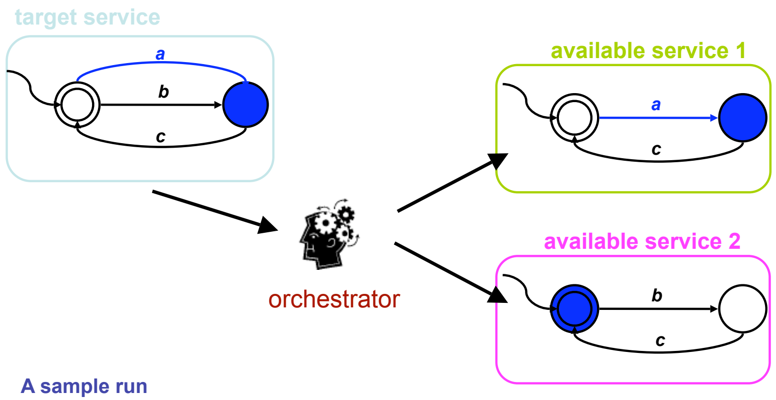
A sample run

action request:

orchestrator response:

Composition: an Example

Dipartimento di Informatica e Sistemistica "Antonio Ruberti"
SAPIENZA Università di Roma

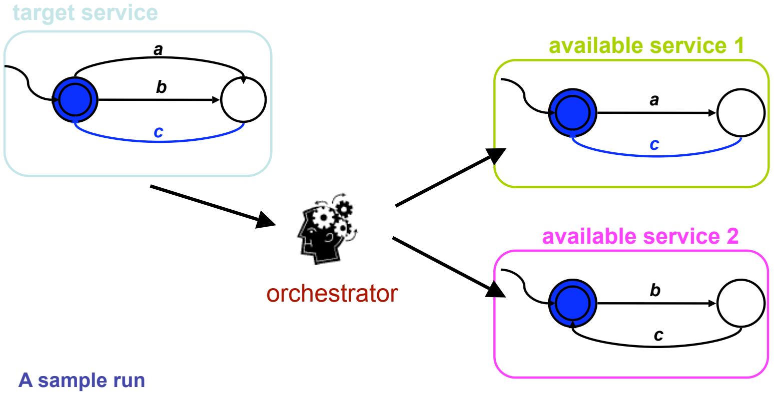


A sample run

action request: *a*

orchestrator response: ***a*,1**

Composition: an Example

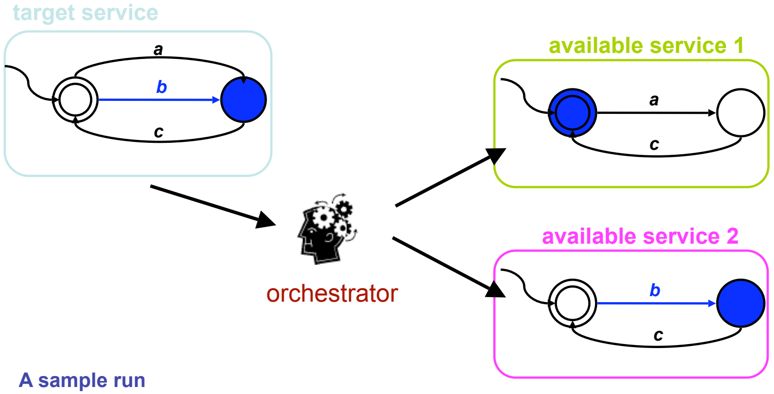


A sample run

action request: *a* *c*

orchestrator response: *a,1* *c,1*

Composition: an Example

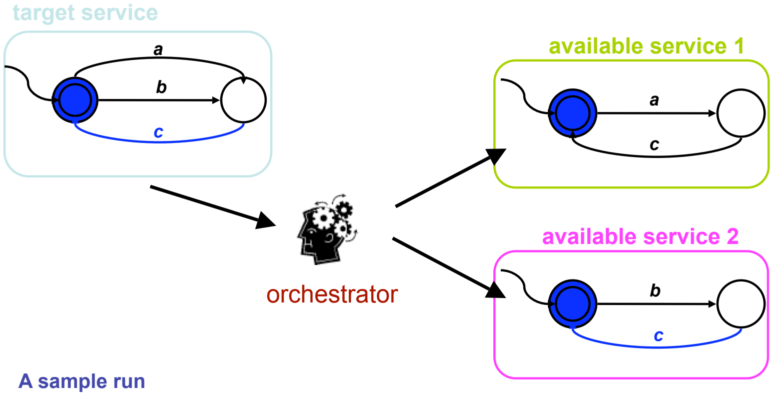


A sample run

action request: *a* *c* *b*

orchestrator response: *a,1* *c,1* *b,2*

Composition: an Example

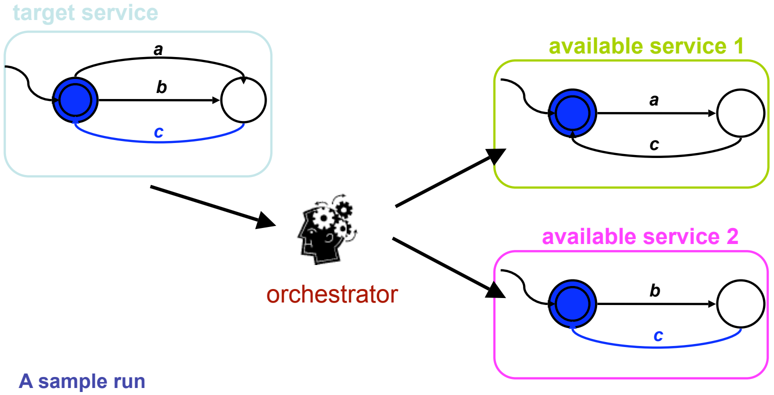


A sample run

action request: *a* *c* *b* *c*

orchestrator response: *a,1* *c,1* *b,2* *c,2*

Composition: an Example

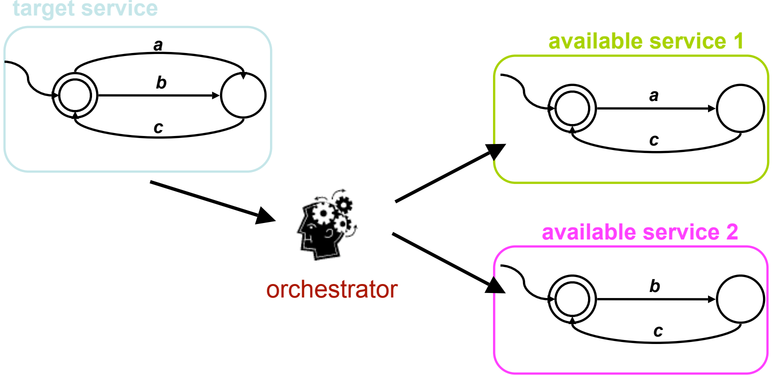


A sample run

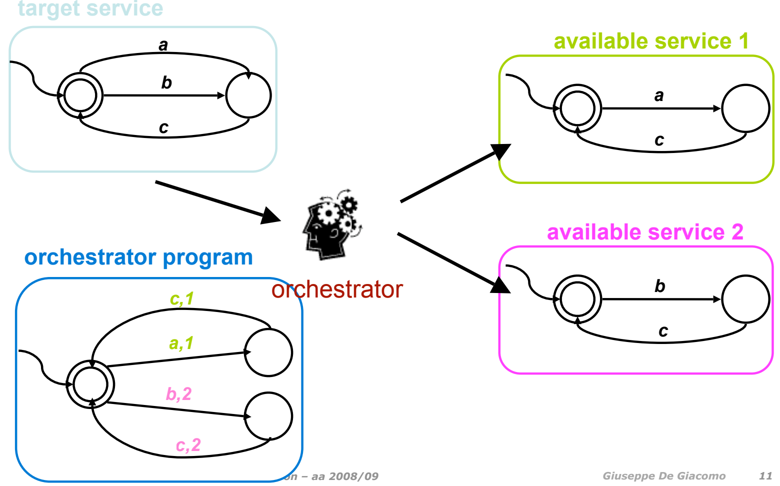
action request: *a* *c* *b* *c* …

orchestrator response: *a,1* *c,1* *b,2* *c,2*

A orchestrator program realizing the target behavior



A orchestrator program realizing the target behavior



Orchestrator programs

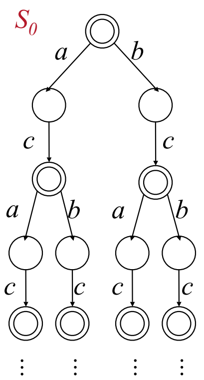
- **Orchestrator program** is *any function* *P(h,a) = i* that takes a **history** *h* and an **action** *a* to execute and **delegates** *a* to one of the available services *i*
- A **history** is the sequence of actions done so far:

  $h = a_1 a_2 \ldots a_k$

- Observe that to take a decision *P* has **full access to the past**, but no access to the future
  - *Note given an history $h = a_1 a_2 \ldots a_k$ an the function P we can reconstruct the state of the target service and of each available service*
    - *$a_1 a_2 \ldots a_k$ determines the state of the target service*
    - *$(a_1, P([\,], a_1))(a_2, P([a_1], a_2)) \ldots (a_k, P([a_1 a_2 \ldots a_{k-1}], a_k))$ determines the state of each 1vailable service*
- ***Problem: synthesize a orchestrator program P that realizes the target service making use of the available services***

Service Execution Tree

By "unfolding" a (finite) TS one gets an (infinite) execution tree
-- yet another (infinite) TS which bisimilar to the original one)



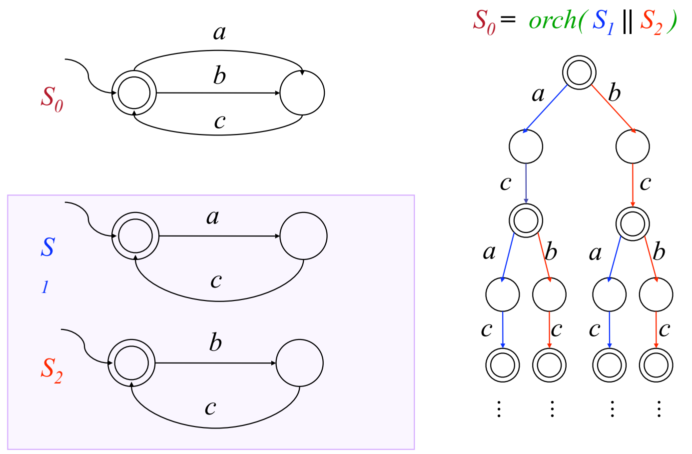
- *Nodes: history i.e., sequence of actions executed so far*
- *Root: no action yet performed*
- *Successor node x·a of x: action a can be executed after the sequence of action x*
- *Final nodes: the service can terminate*

Alternative (but Equivalent) Definition of Service Composition

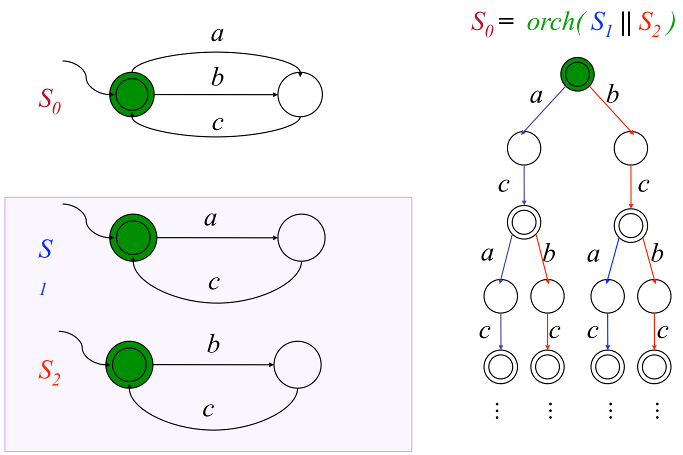
Composition:
- coordinating program …
- … that realizes the target service …
- … by suitably coordinating available services

⇒ Composition can be seen as:
- a labeling of the execution tree of the **target service** such that …
- … each action in the execution tree is labeled by the available service that executes it …
- … and each possible sequence of actions on the target service execution tree corresponds to possible sequences of actions on the available service execution trees, suitably interleaved

Example of Composition

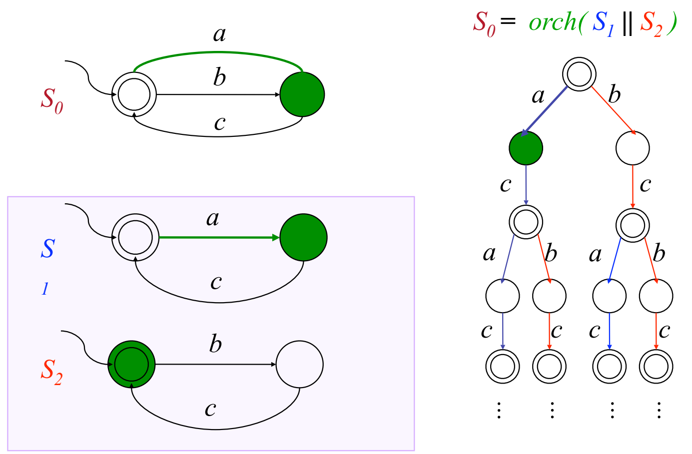


Example of Composition



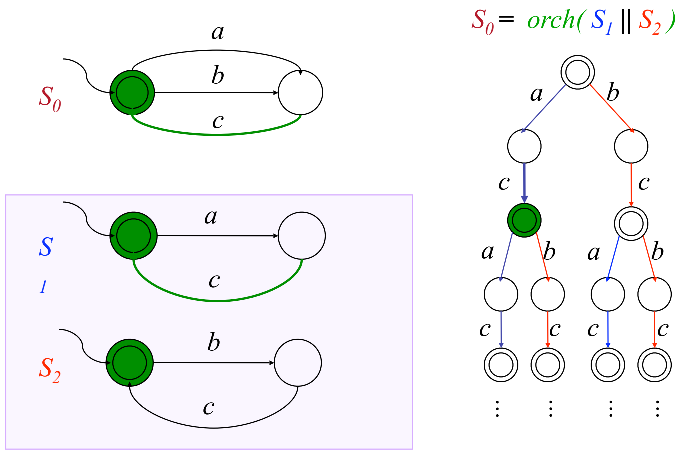
All services start from their starting state

Example of Composition (5)



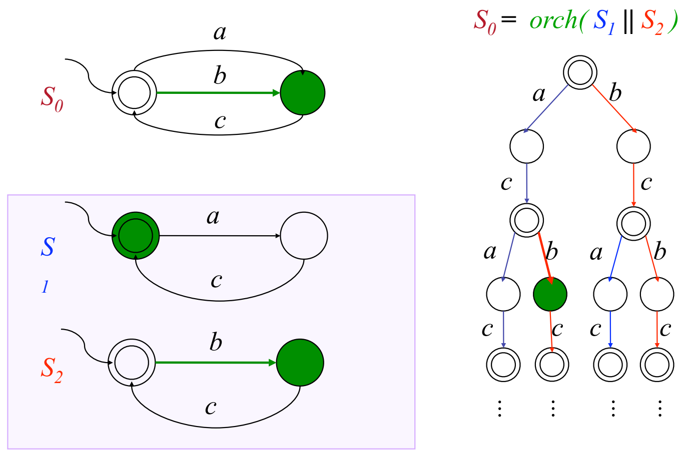
Each action of the target service is executed by at least one of the component services

Example of composition (6)

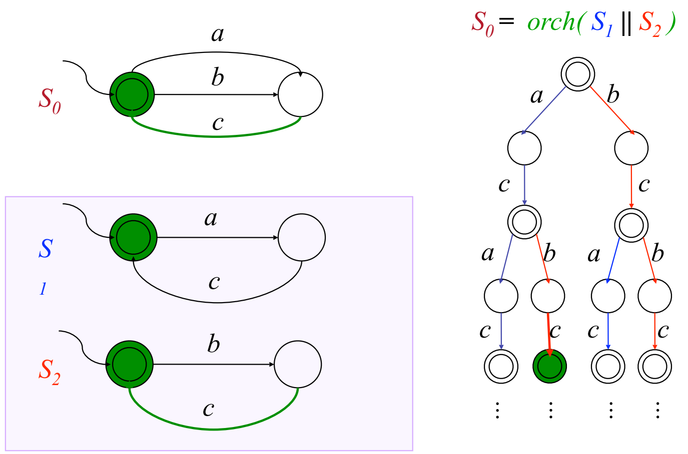


When the target service can be left, then all component services must be in a final state

Example of composition (7)

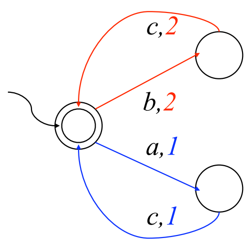


Example of composition (8)



Observation

- This labeled execution tree has a finite representation as a finite TS …
- …with transitions labeled by an action and the service performing the action



Is this always the case when we deal with services expressible as finite TS? See later…

Questions

Assume services of community and target service are finite TSs

- Can we always check composition existence?
- If a composition exists there exists one which is a finite TS?
- If yes, how can a finite TS composition by computed?

To answer ICSOC'03 exploits PDL SAT

Answers

Reduce service composition synthesis to satisfability in (deterministic) PDL

- Can we always check composition existence?
  Yes, SAT in PDL is decidable in EXPTIME
- If a composition exists there exists one which is a finite TS?
  Yes, by the small model property of PDL
- How can a finite TS composition be computed?
  From a (small) model of the corresponding PDL formula

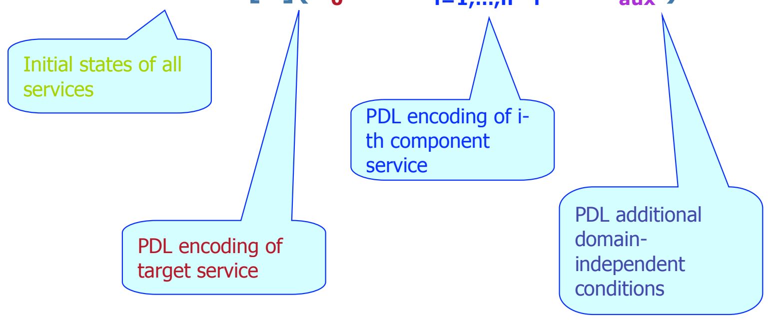
Encoding in PDL

Basic idea:

- A orchestrator program *P* realizes the target service *T* iff at each point:
  - ∀ transition labeled *a* of the target service *T* …
  - … ∃ an available service B_i (the one chosen by *P*) that can make an *a*-transition, realizing the *a*-transition of *T*
- Encoding in PDL:
  - ∀ transition labeled *a* … use **branching**
  - ∃ an available service B_i that can make an *a*-transition … use underspecified predicates **assigned through SAT**

Structure of the PDL Encoding

$$\Phi = \mathbf{Init} \wedge [\mathbf{u}](\Phi_0 \wedge \bigwedge_{i=1,\ldots,n}\Phi_i \wedge \Phi_{aux})$$



PDL encoding is polynomial in the size of the service TSs

PDL Encoding

- Target service $S_0 = (\Sigma, S_0, s^0_0, \delta_0, F_0)$ in PDL we define Φ_0 as the conjunction of:
  - $s \rightarrow \neg s'$ for all pairs of distinct states in S_0
    service states are pair-wise disjoint
  - $s \rightarrow \langle a \rangle T \wedge [a]s'$ for each $s'=\delta_0(s,a)$
    target service can do an a-transition going to state s'
  - $s \rightarrow [a] \perp$ for each $\delta_0(s,a)$ undef.
    target service cannot do an a-transition
  - $F_0 \equiv \vee_{s \in F0}\, s$
    denotes target service final states
- …

PDL Encoding (cont.d)

- available services $S_i = (\Sigma, S_i, s^0_i, \delta_i, F_i)$ in PDL we define Φ_i as the conjunction of:
  - $s \rightarrow \neg s'$ for all pairs of distinct states in S_i
    Service states are pair-wise disjoint
  - $s \rightarrow [a](moved_i \wedge s' \vee \neg\, moved_i \wedge s)$ for each $s'=\delta_i(s,a)$
    if service moved then new state, otherwise old state
  - $s \rightarrow [a](\neg\, moved_i \wedge s)$ for each $\delta_i(s,a)$ undef.
    if service cannot do a, and a is performed then it did not move
  - $F_i \equiv \vee_{s \in Fi}\, s$
    denotes available service final states
- …

PDL Encoding (cont.d)

- Additional assertions Φ_{aux}
  - $\langle a \rangle T \rightarrow [a] \vee_{i=1,\dots,n} moved_i$ for each action a
    at least one of the available services must move at each step
  - $F_0 \rightarrow \wedge_{i=1,\dots,n} F_i$
    when target service is final all comm. services are final
  - $Init \equiv s^0_0 \wedge_{i=1,\dots,n} s^0_i$
    Initially all services are in their initial state

PDL encoding: $\Phi = Init \wedge [u](\Phi_0 \wedge_{i=1,\dots,n} \Phi_i \wedge \Phi_{aux})$

Results

Thm[ICSOC'03,IJCIS'05]:
Composition exists iff PDL formula Φ SAT

From composition labeling of the target service one can build a tree model of the PDL formula and viceversa

Information on the labeling is encoded in predicates $moved_i$

Corollary [ICSOC'03,IJCIS'05]:
Checking composition existence is decidable in **EXPTIME**

Thm[Muscholl&Walukiewicz FoSSaCS'07]:
Checking composition existence is **EXPTIME-hard**

Results on TS Composition

Thm[ICSOC'03,IJCIS'05]:

If composition exists then finite TS composition exists.

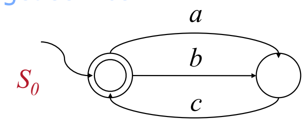
From a small model of the PDL formula Φ, one can build a finite TS machine

Information on the output function of the machine is encoded in predicates $moved_i$

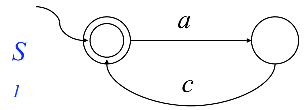
⇒ finite TS composition existence of services expressible as finite TS is EXPTIME-complete

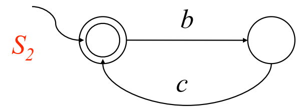
Example (1)

Target service



Available services





PDL

...

...

...

$s_0^0 \wedge s_1^0 \wedge s_2^0$

$\langle a \rangle T \rightarrow [a] (moved_1 \vee moved_2)$

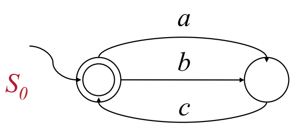
$\langle b \rangle T \rightarrow [b] (moved_1 \vee moved_2)$

$\langle c \rangle T \rightarrow [c] (moved_1 \vee moved_2)$

$F_0 \rightarrow F_1 \wedge F_2$

Example (2)

Target service



$s_0^0 \rightarrow \neg s_0^1$

$s_0^0 \rightarrow \langle a \rangle T \wedge [a] s_0^1$

$s_0^0 \rightarrow \langle b \rangle T \wedge [b] s_0^1$

$s_0^1 \rightarrow \langle c \rangle T \wedge [c] s_0^0$

$s_0^0 \rightarrow [c] \bot$

$s_0^1 \rightarrow [a] \bot$

$s_0^1 \rightarrow [b] \bot$

$F_0 \equiv s_0^0$

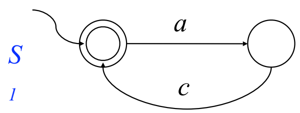
...

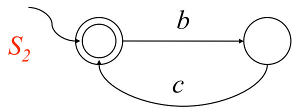
...

...

Example (3)

Available services





...

$s_1^0 \rightarrow \neg s_1^1$

$s_1^0 \rightarrow [a] (moved_1 \wedge s_1^1 \vee \neg moved_1 \wedge s_1^0)$

$s_1^0 \rightarrow [c] \neg moved_1 \wedge s_1^0$

$s_1^0 \rightarrow [b] \neg moved_1 \wedge s_1^0$

$s_1^1 \rightarrow [a] \neg moved_1 \wedge s_1^1$

$s_1^1 \rightarrow [b] \neg moved_1 \wedge s_1^1$

$s_1^1 \rightarrow [c] (moved_1 \wedge s_1^0 \vee \neg moved_1 \wedge s_1^1)$

$F_1 = s_1^0$

$s_2^0 \rightarrow \neg s_2^1$

$s_2^0 \rightarrow [b] (moved_2 \wedge s_2^1 \vee \neg moved_2 \wedge s_2^0)$

$s_2^0 \rightarrow [c] \neg moved_2 \wedge s_2^0$

$s_2^0 \rightarrow [a] \neg moved_2 \wedge s_2^0$

$s_2^1 \rightarrow [b] \neg moved_2 \wedge s_2^1$

$s_2^1 \rightarrow [a] \neg moved_2 \wedge s_2^1$

$s_2^1 \rightarrow [c] (moved_2 \wedge s_2^0 \vee \neg moved_2 \wedge s_2^1)$

$F_2 = s_2^0$

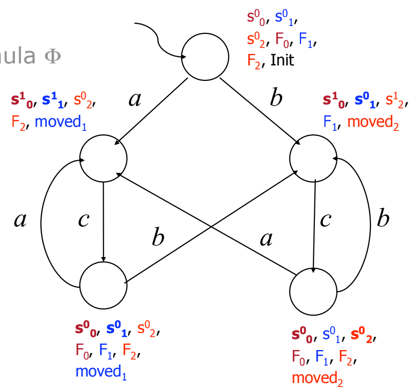
Example (4)

Check: run SAT on PDL formula Φ

Example

Check: run SAT on PDL formula Φ

Yes ⇒ (small) model

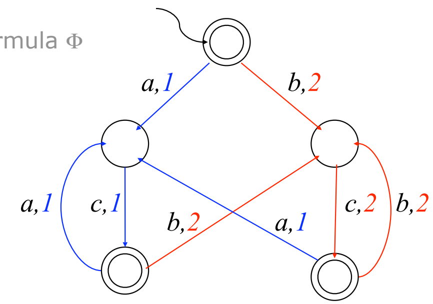


Example

Check: run SAT on PDL formula Φ

Yes ⇒ (small) model

⇒ extract finite TS



Example

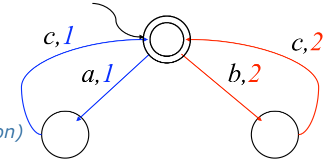
Check: run SAT on PDL formula Φ

Yes ⇒ (small) model

⇒ extract finite TS

⇒ minimize finite TS

(similar to Mealy machine minimization)



Results on Synthesizing Composition

- Using PDL reasoning algorithms based on model construction (cf. tableaux), build a (small) model
  Exponential in the size of the PDL encoding/services finite TS

  Note: SitCalc, etc. can compactly represent finite TS,
  PDL encoding can preserve compactness of representation

- From this model extract a corresponding finite TS
  Polynomial in the size of the model

- Minimize such a finite TS using standard techniques (opt.)
  Polynomial in the size of the TS

  Note: finite TS extracted from the model is not minimal
  because encodes output in properties of individuals/states

Tools for Synthesizing Composition

- In fact we use only a fragment of PDL in particular we use fixpoint (transitive closure) only to get the universal modality …

- … thanks to a tight correspondence between PDLs and Description Logics (DLs), we can use current highly optimized DL reasoning systems to do synthesis …
  Racer, Pellet, Fact++

- … when the ability or returning models will be added …

- … meanwhile we can check for composition existence using such tools.