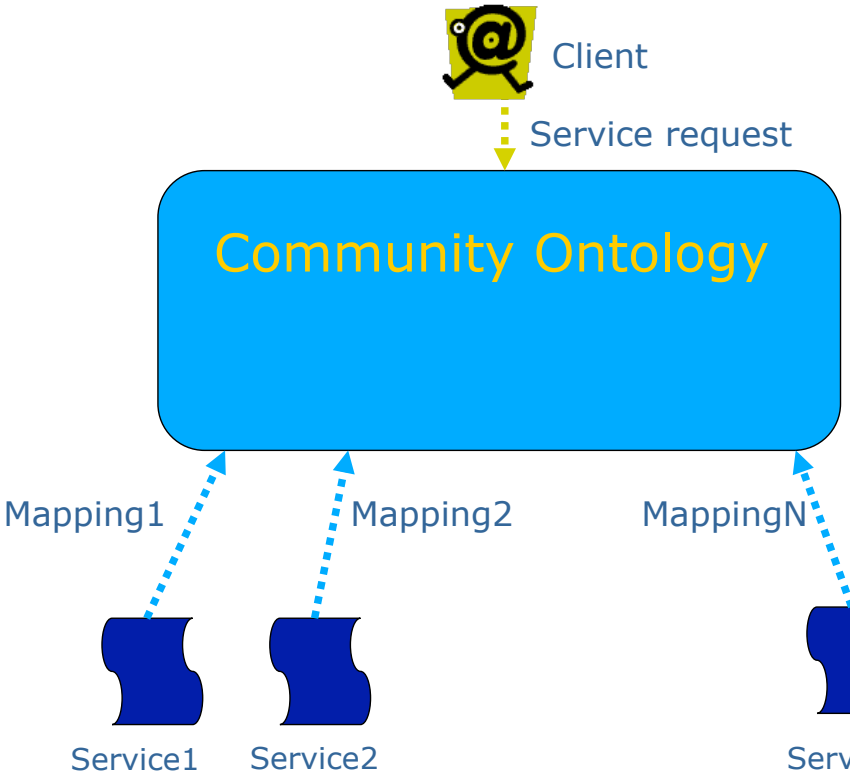
Composition: the "Roman" Approach

Name by
Rick Hull
IBM Research

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

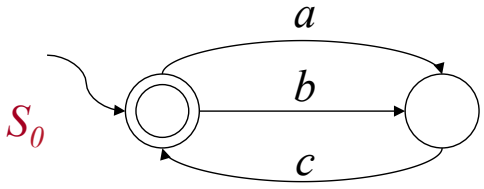
- A community of Services is
  - a set of services …
  - … that share implicitly a *common understanding* on a common set of actions (common ontology limited to the alphabet of actions)…
  - … and export their behavior using (finite) TS over this common set of actions
- A client specifies needs as a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

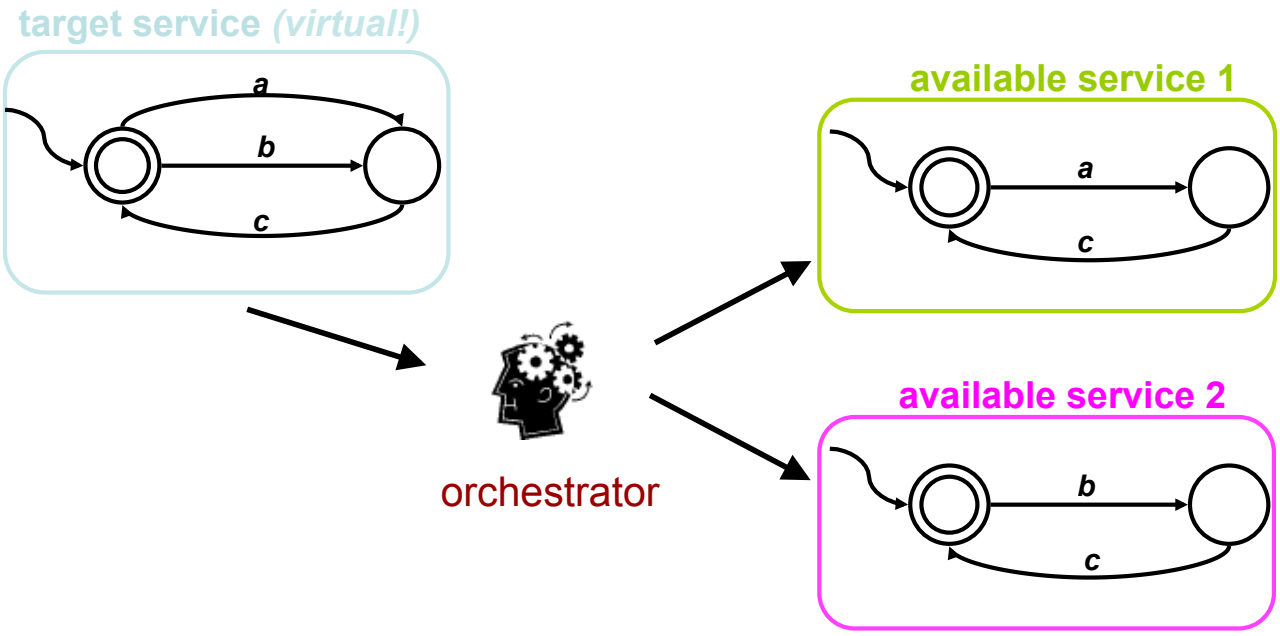
Note: In this way the client entirely controls/chooses the transition to execute

Example:



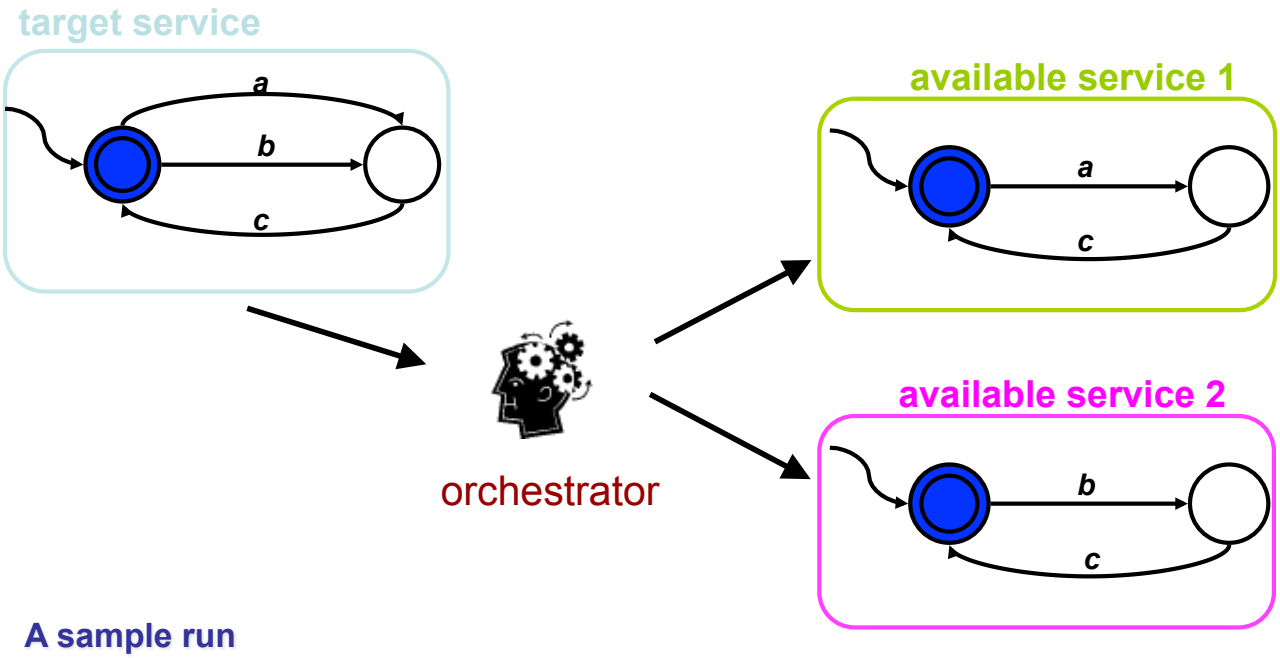
a: "search by author (and select)"
b: "search by title (and select)"
c: "listen (the selected song)"

Composition: an Example



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

Composition: an Example

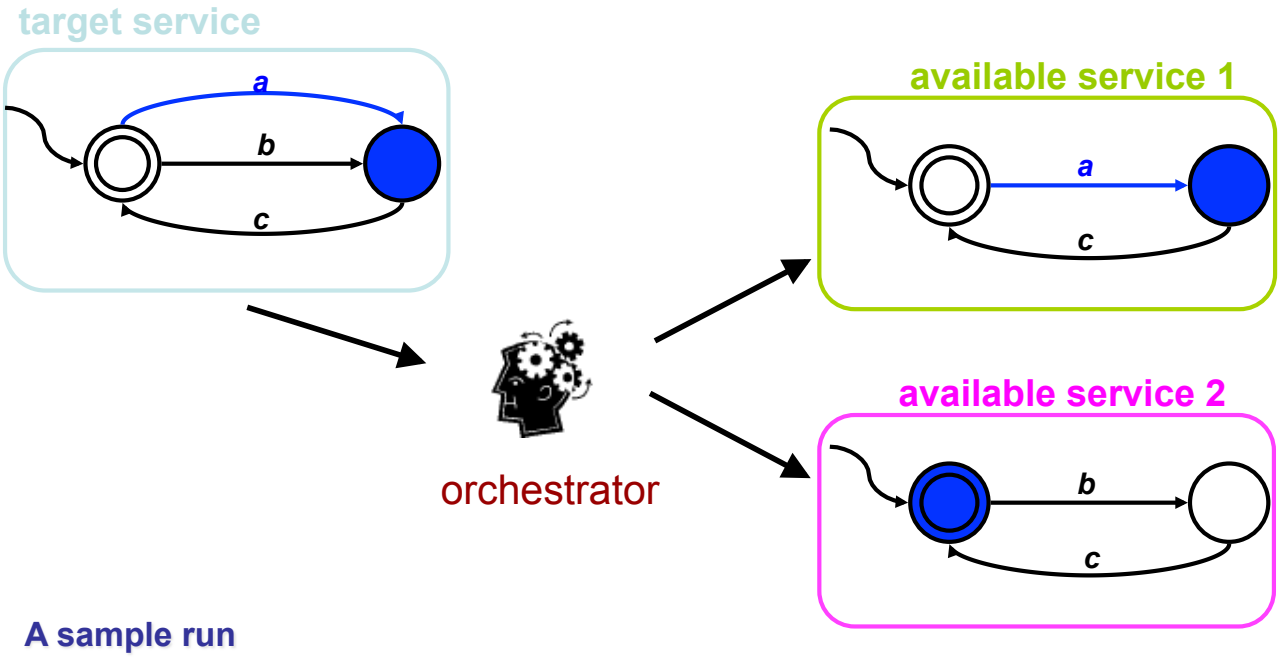


A sample run

action request:

orchestrator response:

Composition: an Example

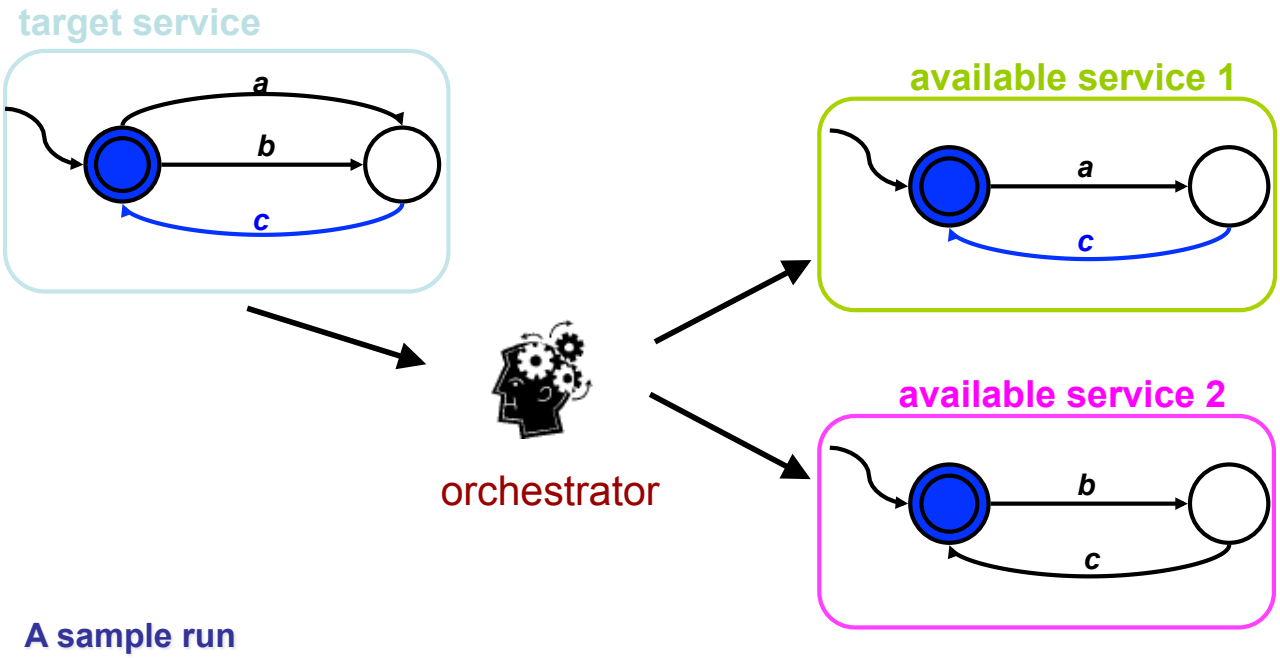


A sample run

action request: *a*

orchestrator response: *a,1*

Composition: an Example

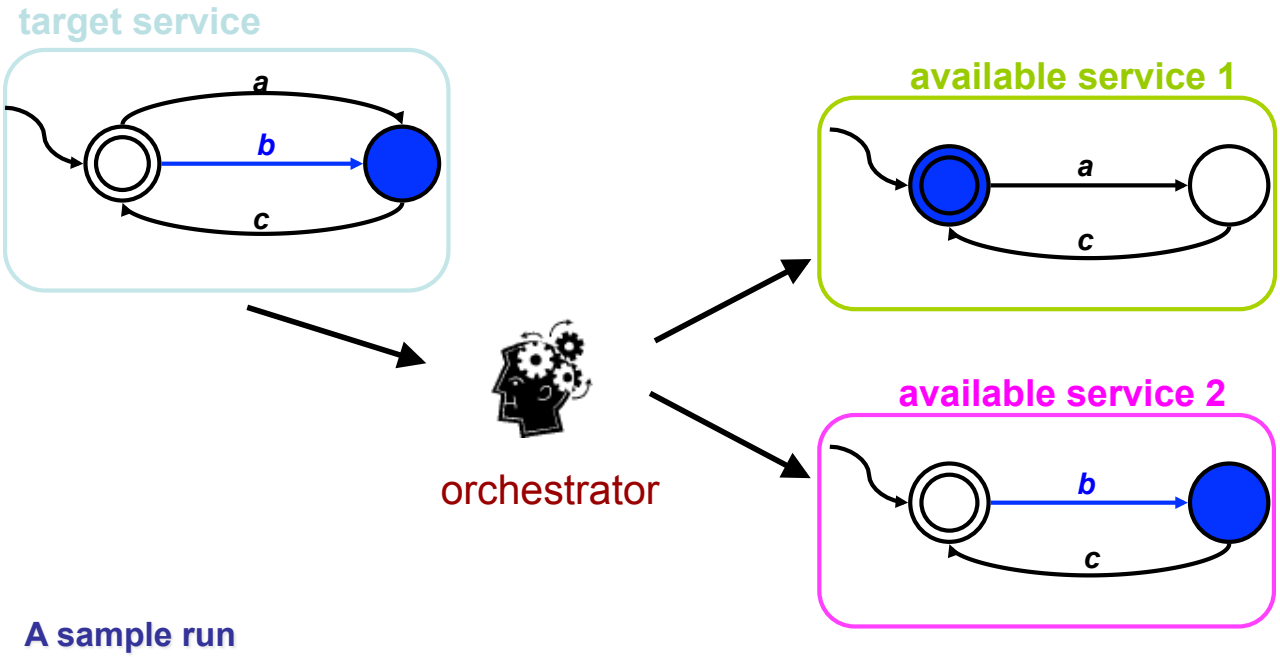


A sample run

action request: *a* *c*

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

Composition: an Example

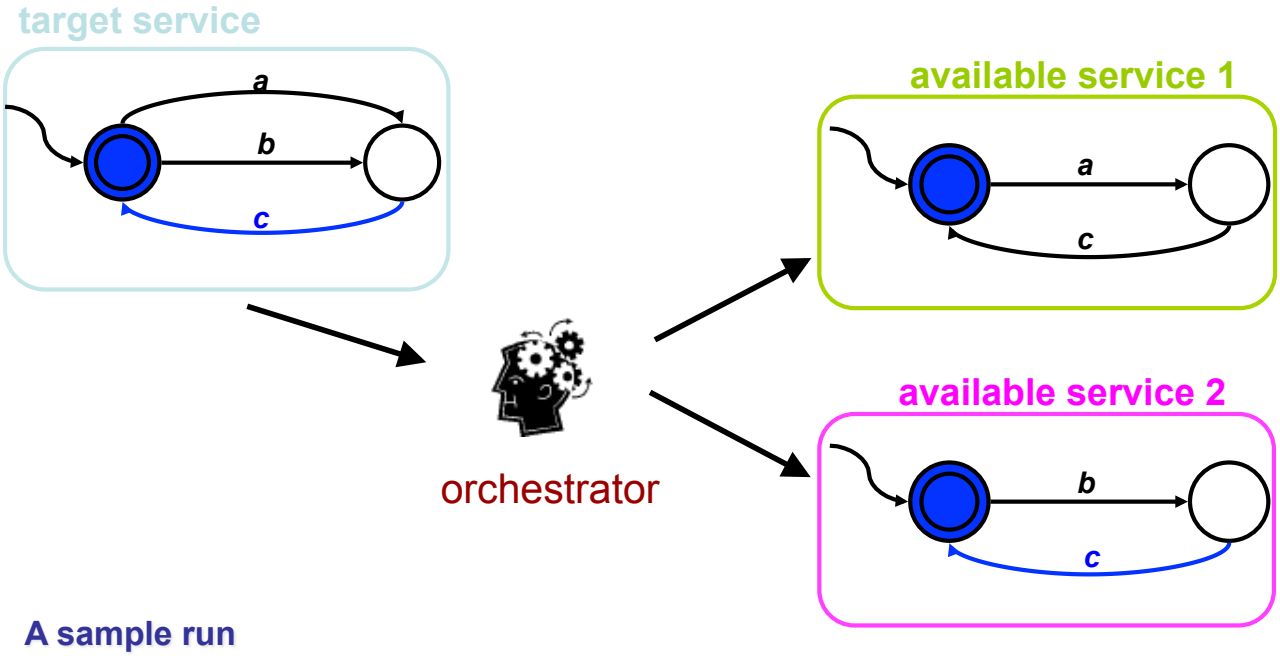


A sample run

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

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

Composition: an Example

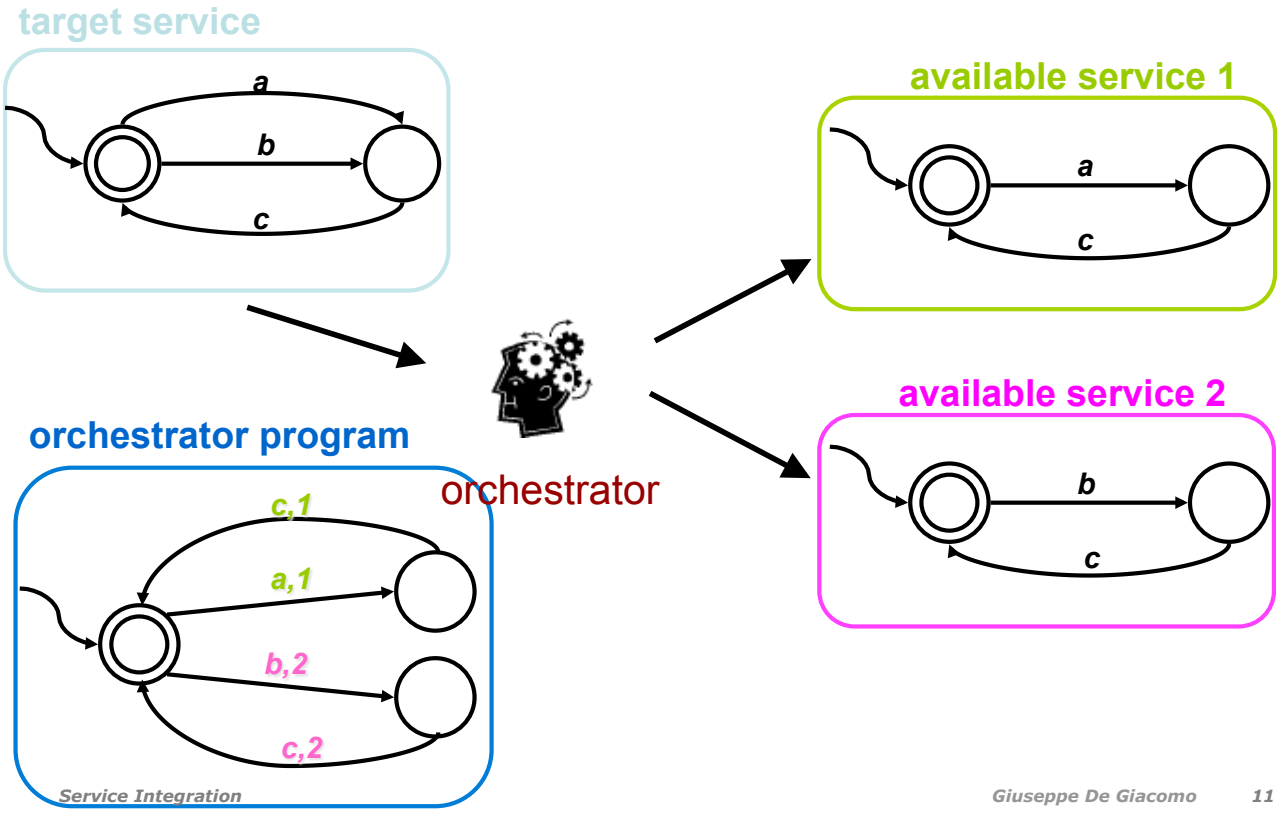


A sample run

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

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

A orchestrator program realizing the target behavior



Orchestrator programs

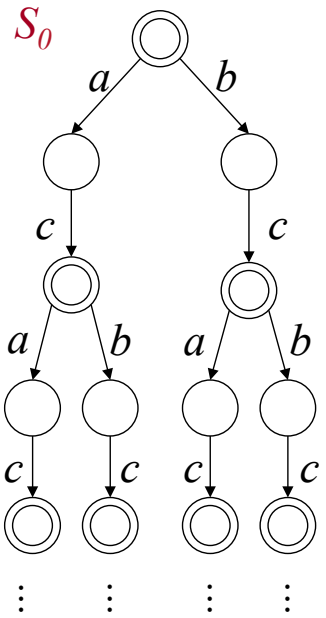
- **Orchestrator program** is *any function P(h,a) = i* that takes a **history** *h* and an **action** *a* to execute and **delegates** *a* to one of the available services *i*
- A **history** is the sequence of actions done so far:

  $h = a_1 a_2 \dots a_k$

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



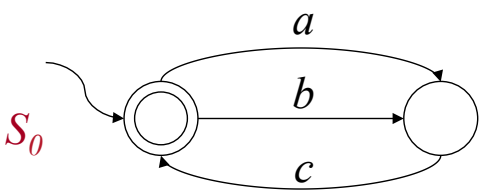
- *Nodes: history i.e., sequence of actions executed so far*
- *Root: no action yet performed*
- *Successor node x·a of x: action a can be executed after the sequence of action x*
- *Final nodes: the service can terminate*

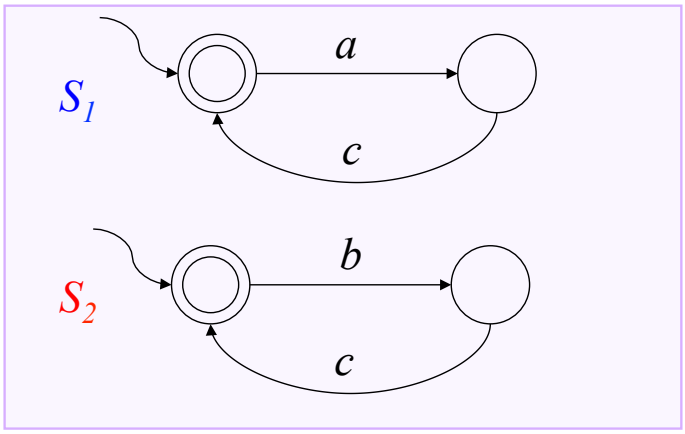
Alternative (but Equivalent) Definition of Service Composition

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

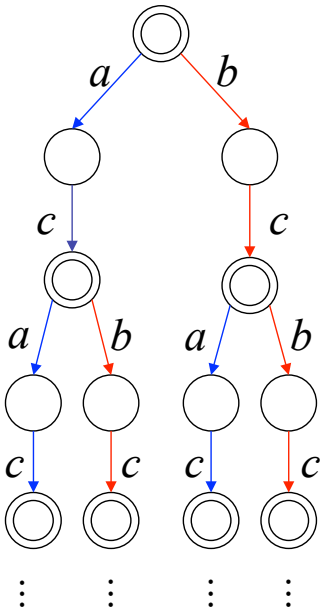
⇒ Composition can be seen as:
- a labeling of the execution tree of the **target service** such that …
- … each action in the execution tree is labeled by the available service that executes it …
- … and each possible sequence of actions on the target service execution tree corresponds to possible sequences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

Example of Composition

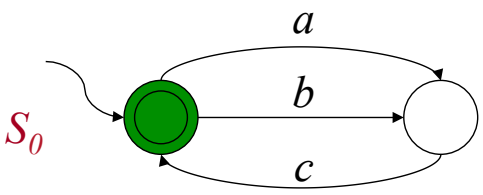


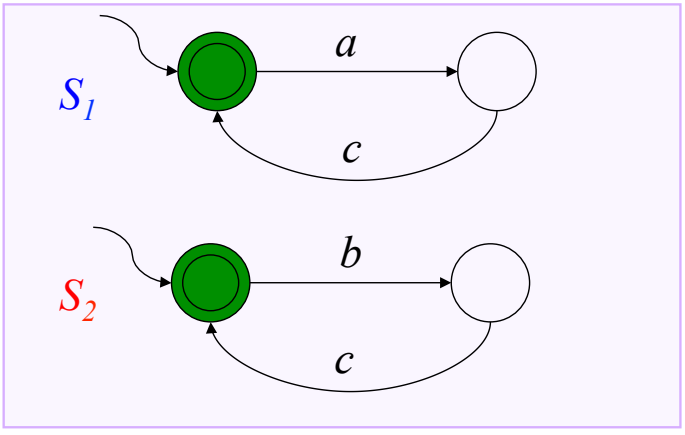


$S_0 = orch(S_1 \| S_2)$

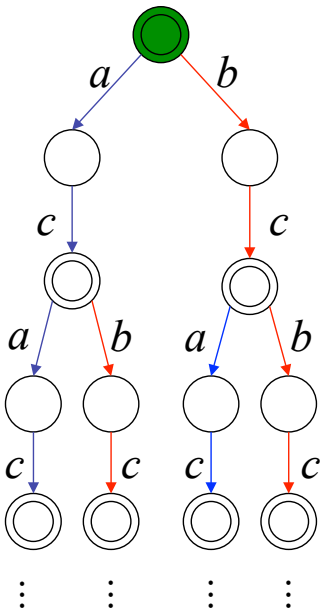


Example of Composition



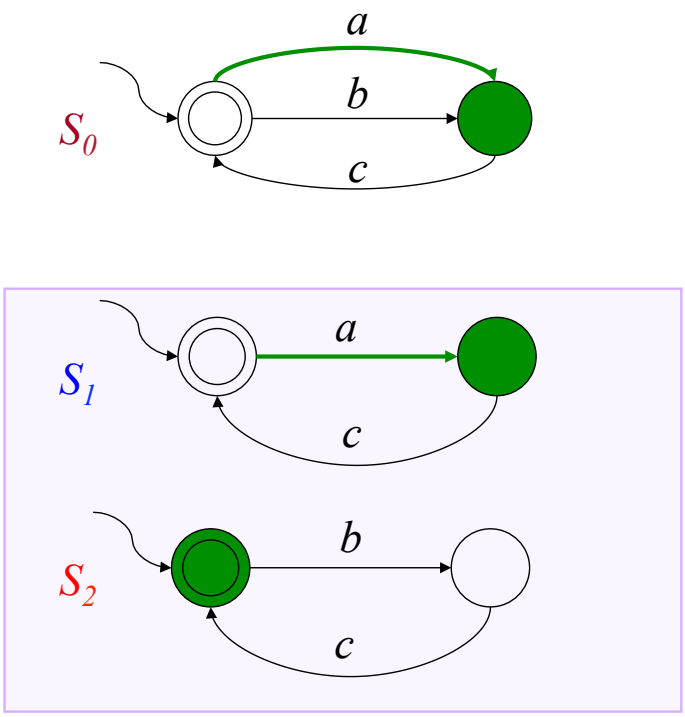


$S_0 = orch(S_1 \| S_2)$

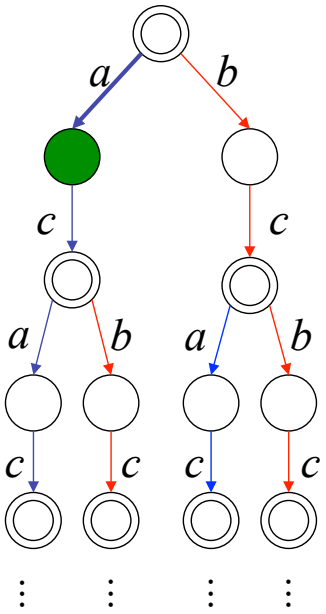


All services start from their starting state

Example of Composition (5)

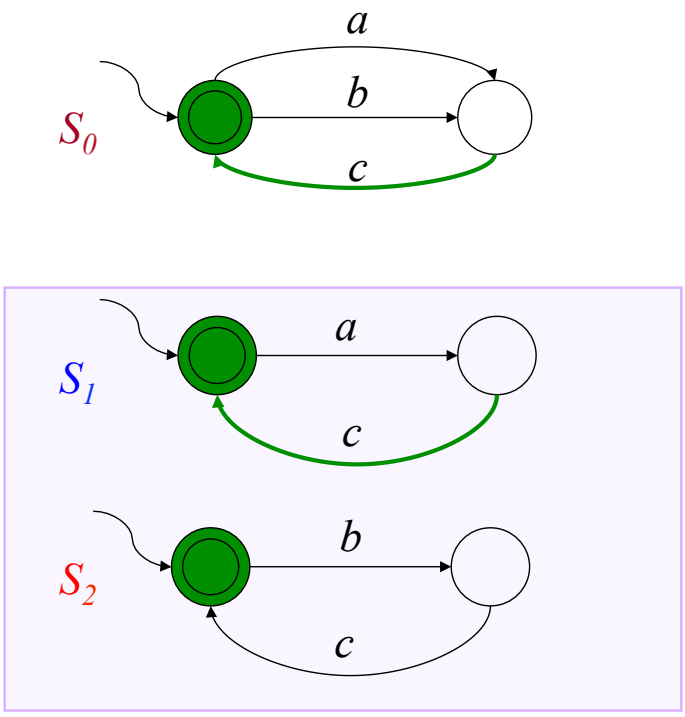


$S_0 = orch(S_1 \| S_2)$

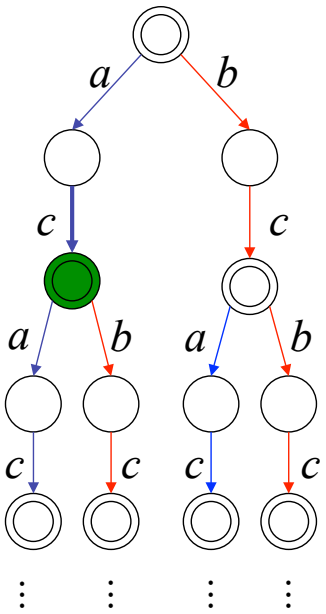


Each action of the target service is executed by at least one of the component services

Example of composition (6)

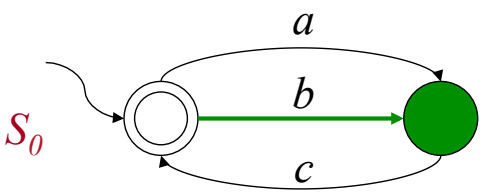


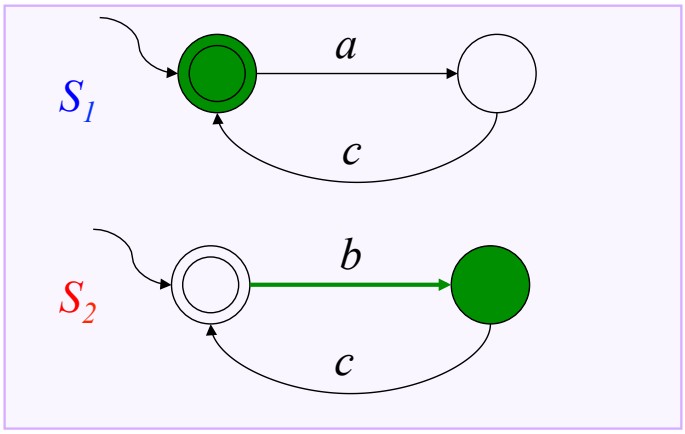
$S_0 = orch(S_1 \| S_2)$



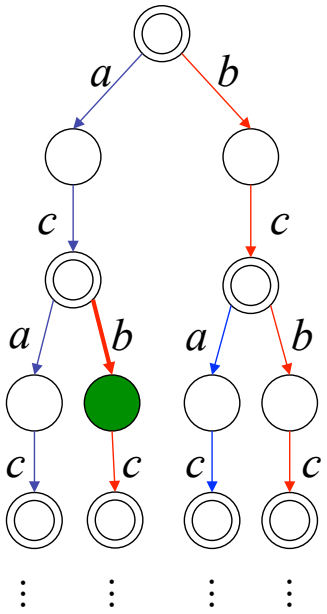
When the target service can be left, then all component services must be in a final state

Example of composition (7)

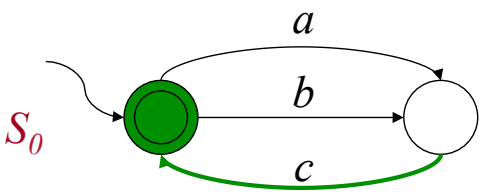


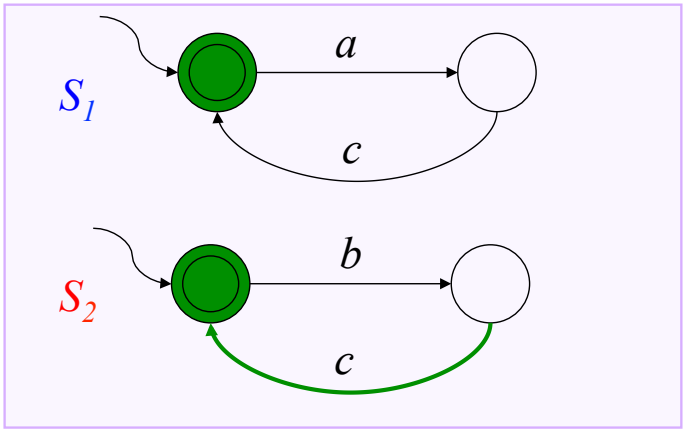


$S_0 = orch(S_1 \| S_2)$

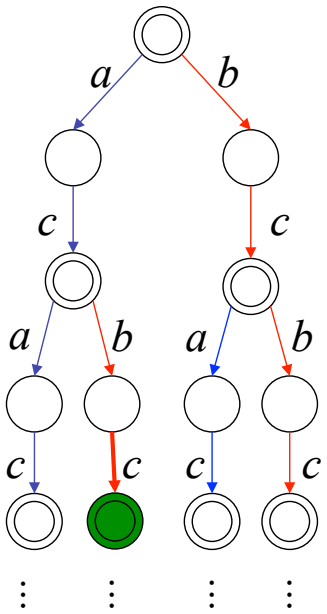


Example of composition (8)



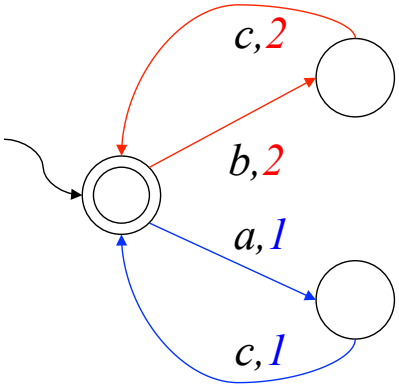


$S_0 = orch(S_1 \| S_2)$



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