Service Composition and Synthesis
The Roman Model

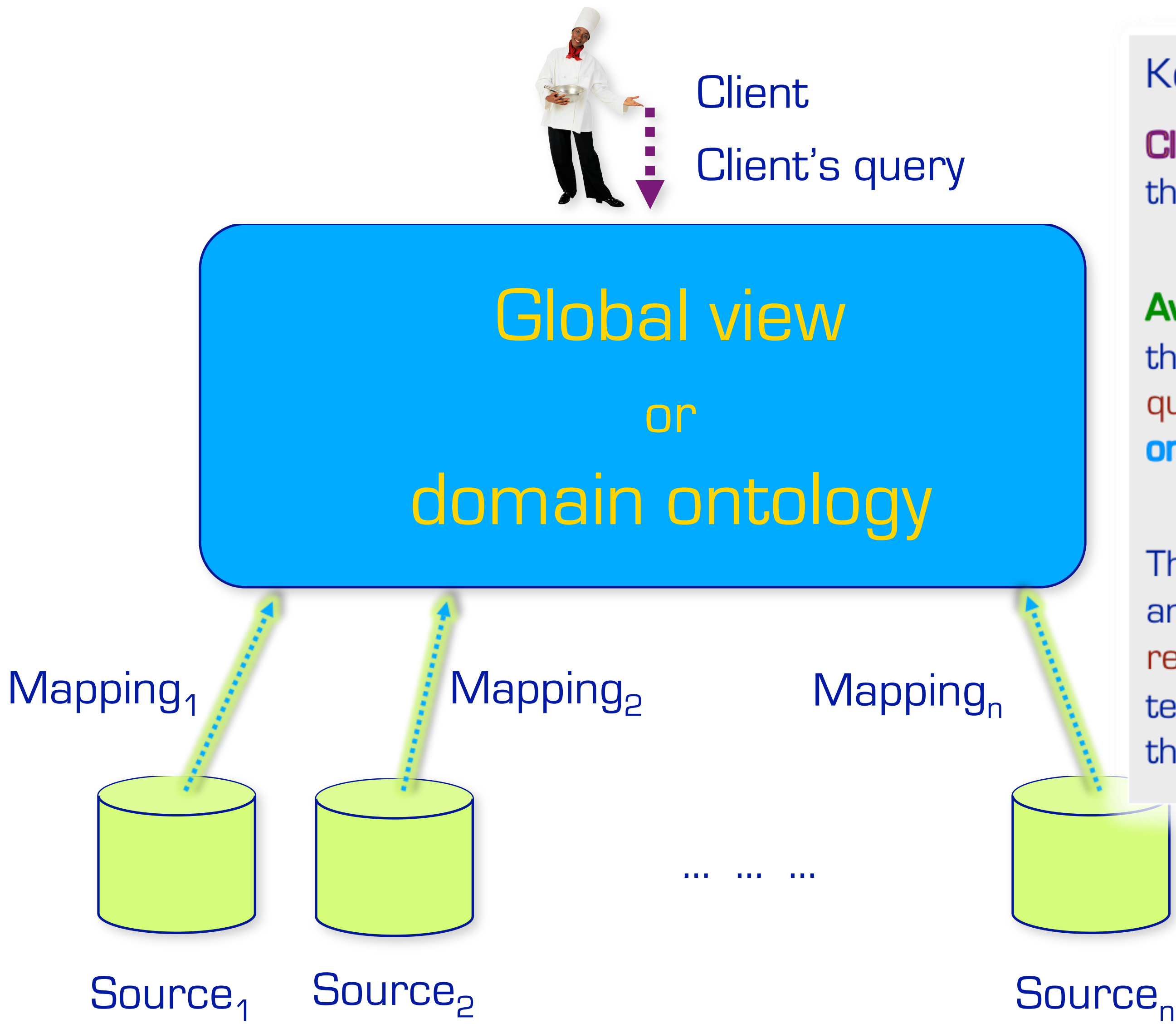
Giuseppe De Giacomo
SAPIENZA Università di Roma, Italy

Joint work with Daniela Berardi, Massimiliano de Leoni, Diego Calvanese, Fahima Cheikh, Rick Hull, Maurizio Lenzerini, Massimo Mecella, Fabio Patrizi, Antonella Poggi, Riccardo Rosati, Sebastian Sardina

Introduction

- The promise of **Service Computing** is to use services fundamental elements for realizing distributed applications/solutions.
- **Services** are processes that export their **abstract specification**
- When no available service satisfies a desired specification, one might check whether (parts of) available services can be **composed** and **orchestrated** in order to realize the specification.
- **Working at an abstract level** enable us to exploit results from **automatic verification and synthesis** to verify and compose services.
- The problem of automatic composition becomes especially interesting in the presence of **stateful** (conversational) services.
- Among the various frameworks proposed in the literature, here we concentrate on the so called ``**Roman Model**'' (name by Rick Hull).

Data Integration



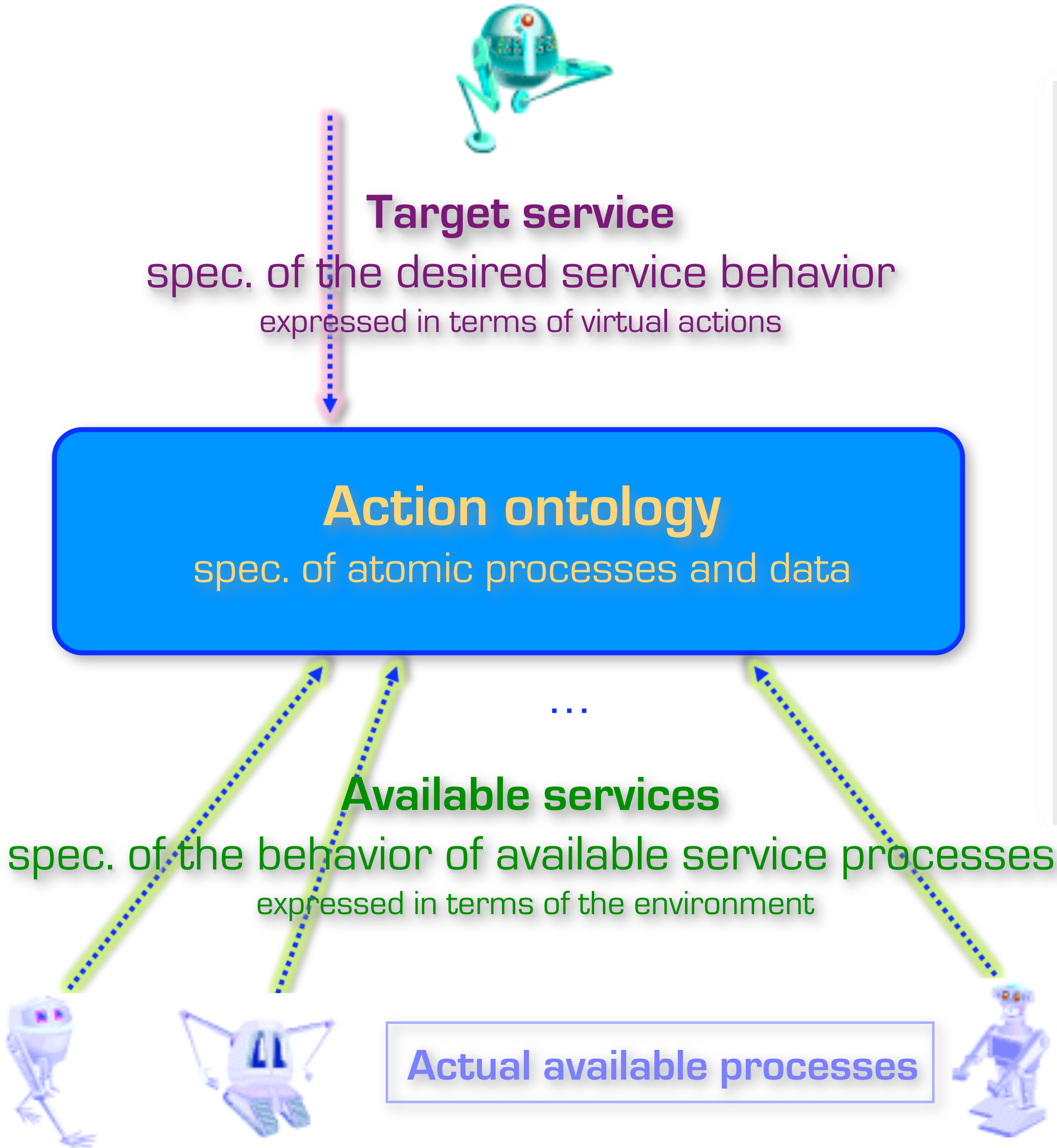
Key points

Client's request: query over the **domain ontology**

Available sources express their information in terms of a query over the **domain ontology**

The **data integration system** answers the **client's query** by reformulating/rewriting it in terms of the information in the **available sources**

Service integration/composition:



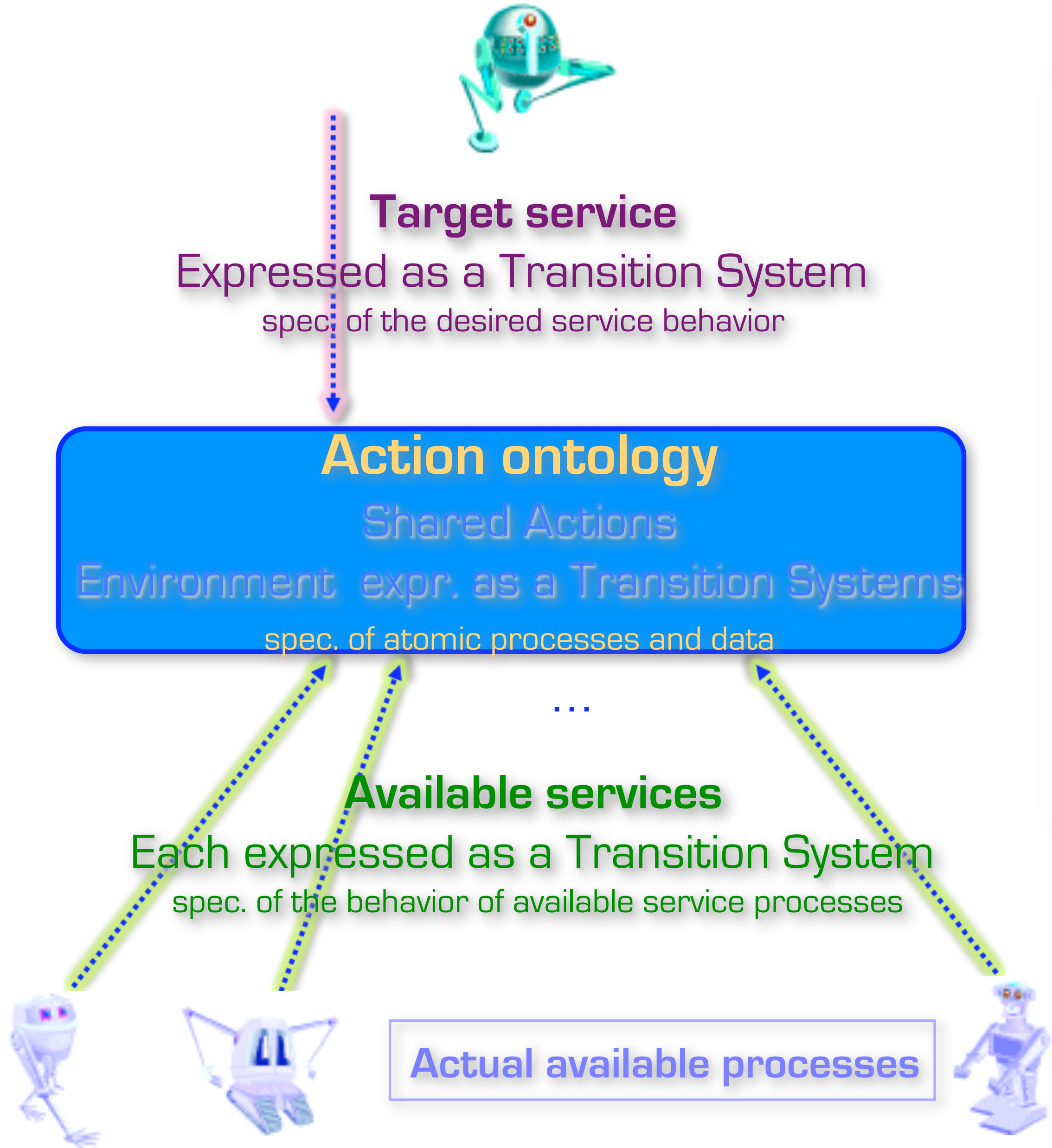
Key points

No available process for the target service

Must realize **target service** by **delegating** actual actions to **available services**

Available services are **stateful**, hence must **realize** the **target** using **fragments** of their **computations**

The Roman Model: basics



Key points

No available process for the target service

Must realize **target service** by **delegating** actual actions to **available services**

Available services are **stateful**, hence must **realize** the **target** using **fragments** of their **computations**

Roman Model's main ingredients

- The Roman Model exemplifies what can be achieved by composing conversational services and uncovers relationships with automated synthesis of reactive processes in Verification and AI Planning.

- Roman Model's main ingredients
  - **Each available service** is formally specified as a **transition system** that captures its possible conversations with a generic client.
  - Desired specification is a **target service**, described itself as a **transition system**.
  - the aim is to **automatically synthesize orchestrators** that realize the target service by delegating its actions to the available services, exploiting fragments of their execution.

Transition systems

- We represent services as **transition systems**:

- A TS is a tuple $< A, S, s_0, \delta>$ where:
  - A is the set shared of actions
  - S is the set of states
  - $s_0 \in S$ is the set of initial states
  - $\delta \subseteq S \times A \times S$ is the transition relation

Service composition

Problem of composition existence

- Given:
  - available services $B_1,\ldots,B_n$
  - target service T

  over the same environment (same set of atomic actions)
- Check whether T can be realized by **delegating** actions to $B_1,\ldots,B_n$ so as to **mimic** T over time (forever!)

Composition synthesis

synthesis of the **orchestrator** that does the delegation

Service composition as a game

There are at least two kinds of games. One could be called finite, the other infinite.

A finite game is played for the purpose of winning …

… an infinite game for the purpose of continuing the play.

Finite and Infinite Games
J. P. Carse, philosopher

Service composition as a game:
Service composition vs Planning

Roman model

Stateless service composition

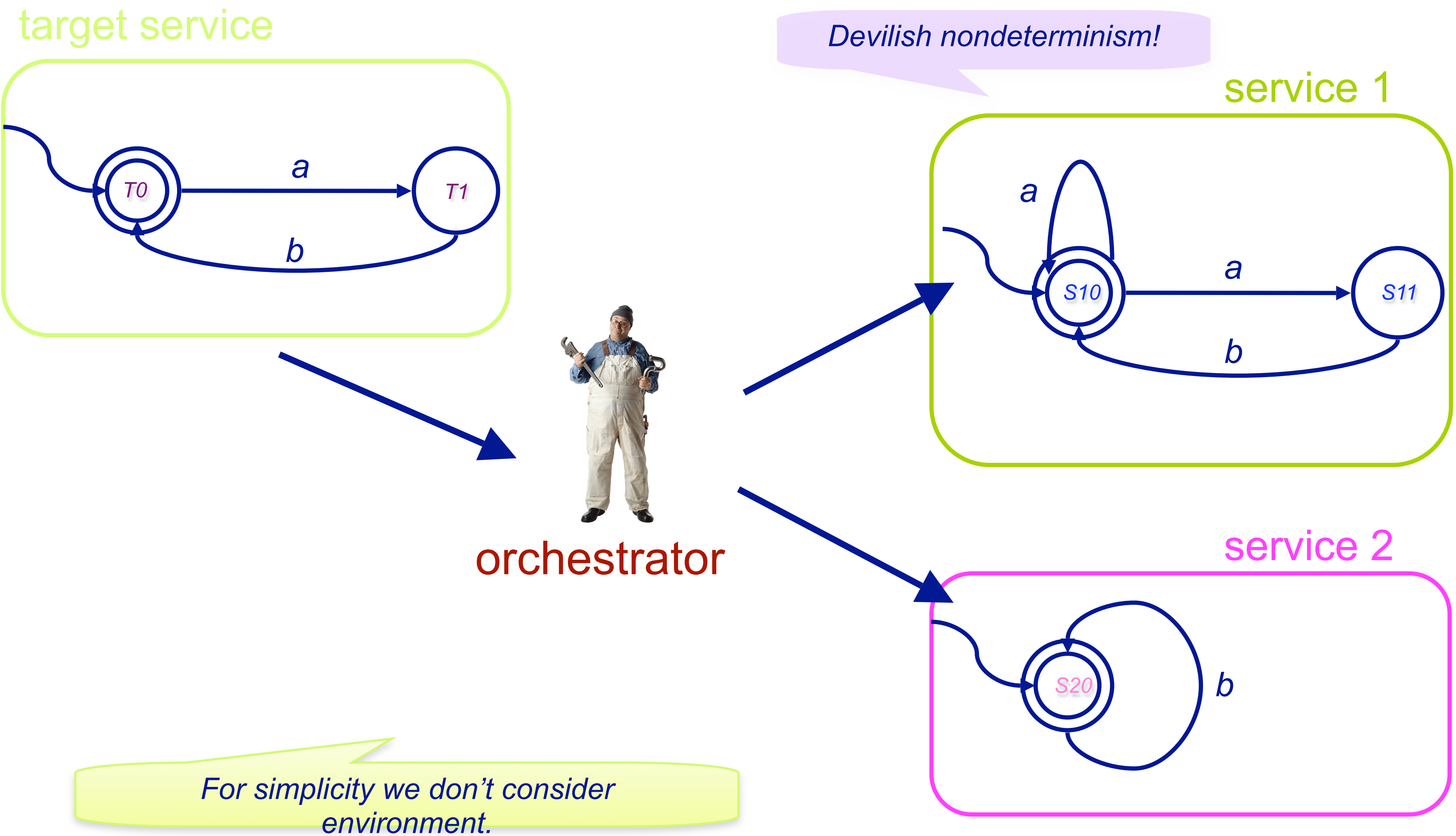
Planning

- **Operators:** atomic actions
- **Goal:** desired state of affair
- **Game: finite!**
  - compose operators sequentially so as to reach the goal
- **Playing strategy:** plan
  (program having operators invocation as atomic instructions)

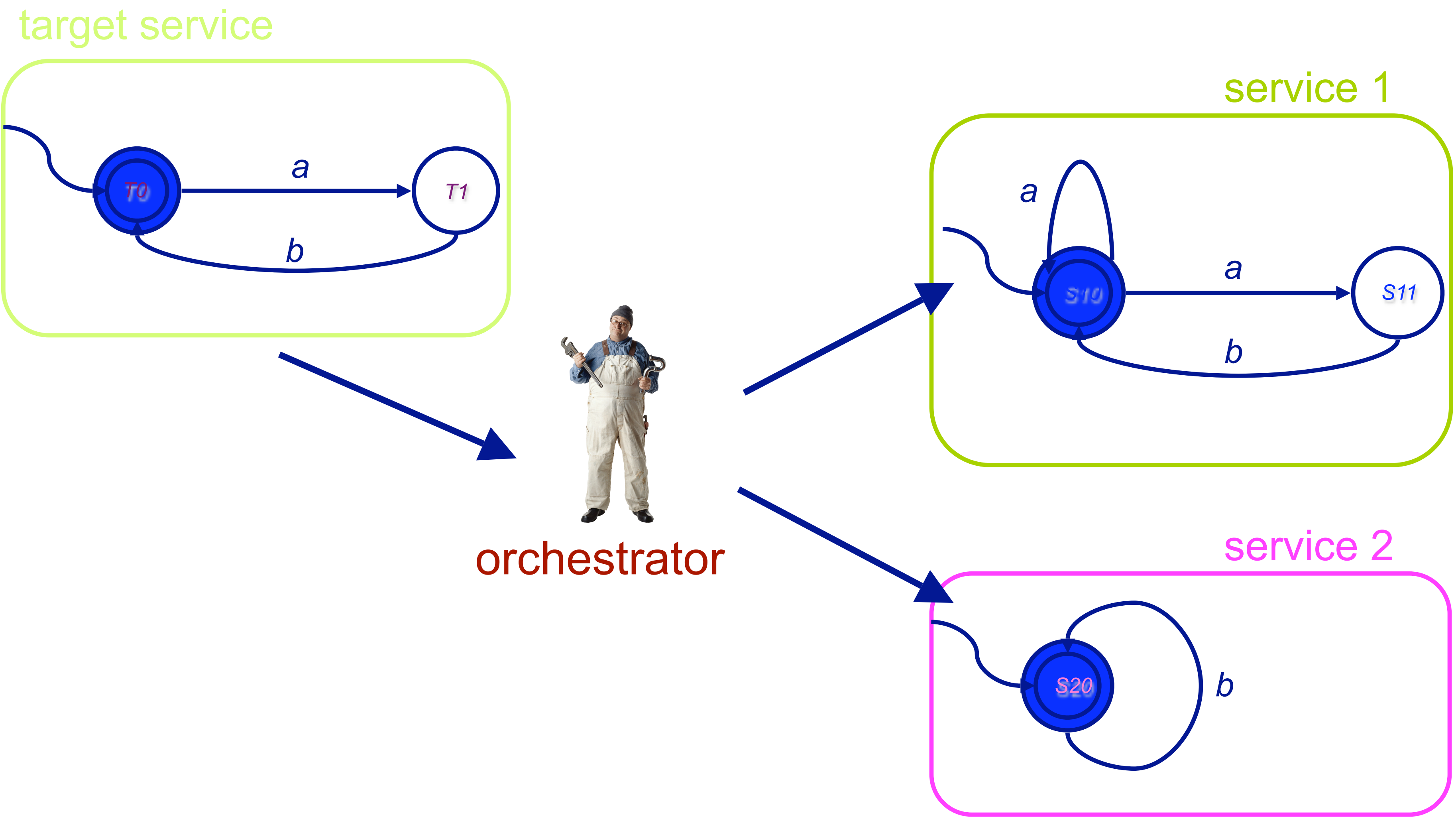
Service composition

- **Operators:** available transition systems
- **Goal:** target transition system
- **Game: infinite!**
  - compose available transition systems concurrently so as to play the target transition system
- **Playing strategy:** orchestrator
  (process that delegate target actions to the available service

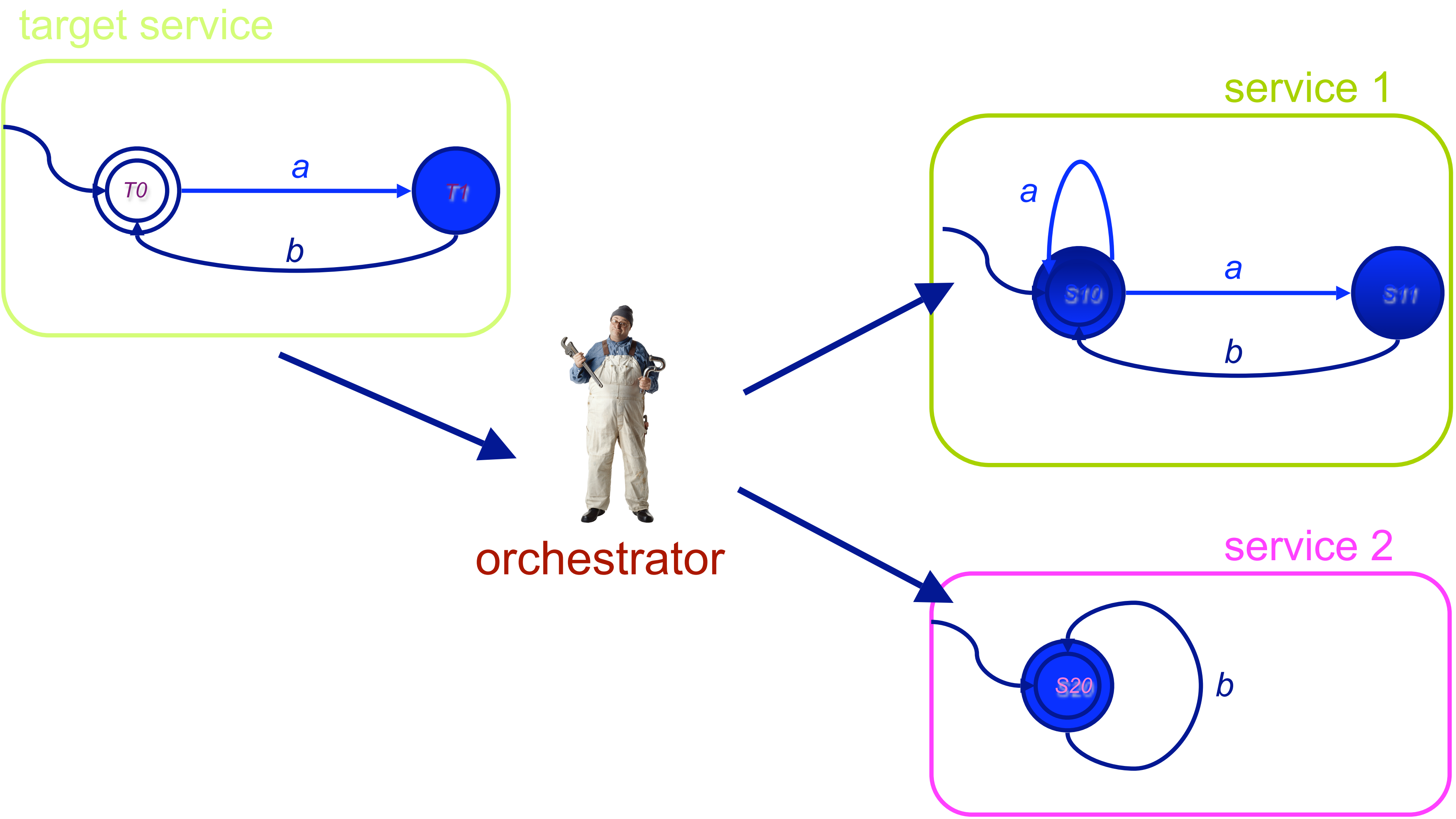
Simple example of service composition



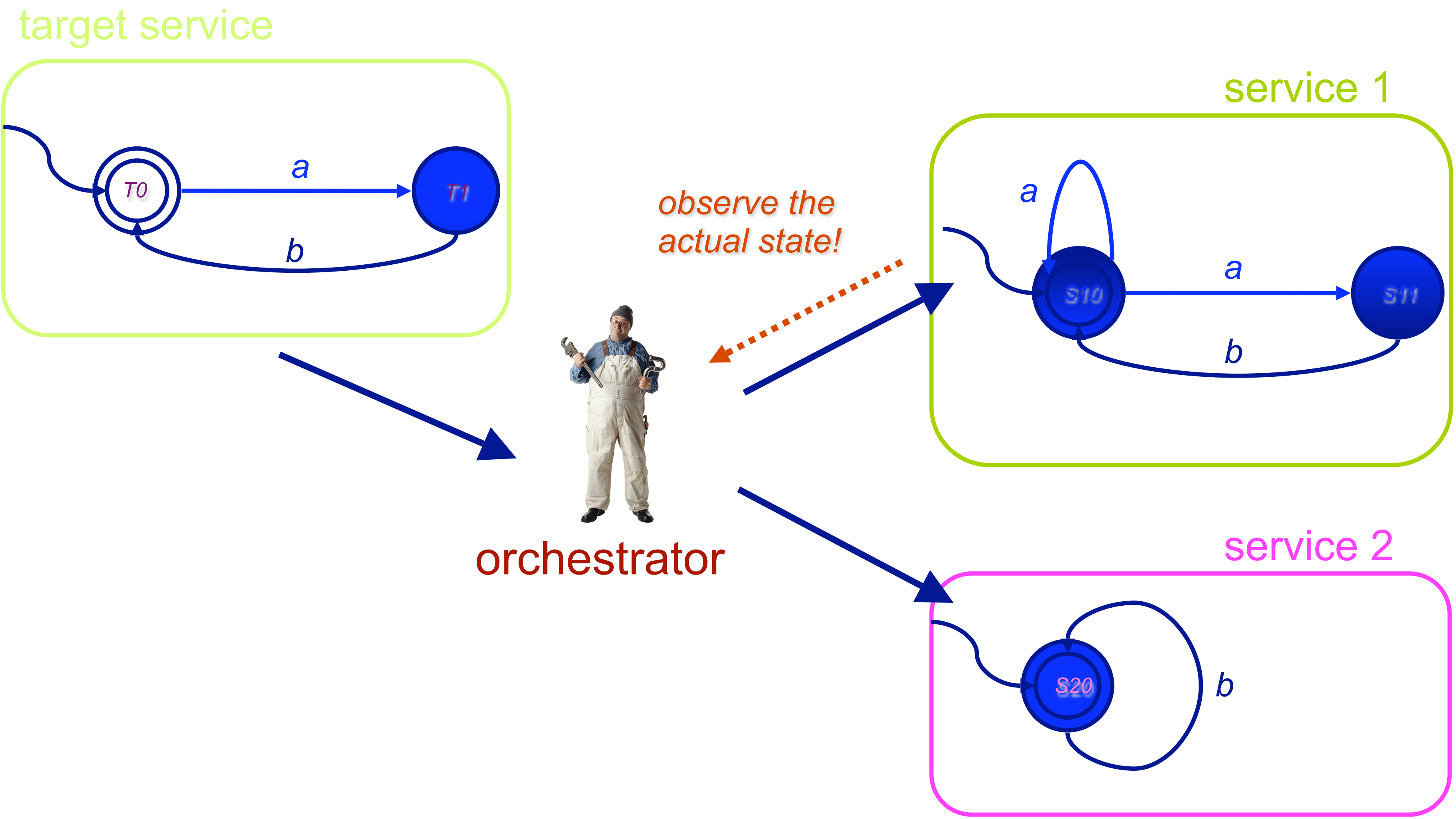
Simple example of service composition



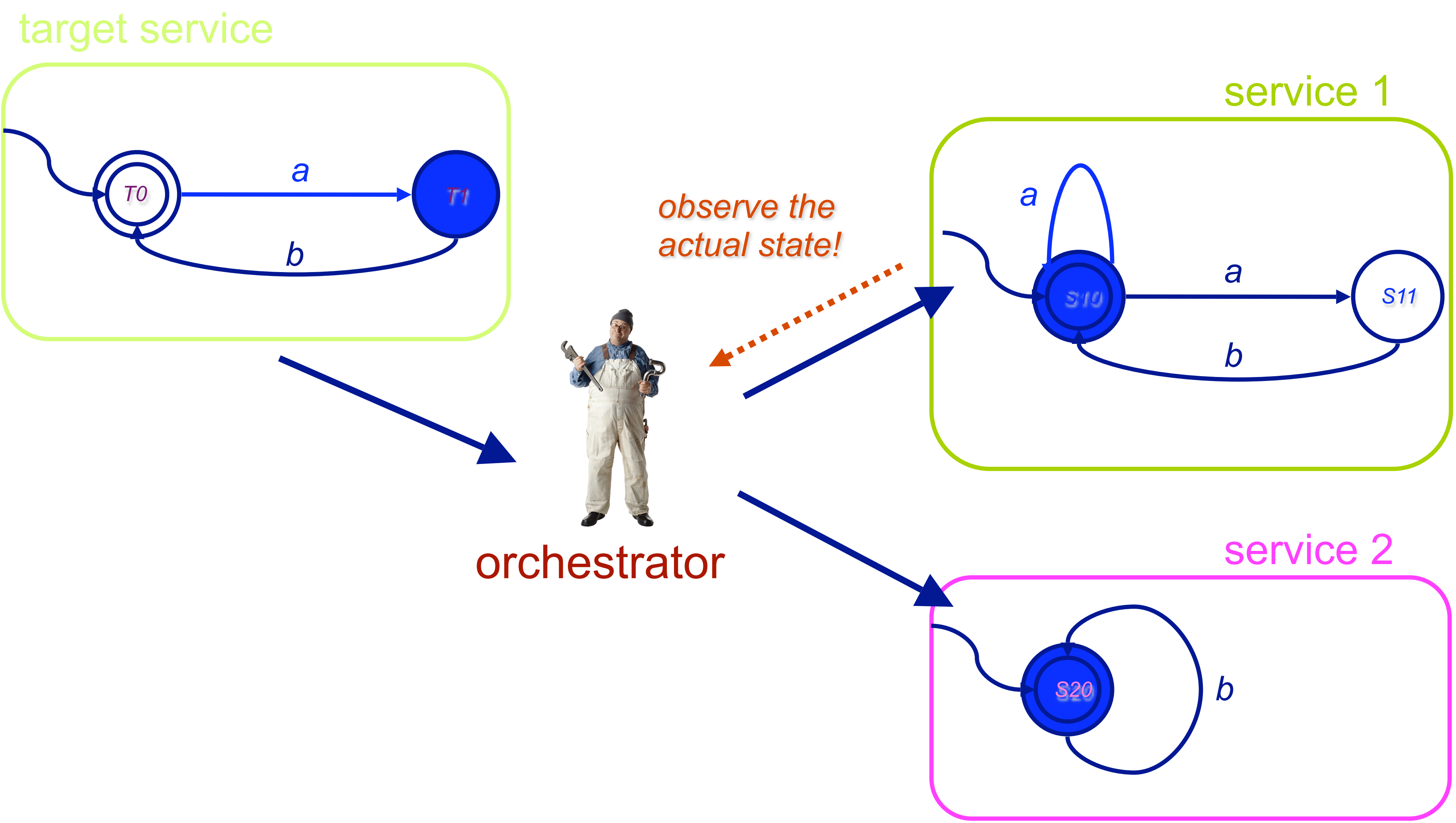
Simple example of service composition



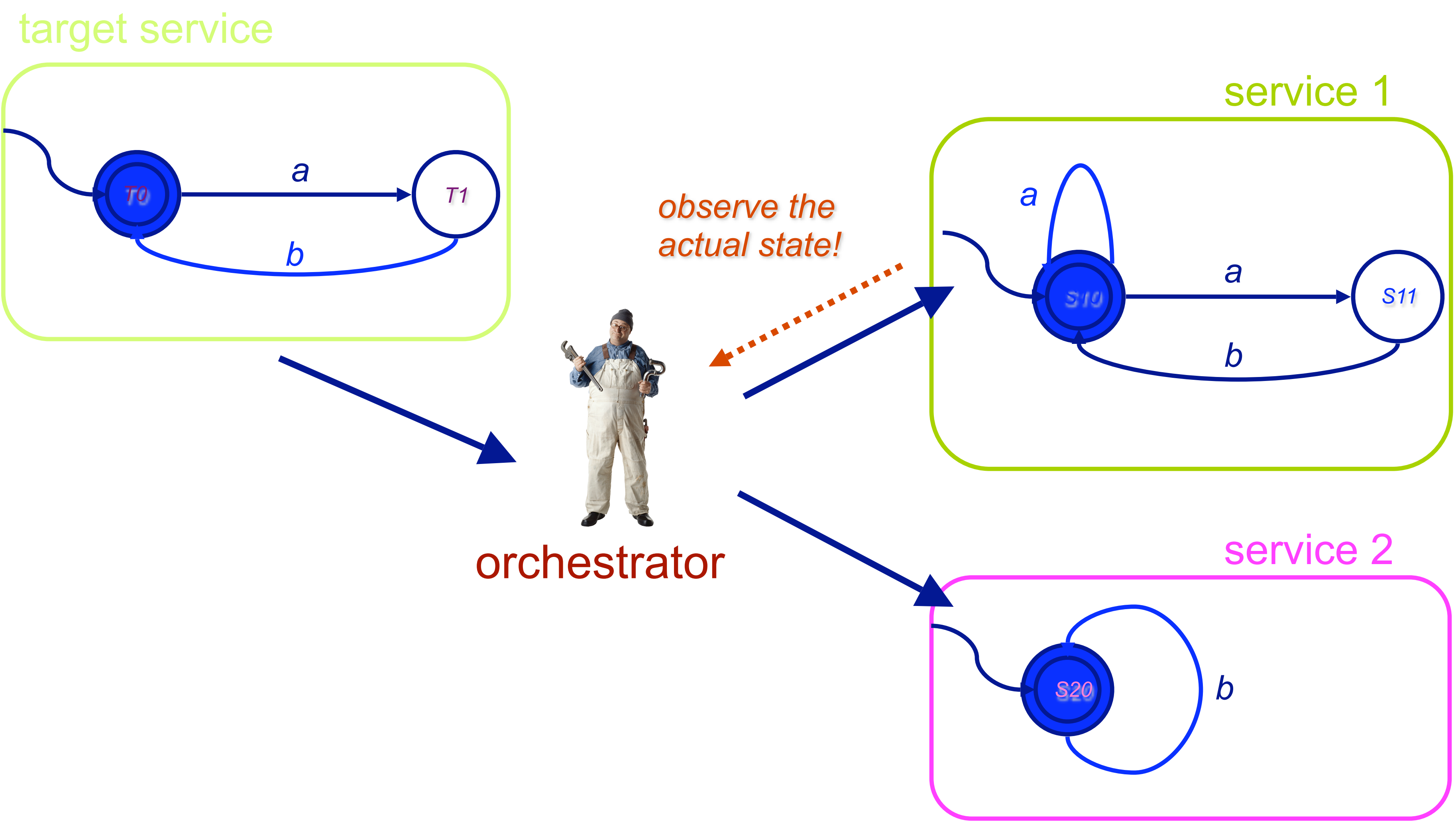
Simple example of service composition



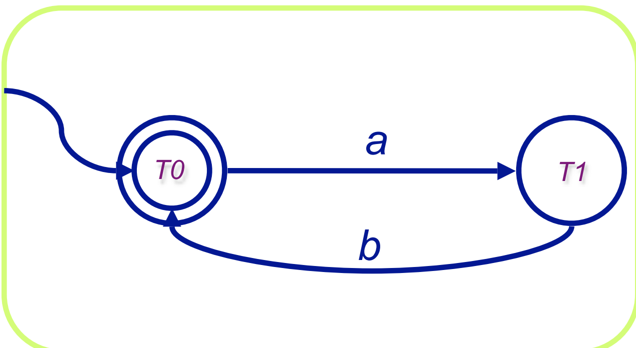
Simple example of service composition

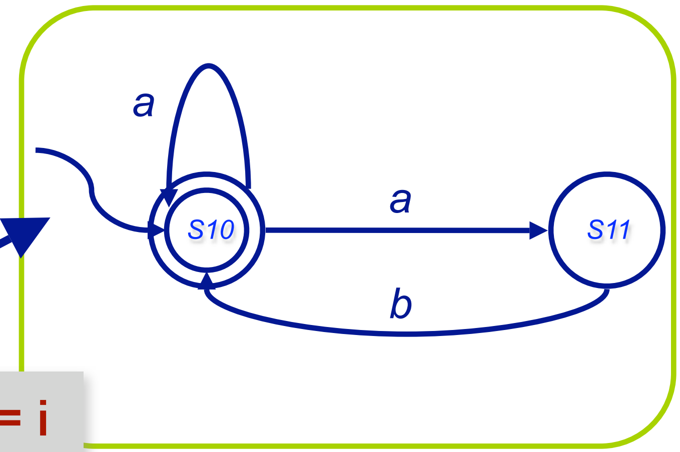


Simple example of service composition



Simple example of service composition



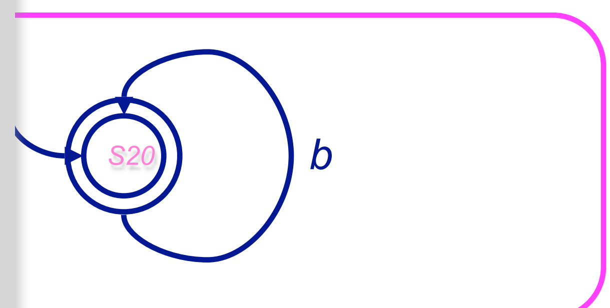


- **Orchestrator program** is any function **P(h,a) = i** that takes a **history h** and an **action a** to execute and **delegates a** to one of the available services **i**

- A **history** is a sequence that alternates states of the available services with actions performed:

  $(s_1^0,s_2^0,\dots,s_n^0)\ a_1\ (s_1^1,s_2^1,\dots,s_n^1)\ \dots\ a_k\ (s_k^1,s_2^k,\dots,s_n^k)$

- Observe that to take a decision **P** has **full access to the past**, but no access to the future



Synthesizing compositions

- Techniques for computing compositions:
- Reduction to PDL SAT
- Simulation-based
- LTL synthesis as model checking of game structure

(all techniques are for finite state services)

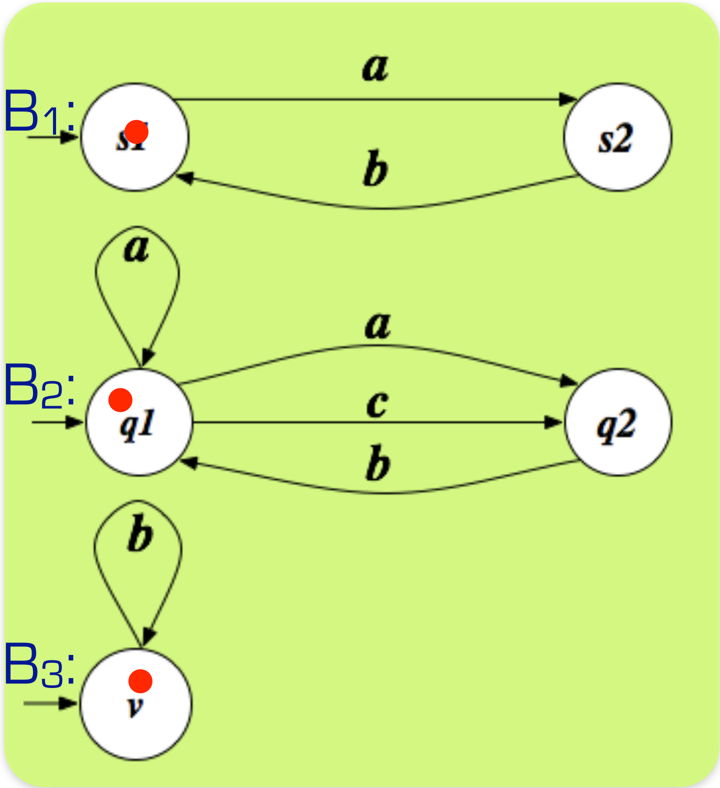
Simulation-based technique

Directly based on

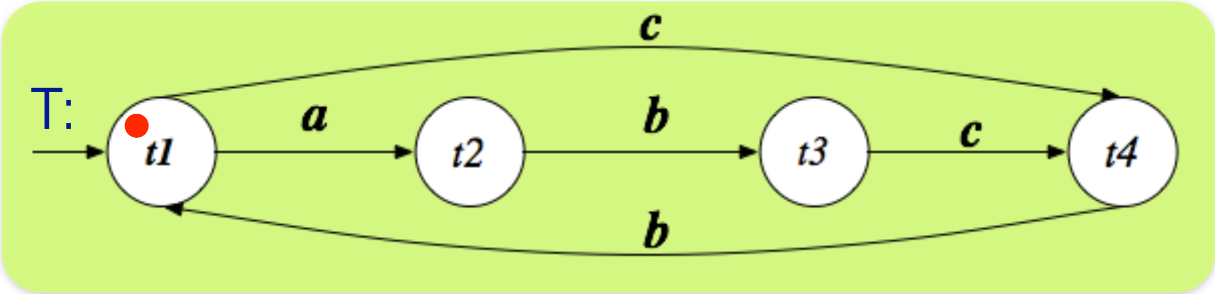
... controlling the concurrent execution of available services $B_1,\ldots,B_n$ so as to **mimic** the target service T

Thm: Composition exists iff the asynchronous (Cartesian) product C of $B_1,\ldots,B_n$ can **(ND-)simulate** T

Example of composition by simulation



Given from available and target service ...



Computing composition via simulation

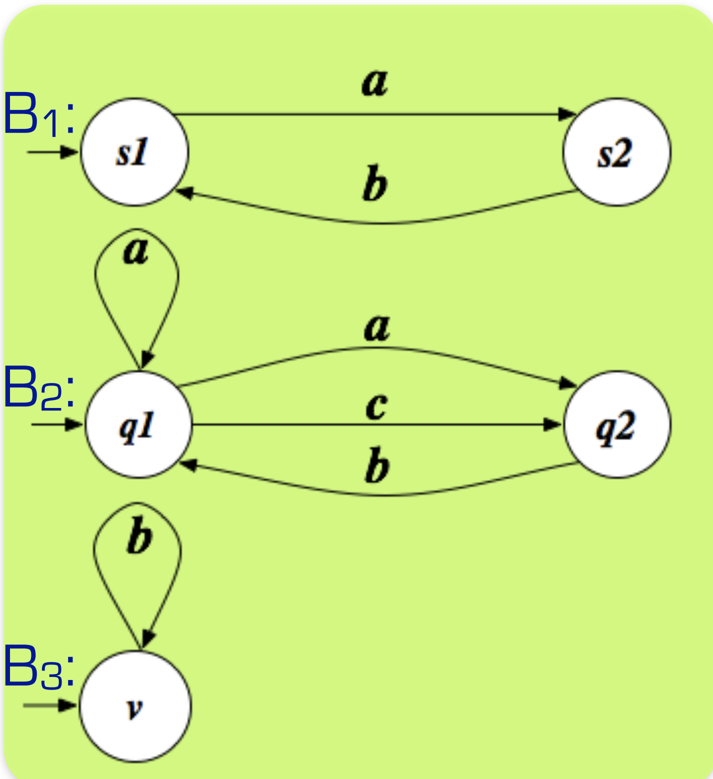
Let $B_1,\ldots,B_n$ be the TSs of the available behaviors.

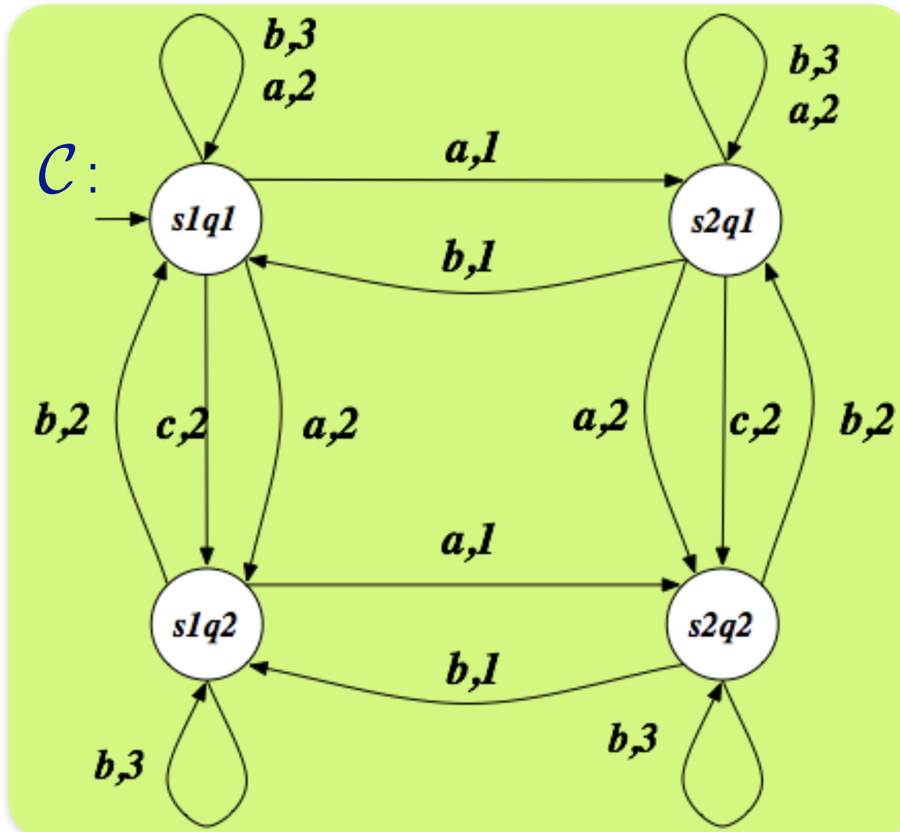
The **Available behaviors TS** $\mathcal{C} = < A, S_\mathcal{C}, s_\mathcal{C}^0, \delta_\mathcal{C}, F_\mathcal{C}>$ is the **asynchronous product** of $B_1,\ldots,B_n$ where:

- A is the set of actions
- $S_\mathcal{C} = S_1 \times\ldots\times S_n$
- $s_\mathcal{C}^0 = (s^0{}_1,\ldots, s^0{}_m)$
- $\delta_\mathcal{C} \subseteq S_\mathcal{C} \times A \times S_\mathcal{C}$ is defined as follows:
- $(s_1 \times\ldots\times s_n) \to_a (s'_1 \times\ldots\times s'_n)$ iff

$$\exists\, i.\ s_i \to_a s'_i \in \delta_i \text{ and } \forall\, j\neq i.\ s'_j = s_j$$

Example of composition by simulation





... consider the **asynchronous product** of the available services ...

22

Simulation relation

Given a target service T and (the asynchronous product of) available services $\mathcal{C}$, a (**ND-**)**simulation** is a relation R between the states $t \in \mathcal{T}$ an $(s_1,..,s_n)$ of $\mathcal{C}$ such that:

$(t, s_1,..,s_n) \in R$ implies that

for all $t \rightarrow_a t'$ in T, exists a $B_i \in \mathcal{C}$ s.t.

- $\exists\ s_i \rightarrow_a s'_i$ in $B_i \wedge$
- $\forall\ s_i \rightarrow_a s'_i$ in $B_i \Rightarrow (t', s_1,..s'_i,..s_n) \in R$

- If **exists a simulation** relation R (such that $(t^0, s_1^0,..,s_n^0) \in R$, then we say that or **T is simulated by** $\mathcal{C}$ (or $\mathcal{C}$ **simulates T**).

- **Simulated-by** is

–(i) a simulation;

–(ii) the largest simulation.

Simulated-by is a coinductive definition

Simulation relation (cont.)

Algorithm Compute (ND-)simulation

Input: target behavior T and (async. prod. of) available behaviors $\mathcal{C}$

Output: the **simulated-by** relation (the largest simulation)

Body

$R = \emptyset$

$R' = S_T \times S_1 \times .. \times S_n$

while $(R \neq R')$ {

$R := R'$

$R' := R' - \{(t, s_1,..,s_n) \mid \exists\, t \rightarrow_a t' \text{ in } T \wedge$

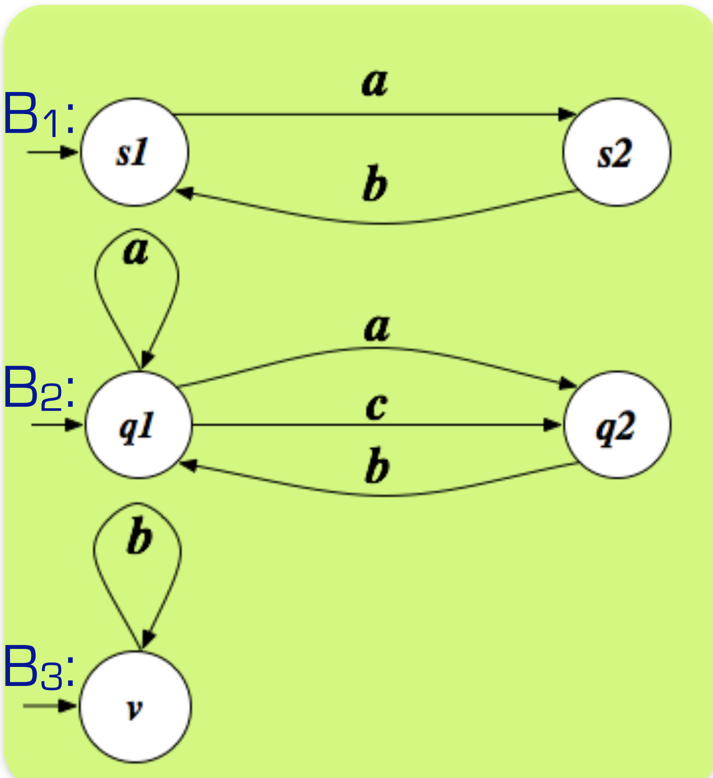
$\neg (\exists\, s_i \rightarrow_a s'_i \text{ in } B_i \wedge \forall\, s_i \rightarrow_a s'_i \text{ in } B_i \Rightarrow (t', s_1,..s'_i,..s_n) \in R')\}$

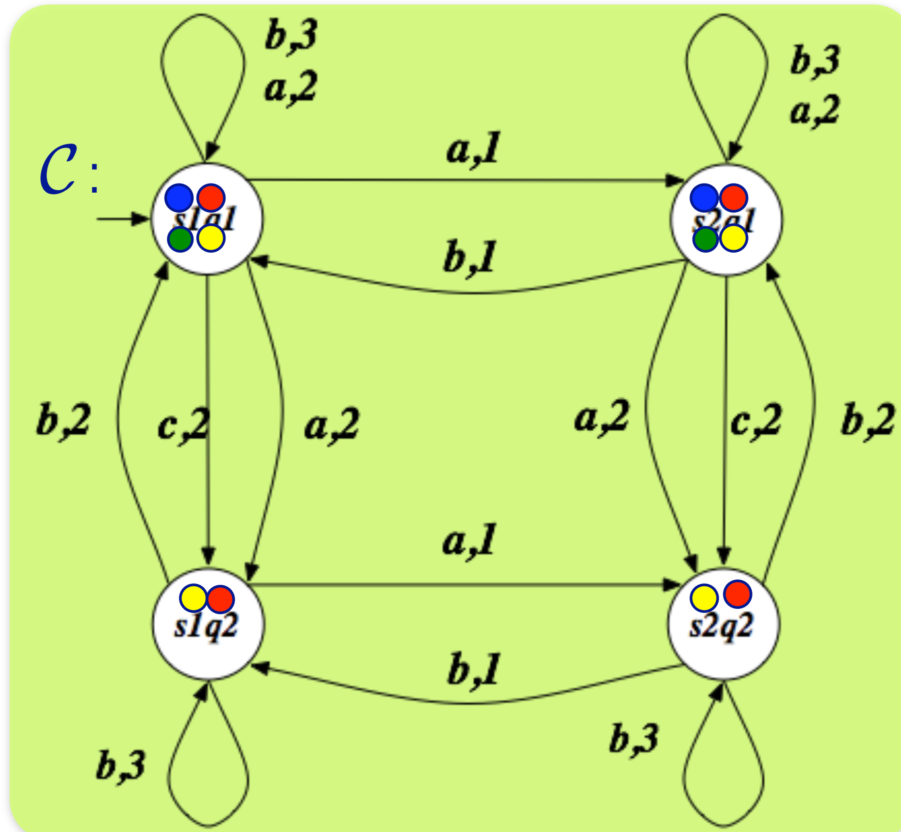
}

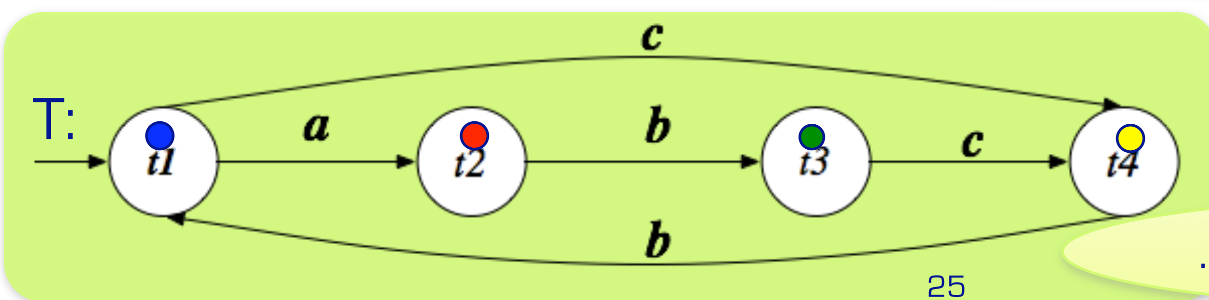
return R'

End

Example of composition by simulation







... compute **ND-simulation**

Using simulation for composition

- Given the largest simulation R of T by C, we can build every composition through the ***orchestrator generator (OG)***.

- **OG** = $< A, [1,\dots,n], S_r, s_r^0, \delta_r, \omega_r, >$ with
- A : the **actions** shared by the behaviors
- $[1,\dots,n]$: the **identifiers** of the available services in the community
- $S_r = S_T \times S_1 \times \dots \times S_n$: the **states** of the orchestrator generator
- $s_r^0 = (t^0, s^0_1, \dots, s^0_n)$: the **initial state** of the orchestrator generator
- $\omega: S_r \times A_r \to 2^{[1,\dots,n]}$: the **output function**, defined as follows:

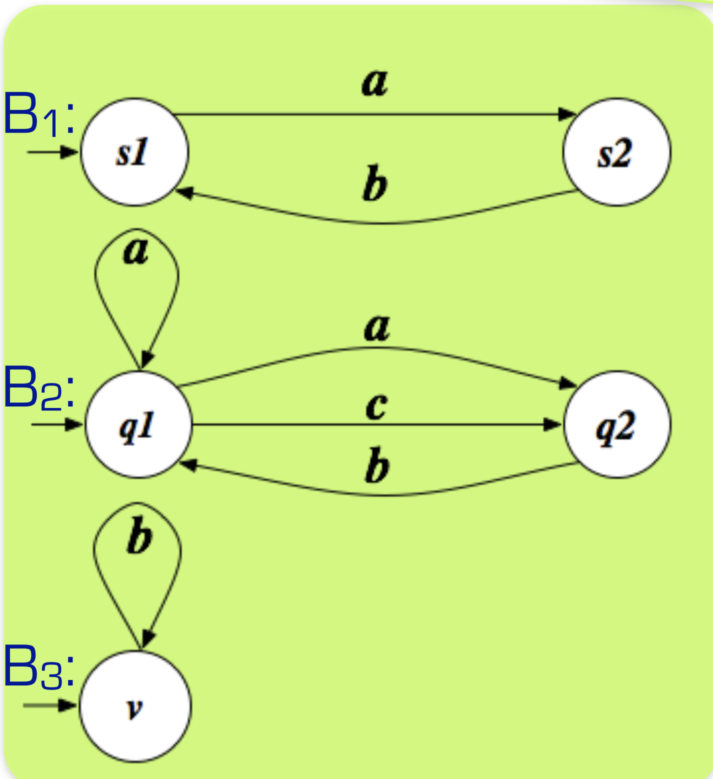
$$\boldsymbol{\omega(t, s_1,..,s_n, a) =}$$
$$\boldsymbol{\{ i \mid \exists\, t \to_{a,} t' \text{ in } T \wedge \exists\, s_i \to_{a,} s_i' \text{ in } B_i \wedge (t', s_1,..,s'_i,..,s_n) \in R\}}$$

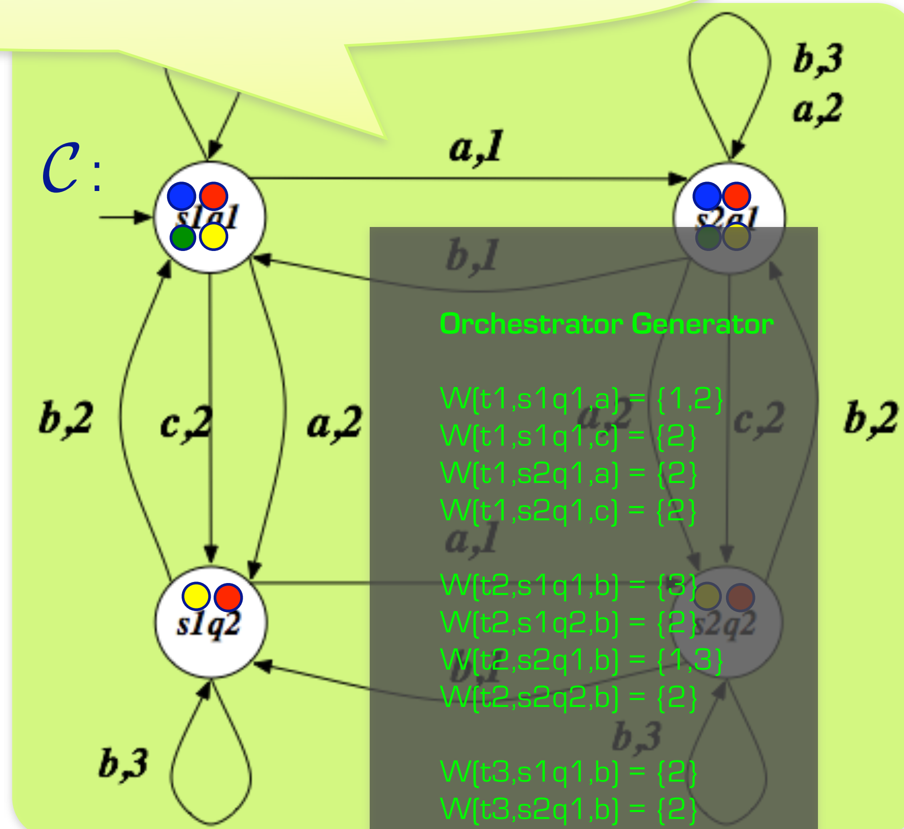
- $\delta \subseteq S_r \times A \times [1,\dots,n] \to S_r$: the **state transition function**, defined as follows

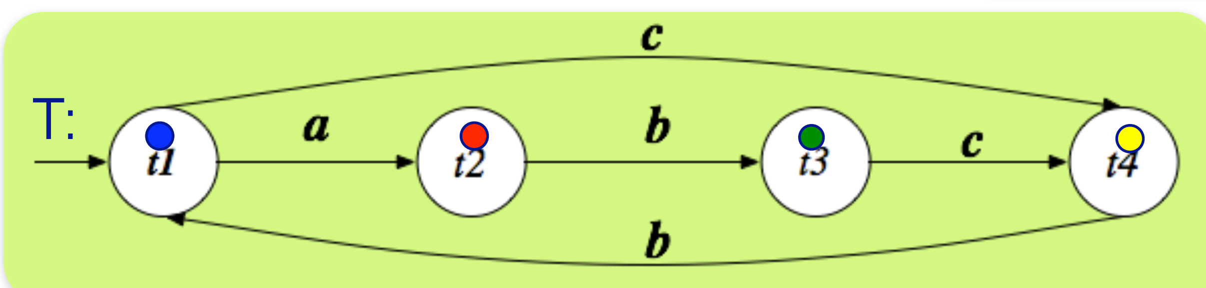
$(t, s_1, \dots, s_i, \dots, s_n) \to_{a,i} (t', s_1, \dots, s'_i, \dots, s_n)$ iff $i \in \omega(t, s_1, .., s_i, .., s_n, a)$

Example of composition by simulation

… compute the **orchestrator generator**







Orchestrator Generator

W(t1,s1q1,a) = {1,2}
W(t1,s1q1,c) = {2}
W(t1,s2q1,a) = {2}
W(t1,s2q1,c) = {2}

W(t2,s1q1,b) = {3}
W(t2,s1q2,b) = {2}
W(t2,s2q1,b) = {1,3}
W(t2,s2q2,b) = {2}

W(t3,s1q1,b) = {2}
W(t3,s2q1,b) = {2}

W(t4,s1q1,b) = {3}
W(t4,s1q2,b) = {2}
W(t4,s2q1,b) = {1,3}
W(t4,s2q2,b) = {2}

Results

- ***Thm:*** choosing at each point any value in returned by the orchestrator generator gives us a composition.

- ***Thm:*** every composition can be obtained by choosing, at each point a suitable value among those returned by the orchestrator generator.

Note: there **infinitely many compositions** but
only **one orchestrator generator** that captures them all

- ***Thm:*** computing the orchestrator generator is EXPTIME, and in fact exponential only in the number (and not the size) of the available behaviors.

Composition in the Roman Model was shown to be EXPTIME-hard
[Muscholl&Walukiewicz07]

Just-in-time composition

- Once we have the orchestrator generator …
- … we can **avoid choosing any particular composition** a priori …
- … and **use directly $\boldsymbol{\omega}$** to choose the available behavior to which delegate the next action.

- We can be ***lazy*** and make such choice ***just-in-time***, possibly adapting reactively to ***runtime*** feedback.

Just-in-time compositions can be used to reactively act upon failures [KR08]!

Tools for computing composition based on simulation

- Computing simulation is a well-studied problem (related to computing **bisimulation** a key notion in process algebra). Tools, like the Edinburgh Concurrency Workbench and its clones, can be adapted to compute composition via simulation.

- Also **LTL-based synthesis** tools, like TLV, can be used for (indirectly) computing composition via simulation [Patrizi PhD09]

We are currently focusing on the second approach.

Adding data to the Roman Model

Adding data is crucial in certain contexts:

- Data - rich description of the **static information** of interest.
- Behaviors - rich description of the **dynamics** of the process

But makes the approach extremely challenging:

- We get to work with infinite transition systems
- Simulation can still be used for capturing composition
- But it cannot be computed explicitly anymore.

We present two orthogonal approaches to deal with them.

The Roman Model: American tweak

with Rick Hull + Jianwen Su

Target service

Expressed as a Guarded TS with parameters

spec. of the desired service behavior

Action ontology

Data-aware Environment or DB/Artifact + atomic action that affect stored data

spec. of atomic processes and data

...

Available services

Each expressed as a Guarded TS with parameters

spec. of the behavior of available service processes

Actual available processes

Key points

No available process for the target service

Must realize **target service** by **delegating** actual actions to **available services**

Available services are **stateful**, hence must **realize** the **target** using **fragments** of their **computations**

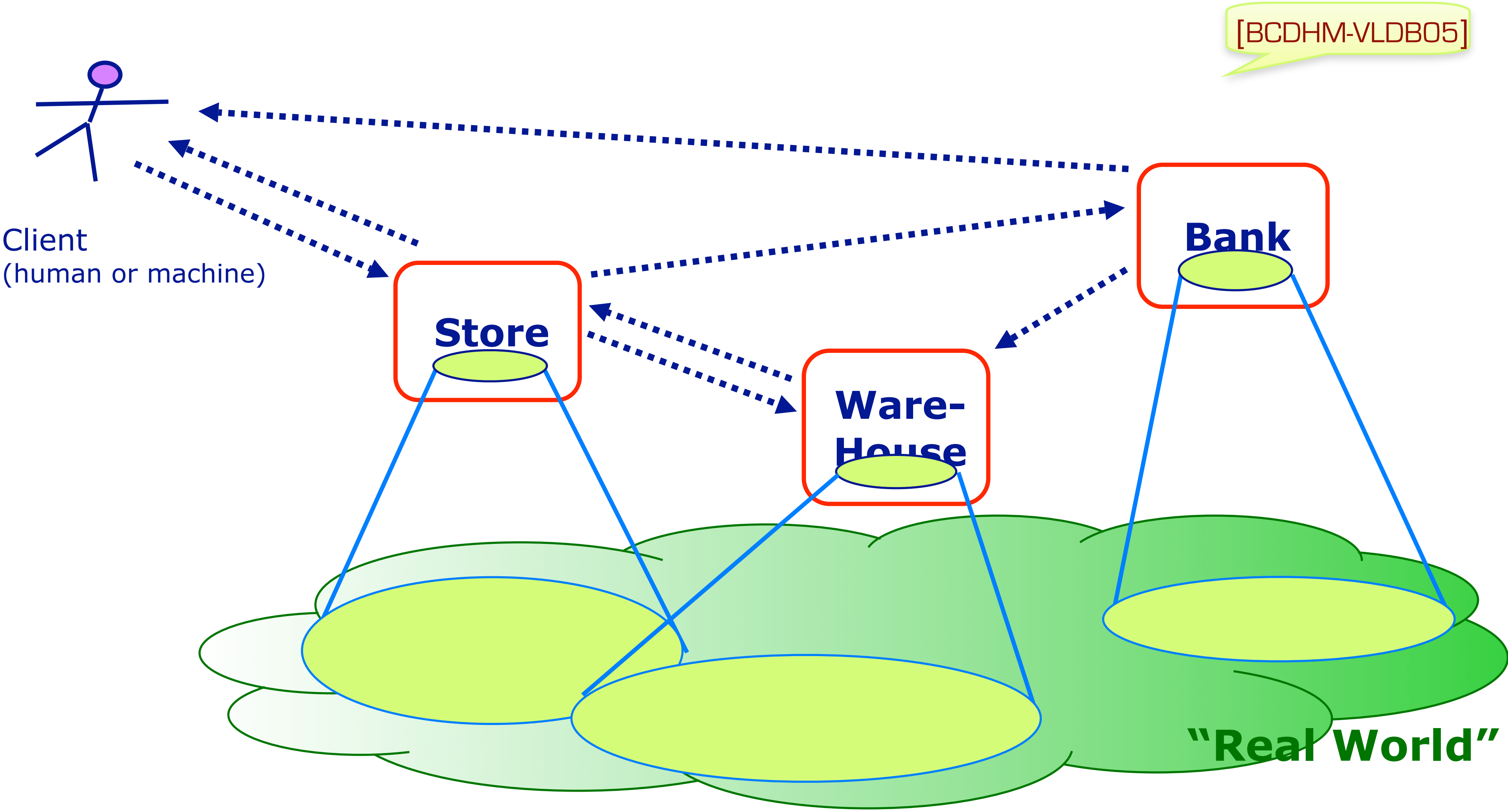
Data-Aware Service Composition

Data-Aware Service Composition

Fabio Patrizi & Giuseppe De Giacomo

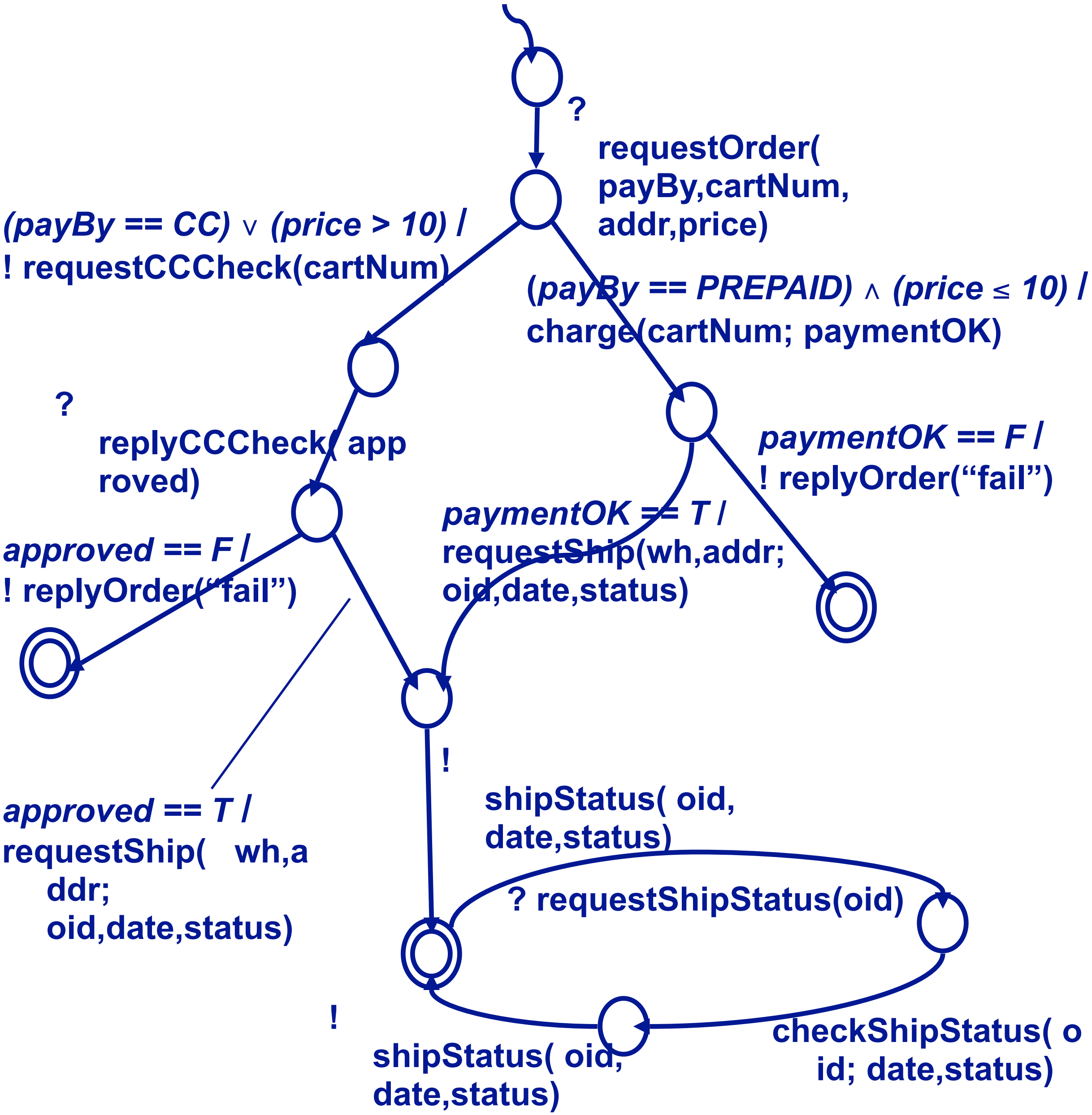
Dipartimento di Informatica e Sistemistica "Antonio Ruberti"
SAPIENZA Università di Roma, Rome, ITALY

Services act on an integrated view of the world …



- Actions may impact "real world" – modeled as FOL relations
- Also actions may be messages between services

Service behavior of as abstract finite state machines that query and act on the infinite state world …



- Local store
- Edge conditions based on local store (and incoming message)
- Edge actions
  - Atomic Process
    - acting on the world
    - set the local store
  - Create/send message
  - Read message

[BCDHM-VLDB05]

The Roman Model: Australian/Canadian tweak

Key points

No available process for the target service

Must realize **target service** by **delegating** actual actions to **available services**

Available services are **stateful**, hence must **realize** the **target** using **fragments** of their **computations**

Composition of ConGolog Programs

Composition of ConGolog Programs

Sebastian Sardina[1] Giuseppe De Giacomo[2]

[1]Department of Computer Science and Information Technology
RMIT University, Melbourne, AUSTRALIA

[2]Dipartimento di Informatica e Sistemistica "Antonio Ruberti"
Sapienza Universita' di Roma, Rome, ITALY

Mixing data and service integration: A real challenge for the whole CS

We have all the issues of data integration but in addition …

- Behavior: description of the **dynamics** of the process!

- Behavior should be formally and **abstractly** described: conceptual modeling of dynamics (not a la OWL-S). Which?
  - Workflows community may help
  - Business process community may help
  - Services community may help
  - Process algebras community may help
  - AI & Reasoning about actions community may help
  - DB community may help
  - … may help

Artifact-centric approach promising!

- Techniques for **analysis/synthesis** of **services** in presence of **unbounded data** can come from different communities:
  - Verification (CAV) community: abstraction to finite states
  - AI (KR) community: working directly in FOL/SOL, e.g., SitCalc

The Roman Model: Italian dream

Very preliminary ideas in DL07

Target service

Expressed in **conceptual process** description language

spec. of the desired service behavior

Action ontology

Expressed as an ontology over the data + related conceptual atomic actions

spec. of atomic processes and data

. . .

Available services

Each expressed **conceptual process** description language

spec. of the behavior of available service processes

Actual available processes

Key points

No available process for the target service

Must realize **target service** by **delegating** actual actions to **available services**

Available services are **stateful**, hence must **realize** the **target** using **fragments** of their **computations**