ELECTIVE IN A.I - 2 - REASONING AGENTS

Service Composition in Stochastic Settings

Lomuscio Antonio 1647493
Esposito Pierfrancesco 1642232
Rrapi Franci 1660405

SAPIENZA
UNIVERSITÀ DI ROMA

Paper: Service Composition in Stochastic Settings

OUTLINE

- Introduction of The Paper
- The Non-Stochastic Model
- The Valued Requirement Model
- Computing an Optimal Orchestrator
- Extensions
- Examples
- Conclusion
- References

Introduction Of The Paper

The “Goal”

- The goal is to build a controller known as an orchestrator, that uses existing services to satisfy the requirements of the target service
- We will talk about a simple stochastic model for service composition
- We will see the target service with the probabilities describing the likelihood of requesting an action:
  - Each action in a state
  - Rewards for being able to execute actions.
- We will show how to solve the resulting problem by solving a certain Markov Decision Process (MDP)

Introduction Of The Paper

Where we can introduce the Orchestrator?

- Thanks to growth of "IoT" and "Online Web Services", we can combine multiple services and see of they work
- For example:
  - Complete "Vacation" by combining Web services
  - Flights
  - Ground Transportation
  - Event tickets
  - Home security services
  - Ect.

Introduction Of The Paper

Roman Model: ‘The Pristine Form’

- Each available service is modeled as a finite state machines (FSM):
  - In Each state => Set of Actions
  - Each action => **Changes** the State of Service
- A Scheduler (Orchestrator) use "Actions" provided by existing services to implement action **requested** by the user

Introduction Of The Paper

Probabilistic model

- We make the requirements probabilistic associating a probability with each action chosen in a state
- This probability captures how likely the user requests the action in that state
- Moreover a reward is associated with the requirement behavior
- We observe that rewards can be related to "QoS"
- We use complex reward specifications in the form of transducers, or formulas in expressive logics such as LTL_f and LDL_f

The Non-Stochastic Model
Roman Model

- We adopt the Roman model for *service composition*
- A *service* is defined as a tuple $S = (\Sigma, \sigma_0, F, A, \delta)$

Σ is the finite set of service's *states*;
$\sigma_0 \in \Sigma$ is the *initial state*;
$F \subseteq \Sigma$ is the set of service's *final* states;
A is the finite set of service's *actions*;
$\delta \subseteq \Sigma \times A \mapsto \Sigma$ is the service's *transition (partial) function*, i.e., *actions are deterministic*.

The Non-Stochastic Model

R.M: Execution of Requested Action

- A *history* h of a service S is a sequence alternating states and actions

$$\sigma^0 \cdot a_1 \cdot \sigma^1 \cdot a_2 \cdot \cdots \cdot a_n \cdot \sigma^n \cdot \cdots$$

- We assume we have a finite set of available services:
  - $S_i = (\Sigma_i, \sigma_{i0}, F_i, A, \delta_i)$ (over same actions A)
- Given S, "*target* service" $T = (\Sigma_t, \sigma_{t0}, F_t, A, \delta_t)$
- The target represents a business process that one would like to offer to clients, where each state represents a decision point

The Non-Stochastic Model

The goal of service composition

- The Goal is to combine the available services as to mimic from a client point of view the behavior of the target service
- This can be done by interposing:

Available Services <“O*rchestrator*" > the Client

- The orchestrator **delegates** the current action requested by the client to some available service
- The *system service* of S is the service:
  - $Z = (\Sigma_z, \sigma_{z_0}, F_z, A_z, \delta_z)$

The Non-Stochastic Model
System Service Z

- Z is the service stemming from the product of the **asynchronous** execution of the services in S
- An *orchestrator* for a community S is a partial function:
  - $\gamma:\Sigma_z \times A \rightarrow \{1,\ldots,n\}$
  - γ: decision maker able to keep track of the way the services in S have evolved up to a certain point
  - This returns the index of a service
- The action selected by the user with the orchestrator choice determine a system history
- The O. Defines a mapping from system state and action to a service and from these elements to the next system state.
- The O. γ is said to *realize* a target service Z if it realizes all histories of Z.
- In this case, γ is also called a *composition* of Z (on S)

The valued requirement model (1)

- Main limitation: no notion of “good” or “approximate” solution exists
- Requests not of uniform importance. We introduce:

$\mathbf{P_t}$ → distribution over the actions given the state

$\mathbf{P_t(s, a)}$ → likelihood that the the user will request *a* in the target state *s*

$\boldsymbol{R_t}$ → *is the reward function (non negative)*

$\boldsymbol{R_t(s, a)}$ → *value associated to the action a in the state s*

The valued requirement model (2)

» Formally a target service is:

$$T = (\Sigma_t, \sigma_{t0}, F_t, A, \delta_t, P_t, R_t)$$

where

$$P_t : \Sigma_t \to \pi(A) \cup \emptyset$$

is the action distribution function

$$R_t : \Sigma_t \times A \to \mathbb{R}$$

is the reward function

P_t introduces a probability density function over P_∞

R_t can depend also on Σ. Can assign identical values to pairs (σ, a).

The valued requirement model (3)

- The definition of *value of history* h_t is the **SUM OF DISCOUNTED REWARDS**

$$v(\sigma_0, a_1, \sigma_1, \cdots) = \sum_{i=0}^{\infty} \lambda^i R_t(\sigma_i, a_{i+1})$$

where 0 < λ < 1 is the discount factor

- The **EXPECTED VALUE** of an orchestrator γ is defined by:

$$v(\gamma) = E_{h_t \sim P_\infty}(v(h_t) \cdot real(\gamma, h_t))$$

where $real(\gamma, h_t)$ is 1 if h_t is realizable, 0 otherwise

- The **OPTIMAL ORCHESTRATOR** is defined as:

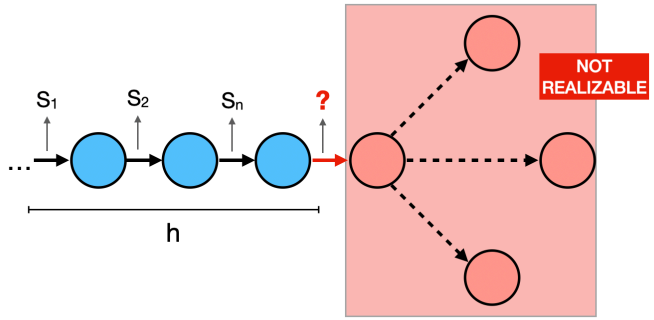
$$\gamma = \arg\max_{\text{orchestrator } \gamma'} v(\gamma')$$

where $v(\gamma')$ is the expected value of histories realizable

The valued requirement model (4)

THEOREM 1: If the target is realizable and every target history has strictly positive value then γ realizes the target iff it is an optimal orchestrator.

» If h is not realizable by γ' there exists a point in h where γ' does not assign *a* in a service that can supply it. So **any history that extends h is not realizable**.



An *optimal orchestrator* works even when the target service is not fully realizable. An *optimal controller* is able to handle more valued histories

Computing the optimal orchestrator (1)

- We can solve the model by formulating an **MDP**. It is a four tuple:

$$M' = (S', A', Tr', R')$$

where

S' → finite set of states

A' → finite set of actions

Tr' → S' x A' → π (s') is the transition function

R' → S' x A' → $\mathbb{R}$ is the transition function

Computing the optimal orchestrator (2)

- The composition MDP is a function of the system service and the target service as follows:

$$M(Z,T) = (S_M, A_M, Tr_M, R_M)$$

where

$$S_M = \Sigma_Z \times \Sigma_T \times A \cup s_{M0}$$

The **set of states** is defined by this product with a distinguished initial state s_{M0}

$$A_M = \{a_{M0}, 1, \ldots, n\}$$

The **actions** correspond to selecting the service with a special initializing action a_{M0}

$$Tr_M(s_{M0}, a_{M0}, (\sigma_{z0}, \sigma_{t0}, a)) = P_t(\sigma_{t0}, a)$$

A **transition in state** s_{M0} is defined only for **action** a_{M0}. From this state we can get to $(\sigma_{z0}, \sigma_{t0}, a)$ with a probability equal to the one that action a is requested from the target service ad its initial state.

Computing the optimal orchestrator (3)

$$Tr_M((\sigma_z, \sigma_t, a), i, (\sigma'_z, \sigma'_t, a')) = P_t(\sigma'_t, a')$$

The next system state is determined by (a, i) and the previous system state.
The next target state is determined by *a* and the previous target state.
The probability associated is the one that the action *a*' will be requested in the new target state.

$$R((\sigma_z, \sigma_t, a), i) = R_t(\sigma_t, a)$$

The reward function associates a reward to states able to perform *a*.
The value of the reward is the value of doing *a* at the target state.

THEOREM 2: Let ρ be an optimal policy for M (Z, T). Then the orchestrator γ such that $\gamma((\sigma_z, \sigma_t), a) = \rho(\sigma_z, \sigma_t, a)$ is an *optimal orchestrator*

An optimal policy for MDP is the one **maximizing** the expected sum of rewards with discount factor λ

Extensions

We can now show some useful extensions that improves the system performance, that are:

- Stochastic available services
- Handling exceptions
- Separate rewards specifications
- Non-Markovian rewards
- High-level programs as target services

Extensions

Stochastic available services

- We assumed so far deterministic scenarios
- Let us extend the model to capture stochastic services
- Service transitions are probabilistic too
- The MDP has to take also into account the stochastic transitions of the system state and target state

Extensions

Handling exceptions

- In real-world scenarios not everything goes well
- While booking a vacation, we book a flight but cannot book a hotel, we must cancel the flight reservation, that can be costly!
- If the target/composite service terminates before a terminal state has been reached, work done so far has to be undone
- For this reason we need the exception handling!
- What we do is to augment the MDP definition such that it takes these costs into account by adding a negative reward to states (s_z , s_t , a) and service choice *i* that cannot supply action *a* in its current state
- Reward size depends on the states of various services s_z

Extensions

Separate rewards specifications (1)

- Rewards coupled with likelihood of the client making certain action requests into the target service to be realized
- Convenient to keep the two specifications separated
- Use the target service only to specify the likelihood of action request
- Rewards expressed dynamically on the history of actions executed so far by the target, through a deterministic transition system called transducer

Extensions

Separate rewards specifications (2)

- A transducer is defined as $R = (\Sigma, \Delta, S, s_0, f, g)$ where:
  - Σ: input alphabet (that in our case corresponds to the set of actions A)
  - Δ: output alphabet (possible rewards expressed in real numbers R)
  - S: set of states
  - s_0 : initial state
  - $f: S \times \Sigma \longrightarrow S$ is the transition function
  - $g: S \times \Sigma \longrightarrow \Delta$ is the output (reward) function

- We obtain that rewards depend on the sequence of actions executed so far and not on the target state

Extensions

Non-Markovian rewards

- Many performance criteria call for more sophisticated reward functions that do not depend on the last state only
- For example we may want to reward a robot for picking up a cup only if it was requested to do so earlier
- The idea is to specify rewards by some variants of LTL $_f$ (linear-time temporal logic over finite traces)
- One key point is that formulas in these logics can be "translated" into standard DFAs (deterministic finite state automata) that recognize exactly the traces that fulfill the formula
- Such DFAs can be then combined with probabilistic transition systems to generate suitable MDPs to be used for generating optimal solutions
- So replace the target specification with a declarative set of logical constraints
- Then we compute the synchronous product with a target transition system that us the likelihood of action choice

Extensions

High-level programs as target services (1)

- Certain non-Markovian specifications can be expressed more naturally by using procedural constraints, in particular a sort of propositional variant of GOLOG:

$$\delta ::= A \mid \varphi? \mid \delta_1 + \delta_2 \mid \delta_1; \delta_2 \mid \delta^* \mid$$
$$\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \mid \textbf{while } \phi \textbf{ do } \delta$$

- Hence we can assign rewards to traces that correspond to successful computations of such programs

Extensions

High-level programs as target services (2)

- **if** and **while** can be seen as abbreviations for regular expression:

$$\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \doteq (\phi?;\delta_1) + (\neg\phi?;\delta_2)$$
$$\textbf{while } \phi \textbf{ do } \delta \doteq (\phi?;\delta)^*;\neg\phi?$$

- Hence these programs can also be translated into *regular expressions* and hence in DFA to be used as above
- We can combine procedural and declarative temporal constraints by adopting LDL $_f$ (a variant of LTL $_f$, called linear-time dynamic logic on finite traces):

$$[true^*]\langle \textbf{while}(cold \wedge heatingOn)) \textbf{ do } (\neg turnOffHeating^*; heat)\rangle$$

- which says that at every point in time, while it is cold and the heating is on then heat, possibly allowing other action except turning off heating

Extensions

Learning

- For Web services, statistics gathering is very simple and is carried out routinely nowadays
- So, it is not difficult to learn the stochastic transition function of existing services online, and use it to specify the probabilistic elements of the model

Example 1 (1)
Robot Assembly (S1 System)

- Let's consider the ROMAN MODEL (non stochastic case). Service is defined as: $S = (\Sigma, \sigma_0, F, A, \delta)$
- The service is a Future Robot Creator with which you can generate from scratch a new robot starting from these category like:
  - Home
  - Company..
- Chosen a category, the Tool shows the Client all the devices which he can put in a robot, for example:
  - Arm
  - Wheel..

s0: startingPage (initial state)
s1: category of robots
s2: category devices
s3: add device
s4: remove device

Example 1 (2)
Robot Store Assembly (S2 System)

- Finished to assembly the Robot, the System shows:
  - Generate Model or return Back
  - Choose the Material
  - Choose the Environment where the robot have to work (like Rain, Snow ect..)
  - Generate the rendering of the model

s5: robotModelConfirmed (init_state)
s6: finalComponent
s7: chooseEnvironment
s8: renderingModel

Example 1 (3)
Robot Store Assembly (S3 System)

- If the client is satisfied with the created model, he can:
  - “Export the model”
  - Choose the "extension file”
  - "Rename" the file
  - "Save" Model
  - Export the model to the Store of application
  - And maybe, in future, “sell" the virtual model!

s9: exportModel (init_state)
s10: extensionFile
s11: renameModel
s12: saveModel
s13: shareToStore

Now the SERVICE COMMUNITY can be expressed like: S = {S1, S2, S3}

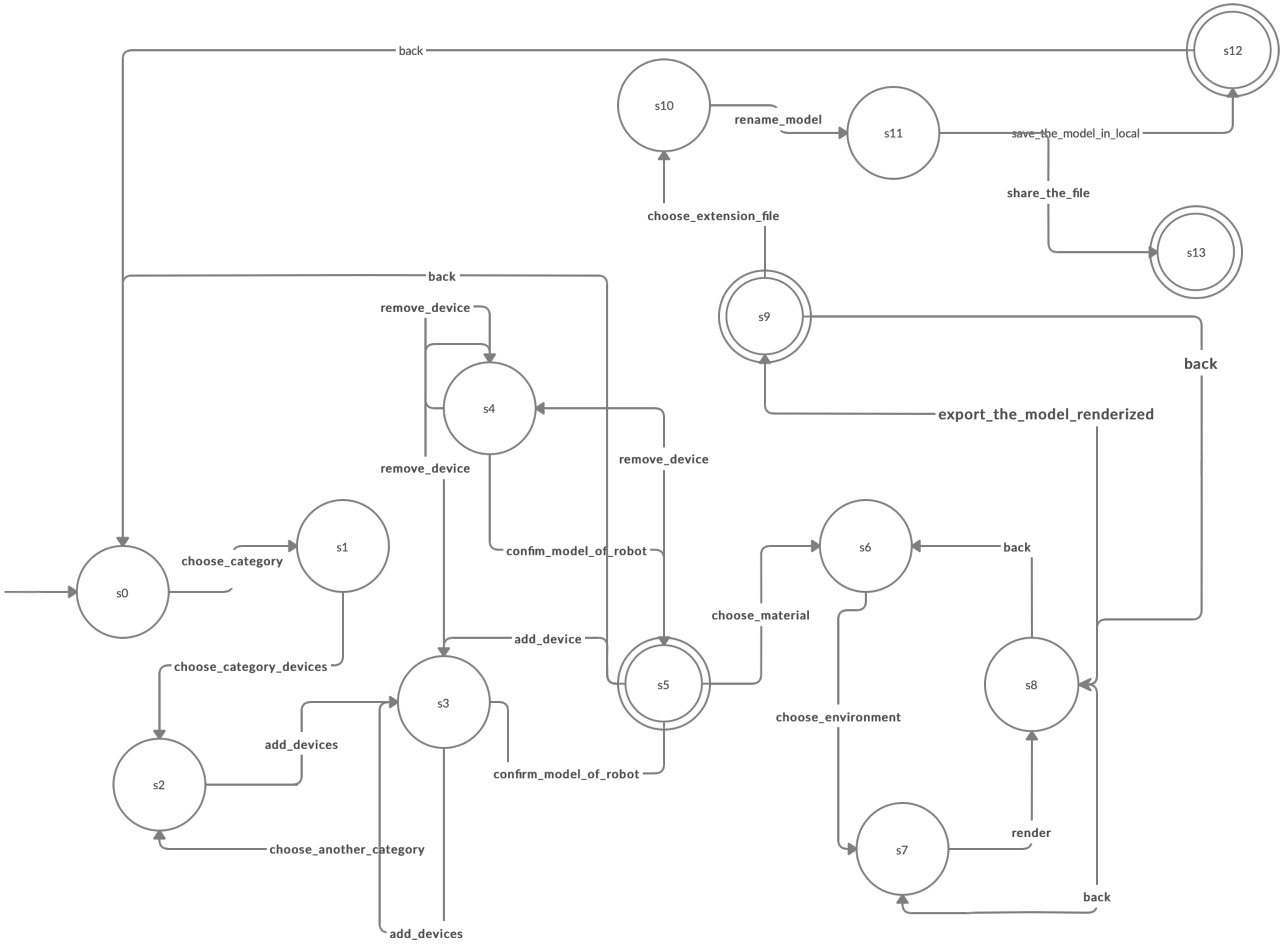
Example 1 (4)
The System Services Z

» Now we conclude with the History of Z:

Ex. of History of S:
s0 · choose_category · s1 · choose_cat_devices · s2
· add_devices · s3 · remove_device · s4
· add_device · s2 · confirm_the_robot · **s5**
s0 · choose_category · s2 · choose_cat_devices · s2
· add_devices · s3 · confirm_robot · s5
· chose_material · s6 · chose_env · s7
· chose_render_model · s8 · export_the_model · s9
· choose_extension · s10 · rename_the_model · s11
· save_model · **s12**
[…]

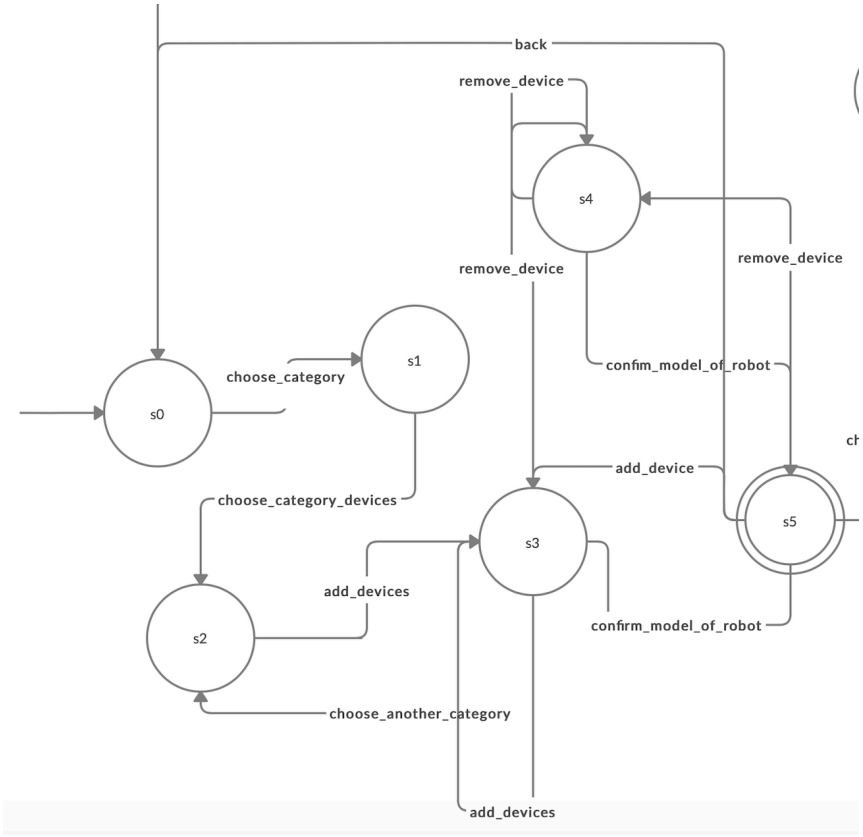
Example 1 (5)
The System Services Z

» Now we illustrate the Automata of System Services Z:



Example 1 (6)
The System Services Z

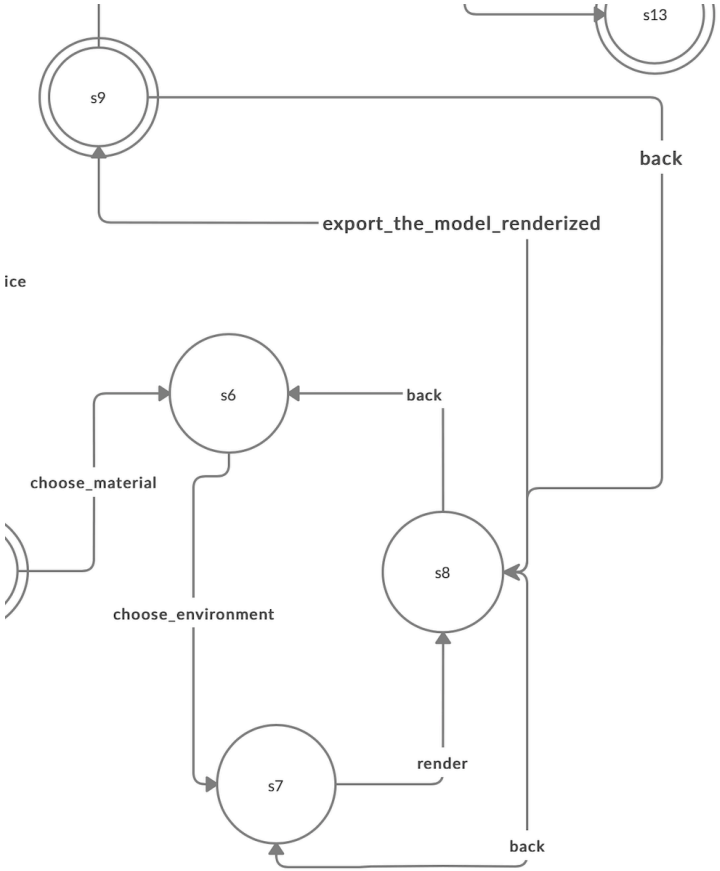
» Now we show the System (S1) which composes Z:



Example 1 (7)

The System Services Z

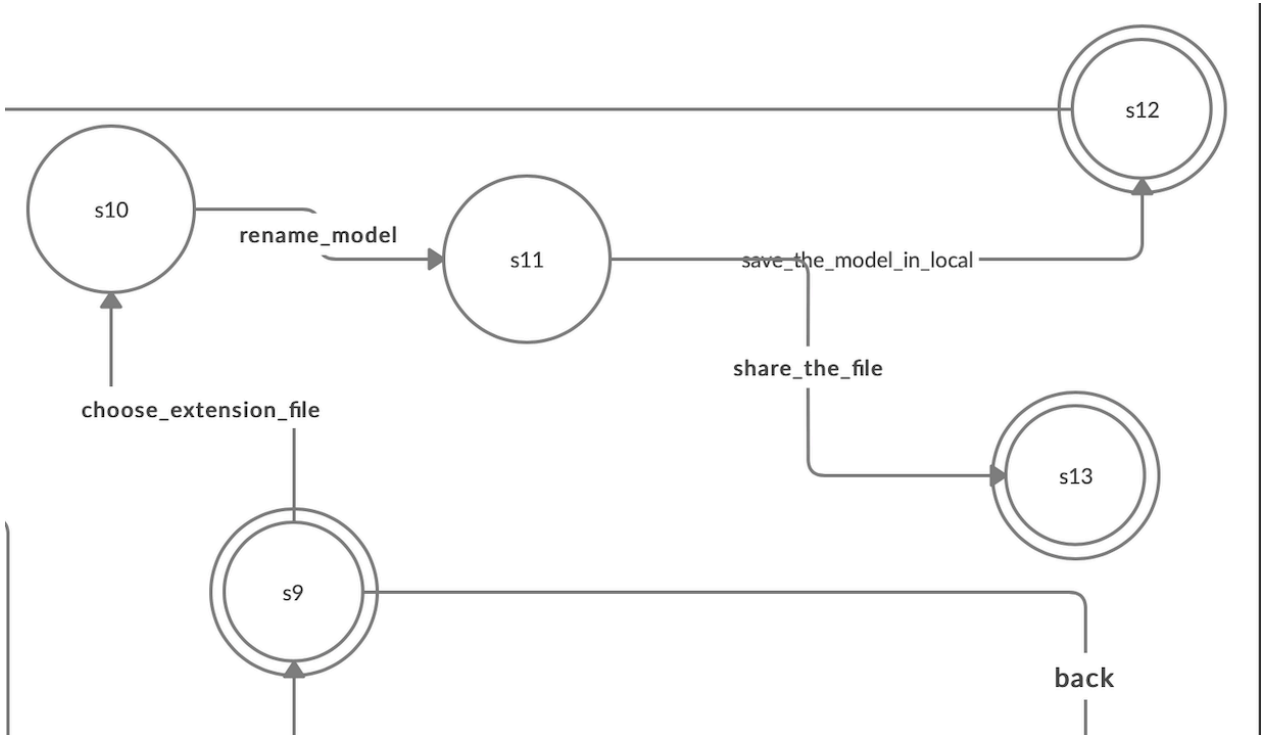
» Now we show the System (S2) which composes Z:



Example 1 (8)

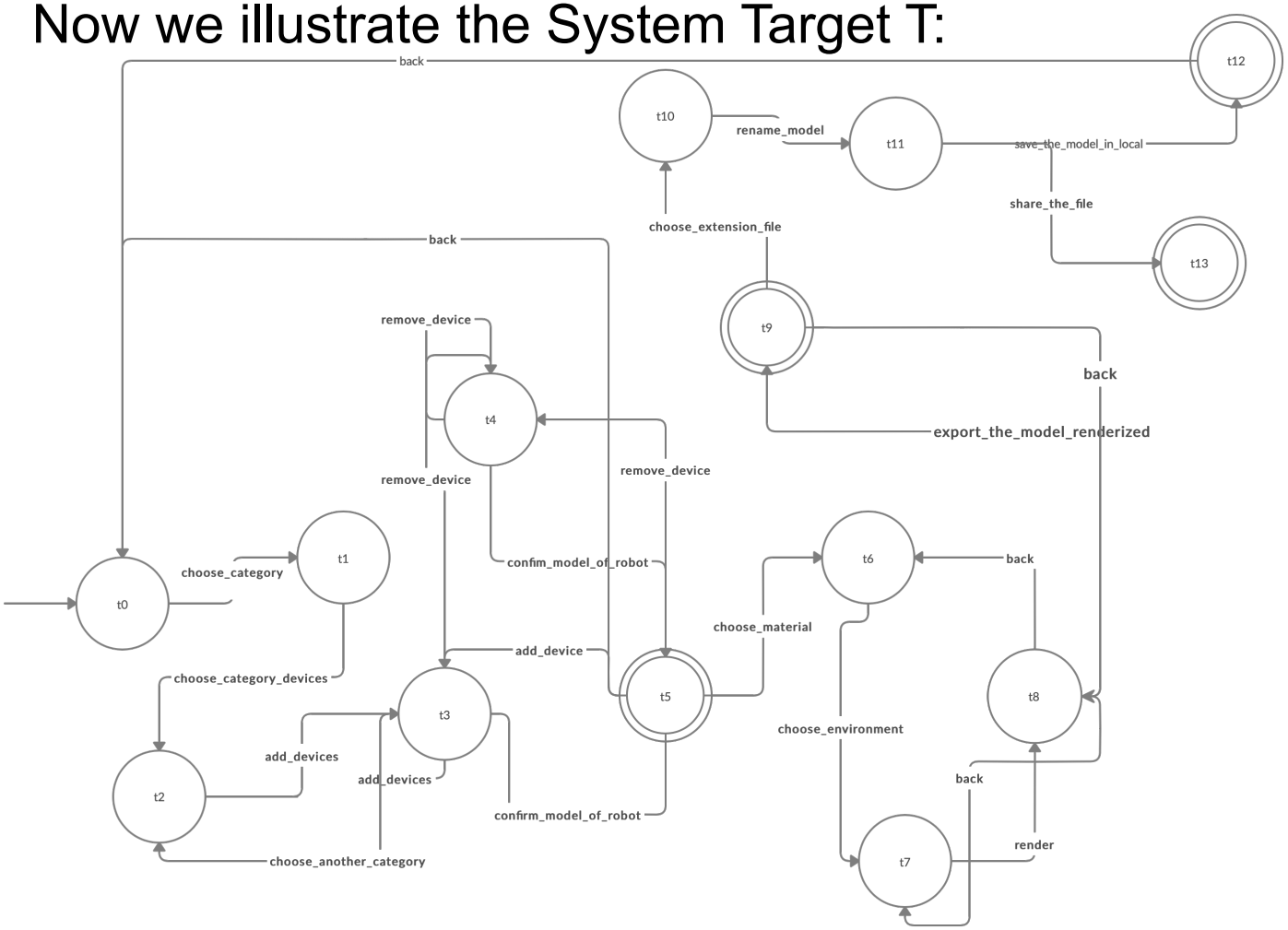
The System Services Z

- Now we show the System (S3) which composes Z:



Example 1 (9)
The System Services T

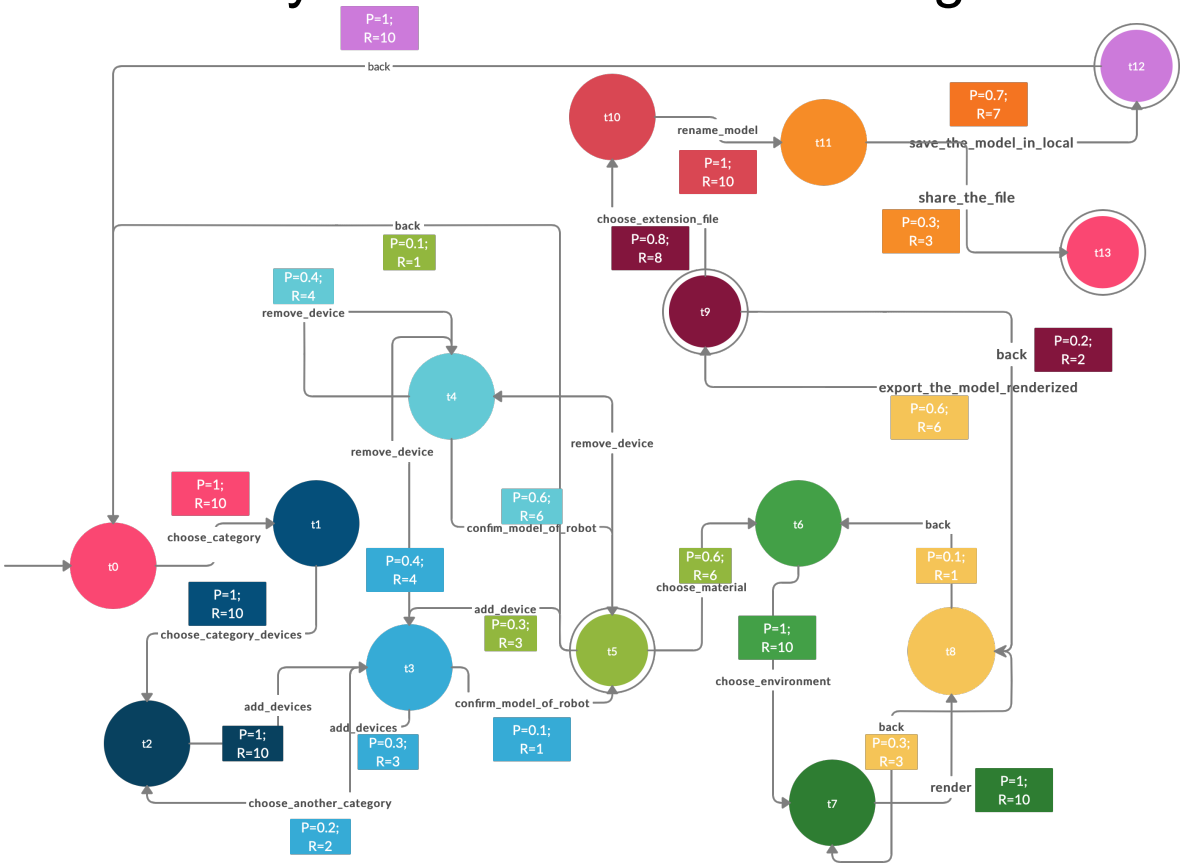
» Now we illustrate the System Target T:



Example 1 (10)

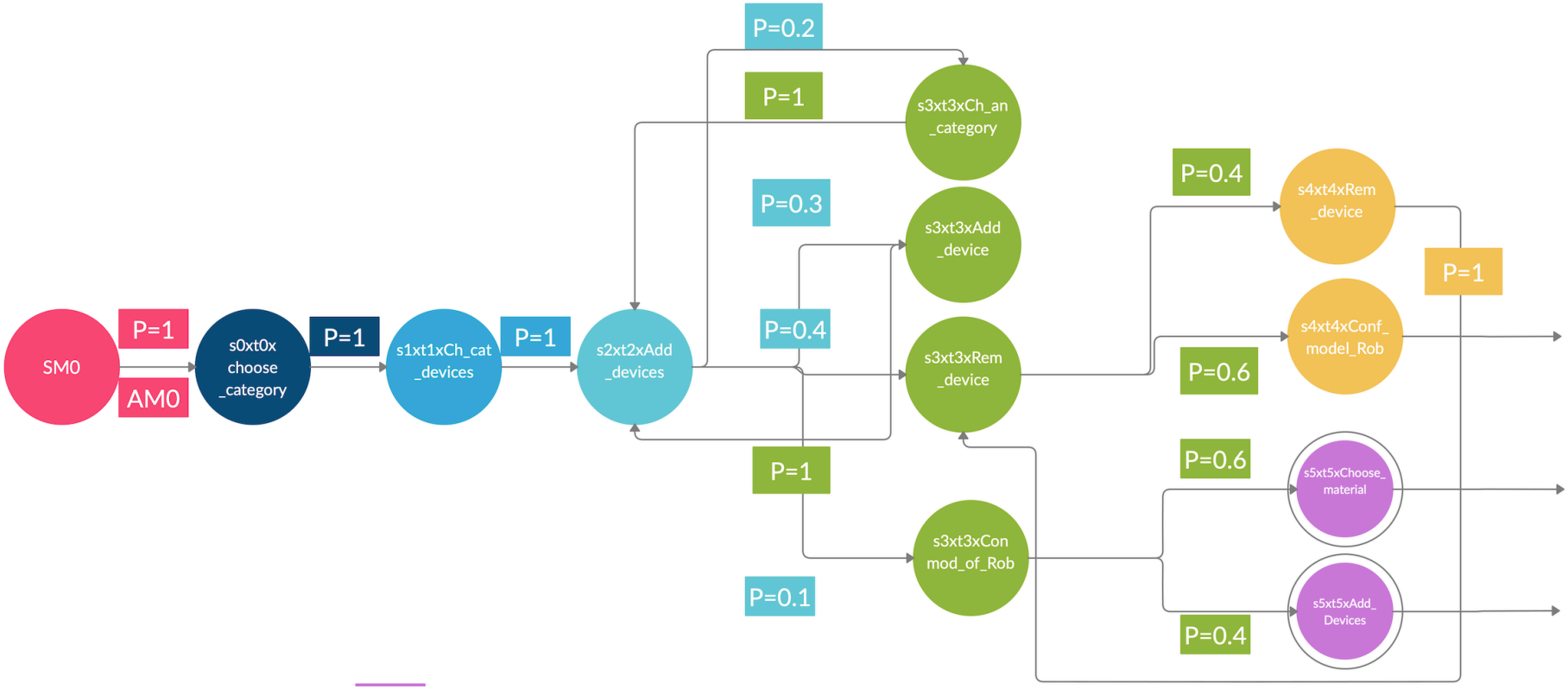
The System Services T

- The System Target T has the intrinsic characteristics of:
  - Probability P and Reward R coming out from action



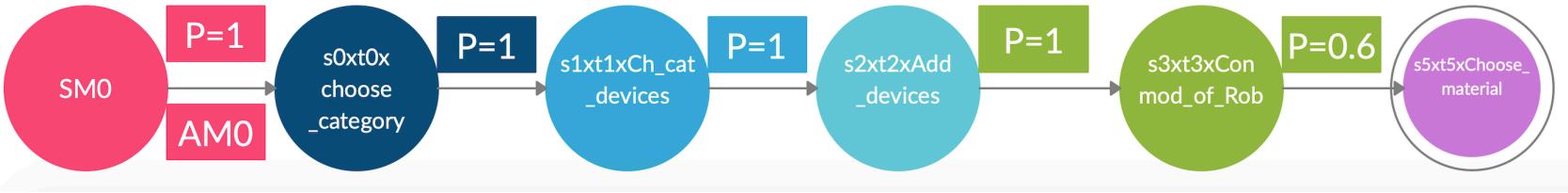
Example 1 (11)
Markov Decision Process

» Now we show the MDP of The First System:



Example 1 (12)
Markov Decision Process

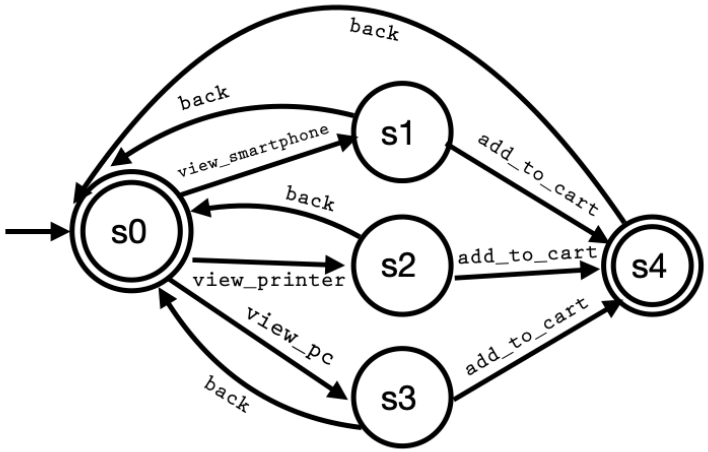
» Now we show the best trace of the MDP (S1):



EXAMPLE 2 (1)
The S1 system

- Let's consider the **ROMAN MODEL** (non stochastic case). Service is defined as: $S = (\Sigma, \sigma_0, F, A, \delta)$

- This service allows the user to choose among **different electronic products**. Once he has done his choice, can visualize the information about the product. If it is not satisfied about it, he can go **back** and chose another product.

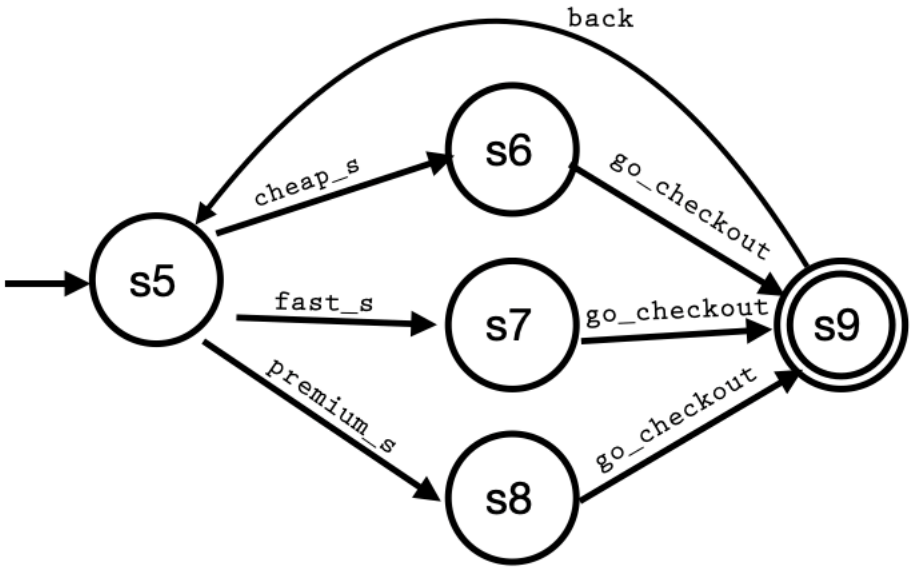


s0: initial state
s1: smartphone description
s2: printer description
s3: pc description
s4: product in the cart

History of S:
s0 · view_smartphone · s1 · add_to_cart · s4
s0 · view_smartphone · s1 · back · s0
s0 · view_printer · s2 · add_to_cart · s4
[…]

EXAMPLE 2 (2)
The S2 system

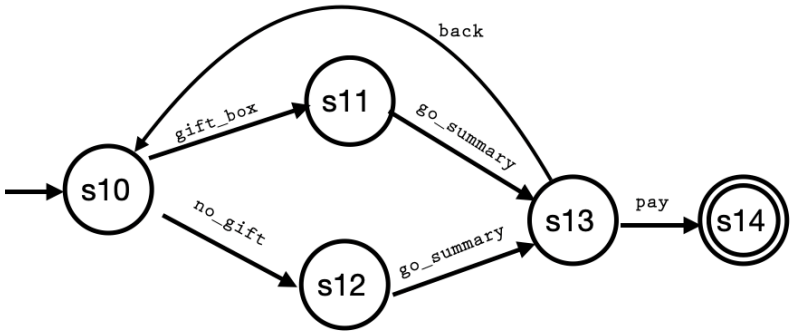
» Once the product is in the cart, the user can chose among different types of shipping: **cheap**, **fast** or **premium**. The type of shipping influences the delivery time and the supplier reward.



s5: product in the cart (initial state)
s6: cheap shipping chosen
s7: fast shipping chosen
s8: premium shipping chosen
s9: checkout page

EXAMPLE 2 (3)
The S3 system

» Once the checkout page is visualized, the user can choose to add a **gift box** to the order, or not. Then he can see a summary before proceeding with the payment. From the summary page, if the user pays, then the order is marked as **completed**.



s10: checkout page (initial state)
s11: gift box added
s12: gift box not added
s13: summary page
s14: order completed

We can express the SERVICE COMMUNITY as: S = {S1, S2, S3}

EXAMPLE 2 (4)
Target and orchestrator

» Suppose to have the following target service where, in the optimistic case, the user chooses a smartphone, the premium shipping, a gift box and pays the order without any afterthought. It can be expressed by:

t0 · view_smartphone · t1 · add_to_cart · t4 · premium_s · t8 · go_checkout · · t9 · gift_box · t11 · go_summary · t13 · pay · t14

We need to **combine** the available services.

Once we have collected all the services in the Z virtual entity, the orchestrator γ will return the **index** of the service able to solve the action.

$$\gamma \begin{cases} \text{action: add\_to\_cart} \\ \text{service: } \mathbf{S1} \end{cases}$$

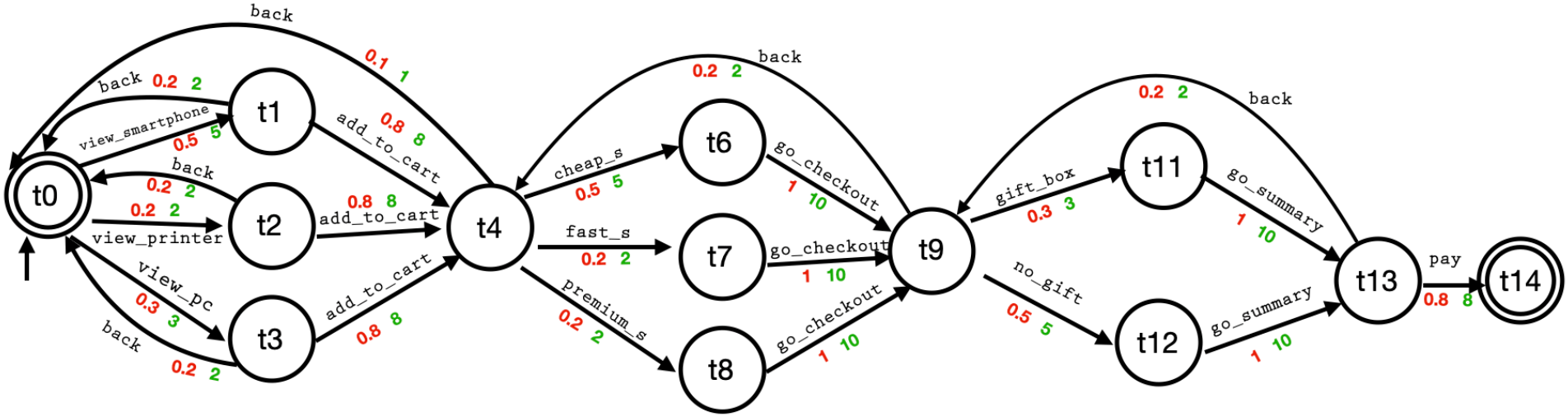
If the action is not available in any of the services?

$$\gamma \begin{cases} \text{action: add\_another\_object} \\ \text{service: } \mathbf{???} \end{cases}$$

Target service is not FULLY REALIZABLE. No solution in this case, but…

EXAMPLE 2 (5)
The valued requirement model

» We introduce the probability P_t and R_t



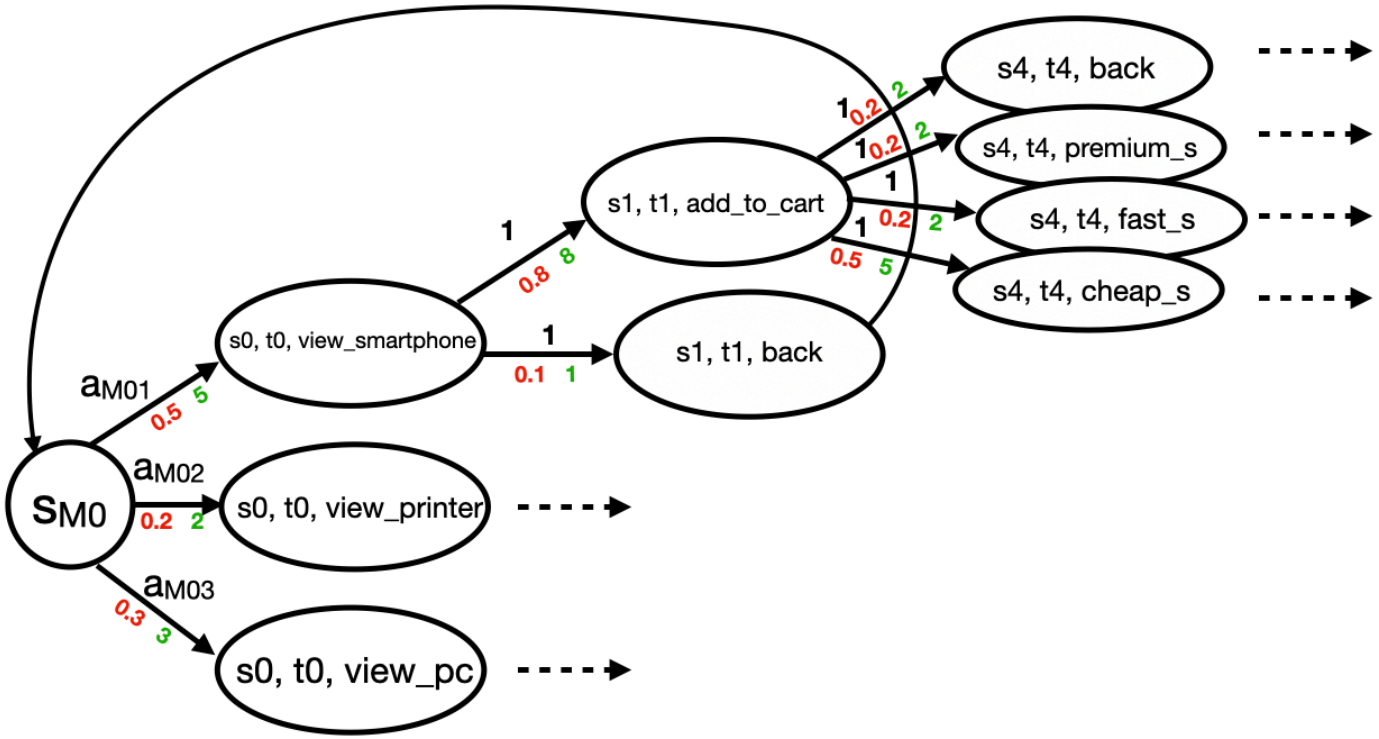
The value associated to the target, choosing λ = 0.9 is:

t0 · view_smartphone · t1 · add_to_cart · t4 · premium_s · t8 · go_checkout · · t9 · gift_box · t11 · go_summary · t13 · pay · t14

$5 + 0.9*8 + 0.9^2*2 + 0.9^3*10 + 0.9^4*3 + 0.9^5*10 + 0.9^6*8 =$ **33.23**

EXAMPLE 2 (6)
The MDP formulation

» Now we can formulate the MDP (partial), considering also Pt and Rt



EXAMPLE 2 (6)
The MDP formulation

» Suppose to have the following target that will define a path in the MDP

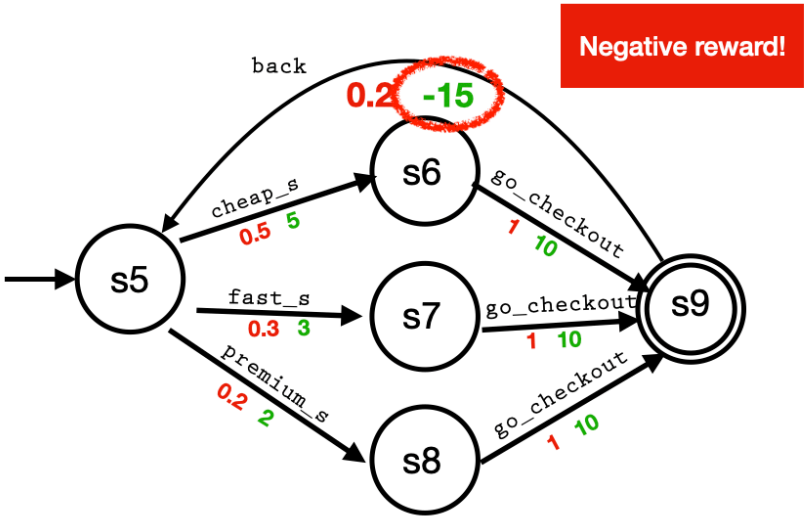
t0 · view_pc · t3 · add_to_cart · t4 · cheap_s · t6 · go_checkout · t9 · no_gift · t12 · · go_summary · t13 · pay · t14

The **optimal policy** for the MDP is the one maximizing the expected discounted sum of rewards

EXAMPLE 2 (7)
Improvement

- It can happens that, when the buyer is viewing the checkout page, he decides that the price is too high and he wants to go **back** and choose another product.
- He can also decide to change the shipping method after have seen that the delivery days are too much. Since it can be translated into a loss of the work done so far, we can define the **size of the reward** in a way that **reflects the work that needs to be undone** in each of the existing services.

  For example, considering S2:



Food delivery (1)
S1 System

- Let's consider the ROMAN MODEL (non stochastic case). Service is defined as: $S = (\Sigma, \sigma_0, F, A, \delta)$
- This service allows you to order and deliver food from a certain restaurant choosing from various categories:
  - Pizza
  - Pasta
  - Drink
  - and so on...
- Chosen a category, the Tool shows the Client all the dishes of that category, for example:
  - Carbonara
  - Amatriciana...

s0: welcomePage (initial state)
s1: choosenRestaurant
s2: choosenCategory
s3: dish added or removed
s4: order done

Food delivery (2)
S2 System

- Completed the order, the System shows a summary screen where you can:
  - Modify the order
  - Confirm the order
  - Cancel order

s5: summary order (initial state)
s6: orderConfirmed
s7: orderCanceled

Food delivery (3)
S3 System

- Once the order has been confirmed, the Client is directed to the shipping and payment page where he can:
  - enter the shipping address
  - enter a coupon
  - select terms of payment
  - proceed to payment

s8: personal informarions (initial state)
s9: enteredAddress
s10: enteredCoupon
s11: choosenPayment
s12: paymentDone

Now the SERVICE COMMUNITY can be expressed as S = {S1, S2, S3}

References

» Ronen I. Brafman, Giuseppe De Giacomo, Massimo Mecella, Sebastian Sardina - *Service Composition in Stochastic Settings.* AI*IA 2017 Advances in Artificial Intelligence, Springer (2017)