Transition Systems and Service Composition

Giuseppe De Giacomo

Seminari di Ingegneria del Software
A.A. 2005/2006

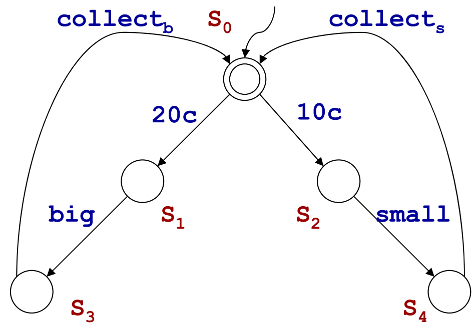
Transition Systems

Transition Systems

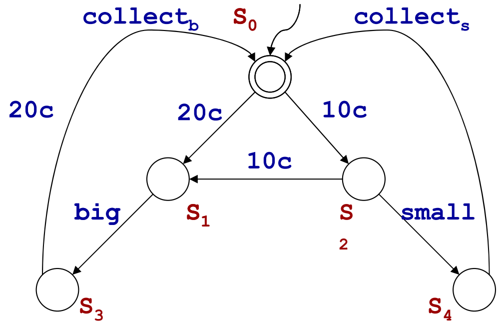
- A transition system TS is a tuple T = < A, S, S^0, δ, F> where:
  - A is the set of actions
  - S is the set of states
  - $S^0 \subseteq S$ is the set of initial states
  - $\delta \subseteq S \times A \times S$ is the transition relation
  - $F \subseteq S$ is the set of final states
- Variants:
  - No initial states
  - Single initial state
  - Deterministic actions
  - States labeled by propositions other than Final/¬Final

(c.f. Kripke Structure)

Example (Vending Machine)

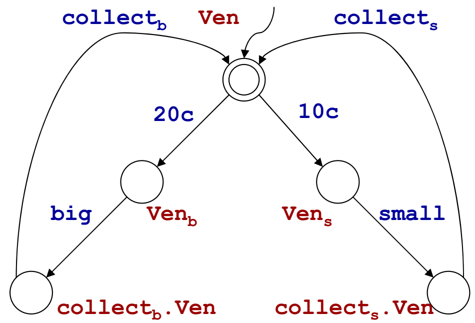


Example (Another Vending Machine)



Process Algebras are Formalisms for Describing TS

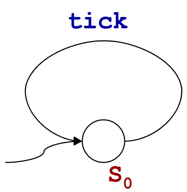
- Trans (a la CCS)
  - $Ven = 20c.Ven_b + 10c.Ven_s$
  - $Ven_b = big.collect_b.Ven$
  - $Ven_l = small.collect_s.Ven$
- Final
  - $\surd$ Ven



- *TS may have infinite states - e.g., this happens when generated by process algebras involving iterated concurrency*
- *However we have good formal tools to deal only with finite states TS*

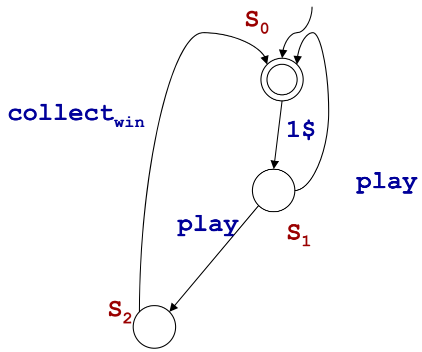
Example (Clock)

TS may describe (legal) nonterminating processes

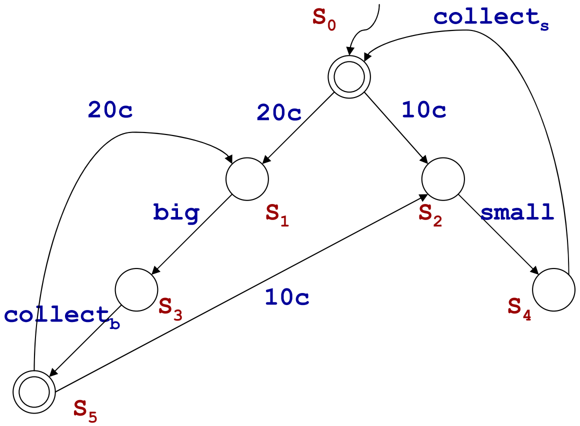


Example (Slot Machine)

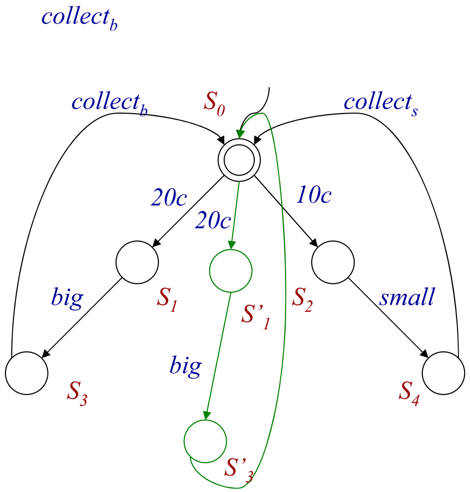
Nondereminisic transitions express choice that is not under the control of clients



Example
(Vending Machine - Variant 1)



Example
(Vending Machine - Variant 2)



Bisimulation

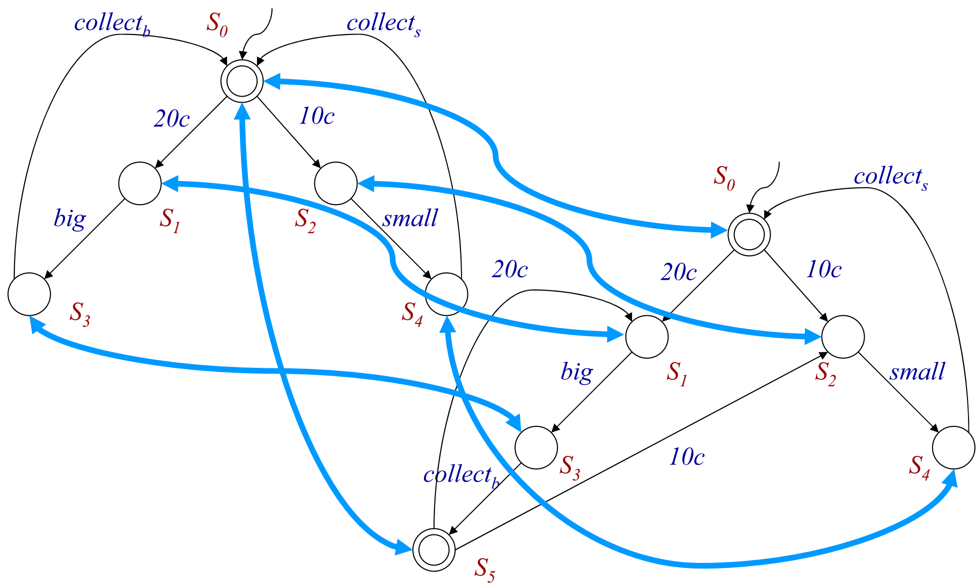
- A binary relation *R* is a **bisimulation** iff:

  (s,t) $\in$ *R* implies that
  - s is *final* iff t is *final*
  - for all actions a
    - if s $\rightarrow_a$ s′ then $\exists$ t′ . t $\rightarrow_a$ t′ and (s′,t′)$\in$ *R*
    - if t $\rightarrow_a$ t′ then $\exists$ s′ . s $\rightarrow_a$ s′ and (s′,t′)$\in$ *R*

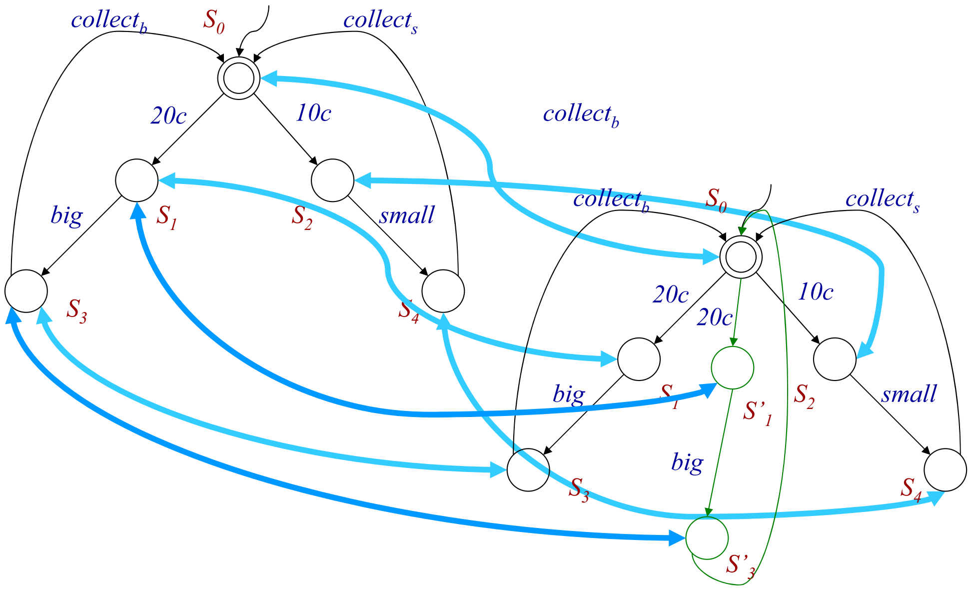
  Note it is a co-inductive definition!

- A state s_0 of transition system S is **equivalent** to a state t_0 of transition system T iff there **exists** a **bisimulation** between the initial states s_0 and t_0.

Example of Bisimulation

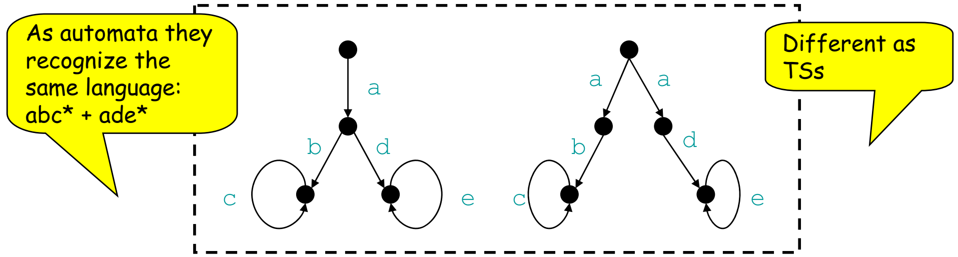


Example of Bisimulation



Automata vs.Transition Systems

- Automata
  - define sets of runs (or traces or strings): (finite) length sequences of actions
- TSs
  - … but I can be interested also in the alternatives "encountered" during runs, as they represent client's "choice points"



Logics of Programs

- Are modal logics that allow to describe properties of transition systems
- Examples:
  - HennesyMilner Logic
  - Propositional Dynamic Logics
  - Modal (Propositional) Mu-calculus
- Perfectly suited for describing transition systems: they can tell apart transition systems modulo bisimulation

HennessyMilner Logic

- Φ := P | *(atomic propositions)*
  $\neg \Phi \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid$ *(closed under boolean operators)*
  $[a]\Phi \mid \langle a\rangle\Phi$ *(modal operators)*
- Propositions are used to denote final states
- $\langle a\rangle\Phi$ means there exists an a-transition that leads to a state where Φ holds; i.e., expresses the capability of executing action a bringing about Φ
- $[a]\Phi$ means that all a-transitions lead to states where Φ holds; i.e., express that executing action a brings about Φ

Logics of Programs: Examples

- Usefull abbreviation:
  - $<any>\ \Phi$ stands for $<a_1>\Phi \vee \cdots \vee <a_n>\Phi$
  - $[any]\ \Phi$ stands for $[a_1]\Phi \wedge \cdots \wedge [a_n]\Phi$
  - $<any - a_1>\ \Phi$ stands for $<a_2>\Phi \vee \cdots \vee <a_v>\Phi$
  - $[any - a_1]\ \Phi$ stands for $[a_2]\Phi \wedge \cdots \wedge [a_v]\Phi$

- Examples:
  - <a>true *cabability of performing action a*
  - [a]false *inability of performing action a*
  - $\neg$Final $\wedge$ <any>true $\wedge$ [any-a]false *necessity/inevitability of performing action a i.e., action a is the only action possible*
  - $\neg$Final $\wedge$ [any]false *deadlock!*

Propositional Dynamic Logic

- $\Phi := P \mid$ *(atomic propositions)*
  $\neg \Phi \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid$ *(closed under boolean operators)*
  $[r]\Phi \mid <r>\Phi$ *(modal operators)*

  $r := a \mid r_1 + r_2 \mid r_1;r_2 \mid r^* \mid P?$ *(complex actions as regular expressions)*

- Essentially add the capability of expressing partial correctness assertions via formulas of the form
  - $\Phi_1 \rightarrow [r]\Phi_2$ *under the conditions Φ_1 all possible executions of r that terminate reach a state of the TS where Φ holds*
- Also add the ability of asserting that a property holds in all nodes of the transition system
  - $[(a_1 + \cdots + a_v)^*]\Phi$ *in every reachable state of the TS Φ holds*
- Useful abbereviations:
  - any stands for $(a_1 + \cdots + a_v)$ *- observe that + can be expressed in HM Logic*
  - u stands for any* *- this is the so called master/universal modality*

Modal Mu-Calculus

- $\Phi := P \mid$ *(atomic propositions)*
  $\neg \Phi \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid$ *(closed under boolean operators)*
  $[r]\Phi \mid <r>\Phi$ *(modal operators)*

  $\mu X.\Phi(X) \mid \nu X.\Phi(X)$ *(fixpoint operators)*

- It is the most expressive logic of the family of logics of programs.
- It subsumes
  - PDL (modalities involving complex actions are translated into fomulas involving fixpoints)
  - LTL (linear time temporal logic),
  - CTS, CTS* (branching time temporal logics)

- Examples:
- $[any^*]\Phi$ can be expressed as $\nu X.\ \Phi \wedge [any]X$

- $\mu X.\ \Phi \vee [any]X$ *along all runs eventually Φ*
- $\mu X.\ \Phi \vee <any>X$ *along some run eventually Φ*
- $\nu X.\ [a](\mu Y.\ <any>true \wedge [any-b]Y) \wedge X$
  every run that that contains a contains later b

Model Checking

- Model checking is polynomial in the size of the TS for
  - HennessyMilner Logic
  - PDL
  - Mu-Calculus
- Also model checking is wrt the formula
  - Polynomial for HennessyMiner Logic
  - Polynomial for PDL
  - Polynomial for Mu-Calculus with bounded alternation of fixpoints and NP$\cap$coNP in general

Model Checking

- Given a TS T, one of its states s, and a formula Φ verify whether the formula holds in s. Formally:

$$T,s \models \Phi$$

- Examples (TS is our vending machine):
  - $S_0 \models$ Final
  - $S_0 \models$ <10c>true   *capability of performing action 10c*
  - $S_2 \models$ [big]false   *inability of performing action big*
  - $S_0 \models$ [10c][big]false   *after 10c cannot execute big*
  - $S_i \models \mu$ X. Final $\vee$ [any] X   *eventually a final state is reached*
  - $S_0 \models \nu$ Z. (μ X. Final $\vee$ [any] X) $\wedge$ [any] Z   *or equivalently*
    $S_0 \models$ [any*](μ X. Final $\vee$ [any] X)
    from everywhere eventually final

Planning as Model Checking

- **Build the TS of the domain:**
  - Consider the set of states formed all possible truth value of the propositions (this works only for propositional setting).
  - Use Pre's and Post of actions for determining the transitions

  Note: the TS is exponential in the size od the description.

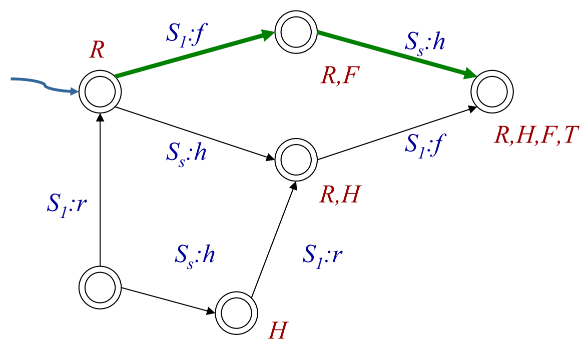
- **Write the goal in a logic of program**
  - typically a single least fixpoint formula of Mu-Calculus
- **Planning:**
  - model check the formula on the TS starting from the given initial state.
  - use the path (paths) used in the above model checking for returning the plan.

- *This basic technique works only when we have complete information (or at least total observability on state):*
  - *Sequential plans if initial state known and actions are deterministic*
  - *Conditional plans if many possible initial states and/or actions are nondeterministic*

Example

- Operators (Services + Mappings)
  - Registered ∧ ¬FlightBooked → [S_1:bookFlight] FlightBooked
  - ¬Registered → [S_1:register] Registered
  - ¬HotelBooked → [S_2:bookHotel] HotelBooked
- Additional constraints (Community Ontology):
  - TravelSettledUp ≡
    FlightBooked ∧ HotelBooked ∧ EventBooked
- Goals (Client Service Requests):
  - Starting from state
    Registered ∧ ¬FlightBooked ∧ ¬ HotelBooked ∧ ¬EventBooked
    check <any*>TravelSettedUp
  - Starting from all states such that
    ¬FlightBooked ∧ ¬ HotelBooked ∧ ¬EventBooked
    check <any*>TravelSettedUp

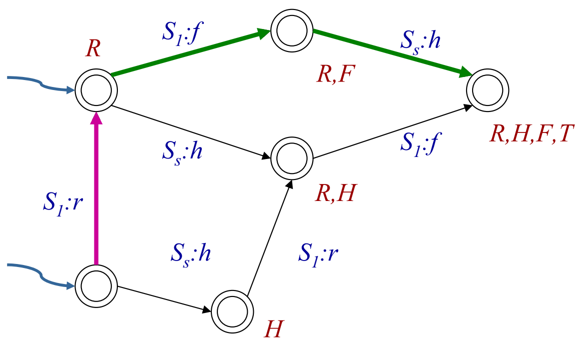
Example



Plan:
S_1:bookFlight;
S_2:bookHotel

Starting from state
Registered ∧ ¬ FlightBooked ∧ ¬ HotelBooked ∧ ¬ EventBooked
check
<any*>TravelSettledUp

Example



Plan:
if(¬Registered) {
S_1:register;
}
S_1:bookFlight;
S_2:bookHotel

Starting from states where
¬ FlightBooked ∧ ¬ HotelBooked ∧ ¬ EventBooked
check
<any*>TravelSettledUp

Satisfiability

- Observe that a formula Φ may be used to select among all TS T those such that for a given state s we have that T,s $\models \Phi$
- SATISFIABILITY: Given a formula Φ verify whether there exists a TS T and a state s such that. Formally:

  check whether exists T, s such that T,s $\models \Phi$
- Satisfiability is:
  - PSPACE for HennesyMilner Logic
  - EXPTIME for PDL
  - EXPTIME for Mu-Calculus

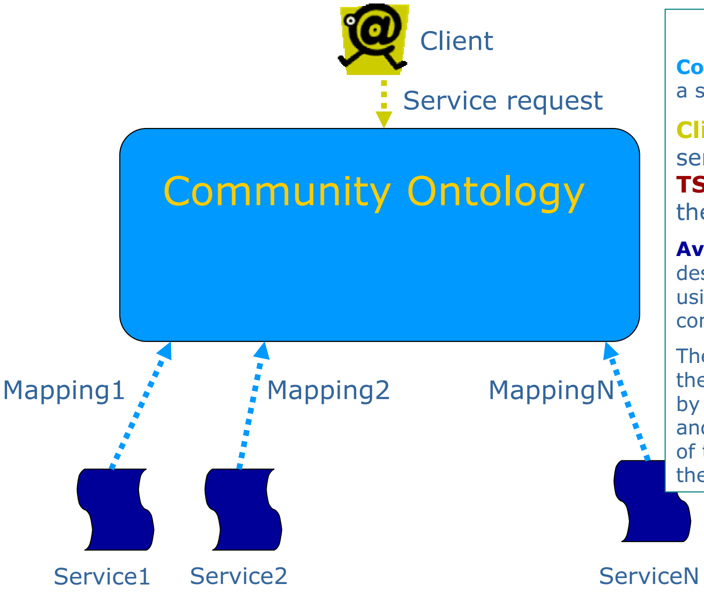
References

[Stirling Banff96] C. Stirling: Modal and temporal logics for processes. Banff Higher Order Workshop LNCS 1043, 149-237, Springer 1996
[Bradfield&Stirling HPA01] J. Bradfield, C. Stirling: Modal logics and mu-calculi. Handbook of Process Algebra, 293-332, Elsevier, 2001.
[Stirling 2001] C. Stirling: Modal and Temporal Properties of Processes. Texts in Computer Science, Springer 2001
[Kozen&Tiuryn HTCS90] D. Kozen, J. Tiuryn: Logics of programs. Handbook of Theoretical Computer Science, Vol. B, 789–840. North Holland, 1990.
[HKT2000] D. Harel, D. Kozen, J. Tiuryn: Dynamic Logic. MIT Press, 2000.
[Clarke& Schlingloff HAR01] E. M. Clarke, B. Schlingloff: Model Checking. Handbook of Automated Reasoning 2001: 1635-1790
[CGP 2000] E.M. Clarke, O. Grumberg, D. Peled: Model Checking. MIT Press, 2000.
[Emerson HTCS90] E. A. Emerson. Temporal and Modal Logic. Handbook of Theoretical Computer Science, Vol B: 995-1072. North Holland, 1990.
[Emerson Banff96] E. A. Emerson. Automated Temporal Reasoning about Reactive Systems. Banff Higher Order Workshop, LNCS 1043, 111-120, Springer 1996
[Vardi CST] M. Vardi: Alternating automata and program verification. Computer Science Today -Recent Trends and Developments, LNCS Vol. 1000, Springer, 1995.
[Vardi etal CAV94] M. Vardi, O. Kupferman and P. Wolper: An Automata-Theoretic Approach to Branching-Time Model Checking (full version of CAV'94 paper).

Name by ***Rick Hull***

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

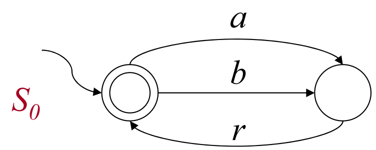
The **community** realizes the **client's target service** by "reversing" the mapping and hence using **fragments** of the computation of the the **available services**

(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

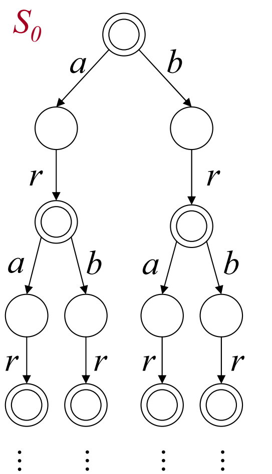
Example:



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

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 (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*

Formalizing Service Composition

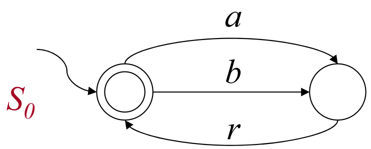
Composition:
- coordinating program …
- … that realizes the target service …
- … by suitably coordinating available services

⇒ Composition can be formalized 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 (1)

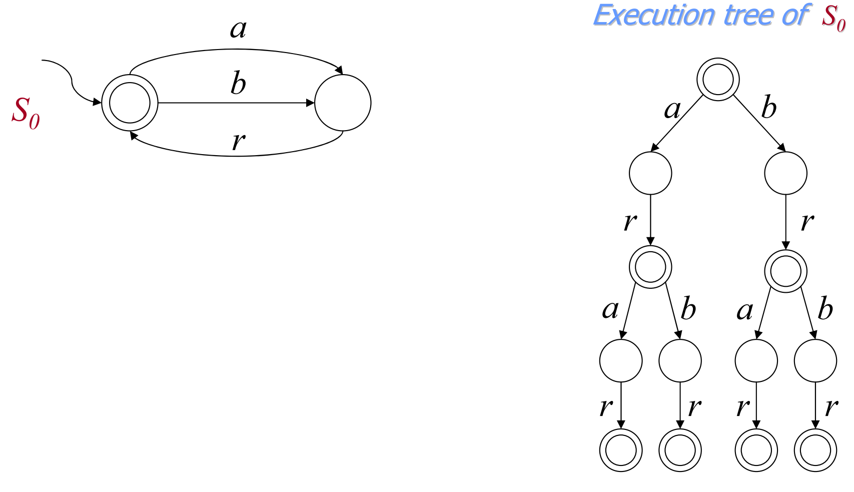
- Available services

- Target service



Example of Composition (2)

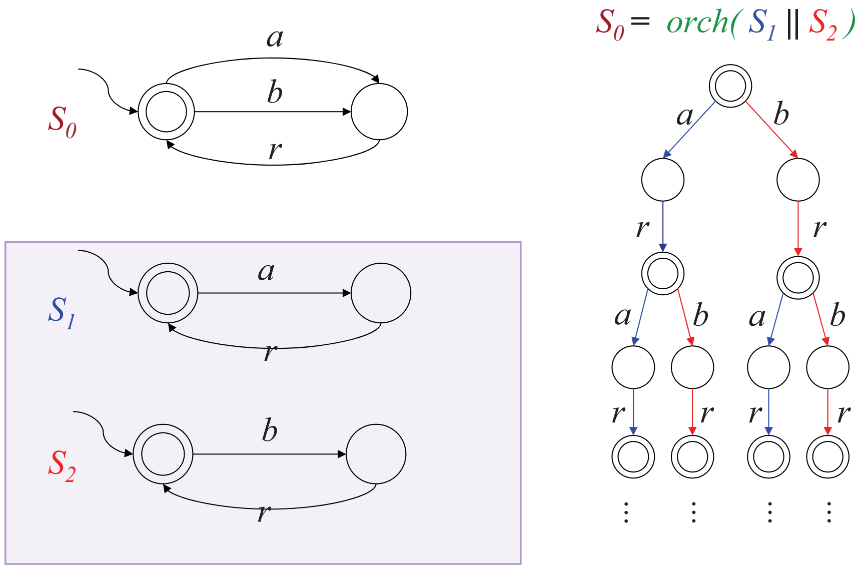
Execution tree of S_0



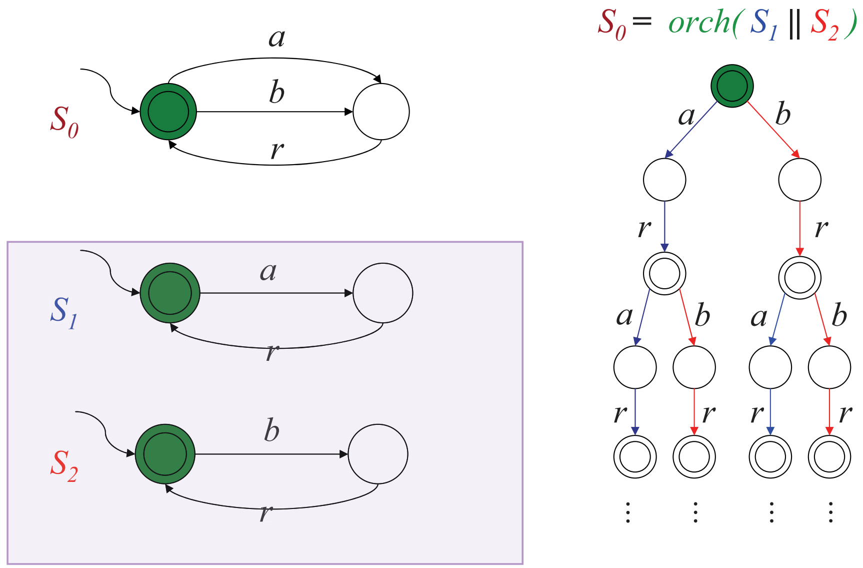
Note: we cannot label the target service TS directly …

… we need to label the execution tree

Example of Composition (3)

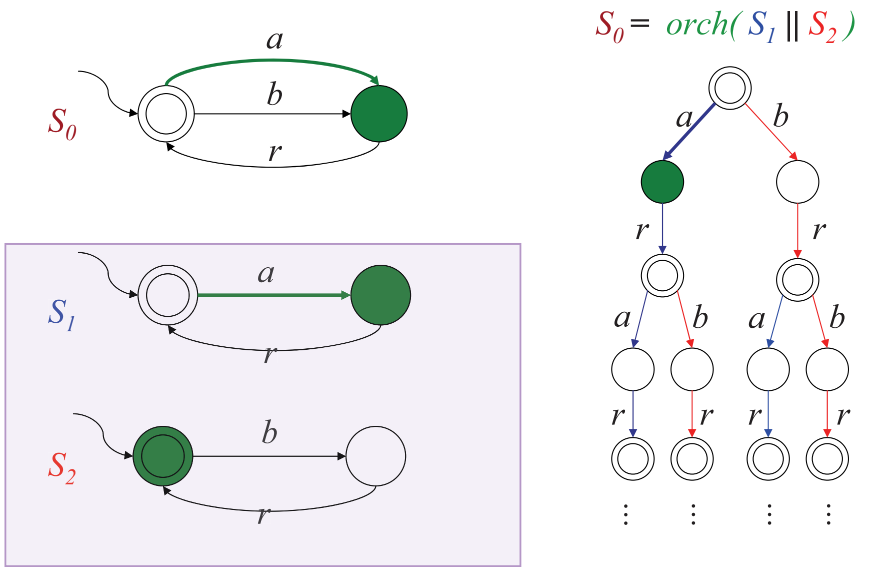


Example of Composition (4)



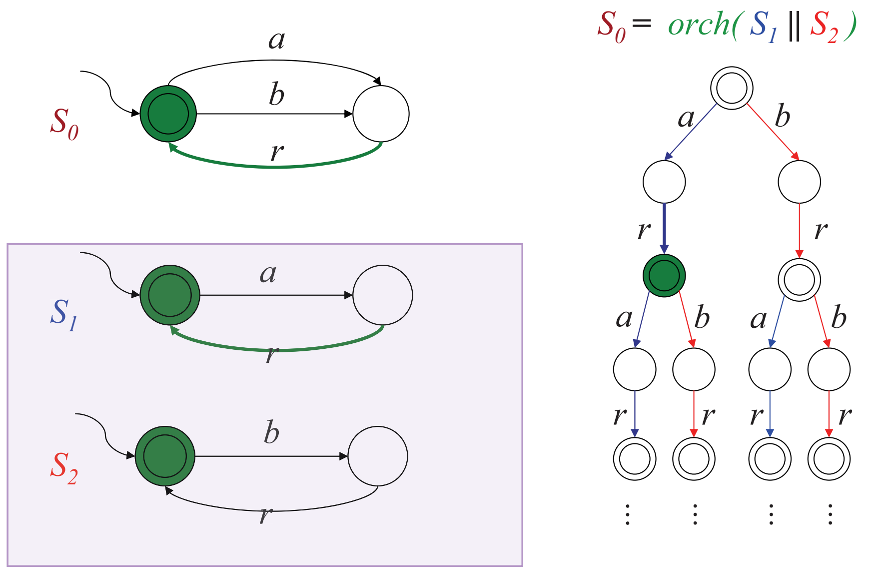
All services start from their starting state

Example of Composition (5)



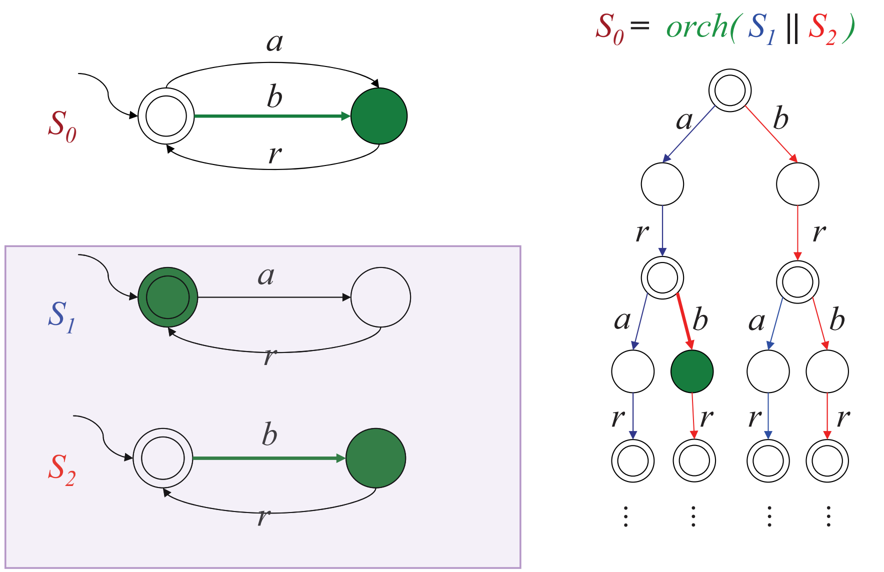
Each action of the target service is executed by at least one of the component services

Example of composition (6)

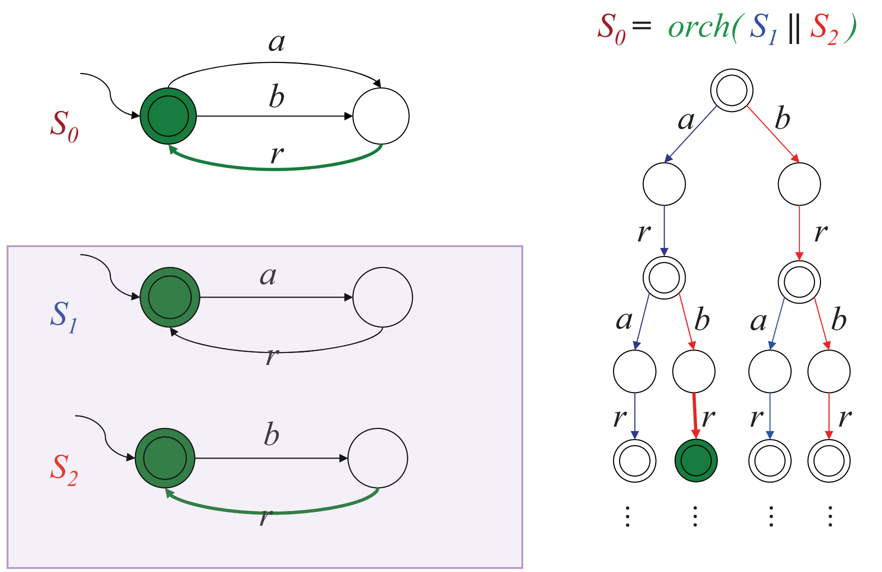


When the target service can be left, then all component services must be in a final state

Example of composition (7)

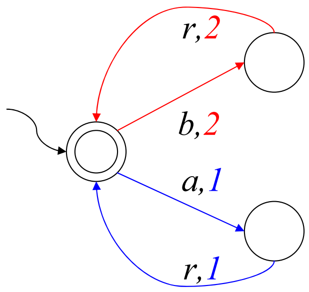


Example of composition (8)



Observation

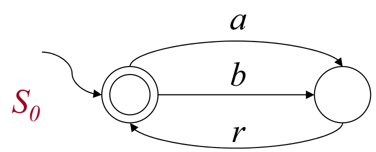
- This labeled execution tree has a finite representation as a finite TS …
- …with transitions labeled by an action and the service performing the action



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

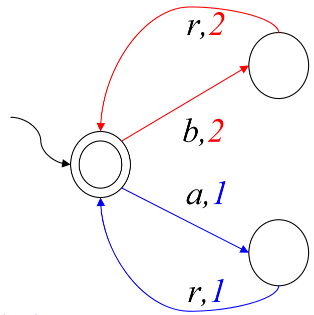
TS for Services and TS for Composition

Finite TS for services



- Deterministic
- Transitions labeled by actions
- Output on state to signal when final

Finite TS for composition



- Deterministic
- Transitions labeled by actions and services
- Output on transition to signal which service

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 we exploit 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

Structure of the PDL Encoding

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

Initial states of all services

PDL encoding of target service

PDL encoding of *i*-th component service

PDL additional domain-independent conditions

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

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

Results

Thm: Composition exists iff PDL formula Φ SAT

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

Information on the labeling is encoded in predicates $moved_i$

$\Rightarrow$ Composition existence of services expressible as finite TS is decidable in EXPTIME

Results on TS Composition

Thm: If composition exists then finite TS composition exists.

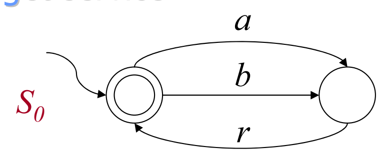
From a <u>small model</u> of the PDL formula Φ, one can build a finite TS machine

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

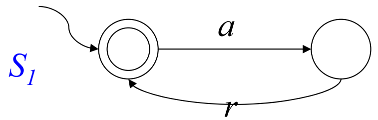
$\Rightarrow$ <u>finite TS</u> composition existence of services expressible as finite TS is decidable in EXPTIME

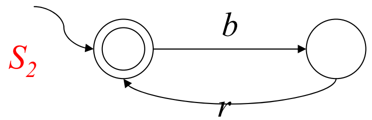
Example (1)

Target service



Available services





DPDL

...

...

...

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

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

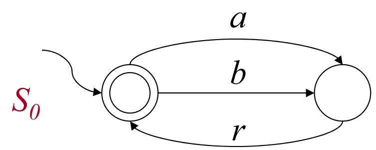
$\langle b\rangle T \rightarrow [b] (moved_1 \vee moved_2)$

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

$F_0 \rightarrow F_1 \wedge F_2$

Example (2)

Target service



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

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

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

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

$s_0^0 \rightarrow [r] \perp$

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

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

$F_0 \equiv s_0^0$

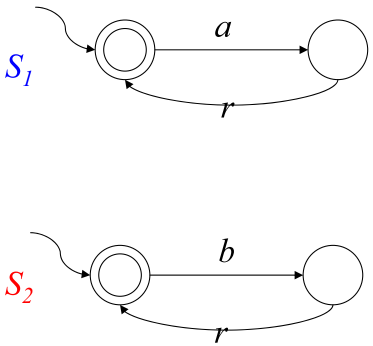
...

...

...

Example (3)

Available services



...

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

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

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

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

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

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

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

$F_1 \equiv s_1^0$

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

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

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

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

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

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

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

$F_2 \equiv s_2^0$

...

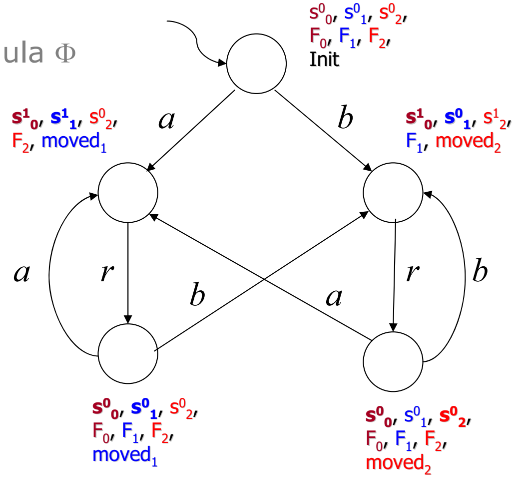
Example (4)

Check: run SAT on PDL formula Φ

Example

Check: run SAT on PDL formula Φ

Yes $\Rightarrow$ (small) model

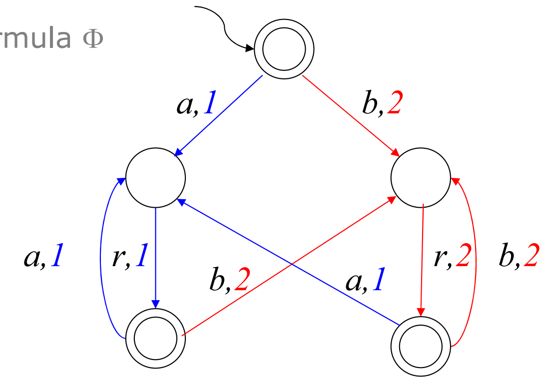


Example

Check: run SAT on PDL formula Φ

Yes $\Rightarrow$ (small) model

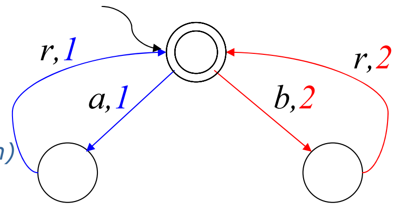
$\Rightarrow$ extract finite TS



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 …
- … when the ability or returning models will be added …
- … meanwhile we have developed a prototype tool on this idea (see ESC – E-Service Composer: **`http://sourceforge.net/projects/paride`**)

END

Composition by Simulation

Simulation

- A binary relation *R* is a **simulation** iff:

  $(s,t) \in R$ implies that
  - if s is *final* then t is *final*
  - for all actions a
    - if $s \to_a s'$ then $\exists t' . t \to_a t'$ and $(s',t') \in R$

  Note it is a co-inductive definition!
  Essentially is one direction of the bisimulation!

- A transition system S is **simulates** a transition system T iff there **exists** a **simulation** between the initial states s_0 and t_0.

Potential behavior of the community

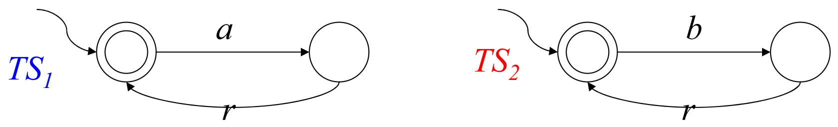
- Let $TS_1, \cdots TS_n$ be the TSs of the component services.
- The Community Big TS is defined as $TS_c = < A, S_c, S_c^0, \delta_c, F_c >$ where:
  - A is the set of actions
  - $S_c = S_1 \times \cdots \times S_n$
  - $S_c^0 = \{(s^0_1, \cdots, s^0_m)\}$
  - $F \subseteq F_1 \times \cdots \times F_n$
  - $\delta_c \subseteq S_c \times A \times S_c$ is defined as follows:
    $(s_1 \times \cdots \times s_n) \to_a (s'_1 \times \cdots \times s'_n)$ iff
    1. $\exists$ i. $s_i \to_a s'_i \in \delta_i$
    2. $\forall$ j. $s_j \to_a s'_j \in \delta_j \vee s'_j = s_j$

Composition by Simulation

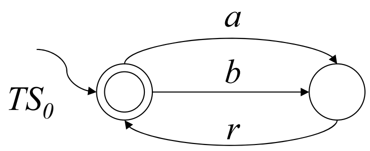
- **Thm**: A composition exists iff there exists a simulation from (the initial state of) TS_c to (the initial state of) TS_t
- Given a simulation R form TS_c to TS_t (which include the initial states), we can build a finite composition as follows:
  $TS = < A_r, S_r, S_r^0, \delta_r, F_r >$ with
  - $A_r = A \times 2^{[n]}$
  - $S_c = S_1 \times \cdots \times S_n \times S_t$
  - $S_c^0 = \{ (s^0_1, \cdots, s^0_m, s^0_t) \}$
  - $F \subseteq \{ (s_1 \times \cdots \times s_n \times s) \mid s \in F_t$
  - $\delta_r \subseteq S_r \times A_r \times S_r$ is defined as follows:
    $(s_1 \times \cdots \times s_n, s) \to_{a,I} (s'_1 \times \cdots \times s'_n, s')$ iff
    1. $s \to_{a,} s'$
    2. $(s_1 \times \cdots \times s_n) \to_{a,I} (s'_1 \times \cdots \times s'_n)$
    3. $((s'_1 \times \cdots \times s'_n), s) \in R$
    4. $I = (j \mid s_j \to_a s'_j \in \delta_j)$

Example of Composition

- Available Services



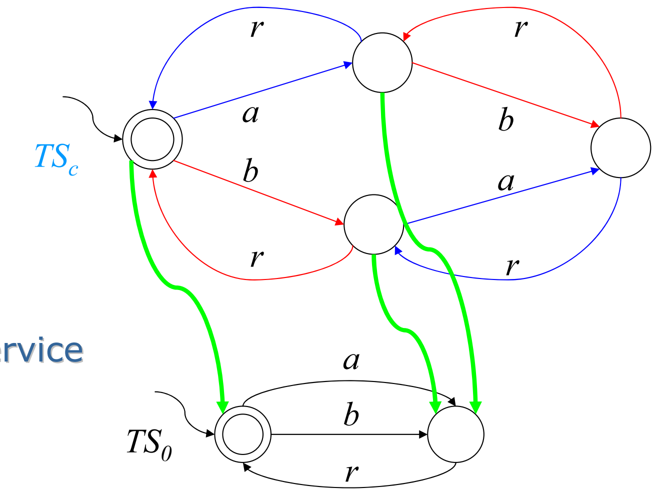
- Target Service



Example of Composition

Community Big Service

Target Service



Composition exists!