Web service composition via TLV

Seminari di Ingegneria del SW
Fabio Patrizi
DIS, Sapienza – Università di Roma

Essential overview

- Computing composition via simulation
- Using TLV for computing composition via simulation

The Problem

Given:

- a community of available services

$$\mathcal{C} = \{S_1,\ldots,S_n\};$$

- a target service

$$T;$$

Find a *composition* (or *orchestrator*) s.t.

$$\mathcal{C} \text{ mimicks } T$$

The Problem (cont.)

We model services as transition systems:

Finding a composition

Strategies for computing compositions:

- Reducion to PDL
- Simulation-based 

Simulation Relation

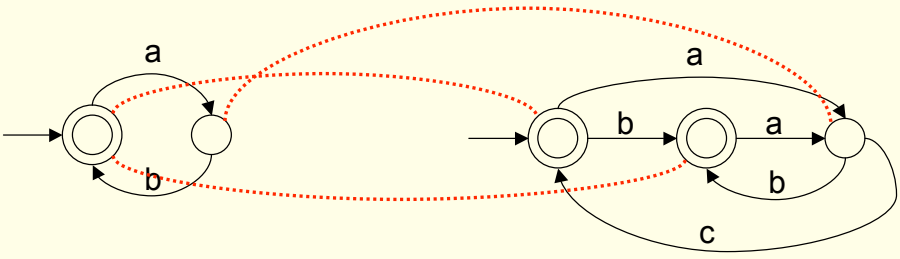
Intuition:

a service S can simulate T if it can reproduce T's behavior over time.

Simulation Relation (cont.)

Simulation Relation (cont.)

T: :



Can $\mathcal{C}$ simulate T?

YES!

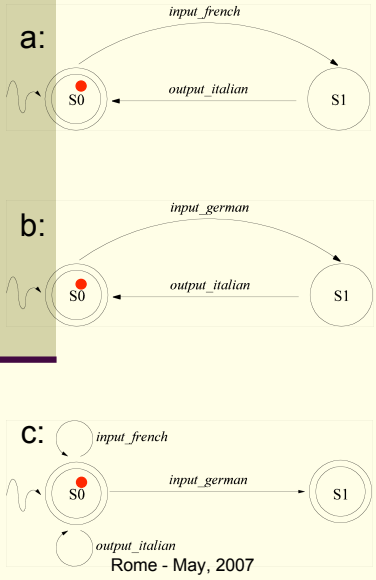
Computing composition via simulation

Idea:

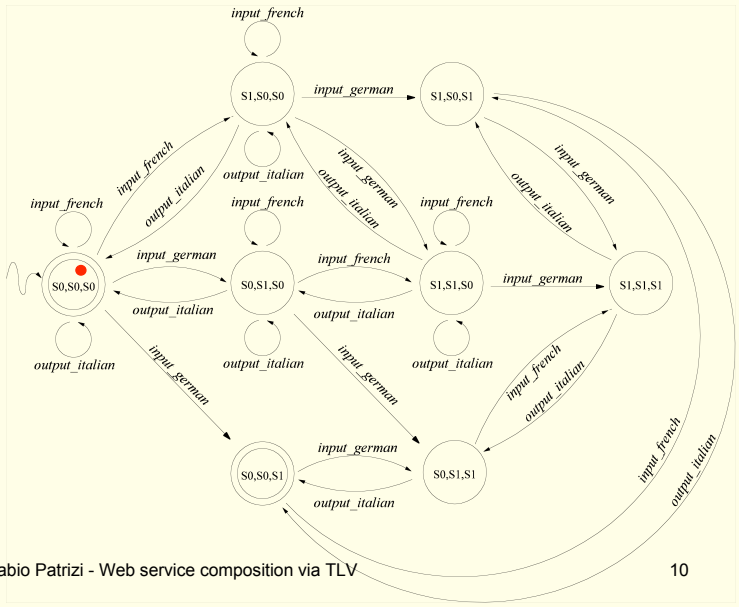
A service community can be seen as the (possibly N-DET) *asynchronous product* of available services…

Computing composition via simulation (cont.)

Available services



Community TS



Computing composition via simulation (cont.)

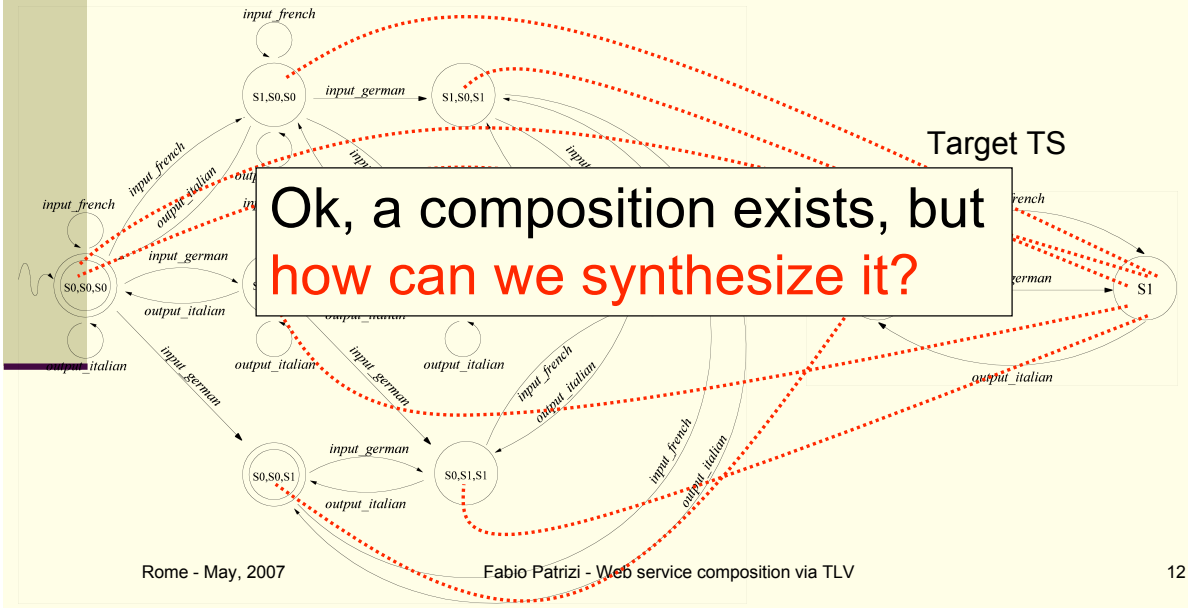
Idea:

Theorem:
A composition exists if and only if $\mathcal{C}$ simulates T

… thus, the problem becomes:
"Can the community TS $\mathcal{C}$ simulate target service T?"

Computing composition via simulation (cont.)

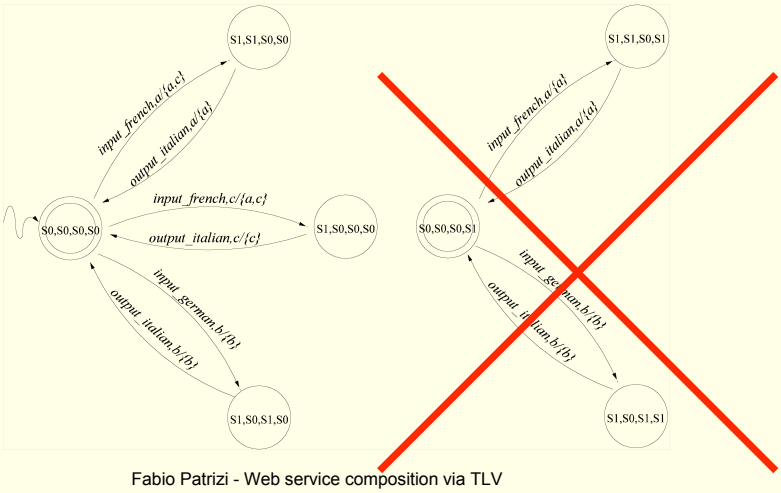
Community TS



Ok, a composition exists, but
how can we synthesize it?

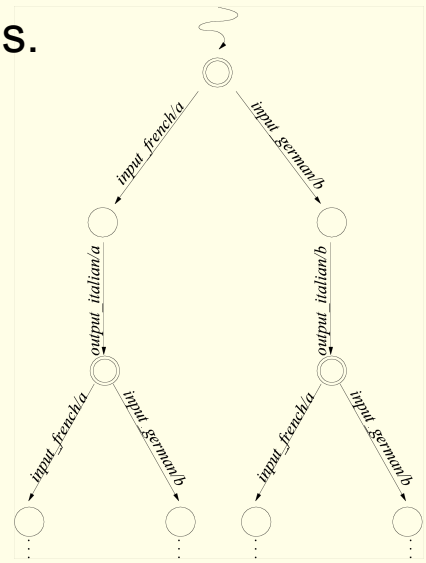
Computing composition via simulation (cont.)

- From the maximal simulation, we can easily derive an *orchestrator generator*, *e.g.:*



Computing composition via simulation (cont.)

From OG, one can select services to perform client actions.



Comments

- *Full observability* is crucial for OG to work properly. In fact, in order to propose services for action execution, state of each available service *needs* to be known.
- This technique is well-suited for deterministic target and available services.
- Interesting extension: dealing with nondeterministic (devilish) available services (a slightly different notion of simulation is needed).

Such points are object of current/future work.

Computing composition via simulation (cont.)

Summing up:

- Compute community TS $\mathcal{C}$;
- Compute the maximal simulation of T by $\mathcal{C}$;
-
  - If simulation exists, compute OG;
  - else return “unrealizable”;
- Exploit OG for available service selection, even in a *just-in-time* fashion.

Essential overview (2)

- Computing composition via simulation
  - Any questions?
- Using TLV for computing composition via simulation

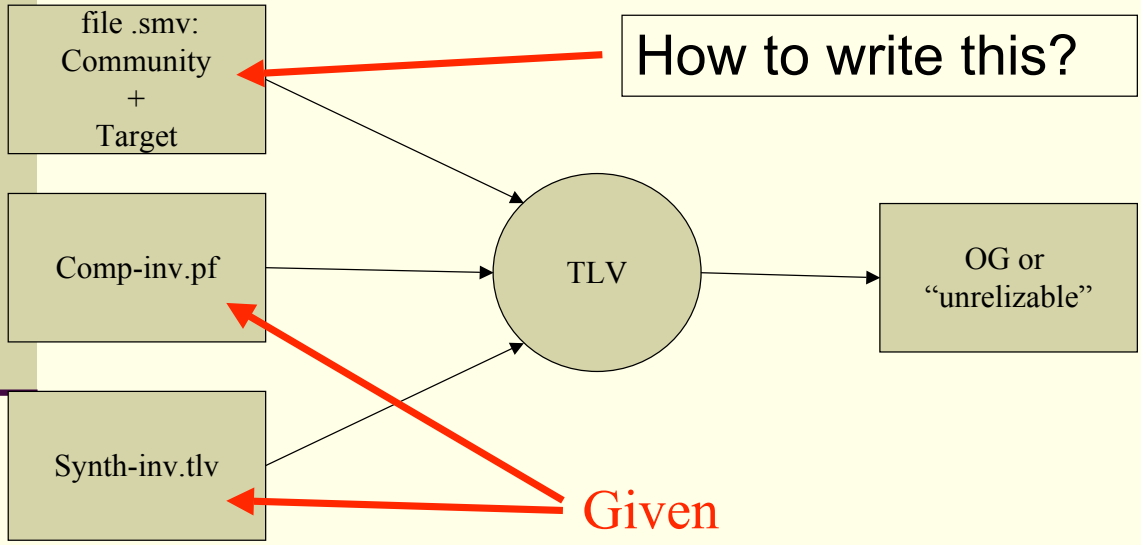
Composing services via TLV

The environment TLV (Temporal Logic Verifier) [Pnueli and Shahar, 1996] is a useful tool that can be used to

automatically compute the orchestrator generator,

given a problem instance.

Composing services via TLV (cont.)



Composing services via TLV (cont.)

We provide TLV a file written in (a flavour of) SMV, a language for specifying TSs.

- SMV specifications are tipically composed of *modules, properly interconnected;*
- Intuitively, a module is a *sort of TS* which may share variables with other modules;
- A module may contain several submodules, properly synchronized;
- Module main is mandatory and contains all relevant modules, properly interconnected and synchronized.

Composing services via TLV (cont.)

A module:

name

parameter(s)

initialization

internal variable(s)

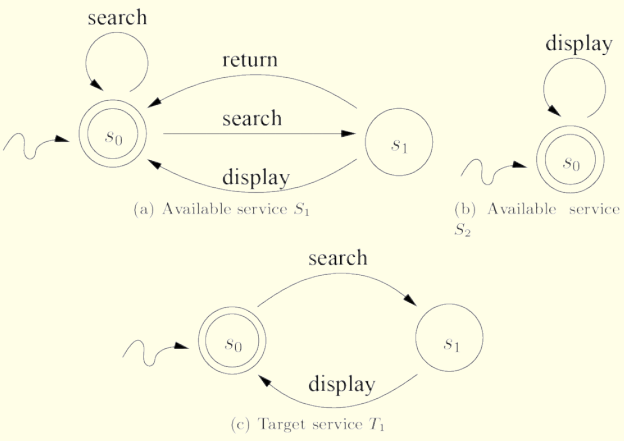
transition relation

```
MODULE mT1(act)
VAR
  loc : 0..1;
ASSIGN
  init(loc) := 0;
  init(act) := nil;
  next(loc) :=
              case
                        loc = 0 & act = search  : 1;
                        loc = 1 & act = display : 0;
                        TRUE                    : loc;
              esac;
  next(act) :=
              case
                        act = nil               : {search};
                        loc = 0 & act = search  : {display};
                        loc = 1 & act = display : {search};
                        TRUE                    : {act};
              esac;
DEFINE
        final := (loc = 0);
```

boolean expression

Composing services via TLV (cont.)

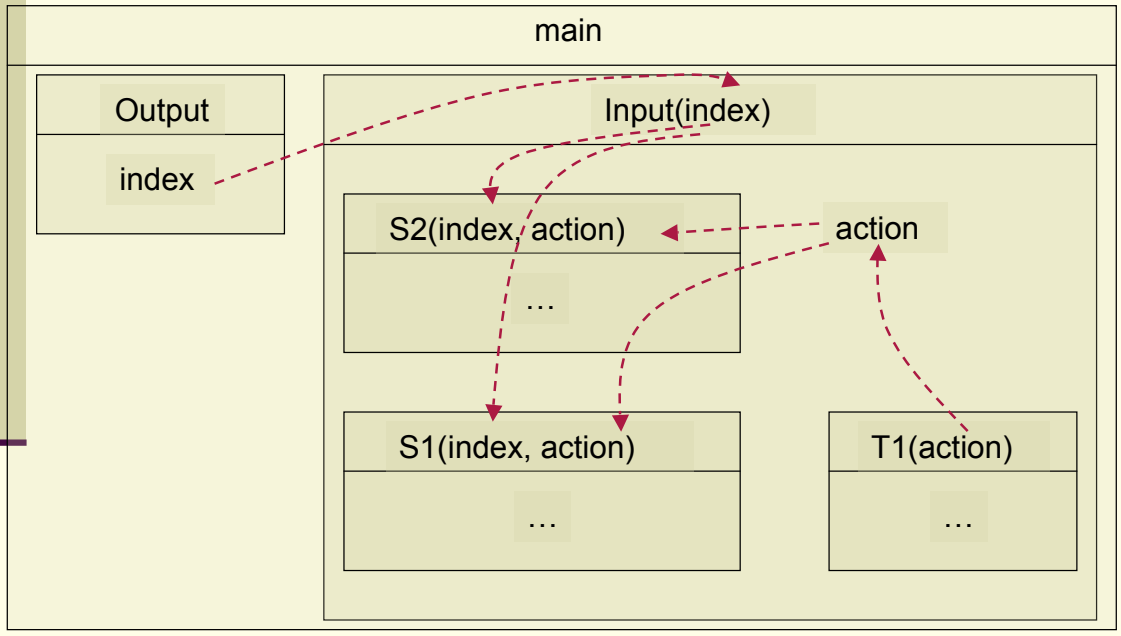
We introduce SMV formalization by means of the following example, proceeding top-down:



Composing services via TLV (cont.)

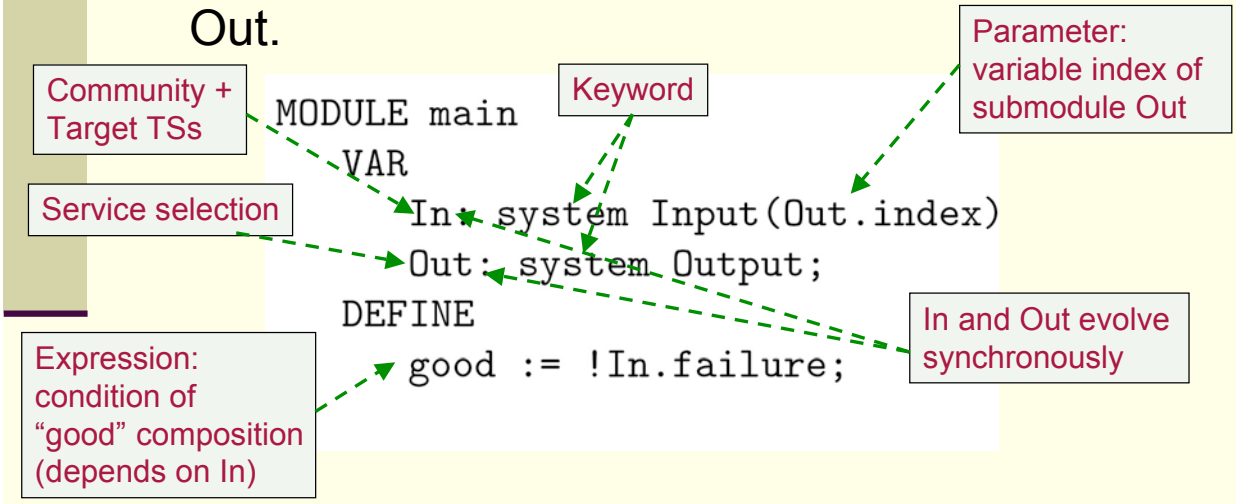
- The application is structured as follows:
  - 1 module main
  - 1 module Output, representing OG service selection
  - 1 module Input, representing the (synchronous) interaction community-target
  - 1 module mT1 representing the target service
  - 1 module mSi per available service

Module interconnections



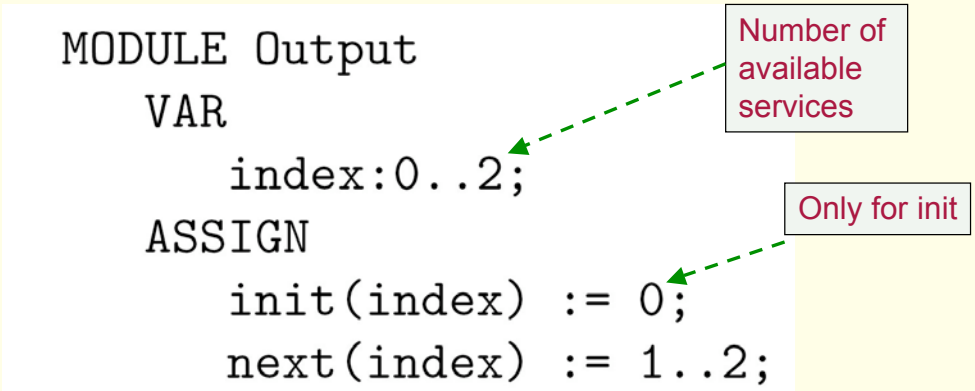
The module main

- Instance independent
- Includes synchronous submodules In and Out.



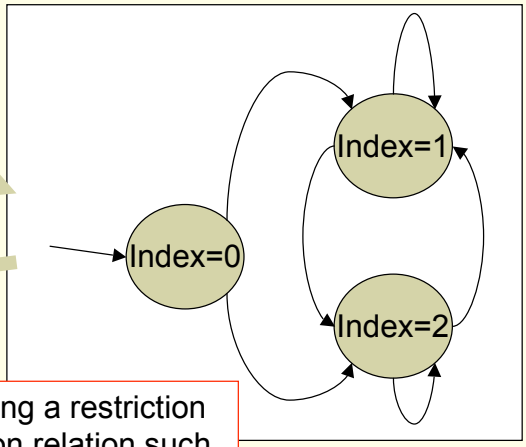
The module Output

- Depends on number of available services. In this case: 2



The module Output (cont.)

```
MODULE Output
   VAR
      index:0..2;
   ASSIGN
      init(index) := 0;
      next(index) := 1..2;
```




```
MODULE main
   VAR
      In: system Input(Out.index)
      Out: system Output;
   DEFINE
      good := !In.failure;
```

Synchronized

The goal is computing a restriction on Output's transition relation such that good is satisfied. RECALL that In is affected by Out through parameter Out.index

The module Input

Action alphabet + special action nil (used for init)

Target service

Available service 1

Available service 2

```
MODULE Input(index)
VAR
   action : {nil,search,display,return};
   T1 : mT1(action);
   S1 : mS1(index,action);
   S2 : mS2(index,action);
DEFINE
   failure := (S1.failure | S2.failure) |
              !(T1.final -> (S1.final & S2.final));
```

Fail if:
- S1 or S2 (… or SN) fail, OR
- T1 can be in a final state when S1 or S2 (… or SN) are not.

The target module mT1

- Think of mT1 as an action producer

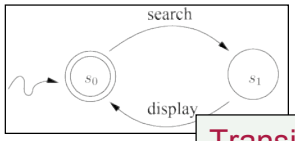
TS States

Init

Output relation (non-deterministic, in general)

Transition function (deterministic, in general)

State 0 is final




```
MODULE mT1(act)
VAR
  loc : 0..1;
ASSIGN
  init(loc) := 0;
  init(act) := nil;
  next(loc) :=
              case
                        loc = 0 & act = search  : 1;
                        loc = 1 & act = display : 0;
                        TRUE                    : loc;
              esac;
  next(act) :=
              case
                        act = nil               : {search};
                        loc = 0 & act = search  : {display};
                        loc = 1 & act = display : {search};
                        TRUE                    : {act};
              esac;
DEFINE
        final := (loc = 0);
```


The target module mT1 (cont.)

1. A statement of the form:

```
next(loc):=
   case
      case_1;
      ...
      case_n;
      TRUE  :  loc;
   esac;
```

is included for defining next `loc` value. Each `case_`i expression refers to a different pair $< s, a > \in S_t \times A_t$ such that $\delta_t(s, a)$ is defined (order does not matter) and assumes the form:

$$\texttt{loc} = ind(s)\ \texttt{\&}\ \texttt{act} = a : \delta_t(s, a)$$

2. A statement of the form:

```
next(act):=
   case
      case_0;
      case_1;
      ...
      case_n;
      TRUE  :  act;
   esac;
```

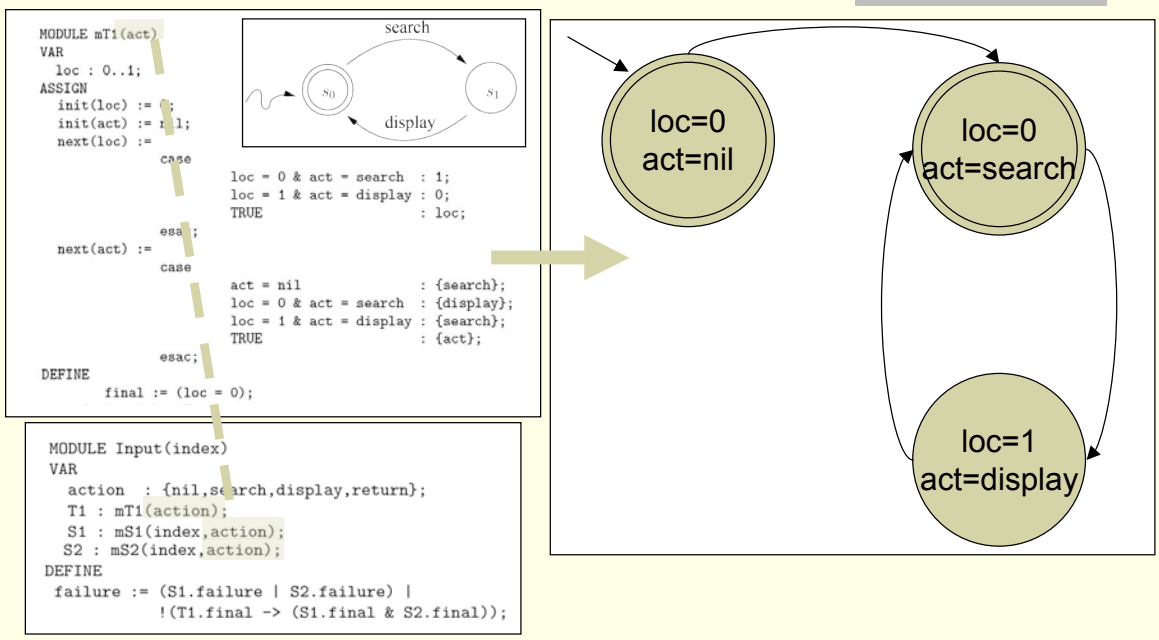
is included for defining next `act` assignment. Let $act : S_t \rightarrow 2^{A_t}$ be defined as $act(s) = \{a \in A_t \mid \exists\ s' \in S_t\ s.t.\ s' = \delta_t(s, a)\}$. Then, `case_0` assumes the form:

$$\texttt{act} = \texttt{nil} : act(s_0)$$

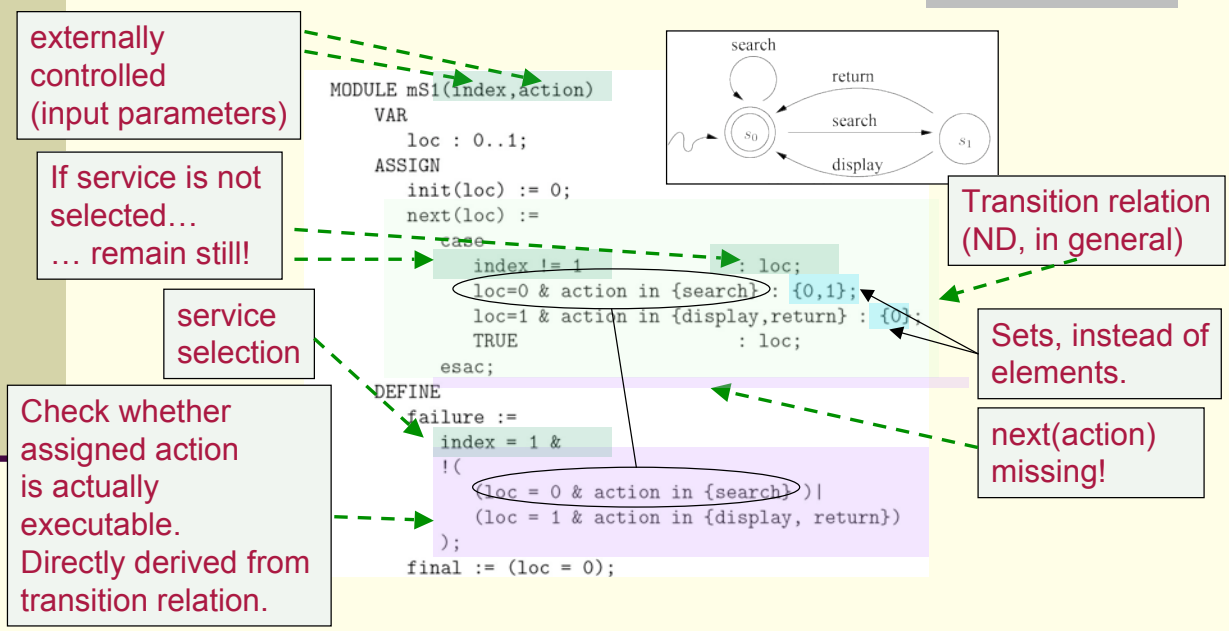
For $i > 0$, each `case_`i expression refers to a different pair $< s, a > \in S_t \times A_t$ such that $act(\delta_t(s, a)) \neq \emptyset$ (order does not matter) and assumes the form:

$$\texttt{loc} = ind(s)\ \texttt{\&}\ \texttt{act} = a : act(\delta_t(s, a))$$

The target module mT1 (cont.)

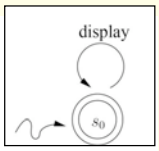


The available service module mS1



The available service module mS2

```
MODULE mS2(index,action)
  DEFINE
     failure :=
        index = 2 & !(action in {display});
        final := TRUE;
```



Stateless system: neither states nor transition relation needed

Putting things together

```
MODULE main
  VAR
     In: system Input(Out.index);
     Out: system Output;
  DEFINE
     good := !In.failure;

MODULE Output
  VAR
     index:0..2;
  ASSIGN
     init(index) := 0;
     next(index) := 1..2;
```

Never changes

Number of available services

Putting things together (cont.)

```
MODULE Input(index)
  VAR
    action : {nil,search,display,return};
    T1 : mT1(action);
    S1 : mS1(index,action);
    S2 : mS2(index,action);
  DEFINE
    failure := (S1.failure | S2.failure) |
          !(T1.final -> (S1.final & S2.final));
```

Whole shared action alphabet plus special action nil

Never changes

Index changes, add one module per available service

Index changes, add one conjunct/disjunct per available service

Putting things together (cont.)

```
MODULE mT1(act)
  VAR
    loc : 0..1;
  ASSIGN
    init(loc) := 0;
    init(act) := nil;
    next(loc) :=
      case
        loc = 0 & act = search : 1;
        loc = 1 & act = display : 0;
        TRUE : loc;
      esac;
    next(act) :=
      case
        act = nil : {search};
        loc = 0 & act = search : {display};
        loc = 1 & act = display : {search};
        TRUE : {act};
      esac;
  DEFINE
    final := (loc = 0);
```

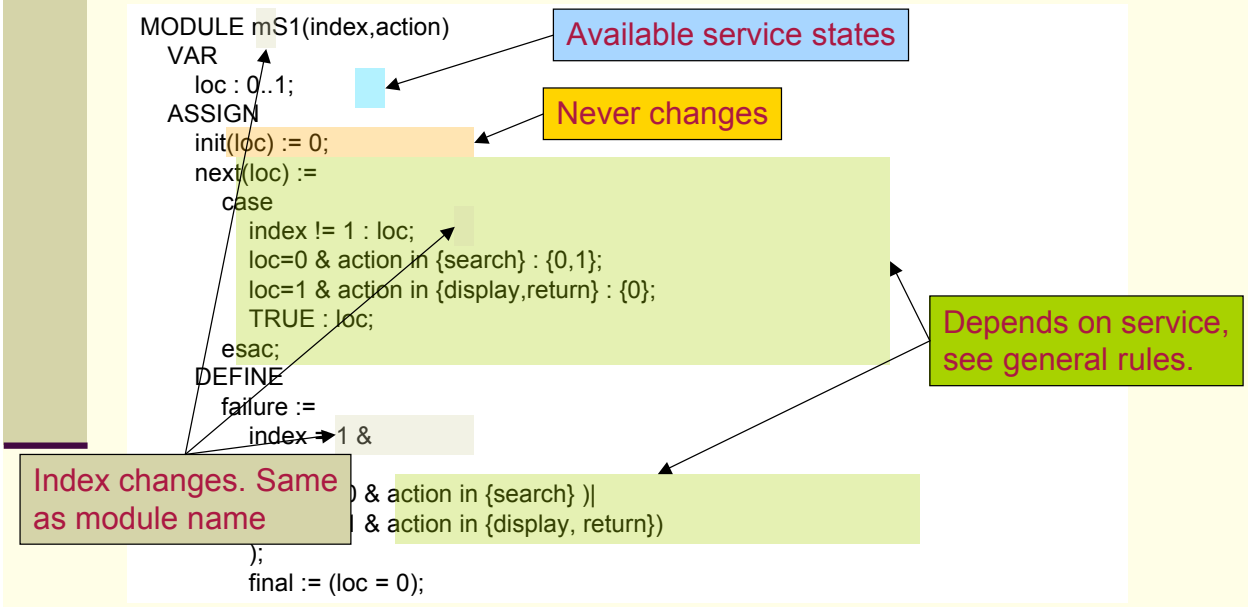
Target service states

Never changes

Depends on service, see general rules.

List final states using either logical OR '|' (e.g., (loc=0|loc=1|loc=3)) or set construction (e.g., (loc={0,1,3})).

Putting things together (cont.)

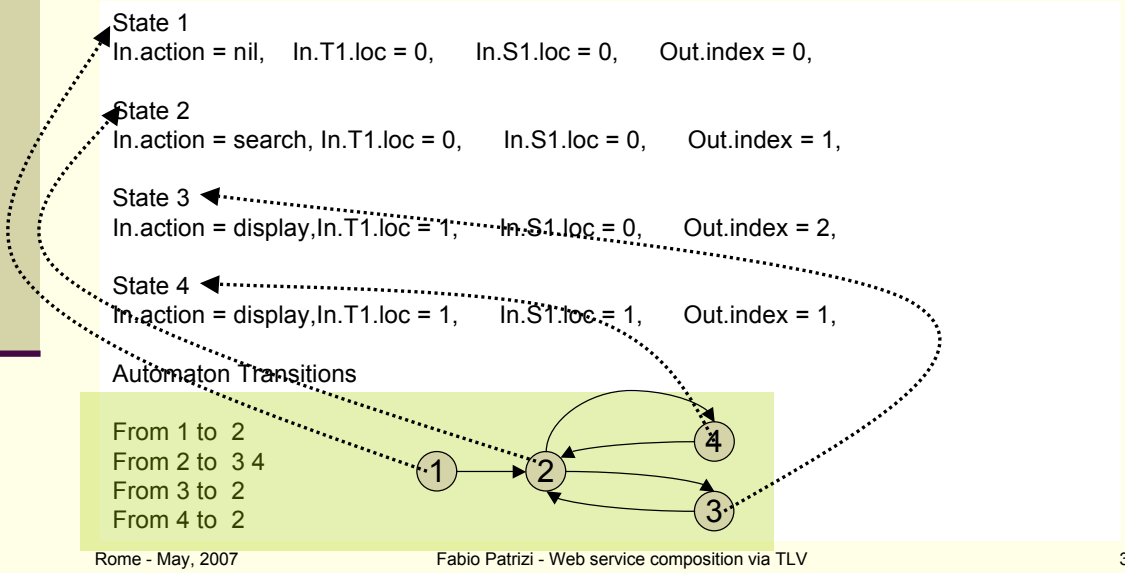


Putting things together (cont.)

```
MODULE mS2(index,action)
   DEFINE
      failure :=
         index = 2 & !(action in {display});
      final := TRUE;
```

Running the specification

Running TLV with our specification as input…



Running the specification (cont.)

That is, the following OG: