Service composition via simulation

Seminari di Ingegneria del SW
Slides by Fabio Patrizi and Giuseppe De Giacomo
DIS, Sapienza – Università di Roma

Essential overview

- Computing composition via simulation
- Using an LTL synthesis tool, 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.

$\mathcal{C}$ mimics T

The Problem (cont.)

We model services as transition systems:

- A TS is a tuple T = < A, S, s_0, δ , F> where:
  - A is the set of actions
  - S is the set of states
  - $s_0 \in S$ 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

Finding a composition

Strategies for computing compositions:

- Reduction to PDL
- Simulation-based ←

Simulation Relation

Intuition:

a service $\mathcal{T}$ can be simulated by community $\mathcal{C}$ if $\mathcal{C}$ can reproduce $\mathcal{T}$'s behavior over time.

Simulation Relation (cont.)

- Given two transition systems $\mathcal{T} = < A, T, t^0, \delta_T, F_T >$ and $\mathcal{C} = < A, S, s_{\mathcal{C}}^0, \delta_{\mathcal{C}}, F_{\mathcal{C}} >$ a **simulation** relation on $T \times \mathcal{C}$ is a binary relation on the states $t \in \mathcal{T}$ an s of $\mathcal{C}$ such that:
  - $(t,s) \in R$ implies that
    - t is *final* implies that s is *final*
    - for all actions a
      if $t \rightarrow_a t'$ then $\exists s'$. $s \rightarrow_a s'$ and $(t',s') \in R$
- If **exists a simulation** relation R (such that $(t^0, s_{\mathcal{C}}^0) \in R$, 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
- *NB1: Simulated by is a co-inductive definition!*
- *NB2: A simulation is just one of the two directions of a bisimulation*

Simulation Relation (cont.)

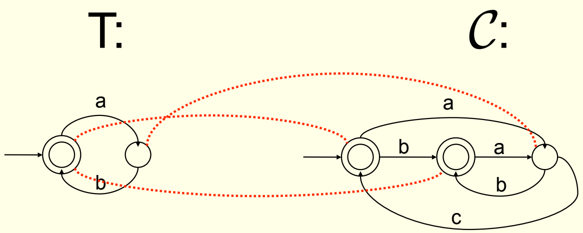
Algorithm ComputingSimulation
Input: transition system $T = <A, T, t^0, \delta_T, F_T>$ and
transition system $\mathcal{C} = <A, S, s_{\mathcal{C}}^0, \delta_{\mathcal{C}}, F_{\mathcal{C}}>$
Output: the **simulated-by** relation (the largest simulation)

Body

```
R = ∅
R' = T × S - {(t,s) | t ∈ Ft ∧ ¬(s ∈ FC)}
while (R ≠ R') {
      R := R'
      R' := R' - {(t,s) | ∃ t',a. t →a t' ∧ ¬∃ s' . s →a s' ∧ (t',s') ∈ R' }
}
return R'
```

Ydob

Simulation relation (cont.)



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

Let $S_1, \ldots, S_n$ be the TSs of the component services.

The **Community TS** $\mathcal{C} = \langle A, S_\mathcal{C}, s_\mathcal{C}^0, \delta_\mathcal{C}, F_\mathcal{C} \rangle$ is the **asynchronous product** of $S_1, \ldots, S_n$ where:

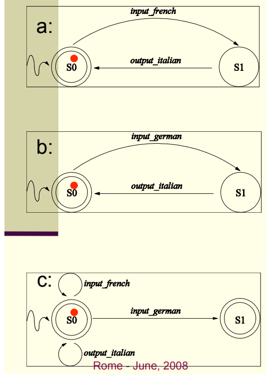
- A is the set of actions
- $S_\mathcal{C} = S_1 \times \ldots \times S_n$
- $s_\mathcal{C}^0 = (s^0_1, \ldots, s^0_m)$
- $F \subseteq F_1 \times \ldots \times F_n$
- $\delta_\mathcal{C} \subseteq S_\mathcal{C} \times A \times S_\mathcal{C}$ is defined as follows:

  $(s_1 \times \ldots \times s_n) \rightarrow_a (s'_1 \times \ldots \times s'_n)$ iff

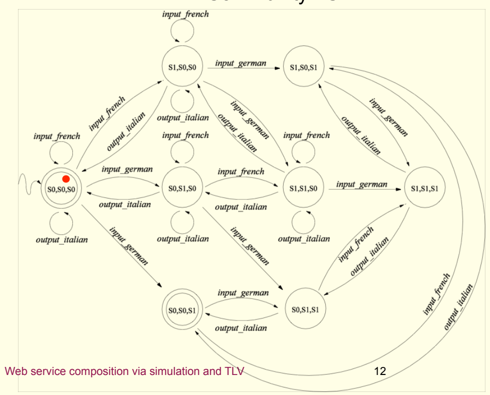
  - $\exists$ i. $s_i \rightarrow_a s'_i \in \delta_i$
  - $\forall$ j≠i. $s'_j = s_j$

Computing composition via simulation (cont.)

Available services



Community TS



Computing composition via simulation (cont.)

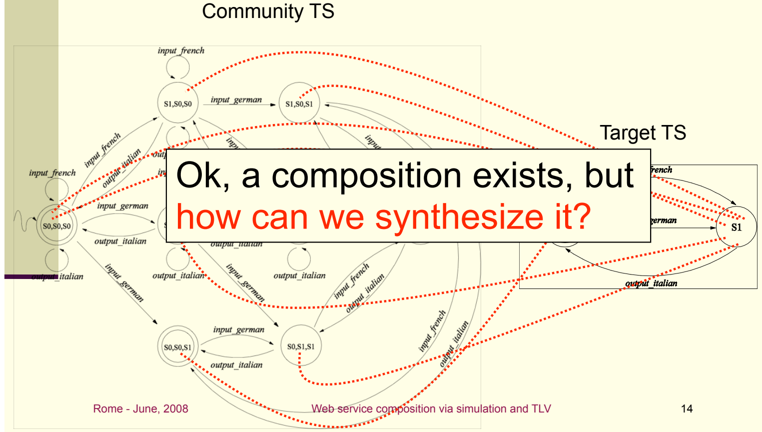
Idea:

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

including all behaviors of any feasible

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

Computing composition via simulation (cont.)



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

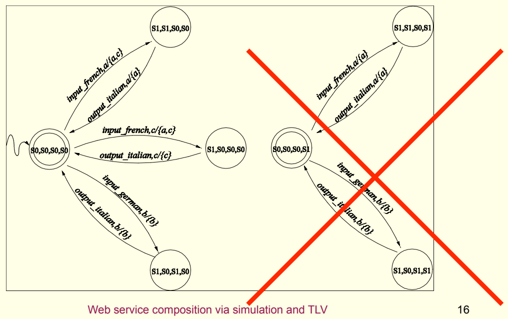
The orchestrator generator

- Given the largest simulation ***S*** form TS_t to TS_c(which include the initial states), we can build the ***orchestrator generator***.
- This is an orchestrator program that can change its behavior reacting to the information acquired at run-time.

Def: **OG** = $< A, [1,\ldots,n], S_r, s_r^0, {}_r, {}_r, F_r >$ with
- A : the **actions** shared by the community
- $[1,\ldots,n]$: the **identifiers** of the available services in the community
- $S_r = S_t \times S_1 \times \ldots \times S_n$: the **states** of the orchestrator program
- $s_r^0 = (s^0_t, s^0_1, \ldots, s^0_m)$: the **initial state** of the orchestrator program
- $F_r \subseteq \{ (s_t , s_1 , \ldots, s_n) \mid s_t \in F_t$: the **final states** of the orchestrator program
- ω_r: $S_r \times A_r \rightarrow [1,\ldots,n]$: the **service selection function**, defined as follows:
  - If $s_t \rightarrow_a s'_t$ then ***choose*** k s.t. $\exists\, s_k'.\ s_k \rightarrow_a s_k' \wedge (s_t', (s_1 , \ldots, s'_k , \ldots, s_n)) \in$ ***S***
- $\delta_r \subseteq S_r \times A_r \times [1,\ldots,n] \rightarrow S_r$: the **state transition function**, defined as follows:
  - Let $\omega_r(s_t, s_1 , \ldots, s_k , \ldots, s_n, a) = k$ then
    $(s_t, s_1 , \ldots, s_k , \ldots, s_n) \rightarrow_{a,k} (s_t', s_1 , \ldots, s'_k , \ldots, s_n)$ where $s_k \rightarrow_a s'_k$

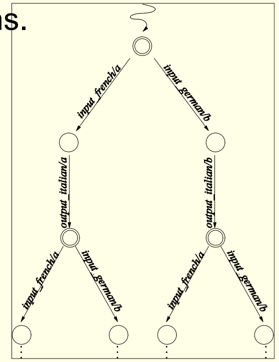
The orchestrator generator (cont.)

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



The orchestrator generator (cont.)

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



Computing composition via simulation (cont.)

Summing up:

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

On-the-fly failure recovery with OG [KR08]

OG already solves:

- **Temporary freezing** of an available service *k*
  - Stop selecting *k* in OG until service k comes back!
- **Unexpected state change** of an available service
  - Recompute OG / simulated-by from new initial state ...
  - ... but OG / simulated-by independent from initial state!
  - Simply use old OG / simulated-by from the new state!!

Parsimonious failure recovery with OG [KR08]

Algorithm ComputingSimulation - parametrized version
Input: transition system T = <A, T, t^0, δ_T, F_T> and
transition system $\mathcal{C}$= <A, S, $s_{\mathcal{C}}^0$, $\delta_{\mathcal{C}}$, $F_{\mathcal{C}}$>
relation $\mathbf{R_{init}}$ including then simulated-by
relation $\mathbf{R_{sure}}$ included then simulated-by
Output: the **simulated-by** relation (the largest simulation)

Body
R = ∅
R' = R_{init} - {(t,s) | $t \in F_t \wedge \neg(s \in F_{\mathcal{C}})$}
while (R ≠ R') {
R := R'
R' := R' - {(t,s) | $\exists t',a.\ t \rightarrow_a t' \wedge \neg\exists s'.\ s \rightarrow_a s' \wedge (t',s') \in R' \cup R_{sure}$ }
}
return $R' \cup R_{sure}$
Ydob

Parsimonious failure recovery with OG (cont.) [KR08]

Let [1,.., n] = WUF be the available services.

Let $\boldsymbol{R} = \boldsymbol{R}_{WUF}$ be the **simulated-by** relation of target by services WUF.

Then consider the following relations [KR08]:

- $\boldsymbol{R}_W \subseteq \pi_W(\boldsymbol{R}_{WUF})$
  - ($\pi_W(\boldsymbol{R})$ is not a simulation of target by services W)
  - $\pi_W(\boldsymbol{R}_{WUF})$ is the **projection on W** of a relation: easy to compute
- $\boldsymbol{R}_W \times F \subseteq \boldsymbol{R}_{WUF}$
  - ($\boldsymbol{R}_W \times F$ is a simulation of target by services WUF
  - $\boldsymbol{R}_W \times F$ is the **cartesian product** of 2 relations (F is trivial): easy to compute

Parsimonious failure recovery with OG (cont.) [KR08]

When **services F die**

compute simulated-by $\boldsymbol{R}_W$ starting $\pi_W(\boldsymbol{R}_{WUF})$!

If **dead services F** come back

compute simulated-by $\boldsymbol{R}_{WUF}$ starting $\boldsymbol{R}_W \times F$!

Remember:

- $\boldsymbol{R}_W \subseteq \pi_W(\boldsymbol{R}_{WUF})$
  - ($\pi_W(\boldsymbol{R})$ is not a simulation of target by services W)
- $\boldsymbol{R}_W \times F \subseteq \boldsymbol{R}_{WUF}$
  - ($\boldsymbol{R}_W \times F$ is a simulation of target by services WUF)

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.
- *Partial observability* possible through knowledge operator [to be done]
- Interesting extension: dealing with nondeterministic (devilish) available services (a slightly different notion of simulation is needed). [KR08]
- OG allows for failure tolerance! [KR08]

Tools for computing composition based on simulation

- Computing composition via simulation
- Use simulation computing tools for composition [to be done]
- Use LTL-based syntesis tools, like TLV, for indirectly computing composition via simulation [Patrizi PhD08]

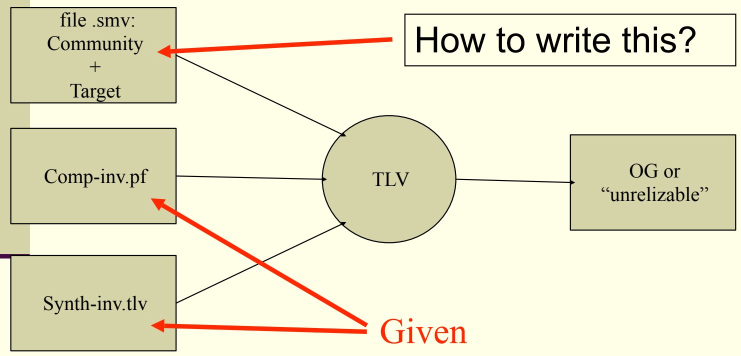
Composing services via TLV

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.)



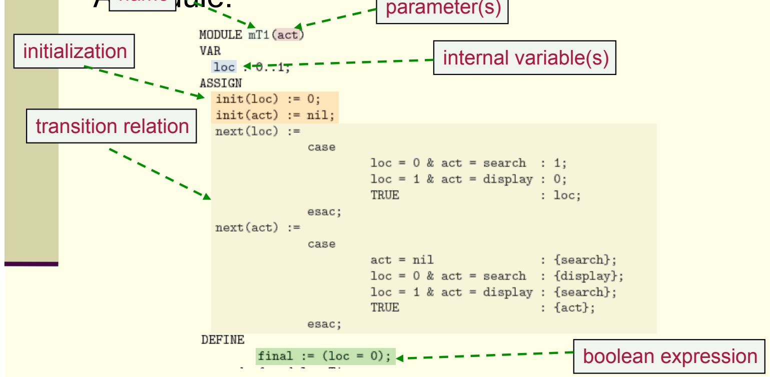
Composing services via TLV (cont.)

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

- SMV specifications are typically 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.

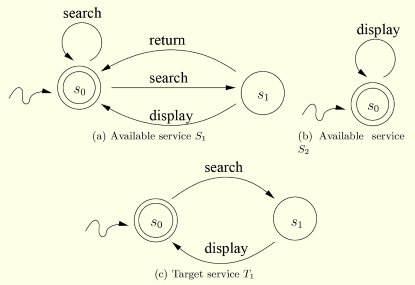
Composing services via TLV (cont.)

A dule:



Composing services via TLV (cont.)

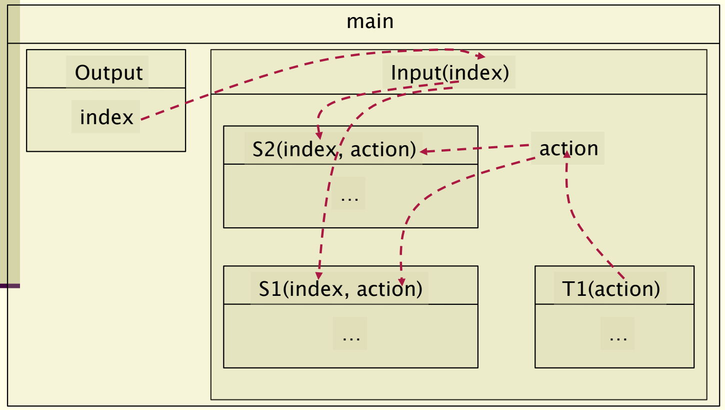
We introduce SMV formalization by means of the following example, proceeding top-down:



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.

Community + Target TSs

Keyword

Parameter: variable index of submodule Out

Service selection

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

In and Out evolve synchronously

Expression: condition of "good" composition (depends on In)

The module Output

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

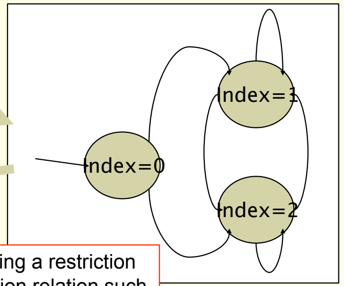
```
MODULE Output
   VAR
      index:0..2;
   ASSIGN
      init(index) := 0;
      next(index) := 1..2;
```

Number of available services

Only for init

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

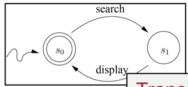
- Think of mT1 as an action producer

TS States

Init

Output relation (non-deterministic, in general)

```
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);
```



Transition function (deterministic, in general)

State 0 is final

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:

`loc = `$ind(s)$` & 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:

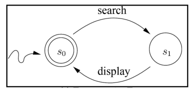
`act = nil : `$act(s_0)$

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:

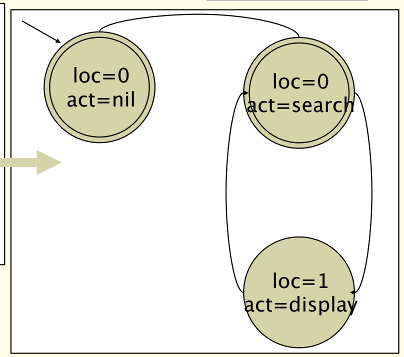
`loc = `$ind(s)$` & act = `a` : `$act(\delta_t(s, a))$

The target module mT1 (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);
```




```
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));
```



The available service module mS1

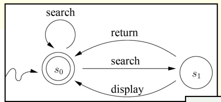
externally controlled (input parameters)

If service is not selected… … remain still!

service selection

Check whether assigned action is actually executable. Directly derived from transition relation.

```
MODULE mS1(index,action)
    VAR
      loc : 0..1;
    ASSIGN
      init(loc) := 0;
      next(loc) :=
        case
          index != 1                          : loc;
          loc=0 & action in {search}          : {0,1};
          loc=1 & action in {display,return}  : {0};
          TRUE                                : loc;
        esac;
    DEFINE
      failure :=
        index = 1 &
        !(
          (loc = 0 & action in {search}) |
          (loc = 1 & action in {display, return})
        );
      final := (loc = 0);
```



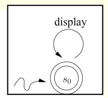
Transition relation (ND, in general)

Sets, instead of elements.

next(action) missing!

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.)

```
MODULE mS1(index,action)
  VAR
    loc : 0..1;
  ASSIGN
    init(loc) := 0;
    next(loc) :=
      case
        index != 1 : loc;
        loc=0 & action in {search} : {0,1};
        loc=1 & action in {display,return} : {0};
        TRUE : loc;
      esac;
  DEFINE
    failure :=
        index = 1 &
        = 0 & action in {search} )|
        = 1 & action in {display, return})
        );
        final := (loc = 0);
```

Available service states

Never changes

Depends on service, see general rules.

Index changes. Same as module name

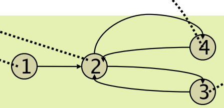
Putting things together (cont.)

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


Running the specification

Running TLV with our specification as input…

```
State 1
In.action = nil,    In.T1.loc = 0,    In.S1.loc = 0,    Out.index = 0,

State 2
In.action = search, In.T1.loc = 0,    In.S1.loc = 0,    Out.index = 1,

State 3
In.action = display,In.T1.loc = 1,    In.S1.loc = 0,    Out.index = 2,

State 4
In.action = display,In.T1.loc = 1,    In.S1.loc = 1,    Out.index = 1,

Automaton Transitions

From 1 to  2
From 2 to  3 4
From 3 to  2
From 4 to  2
```



Running the specification (cont.)

That is, the following OG: