Using TLV for Service Composition

Elective in Software and Services

Fabio Patrizi

DIS – SAPIENZA, Università di Roma

fabio.patrizi@dis.uniroma1.it

www.dis.uniroma1.it/~patrizi

Using TLV for Service Composition

1. How to represent a service composition problem instance as a safety game?
2. Using TLV to solve the safety game.

Reduction to Safety-Games

PROBLEM

INPUT: an instance of the service composition problem

- Community of available services: $\mathcal{C}=\{\mathcal{S}_1,\ldots,\mathcal{S}_n\}$
- Target service specification: $\mathcal{S}_t$

OUPUT: Safety-game G “equivalent” to above instance

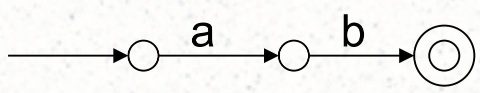
$$G = < \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box\varphi >$$

Equivalence: OG extracted from G’s WINNING set.

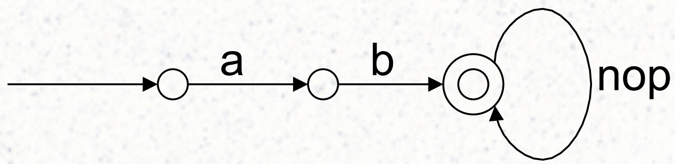
Reduction to Safety-Games (2)

Assumption: TSs have infinite runs

If not…



… do this



States always have a successor!

Reduction to Safety-Games (3)

GAME STATE VARIABLES

- $\mathcal{V}=\{s_t, s_1,\ldots, s_n,o,ind\}$
  - s_t: (over S_t) target service state
  - s_i: (over S_i) i-th service state
  - ind: (over {1,…n}) selected service
- $\mathcal{X}=\{s_t, s_1,\ldots, s_n,o\}$ (environment)
- $\mathcal{Y}=\{ind\}$ (system)

Reduction to Safety-Games (4)

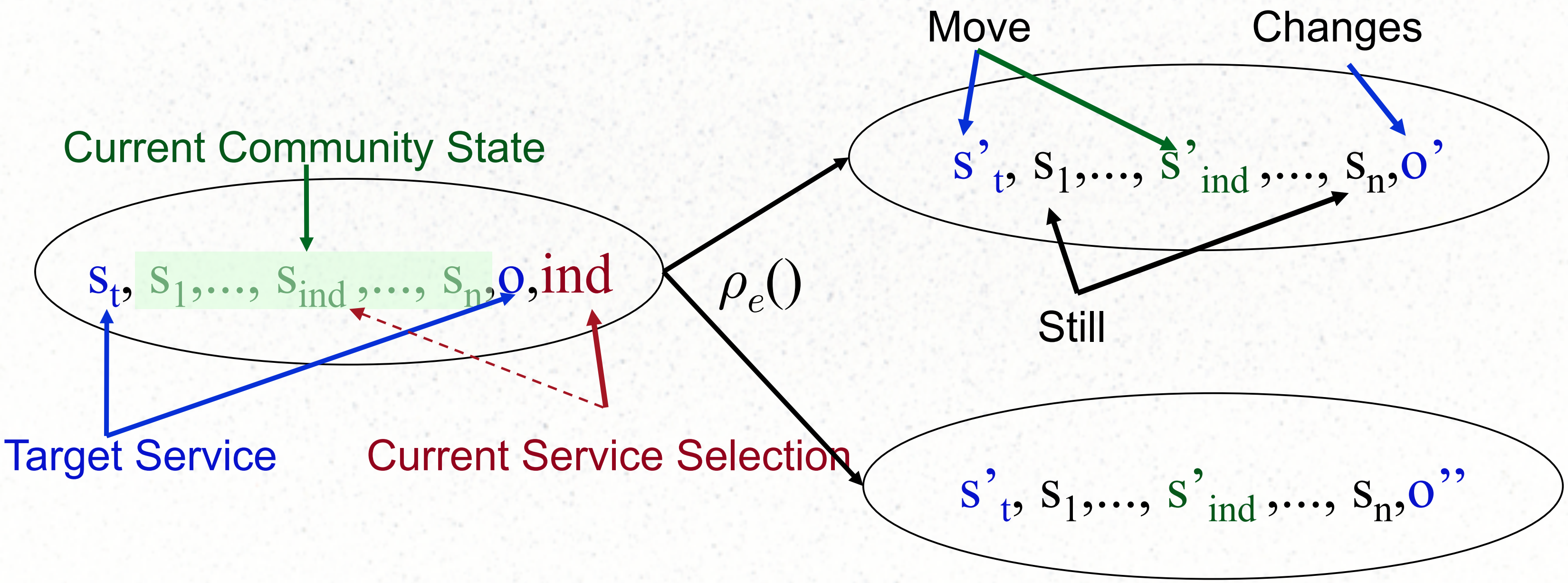
INITIALIZATION

- Θ states that all services are in their initial state
- An $\mathtt{init}$ state is introduced

Reduction to Safety-Games (5)

GAME STATE TRANSITIONS

- $\rho_e()$ defines how, given a complete current state,
  - The community changes state
  - The target service changes state and selects next op

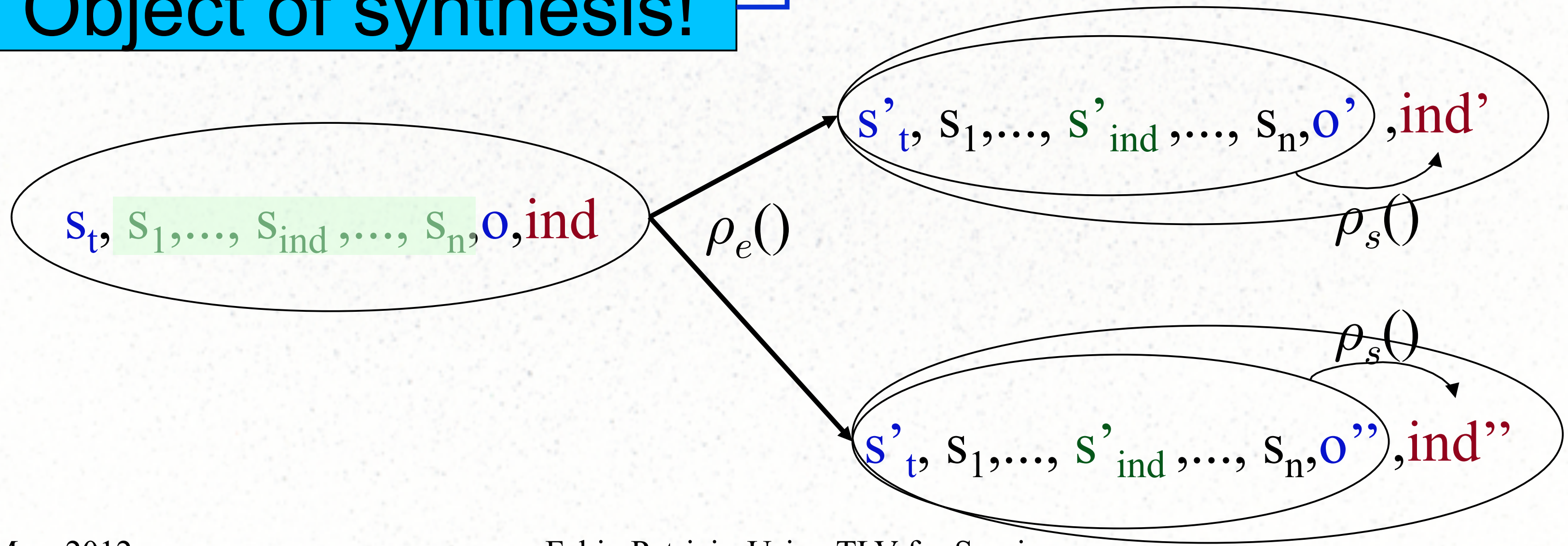


Reduction to Safety-Games (6)

GAME STATE TRANSITIONS

- $\rho_s()$ defines how, given a complete previous state and a current environment state (community + target service), the system chooses next "ind".

Object of synthesis!



Reduction to Safety-Games (7)

- $\rho_s()$ defines how, given a complete previous state and a current environment state (community + target service), the system chooses next "ind"
- $\rho_s()$ can choose any ind at each step
- The goal of synthesis is to restrict $\rho_s()$ so that the system wins the game, i.e., satisfies the invariant formula

Reduction to Safety-Games (8)

GAME INVARIANT

$$\varphi = \bigwedge_{i=1}^{n} \neg fail_i \wedge (final_t \rightarrow \bigwedge_{i=1}^{n} final_i)$$

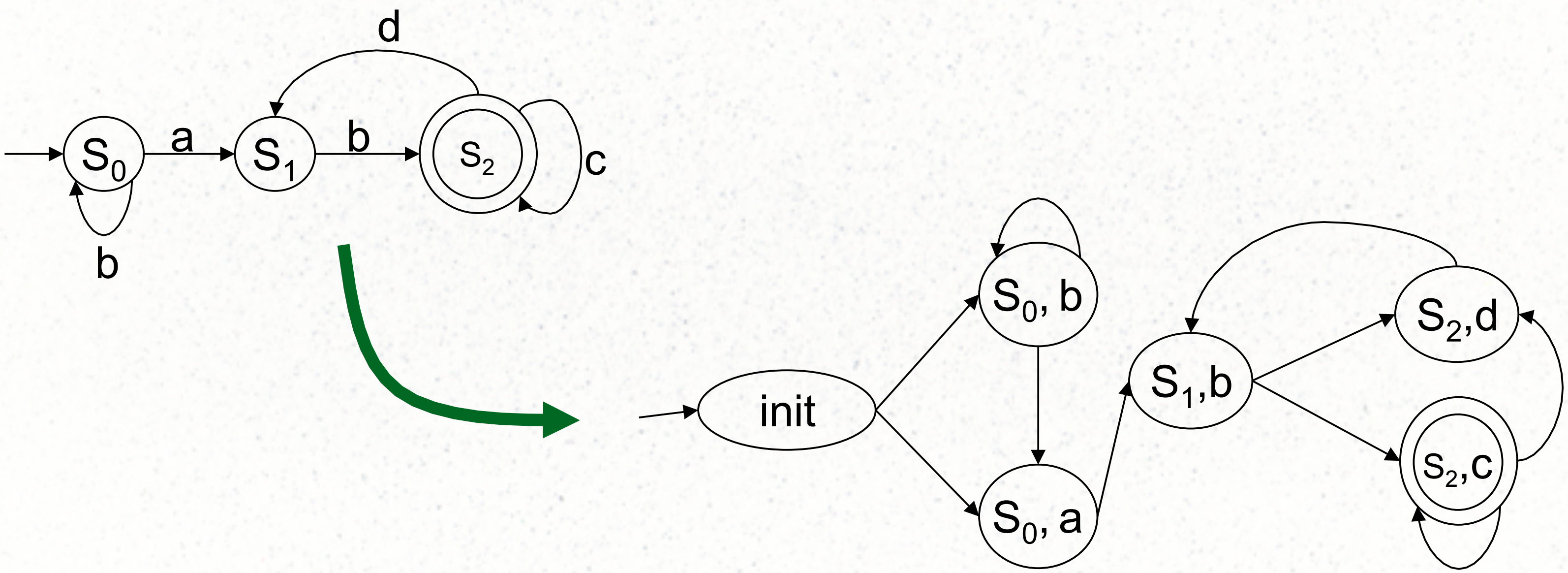
$fail_i$ holds if $\mathcal{S}_i$ is selected but is not able to perform the requested operation

If the target service is in a final state then all the other available services do

Reduction to Safety-Games (9)

GAME STATE TRANSITIONS

Observation: target operations are moved into states



Reduction to Safety-Games (11)

Once we have encoded the service composition problem in a safety-game:

Theorem:

1. A composition exists iff the maximal winning set contains all the initial game states

2. From the maximal winning set one can derive the composition generator, i.e., the set of all possible compositions

Reduction to Safety-Games (12)

"2. From the maximal winning set one can derive the composition generator, i.e., the set of all possible compositions"

Great! But…

… how to compute the maximal winning set?

Use TLV!

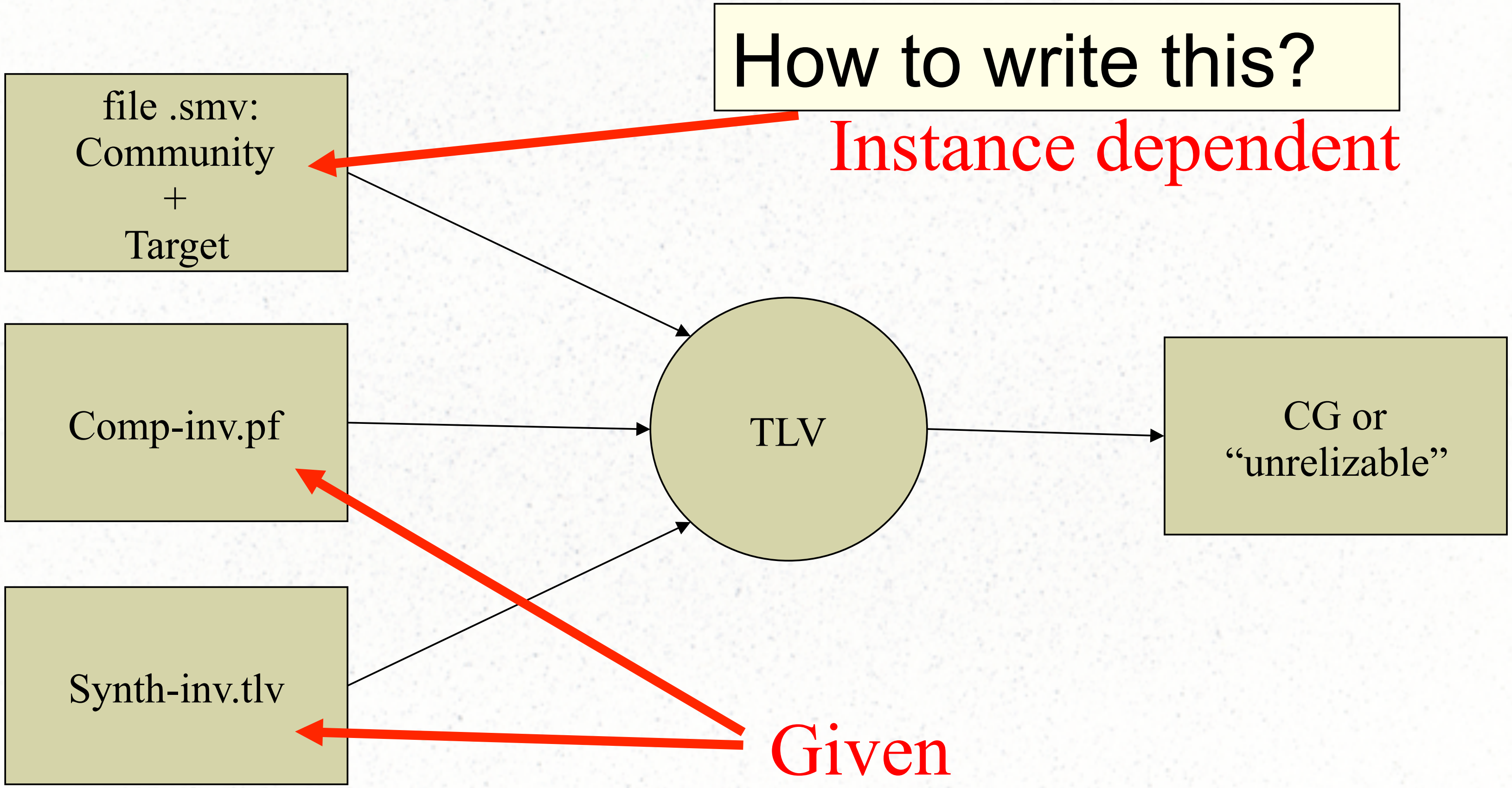
TLV

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.

TLV (2)


How to write this?
Instance dependent
file .smv:
Community
+
Target
Comp-inv.pf
Synth-inv.tlv
TLV
CG or
“unrelizable”
Given

TLV and SMV

- TLV is the software system
- SMV is the language used to write input specifications
- SMV-BASIC is the language used to write TLV algorithms

SMV Specifications

- SMV specs are composed of *modules*:
  - modules are *sorts of TS* which may share variables with other modules
  - modules may contain submodules, running in parallel
  - special module main is mandatory and contains all relevant modules

SMV Modules

A module:

name

parameter(s)

internal variable(s)

initialization

transition relation

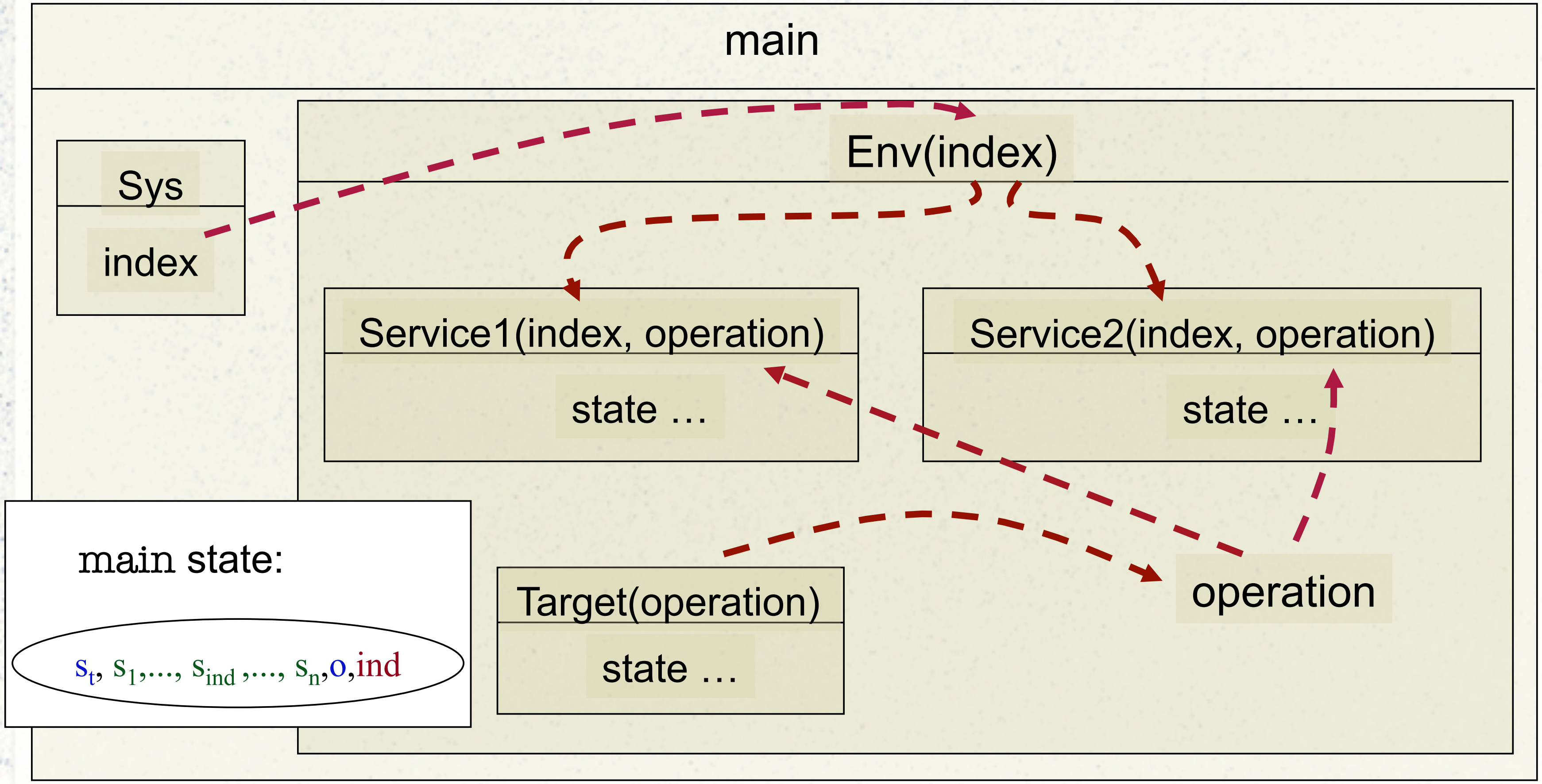
boolean expressions

```
MODULE Target(op)
VAR
    state : {start_st,t0,t1,t2};
INIT
    state = start_st & op = start_op
TRANS
    case
        state = start_st & op = start_op : next(state) = t0 & next(op) in {a};
        state = t0  & op = a : next(state) = t1 & next(op) in {b};
        state = t1  & op = b  : next(state) = t2 & next(op) in {b,c};
        state = t2  & op = b : next(state) = t2 & next(op) in {b,c};
        state = t2  & op = c: next(state) = t0 & next (op) in {a};
    esac
DEFINE
    initial := state=start_st & op=start_op;
    final := state in {t0,t2};
```

SMV specification structure

- Our specifications are structured as follows:
  - 1 module main representing the specification
  - 1 module Sys representing the orchestrator
  - 1 module Env combining $\mathcal{C}$ and $\mathcal{S}_t$
  - 1 module Target representing the target service
  - 1 module $Service_i$ per available service $\mathcal{S}_i$

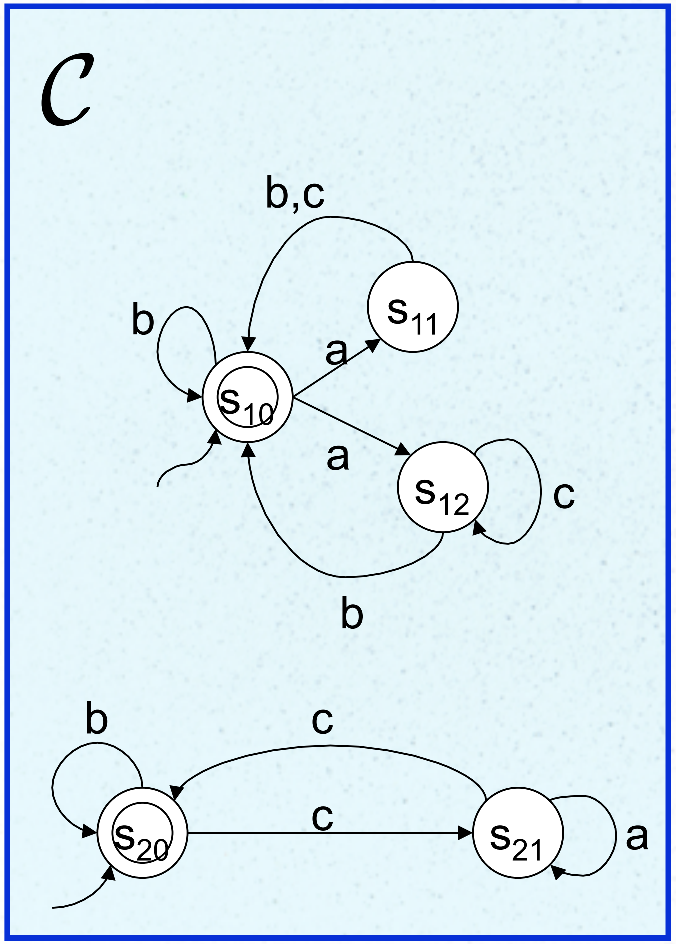
Module Interconnections

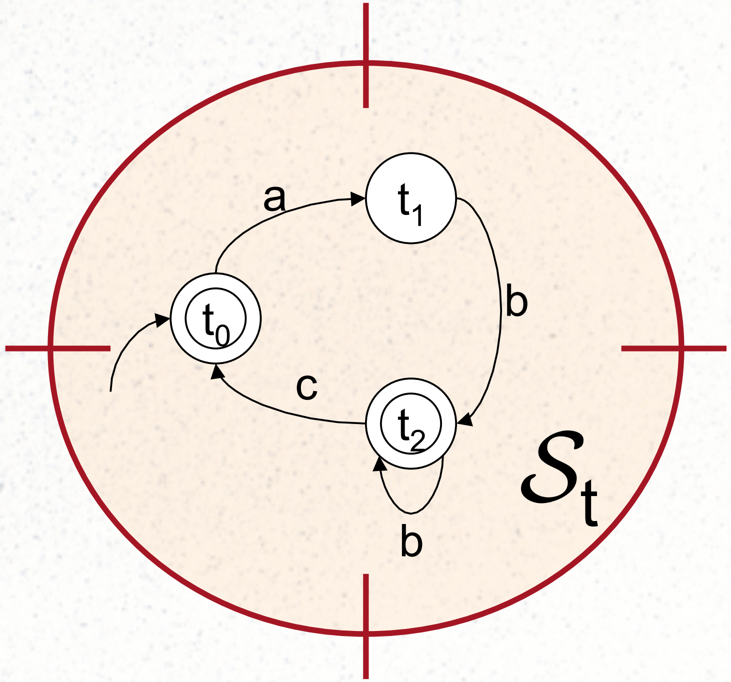


Module Transitions

- All submodules of `main` run in parallel
- At each clock tick they all move, according to their current state and specification
- We constrain non-selected modules to loop on their current state
- `main` is a (compound) transition system itself

SMV encoding by examples





Module main

- Instance independent

```
MODULE main
VAR
  env: system Env(sys.index);
  sys: system Sys;
DEFINE
  good := (sys.initial & env.initial) | !(env.failure);
```

Keywords

Parameter: variable index of submodule sys

Community + Target TSs

Orchestrator

condition of "good" state

Module Sys

- Depends on number of available services.

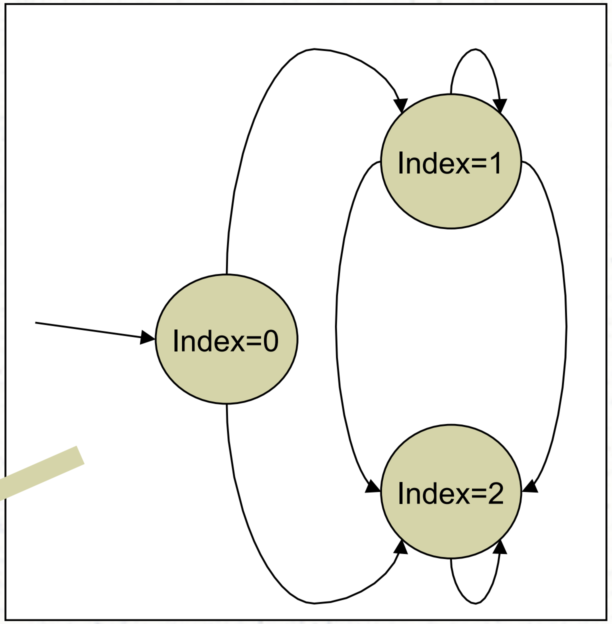
```
MODULE Sys
VAR
 index : 0..2;
INIT
  index = 0
TRANS
  case
       index=0 : next(index)!=0;
       index!=0 : next(index)!=0;
  esac
DEFINE
  initial := (index=0);
```

Number of available Services

0 used for init

Module Sys (2)

```
MODULE Sys
VAR
 index : 0..2;
INIT
  index = 0
TRANS
  case
       index=0 : next(index)!=0;
       index!=0 : next(index)!=0;
  esac
DEFINE
   initial := (index=0);
```




```
MODULE main
VAR
  env: system Env(sys.index);
  sys: system Sys;
DEFINE
  good := (sys.initial & env.initial) | !(env.failure);
```

The goal is to restrict sys transition relation so that "good" is always satisfied.
env is affected by sys through parameter sys.index

Module **Env**

Operation alphabet + special operation start_op used for init

Target service

Available services

```
MODULE Env(index)
VAR
  operation  : {start_op,a,b,c};
  target : Target(operation); -- "produces" operations
  s1 : Service1(index,operation); -- "consumes" current index and operation
  s2 : Service2(index,operation); -- same as above
DEFINE
  initial := (s1.initial & s2.initial & target.initial & operation=start_op);
  failure := (s1.failure | s2.failure) |
           (target.final & !(s1.final & s2.final));
```

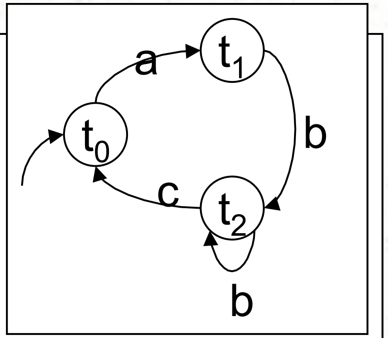
Fail if:

- S1 or S2 (… or SN) fail, OR
- T1 is in a final state when S1 or S2 (… or SN) are not.

Module Target

- Think of Target as an operation "producer"

```
MODULE Target(op) --op is an output parameter
VAR
    state : {start_st,t0,t1,t2};
INIT
    state = start_st & op = start_op
TRANS
    case
        state = start_st & op = start_op : next(state) = t0 & next(op) in {a};
        state = t0  & op = a : next(state) = t1 & next(op) in {b};
        state = t1  & op = b  : next(state) = t2 & next(op) in {b,c};
        state = t2  & op = b : next(state) = t2 & next(op) in {b,c};
        state = t2  & op = c: next(state) = t0 & next (op) in {a};
    esac
DEFINE
    initial := state=start_st & op=start_op;
    final := state in {t0,t2}; -- final state(s)
```



State

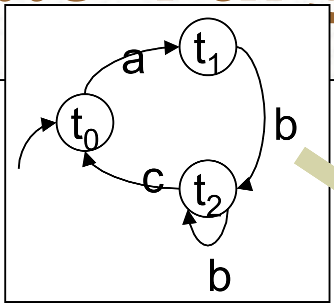
Init section

Initial/final states

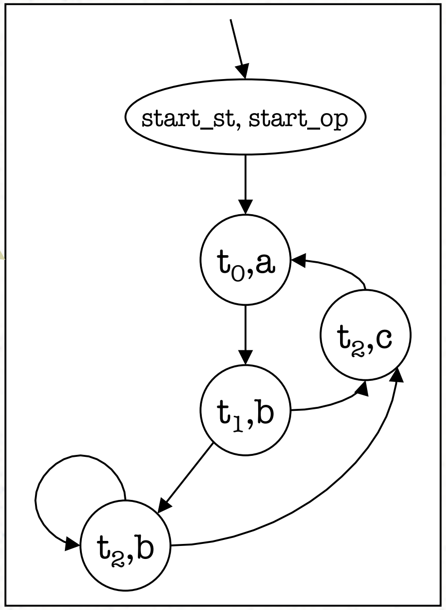
Next operation (non-deterministic)

Next state (deterministic)

Module Target (2)




```
MODULE Target(op) --op is an output parameter
VAR
        state : {start_st,t0,t1,t2};
INIT
        state = start_st & op = start_op
TRANS
        case
                state = start_st & op = start_op : next(state) = t0 & next(op) in {a};
                state = t0  & op = a : next(state) = t1 & next(op) in {b};
                state = t1  & op = b  : next(state) = t2 & next(op) in {b,c};
                state = t2  & op = b : next(state) = t2 & next(op) in {b,c};
                state = t2  & op = c : next(state) = t0 & next (op) in {a};
        esac
DEFINE
        initial := state=start_st & op=start_op;
        final := state in {t0,t2}; -- final state(s)
```




```
MODULE Env(index)
VAR
 operation  : {start_op,a,b,c};
 target : Target(operation);
 s1 : Service1(index,operation);
 s2 : Service2(index,operation);
DEFINE
 initial := (s1.initial & s2.initial & target.initial & operation=start_op);
 failure := (s1.failure | s2.failure) |
          (target.final & !(s1.final & s2.final));
```

Controlled by System

Available Service Modules

- Depend on problem instance (same as target)
- Nondeterministic in general

Available Service Modules (2)

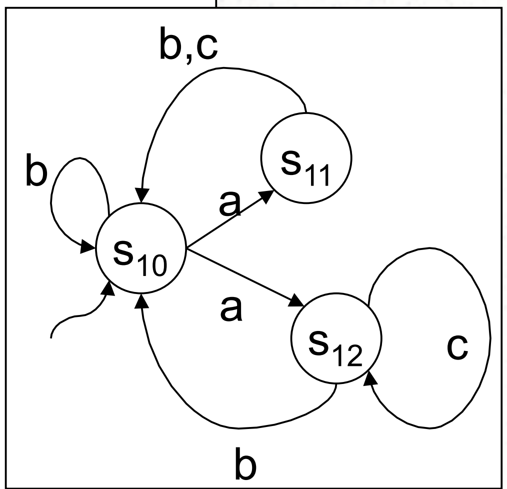
```
MODULE Service1(index,operation)
VAR
 state : {start_st,s10,s11,s12};
INIT
       state=start_st
TRANS
        case
            state=start_st & operation=start_op & index=0: next(state)=s10;
            (index != 1) : next(state) = state; -- if not selected, remain still
            (state=s10 & operation = a) : next(state) in {s11,s12};
            (state=s10 & operation = b) : next(state) in {s10};
            (state=s11 & operation=b) : next(state) in {s10};
            (state=s11 & operation=c) : next(state) in {s10};
            (state=s12 & operation=c) : next(state) in {s12};
            (state=s12 & operation=b) : next(state) in {s10};
        esac
DEFINE
        initial := state=start_st & operation=start_op & index = 0;
        failure :=
            index = 1 & !((state = s10 & operation in {a,b})|
                          (state = s11 & operation in {b,c})|
                          (state = s12 & operation in {b,c})
                     );
    final := state in {s10};
```

Initialization

ND state transition

If not selected, remain still

Fail if selected and operation not executable



Encoding summary

```
MODULE main
VAR
 env: system Env(sys.index);
 sys: system Sys;
DEFINE
 good := (sys.initial & env.initial)|!(env.failure);
```

Always the same

```
MODULE Sys
VAR
 index : 0..2;
INIT
 index = 0
TRANS
 case
        index=0 : next(index)!=0;
        index!=0 : next(index)!=0;
 esac
DEFINE
 initial := (index=0);
```

Number of available services

```
MODULE Env(index)
VAR
 operation  : {start_op,a,b,c};
 target : Target(operation);
 s1 : Service1(index,operation);
 s2 : Service2(index,operation);
DEFINE
 initial := (s1.initial & s2.initial & target.initial
               & operation=start_op);
 failure := (s1.failure | s2.failure) |
          (target.final & !(s1.final & s2.final));
```

- Operation alphabet
- Available services
- Initial expression
- Failure expression

Encoding summary (2)

```
MODULE Target(op) --op is an output parameter
VAR
        state : {start_st,t0,t1,t2};
INIT
        state = start_st & op = start_op
TRANS
        case
                state = start_st & op = start_op : next(state) = t0 & next(op) in {a};
                state = t0  & op = a : next(state) = t1 & next(op) in {b};
                state = t1  & op = b  : next(state) = t2 & next(op) in {b,c};
                state = t2  & op = b : next(state) = t2 & next(op) in {b,c};
                state = t2  & op = c: next(state) = t0 & next (op) in {a};
        esac
DEFINE
        initial := state=start_st & op=start_op;
        final := state in {t0,t2}; -- final state(s)
```

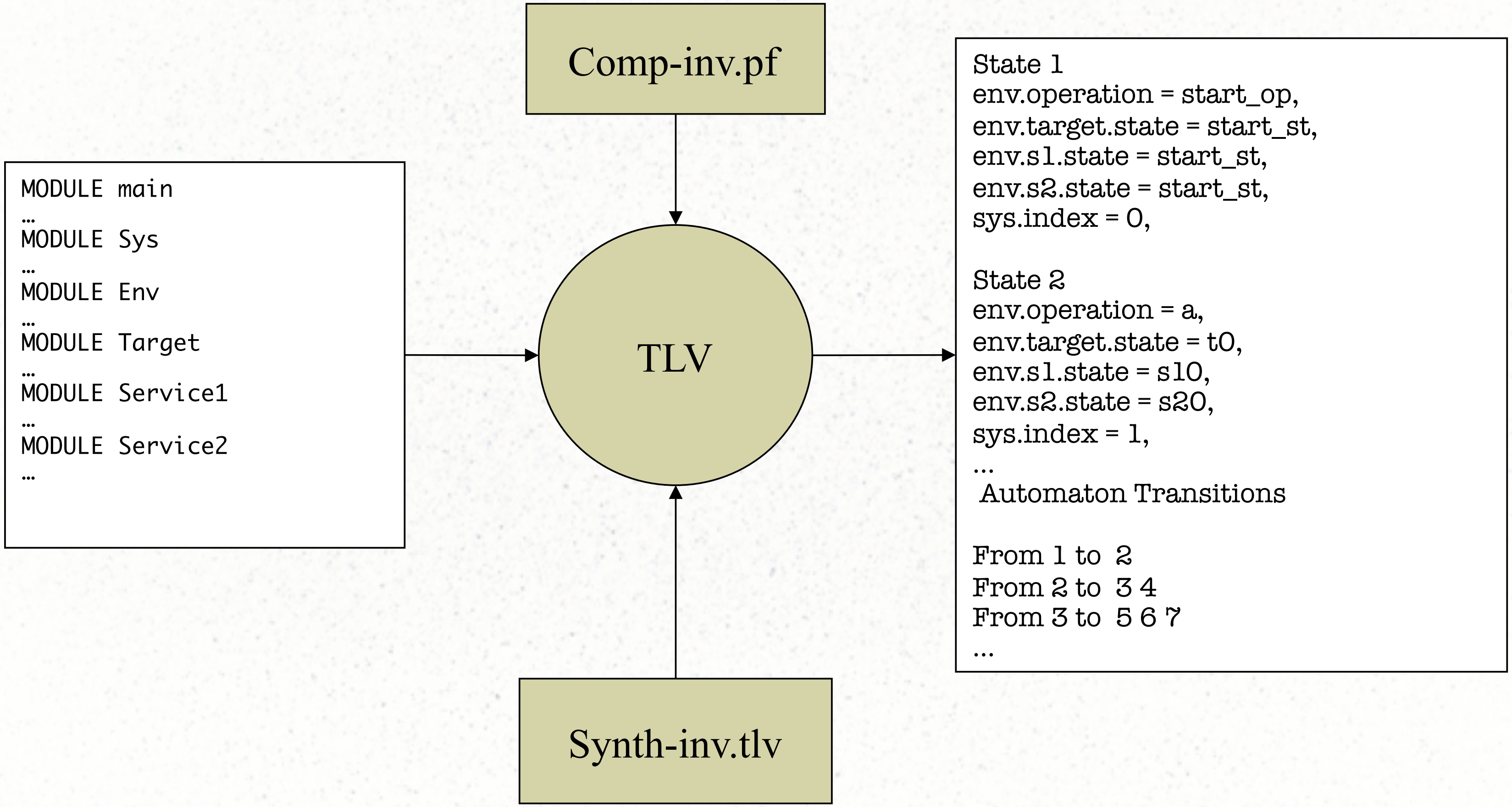
- Keep name and interface
- Change states and transitions
- Define final/init expr's

Encoding summary (3)

```
MODULE Service1(index,operation)
VAR
 state : {start_st,s10,s11,s12};
INIT
        state=start_st
TRANS
          case
                         state=start_st & operation=start_op & index=0: next(state)=s10;
                         (index != 1) : next(state) = state; -- if not selected, remain still
                         (state=s10 & operation = a) : next(state) in {s11,s12};
                         (state=s10 & operation = b) : next(state) in {s10};
                         (state=s11 & operation=b) : next(state) in {s10};
                         (state=s11 & operation=c) : next(state) in {s10};
                         (state=s12 & operation=c) : next(state) in {s12};
                         (state=s12 & operation=b) : next(state) in {s10};
                esac
DEFINE
                initial := state=start_st & operation=start_op & index = 0;
    failure :=
                         index = 1 & !((state = s10 & operation in {a,b})|
                                                (state = s11 & operation in {b,c})|
                                                (state = s12 & operation in {b,c})
                                       );
    final := state in {s10};
```

- Keep interface
- Define name
- States and transitions
- Define final, init and failure

Running TLV


Comp-inv.pf
MODULE main
…
MODULE Sys
…
MODULE Env
…
MODULE Target
…
MODULE Service1
…
MODULE Service2
…
TLV
State 1
env.operation = start_op,
env.target.state = start_st,
env.s1.state = start_st,
env.s2.state = start_st,
sys.index = 0,
State 2
env.operation = a,
env.target.state = t0,
env.s1.state = s10,
env.s2.state = s20,
sys.index = 1,
...
Automaton Transitions
From 1 to 2
From 2 to 3 4
From 3 to 5 6 7
...
Synth-inv.tlv

Running TLV (2)

Automaton States

State 1
env.operation = start_op, env.target.state = start_st,
env.s1.state = start_st, env.s2.state = start_st,
sys.index = 0,

State 2
env.operation = a, env.target.state = t0,
env.s1.state = s10, env.s2.state = s20,
sys.index = 1,

State 3
env.operation = b, env.target.state = t1,
env.s1.state = s12, env.s2.state = s20,
sys.index = 1,

State 4
env.operation = b, env.target.state = t1,
env.s1.state = s11, env.s2.state = s20,
sys.index = 1,
...

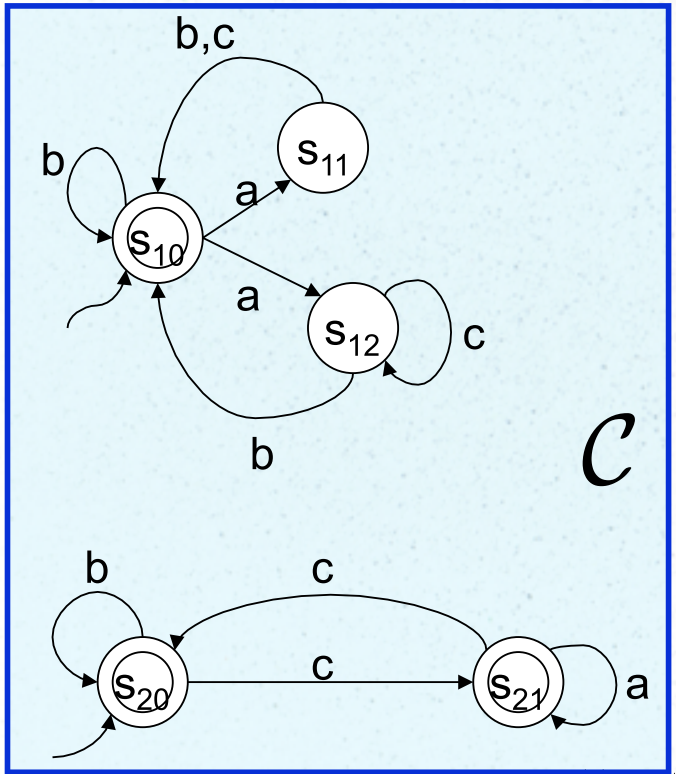
Automaton Transitions

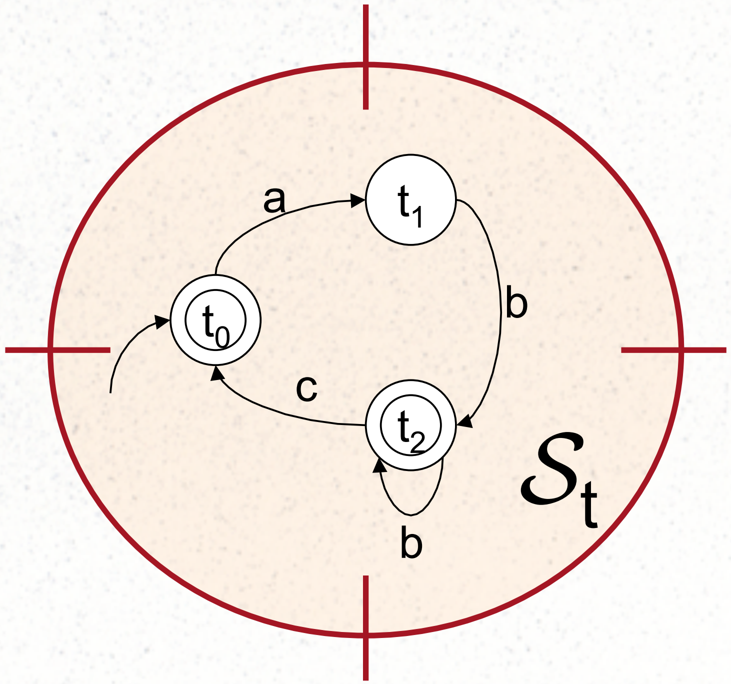
From 1 to 2
From 2 to 3 4
From 3 to 5 6 7
...

From this structure,
We can generate
All possible compositions!

Exercise 1

Encode in SMV and run with TLV the following specification





Exercise 2

Check whether there exists a composition for the following specification. If not, propose a (minimal) modification of the available services such that a composition exists.