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: a service composition problem instance

- 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

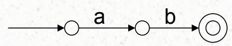
$$G =< \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box\varphi >$$

Equivalence: OG extracted from G’s WINNING set.

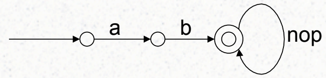
Reduction to Safety-Games (2)

Assumption: TSs have infinite runs

If not…



… do this



States have always 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 $\{1,\ldots n\}$) selected service
- $\mathcal{X}=\{s_t, s_1,\ldots, s_n,o\}$ (environment)
- $\mathcal{Y}=\{ind\}$ (system)

Reduction to Safety-Games (4)

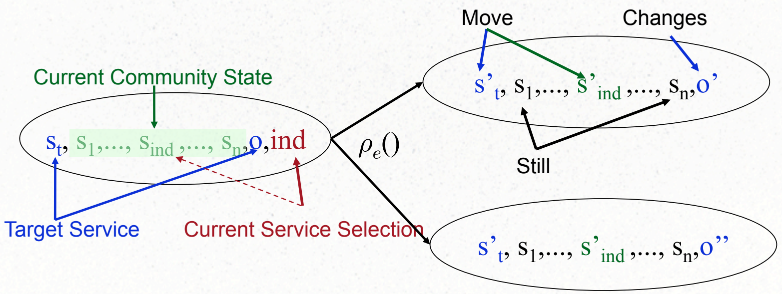
INITIALIZATION

- Θ states that all services are in their initial state
- Actually, an artificial "init" state is introduced (see Lecture Notes for details)

Reduction to Safety-Games (5)

GAME STATE TRANSITIONS

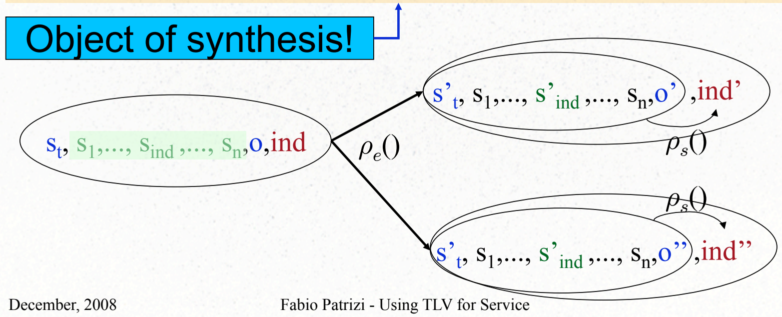
- $\rho_e()$ defines how, given a <u>complete</u> current state,
  - The community changes state
  - The target service changes state and selects next op



Reduction to Safety-Games (6)

GAME STATE TRANSITIONS

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



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
- Synthesis goal is to restrict $\rho_s()$ so that the system wins the game, i.e., satisfies invariant formula

Reduction to Safety-Games (8)

GAME INVARIANT

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

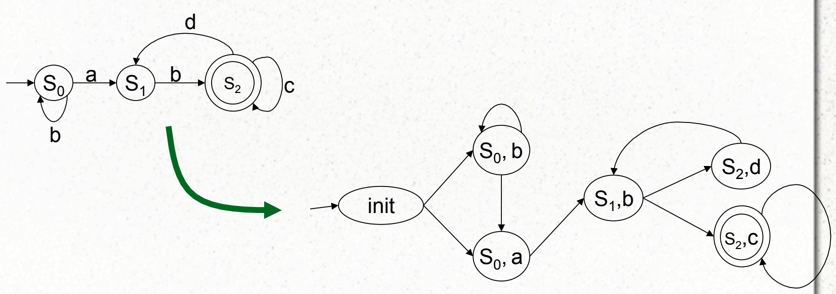
$fail_i$ holds if S_i is selected but is not able to perform requested operation

If target service is in a final state then all available services do, as well

Reduction to Safety-Games (9)

GAME STATE TRANSITIONS

Observation: target operations moved into states!



Reduction to Safety-Games (10)

For general rules, see Lecture Notes

(soon available online)

Reduction to Safety-Games (11)

Once we encoded our service composition problem in a safety-game…

Theorem:

1. A composition exists iff the maximal winning set contains all initial game states
2. Compute the maximal winning set and you get the composition generator, i.e., the whole set of compositions

Reduction to Safety-Games (12)

"2. Compute the maximal winning set and you get the composition generator, i.e., the whole set of compositions"

Good! But…

… how to compute the maximal winning set?

Use TLV!

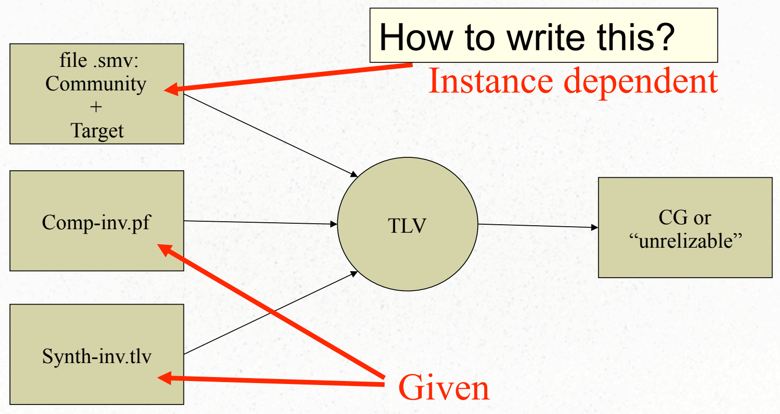
TLV

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.

TLV (2)



TLV and SMV

- TLV is the software system
- SMV is the language used for input specification

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

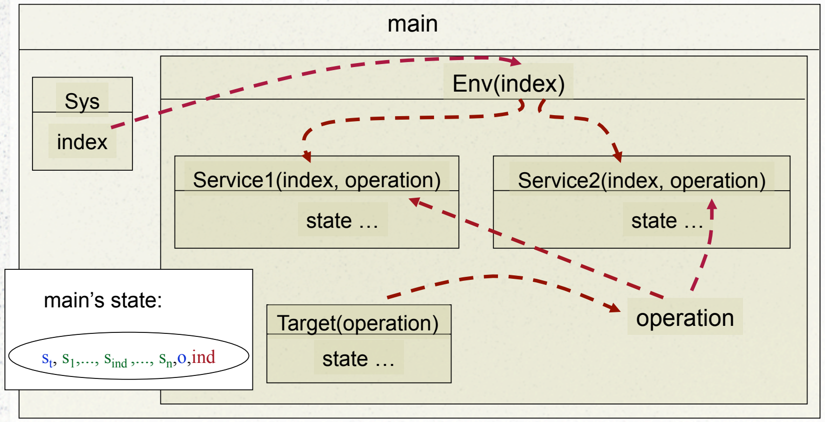
boolean expressions

```
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

- A specification is 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 $\mathcal{S}_i$

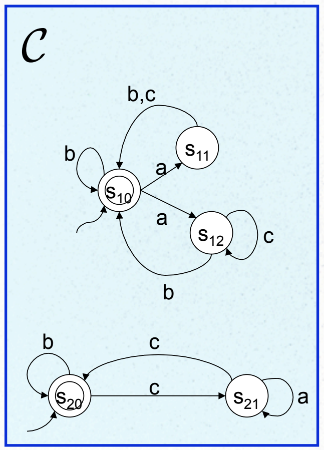
Module Interconnections

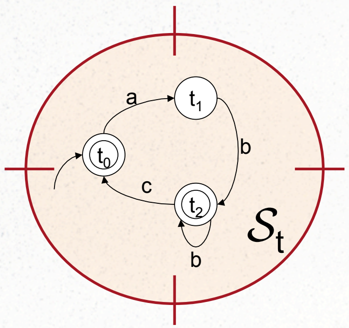


Module Transitions

- main's submodules run in parallel
- At each clock tick they all move, according to their current state and specification
- We constrain non-selected modules to loop in their current state
- The whole module main is itself a transition system

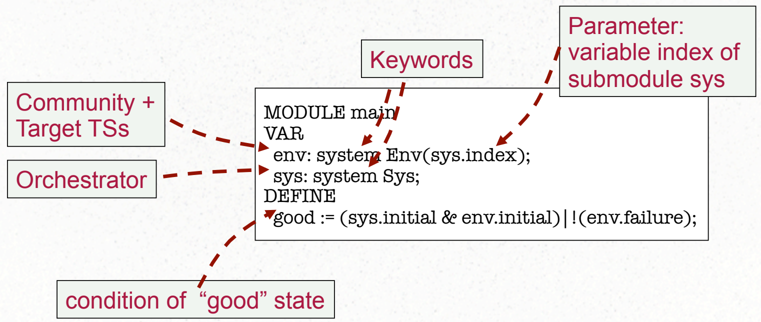
SMV encoding by examples





Module main

- Instance independent



Module Sys

- Depends on number of available services.

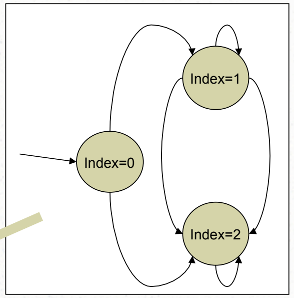
```
MODULE Sys
VAR
 index : 0..2;
INIT
 index = 0
TRANS
 case
      index=0 : next(index)!=0;
      index!=0 : next(index)!=0;
 esac
DEFINE
 initial := (index=0);
```

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

Module Env

```
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));
```

Target service

Available services

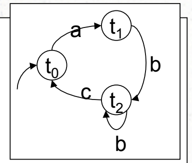
Operation alphabet + special operation start_op used for init

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

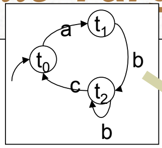
Initial/final states

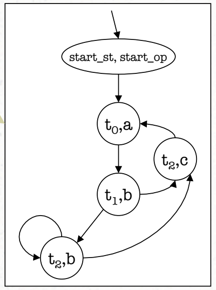
Next operation (non-deterministic)

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

- Strong dependence on instance (same as target service)
- Nondeterministic in general

Available Service Modules (2)

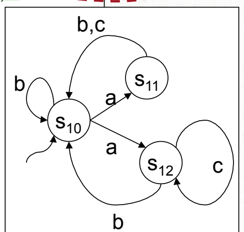
```
MODULE Service1(index,operation)
VAR
 state : {start_st,s10,s11,s12};
INIT
      state=start_st
TRANS
        case
          state=start_st & operation=start_op & index=0: next(state)=s10;
          (index != 1) : next(state) = state; -- if not selected, remain still
          (state=s10 & operation = 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)
```

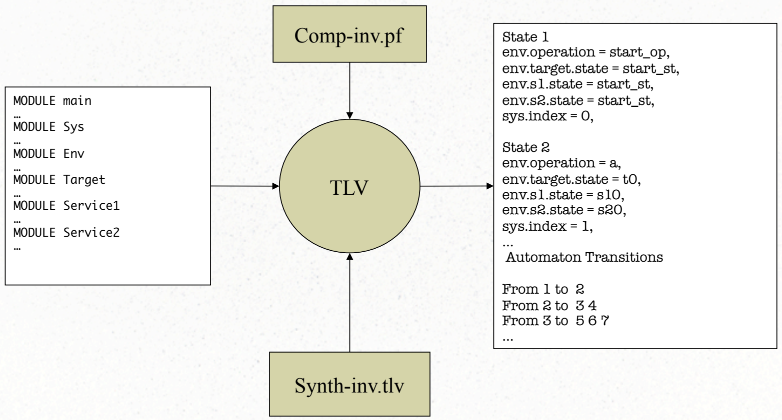
- Keep name and interface
- Change states and transitions
- Define final/init expr's

Encoding summary (3)

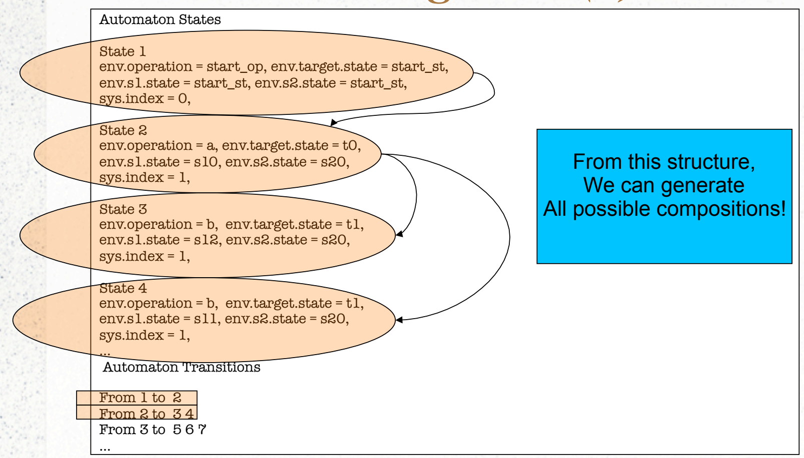
```
MODULE Service1(index,operation)
VAR
 state : {start_st,s10,s11,s12};
INIT
        state=start_st
TRANS
        case
                state=start_st & operation=start_op & index=0: next(state)=s10;
                (index != 1) : next(state) = state; -- if not selected, remain still
                (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
- Set (standard) name
- Set states and transitions
- Define final, init and failure

Running TLV



Running TLV (2)



Lecture Notes

- Lecture Notes about first part (Reduction to Game Structure)
- Slides (for SMV encoding)
- Online before next week's practice. Bring your laptop!
- See course website

`www.dis.uniroma1.it/~degiacom/didattica/software-services/`