Using TLV for Web Service Composition via Simulation

Abstract.

1 Introduction

2 Composition of Web Services

2.1 The problem

- Web services and web service composition
- Available services, community, target service
- Problem statement

2.2 Composing Web Services via simulation

- Simulation
- Composing Web Services via simulation
- References to Simulation Paper: Orchhestrator generator, orchestartor, etc...

3 Formalizing the problem for TLV

3.1 Target and available services as Transition Systems

We deal with deterministic *target* service and non-deterministic *available* services, described as transition systems, according to the following definition:

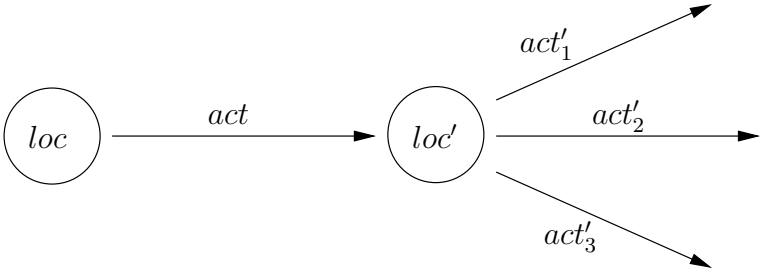
Definition 1 (Transition System). *A transition system (TS) is a tuple* $TS_i = \langle S_i, A_i, s_i^0, \delta_i, F_i \rangle$ *(*$i = 1, \ldots, n, t$*) where, for* $i = 1, \ldots, n, t$*:*

- S_i *is the finite set of* TS_i *states;*
- A_i *is the set of* TS_i *actions.* $A = \bigcup_{i=1}^{n} A_i$ *is called the* shared action alphabet*;*
- s_i^0 *is the initial state of* TS_i*;*
- δ_i *is the transition relation* $\delta_i \subseteq S_i \times A_i \times S_i$ *of* TS_i*. For* $i = t$ δ_t *is* functional *and can be rewritten, for convenience, as* $\delta_t : S_t \times A_t \rightarrow S_t$ *where* $\delta_t(s, a) = s'$ *iff* $\langle s, a, s' \rangle \in \delta_t$*;*
- $F_i \subseteq S_i$ *is the set of* TS_i *final states.*

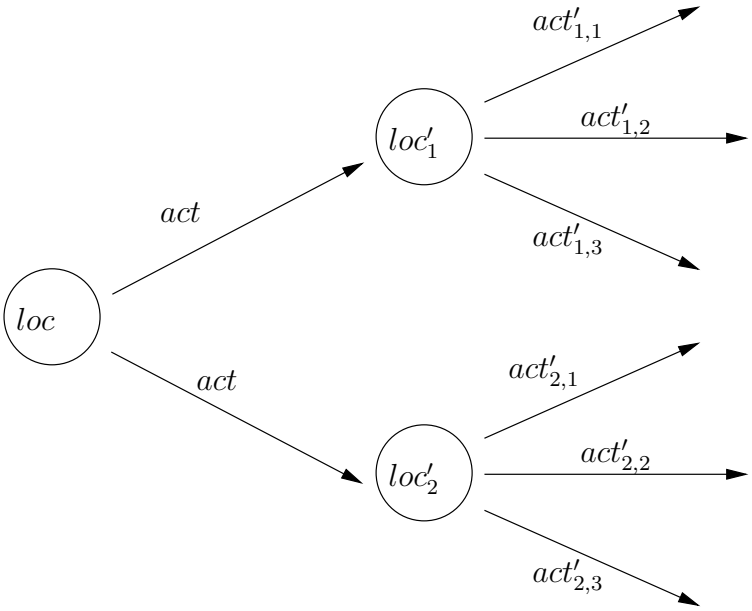
Moreover, TS_t is called target *service, while TS_i, for $i \neq t$, is called* available *service.*

Figures 1(a) and 1(b) represent two sample fragments of target and available service, respectively. Note that the target service of Figure 1(a) is deterministic while available service of Figure 1(b) is not.

Our aim is to provide an exact procedure for describing a community of available services plus a target service in the language SMV and, consequenlty, exploiting the tool TLV for synthesizing the target service automatically, using a procedure which relies on the notion of *simulation*.



(a) A target service fragment

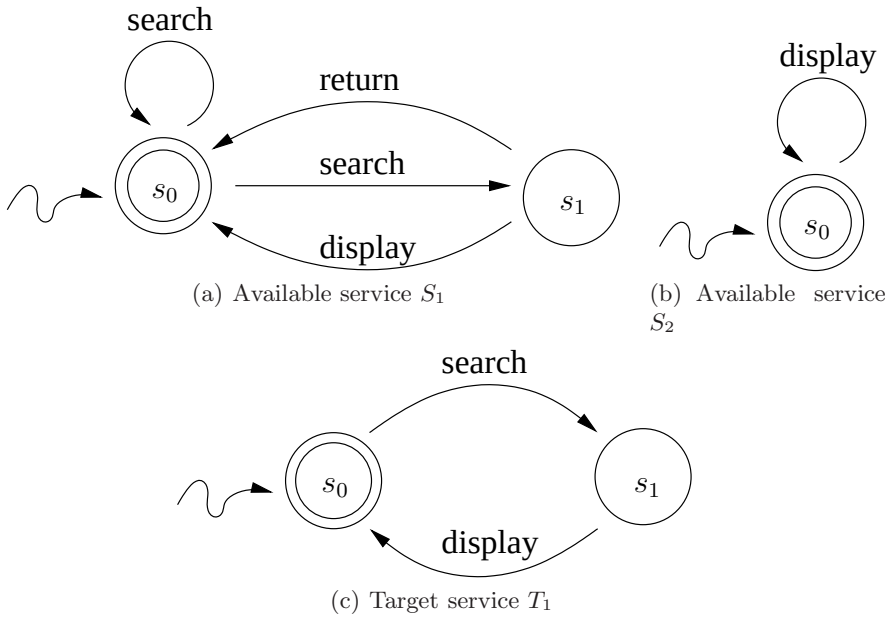


(b) An available service fragment

Fig. 1. Sample transition system fragments

3.2 Writing transition systems for web service composition via TLV

We will present our methodology by using, as an example, the problem instance depicted in Figure 2, where a community composed of 2 available services, $\mathcal{C} = \{S_1, S_2\}$, and a target service T_1 are described. Note that available service S_1 is non-deterministic and all services include final states, represented by double circles.



(a) Available service S_1

(b) Available service S_2

(c) Target service T_1

Fig. 2. Available and target services

The module main In order to perform the synthesis, we need two basic components: the *system* and the *environment*, both modeled as SMV modules, properly interconnected. In addition, an assertion `good` needs to be defined which represents the invariant property that has to be guaranteed by the synthesized controller (which, indeed, implements the strategy). Specification always include the following module `main`, which is independent of the particular instance, where modules `Input` and `Output` are described in forthcoming paragraphs:

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

```
-- end of module main
```

Basically, the synthesis procedure computes a restriction, if any, on the transition relation of `Out`, which represents the *orchestrator*, such that property `good` is always satisfied ($\Box$`good`) however module `In` behaves (according to its own specification). The way `Out` may affect `In` evolution is through parameter `index`, which is a *write-only* parameter for the latter module and, thus, does not affect `Out`.

The module Output This module represents the orchestrator. It implements a strategy which selects, at each time point, an available service for executing current target action. At each time point, it assigns the variable `index` a value representing such service. Of course, each available service is assigned a different `index` value. Hence, if we deal with 2 available services, SMV specification of module `Ouptut` is as follows, where `index=0` corresponds to assigning available services no action and is needed only for system initialization:

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

As it can be seen, apart from initialization, no restriction on `Output`'s transition relation is given. That is, `Output` may *produce*, upon execution, any sequence of `index` assignments of the form: $0(1|2)*$. The synthesis procedure is exactly aimed at restricting the set of such sequences, in order to properly constrain environment evolution in a way such that invariant property `good` is guaranteed.

The module Input This is the central module of our specification. Refer again to the community $\mathcal{C}$ and the target service T_1 of Figure 2. Provided all systems start in their respective initial state, T_1 can be obtained by *composition* of S_1 and S_2 if and only if any sequence of actions it produces can be *reproduced* by properly assigning each action to either S_1 or S_2, according to their transition relations. In addition, whenever T_1 is in a final state, *all* community services are required to be in a final state.

Consider the following SMV code fragment, which represents module `Input` specification for community $\mathcal{C} = \{S_1, S_2\}$, with shared action alphabet $A = \{\texttt{search}, \texttt{return}, \texttt{display}\}$:

```
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));
-- end of module Input
```

Basically, this module represents the *synchronous product* (i.e., all systems evolve at each clock tick) of transition systems `T1`, `S1` and `S2`, which respectively model services $T1$, S_1 and S_2. Variable `action` and assertion `failure` are defined. The former may take values from the shared action alphabet `{search,display,return}` plus the special action `nil` (needed for initilization) and represents the current target service action to be performed by available services, while the latter depends on all services evolution and represents the violation of some constraint. In particular, it is `TRUE` iff:

- either `S1` or `S2` generate a failure (see below), or
- it is not the case that whenever `T1` reaches a final state, both `S1` and `S2` also do.

Note that invariant property `good`, defined within module `main`, corresponds exactly to the negation of `failure`. Hence, assuming `mT1`, `mS1` and `mS2` properly defined, when TLV is executed with such a specification as input, it tries to synthesize a *strategy* (or *orchestrator*, or *controller*) for `Out` which forces `In` to evolve in a way such that i) no available system generates a failure and ii) the target service being in a final state implies each available service being in a final state, too. The intuition behind module `Input` is to encode the following behavior:

- according to its state at time step $t-1$, `T1` evolves and produces a new action (initial action is `nil`);
- `Out` assigns `index` a new value (initial value is 0);
- if $i =$ `index` at time step $t-1$, then service S_i executes target action of time step $t-1$ and reaches consequent state at time step t; otherwise, S_i remains still (initially, available services are in their respective initial state).

Note how the use of `index` allows, at each time step, only one available service to change state, while target service and orchestrator are always allowed to.

The target service module As already discussed, a target service is a deterministic finite state machine $TS_t = \langle S_t, A_t, s_t^0, \delta_t, F_t \rangle$, where δ_t is, indeed, a function. In SMV, we associate each service a module which describes a transition system. In particular, the following module represents an SMV specification for target service T_1 of Figure 2(c):

```
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);
-- end of module mT1
```

Each state of the SMV target transition system is characterized by:

- the current *location* `loc`, that is, the state of the computation it is in. The domain of variable `loc` is the set of S_t state indices, with obvious correspondences (`loc=0` corresponds to state s_0 and so on). Let $ind(s_i) = i$ be the bijection which defines such correspondences.
- the current *action* `act`, that is, the action to be executed next (by some available service). The domain of `act` is exactly the domain A_t plus the special action `nil`.

Since `act` needs to be passed as input to other modules (in particular, to module `Input`) it has been defined externally of the target module. However, a mechanism analogous to the one exploited for making variable `index` of module `Out` available to module `In` (see module `main` above) might be equivalently used. Recall that we interpret the target service as an action *producer* and, consequenlty, it is the module which assigns (output parameter) `act` some value. The way such values are assigned depends on transition function δ_t, as it directly defines the `ASSIGN` section of module `mT1`.

Initially, the system is in location `0` (`init(loc):=0`), which is, by convention, the initial one. Since target and orchestrator evolve one time step before available services –"first, choose action and service, then execute"–, a `nil` action is initially performed (`init(act):=nil`) which forces all services to remain still without generating any failure (see available service SMV specification below). Then, next location and action can be defined, depending on current ones. Note that target evolution does not depend on any other variables but location and action, i.e., no *feedback* from available services is considered.

The rest of `mT1 ASSIGN` section is directly derived from target service transition function, according to the following rules:

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:

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

These rules apply in general, only if a *deterministic* target service is to be modeled. Otherwise, for the non-deterministic case, a different modeling schema must be adopted. Note that specifications obtained by applying such procedure are not *optimal*, that is, they are not guaranteed to be the most concise and/or of fastest resolution. However, further (semantics-preserving) modifications can be applied.

Finally, an expression `final` is defined within the `DEFINE` section, which is `TRUE` iff target service is in a location corresponding to a final state. As already discussed, such expression is useful in module `Input`, for defining `failure` expression.

The available service module mS1 Now, focus on available service S_1 of Figure 2(a). Such service is modeled in SMV by adapting the approach used for target service T_1 to deal with two major differences, that is:

- available services are, in general *non-deterministic*;
- available services are action *consumer*, that is, instead of assigning actions, they *execute* them (or do nothing, if not selected).

The following excerpt represents the SMV code which models the transition system associated to available service S_1:

```
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);
-- end of module mS1
```

Note that `mS1`'s evolution depends on (input) parameters `index` and `action`. As previously anticipated, each service is assigned a unique `index` value with the purpose of *selecting* the service itself. When a service is selected, it is supposed to execute the current action and evolve consequently.

Previous discussion about variable `loc` and its respective initialization (see target service section) also applies to this case. For what concerns sections `ASSIGN` and `DEFINE`, however, some differences have been introduced.

First, `ASSIGN` section includes no statement involving `action` since, in fact, available services simply react to target evolution and, hence, they need only to read it.

Second, `case` statement, which defines next system location, includes the following `case_0` condition:

```
index != 1 : loc
```

which forces the system to remain still whenever `index` is different from 1, such value being the unique identifier assigned to service S_1. Remaining `case` conditions are obtained by applying the same construction exploited for target services, i.e.,

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

however, in this case, note that δ_i is not *functional*, the service being, in general, non deterministic and, therefore, $\delta_i(s, a)$ is a set, instead of a single element.

Lastly, as for section `DEFINE`, assertion `final` is defined as exactly as within previous module `mT1`. In addition, a second assertion, `failure`, is defined which is true iff service S_1 is selected (`index=1`) but it is asked for executing some not allowed action. Recall that the negation of such assertion (cf. modules `Input` and `main`) appears as a conjunct of invariant property `good`, which must be ensured by the strategy TLV searches for. In other words, by defining it, we are asking TLV for synthesizing an orchestrator (i.e., restricting `Output` transition relation) which *never asks available services for executing actions they cannot execute (at the given time point).*

The available service module mS2 For completeness, we report the SMV code associated to service S_2. Note that such service is stateless and deterministic, leading to the following, simplified, specification:

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