Sapienza universita' di Roma

Dipartimento di Informatica e Sistemistica

User guide
WSCE-Lite
Web Service Composition Engine

v 0.1

Valerio Colaianni

Contents

1 Installation **5**
1.1 Installing TLV 5
1.2 Installing WSCE-Lite 6

2 Usage **7**
2.1 Using the tool 7
2.2 WS-TSL 8
2.2.1 Syntax 8
2.2.2 Mapping TS in WSTSL 10
2.3 Running the tool 12

Chapter 1

Installation

In order to run and use Wsce-lite you will need to setup the following enviroment:

1. install the latest version of Java2 Standard Edition (version 6.0 or higher)

2. install TLV

3. Install the tool

1.1 Installing TLV

In order to install TLV on your system

1. Copy the directory TLV-4.14 in your local file system;

2. Copy library cygwin1.dll in your local file system;

3. Set enviroment variable TLV_PATH with the absolute path of directory rules of TLV;

4. Append to the enviroment variable PATH with

   - The absolute path of bin directory (in which is present the executable code of TLV);
   - The absolute path of the directory in which is present cygwin1.dll;

5. Check the correctness of the installation let running the example provided. To run TLV use this command: tlv comp-inv.pf $< path_smv_file >$ (example file can be found in the sub directory example)

1.2 Installing WSCE-Lite

To install Wsce-lite all you have to do is copy the directory WSCE in your local file system.

In WSCE directory you will find two directory:

1. *lib* directory, that contains libries Wsce-lite needed to properly work. **Never** edit the content of this directory;
2. *composizioni* directory, that is the working directory. Here you will put your input file for the composition problem you want to solve.

Once you correctly setup your enviroment you are ready to use the tool for automatic service composition. Before using it we suggest you to run an example provided in the composizioni directory.

Chapter 2

Usage

To use the tool just run the jar file of the tool with the following sintax:

$$java\ \text{-}jar\ wsce.jar < working_dir_name >$$

You can run an example provided with the distribution of the tool (e.g. test1), if everything work you have correctly installed the tool. If an error occurs please check the steps of chapter 1, before you continue using the tool.

2.1 Using the tool

Wsce-lite is a tool for automatic service composition. To use the tool follow these steps:

1. Create a sub directory into composizioni directory, for example let's call this directory my-comp;

2. Into my-comp directory create wstsl sub directory, into this directory you will put the description of the transition systems of the available services;

3. into wstsl directory create target sub directory, into this directory you will put the description of the transition system of the target service you intend to synthesize. Please notice that the name of the target must be the name of your working directory (in our example my-comp.wstsl);

All your input file must be written in WS-TSL language (more later in the next section)

2.2 WS-TSL

WSTSL is an XML based language. It describes the Transition System that models the Web Service, that is its behavior, without provide its implementation. It allows to represent what is observable from the point of view of the user (the change of state of the Web Service after performing an action).

2.2.1 Syntax

The syntax of a WSTSL document is described with an XML Schema Definition Language (XSD) document. XML Schema can be used to express a schema: a set of rules to which an XML document must conform in order to be considered 'valid' according to that schema. An XSD defines a type of XML document in terms of constraints upon what elements and attributes may appear, their relationship to each other, what types of data may be in them, and other things.

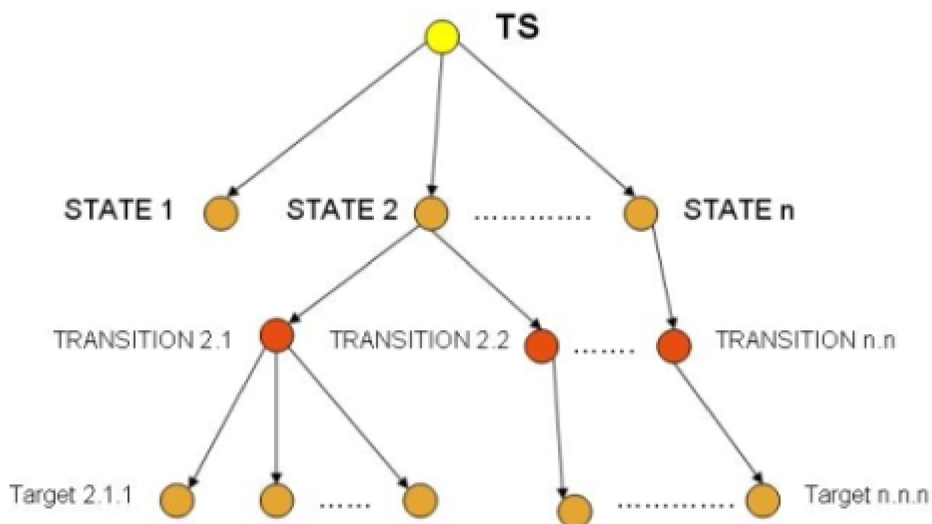

Figure 2.1: example of a syntactic WSTSL tree

Tag TS A WSTSL document is composed by a root tag TS which is structured as a sequence of STATE tag, minimum one, which are the states of the given Transition System, and with a mandatory attribute "service", which its use is required according to XSD grammar, that is the name of the Web Service we are describing as shown into the XSD fragment below.

```
<xsd:complexType name="TSType">
        <xsd:sequence>
                <xsd:element name="STATE" type="ns:StateType"
                                minOccurs="1" maxOccurs="unbounded">
```

```
                </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="service" type="xsd:string"
                use="required"/>
</xsd:complexType>
```

Tag STATE The STATE tag is composed as a sequence of TRANSITION tag, that represents a transition, of the Transition System that have as source state the given state; More over, the STATE tag has two attributes: name, that is the name of the state, and tipology which is the description of the state.

Each tipology of state can be in this set: $\{initial{-}final, final, transient\}$ according with the conceptual schema. We used a XSD enumeration to represents it.

```
<xsd:complexType name="StateType">
        <xsd:sequence>
                <xsd:element name="TRANSITION" type="ns:TransitionType"
                        maxOccurs="unbounded">
                </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="tipology" type="ns:StateTipology"
                use="required"/>
</xsd:complexType>

<xsd:simpleType name="StateTipology">
                <xsd:restriction base="xsd:string">
                        <xsd:enumeration value="initial-final"/>
                        <xsd:enumeration value="final"/>
                        <xsd:enumeration value="transient"/>
                </xsd:restriction>
</xsd:simpleType>
```

Tag TRANSITION The TRANSITION tag is composed as a sequence of TARGET tag, that are the target state of this transition. The cardinality are minimum one maximum unbounded. Minimum cardinality must be one 'cause we need at least an endpoint for the transition. Maximum cardinality is unbounded 'cause in non deterministic transition system we have multiple endpoint for an action originating from one state. We decide to not repeat transition tag with the same action for every transition, we prefer to write in once, and for every state reached after performing that action to write a target tag. This ease the compactness of the document, the readable and, from the author point of view, the elegance. Transition tag has a mandatory attribute named action, that is the name of the action that cause this transition.

The TARGET tag, we have already introduced, has only the attribute (mandatory) state, and as said before, it is the name of the target state of the transition.

```
<xsd:complexType name="TransitionType">
                <xsd:sequence>
                        <xsd:element name="TARGET" type="ns:Target"
                                minOccurs="1" maxOccurs="unbounded">
                        </xsd:element>
                </xsd:sequence>
                <xsd:attribute name="action" type="xsd:string"
                        use="required"/>
</xsd:complexType>

<xsd:complexType name="Target">
                 <xsd:attribute name="state" type="xsd:string"
                        use="required"/>
</xsd:complexType>
```

CONSTRAINTS Now that we have build the syntactical rules to write a WSTSL document, we have to ensure integrity references. To do this we used two XSD constraints: uniqueness of the name of the state (there cannot exist two different states with the same name), and a foreign key between the target state of the transition and the name of the state, to ensure that don't exists transitions with ghost states, remember that TARGET elements must be a subset of STATE elements.

```
<xsd:element name="TS" type="ns:TSType">

                <xsd:unique name="uniqueState">
                        <xsd:selector xpath="./STATE" />
                        <xsd:field xpath="@name"></xsd:field>
                </xsd:unique>

                <xsd:key name="stateKey">
                        <xsd:selector xpath="./STATE" />
                        <xsd:field xpath="@name"></xsd:field>
                </xsd:key>

                <xsd:keyref name="transitionTarget" refer="ns:stateKey">
                        <xsd:selector xpath="./STATE/TRANSITION/TARGET" />
                        <xsd:field xpath="@state"></xsd:field>
                </xsd:keyref>

</xsd:element>
```

2.2.2 Mapping TS in WSTSL

Now that we have defined how to write a WSTSL document, let's see how to map a Transition System of a Web Service into WSTSL.

We map the Web Service in a TS tag, with the name of the service in the community. In our example the name is "Service1". The mapping of states and transitions is made up using this general procedure:

- For each state of the service, declare a tag STATE, in which write the name of the state, and the type of the state.

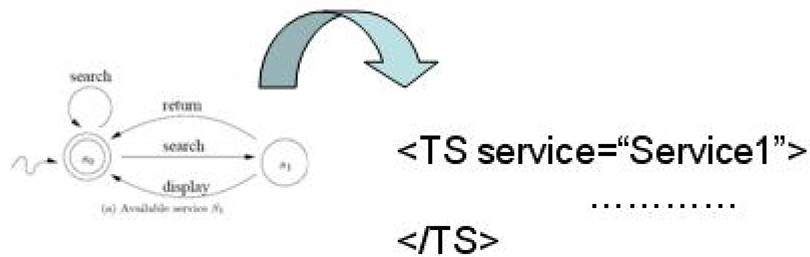

Figure 2.2: Mapping the TS

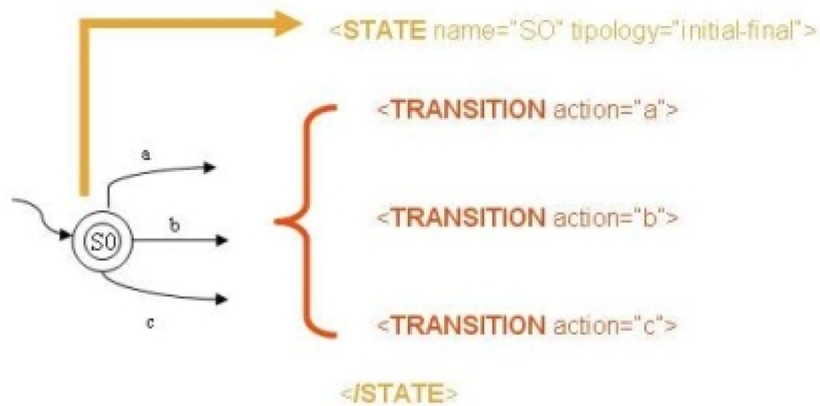

Figure 2.3: Mapping a state and outgoing transitions

- For each outgoing transition of this state, declare a tag TRANSITION with the name of the action that cause that transition.
- For each state reached by that transition, declare a tag TARGET in which write the name of the state.

As we can see the mapping is very simple, and this was one of the requirement of WSTSL specification. Below we present an example of the mapping of web service shown in figure 2.5

```
<TS
    xmlns='http://www.dis.uniroma1.it/WS-TSL'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:schemaLocation='http://www.dis.uniroma1.it/WS-TSL'
    service="Service1">

        <STATE name="S0"  tipology="initial-final">
        <TRANSITION  action="search">
```

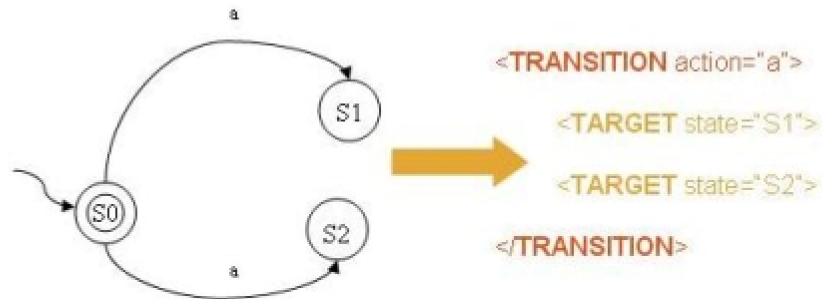

Figure 2.4: Mapping of a (non deterministic) transition

```
            <TARGET state="S1"/>
                        <TARGET state="S0"/>
        </TRANSITION>
    </STATE>

        <STATE name="S1" tipology="transient">
        <TRANSITION action="back">
            <TARGET state="S0"/>
        </TRANSITION>
                <TRANSITION action="display">
            <TARGET state="S0"/>
        </TRANSITION>
    </STATE>
</TS>
```

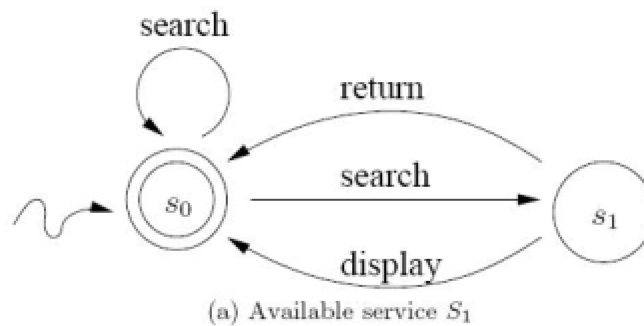

Figure 2.5: transition system of service1

2.3 Running the tool

Once you've created the xml file of the problem you want to solve you can run the tool to calculate the composition. After calculate the composition in your working directory (my-comp in the example) you will find

- the SMV file of the problem, this is the input for TLV (.smv file);
- the TLV output of the problem (.txt and .nor file);
- an xml representation of the transition system (Mealy automata) representing the composition, obtain by the TLV's output (composition.xml);
- Sql script to store the composition into a relation database (composition.sql);