Manual for Jambition (Version PLASTIC)

Model-Based Testing of Stateful Web Services

Lars Frantzen[1,2]

[1] Instituto di Scienza e Tecnologie della Informazione "Alessandro Faedo"
Consiglio Nazionale delle Ricerche, Pisa – Italy
lars.frantzen@isti.cnr.it
[2] Institute for Computing and Information Sciences
Radboud University Nijmegen – The Netherlands
lf@cs.ru.nl

Abstract. Jambition is a prototype tool to automatically test the functionality of stateful Web Services based on Service State Machine (SSM) specifications. SSMs are similar to UML state machines, but tailored to the domain of Web Services. They have a precise semantics and are especially suited to perform Model-Based Testing. SSMs specify a single port type. The Web Service to be tested is assumed to be passive, meaning that only `request-response` and `one-way` operations are defined in the corresponding WSDL. Jambition supports a subset of the XML-Schema data types. This short manual briefly motivates and explains the tool and the specifications needed to perform model-based testing of a Web Service port.

1 Overview

Every Web Service provides a set of operations to its potential users. To know which operations are available, an interface specification is needed. Commonly, Web Service interfaces are specified in the *Web Service Description Language (WSDL)*.

For instance, a Web Service representing a warehouse may offer operations to check the availability of products, and to order such products. In the first case, an object representing a quote request is sent, and an object representing a quote is returned. The corresponding WSDL file makes the signature of these operations public. This information is sufficient to connect to the Web Service, and to invoke the operations, but it does not give any kind of semantic information. For instance, the warehouse may only accept quote-requests of a certain quantity. Or it may only allow to order products when the availability has been requested beforehand. Or it may guarantee that every offered quote deals with the same quantity as the requested quote has indicated. WSDL files are not intended to provide such kind of information.

One natural way to extend a Web Service description in this direction is to use state machines. Jambition uses a dedicated variant of state machines which is especially useful for Model-Based Testing – *Service State Machines (SSM)*. Such a state machine can be used to express constraints on the data as it is passed via the operations, and it gives a legal ordering of the operation invocations. Hence, properties like the ones stated above for the warehouse can be expressed via SSMs.

Such an SSM model can suit several needs. For instance, it gives a specification of the dynamic aspects of the Web Service invocations. A user of the Web Service knows which

operations are allowed to be invoked at what point in time. She also knows the restrictions on the data to be sent and received. SSMs are a valuable means to extend Web Service specifications.

Furthermore, an SSM can be used to automatically test a Web Service. This is what Jambition does. It takes a WSDL and an SSM specification of a Web Service as input. Based on these it fully automatically generates invocations to the Web Service, receives the returned messages, and checks if this data is conforming to the SSM specification.

Following the warehouse example, Jambition will respect the protocol as it is encoded in the SSM. For instance, it will always check the availability before making an order. It will also respect the data constraints, for instance making orders with a quantity of at least three products. With respect to testing, Jambition will receive the responses of the operation calls it makes, and then check if also the warehouse respects its constraints on the data. For instance, Jambition will check if the warehouse makes only offers for the requested products.

If Jambition spots a failure it will report so and stop the testing. While no failure is found, it does a random walk through the SSM, meaning that when there are several inputs specified, it chooses one randomly. With respect to the parameters of operations, Jambition only chooses data values which respect the constraints (like a quantity ≥ 3). Usually the first solution of the constraint is chosen (like quantity $= 3$).

To visualize and to keep track of how the test proceeds, Jambition offers a GUI which monitors the testing events. It is also possible to log the ongoing testing to rotating log files. Furthermore, Jambition can be connected to the open source tool `Quick Sequence Diagram Editor`, which displays the communication between Jambition and the Web Service in real time as an UML sequence diagram.

2 Technical info

Intended audience Developers of stateful Web Services

License GPLv3

Software Java 6 Runtime Environment (mandatory), Minerva library [8] (mandatory), treeSolver [7] (mandatory), dot [3] (optional), Quick Sequence Diagram Editor [6] (optional). The treeSolver is available for Linux and Windows. Other operating systems are supported by recompiling. Ideally, the Web Service to be tested should be deployed on a Glassfish [1] server. Other applications servers may demand minor adaptations of Jambition.

Download The official version can be retrieved at [8]

Future Tasks SSMs can in principle also be used to model the communication between several Web Services. To do so they have to involve the message flow at several ports. Testing based on such multi-port SSM would allow to test more complex scenarios like coordinated and composed Web Services.

Many Web Services use lists as data types. Lists correspond to an unbounded sequence in XML, which is not supported by the current version. Supporting lists is another eligible future step.

With respect to the testing itself, the purely random approach might not always be satisfactory. Specific coverage criteria can be conceived.

The constraint solving can be more flexible and general. For instance, dealing with negative integers can be achieved by changing the underlying constraint solver. Also, a "first solution found" approach is sometimes too restrictive.

Bugs Jambition in its current version is not always very helpful in providing information when something is/goes wrong. Running it via a command-line is definitely recommended to at least see potential exceptions which are not reported by Jambition. Several features are experimental, and several features are missing to make it applicable for the general case.

Contact Lars Frantzen<lars@frantzen.info>

3 Deployment

The relevant concepts of Jambition are depicted in Fig. 1. The structural aspects (data

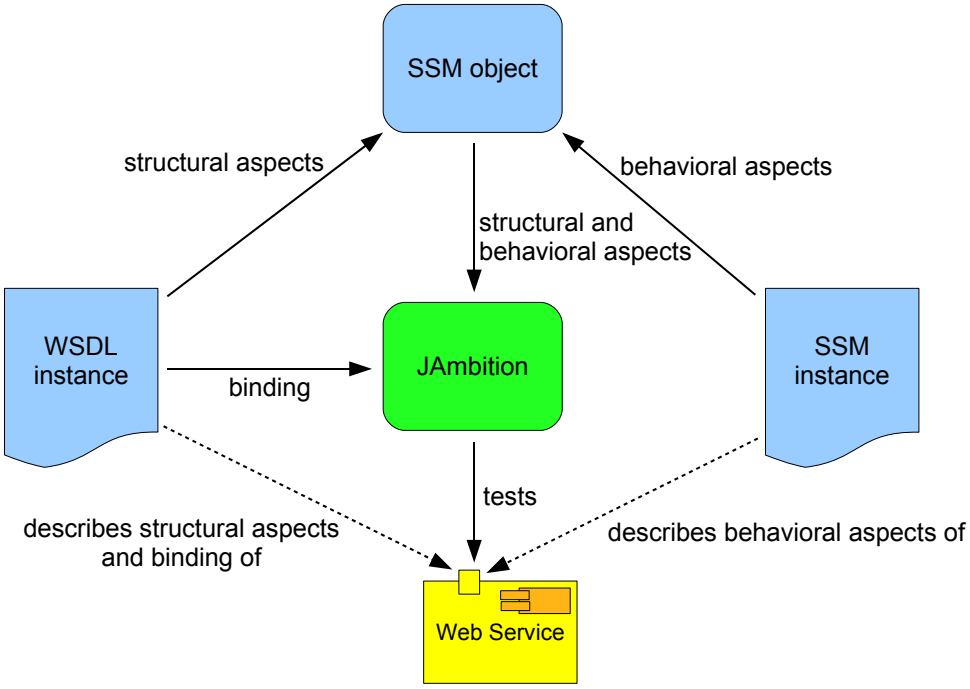

Fig. 1. Jambition Concepts

types, messages, operations, port types) of the Web Service to be tested are described in a WSDL instance. The behavioral aspects (states of the Web Service, ordering of invocations, constraints on the data flow) are described in an SSM instance. Both aspects are combined together in an SSM object. This object is the main reference of Jambition to do the

automatic testing. Furthermore, the binding information of the WSDL file (service, port) is used to physically connect to the Web Service.
To model the SSM, Jambition offers two ways:

- the SSM can be described textually in an XML document
- the SSM is visually designed with the `MagicDraw` [5] tool, a commercial UML modeler

In this manual we will only describe the XML-based option. Using `MagicDraw` to model SSMs is described in the manual of the Minerva [8] library. When using an XML description, Jambition needs four inputs, see Fig. 2.

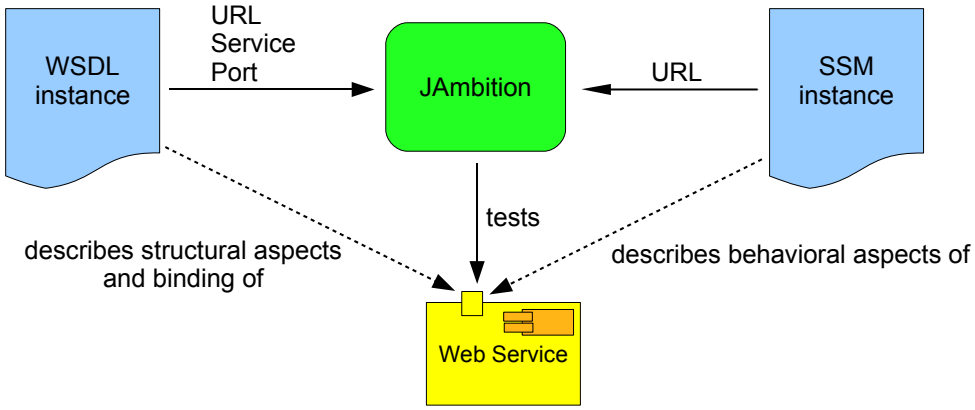

Fig. 2. Jambition Inputs

1. The URL of the WSDL instance
2. The service name of the Web Service to be tested as being given in the WSDL instance (since a single WSDL can define several services)
3. The port name of the Web Service to be tested as being given in the WSDL instance (since a single WSDL can define several ports per service)
4. The URL of the XML SSM instance

Based on these inputs, the SSM object is generated and the testing can be started.

3.1 Install

Jambition is written in Java 6, hence it needs a Java 6 JRE installed. It comes as a `Jambition.zip` archive. Within the archive there is the `Jambition` directory. Just extract this directory to a desired place.

To do anything useful, Jambition needs the Minerva [8] library, and all the libraries Minerva brings along. Minerva can be downloaded here:

`http://plastic.isti.cnr.it/wiki/tools`

In the Minerva archive you find the `minerva.jar`, and several other libraries in its `lib` directory. Copy all these libraries into the `lib` directory of the `Jambition` directory.

Jambition needs access to a constraint solver to do the testing. It uses GNU Prolog [2] for this purpose. It is necessary to install and run the `treeSolver` program, which opens a socket connection to the constraint solver of GNU Prolog (which is not necessary to be downloaded separately). `treeSolver` can be freely downloaded here:

http://www.cs.ru.nl/~lf/tools/treesolver/

The web page provides executables for Windows and Linux. Also the source code is available, so it can be compiled for several other operating systems, see http://www.gprolog.org/#platform. The `treesolver` gets as only input the socket-port to open. Before starting Jambition, invoke `treeSolver` for instance with the port number `60002`. You should get something like:

```
user@pc-plastic:~$ treeSolver 60002
treeSolver V180607
By Lars Frantzen (lars@frantzen.info)
A socket interface to the constraint solver of GNU Prolog
GNU Prolog is copyright (C) Daniel Diaz
Listening on pc-plastic port 60002
```

A tool which is not mandatory for Jambition, but useful to visualize the testing process, is the `Quick Sequence Diagram Editor - sdedit` [6]. It is written in Java and can be freely downloaded here:

http://sdedit.sourceforge.net/

After `sdedit` is launched, turn on its real-time server by choosing in the `Extras` menu `Start/stop RT server`. For the socket-port accept the suggestion `60001`.

Another tool which is not mandatory for Jambition, but useful to visualize the SSM, is `dot` [3]. It is part of the `Graphviz` toolsuite. Binaries for Windows and Linux can be freely downloaded here:

http://www.graphviz.org/

Now Jambition can be launched. Within the directory there is the `Jambition.jar` archive. A simple double click should start Jambition on all common operating systems. To start it from a command-line, do a `java -jar Jambition.jar`. See the screenshot in Fig. 3.

3.2 Configure

Via the `Jambition` menu the `Preferences` window can be displayed, see the screenshot in Fig. 4. We explain the options from top to bottom.

At the top the socket of the `treeSolver` can be modified. The default values here are port `60002` on the local host. The option `Always New Inputs` is experimental, it means that whenever Jambition has to find a solution for a constraint, it tries to find a new solution. For instance, the constraint `quantity` $\geq$ `3` will always be solved by choosing `quantity` $=$ `3`, since this is the first solution found. This may not always be desirable, turning this option on will find the solutions `3`,`4`,`5`, etc. But this generates very huge and inefficient constraints, so this option should be treated with care.

Fig. 3. Jambition Initial Screenshot

Next the `sdedit` socket can be modified. The `Enabled` option allows to turn on and off the usage of `sdedit`. This means that Jambition does not send the testing messages to the `sdedit` tool, which can nicely display those in form of a sequence diagram. Still, it is possible to generate the input for `sdedit` later, for instance when a failure was found. To do so it is necessary to turn on the `Remember Trace` option.

Jambition logs per default to its `Monitor` window. This can be turned off. Note that logging to the `Monitor` increasingly consumes memory since the window remembers all test events. Instead, or additionally, it is possible to log to rotating log files. The name of the file, the bytes per file, and the number of files can be set.

Finally, the `Debug Mode` can be turned on or off. This means that additional information is logged, for instance the exchanged SOAP messages. All options can be reset to its default values, and saved.

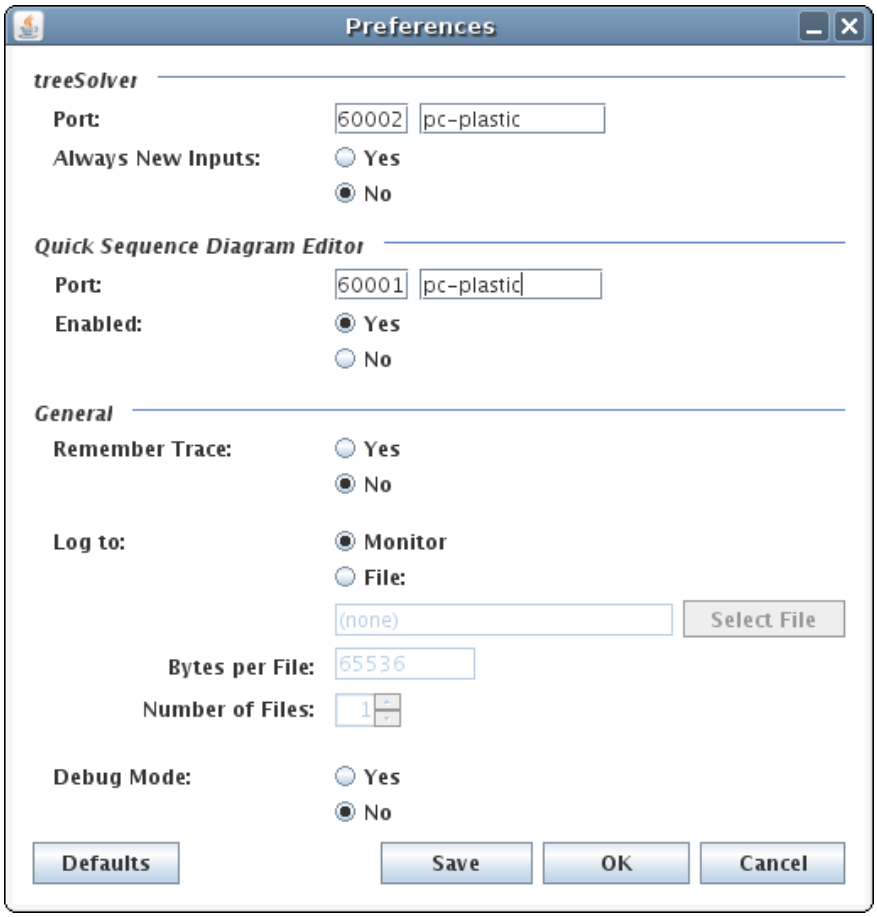

Fig. 4. Jambition Preferences Screenshot

4 Specifying the Web Service

4.1 Web Service Description Language

The structural aspects (data types, messages, operations, port types) of the Web Service to be tested are described in a WSDL instance. Also the binding information is encoded here. Normally, WSDL files are automatically generated by the IDE which is used to develop the Web Service. To define the data types of the message parameters, commonly the data types of XML Schema are used in the WSDL.

XML Schema Simple Types Jambition supports the following simple types:

- xs:boolean
- xs:integer
- xs:string
- xs:enumeration (a restriction on xs:string)
- xs:double

Two limitations are important to note. Firstly, Jambition can only deal with positive integers. This is due to the fact that GNU Prolog (which is used by the `treeSolver`) can only solve constraints over positive integers.

Secondly, the support for doubles is very limited. Constraint solving over double variables is not feasible in the way Jambition needs it. What Jambition does is supporting double variables with a fixed precision. For instance, doubles with a fixed precision of two positions after the decimal point can be used to model amounts of money. Such "fake" doubles can be mapped internally to integers. But this mapping easily leads to large integers, which may be too large again for the constraint solver. In its current version Jambition assumes a fixed precision of two positions after the decimal point.

XML Schema Complex Types Complex types are for instance used to represent classes of Object-Oriented languages in the WSDL. For instance, given a Java method which returns an object of class c. The class c has two fields. If this method is exposed as a Web Service method, the corresponding WSDL can define a complex type for c which is a sequence of two elements representing the two fields. Jambition supports such complex types. Jambition does not support elements in a sequence which are of the kind `maxOccurs = "unbounded"`. Such elements are used to model, for instance, lists.

WSDL Operations Four kinds of operations can be defined in a WSDL:

- request-response
- one-way
- solicit-response
- notification

Jambition does only support the first two kinds — request-response and one-way. We call such a service a *passive* service since it does never send messages actively, only after being requested.

SOAP Binding The XML parameter mapping is defined by the `soap:body use` and the `soap:binding style` attributes. Jambition assumes that:

- `use = "encoded"` or
- `use = "literal"` and `style = "document"` and each message has exactly one part called `parameters`.

4.2 Service State Machines

The behavioral aspects (states of the Web Service, ordering of invocations, constraints on the data flow) are described in an SSM instance. An SSM is a state machine, consisting of typed variables, states, and transitions between the states. Every transition consists of these elements:

1. The state where the transition starts.

2. The name of a WSDL operation.
3. The kind of the corresponding message. This can be either `input` or `output` for request-response operations, and is always `input` for one-way operations. A special kind is `unobservable`, which defines a switch which does not corresponds to any operation, it is taken by the SSM without an external trigger. Thus. it does not send or receive any message.
4. A guard which restricts the conditions under which the transition can be executed. The language of the guard is described below.
5. An update of the variables.

Every SSM has an initial state.

The Language of the Guards There are five kind of *literals*, i.e. constants of a certain type:

- **Integer literal** – a numberstring (e.g. 3423432)
- **Boolean literal** – `true` or `false`
- **Double literal** – numberstring followed by a dot '.' and optional another numberstring (e.g. 23.54)
- **String literal** – string between double quotes, e.g., "Hello World"
- **Enumeration literal** – a string of the form *element@enumname*, where *enumname* is the name of an enumeration type, and *element* one of its elements, e.g. `BOOK@product`

An *identifier* is a name of a variable. If a variable has a complex type, then one can refer to the fields by the dot-notation, e.g. `q.product`. Every literal and identifier are expressions which have a corresponding type, i.e. one of integer, boolean, float, string, complex or enumeration. One can combine literals and identifiers via *operations*, yielding more complex expressions, which again have a type. Next we mention the operations currently supported by the parser. By *number* we mean integers or doubles. Note that here by writing for instance *boolean* or *number* we refer to any expression of this type.

First we mention the operations which lead to an expression of type boolean:

- `==` compares booleans, numbers, strings and enumeration instances for equality
- `!=` compares booleans, numbers, strings and enumeration instances for inequality
- `<` compares numbers for being *less than*
- `<=` compares numbers for being *less than or equal*
- `>` compares numbers for being *greater than*
- `>=` compares numbers for being *greater than or equal*
- `&&` the logical *and* of two booleans
- `||` the logical *or* of two booleans
- `!` the logical *not* of a boolean

Next we mention the operations which lead to expressions of type integer or double:

- `++` increments (+1) an integer

– `--` decrements (-1) an integer
– `+` adds numbers or strings (i.e., a concatenation of two strings)
– `-` subtracts two numbers
– `*` multiplies two numbers
– `%` The remainder operator for numbers
– `/` divides two numbers

A guard is an expression of type boolean. Sometimes one wants to express an empty guard, meaning an expression which has the logical value *always true*. This cannot be done by giving an empty string, nor by writing just the boolean literal `true`, one has to write `true == true` here. One also cannot just mention the name of a variable of type boolean like `p` or `!p`, it is necessary to write `p == true` or `p == false`.

Variable Updates There is another operator we have not mentioned yet, it is the assignment operator '='. Here we assign a value to a variable. An update can consist of several such assignments, but one can also have no assignment at all by giving the empty string "", meaning that all variables remain unchanged. All assignments must be terminated by a semicolon `;`

Designing SSM in MagicDraw One way to design an SSM is via the `MagicDraw` tool. This is explained in the documentation of the Minerva library.

Defining SSM directly in XML Jambition expects an instance of an XML schema for SSM called `SP-SSM`. After giving the the full schema it is further explained below. The schema-file is also part of the Jambition archive.

An XML Schema for SSM

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace=
            "http://frantzen.info/testing/ssmsimulator/schema/"
            xmlns:tns="http://frantzen.info/testing/ssmsimulator/schema/"
            elementFormDefault="qualified">

    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A schema for Singleport Symbolic State Machines (SP-SSM).
            Version: 0.1
            Copyright (C) 2007,2008 Lars Frantzen
        </xsd:documentation>
    </xsd:annotation>
```

```
<xsd:element name="spssm" type="tns:SPssm"></xsd:element>
<xsd:complexType name="SPssm">
    <xsd:sequence>
        <xsd:element name="wsdlURI" type="xsd:anyURI"/>
        <xsd:element name="porttype" type="xsd:token"/>
        <xsd:element name="states">
            <xsd:complexType>
                <xsd:sequence maxOccurs="unbounded">
                    <xsd:element name="state" type="xsd:token"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="initialState" type="xsd:token"/>
        <xsd:element name="variables">
            <xsd:complexType>
                <xsd:sequence maxOccurs="unbounded">
                    <xsd:element name="var" type="tns:Variable"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="switches">
            <xsd:complexType>
                <xsd:sequence maxOccurs="unbounded">
                    <xsd:element name="switch" type="tns:Switch"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Variable">
    <xsd:sequence>
        <xsd:element name="name" type="xsd:token"/>
        <xsd:element name="type" type="xsd:token"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Switch">
    <xsd:sequence maxOccurs="1">
        <xsd:element name="from" type="xsd:token"/>
        <xsd:element name="operationName" type="xsd:token"/>
        <xsd:element name="kind" type="tns:Kind"/>
        <xsd:element name="restriction" type="xsd:string"/>
        <xsd:element name="update" type="xsd:string"/>
        <xsd:element name="to" type="xsd:token"/>
```

```
        </xsd:sequence>
    </xsd:complexType>
    <xsd:simpleType name="Kind">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="input"/>
            <xsd:enumeration value="output"/>
            <xsd:enumeration value="unobservable"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:schema>
```

Explanation of the Schema
We describe its elements:

- `<wsdlURI>` Every SP-SSM instance specifies a port-type which is given in an external WSDL file. This external file is referenced by this tag via an URI. For Jambition, this tag has no relevance.
- `<porttype>` Here the specific port-type from the referenced WSDL is named (there could be several in the WSDL). For Jambition, this tag has no relevance.
- `<states>` Here all the state-names are listed.
- `<initialState>` The name of the initial state is given here.
- `<variables>` Here all the variables are listed.
  - `<name>` Every variable has a name.
  - `<type>` Every variable has a type. This is either one of the XML schema simple types like `xs:boolean`, or the name of a complex type or an enumeration. In case of a complex type or an enumeration the namespace indicator must be the same as the target namespace of the referenced WSDL.
- `<switches>` Here the SSM transitions, called switches, are given.
  - `<from>` This is the state-name where the switch starts.
  - `<operationName>` This is the name of the operation (as defined in the WSDL) where the switch-message belongs to.
  - `<kind>` This is he kind of the message which is transported by this switch. For every operation this can be either `input` or `output`. A special third kind is `unobservable` which denotes an internal message which is not observable at the interface.
  - `<restriction>` This is guard.
  - `<update>` This is the variable update.
  - `<to>` This is the state-name where the switch leads to.

Note that especially when dealing with `<restriction>` and `<update>` elements there may be the need to map special characters like '&' or '<' to a different representation in the schema instances. The next section gives an example instance of this schema.

5 The Warehouse Example

In this section a simple example is given which demonstrates the usage of Jambition. We follow here again the warehouse scenario. We have seen that Jambition needs as input the WSDL file, and an SSM specification, of the Web Service to be tested. Since the WSDL file is usually generated automatically by some programming language specific tool, we start this example by showing the warehouse operations as Java methods. Note that for Jambition it does not matter how the WSDL was generated, by a Java specific tool, another language's tool, or manually.

The Warehouse Web Service in Java We use here the JAX-WS [4] implementation to handle Web Services in Java. We assume the familiarity of the reader with the JAX-WS (or a similar) package, and will not explain its usage further.
The warehouse service shall offer three operations, which appear in Java as methods:

- `Quote checkAvail(QuoteRequest r)`
  This operation allows a user of the warehouse to request a quote for a given product and quantity. This is realized via a parameter of type `QuoteRequest` having the attributes `Product` and `quantity`. We assume the class `Product` to be a simple enumeration, and `quantity` to be an integer. As a return value, this operation shall send a `Quote` object, which consists also of a `Product` and `quantity`; additionally it has the attributes `price` of type double, and `refNumber` of type integer.
- `void cancelTransact(int ref)`
  This operation shall be invoked if the user rejects a quote offered. To do so, the user sends as a parameter the reference number which has been issued in a corresponding quote (see attribute `refNumber` of the `Quote` object in operation `checkAvail` above).
- `void orderShipment(int ref, Address adr)`
  This operation shall be invoked if the user accepts a quote offered. To do so, the user sends also here as a parameter the reference number which has been issued in a corresponding quote, and a shipment `Address` object.

Next we show the signatures of the corresponding Java methods: The complete Java files are also part of the Jambition archive.

```
@WebMethod(operationName = "checkAvail") public Quote
checkAvail(@WebParam(name = "r") QuoteRequest r)

@WebMethod(operationName = "cancelTransact")
@Oneway
public void cancelTransact(@WebParam(name = "ref") int ref)

@WebMethod(operationName = "orderShipment")
@Oneway
public void orderShipment(@WebParam(name = "ref") int ref,
                          @WebParam(name = "adr") Address adr)
```

The auxiliary classes and their attributes are as follows:

```
public class QuoteRequest {
    Product product;
    int quantity;
    ...
}

public enum Product {
    foo,
    bar
}

public class Quote {
    Product product;
    int quantity;
    double price;
    int refNumber;
    ...
}

public class Address {
    String firstName;
    String lastName;
    ...
}
```

The Warehouse WSDL We assume here that the reader has a basic understanding of WSDL files and XML Schema. We have created the Java-files in a package called `services`. Next we generate the corresponding WSDL for the warehouse Web Service, which is one of the input files of Jambition. Important for the further modeling of the service are the operation names, and the names of the operation parameters as they appear in the WSDL. The operations names usually match exactly the Java method names. The parameter names should also match the parameter names given in the Java code (in case of JAX-WS this name can be explicitly set via the `@WebParam` annotation, see above). A special case is the return parameter. In Java this parameter has no name on its own, just a type. For instance, the method `checkAvail` returns a `Quote` object, but this return parameter has no extra name. In the WSDL, also these returned parameters have names.

Let us check the WSDL generated by JAX-WS. We only give the relevant parts, here. First we check the definition of the messages and operations. In a WSDL, every operation consists of messages. Input parameters are modeled by input messages, and the returned parameter is modeled via an output message. In case of a Java `void` operation just an input message is present. In the WSDL this looks like this:

```
  <message name="checkAvail">
    <part name="parameters" element="tns:checkAvail"/>
  </message>
  <message name="checkAvailResponse">
    <part name="parameters" element="tns:checkAvailResponse"/>
  </message>
  <message name="cancelTransact">
    <part name="parameters" element="tns:cancelTransact"/>
  </message>
  <message name="orderShipment">
    <part name="parameters" element="tns:orderShipment"/>
  </message>

  <portType name="Warehouse">
    <operation name="checkAvail">
      <input message="tns:checkAvail"/>
      <output message="tns:checkAvailResponse"/>
    </operation>
    <operation name="cancelTransact">
      <input message="tns:cancelTransact"/>
    </operation>
    <operation name="orderShipment">
      <input message="tns:orderShipment"/>
    </operation>
  </portType>
```

We see here two things. First of all, the names of the operations equal the names of the Java methods (the `<operation name=XYZ">` attribute). Secondly, the definition of the message parameters refers to XSD elements (e.g. `tns:checkAvail`). To find these elements, we next check the definition of the types:

```
  <types>
    <xsd:schema>
      <xsd:import namespace="http://services/"
         schemaLocation="http://localhost:8158/WarehouseService.../>
    </xsd:schema>
</types>
```

Here we see that the WSDL refers to an external XML Schema file which defines the types (the `schemaLocation`). We check that file, next:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:tns="http://services/"
           xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0"
           targetNamespace="http://services/">
```

```
<xs:element name="cancelTransact" type="tns:cancelTransact"/>
<xs:element name="checkAvail" type="tns:checkAvail"/>
<xs:element name="checkAvailResponse" type="tns:checkAvailResponse"/>
<xs:element name="orderShipment" type="tns:orderShipment"/>

<xs:complexType name="checkAvail">
  <xs:sequence>
    <xs:element name="r" type="tns:quoteRequest" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="checkAvailResponse">
  <xs:sequence>
    <xs:element name="return" type="tns:quote" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="orderShipment">
  <xs:sequence>
    <xs:element name="ref" type="xs:int"/>
    <xs:element name="adr" type="tns:address" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="cancelTransact">
  <xs:sequence>
    <xs:element name="ref" type="xs:int"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="quoteRequest">
  <xs:sequence>
    <xs:element name="product" type="tns:product" minOccurs="0"/>
    <xs:element name="quantity" type="xs:int"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="quote">
  <xs:sequence>
    <xs:element name="price" type="xs:double"/>
    <xs:element name="product" type="tns:product" minOccurs="0"/>
    <xs:element name="quantity" type="xs:int"/>
```

```
      <xs:element name="refNumber" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="address">
    <xs:sequence>
      <xs:element name="firstName" type="xs:string" minOccurs="0"/>
      <xs:element name="lastName" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

  <xs:simpleType name="product">
    <xs:restriction base="xs:string">
      <xs:enumeration value="bar"/>
      <xs:enumeration value="foo"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

We find here the names of the parameters, and their corresponding XML-Schema types. Summarized, we get:

- Operation `checkAvail`
  Input message parameters: `r` of type `quoteRequest`
  Output message parameters: `return` of type `quote`
- Operation `cancelTransact`
  Input message parameters: `ref` of type `xs:int`
- Operation `orderShipment`
  Input message parameters: `ref` of type `xs:int` and `adr` of type `address`

The corresponding complex types are:

- `quoteRequest`
  Elements: `product` of type `product` and `quantity` of type `xs:int`
- `quote`
  Elements: `price` of type `xs:double`, `product` of type `product`, `quantity` of type `xs:int`, and `refNumber` of type `xs:int`
- `address`
  Elements: `firstName` of type `xs:string` and `lastName` of type `xs:string`

This is very close to what we have written in the Java code. Still, it is necessary to check the concrete names and types, since this is crucial for the definition of the SSM, as we will see next.

Note: this information can also be automatically extracted and displayed by the *showWSDL* tool. It is part of the Minerva archive.

The Warehouse SSM Now comes the part where we specify the dynamic aspects of the warehouse via an SSM. Remember that an SSM consists of static constituents like types, messages, parameters, and operations. This information is already present in the WSDL file, as we have seen in the last section. Still to be modeled are the dynamic constituents like states, and transitions between the states. SSMs can be seen as a dynamic extension of a WSDL. They specify the legal ordering of the message flow at the service port, together with constraints on the data exchanged via message parameters.

A SSM can store information in SSM-specific variables. Every SSM transition corresponds to either a message sent to the service (input), or a message sent from the service (output), or is unobservable. Furthermore, a transition can be guarded by a logical expression. After a transition has fired, the values of the variables can be updated. Take the WSDL operation `checkAvail`. This request-response operations has an input message with parameter `r` of complex type `quoteRequest`, and an output message `return` of complex type `quote`. In the SSM we model the call of the `checkAvail` operation by two succeeding transitions, the first representing the input message, and the second the output message.

Figure 5 shows the SSM we want to use for this example, which specifies the warehouse service. What we specify here is the session protocol a user of the warehouse service has to follow. Initially, the warehouse is in state `1`. Now a user of the warehouse can invoke

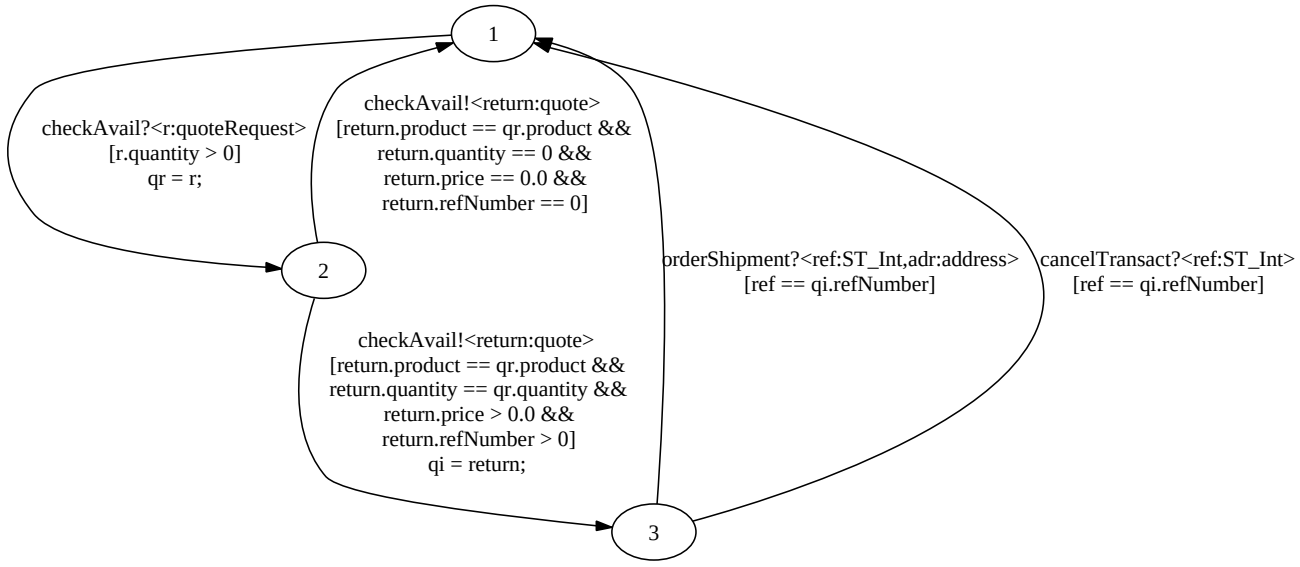

Fig. 5. The Warehouse SSM

the `checkAvail` operation by sending the corresponding input message with parameter `r` of complex type `quoteRequest`. This corresponds to the transition from state `1` to state `2`. The first line of the label is `checkAvail?<r:QuoteRequest>`. This states that this transition refers to the `checkAvail` operation. The appended question mark indicates that here the input message is modeled. The guard of the transition `r.quantity > 0` restricts the

attribute `quantity` of parameter `r` to be greater than zero. After the transition has fired, `r` is saved in the SSM-specific variable `qr` (which must also be of type `QuoteRequest`). Next, the warehouse has to return a `Quote` object via the return parameter `return`. Two things can happen. Firstly, the requested product may not be on stock with the requested quantity. In this case a `Quote` object is returned with a zero `quantity`, `refNumber` and `price` (transition from `2` to `1`). Additionally it is ensured that the returned `product` attribute is the same as the requested one. This is achieved by relating the returned attribute `product` of `return` with the saved requested attribute `product` of `qr`: `return.product == qr.product`. Secondly, if the product is on stock, a `Quote` object is returned with the same `quantity` as being requested, and a `price` and `refNumber` greater than zero (transition from `2` to `3`). We save the issued quote in the SSM-specific variable `qi`. Not that the appended exclamation mark indicates that these two transitions model possible output messages of the `checkAvail` operation. Now again two things can happen. Either the user of the warehouse decides to reject the quote. She invokes the one-way operation `cancelTransact` by sending the message `?cancelTransact` (rightmost transition `3` to `1`). Here she must refer to the correct issued reference number `refNumber`. Or she decides to accept the quote. In this case, in addition to the correct reference number, an address must be provided as a second parameter (leftmost transition `3` to `1`).

Note that in the picture the XML Schema type `xs:int` appears as `ST_Int`. This is due to an automatic mapping of the XML Schema types to internal SSM types when the WSDL is read by Jambition.

Denoting the SSM in XML We denote the SSM now directly in XML, meaning we create an instance of the `SP-SSM` schema which denotes exactly the SSM from Fig. 5. The following SSM XML file is also part of the Jambition archive.

```
<?xml version="1.0" encoding="UTF-8"?>

<spssm xmlns='http://frantzen.info/testing/ssmsimulator/schema/'
       xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
       xmlns:xs="http://www.w3.org/2001/XMLSchema"
       xmlns:package='http://services/'
       xsi:schemaLocation='http://frantzen.info/testing/ssmsimulator/schema/
         file:///home/lf/NetBeans/Audition/.../SP-SSM.xsd'>

    <wsdlURI>not needed</wsdlURI>
    <porttype>not needed</porttype>
    <states>
        <state>1</state>
        <state>2</state>
        <state>3</state>
    </states>
    <initialState>1</initialState>
```

```
<variables>
    <var>
        <name>qr</name>
        <type>package:quoteRequest</type>
    </var>
    <var>
        <name>qi</name>
        <type>package:quote</type>
    </var>
</variables>
<switches>
    <switch>
        <from>1</from>
        <operationName>checkAvail</operationName>
        <kind>input</kind>
        <restriction>r.quantity > 0</restriction>
        <update>qr = r;</update>
        <to>2</to>
    </switch>
    <switch>
        <from>2</from>
        <operationName>checkAvail</operationName>
        <kind>output</kind>
        <restriction>return.product == qr.product &amp;&amp;
         return.quantity == qr.quantity &amp;&amp;
         return.price > 0.0 &amp;&amp; return.refNumber > 0</restriction>
        <update>qi = return;</update>
        <to>3</to>
    </switch>
    <switch>
        <from>2</from>
        <operationName>checkAvail</operationName>
        <kind>output</kind>
        <restriction>return.product == qr.product &amp;&amp;
         return.quantity == 0 &amp;&amp; return.price == 0.0
         &amp;&amp; return.refNumber == 0</restriction>
        <update></update>
        <to>1</to>
    </switch>
    <switch>
        <from>3</from>
        <operationName>cancelTransact</operationName>
        <kind>input</kind>
```

```
            <restriction>ref == qi.refNumber</restriction>
            <update></update>
            <to>1</to>
        </switch>
        <switch>
            <from>3</from>
            <operationName>orderShipment</operationName>
            <kind>input</kind>
            <restriction>ref == qi.refNumber</restriction>
            <update></update>
            <to>1</to>
        </switch>
    </switches>
</spssm>
```

Starting Jambition, treeSolver, sdedit, and Checking the Preferences We now have the necessary specifications ready – the WSDL and the SSM. We assume that the corresponding warehouse Web Service is successfully deployed. We start Jambition and see the main GUI (see Fig. 3). In the `WSDL URL` textfield we enter the URL of the WSDL, and in the `SSM URL` textfield the URL of the SSM. Not that in case of a local file one can use the `file://PathToFile` notation. Since a WSDL can in principle define several services and ports, it is additionally necessary to name the specific service and port to be tested. If unsure what to put here, consult here the WSDL. For our example the relevant WSDL-part looks like this:

```
<service name="WarehouseService">
  <port name="WarehousePort" binding="tns:WarehousePortBinding">
    [...]
  </port>
</service>
```

Hence, we put `WarehouseService` in the `Service` textfield, and `WarehousePort` in the `Port` textfield.

Next we check the preferences by choosing `Preferences` from the `Jambition` menu. For all preferences options we keep the default values. Make sure that the `treeSolver` is running under the port given. Also start `sdedit` and make sure that its real-time server is running on the right port (via its `Extras` menu item `Start/stop RT server...`). Close the preferences.

Generating the SSM Object Now we are ready to generate the SSM object. We do so by pressing the `Generate SSM` button. After a short while a little notification window should report `SSM successfully generated!`. The Monitor of the Jambition GUI should report something similar to:

```
08/10/07 11:54:10:161: treeSolver is bound to local port 59132
08/10/07 11:54:10:283: treeSolver sends: treeSolver V180607 ready
08/10/07 11:54:10:285: qsdeServer is bound to local port 56311
08/10/07 11:54:10:451: STSimululator ready.
```

The `treeSolver` reports something like `Connected with 127.0.0.1`, and `sdedit` shows in a new tab called `Jambition` the two actors `Jambition` and `SUT` (which is the Web Service Under Test).

Using `dot` to Visualize the SSM Note that after having successfully generated the SSM object, the button `dot Input` is enabled. Pressing this button outputs the generated SSM in a textual format understood by the `dot` tool from the `Graphviz` toolsuite. For our example, we get this output in the Jambition Monitor:

```
digraph STS {
1 -> 2 [label = "checkAvail?<r:quoteRequest[product:product{bar,foo},
        quantity:ST_Int]>\n[r.quantity > 0]\nqr = r;"];
2 -> 1 [label = "checkAvail!<return:quote[price:ST_PseudoPosDouble,
        product:product{bar,foo},quantity:ST_Int,refNumber:ST_Int]>\n
       [return.product == qr.product &&\n return.quantity == 0 &&\n
        return.price == 0.0 &&\n return.refNumber == 0]\n"];
2 -> 3 [label = "checkAvail!<return:quote[price:ST_PseudoPosDouble,
        product:product{bar,foo},quantity:ST_Int,refNumber:ST_Int]>\n
        [return.product == qr.product &&\n return.quantity == qr.quantity
        &&\n return.price > 0.0 &&\n return.refNumber > 0]\nqi = return;"];
3 -> 1 [label = "orderShipment?<ref:ST_Int,adr:address[firstName:ST_String,
        lastName:ST_String]>\n[ref == qi.refNumber]\n"];
3 -> 1 [label = "cancelTransact?<ref:ST_Int>\n[ref == qi.refNumber]\n"];
}
```

Now one can copy-paste this into a text editor and save it as a file. Running the `dot` tool on it generates a graph representing the SSM. Figure 5 shows a slightly simplified version of the generated graph where the details of the complex types have been removed.

Testing the Warehouse Service Finally we are in the position to test if the warehouse service does conform to the SSM specification we have developed. Let us recapitulate what we have done so far. The initial assumption in this running example was that we have developed a warehouse Web Service in the Java programming language. To access this service the `JAX-WS` package automatically generates a corresponding WSDL file which specifies the operations available at the service's port, and its physical address (also called the *endpoint*) of the service.

To specify dynamic aspects of the service we have additionally modeled an SSM. This SSM specifies the legal ordering in which the warehouse operations must be invoked, and restricts the data which is exchanged via the operations. On the one hand, the SSM specifies

a protocol for a potential user of the warehouse. For instance, it postulates that firstly the `checkAvail` operation must be called with a positive quantity. On the other hand, the SSM specifies a protocol to which the warehouse service itself must conform. For instance, the quote returned by the `checkAvail` operation must always deal with the same product that being requested.

In other words, the SSM deals with two actors. One is the user of the warehouse invoking its operations. The other is the warehouse itself receiving and responding to the operation calls. When testing the warehouse, Jambition is playing the role of the user of the warehouse. Initially, being in state `1`, the only specified call is `checkAvail`. To invoke this operation, a parameter of type `quoteRequest` must be constructed. The guard additionally restricts the field `quantity` to be greater zero (see transition from state `1` to `2`). Jambition will randomly choose a product (`foo` or `bar`). For the quantity, the first solution found is `quantity = 1`. Jambition will always set the quantity to `1` unless the option `Always New Inputs` is set in its preferences. In that case it will first choose a quantity of `1`, then `2`, and so on (this does only work for a limited number of steps in the current version). With the constructed `quoteRequest` object it then invokes the `checkAvail` operation on the service. Now, being in state `2`, Jambition receives the quote object returned by the warehouse. The SSM dictates, that this quote has to have a zero quantity, price, and reference number (transition from `2` to `1`), or the same quantity as requested and a positive price and reference number (transition from `2` to `3`). For both transitions must hold that the quote's product must be the same as the requested product. Jambition checks now, which of the two cases holds for the received quote, and moves to the respective next state (`1` or `3`). If none holds, a failure has been detected. For instance, this is the case if the returned quote deals with a different product than requested. Or, if the quantity and price are zero, but the reference number is not. Assuming the quote was correct and Jambition moved to state `3`, then it will next send either a `cancelTransact` or an `orderShipment` to the warehouse (taking care of using the correct reference number). And so on.

While no failure is found, the testing continues. Jambition reports the currently achieved state coverage (called location coverage) and transition coverage (called switch coverage) in its window. If enabled, `sdedit` shows the messages exchanged in realtime. Figure 6 shows a possible output of `sdedit` while testing the warehouse. The user can halt the testing by pressing `Stop Testing`. Doing so enables the `Detailed Coverage` button. Pressing it, detailed coverage information of the SSM are displayed (which state and which transition has been visited how often up to now). The testing can be continued by pressing `Continue Testing`, or ended by pressing `End Testing`.

6 Appendix

6.1 The Dumont Grammar in BNF

```
SwitchRestriction ::= BooleanExpression <EOF>
    UpdateMapping ::= ( VarAssignment )* <EOF>
    VarAssignment ::= Id "=" TermExpression ";"
```

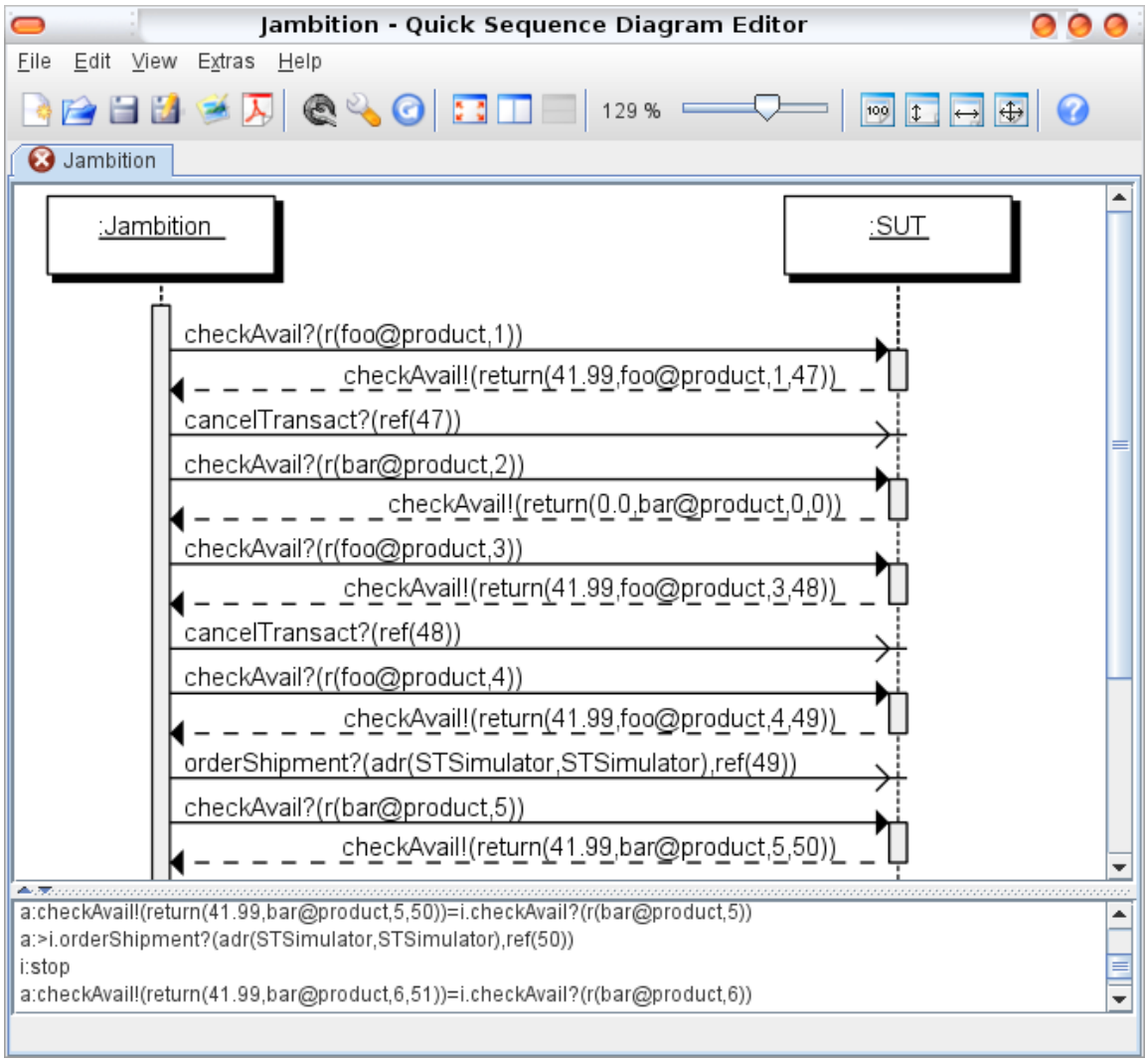

Fig. 6. The Quick Sequence Diagram Editor

```
    BooleanExpression ::= LogicalOrExpression ( "&&" LogicalOrExpression )*
   LogicalOrExpression ::= LogicalNotExpression ( "||" LogicalNotExpression )*
  LogicalNotExpression ::= "!" LogicalNotExpression
                         |  TermExpression TermEqualityExpression
                         |  "(" BooleanExpression ")"
TermEqualityExpression ::= "==" TermExpression
                         |  "!=" TermExpression
                         |  "<=" TermExpression
                         |  "<" TermExpression
                         |  ">=" TermExpression
                         |  ">" TermExpression
        TermExpression ::= TermAddExpression
```

```
     TermAddExpression ::= TermSubtractExpression
                           ( "+" TermSubtractExpression )*
TermSubtractExpression ::= TermMultExpression ( "-" TermMultExpression )*
    TermMultExpression ::= TermDivExpression ( "*" TermDivExpression )*
     TermDivExpression ::= TermModExpression ( "/" TermModExpression )*
     TermModExpression ::= TermUnaryExpression ( "%" TermUnaryExpression )*
   TermUnaryExpression ::= "++" TermUnaryExpression
                         |  "--" TermUnaryExpression
                         |  "(" TermExpression ")"
                         |  PrimaryTermExpression
 PrimaryTermExpression ::= Literal
                         |  Id
                  Name ::= <IDENTIFIER> ( "." <IDENTIFIER> )*
                    Id ::= Name
               Literal ::= ( <INTEGER_LITERAL> )
                         |  ( <BOOLEAN_LITERAL> )
                         |  ( <STRING_LITERAL> )
                         |  ( <ENUMERATION_LITERAL> )
```

References

1. GlassFish Application Server homepage. https://glassfish.dev.java.net.
2. GNU Prolog homepage. http://www.gprolog.org/.
3. Graphviz homepage. http://www.graphviz.org/.
4. JAX-WS homepage. https://jax-ws.dev.java.net/.
5. MagicDraw homepage. http://www.magicdraw.com.
6. Quick Sequence Diagram Editor homepage. http://sdedit.sourceforge.net/.
7. treeSolver homepage. http://www.cs.ru.nl/~lf/tools/treesolver/.
8. PLASTIC tools homepage. http://plastic.isti.cnr.it/wiki/tools.