JSXM Manual

Dimitris Dranidis

Computer Science Department
CITY College
An International Faculty of the
University of Sheffield

Version: v1.2
Last update: 20 May 2010

Contents

Major changes from version 1.1

1. Support for XSD types:
   (a) Standard XSD types supported (e.g. xs:int, xs:string, etc.) for defining the types of the input and output arguments.
   (b) User defined types are defined using XML schema definition (XSD) files. User defined XSD types are transformed to Java types using JAXB binding.
2. The XML syntax for specifying the inputs and outputs has been changed and is now compatible to XSD typing.
3. Ant task provided for test execution: runjunit

1 Introduction to JSXM

1.1 What is JSXM?

JSXM is a model-based testing tool. The tool is implemented in Java and allows the automated generation of test cases based on a model of the software to be tested. The model is expressed as a Stream X-machine (SXM). With the help of JSXM the specified model can be used for:

- Model Animation
- Test Generation
- Test Transformation

1.1.1 Model Animation

Animation of the model means execution of the model or simulation of the modeled software. Interactive or batch animation allow the model designer to validate the specification, i.e. ensure that the correct functionality is modeled.

1.1.2 Test Generation

Once a model is created, the model can be used for the automated generation of test cases. The generated test cases are in XML format and they are independent of the technology or programming language of the implementation. This means that the implementation under test can be an application in any programming language or implementation platform (for instance the application could be a Web service).

The test generation algorithm is based on the testing theory of Stream X-machines. More details about the SXM testing theory can be found in [2, 1].

1.1.3 Test Transformation

Since the generated test cases are independent of the underlying technology of the implementation under test, the tests cannot be directly used for testing. A Test Transformer is responsible for transforming the general test cases to concrete test cases. It is clear that different Test Transformers are required for different programming languages or different software technologies.

Currently, a Java Test Transformer is available which generates JUnit test cases for the automated testing of Java applications.

2 First steps

2.1 Installing JSXM

1. Install Ant. Ant automates the processes of compilation, test generation and test transformation with the provided build files. You can download and install Ant from `http://ant.apache.org/`
2. Extract the zipped file in a folder. Let us assume that this folder is C:\JSXM. (Make sure that the directory you install has no spaces in its name).
3. Check the contents:
   (a) lib\jdom.jar
   (b) lib\junit.jar
   (c) lib\jaxb-impl.jar
   (d) lib\jaxb-xjc.jar
   (e) lib\jsxm.jar
   (f) build.xml (ANT build file)
4. Set the environment variable JSXM to point to the installation folder.

In Windows shell you can execute:
```
set JSXM=C:\JSXM
```
In UNIX shell you can execute (assuming JSXM is located at /home/user/):
```
export JSXM=/home/user/JSXM
```

2.2 Checking the installation

Change to the Book_Borrower directory in the examples and either run (with ANT):

```
ant -Dname=Book
```

or

```
java core.SXMBaseGenerator -path temp\ -overwrite Book.xml
javac -d temp temp\*.java
```

2.3 Executing the examples

In the following sections all the components of JSXM are introduced with the help of the existing examples. We suggest to try out the examples before proceeding with writing your own specifications (Section 6.3.1).

2.4 Filename conventions

Assuming *specName* is the name of the specification model, the following conventions are used for naming files:

- The specification of the model should be stored in a file: *specName*.xml
- The name attribute of the SXM element in *specName*.xml should be *specName*
- The generated Java files have the name *specName*SXM (actually types. *specName*SXM)
- The test generation searches for the file *specName*_sets.xml and generates a file called *specName*_test.xml

Although not necessary, it is advised to name input and output files for batch animation with the name of the specification and the suffix inputs/outputs respectively.

Other files:
Definitions of prototype SXM instances are stored in a file called definitions.xml.

3 Compiling the specification

Required files:

- The specification in an XML file: *specName*.xml
- A folder named temp (this is created for you if you use ant)
- An ant build.xml file (if you use ant).

Generated files:

- Java files are generated and compiled to class files in the temp folder

It is assumed that the build.xml is located in the current folder (you can copy build.xml from the root JSXM folder)**.** The following command generates and compiles all necessary Java files. Java files and the corresponding class files are kept in the folder temp:

```
ant -Dname=specName
```

Example:

```
ant -Dname=Book
```

3.1 Notes about the compilation

The temp folder stores all the necessary Java source and class files for animating the specifications and generating the tests. These files should not be manually edited. If the compilation of files reveals errors you should edit the SXM specification and not the generated Java files.

4 Animating the SXM specification

The animator can be executed in two modes:

- Batch mode
- Interactive mode

Note that you can directly invoke the animator and compilation will also take place.

4.1 Executing the animator in batch mode

Required files:

- The inputs to the SXM: *inputfile*.xml

Generated files:

- The outputs of the SXM: *outputfile*.xml

```
ant banimate -Dname=specName
```

Example:

```
ant banimate -Dname=Book
```

The default input/output files for batch animation are: *specName*_input.xml and *specName*_output.xml. These files can be set by using the ifile and ofile properties of the build file:

```
ant banimate -Dname=Book -Difile=input.xml -Dofile=output.xml
```

4.2 Executing the animator in interactive mode

```
ant animate -Dname=specName
```

Example:

```
ant animate -Dname=Book
```

4.2.1 Using the animator in interactive mode

The animator creates the machine to be animated, resets the machine to the initial state and initial memory, and then waits for some input to be entered at the input prompt:

```
Input :
```

Instead of an input, a command can be entered at the input prompt. Entering .h. displays the available commands:

```
Input :  .h.
******************************************************
.r.  resets the SXM to its initial state and memory.
.i.  lists available inputs.
.q.  quits the animator.
.h.  displays this list.
******************************************************
```

An example of an interactive animation can be found in the Appendix.

Note that there is a line return after the "Input" prompt but this does not effect the animation.

5 Generating Test Cases

Required files:

- The specification in an XML file: *specName*.xml
- A file defining sets: the characterization set and the state cover set in the file: *specName*_sets.xml
- A file containing prototype definitions in the case of Sxm inputs: definitions.xml
- The ANT build.xml

Generated files:

- The test file: *specName*_test.xml
- A log file: out

The following command generates test cases after generating and compiling all necessary Java files. Java files and the corresponding class files are kept in the folder temp. Note that it is not necessary to compile the specification first:

```
ant -Dname=specName tests
```

Example:

```
ant -Dname=Book tests
```

The test generation creates (overwrites) the specName_test.xml file. The output of the execution and any errors are stored in the out text file.

Note that the default k value is 2. A different k value can be set by executing:

```
ant -Dname=specName -Dk=3 tests
```

Example:

```
ant -Dname=Book -Dk=3 tests
```

5.1 Generated test cases

The following is an extract of the file with the generated test cases:

```
<?xml version="1.0" encoding="UTF-8"?>
<set k="3">
  <definitions />
  <sequences>
    <sequence name="releasePF">
      <call>
        <function name="releasePF" />
        <input name="release" />
        <output name="release_Error" />
      </call>
    </sequence>
    <sequence name="bookIsNotAvailablePF">
      <call>
        <function name="bookIsNotAvailablePF" />
        <input name="isAvailable" />
        <output name="isAvailableOut">
          <result type="xs:boolean">true</result>
        </output>
      </call>
    </sequence>
    <sequence name="setBorrowedPF_bookIsAvailablePF">
      <call>
        <function name="setBorrowedPF" />
        <input name="setBorrowed" />
        <output name="setBorrowedOut" />
      </call>
      <call>
        <function name="bookIsAvailablePF" />
        <input name="isAvailable" />
        <output name="isAvailableOut">
          <result type="xs:boolean">false</result>
        </output>
      </call>
    </sequence>
...
  </sequences>
</set>
```

The top element set has as an attribute the number k used for the test generation. The set consists or definitions used in the test generation and a list of sequences of inputs-outputs. Each sequence consists of one or more calls. The name of the sequence lists the sequence of the processing functions which must be triggered. Each call has the following structure:

- The function is the processing function which must be triggered.
- The input is the necessary input to trigger the function. The input may be complex and can contain one or more arguments.
- The output is the produced output by the SXM oracle, i.e. the expected output. The output may be complex and can contain one or more results.
- Each argument or result is characterized by its type, its name, and its value.

The definitions define prototypes which are necessary when the test cases involve arguments of type Sxm. More details about SXM definitions may be found in Section 7.4.2

6 Transforming Test Cases to JUnit

The test cases generated by the JSXM test generation are abstract, i.e. language-independent; they are not specific to any implementation language. These abstract test cases can be used to generate concrete test cases which can be executed to test a system under test in a specific language. A transformation process is necessary to transform the abstract test cases to concrete test cases.

A JUnit transformer is implemented which transforms the abstract test cases to JUnit test cases in order to test Java programs. Let us assume that the Java class under test has the same name as the specification: *specName*.java.

Required files:

- The abstract test cases xml file: *specName*_test.xml

Generated files:

- The JUnit test file: *specName*AdapterTest.java

  The generated JUnit java class intends to test a Java adapter class: *specName*Adapter.java. The purpose of the adapter is to wrap all method calls of the class under test and to make them return outputs compatible to the outputs of the specification.

The following command generates JUnit test cases after generating and compiling all necessary Java files and generating the abstract test cases. The generated JUnit file is stored in the junit folder:

```
ant junit -Dname=specName
```

Example:

```
ant junit -Dname=Book
```

The generated JUnit test class is called BookAdapterTest.java.

6.1 Requirements for Java files under test

- There must exist a default constructor. The JUnit test creates objects using the default constructor.
- All paths of all methods should either return a value or throw an exception. Different failure paths should throw different exceptions (observability).

As an example we present a possible implementation of the Book Java class:

```
public class Book{

  private boolean available = true;

  public boolean isAvailable() {
    return available;
  }

  public void setBorrowed() {
    if ( available ) {
      available = false;
    }
    else
      throw new RuntimeException();
  }

  public void release() {
    if (! available ) {
      available = true;
    }
    else
      throw new RuntimeException();
  }
}
```

6.1.1 Example of a generated JUnit file

In the following Java code one can see two of the generated test methods. The name of the method consists of the k value, the number of the test and the function(s) called. Each assertion compares the actual output of the adapter with the expected output generated by the animation of the specification. Note that complex outputs, such as isAvalaibleOut(true), have been serialized to strings as isAvailableOut_true.

```
import junit.framework.TestCase;

public class BookAdapterTest extends TestCase {

    public void test_k3_1_releasePF() {
        BookAdapter obj = new BookAdapter();
        assertEquals("release_Error", obj.release());
    }

    public void test_k3_2_bookIsNotAvailablePF() {
        BookAdapter obj = new BookAdapter();
        assertEquals("isAvailableOut_true", obj.isAvailable());
    }

    public void test_k3_3_setBorrowedPF_bookIsAvailablePF() {
        BookAdapter obj = new BookAdapter();
        assertEquals("setBorrowedOut", obj.setBorrowed());
        assertEquals("isAvailableOut_false", obj.isAvailable());
    }
...
//
// Helper methods for definitions...
//


}
// End of generated Test Case
```

The adapter should convert all results of successful method execution or any exceptions thrown to serialized string outputs. In the following we present an example of the adapter for the Book Java class. Note how the boolean result of the isAvailable method is serialized to a string containing the method call and the result. Also note how the exception is converted to the string containing an error as in the specification.

```
public class BookAdapter {
  private Book book = new Book();

  public String isAvailable() {
    return "isAvailableOut_"+ book.isAvailable();
  }

  public String setBorrowed() {
    try {
      book.setBorrowed();
    } catch (Exception e) {
      return "setBorrowed_Error";
    }
    return "setBorrowedOut";
  }

  public String release() {
    try {
      book.release();
    } catch (Exception e) {
      return "release_Error";
    }
    return "releaseOut";
  }
}
```

6.2 Executing the JUnit tests

Assuming you have placed the Java under test class and its adapter in the java folder, compile them:

```
javac java\*.java
```

Required files:

- The ANT build.xml
- The generated JUnit java file.

The following command executes the JUnit tests generated by the previous step. Note that there is no dependency to test generation, which means that the test cases need to be generated with the junit ANT task beforehand.

```
ant runjunit -Dname=specName
```

Example:

```
ant runjunit -Dname=Book
................................
Time:  0
OK (34 tests)
```

7 JSXM specification

The SXM pecifications for JSXM are written in XML with some inline Java code in some elements. In the following DTD fragments are used for the presentation of the XML syntax of the specification.

The *SXM* element is the top element. The attribute *name* is the name of the specification. All other elements are children of the *SXM* element:

```
<!ELEMENT SXM ( states, initialState, transitions, memory, inputs, outputs, functions,
definitions?, testinputgeneration?  )  >
<!ATTLIST SXM name NMTOKEN #REQUIRED >
```

The *initialState* element defines the initial state. The initial state must be one of the states defined in the states element.

```
<!ELEMENT initialState EMPTY >
<!ATTLIST initialState state IDREF #REQUIRED >
```

The *states* element has as children all the *state* elements. Each *state* has an attribute *name*.

```
<!ELEMENT states ( state+ ) >
<!ELEMENT state EMPTY >
<!ATTLIST state name ID #REQUIRED >
```

Each *transition* is defined starting at a state (*from*) and going to a state (*to*) and is triggered by a function (*function*). The *to* and *from* attributes each refer to a state. The *function* attribute refers to a function.

```
<!ELEMENT transitions ( transition+ ) >
<!ELEMENT transition EMPTY >
<!ATTLIST transition from IDREF #REQUIRED >
<!ATTLIST transition function IDREF #REQUIRED >
<!ATTLIST transition to IDREF #REQUIRED >
```

The *memory* element has four children:

- The *javaImports* element contains Java code which defines all necessary import statements.
- The *declaration* element contains Java code which defines all necessary memory variables.
- The *initial* element contains Java code which initializes all variables declared in the memory element.
- The *display* element contains a Java expression used by the animator to display the memory contents. The Java expression in the display element should evaluate to a string.

```
<!ELEMENT memory ( javaImports, declaration, initial, display ) >
   <!ELEMENT javaImports ( #CDATA)>
   <!ELEMENT declaration ( #CDATA)>
   <!ELEMENT display ( # CDATA)>
   <!ELEMENT initial ( # CDATA)>
```

The *inputs* element has as children all the *input* elements. Each *input* element defines a SXM input. An input has a *name* and may have zero or more arguments (*arg*) elements. An argument of an input is specified by two attributes: its *name* and its *type*. Note that these types are not Java types but JSXM types. Refer to Section 7.1 for the available JSXM types.

```
<!ELEMENT inputs ( input+ ) >
<!ELEMENT input ( arg* ) >
<!ATTLIST input name ID #REQUIRED >
<!ELEMENT arg EMPTY >
<!ATTLIST arg name NMTOKEN #REQUIRED >
<!ATTLIST arg type NMTOKEN #REQUIRED >
```

Outputs are structured similarly to inputs. Each output has a name and my have zero or more results. Each result (similarly to arguments) has a name and a (JSXM) type.

```
<!ELEMENT outputs ( output+ ) >
<!ELEMENT output ( result* ) >
<!ATTLIST output name ID #REQUIRED >
<!ELEMENT result EMPTY >
<!ATTLIST result name NMTOKEN #REQUIRED >
<!ATTLIST result type NMTOKEN #REQUIRED >
```

The *functions* element consists of the *function* declarations. A *function* has three attributes: its *name*, the *input* it is triggered by and the *output* it produces. A function may have a *precondition* and an *effect*:

- A processing function may have a *precondition*. The precondition contains a Java boolean expression over memory variables and input arguments. The function is triggered only if the Java expression evaluates to true.
- A processing function may have an *effect*. The effect contains Java statements over memory variables, input arguments and output results. The Java statements in the effect are executed when the precondition evaluates to true.

```
<!ELEMENT functions ( function+ ) >
   <!ELEMENT function ( effect?  , precondition?  )  >
   <!ATTLIST function name ID #REQUIRED >
   <!ATTLIST function input IDREF #REQUIRED >
   <!ATTLIST function output IDREF #REQUIRED >
   <!ELEMENT effect ( #CDDATA ) >
   <!ELEMENT precondition ( #CDDATA ) >
```

Refer to Section 7.2 for more details about function declarations.

The definitions element is optional. It is only necessary if an input has an argument of type Sxm. Each defsxm element defines a SXM by providing a name, the type of the SXM and optionally the inputs provided to the machine in order to drive the machine to a specific state and memory. The definitions are used both for animation and test generation. Refer to Section 7.4.2 for more details about SXM definitions.

```
   <!ELEMENT definitions ( defsxm+ ) >
   <!ELEMENT defsxm ( input* ) >
   <!ATTLIST defsxm name NMTOKEN #REQUIRED >
   <!ATTLIST defsxm type NMTOKEN #REQUIRED >
```

Finally the testinputgeneration element is optional and it is used for the generation of tests. The element has as children a different inputgenerator for each function in the specification that receives complex inputs (inputs with argument values). The inputgenerator contains Java code which is executed for setting the arguments of inputs. Refer to Section 7.3 for more details about input generators.

```
   <!ELEMENT testinputgeneration ( inputgenerator+ ) >
   <!ELEMENT inputgenerator (#CDDATA ) >
   <!ATTLIST inputgenerator function IDREF #REQUIRED >
```

7.1 JSXM types: XSD

JSXM currently supports the following XSD basic types:

- **xs:int**: integer type. The corresponding Java type is Integer.
- **xs:boolean**: boolean type. The corresponding Java type is Boolean.
- **xs:string**: string type. The corresponding Java type is String.
- **xs:decimal**: decimal type. The corresponding Java type is BigDecimal.

Special types:

- **sxm**: A Stream X-machine type. Values for the arguments of type Sxm are any SXM model instances defined in the definitions element of a specification or an external definitions.xml file.

7.1.1 User-defined types

New types can be defined by providing their XSD definitions. More details can be found in Section 8.

7.2 Function definitions

The following extract of a specification illustrates the basic concepts of a JSXM processing function definition.

```
<transitions>
  <transition from="normal" function="withdraw0" to="opened" />
...
</transitions>

<memory>
  <declaration>
    int balance
  </declaration>
  <initial>
    balance = 0
  </initial>
</memory>

<inputs>
...
  <input name="withdraw"><arg name="amount" type="xs:int" /></input>
</inputs>
<outputs>
...
  <output name="withdrawOut"><result name="amount" type="xs:int" /></output>
</outputs>

<functions>
  ...
  <function name="withdraw0" input="withdraw" output="withdrawOut">
    <precondition>
      withdraw.get_amount() > 0 &amp;&amp; balance == withdraw.get_amount()
    </precondition>
    <effect>
      balance = balance - withdraw.get_amount();
      withdrawOut.amount = withdraw.get_amount();
    </effect>
  </function>

  <function name="readbalance" input="getBalance" output="getBalanceOut">
    <effect>
      getBalanceOut.amount = balance;
    </effect>
  </function>

</functions>
```

The memory of the SXM consists of a single integer variable called balance which is initialized to zero.

The processing function *withdraw0* labels a transition from the state *normal* to the state *opened*.

The function may be triggered by a withdraw *input* and produces a *withdrawOut* output. Note that both input and output are complex (i.e. have arguments or results).

The function is triggered when the precondition evaluates to true: when the amount to be withdrawn is positive and equal to the balance. Thus the processing function represents the situation in which all the money are withdrawn from the account (withdraw0). Note that the syntax:

$$inputName.\text{get}\_argName()$$

is used to access the argument of an input as in the example:

withdraw.get_amount()

Furthermore notice how the Java boolean "and" operator is expressed as: "&amp;&amp;" instead of the usual "&&", since the character "&" is not directly allowed in XML documents.

If the precondition is true the effect is executed: The balance is decreased by the amount withdrawn and the output of the processing function is set to the withdrawn amount. Note that for setting a result of an output the syntax is simpler:

$$outputName.resultName$$

as in the example:

withdrawOut.amount

Arguments of inputs are treated differently since their values should not be modified. For each argument an accessor get The Boat That Rocked

method is defined and must be used in order to access the value. Output results, on the other hand, are directly accessible fields and are usually modified in the effect part of a processing function.

7.3 Test Input generators

Input generators are used during the test generation process. Each function which receives a complex input (having one or more arguments) needs an input generator.

```
<testinputgeneration>
  <inputgenerator function="deposit">
    deposit.set_amount(5);
  </inputgenerator>
  <inputgenerator function="withdrawN">
    if (balance != 1) withdraw.set_amount(1);
  </inputgenerator>
  <inputgenerator function="withdraw0">
    withdraw.set_amount(balance);
  </inputgenerator>
</testinputgeneration>
```

The Java fragment inside the *inputgenerator* element is executed whenever a value is needed to be set for the arguments of the input. As the example of withdrawN and withdraw0 illustrate the values usually depend on the current memory. In the case of withdraw0 the amount (1st argument) of the withdraw input needs to be set equal to the balance.

Two ways are supported for setting the inputs:

The syntax used in the example uses setter methods which are provided, such as the set_amount method:

withdraw.set_amount(balance);

The alternative syntax:

setArg($argNum$, $valueAsAString$)

is used for setting the ($argNum$+1)th argument of the input to the value corresponding to the *valueAsAString* expression. The conversion of *valueAsAString* to a value is performed by the *parseType* method of the corresponding JSXM type (see Section 7.1.1 for more details). The previous example is as follows:

setArg(0, ""+balance);

7.3.1 The test generation process

The test generation process consists of two phases:

- In the first phase a set of processing function sequences is generated. The execution of these sequences and the comparison of the expected outputs to the actual outputs guarantees the equivalence of the specification to the implementation.
- In the second phase for each sequence of processing functions a sequence of inputs should determined which will force the execution of every functions in the sequence.

For each prefix of the function sequence there might be more than one input sequence which may be used. Thus, in the general case, one needs to select a specific input prefix and then incrementally extend this until a complete input sequence which triggers the complete function sequence is found. In some cases, when it is not possible to extend the input sequence, previous prefix sequence choices need to be rejected and other prefixes need to be selected (backtracking). This makes the input generation process an expensive and time-consuming process. Uniform specifications allow an easier and faster input sequence selection algorithm.

In the case of uniform specifications [3] any input prefix may be chosen in order to incrementally build the whole input sequence. Essentially this reduces to the process of selecting the inputs for the next processing function without caring about what follows (any input which triggers the current function will do). This is how the input generators in JSXM work. The selection of the input is based only on the knowledge of the current memory.

The Account is an example of a non-uniform specification. The knowledge of the current memory is not enough for deciding the input value. To demonstrate that assume we have to generate inputs for the following function sequence:

<open, deposit, withdrawN, withdrawN, withdrawN, withdrawN, withdrawN>

The current input generator will generate the following input sequence

<open, deposit(5), withdraw(1), withdraw(1), withdraw(1), withdraw(1)>

and it will fail to generate an input for the last withdrawN since the withdrawal of 1 will fail in the precondition of withdrawN; actually the last withdraw(1) input will trigger the withdraw0 processing function instead of the intended withdrawN. The problem is worse if we choose a smaller value for the first deposit argument. We manage to avoid this problem with values for k as large as 3 since the longest generated sequences of withdrawNs after a deposit are:

<open,deposit,withdrawN,withdrawN,withdrawN,withdrawN,open>
<open,deposit,withdrawN,withdrawN,withdrawN,withdrawN,close>
<open,deposit,withdrawN,withdrawN,withdrawN,withdrawN,deposit>

The prefix <open,deposit> is member of the state cover and reaches the state normal, the sequence of the next four withdrawNs is the result of an exploration in depth k+1 of the functions, and the suffixes <open>, <close> and <deposit> are the members of the characterization set. A higher k would require a different input generator for deposit, as for example: deposit(6).

7.4 Specifications for interacting SXMs

A special type of argument is Sxm whose value is a SXM instance. A processing function receiving input with a Sxm argument may send inputs to another SXM. That way the interaction between two machines is achieved. The interaction resembles the object-oriented model of method invocation: an object may send a message to another object if it knows its identity. The message as usually is a complex input with arguments. The returned value can be accessed by accessing the output message ot the machine.

In the following example we present the memory, a processing function and its corresponding input from a Borrower specification. The borrower may borrow a book if a book is available. A Book specification specifies the behavior of a book (borrowing, returning, checking availability).

```
<memory>
  <declaration>
    BookSXM book;
  </declaration>
  <initial>
    book = null;
  </initial>
</memory>

<inputs>
  <input name="borrowBook" >
    <arg name="book" type="sxm"/>
  </input>
...
</inputs>

<functions>
  <function name="borrowBookPF" input="borrowBook" output="borrowBookOut">
    <precondition>
      <!--
        borrowBook.book is an Sxm.
        Send the input isAvailable() to the Sxm.
        From the response get the value of the part named "result".
      -->
        ((BookSXM) (borrowBook.get_book())).isAvailable().result;
    </precondition>
    <effect>
      book = (BookSXM) borrowBook.get_book();
      book.setBorrowed();
    </effect>
  </function>
</functions>
```

Note that the argument to the borrowBook input is of type Sxm, a reference to a SXM instance.

The precondition of the borrowBookPF is:

```
<precondition>
  ((BookSXM) (borrowBook.get_book())).isAvailable().result;
</precondition>
```

Recall that borrowBook.get_book() returns the argument book of the borrowBook input. This argument is a SXM instance of type Sxm (corresponding to a Java SXM abstract class). We need to downcast the general SXM type to the specific SXM type of the receiver: BookSXM. Then, the input isAvailable() is sent to the BookSXM machine and the output is read by accessing the result property. So, in a way, the function asks the book whether it is available. The result in this case is a boolean value as it is shown in the definition of the isAvailableOut output in the Book specification:

```
<output name="isAvailableOut">
  <result name="result" type="xs:boolean" />
</output>
```

The effect of the borrowBookPF is:

```
<effect>
  book = (BookSXM) borrowBook.get_book();
  book.setBorrowed();
</effect>
```

The memory variable is set to the book argument of the borrowBook input. Both are of type BookSXM. Then the input setBorrowed() is sent to the book SXM so that the book changes its state to borrowed.

7.4.1 Preconditions should have no side-effects

Another processing function of the same specification describes the situation in which the book to be borrowed is not available:

```
<function name="borrowBookNotAvailablePF" input="borrowBook" output="borrowBook_NotAvailable">
  <precondition>
    !((BookSXM) (borrowBook.get_book())).isAvailable().result;
  </precondition>
  <effect>
  </effect>
</function>
```

Note that the precondition is the negation of the precondition of the borrowBookPF processing function. So when this precondition is also evaluated an input isAvailable() is sent to the book SXM.

When an input is sent to a SXM the preconditions of all processing functions are evaluated in order to determine which of the processing function will be triggered. This means that when the borrower SXM receives the borrowBook input the evaluation of both the processing function will have as a result that transmission of two isAvailable() inputs to the same book SXM. This is potentially a problem if the isAvailable() input causes the book SXM to change its state or its memory. In that case the results of the input will be different.

So preconditions should have no side-effects: inputs sent to other SXMs as part of the precondition statements should not modify the state or the memory of the receiver SXM.

This is not a problem if the system to be tested is designed properly, i.e. each function is either a mutator or an accessor, and there are no functions which are both.

- A mutator is a function that can modify a system's state. A mutator may perform some computation and modify the values of system variables. Mutators should not return a value.
- An accessor is a function that accesses the contents of a system but does not modify the system. An accessor returns a value of a system variable or the result of a computation using system variables. An accessor should not modify any of the system variables, i.e should not change the state of the system.

In the book example the function isAvailable corresponding to the input isAvailable should be an accessor.

7.4.2 SXM instances definitions

In order to animate the Borrower SXM we need to provide an argument of type Sxm when sending the borrowBook input. The same need occurs when we wish to generate inputs for test generation.

New instances of SXMs can be created by cloning prototype instances of SXMs. The definitions of these prototypes are found either inside the specification file or in an external file: definitions.xml

The Borrower specification includes the following definitions of prototypes:

```
<definitions>
  <defsxm name="availableBook" type="BookSXM">
  </defsxm>

  <defsxm name="borrowedBook" type="BookSXM">
    <input name="setBorrowed"/>
  </defsxm>
</definitions>
```

The availableBook prototype instance is a SXM found in the initial state and memory.

The borrowedBook prototype instance is a SXM which has received the setBorrowed() input so it is found in the borrowed state.

One can define SXMs at any state and memory reachable by sequence of inputs using the defsxm element with attributes the name of the prototype instance and its specific type and children the inputs which need to be provided to the machine. If no inputs are provided then the instance is found in the initial state and memory.

One can create new unnamed instances of SXMs by using the syntax:

newsxm:*prototypeName*

For example one can provide the following input to the animation of the Borrower specification:

borrowBook (newsxm:borrowedBook)

This will create a new clone of the borrowedBook prototype instance as a value of the argument of borrowBook input.

When test cases are generated, the definitions are included in the test file as in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<set k="2">
  <definitions>
    <defsxm name="borrowedBook" sxm="BookSXM">
      <input name="setBorrowed" />
    </defsxm>
    <defsxm name="availableBook" sxm="BookSXM" />
  </definitions>
  <sequences>
...
    <sequence name="borrowBookNotAvailablePF_returnBookPF">
      <call>
        <function name="borrowBookNotAvailablePF" />
        <input name="borrowBook">
          <book type="sxm">newsxm:borrowedBook</book>
        </input>
        <output name="borrowBook_NotAvailable" />
      </call>
      <call>
        <function name="returnBookPF" />
        <input name="returnBook" />
        <output name="returnBook_Error" />
      </call>
    </sequence>
</set>
```

Definitions are always included in the test file since this file should be totally self-containing.

Note that the test generation will fail if there are inputs of type sxm and there is no definitions.xml file or the file does not contain the definition for the specific value.

8 Defining new JSXM types with XSD

New types can be defined by providing their XSD definitions. For each user-defined a type a corresponding .xsd file must exist in the same directory as the specification. The following specification of inputs and output uses two user-defined types: CartType and OrderDetailsType:

```
<inputs>
  <input name="addToCart">
    <arg name="prodId" type="xs:string" />
    <arg name="quantity" type="xs:int" />
  </input>
  <input name="removeFromCart">
    <arg name="prodId" type="xs:string" />
  </input>
  <input name="getCart" />
  <input name="setCart" >
    <arg name="cart" type="CartType" />
  </input>
  <input name="checkOut" >
    <arg name="orderDetails" type="OrderDetailsType" />
  </input>
  </inputs>
<outputs>
  <output name="addToCartOut" />
  <output name="removeFromCartOut" />
  <output name="getCartOut">
    <result name="cart" type="CartType" />
  </output>
  <output name="setCartOut" />
  <output name="checkOutOut" >
    <result name="orderDetails" type="OrderDetailsType" />
  </output>
</outputs>
```

The CartType is defined in the Cart.xsd file:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:complexType name="CartItemType">
    <xs:attribute name="prodId" type="xs:string" />
    <xs:attribute name="quantity" type="xs:int" />
  </xs:complexType>

  <xs:complexType name="CartType">
    <xs:sequence>
      <xs:element name="item" type="CartItemType"
        maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>

<xs:element name="cart" type="CartType"></xs:element>

</xs:schema>
```

The OrderDetailsType is defined in the OrderDetails.xsd file:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:simpleType name="PaymentType">
    <xs:restriction base="xs:string">
```

```
      <xs:enumeration value="Cash" />
      <xs:enumeration value="Visa" />
      <xs:enumeration value="PayPal" />
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="OrderDetailsType">
    <xs:sequence>
      <xs:element name="paymentWith" type="PaymentType" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="orderDetails" type="OrderDetailsType"></xs:element>
</xs:schema>
```

The specification of the processing functions getCartPF and setCartPF which use the new CartType and CartItemType types is shown below:

```
<function name="getCartPF" input="getCart"
  output="getCartOut">
  <precondition>true</precondition>
  <effect>
    CartType cartOut = new CartType();
    for (String i: cart.keySet() ) {
      Integer q = cart.get(i);
      CartItemType cartItem=new CartItemType();
      cartItem.setProdId(i);
      cartItem.setQuantity(q);
      cartOut.getItem().add(cartItem);
    }
    getCartOut.cart = cartOut;

  </effect>
</function>
<function name="setCartPF" input="setCart"
  output="setCartOut">
  <precondition>true</precondition>
  <effect>
    cart.clear();
    for(CartItemType cartItem: setCart.get_cart().getItem()) {
      cart.put(cartItem.getProdId(), cartItem.getQuantity());
    }
  </effect>
</function>
```

In the above in-line Java fragment the Java CartType and CartItemType types and their methods are used. Note that these are generated by JAXB from the source XSD schema. The Java code for the theses class is found in the xjcgen directory.

Part of the XML test case file generated:

```
<?xml version="1.0" encoding="UTF−8"?>
<set k="2">
  <definitions />
  <sequences>
...
    <sequence>
...
      <call>
        <function name="setCartPF" />
        <input name="setCart">
          <cart type="CartType">
```

```
            <item quantity="2" prodId="d34" />
            <item quantity="4" prodId="370" />
            <item quantity="3" prodId="f82" />
            <item quantity="3" prodId="91b" />
            <item quantity="6" prodId="032" />
            <item quantity="8" prodId="84f" />
          </cart>
        </input>
        <output name="setCart_Error" />
      </call>
...
    </sequence>
...
  </sequences>
</set>
```

References

[1] Mike Holcombe and Florentin Ipate. *Correct Systems: Building Business Process Solutions.*

[2] Florentin Ipate and Mike Holcombe. An integration testing method that is proved to find all faults. *International Journal of Computer Mathematics*, 63:159–178, 1997.

[3] Florentin Ipate and Mike Holcombe. Testing data processing-oriented systems from stream x-machine models. *Theor. Comput. Sci.*, 403(2-3):176–191, 2008.

A Appendix

A.1 The complete book specification:

```
<SXM name="Book">
  <states>
    <state name="available" />
    <state name="borrowed" />
  </states>
  <initialState state="available" />

  <transitions>
    <transition from="available" function="setBorrowedPF" to="borrowed" />
    <transition from="borrowed" function="releasePF" to="available" />
    <transition from="available" function="bookIsAvailablePF" to="available" />
    <transition from="borrowed" function="bookIsNotAvailablePF" to="borrowed" />
  </transitions>

  <memory>
    <declaration>
    </declaration>
    <initial>
    </initial>
    <display>
    </display>
  </memory>
  <inputs>
    <input name="setBorrowed" />
    <input name="release" />
    <input name="isAvailable" />
  </inputs>
  <outputs>
    <output name="setBorrowedOut" />
    <output name="releaseOut" />
    <output name="isAvailableOut"><result name="result" type="xs:boolean" /></output>
  </outputs>

  <functions>
    <function name="setBorrowedPF" input="setBorrowed" output="setBorrowedOut"/>
    <function name="releasePF" input="release" output="releaseOut"/>

    <function name="bookIsAvailablePF" input="isAvailable" output="isAvailableOut">
      <effect>
        isAvailableOut.result = true;
      </effect>
    </function>

    <function name="bookIsNotAvailablePF" input="isAvailable" output="isAvailableOut">
      <effect>
        isAvailableOut.result = false;
      </effect>
    </function>
  </functions>

  <testinputgeneration>
    <inputgenerator function="setBorrowedPF" />
    <inputgenerator function="releasePF" />
    <inputgenerator function="bookIsAvailablePF" />
    <inputgenerator function="bookIsNotAvailablePF" />
  </testinputgeneration>
</SXM>
```

A.2 Animation example:

```
Sxm created:types.BookSXM@e09713
   SXM Animator in interactive mode.
   Enter .h.  to see a list of commands.
   Machine has been reset.
   --------------------------------------
   STATE : available
   MEMORY :
   --------------------------------------
   Input :
```

You can see that a SXM of the type BookSXM is created and its current state is available. Enter .i. to see the available inputs:

```
Input :  .i.
======================================
List of inputs:
isAvailable()
setBorrowed()
release()
======================================
STATE : available
MEMORY :
--------------------------------------
Input :
```

Let's change the state to borrowed:

```
Input :  setBorrowed
---types.BookSXM@e09713
---types.BookSXM@e09713 STATE : available
---types.BookSXM@e09713 MEMORY:
---types.BookSXM@e09713 INPUT : setBorrowed()
---types.BookSXM@e09713 OUTPUT : setBorrowedOut
---
Output :  setBorrowedOut
--------------------------------------
STATE : borrowed
MEMORY :
--------------------------------------
Input :
```

The actual input is setBorrowed() but we can omit the parentheses. You can observe the new state of the machine.

Trying the same input would bring an error without changing the state. This is the default output when an input is not accepted at the current state:

```
Input :  setBorrowed
---types.BookSXM@e09713
---types.BookSXM@e09713 STATE : borrowed
---types.BookSXM@e09713 MEMORY:
---types.BookSXM@e09713 INPUT : setBorrowed()
---types.BookSXM@e09713 ERROR : setBorrowed_Error
---
Output :  setBorrowed_Error
--------------------------------------
STATE : borrowed
MEMORY :
--------------------------------------
Input :
```

Enter .q. to quit the animator:

```
Input :  .q.
...\examples\Book_Borrower>
```

A.3 The complete Cart specification:

```
<SXM name="Cart">
  <states>
    <state name="shopping" />
    <state name="pending" /><!--
      <state name="completed" />

    -->
  </states>

  <initialState state="shopping" />

  <transitions>
    <transition from="shopping" function="addToCartPF"
      to="shopping" />
    <transition from="shopping" function="removeFromCartPF"
      to="shopping" />
    <transition from="shopping" function="getCartPF" to="shopping" />
    <transition from="shopping" function="setCartPF" to="shopping" />
    <transition from="shopping" function="checkOutPF" to="pending" /><!--
      <transition from="pending" function="checkStatusPF" to="pending" />
      <transition from="pending" function="completePF" to="completed" />
      <transition from="completed" function="checkStatusPF" to="completed" />
    -->
  </transitions>

  <memory>
    <javaImports>
    import java.util.*;
    </javaImports>
    <declaration>
      <!-- cart stores the quantity for each product id -->
      HashMap&lt;String, Integer&gt; cart;
    </declaration>
    <initial>cart = new HashMap&lt;String,Integer&gt;();</initial>
    <display></display>
  </memory>

  <inputs>
    <input name="addToCart">
      <arg name="prodId" type="xs:string" />
      <arg name="quantity" type="xs:int" />
    </input>
    <input name="removeFromCart">
      <arg name="prodId" type="xs:string" />
    </input>
    <input name="getCart" />
    <input name="setCart" >
      <arg name="cart" type="CartType" />
    </input>
    <input name="checkOut" >
      <arg name="orderDetails" type="OrderDetailsType" />
    </input>
    </inputs>
  <outputs>
    <output name="addToCartOut" />
    <output name="removeFromCartOut" />
    <output name="getCartOut">
      <result name="cart" type="CartType" />
    </output>
    <output name="setCartOut" />
    <output name="checkOutOut" >
      <result name="orderDetails" type="OrderDetailsType" />
    </output>
  </outputs>

  <functions>

    <function name="addToCartPF" input="addToCart"
      output="addToCartOut">
      <precondition>true</precondition>
      <effect>
        String id = addToCart.get_prodId();
        int quantity = addToCart.get_quantity();
        int oldQuantity = 0;
        if (cart.containsKey(id))
          oldQuantity = cart.get(id);
        cart.put(id, oldQuantity+quantity);
      </effect>
    </function>

    <function name="removeFromCartPF" input="removeFromCart"
      output="removeFromCartOut">
      <precondition>
        cart.containsKey(removeFromCart.get_prodId())
      </precondition>
      <effect>
        cart.remove(removeFromCart.get_prodId());
      </effect>
    </function>

    <function name="getCartPF" input="getCart"
      output="getCartOut">
      <precondition>true</precondition>
      <effect>
        CartType cartOut = new CartType();
        for (String i: cart.keySet() ) {
          Integer q = cart.get(i);
          CartItemType cartItem=new CartItemType();
          cartItem.setProdId(i);
          cartItem.setQuantity(q);
          cartOut.getItem().add(cartItem);
        }
        getCartOut.cart = cartOut;

      </effect>
    </function>
    <function name="setCartPF" input="setCart"
      output="setCartOut">
      <precondition>true</precondition>
      <effect>
        cart.clear();
        for(CartItemType cartItem: setCart.get_cart().getItem()) {
          cart.put(cartItem.getProdId(), cartItem.getQuantity());
        }
      </effect>
    </function>
    <function name="checkOutPF" input="checkOut"
      output="checkOutOut">
      <precondition>true</precondition>
      <effect>
        checkOutOut.orderDetails = checkOut.get_orderDetails();
      </effect>
    </function>

  </functions>

  <testinputgeneration>
    <inputgenerator function="addToCartPF">
      String id = UUID.randomUUID().toString().substring(0, 3);
      Integer quantity = (new Random()).nextInt(10);
      addToCart.set_prodId(id);
      addToCart.set_quantity(quantity);
    </inputgenerator>
    <inputgenerator function="removeFromCartPF">
      if (cart.size()> 0)
        removeFromCart.set_prodId((String) cart.keySet().toArray()[(new Random()).nextInt(cart.size())]);
    </inputgenerator>
    <inputgenerator function="getCartPF">
    </inputgenerator>
    <inputgenerator function="setCartPF">
      CartType cart = new CartType();
      // put at least 5 items
      int numItems = (new Random()).nextInt(3)+5;
      for (int i=0; i &lt; numItems;i++) {
        String id = UUID.randomUUID().toString().substring(0, 3);
        Integer quantity = (new Random()).nextInt(10);
        CartItemType cartItem=new CartItemType();
        cartItem.setProdId(id);
        cartItem.setQuantity(quantity);
        cart.getItem().add(cartItem);
      }
      setCart.set_cart(cart);
    </inputgenerator>
    <inputgenerator function="checkOutPF">
      OrderDetailsType od = new OrderDetailsType();
      od.setPaymentWith(PaymentType.fromValue("Visa"));
      checkOut.set_orderDetails(od);
    </inputgenerator>
  </testinputgeneration>
</SXM>
```

B Developer's reference guide

Two Java files are created for each specification: the nameSXM_base and the nameSXM versions.

The nameSXM Java class extends the nameSXM_base and it is the class representing the final Java program for JSXM.

B.1 nameSXM

This file contains the in-line Java code from the specification. Apart from the in-line code there is minimal Java code for the operation of the machine. This enables easier error identification in the specification since if there is an error in the Java in-line code of the specification, the Java parser will report the line numbers corresponding to this file.

B.2 nameSXM_base

This file is where everything is defined for the operation of the machine. It extends the abstract SXM Java class.