The Service Specification Guide, v1.0

Anthony J H Simons <a.j.simons@sheffield.ac.uk>
Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello, Sheffield S1 4DP, UK.

Abstract

This document describes how to develop a cloud service specification using the XML specification language and supporting tools developed for the EU FP7 Broker@Cloud Project. A specification describes the interface and semantic behaviour of a software service in an abstract way, using simple mathematical datatypes to model the service. A specification is constructed in stages, by considering firstly the finite state machine (FSM) of the service UI, and secondly the detailed behaviour of each operation, in terms of its inputs, outputs, preconditions and effects (IOPE) on the memory of the service. A specification should be checked using tools provided on the *Cloud Service Quality Control* website, before using it to generate tests.

Table of Contents

1. Introduction

The Broker@Cloud Service Specification Language is a simple mathematical specification language, expressed in XML. It was designed to overcome the limitations of other XML service description languages, particularly WSDL, and other RDF service description languages, such as Linked USDL-Core and MSM (Minimal Service Model), which focus on describing syntactic interfaces. The new specification language also describes the semantic behaviour of a service, which is necessary when reasoning about sequences of interactions, such as when testing the service.

The Service Specification Language (SSL) captures behaviour using two popular semantic models: Finite State Machines (FSM), used to describe the coarse-grained state-based behaviour of the service's UI; and the Input, Output, Precondition and Effect (IOPE) paradigm, which is used to describe the fine-grained behaviour of the service's operations and their effect upon memory. The two models are linked through the labels used on the transitions of the state machine, which correspond exactly to the labels used for each scenario of an operation (roughly, each distinct branching path within that operation).

The SSL has notations for describing an FSM with states and transitions. It has notations for describing the protocol of the service, with its operation signatures, inputs and outputs, preconditions and effects. It has a mathematical language of simple datatypes used to model the memory of the service, including integers, strings, lists, sets and maps, with all attendant functions for manipulating these datatypes. The SSL is defined by the XML Schema:

http://staffwww.dcs.shef.ac.uk/people/A.Simons/broker/ServiceSchema.xsd

This schema and several example instances of services described in the specification language are available at the Cloud Service Quality Control website:

http://staffwww.dcs.shef.ac.uk/people/A.Simons/broker/

2. Model-based Specification

Why go to the trouble of developing a quasi-formal specification of a software service? The reason is that it is possible to generate very strong and complete test suites from compact specifications, which are more effective than developer-based tests. It is well known that handwritten developer tests may be good for revealing whether a system exhibits (some of) its expected behaviour, but are quite bad at anticipating all the ways in which a system might be abused. For this, model-based test generation is more effective, since it covers all the different test cases automatically, by exhaustive algorithm. However, first you need a model specification.

2.1 The Finite State Machine

The FSM part of the specification allows you to specify the high-level states of a service, that is, the different responsive modes in which different subsets of its operations are enabled, or disabled. The FSM is created to show explicitly which operations are allowed in which states, and by implication, which operations should be ignored in certain states (the designer only has to specify the expected transitions; missing transitions represent ignored operations).

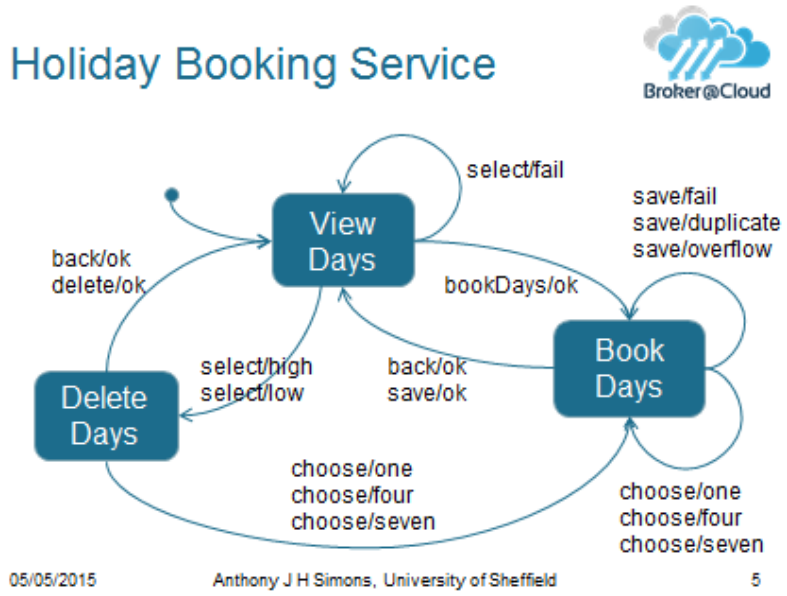

Fig. 1: Visualising the states and transitions of a holiday booking service.

A correctly specified FSM consists of a number of states that are linked by transitions, as visualised in fig. 1 (rendered as a diagram, rather than the actual XML). Exactly one state must be the initial state (shown by the initial arrow to the state *View Days* in fig. 1). The transitions are all labelled with the names of actions available in that state. More precisely, each transition label consists of two parts: the *request* and the *response*, in the format: *request/response*. Examples of these include: *bookDays/ok*, which denotes the successful scenario of the *bookDays* operation (which has only one scenario); and *select/fail*, which denotes the error-handling scenario of the *select* operation (which has three scenarios altogether). The specification of operations with multiple scenarios is described in more detail in section 2.2 below.

The simple app depicted in fig. 1 exists in three states: *{ViewDays, BookDays, DeleteDays}*. In the state *ViewDays*, it is possible to attempt only the two operations: {*bookDays, select}*. Whereas the operation *bookDays* has a single scenario *bookDays/ok*, the *select* operation has multiple scenarios, described by the three transitions: *{select/fail, select/high, select/low}*, which are triggered according to the underlying memory state of the service. Each of these transitions describes a different kind of branching behaviour within the *select* operation.

From fig. 1, it is also apparent that no transitions are specified in the state *ViewDays* for the operations *{back, choose, save, delete}*, so all of these operations are implicitly ignored in this state. When exploring the specification, the tools will automatically add transitions with names like: *save/ignore, choose/ignore* etc. to complete the state machine. Ignored transitions represent null operations that loop back to the same state.

2.2 The IOPE Protocol

The IOPE Protocol part of the specification allows you to specify the behaviour of individual service operations in more detail. The protocol is both a syntactic description of the service's interface, and also a semantic description of how operations behave under different input and memory conditions, stating what outputs should be observed, and what updates should be performed on the memory, which is an abstraction of the implemented service's variables.

In more detail, the protocol consists of a description of memory, in terms of the (abstract) constants and variables used by the service, followed by a description of each operation, with its inputs, outputs, preconditions and effects. Each operation is broken down into a number of scenarios, which each describe one distinct path through the operation[1]. A scenario is triggered by a condition on inputs and memory. All the scenarios of an operation must have conditions which are *mutually exclusive* (they don't overlap) and *exhaustive* (together they cover all possible inputs and memory). This property is checked by the tools. A scenario may also specify a side-effect that it has upon the system, in terms of assignments to memory variables, and how the outputs are bound.

Operation withdraw
input: amount : Integer
output: result : Boolean
Scenario withdraw/ok
condition: amount > zero and amount <= balance
effect: balance := balance - amount, result := true
Scenario withdraw/blocked
condition: amount > zero and amount > balance
effect: result := false
Scenario withdraw/error
condition: amount <= zero
effect: result := false

Fig. 2: Visualising the specification of the *withdraw* operation (incomplete).

[1] Note: there is a subtle distinction between the notion of a *code branch* and a *scenario*. Sometimes a designer needs to specify more scenarios than a programmer would naturally create branches for an operation; this is in order to force the selection of certain data in paths that eventually cause different parts of the service to be exercised. The notion is quite close to the UML notion of *scenario* in the use case model.

Fig. 2 visualises an operation specification (in pretty syntax, rather than the actual XML. See also fig. 4). This is the *withdraw* operation, taken from a banking service. The operation accepts an *amount* as input and produces a Boolean *result*. It has three scenarios, of which the first, *withdraw/ok*, describes the successful withdrawal case and is triggered by a two-fold condition on the input *amount* (which must be greater than zero) and the memory variable *balance* (which must be greater than, or equal to the *amount*) and has a two-fold effect: the *balance* is decremented by *amount*, and the result is set to true. The second scenario, *withdraw/blocked*, describes the unsuccessful case, showing how no withdrawal can be made if the requested *amount* is greater than the *balance*. The final scenario, *withdraw/error*, describes an attempt to withdraw an invalid zero or negative *amount* (a failed precondition). The *error* response is typically reserved to signal failed preconditions.

Notice how the three conditions cover all possible values of the input *amount* and the memory *balance* (exhaustive requirement); and none of them overlaps with any of the other conditions (mutually exclusive requirement). If any conditions overlapped, the operation would be non-deterministic (the system could behave arbitrarily); and if the conditions did not cover the whole input and memory space, then the operation would be blocking (for some input and memory, the system would crash). The verification tool will check this.

An operation may define zero-to-many inputs; and also zero-to-many outputs. In the case of many outputs, the implemented system may return a list, or an array of results; or the SOAP or JSON response may wrap many outputs. It is legal for an operation to have no *inputs* (e.g. if it acts only on memory variables). It is legal for an operation to have no *outputs* (e.g. if it only updates memory variables). It is only legal for an operation to have no *condition*, if it consists of exactly one universally-executed scenario. It is only legal for an operation to have no *effect*, if it has no outputs, failures or updates to memory.

In the above, certain transitions were styled as *fail* or *error* responses. This is the usual way to handle broken preconditions in a fail-safe way. In this example, the failure was handled as part of the normal logic, by returning the result *false*. This contrasts with an alternative approach, in which failures are treated as exceptions on the server-side. In this case, it is possible to bind a special *failure* output, to indicate that the server has raised an exception (c.f. Not Found, or Not Authorized HTTP errors). Finally, *error*-responses (for handling broken preconditions) contrast with implicit *ignore*-responses (for temporarily disabled operations), which are always null-operations.

2.3 The Memory and Binding

The Memory describes the constants and variables used to model the more detailed states of the system. Note that these variables are still abstract, and do not have to follow exactly the implementation of the system, so long as they are sufficient to describe and model the conditions and effects of operations. Symbolic constants must be declared for every constant value used elsewhere in the specification. Variables may be declared to represent as much detailed structure as is needed to capture the desired specification. All constants and variables are declared with a name given in “camelCase”, and a type given in “CapitalCase”.

The expression language supports simple types: *Integer, String, Double, Boolean, Character, Long, Short, Float, Byte* and *Void*; and also generic mathematical datatypes: *List[T], Set[T], Map[K, V]* and *Pair[K, V],* where square brackets are used to enclose generic parameters. In usage, the generics will

be replaced by actual types, for example: *List[Integer], Set[String], Map[String, Integer],* or *Map[String, List[Integer]]*. This last example shows a nested generic type. It is possible to have arbitrarily nested generic constructions.

Furthermore, it is possible for users to declare their own types, such as: *Customer, Product* or *Vehicle*, but these types are *uninterpreted*, in that they exist, but nothing else is known about them. Instances of such types have unique ids. They can be compared for equality; and so they can be associated with other values in map constructions.

Memory is declared at the head of the Protocol. It typically consists of a number of *constant* declarations, followed by a number of *variable* declarations, followed by an initial *binding* of the variables to constant values. The Memory for the banking example is visualised in Fig. 3 (rendered in pretty syntax, rather than the actual XML):

```
Memory
        constant zero : Integer == 0
        constant emptyString : String == ""
        variable balance : Integer
        variable holder : String
        binding: balance := zero,
                 holder := emptyString
```

Fig. 3: Visualising the declaration of memory and binding

Constants are always declared with their constant values (see 3.1 below), whereas variables are not given values. However, the *binding* clause declares how memory variables should be initialised (or re-initialised, when the system is reset). A binding is simply a list of assignments, which execute in the order of declaration.

If an operation has *inputs*, each scenario will also have a *binding* clause, to bind the inputs of an operation to suitable values that would typically trigger the condition of that scenario. In this case, the binding clause is inserted before the condition, visualised in fig. 4:

```
Operation withdraw
        input:  amount : Integer
        output:  result : Boolean
    Scenario withdraw/ok
        binding: amount :> zero
        condition: amount > zero and amount <= balance
        effect:    balance := balance - amount, result := true
    Scenario withdraw/blocked
        binding: amount :> zero
        condition: amount > zero and amount > balance
        effect:    result := false
    Scenario withdraw/error
        binding: amount := zero
        condition: amount <= zero
        effect:    result := false
```

Fig. 4: Visualising the specification of the *withdraw* operation (complete).

Note that binding can assign in a relative way: *amount :> zero* assigns “just more than zero”; as well assign in an exact way: *amount := 0.* The tools interpret such relational binding according to the different types (see section 3.6, below). Above, we said that a binding should *typically* trigger the scenario, because sometimes the memory may not be in a suitable state. In fig. 4, if the *balance* is zero, no input will trigger *withdraw/ok*, which is fine in this case. The binding must be chosen so that the scenario will eventually be triggered, when memory is in a suitable state.

If an operation *modifies variables*, or has *outputs* or *failures*, then it will have an *effect* clause that binds the posterior value of these parameters. The effect clause is essentially a posterior binding, in that it consists of a series of assignments that are executed in the declared order. The effect clause is always inserted after the condition, as shown in fig. 4. The effect must typically bind all outputs, or bind a single failure to some error-message String.

Binding a *failure* is equivalent to raising an exception, so is best used to indicate serious abuse by the client (c.f. 400-series HTTP errors), or inability of the server to deliver due to some internal fault (c.f. 500-series HTTP errors). When tests are generated from specifications with *failures*, these will expect to raise exceptions at the appropriate point (testing must check that the exception is raised).

3. The Expression Language

Conditions and effects contain expressions written in the expression language subset (EL) of the Service Specification Language (SSL). The concepts of the EL correspond to familiar programming language constructions. The kinds of expression are classified according to the meta-model of fig. 5:

Expression (all expressions)
- *Parameter* (atomic expressions)
  - ***Constant*** (immutable constants)
  - ***Variable*** (memory variables)
  - ***Input*** (operation inputs)
  - ***Output*** (operation outputs)
  - ***Failure*** (operation exceptions)
- *Function* (compound expressions)
  - ***Assignment*** (update operations)
  - ***Arithmetic*** (arithmetic operations)
  - ***Projection*** (pairs and projections)
  - ***Manipulation*** (set-theoretic manipulations)
  - *Predicate* (logical operations)
    - ***Comparison*** (totally ordered comparison)
    - ***Membership*** (set-theoretic relations)
    - ***Proposition*** (compound Boolean logic)

Fig. 5: Meta-model hierarchy of the different kinds of expression.

All expressions are either *parameters*, or *functions*. The terminal nodes highlighted in bold font are the only ones used directly in specifications. Each node models a family of related operations, for example, the *Arithmetic* node models the set of arithmetic operations: *{plus, minus, times, divide, modulo, negate}*, whereas the *Comparison* node models the set of predicates: *{equals, notEquals, lessThan, moreThan, notLessThan, notMoreThan}*. The *predicate* hierarchy is elaborated in more detail, due to the tools' need to reason about different kinds of predicate.

Expressions are constructed by enclosing XML terms inside other XML terms. The outer terms are *functions* acting on the inner terms, which should be the expected operands of the function, and could themselves be either nested *functions* or *parameters*. In parameters, the *name*-attribute is the name of that parameter. In functions, the *name*-attribute stores the specific operator-name (see above for examples) of the function to be invoked. Names should be unique within the global and local scope. For example, if an operation has an *input* that was inadvertently given the same name as a global *variable*, the latter will be hidden inside the operation's scope.

3.1 Constants

All parameters are declared with a name and a type. Constant parameters usually have a value, but if declared without a value, default initialisation rules bind the constant to a default initial value, corresponding to *zero, false, null* or *empty*, depending on its type. The following XML declarations are therefore equivalent:

```
<Constant name="zero" type="Integer">0</Constant>
<Constant name="zero" type="Integer"/>
```

Constants play a significant part in a specification, in that they are used to set up the memory, defining sets of values that will become useful later during testing. For example, the holiday booking specification of fig. 1 pre-defined some dates as constants, which were later used in scenario binding expressions to book different periods of holiday:

```
<!-- some predefined dates to choose -->
<Constant name="twelfth" type="Integer">12</Constant>
<Constant name="sixteenth" type="Integer">16</Constant>
```

Designers are encouraged to use constants when picking out the desired input values to be used for different test-cases, for example, a pair of names where the second is the invalid empty String:

```
<Constant name="validName" type="String">John Smith</Constant>
<Constant name="invalidName" type="String"/>
```

Constants can also be declared of the compound types. All the types List, Set, Map and Pair have a literal representation based on the natural Java printed representation of those datatypes[2], using square brackets to surround lists and sets of values, and curly braces to surround maplet pairs, where the key and value parts of a pair are separated using the equals-sign. The element separator in collections is always comma-space, and no other extraneous space is permitted (the parser will currently reject literal values that contain extra space):

```
<Constant name="emptySet" type="Set[String]"/>
<Constant name="drinkSet" type="Set[String]">[wine, beer, schnapps]</Constant>
<Constant name="scores" type="List[Integer]">[3, 7, 12, 6]</Constant>
<Constant name="drinkStock" type="Map[String, Integer]">{wine=60, beer=350}</Constant>
```

Constants of uninterpreted types are declared with some kind of value serving as the identifier for that instance, for example, the constant below named *actionFilm* refers to an instance of a user-defined uninterpreted type called *Dvd*, and the instance has the unique id "*dvd1*":

```
<Constant name="actionFilm" type="Dvd">dvd1</Constant>
<Constant name="romanceFilm" type="Dvd">dvd2</Constant>
```

By convention, the ids of uninterpreted types are the first three letters of the type name, in lower case, followed by a unique digit, incrementing from 1. Constant names are used to refer to these constants elsewhere in the specification; the id values are used to distinguish different instances when comparing for equality (or hashing into sets or maps).

3.2 Other Parameters

Variables, inputs, outputs and failures are all declared in a similar way, but without initial values:

```
<Input name="amount" type="Integer"/>
<Output name="holder" type="String"/>
<Failure name="error" type="String"/>
```

[2] The tools currently use the standard Java types: *ArrayList, HashSet, HashMap* to model the set-theoretic collection types, and the public class *SimpleEntry* to model the pair-type. The latter shadows the inner type *AbstractMap.SimpleEntry*, which is not amenable to marshalling in certain web services.

These kinds of parameter are only bound during the simulation of the specification, by the action of *binding* and *effect* clauses. When a specification is parsed by the tools, name resolution is performed on all parameters, to ensure that all occurrences of the same name refer to the same parameter, wherever it occurs in the specification. If a specification references a name that was not properly declared, the tools report an error. All constants and variables are global and must be declared in the *Memory* clause; whereas all inputs, outputs and failures are local and must be declared in the specific *Operation* clause to which they belong.

When first declared, these parameters must state their type. Later occurrences of the same parameter may choose to omit the type (it is optional, and can be inferred):

```
<Input name="amount" type="Integer"/>  <!-- first declaration -->
<Input name="amount"/>  <!-- later usage -->
```

The two occurrences of *amount* above will be resolved to the same input node in the model.

3.3 Assignment

Assignment is the only function which causes side-effects. All other functions have pure-functional semantics, constructing a new result from their inputs. Assignment is "a kind of function" with no result (of type *Void*), which has the side-effect of binding the first parameter argument to the value of the second expression argument. The legal operator-names used to perform assignment are: *{equals, moreThan, lessThan},* explained below. Examples of assignment include:

```
<Assignment name="equals">
        <Variable name="balance"/>
        <Constant name="zero"/>
</Assignment>

<Assignment name="moreThan">
        <Input name="amount"/>
        <Constant name="zero"/>
</Assignment>

<Assignment name="lessThan">
        <Variable name="counter"/>
</Assignment>
```

The first assigns the exact value *zero* to a variable named *balance*. The second assigns "just more than" *zero* to the input named *amount*. The interpretation of this depends on the type being modified: for an *Integer*, this is a unit increment; for a *Double*, a decimal fraction just greater than zero is assigned; for a String, a lexicographically later String value is assigned; for a Boolean, the truth-value is flipped. The third assignment example is different again, in that it only has a single operand. The operators *{moreThan, lessThan}* may be used with single Integer parameters to perform unit increment, or decrement. Excluding constants, all other kinds of parameter may be re-assigned values. It is a semantic error to try to assign to any other kind of expression.

3.4 Arithmetic

Arithmetic is a kind of function describing arithmetic expressions. The legal types used in arithmetic are the numeric types: *{Integer, Double, Long, Short, Float, Byte}*. The legal operator-names used to

perform arithmetic are: *{plus, minus, times, divide, modulo, negate}*. All except *negate* are binary operators expecting two arguments, whereas *negate* is unary, expecting a single operand. If not given explicitly, the result type of *Arithmetic* may be inferred from its operands.

For example, the following sum might be used as part of the update effect in the *deposit/ok* scenario of the banking service:

```
<Arithmetic name="plus">
        <Variable name="balance"/>
        <Input name="amount"/>
</Arithmetic>
```

This expresses the sum of the *balance* and *amount* (and this sum would have to be re-assigned to *balance* for the result to persist). The precise meaning of each operator may also be affected by the type of operand. The operator *divide* computes the integer quotient for *Integer*, but the floating point division for *Double*. The operator named *modulo* computes the remainder for all types; similarly the operator *negate* reverses the sign for all types.

3.4 Projection

Projection is the node describing pair-construction and deconstruction. Pairs are used to associate two values, such as a search key and an associated value. Pairs are the natural elements of Maps. The legal operator-names used to create and manipulate pairs are: *{pair, first, second}*. For any pair of the type *Pair[K, V]* containing two objects respectively of the types *K, V*, the legal operand and result types used with a Projection are given below (expressed in pretty syntax):

```
pair (key : K, value : V) : Pair[K, V]
first (pair : Pair[K, V]) : K
second (pair : Pair[K, V]) : V
```

That is, the *pair* operator expects two operands and constructs a Pair instance; whereas *first* and *second* project out the first or second elements of a pair. An example in XML is:

```
<Projection name="pair">
        <Variable name="holder" type="String"/>
        <Input name="balance" type="Integer"/>
</Projection>
```

This constructs a new pair, of the type *Pair[String, Integer],* associating a *balance* with an account *holder*. The result of this function, the pair-instance, could be passed to some enclosing outer expression, to insert the pair into a map, or to assign it to a variable, etc. If not given explicitly, the result type of *Projection* may be inferred from its operands.

3.5 Manipulation

Manipulation is the node describing set-theoretic operations on collection data structures, such as *Set, List* and *Map* objects. The legal operator-names for manipulating such data structures are: *{size, insert, remove, insertAll, removeAll, searchAt, replaceAt, insertAt, removeAt}*. These apply in a polymorphic way to objects of the different collection types. All manipulation operations are functionally pure, that is, all updates return new instances of the collection type. For all collections,

the operator *size* returns the element-count. If not given explicitly, the result type of *Manipulation* may be inferred from its operands.

3.5.1 List Manipulation

For all lists of the type *List[T]* containing elements of the type *T*, the legal signatures for expressions manipulating list instances are given below (in pretty syntax):

size (list : List[T]) : Integer
insert (list : List[T], elem : T) : List[T]
remove (list : List[T], elem : T) : List[T]
insertAll (list : List[T], extra : List[T]) : List[T]
removeAll (list : List[T], togo : List[T]) : List[T]
searchAt (list : List[T], index : Integer) : T
replaceAt (list : List[T], index : Integer, value : T) : List[T]
insertAt (list : List[T], index : Integer, value : T) : List[T]
removeAt (list : List[T], index : Integer) : List[T]

These perform the obvious manipulations on a List. Fetching and storing at indices is performed using *{searchAt, replaceAt}*. Insertion and removal at indices is performed using *{insertAt, removeAt}*. Basic appending is performed using *{insert, insertAll}* and bag-deletion and bag-difference are performed using *{remove, removeAll}*.

Note: In common with the Z formal specification notation, the indexing of lists runs from 1..n, rather than the programming language convention, 0..n-1. This means that implementations may need to translate index values carefully.

3.5.2 Set Manipulation

For all sets of the type *Set[T]* containing elements of the type *T*, the legal signatures for expressions manipulating set instances are given below (in pretty syntax):

size (set : Set[T]) : Integer
insert (set : Set[T], elem : T) : Set[T]
remove (set : Set[T], elem : T) : Set[T]
insertAll (set : Set[T], extra : Set[T]) : Set[T]
removeAll (set : Set[T], togo : Set[T]) : Set[T]

These perform the obvious manipulations on a Set that contains unique elements. Basic insertion and set-union is performed using *{insert, insertAll}* and basic removal and set-difference are performed using *{remove, removeAll}*. The operations involving an index are illegal if applied to a *Set* (and raise exceptions in the model).

3.5.3 Map Manipulation

For all maps of the type *Map[K, V]* containing keys of type *K* and values of type *V*, the legal signatures for expressions manipulating map instances are given below (in pretty syntax):

size (map : Map[K, V]) : Integer
insert (map : Map[K, V], pair : Pair[K, V]) : Map[K, V]
remove (map : Map[K, V], pair : Pair[K, V]) : Map[K, V]
insertAll (map : Map[K, V], extra : Map[K, V]) : Map[K, V]
removeAll (map : Map[K, V], togo : Map[K, V]) : Map[K, V]

searchAt (map : Map[K, V], key : K) : T
replaceAt (map : Map[K, V], key : K, value : T) : Map[K, V]
insertAt (map : Map[K, V], key : K, value : T) : Map[K, V]
removeAt (map : Map[K, V], key : K) : Map[K, V]

These perform the obvious manipulations on a Map. Fetching and storing at keys is performed using *{searchAt, replaceAt}*. Insertion and removal at keys is performed using *{insertAt, removeAt}*. Inclusion and deletion of Pair-elements are performed using *{insert, remove}* and map-union-with-override and map-difference are performed using *{insertAll, removeAll}*. To all intents and purposes, *replaceAt* has the same meaning as *insertAt* when applied to a Map, since both create a new key-value mapping if one is not present, and both replace an old mapping with the same key.

3.6 Comparison

Comparison is the predicate node for performing ordered comparisons. The legal types for the operands of comparison include all the ordered simple types: *{Integer, String, Double, Boolean, Character, Long, Short, Float, Byte}*. The legal operator-names for comparing two values of the same types are: *{equals, notEquals, lessThan, moreThan, notLessThan, notMoreThan}.* The operator *notLessThan* has the meaning of "greater than, or equal to"; and the operator *notMoreThan* has the meaning of "less than, or equal to". An example in XML is the following:

```
<Comparison name="moreThan">
        <Variable name="balance"/>
        <Input name="amount"/>
</Comparison>
```

Comparison is polymorphic and applies to two operands of the same type. In addition to comparing ordered simple types, two of the comparison operators *{equals, notEquals}* may be applied to a pair of objects of any common type. The result-type of Comparison is always *Boolean*; this may be inferred.

3.7 Membership

Membership is the predicate node for performing set-theoretic relational comparisons. The legal types for the operands of membership relations are the collections: *{List[T], Set[T], Map[K, V]}* and their related generic element types *{T, K, V, Pair[K, V]}*. The legal operator-names for membership relations are: *{isEmpty, notEmpty, includes, excludes, includesAll, excludesAll, includesKey, excludesKey}.* These operators are polymorphic and have the following meanings.

The predicates *{isEmpty, notEmpty}* report respectively whether the collection has no elements, or has at least one element. The predicates *{includes, excludes}* report respectively whether a collection contains, or does not contain a specified element. In the case of maps, a map element is a pair of the type *Pair[K, V]*, where *K* and *V* are the key and value types of the *Map[K, V].* The predicates *{includesAll, excludesAll}* report respectively whether a collection contains all elements of, or contains no element of, another collection. In the case of lists, element order is not significant. In the case of maps, the equality of pairs is based on the whole pair, not just the key. The predicates *{includesKey, excludesKey}* report respectively whether an indexed *List* or keyed *Map* contains, or does not contain, the specified index or key.

The expected numbers of arguments to each are illustrated in the following examples (in pretty syntax, rather than full XML). The same operations are applicable to more types than those shown here, as described in the polymorphic scheme above.

isEmpty (list : List[T]) : Boolean
notEmpty (map : Map[K, V]) : Boolean
includes (set : Set[T], elem : T) : Boolean
excludes (list : List[T], elem : T) : Boolean
includesAll (set : Set[T], subset : Set[T]) : Boolean
excludesAll (list : List[T], bag : List[T]) : Boolean
includesKey (map : Map[K, V], key : K) : Boolean
excludesKey (list : List[T], index : Integer) : Boolean

These operations are in some cases slightly more general than the usual programming language predicates, for example, a *List* may also be viewed as a keyed collection, whose keys are the indices. The result type of a Membership is always *Boolean*; this may be inferred.

3.8 Proposition

Proposition is the predicate node for performing full propositional logic, that is, for expressing compound Boolean expressions. These are often needed in conditions, which have multiple-part conditions, usually a logical conjunction, but sometimes a logical disjunction. The legal operands for a proposition must have the type *Boolean*, and may be other predicates, or *Boolean*-valued parameters. The legal operand-names for constructing compound propositions are: *{not, and, or, implies, equals}*, which have the usual logical meanings.

Two examples of complementary usage (in pretty syntax) include the following compound Boolean expressions, which describe a single successful case and single unsuccessful case, in conditions prior to withdrawing an amount from a given bank balance:

and(moreThan(amount, zero), notMoreThan(amount, balance))
or(notMoreThan(amount, zero), moreThan(amount, balance))

These two compound expressions are the logical complement of each other, and cover the whole space of inputs and memory. This illustrates the usefulness of the duality between *{moreThan, notMoreThan}* and likewise between *{lessThan, notLessThan}*. The complement of a conjoined predicate (with *and*) is always a disjoined predicate (with *or*). The result type of a Proposition is always *Boolean*; this may be inferred.

Note: Fig. 5 deliberately omits a secret internal *Predicate* subclass called *Atomic*. This is a wrapper that the tools automatically place around Boolean parameters, so that these may be treated as nested predicates inside *Propositions*. The user need not be concerned with this; it is a work-around to allow the tools to reason about Boolean *Parameters* as though they were *Predicates*.

4. Creating a Specification

The best way to create a specification is in stages. We suggest the following procedure to create a simple *ShoppingCart* service specification.

4.1 Service Outline Skeleton

We recommend producing the outline skeleton specification first, then filling in sub-parts as needed. You start by picking a name for the service: this will affect the type names generated for the service client, and for the test-driver, so you may need to consider this.

```
<?xml version="1.0" encoding="UTF-8"?>
<Service name="ShoppingCart">
  <Protocol name="ShoppingCart">
    <Memory name="ShoppingCart">
    </Memory>
  </Protocol>
  <Machine name="ShoppingCart">
  </Machine>
</Service>
```

Note that the name-attributes of the *Service, Protocol, Machine* and *Memory* nodes must be consistent with each other. The service name *ShoppingCart* will be used during test grounding; for example in the JAX-WS grounding, this will produce a Java service client called: *ShoppingCartService*, and an API to access this client called: *ShoppingCartInterface*. The grounding will also generate a JUnit test-driver called: *ShoppingCartTest*.

4.2 Finite State Machine

Next, you should sketch out the stateful behaviour of the UI, by introducing the states that you desire, then designing the transitions that reach other states. At this stage, you will consider names for the transitions, using the *request/response* notation. The following is the *Machine*-part of the above specification:

```
<Machine name="ShoppingCart">
  <State name="Initial" initial="true">
    <Transition name="enterShop/ok" source="Initial" target="Shopping"/>
    <Transition name="exitShop/ok" source="Initial" target="Final"/>
  </State>
  <State name="Shopping">
    <Transition name="addItem/ok" source="Shopping" target="Shopping"/>
    <Transition name="addItem/error" source="Shopping" target="Shopping"/>
    <Transition name="removeItem/ok" source="Shopping" target="Shopping"/>
    <Transition name="removeItem/error" source="Shopping" target="Shopping"/>
    <Transition name="checkout/ok" source="Shopping" target="Checkout"/>
    <Transition name="checkout/error" source="Shopping" target="Shopping"/>
    <Transition name="clearItems/ok" source="Shopping" target="Shopping"/>
    <Transition name="exitShop/ok" source="Shopping" target="Final"/>
  </State>
  <State name="Checkout">
    <Transition name="payBill/ok" source="Checkout" target="Final"/>
    <Transition name="payBill/error" source="Checkout" target="Checkout"/>
    <Transition name="enterShop/ok" source="Checkout" target="Shopping"/>
    <Transition name="exitShop/ok" source="Checkout" target="Final"/>
  </State>
  <State name="Final"/>
</Machine>
```

This FSM has the states: *{Initial, Shopping, Checkout, Final}.* The *Initial* state is also identified as such: *initial=true*. Every transition has a label, a source and a target state. The captured business logic says that: you can enter the shop from the *Initial* and *Checkout* states; you can exit the shop from any state; you can add and remove items only in the *Shopping* state; and you can pay your bill only in the *Checkout* state.

Once this stage is complete, you may submit the specification to the validation tool, to check whether you have missed any transitions that you desire to handle explicitly.

4.3 Protocol Operations

Next, you should sketch out the operations in the protocol. For each bundle of transitions having the same *request*-name, you will need an operation with that name. For each operation, you decide what its inputs, outputs (and failures, if desired) should be. The following is the Protocol-part of the above specification:

```
<Protocol name="ShoppingCart">
    <!-- memory still to be decided -->
  <Memory name="ShoppingCart">
  </Memory>
    <!-- the interface to the service -->
  <Operation name="enterShop"/>
  <Operation name="exitShop"/>
  <Operation name="addItem">
    <Input name="item" type="Video"/>
    <Output name="quantity" type="Integer"/>
  </Operation>
  <Operation name="removeItem">
    <Input name="item" type="Video"/>
    <Output name="quantity" type="Integer"/>
  </Operation>
  <Operation name="clearItems"/>
  <Operation name="checkout"/>
  <Operation name="payBill">
    <Input name="billingInfo" type="String"/>
    <Output name="accepted" type="Boolean"/>
  </Operation>
</Protocol>
```

Here, we have identified seven distinct operations from the bundles of transitions that have the same request-name. Of these, only *{addItem, removeItem, payBill}* have inputs and outputs. The first two allow the user to add one item to, or remove one item from, their shopping cart and be told in response what quantity they have so far. The type *Video* above is a user-defined type, not one of the standard types.

4.4 Operation Scenarios

Next, you determine how many scenarios each operation must have. These are named according to the names of the transitions from the earlier bundles - there is a direct correspondence between transitions and scenarios. Some operations may have one scenario, others may have many. An example of an operation with only one scenario is *enterShop*, which succeeds with one path. Operations with solo scenarios do not require any condition; so the following is all you need to specify:

```
<Operation name="enterShop">
  <Scenario name="enterShop/ok"/>
</Operation>
```

For each scenario in a group, you should then insert a condition satisfying the mutually exclusive and exhaustive requirements. Each condition will contain a single predicate from the Expression Language. The operation *addItem* has two scenarios *{addItem/ok, addItem/error},* representing success and failure to pick an item. We want success to mean when there is sufficient stock; and failure to mean when the item has run out of stock.

```
<Operation name="addItem">
  <Input name="item" type="Video"/>
  <Output name="quantity" type="Integer"/>
  <Scenario name="addItem/ok">
    <!-- we have sufficient stock -->
    <Condition>
      <Comparison name="moreThan">
        <Manipulation name="searchAt">
          <Variable name="currentStock"/>
          <Input name="item"/>
        </Manipulation>
        <Constant name="zero"/>
      </Comparison>
    </Condition>
  </Scenario>
  <Scenario name="addItem/error">
    <!-- we have no more stock -->
    <Condition>
      <Comparison name="notMoreThan">
        <Manipulation name="searchAt">
          <Variable name="currentStock"/>
          <Input name="item"/>
        </Manipulation>
        <Constant name="zero"/>
      </Comparison>
    </Condition>
  </Scenario>
</Operation>
```

The complementary conditions test whether the available quantity of the input *item* is, or is not, more than *zero*. The predicate used is a *Comparison*; it tests the result of a nested *Manipulation*, which searches a variable called *currentStock* (which is a map) for the quantity associated with *item*. The above expressions are not annotated with types: these can be added if desired, but may also be inferred from the declared types of their operands.

4.5 Variables and Constants

While writing conditions, which may test both inputs and variables, you will eventually need to introduce some variables to model system states. These variables capture the detailed business logic, but are still more abstract than in any implementation. Declare these variables in memory; and remember also to define a constant that may be used to initialise each variable.

Above, we referred to the constant *zero*, an *Integer*, and also to a variable *currentStock*, which logically should be a map from videos to integers, therefore having the type: *Map[Video, Integer]*. We define these in memory:

```
<Memory name="ShoppingCart">
    <!-- far from complete -->
  <Constant name="zero" type="Integer"/>
  <Variable name="currentStock" type="Map[Video, Integer]"/>
</Memory>
```

We also need to work out what a suitable initial value for *currentStock* should be. The most effective approach is to consider what test-cases we will need to trigger each condition above. So we need an *initStock* with some items in large quantity, and some that are out of stock:

```
<Memory name="ShoppingCart">
    <!-- slightly better -->
  <Constant name="zero" type="Integer"/>
  <Constant name="initStock" type="Map[Video, Integer]">{vid1=100, p2=0}</Constant>
  <Variable name="currentStock" type="Map[Video, Integer]"/>
</Memory>
```

Below, we will find out that further constants and variables are needed, such as a variable to hold the state of the shopping cart. This is also a map, which has an initial value that is all empty:

```
<Memory name="ShoppingCart">
    <!-- slightly better again -->
  <Constant name="zero" type="Integer"/>
  <Constant name="initStock" type="Map[Video, Integer]">{vid1=100,
vid2=0}</Constant>
  <Constant name="initCart" type="Map[Video, Integer]">{vid1=0, vid2=0}</Constant>
  <Variable name="currentStock" type="Map[Video, Integer]"/>
  <Variable name="shoppingCart" type="Map[Video, Integer]"/>
</Memory>
```

You will need to revisit the contents of memory many times. For example, later you will find that you need some more constants *vid1, vid2* to use as test arguments to the *addItem* method:

```
<Memory name="ShoppingCart">
    <!-- even better -->
  <Constant name="zero" type="Integer"/>
  <Constant name="available" type="Video">vid1</Constant>
  <Constant name="unavailable" type="Video">vid2</Constant>
  <Constant name="initStock" type="Map[Video, Integer]">
     {vid1=100, vid2=0}</Constant>
  <Constant name="initCart" type="Map[Video, Integer]">{vid1=0, vid2=0}</Constant>
  <Variable name="currentStock" type="Map[Video, Integer]"/>
  <Variable name="shoppingCart" type="Map[Video, Integer]"/>
</Memory>
```

Later, we shall return to the memory to add the initial binding of variables to constant values.

4.6 Update Effects

Once the scenario conditions are completed, the next step is to work out what effect each scenario will have. Some scenarios will have no effect (especially ignored *error*-responses), but some will eventually bind outputs (or a failure) and update memory variables. For this, you will create assignments that bind outputs and variables to their posterior values. All assignments are executed serially, in the order declared. This allows the designer to use (some) memory variables to store temporary results; however, each variable may only be updated once per effect (a security issue, to prevent designers from attempting to do complex programming in the effect-clause).

The values to be assigned are constructed in the Expression Language, according to the operations needed to construct the required posterior values. The following sketches the three-fold effect we want for the scenario *addItem/ok:*

```
<Operation name="addItem">
  <Input name="item" type="Video"/>
  <Output name="quantity" type="Integer"/>
  <Scenario name="addItem/ok">
    <!-- we have sufficient stock -->
    <Condition>
      <Comparison name="moreThan">
        <Manipulation name="searchAt">
          <Variable name="currentStock"/>
          <Input name="item"/>
        </Manipulation>
        <Constant name="zero"/>
      </Comparison>
    </Condition>
    <Effect>
      <!-- subtract item from current stock -->
      <!-- add this item to the shopping cart -->
      <!-- return how many copies in the cart -->
    </Effect>
  </Scenario>
  <Scenario name="addItem/error">
    <!-- details omitted -->
  </Scenario>
</Operation>
```

Each of the three updates produces quite a lot of XML in the Expression Language. We show each of these steps below, first in pretty functional syntax, and then in full XML:

currentStock := replaceAt(currentStock, item, minus(searchAt(currentStock, item), 1))

```
<Assignment name="equals">
  <Variable name="currentStock"/>
  <Manipulation name="replaceAt">
    <Variable name="currentStock"/>
    <Input name="item"/>
    <Arithmetic name="minus">
      <Manipulation name="searchAt">
        <Variable name="currentStock"/>
        <Input name="item"/>
      </Manipulation>
      <Constant name="one"/>
    </Arithmetic>
  </Manipulation>
</Assignment>
```

shoppingCart := replaceAt(shoppingCart, item, plus(searchAt(shoppingCart, item), 1))

```
<Assignment name="equals">
  <Variable name="shoppingCart"/>
  <Manipulation name="replaceAt">
    <Variable name="shoppingCart"/>
    <Input name="item"/>
    <Arithmetic name="plus">
      <Manipulation name="searchAt">
```

```
        <Variable name="shoppingCart"/>
        <Input name="item"/>
      </Manipulation>
      <Constant name="one"/>
    </Arithmetic>
  </Manipulation>
</Assignment>
```

quantity := searchAt(shoppingCart, item)

```
<Assignment name="equals">
  <Output name="quantity"/>
  <Manipulation name="searchAt">
    <Variable name="shoppingCart"/>
    <Input name="item"/>
  </Manipulation>
</Assignment>
```

The first two of these update the two memory variables, *currentStock* and *shoppingCart*. The last binds a value to the output *quantity*. In general, if an operation has outputs, the effect should always bind each output to some value. The only case where this is not necessary is when an *Operation* has a *Failure* node declared along with its *Input* and *Output* nodes. This indicates that the operation throws an exception, to be handled by the client of the service. Then, each scenario must either bind all of its outputs, or bind the *Failure* node to some error string constant.

4.7 Initial Binding

The final step is to create an initial binding for the memory variables; and to write a suggested binding for each scenario that has inputs. We have only needed two memory variables in this example *{currentStock, shoppingCart}*, and these are bound as shown below:

```
<Memory name="ShoppingCart">
    <!-- simple constants -->
  <Constant name="zero" type="Integer"/>
  <Constant name="one" type="Integer">1</Constant>
    <!-- test values -->
  <Constant name="available" type="Video">vid1</Constant>
  <Constant name="unavailable" type="Video">vid2</Constant>
    <!-- initial values -->
  <Constant name="initStock" type="Map[Video, Integer]">{vid1=100,
vid2=0}</Constant>
  <Constant name="initCart" type="Map[Video, Integer]">{vid1=0, vid2=0}</Constant>
    <!-- memory variables -->
  <Variable name="currentStock" type="Map[Video, Integer]"/>
  <Variable name="shoppingCart" type="Map[Video, Integer]"/>
    <!-- the initial binding -->
  <Binding>
    <Assignment name="equals">
      <Variable name="currentStock"/>
      <Constant name="initStock"/>
    </Assignment>
    <Assignment name="equals">
      <Variable name="shoppingCart"/>
      <Constant name="initCart"/>
    </Assignment>
  </Binding>
</Memory>
```

This will ensure that the current stock and the shopping cart are always reset to their "clean state" initial values, whenever the specification is simulated and the system is reset to its initial state.

We have three operations: *{addItem, removeItem, payBill}* that require inputs. Each of these consists of a pair of scenarios, representing a successful case, and an error case. If we just consider for now the operation *addItem*, the following bindings will trigger the two scenarios:

```
<Operation name="addItem">
  <Input name="item" type="Video"/>
  <Output name="quantity" type="Integer"/>
  <Scenario name="addItem/ok">
    <!-- we have sufficient stock -->
    <Binding>
      <Assignment name="equals">
        <Input name="item">
        <Constant name="available"/>
      </Assignment>
    </Binding>
    <Condition>…</Condition>            <!-- details elided -->
    <Effect>…</Effect>
  </Scenario>
  <Scenario name="addItem/error">
    <!-- we have no more stock -->
    <Binding>
      <Assignment name="equals">
        <Input name="item">
        <Constant name="unavailable"/>
      </Assignment>
    </Binding>
    <Condition>…</Condition>            <!-- details elided -->
    <Effect>…</Effect>
  </Scenario>
</Operation>
```

This is because we defined two *Video* constants in memory earlier: *{available, unavailable}*; and their id-values *{vid1, vid2}* correspond to items in *currentStock* that respectively have 100 and 0 copies in stock. So, binding the input *item* to available value *vid1* should trigger *addItem/ok*; conversely binding the input *item* to unavailable value *vid2* should trigger *addItem/error*.

Once this stage is complete, you may submit the specification to the verification tool, which will check that every variable and every input is correctly bound; and check that every specified operation is deterministic (does not block, or allow arbitrary choice).

4.8 Refining Scenarios

When generating tests using the testing tool, you may occasionally find that you have designed a specification in such a way that the current scenarios do not give you sufficient test cases to cover every state and transition of the FSM. The test generator will suggest whether you need to increase the path length, or refine your scenarios. Usually, increasing the maximum path length of generated test sequences is what is needed, so that the specification can exercise enough operations to set up the state of memory, so that every operation can eventually be triggered.

Sometimes, however, you need to refine one of your scenarios. This is appropriate, if you need to exercise more test scenarios than you have natural branches in your code, to ensure that all states of

memory can be reached. Refining a scenario is rather like taking a code branch, and splitting it up into two code branches that together cover the same original condition (but with two mutually-exclusive conditions that cover the two parts of the original condition). Each of the two new scenarios may bind its inputs differently, to help the system reach the desired memory state.

As an example, consider the *select* operation in fig. 1. Originally, the designer came up with two scenarios for the *select* operation: *{select/ok, select/fail}*, based on the idea that the *ok* response should happen when the user selects an in-range value (*1..n*) and the *fail* response should happen when the user selects an out-of-range value (either *0*, or greater than *n*).

It was later found that the desired behaviour of the application depended on whether the user selected the *high-index* element in a list-box (in which case, the selection should disappear when that element is deleted), or a low-index element in a list-box (in which case, when that element is deleted, the selection should be transferred onto the next element). So the specification was later adjusted to describe better how the application behaved.

The original scenario *select/fail* had a condition which tested for all invalid indices outside the valid selection range *1..rowCount* (remembering that valid indices are in the range 1..n)

or(notMoreThan(rowNumber, zero), moreThan(rowNumber, rowCount))

The original scenario *select/ok* had the complementary condition:

and(moreThan(rowNumber, zero), notMoreThan(rowNumber, rowCount))

This could be triggered by any value in the range *1..rowCount*. Now, it so happens that the binding chosen to trigger this was the assignment: *rowNumber := 1* (the lowest value greater than zero). So the *select/ok* scenario always tested a *low-index* selection. When the testing tool discovered later that certain paths in the FSM were never covered, this was traced back to the low-index selection.

To address the problem, the specification had to include a transition that would also test the high-index selection case. So, the *select/ok* transition was refined (and replaced) by two transitions: {*select/low, select/high*} to cover these cases. The triggering conditions for these were respectively the following:

and(moreThan(rowNumber, zero), lessThan(rowNumber, rowCount))
and(moreThan(rowNumber, zero), equals(rowNumber, rowCount))

It is clear that these two conditions partition completely the condition for the original *select/ok* case. The binding for the *select/low* case was still *rowNumber :=1*, but the appropriate triggering binding for the *select/high* case could now only be: *rowNumber := rowCount*. Afterwards, the test generator was used to simulate the specification again, and successfully covered all the states and transitions of the model.