Reasoning with XML Data

Wolfgang Laun
Thales Austria GmbH

IntelliFest 2012
San Diego, CA
10/25/2012

Jede Vervielfältigung, Weitergabe des Inhaltes oder sonstige Verwertung darf nur auf Grund schriftlicher Ermächtigung erfolgen. Ein Zuwiderhandeln wird gerichtlich verfolgt.

© Thales Austria GmbH, Wien

All rights reserved. Passing on and copying of this document, use and communication of its contents not permitted without authorization.

Autor: Wolfgang Laun
Datum: September 15, 2013
File: `IntelliFest2012/talk/RulesXML.tex`

Contents

Chapter 1

Introduction

This report combines experiences from two separate projects. There is one connection: both use a rule based system for implementing the "business logic". This is what makes them interesting to an audience as the one attending IntelliFest, but the logical connection is given by the way data provided for reasoning is handled on its way from configuration files into memory and, ultimately, the working memory of the rule based system.

Apparently this topic isn't one that attracts attention. There is, however, evidence that rule authors are frequently suprised by difficulties that can be attributed to fact data not being available in a form that lends itself conveniently to writing rules.

Production rule systems of the first generation, such as the ones that came into existence after Dr. Charles L. Forgy's innovative algorithmic invention, provided conditional elements based on patterns relating to data items through constraint expressions on the data item's fields, and their Cartesian products.

Later on, RBS introduced constructs operating on fields containing collections, either creating Cartesian products with data items that aren't necessarily facts, or accumulating data from the elements of such collections into another data item, temporarily acting as a fact.

Reasoning has become more complex, as can be seen from FAQs on any user list. Often enough, the answers provided suggest that the elements of a fact field's collection are inserted as facts, too, so that they become elible for immediate access in a pattern's elementary constraints. Reactions show, that this "flattening" of the data is not always considered as an easy way out.

The first chapter investigates a data representation that is likely to create hierarchical objects: XML documents.

The second chapter focusses on a project where XML data provides the

backbone for reasoning with rules, and how the issue of creating application objects from the XML input has been solved.

Examples are written in Drools Rule Language (DRL), with Java code for the embedding applications.

Chapter 2

Reasoning with XML Data

2.1 Problem Statement

Given data represented in XML, the overall requirement is to use a production rule system for reasoning over the data to detect arbitrary constellations which are erroneous or unusual—in short, to perform data validation.

Shoddy data, the effort to detect and correct it has been identified as one of the major cost drivers for deploying embedded systems in combination with site-specific provisioning data. What is true for errors in the software is also true for mistakes in the accompanying data: the earlier a problem is detected, the cheaper it is to remove it.

Rules are an excellent instrument for checking complex data, as they can be added and modified without changing the application code.

Still, the effort to deal with some specific XML structure to make it available to the RBS can be considerable, even when no transformation of the objects resulting from unmarshalling an XML document is required. Moreover, any change in the structure of a DOM will require a change in the code processing that data—unless this code is generic.

Is it possible to create a generic algorithm for inserting all element of a DOM tree as facts into a rule base system's working memory? Is all of the information contained by this tree still available after taking it apart?

2.2 Facts and Data Structures

2.2.1 Recognizing Information Hidden in the Structure

The key issue with data about to be entered as facts into working memory is whether the form in which it is available is well suited to be accessed in rules. Two measures of quality matter with respect to conditions are:

- ease of coding rules
- efficiency of rule evaluation

It should be obvious that these criteria may vary—even considerably—between different systems. But there are general principles that can be observed irrespective of the concrete system under consideration. They are the result of properties of the structure of the data, more than those of the RBS itself.

A fundamental statement that can be made about any data structure is that there is information *implied in the structure itself.* The amount and quality of this implied information varies with the data structure and the representation chosen for the data. It is the latter point that affects the interpretation of that data.

2.2.2 A Simple Example: Arrays

It is often overlooked that an array implies that full information about an element requires knowledge of it's index value. Some languages promote this princible by permitting arbitrary ranges of discrete types such as integers or enumerations as index values. Programmers use this, often carelessly, which burdens the client accessing the data structure with associating index and element value for further processing. Consider:

```
int[] unitsPerMonth = new int[12];

int getUnitsOfMonth( int i ){
    return unitsPerMonth[i]; // or i+1
}
```

As soon as, for instance, the maximum value and the corresponding month number need to be passed on for further processing the association defined by the array is lost and must be maintained by an additional data structure, e.g.

```
class UnitsAndMonth {
    int units;
    int month;
}
```

2.2.3 Another Common Case: Lists

If index values just represent ordinals without any other implied meaning, the information contained in a list implies an ordering relation, expressed by "precedes" (or <). Sometimes also the relations "is-successor" and "is-predecessor" are of interest. Once you know the ordinal, you can easily determine the result of any of these relation is trivial to compute. But, again, the basis for these simple formulas is lost unless you know the index of the list elements.

2.2.4 A Closer Look on Maps

A map is a data structure that represents a function, mapping objects of the domain set ("key") domain to the range set ("value"). As a container object, it maintains the relations between the keys and their values, and provides views of all keys ("key set") and all values ("value set").

One thing that is not represented in the map data is the identity of the mapping, i.e., an indication of the function it represents. In a program this may be represented by the name of the map variable, or it may be implied by the map being a property of a higher level data structure. (Of course, this is valid for all variables containing data structures.)

There are some basic scenarios for reasoning about the contents of a map. Let's assume that we have inserted a (Java) Map¡String,Integer¿. This immediately permits us to write a (Drools) rule for retrieving the value for a key given in a `Lookup` fact:

```
rule locateValueFromKey_1
when
  Lookup( $key: key )
  $map: Map( keySet contains $key )
then
  System.out.println( "Map: " + $key + " -> " + $map.get( $key ) );
end
```

If the value itself is needed in the condition, it has to be extracted using a technique that may or may not be provided by the RBS language:

```
rule locateValueFromKey_2
when
  Lookup( $key: key )
  $map: Map( keySet contains $key )
  $value: Integer() from $map.get( $key )
  // ... more CEs referencing $value
then
  System.out.println( "Map: " + $key + " -> " + $value );
end
```

These rules are based on the assumption that there is just one such Map inserted as a fact since the type `Map` doesn't provide a unique identifying attribute.

Ad question that is somewhat awkward to answer is the retrieval of all keys that are mapped to some specific value. This requires iterating over the key set, testing each key:

```
rule locateKeyFromValue_1
when
  Lookup( $value: value )
  $map: Map( values contains $value )
  $key: String( eval( $map.get($key).equals( $value) ) )
        from $map.keySet()
then
  System.out.println( "Map: " + $key + " -> " + $value );
end
```

To simplify reasoning over maps we introduce classes `MapId` and `Mapping`, simple wrappers for `Map` and `Map.Entry`, respectively:

```
public class MapId {
  private Map<?,?> map;
  private int mapId;
  //...
}

public class Mapping<K,V> implements Map.Entry<K,V> {
  Map<K,V> map;
  int mapId;
  Map.Entry<K,V> entry;
  //...
}
```

The reasons for class `MapId` and attribute `mapId` are explained below.

Maps are now inserted wrapped into a `MapId` object and also as a set of `Mapping` objects wrapping the map's `Map.Entry` objects:

```
private void insertMap( StatefulKnowledgeSession kSession,
                        Map<?,?> map ){
  for( Map.Entry<?,?> entry: map.entrySet() ){
    Mapping<?,?> mapping = new Mapping( map, entry );
    kSession.insert( mapping );
  }
  kSession.insert( new MapId( map ) );
}
```

A simple lookup isn't much different from the rules shown previously:

```
rule locateValueFromKey_3
when
  Lookup( $key: key )
  Mapping( key == $key, $value: value )
then
  System.out.println( "Map: " + $key + " -> " + $value );
end
```

But the benefit of maintaining the structural relationship by individual `Mapping` facts becomes obvious with a rule retrieving keys for a given value.

```
rule locateKeyFromValue_2
when
  Lookup( $value: value )
  Mapping( $key: key, value == $value )
then
  System.out.println( "Map: " + $key + " -> " + $value );
end
```

Several additional tasks become quite simple. Here is, for instance, a first effort for writing a rule that locates key-value pairs common to two different maps:

```
rule findIdenticalEntries_1
when
  Lookup( $key: key )
  Mapping( $map: map, key == $key, $value: value )
  Mapping( map != $map, key == $key, value == $value )
```

```
then
  System.out.println( "In two Maps: " + $key + " -> " + $value );
end
```

But there is a gross error in this rule, due to `map != $map` silently being replaced by a method call `! map.equals( $map )`. This is not only quite inefficient but it even fails if two different maps have equal contents. We have to compare the map *identifiers* to ensure that we refer to different map objects, as shown in the rule given below.

```
rule findIdenticalEntries_2
when
  Lookup( $key: key )
  Mapping( $mapId: mapId, key == $key, $value: value )
  Mapping( mapId > $mapId, key == $key, value == $value )
then
  System.out.println( "In two Maps: " + $key + " -> " + $value );
end
```

Note that comparing the references to the respective maps is not a good idea since Drools will silently replace a test for (in)equality with an invocation of `equals`, which is not only quite inefficient but will even fail if two different maps have equal contents.

Demonstrating the usefulness of the wrapper `MapId`, here is a check whether there is one map that is the subset of another map we can now write:

```
rule findMapSubsetOfMap
when
  $m1: MapId( $mapId1: mapId )
  $m2: MapId( $mapId2: mapId < $mapId1 )
  forall( Mapping( mapId == $mapId1, $key: key, $value: value )
          Mapping( mapId == $mapId2, key == $key, value == $value ) )
then
  System.out.println( "Map: " + $mapId1 + " < " + $mapId2 );
end
```

2.3 Fact Data Represented in XML

After exploring the usefulness of representing structural relationships for simple data structures we'll now tackle a much more complex structure: XML document trees.

XML is a frequentely used representation, typically defined by an XML Schema or a similar declarative language. There are several out-of-the-box solutions for marshalling and unmarshalling, with the latter conveniently producing an object tree, i.e., a single hierarchy of (Java) objects. For this we have reference implementation for Java Architecture for XML Binding (JAXB), and several similar implementations can be found on the internet.

XML relates data elements to each other using a handful of elementary relationships and data types. An element may have any combination of children: text nodes, attributes, and other elements. XML Schema also introduces a notation for references between elements by use of data types `ID` (a unique element identifier) and `IDREF`, a reference to another element.

Given a hierarchical object tree (not the same as the tree defined by the XML Document Object Model) as the result of an unmarshalling operation, we'll investigate these two questions:

- How can you condition the data for use in a Production Rule Based System, i.e., how is it best inserted into the system's Working Memory?
- Is there a way of preserving information implied by the DOM structure?

2.4 A Generic Algorithm for Fact Insertion

Code traversing the tree of a DOM with specific element types is not reusable. But we know that a tree traversal can be expressed as an abstract algorithm. Thus, an answer to the first question is to provide a generic algorithm for inserting all elements of an object tree.

Using Java's reflection capabilities we can probe into any object, examining its fields while distinguishing between simple types and object references. Recursion sets in whenever a field references an object. Special handling is required for certain Collection types, as it will not be useful to descend into their implementations. (We may safely assume that only a limited number of such types results from unmarshalling a DOM tree.)

The following requirements apply to the envisaged algorithm for processing an object hierarchy:

1. Process all children of the document root to insert these objects as facts.
2. Process inserted facts by inspecting their fields.
3. Any field type that's not `java.lang.*` or `java.util.*` is processed recursively.

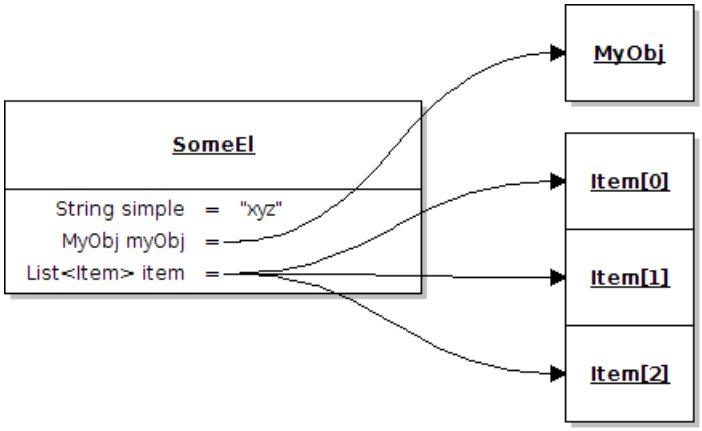

Figure 2.1: An XML Element after Unmarshalling

4. Iterate over `List` elements.

5. Insert isolated Java objects into Working Memory.

2.5 Information Lost

Figure 2.1 displays a group of Java objects after their construction in the unmarshalling process from an XML snippet like this:

```
<docRoot>
  ...
  <someElement simple="xyz">
    <myObject>...</myObject>
    <item>...</item>
    <item>...</item>
    <item>...</item>
  </someElement>
  ...
</docRoot>
```

Reasoning with facts of the types `SomeElement`, `MyObject` and `Item` is now possible, and the links represented by arrows are still there, implied by some object being referenced by a containing ("parent") object.

But some information that is available in the DOM tree is lost, for instance:

- A *child* object does not contain a reference to its *parent*.

- Given two or more elements contained in a list, we do not know their absolute position, or in which order they appear within the list.
- There's no way we can determine the *ancestor axis* for some object, i.e., its parent, grandparent and so on, towards the document root object.
- The nesting level (or depth) of the element in the DOM tree isn't known.
- And, assuming we have unmarshalled and inserted more than one XML document, given any two objects we do not know whether they stem from the same document or not.

The terminology used in the preceding paragraphs has freely borrowed from the one used in the W3C Recommendation for the XML Path Language (XPath). Can we borrow from XPath concepts to overcome these problems?

2.6 Some Deficiencies in an Object Hierarchy

Reflection is powerful, but it can't deliver what isn't in the runtime representation of a `Class` object and its appendages. One example is the distinction between XML elements and attributes, another one is the loss of XML tag names where a mixture of various elements ends up in a list property.

This means that the information that can be obtained by inspecting the traditional Java entities is insufficient. If we accept that the investigation of the object hierarchy may also rely on JAXB annotations we can overcome this difficulty.

- The `propOrder` property of `XmlType` is useful for obtaining a list of the class properties.
- The `XmlElements` annotation on a list field provides a list of `XmlElement` annotations with the original node name.
- A field results from an XML attribute if it is annotated with `XmlAttribute`.
- Detecting annotation `XmlIDREF` on a field reveals that an object reference is not due to a child element or attribute but stems from an arbitrary element linkage.

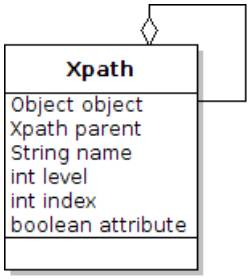

Figure 2.2: Class Xpath

2.7 Is It a Tree or—What?

Even though an XML document *is* a tree, the object hierarchy that results from an unmarshalling operation isn't necessarily such a simple data structure. The existence of the XML Schema types `IDREF` and `ID` is to be blamed, which permit the representation of an object aggregation connected in arbitrary ways.

A relatively harmless situation arises where several references that are stored (in XML) as a value of type `IDREF` leads to a terminal node of the document tree. But it is also possible to create cycles...

Keeping track of inserted objects and skipping any references leading to an already inserted object is possible by filling a (temporarily used) `IdentityHashMap`. And this is also the place where the insertion of `Xpath` objects must stop.

The resume of this insight is that there are limits to the information that can be obtained *generically*, to be made available for reasoning.

If you need the "full picture", it's possible to design a custom set of node elements, to be used in a similar manner to represent arbitrary graphs and to permit reasoning in a dedicated fashion.

2.8 Information Regained

Class `Xpath`, with its class diagram shown in Figure 2.2, stores information that can be obtained from walking an object hierarchy resulting from an XML document.

The list of requirements is extended with the following items.

1. Construct Xpath objects, linking each object resulting from an XML element and its parent Xpath object.

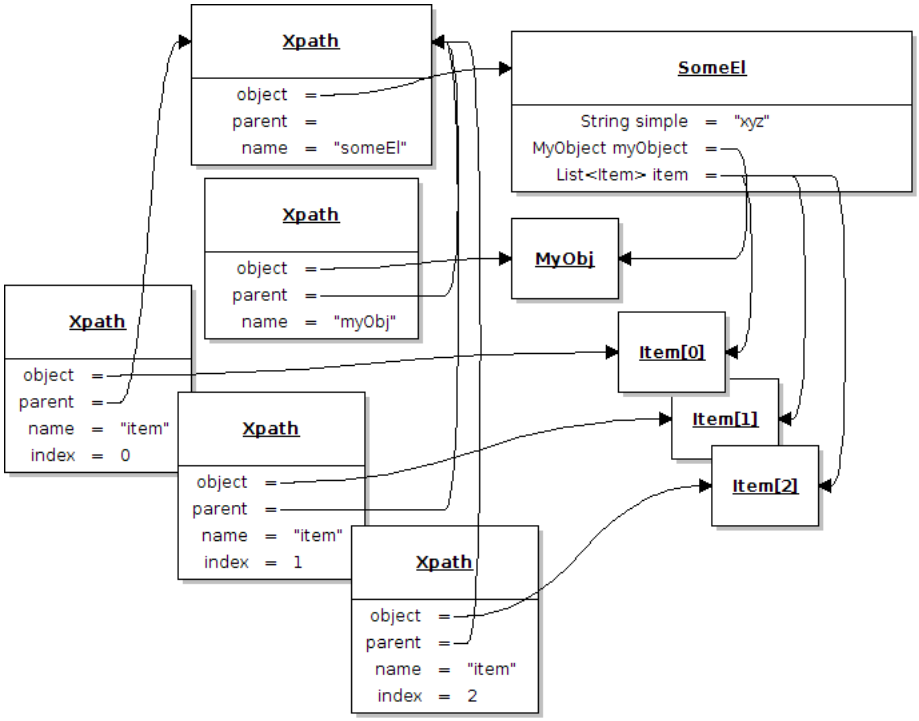

Figure 2.3: Xpath Object Linkage

2. Store the field name and set property `level` to the depth of the tree node.

3. For repeating elements (stored in a parent's `List`), add the list index.

4. Insert the resulting object into Working Memory.

Figure 2.3 shows the resulting object structure after adding `Xpath` objects to the ones depicted in Figure 2.1.

2.9 Exploring the New Possibilities

The usefulness of the generic tree insertion in combination with `Xpath` objects is demonstrated on an XML tree containing data describing a project. Classes have been designed in a way to permit the creation of a hierarchy that is convoluted enough to make rule authoring difficult.

The class diagram in Figure 2.4 shows the (admittedly weird) relationships between the classes headed by `Project`, with `Subproject` and `Task` permitting recursive composition. `Material` and `Risk` represent leaves in the object graph.

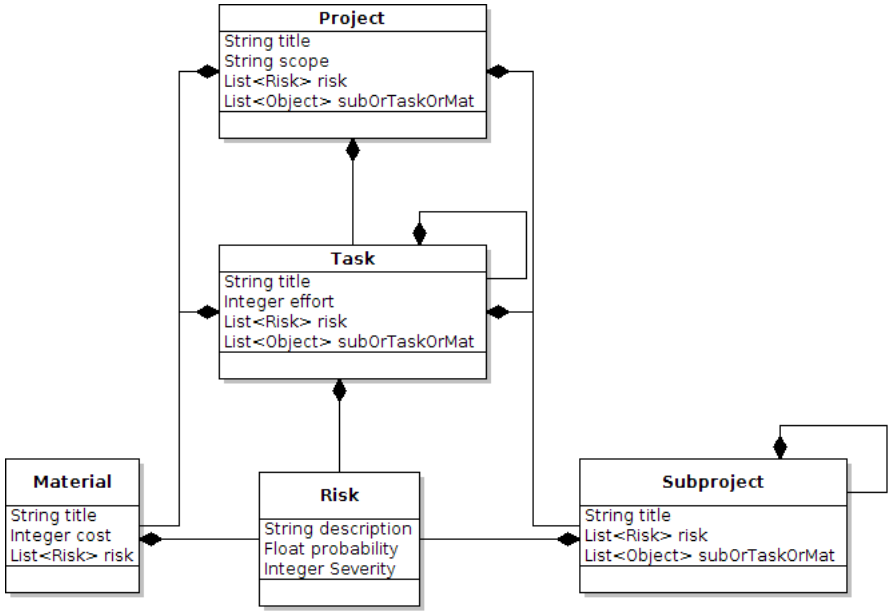

Figure 2.4: Class Project and Components

A small project for building a barn is described by this XML data:

```
<project title="Abigail Duck's new barn">
 <scope>Planing and construction of a barn for A. Duck</scope>

 <task title="Obtain construction permit" effort="10">
  <risk description="Refusal by authority" probability="0.05"
        severity="100"/>
 </task>

 <task title="Construction plan" effort="250">
  <risk description="Rework" probability="0.2" severity="10"/>
 </task>

 <subproject title="Foundation">
  <risk description="Delayed completion" probability="0.01"
        severity="20"/>
  <material title="Timber floor boards" cost="600">
   <risk description="Poor quality" probability="0.2"
         severity="30"/>
  </material>
  <task title="Lay foundation" effort="40">
```

```
  <risk description="Poor workmanship" probability="0.25"
        severity="30"/>
  <risk description="Strike" probability="0.01" severity="5"/>
  <subproject title="Planing">
   <task title="remove shrubs" effort="10">
    <risk description="environment" probability="0.01"
          severity="10"/>
   </task>
   <task title="blast rocks" effort="20">
    <risk description="accident" probability="0.001"
          severity="500"/>
    <material title="dynamite" cost="100"/>
   </task>
  </subproject>
 </task>
</subproject>

<subproject title="Walling">
 <risk description="Delayed completion" probability="0.10"
       severity="20"/>
 <task title="Erect walls" effort="100">
  <risk description="Poor workmanship" probability="0.05"
        severity="40"/>
 </task>
 <task title="Rig barn door" effort="50">
  <risk description="Poor workmanship" probability="0.05"
        severity="5"/>
 </task>
 <task title="Insert windows" effort="200">
  <risk description="Poor workmanship" probability="0.05"
        severity="5"/>
 </task>
 <material title="Beams and boards" cost="600"/>
 <material title="Windows" cost="400"/>
 <material title="Door" cost="200"/>
 <subproject title="Paintwork">
  <material title="Paint" cost="20"/>
  <task title="Prime coat priming" effort="30">
   <risk description="Delayed completion" probability="0.05"
         severity="5"/>
  </task>
```

```
   <task title="Finishing coat" effort="30">
    <risk description="Delayed completion" probability="0.05"
          severity="5"/>
   </task>
  </subproject>
 </subproject>

 <subproject title="Roof">
  <material title="Beams and joists" cost="100"/>
  <material title="Roof tiles" cost="100">
   <risk description="Delayed delivery" probability="0.25"
         severity="5"/>
  </material>
  <task title="Carpentering roof truss" effort="100">
    <risk description="Delayed completion" probability="0.10"
          severity="20"/>
  </task>
  <task title="Roofing" effort="110">
    <risk description="Delayed completion" probability="0.10"
          severity="10"/>
  </task>
 </subproject>

 <task title="Roofing ceremony">
  <material title="Food and beverages" cost="100"/>
 </task>

</project>
```

The first rule merely prints assorted information obtained from an `Xpath` fact and its related fact.

```
rule dump
when
  $object:  Object()
  $xpath: Xpath( object == $object )
then
  System.out.println( "Object, class = " +
    $object.getClass().getSimpleName() + " at " + $xpath +
    ", level = " + $xpath.getLevel() +
    ", name = " + $xpath.getName() );
end
```

More interesting is the possibility to detect facts resulting from a node deeper than a certrain limit. Here it is sufficient to use a pattern referring to an `Xpath` fact.

```
rule "level > 4"
when
  Xpath( $object: object, $level: level >= 5 )
then
  System.out.println( "Level " + $level + ": " + $object );
end
```

This rule produces the following output:

```
Level 5: Risk: Delayed completion
Level 5: Risk: Delayed completion
Level 6: Material: dynamite
Level 6: Risk: accident
Level 5: Task: blast rocks
Level 6: Risk: environment
Level 5: Task: remove shrubs
```

The next rule contains a pattern selecting some specific risk facts, to determine the parent element. The result of this rule simplifies locating the XML element.

```
rule "high risk: anywhere, locate containing element"
when
  $risk: Risk( $severity: severity > 30 )
  $xpath: Xpath( object == $risk )
  $parentXpath: Xpath( $parent: object,
                       this == ($xpath.getParent()) )
then
  System.out.println( "Severity = " + $severity +
    ", for " + $risk + ", in " + $parent );
end
```

Output from this rule is given below.

```
Severity = 40, for Risk: Poor workmanship, in Task: Erect walls
Severity = 500, for Risk: accident, in Task: blast rocks
Severity = 100, for Risk: Refusal by authority, in Task:...
```

We can write a simple rule to obtain all ancestors of a fact:

```
rule "ancestors of a task"
when
  $task: Task( effort >= 110 )
  $xpath: Xpath( object == $task )
  $ancPath: Xpath( $ancestor: object )
  eval( $xpath.isAncestor( $ancPath ) )
then
  System.out.println( "Ancestor of " + $task );
  System.out.println( "  " + $ancestor );
end
```

The pattern locates three matching `Task` facts, and all elements on the path from the corresponding node to the root are found:

```
Ancestor of Task: Roofing
  Subproject: Roof
Ancestor of Task: Roofing
  Project: Abigail Duck's new barn
Ancestor of Task: Insert windows
  Subproject: Walling
Ancestor of Task: Insert windows
  Project: Abigail Duck's new barn
Ancestor of Task: Construction plan
  Project: Abigail Duck's new barn
```

We can accumulate all children of a fact, irrespective of their field names and types, by selecting them via the relationship of the pertaining `Xpath` facts:

```
rule "all children of a complex subproject"
when
  $subproj: Subproject()
  $xpath: Xpath( object == $subproj )
  $list: List( size > 7 )
  from accumulate( $childPath: Xpath( $object: object,
    eval( $xpath == $childPath.getParent() ) ),
    collectList( $object ) )
then
  System.out.println( "Elements of complex subproject " +
                      $subproj );
  for( Object object: $list ){
    System.out.println( "   " + object );
```

```
  }
end
```

The rule locates one complex subproject and prints all facts stemming from child nodes. Note that the accumulated list comprises elements from different list fields.

```
Elements of complex subproject Subproject: Walling
   Risk: Delayed completion
   Task: Erect walls
   Task: Rig barn door
   Task: Insert windows
   Material: Beams and boards
   Material: Windows
   Material: Door
   Subproject: Paintwork
```

The specification for our project data says that a subproject should not be without material. Due to the mixup in the list field, missing material entries cannot be detected straightforwardly. But the following combination of `Xpath` patterns detects the situation:

```
rule "subproject without a material"
when
  $xpath: Xpath( $path: path, name == "subproject"  )
  not Xpath( parentPath == $path, name == "material" )
then
  System.out.println("no materials for " + $xpath.getObject());
  System.out.println("  " + $path);
end
```

The rule produces an output that includes the XPath to the located element.

```
no materials for Subproject: Planing
  /project/subproject[1]/task[1]/subproject[1])
```

Finally we show a rule that locates nodes of different types identified by the existence of more than n children of a kind. It can locate both `Task` and `Subproject` facts. It would even do so for facts of any newly introduced class that also has children of class `Material`.

```
rule "task/subproject with 3 or more materials"
when
  $xpath: Xpath( $object: object )
```

```
  exists $cp: Xpath( name == "material",
                          index >= 3,
                          eval( $xpath == $cp.getParent() ) )
then
  System.out.println("more than 3 materials for " + $object);
  System.out.println("  " + $path);
end
```

The following subproject is found:

```
more than three materials for Subproject: Walling
  /project/subproject[2])
```

Chapter 3

A Railway Signal Simulation

3.1 Introduction

3.2 The Application Domain

The example presented in this section deals with on particular feature of railway signalling: the control of a main signal which does not only indicate "stop" and "clear" to the engineer. A main signal is also capable of

- indicating a maximum speed for progress from the signal onward,
- announcing the maximum speed that will be indicated on the subsequent main signal,
- showing an "on sight" aspect, reserved for unusual situations,
- showing clear for shunt movements.

To achieve all this, the software controlling such a railway signal is responsible for

- accepting commands from an interlocking core subsystem (ICS), requesting some aspect to be shown,
- monitoring the signal's state as provided by some trackside control unit (TCU),
- relaying the signal's state back to the ICS,
- reacting to lamp faults by automatically assuming a fall-back aspect,
- forwarding fault and repair reports to a diagnostic processor.

Some operative requirements common to all railway signals:

- A signal consists of a set of light points which may be off, on or blinking. (Rarely, there are two different blinking frequencies.)
- Aspects not only provide a movement authority distinguishing between "stop" and "clear"; they also signal a maximum permitted speed.
- *Transitions between aspects are restricted.*
- Any valid transition must be effected without showing a valid, less restrictive aspect in between, without the assumption that lamps to be turned on do not fail. This results in a series of *temporary* aspects that need to be accomplished during a transition.

We at Thales (Austria, Germany, Spain,...) are confronted with signals from various European countries, with railway signals varying considerably from country to country.

It will be demonstrated that all of the aforementioned features and requirements can be fulfilled by a single implementation that deals with abstract concepts such as "aspect", "transition", and "fall-back". While the actuals sets of light points, aspects, transitions and fall-back definitions define some specific signal, the universally valid requirements enable us to develop an interpretative implementation of a software process parameterized with data that

- enumerates the set of light points;
- defines a set of valid aspects in terms of light points that are on or blinking;
- specifies all permitted transitions between aspects, and the temporary aspects required in between;
- associates all valid aspects with a fall-back aspect.

The logic used in this singular process deals with the signal unit's *state*, the commanded aspect, and events relayed from the signal interface. It is paramount that this logic must cope consistently and unambiguously with commands and events arriving in any state. The approach described in the subsequent sections guarantees this by using the conditions of a rule-based implementation of the process to *prove* completeness and unambiguity. This proof is based on simple evaluation of Boolean expressions.

3.2.1 A Variation of the Finite State Machine Model

Essentially, the abstract model represents a Finite State Machine, i.e., an entity which can be in any state from a finite set of states. Transitions from state to state is defined by a table, associating each state and input event combination with a successor state and some action to be done during this transition.

This well-known model is modified for two reasons.

1. If the number of states increases, the state machine becomes difficult to define, implement and validate.

2. The implementation of one specific FSM may not be reused for a similar one from the same application domain.

3.2.2 A Rule-Based Implementation

The model outlined in the previous subsections could be implemented in several ways. The rule-based approach has several advantages:

- Rules exhibit conditions in a way that stands out from the technical code implementing the reactions.

- An alternative (simpler or more complex) set of conditions could be deployed simply by using another set of rules.

- The signal's definition must be available as a set of facts, separating it both from the rules and the technical code.

- Additional rules can be provided for running consistency checks on the signal definition data.

3.3 Example: Signal States

This section presents an illustrative example. (Note that the signal shown does not represent the actual arrangement of light points on the signal's front.)

Figure 3.1 shows a main signal, (a) with all light points lit (not a valid aspect), (b) permitting 80km/h and announcing "stop", (c) showing "clear for shunting".

Figure 3.2 h2-m2du.jpg presents the required temporary aspects, i.e., the controller must sense in the order shown that the amber and green light points are working before the red light is switched off.

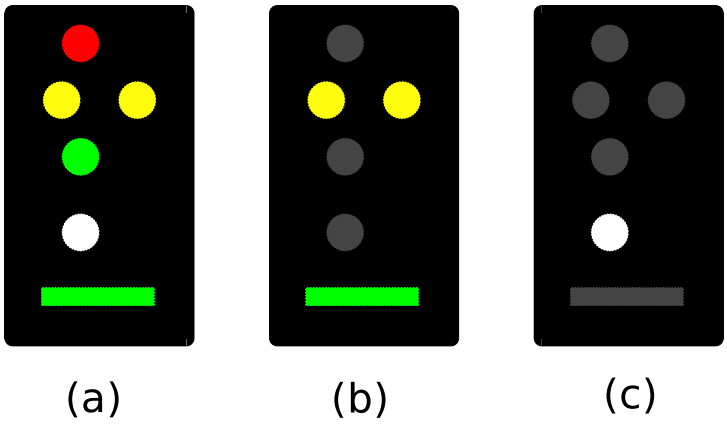

Figure 3.1: Light Points of a Signal (Schematic Illustration)

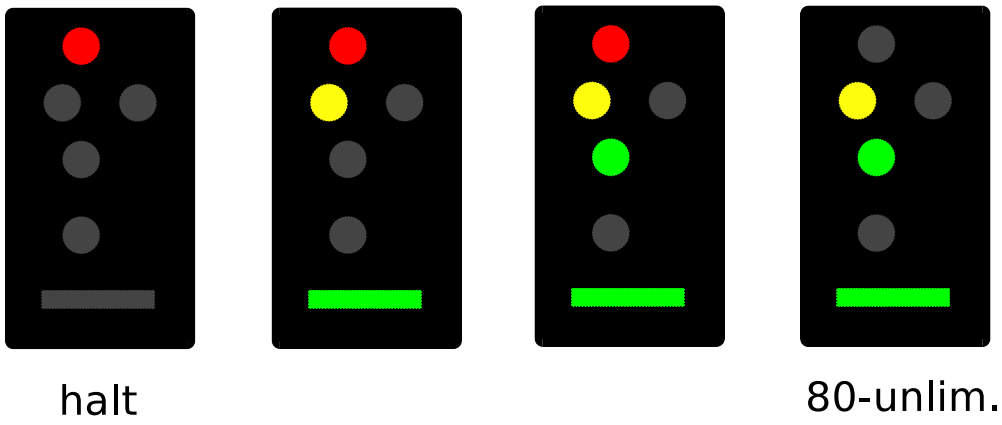

Figure 3.2: Temporary Aspects (Schematic Illustration)

The state transitions required between "halt" and "80km/h-unlimited" are depicted in Figure 3.3. Note that there are several more transitions from states where the result of a command switching a light point on or off are possible, e.g., when a light point cannot be lit in time, or whenever a light point's filament fails.

All in all, there are 15 aspects and 27 temporary aspects. Given that these 42 states require at least one state where the aspect has been commanded but no confirmation has been sensed yet, we arrive at an estimated more than 90 states.

3.4 Static Model

3.4.1 Informal Description

A light point can be in any of the following states: on, off, or blinking. (A second frequency for blinking may have to be added.)

A signal is a combination of individual light points, and its state is the combination of the states of its light points. Only a subset of these states is *valid:* the states representing aspects, as a signal should appear to the engineer. Other states are *temporary* states, required on transitions between aspects and will only be shown very briefly (0.2 seconds max.). Some states should never be reached at all.

For each valid state and aspect, there is a single valid state that must be reached next, i.e., it is not permitted to turn light points on and off arbitrarily.

Since filament failures may prohibit reaching any valid state, each of these valid states has an associated *fall-back* state.

Example 1. Austrian traffic lights have three light points: red, amber, green. Light point states are on, off and blinking. Valid states are {red}, {red, amber}, {amber}, {amber/blinking} (optionally, during the night), {green}, {green/blinking}. No temporary states are required. The fall-back for all valid states is {amber/blinking}, and {} (dark, all light points are off) for {amber/blinking}.

3.4.2 Formal Definition

A *state point* $p \in P$, with P denoting the set of all state points, is an obect that is, at any time, in a *state*, with all possible states coming from a finite set S of state values.

$$S = \{s_1, s_2, ...s_m\}$$

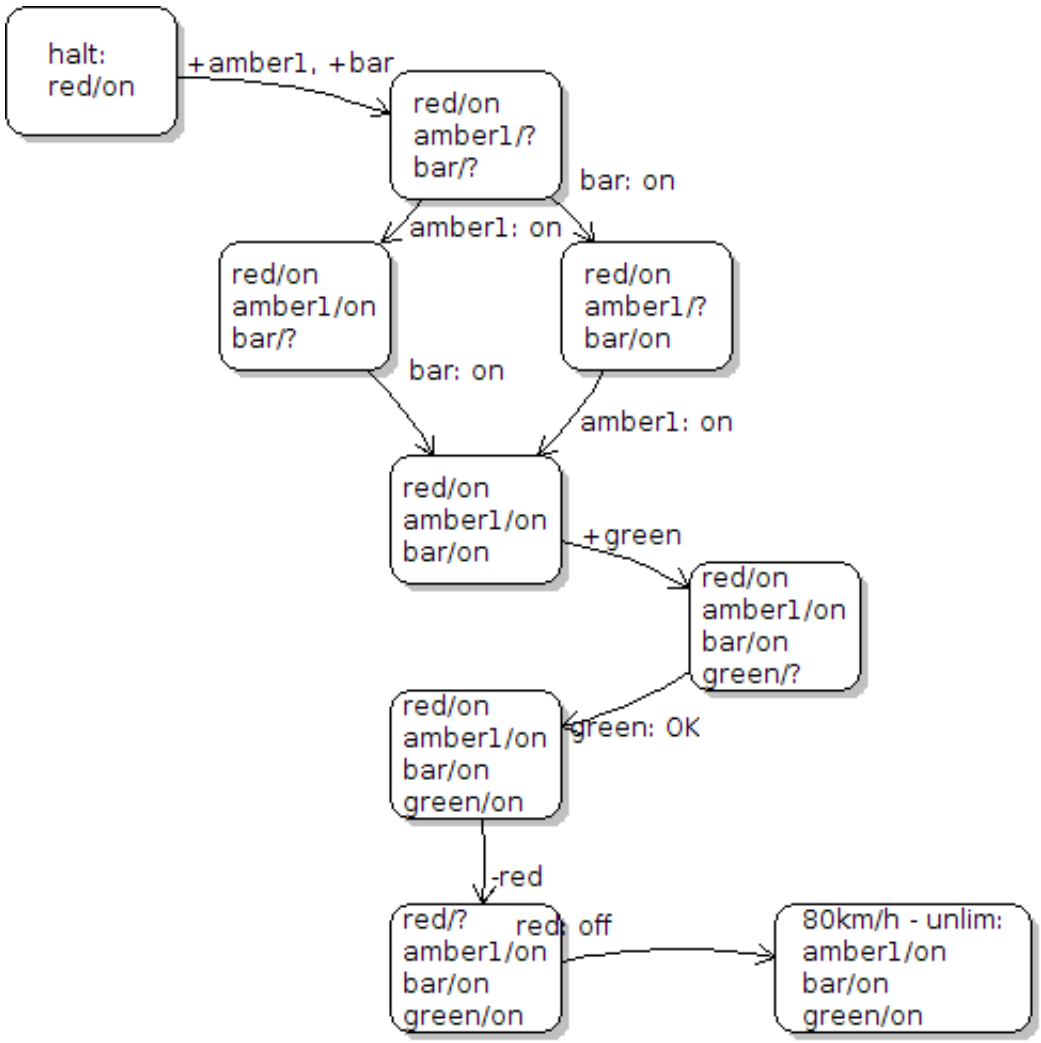

Figure 3.3: States between Halt and 80km/h-unlimited

Example 2. The set of state values for a railway signal's light points is $S = \{\text{off}, \text{on}, \text{b1}, \text{b2}\}$, where "b1" and "b2" stand for two blinking states with different frequencies. ∎

Each state point has a unique invariant property $\nu(p)$ with values from some suitable domain, e.g., the set of integers $I_U = \{x \in \mathbb{N} \,|\, 1 \leq x \leq n\}$.

A *unit* P is the set of all state points p. But to simplify the notation, we shall from now on describe a unit by a finite ordered tuple U of its state points $p \in P$.

$$U = \langle p_1, p_2, ...p_n \rangle \in S^n$$

A tuple U may then also be used as a function, defined on index values from I_U, to obtain a single state point from the tuple of state points. Since an index value is a property of a state point, the following identity holds:

$$U(\nu(x)) = x \qquad \forall x \in P$$

We introduce $V \subseteq S^n$ as the symbol denoting the subset of state tuples that are, by definition, the *valid states* for the unit. Usually V is a proper subset of S^n.

Each state point is associated with a state value from the set S of states. A tuple of state values given in the same order as the points within the unit tuple U represents the state of the unit. We use $\sigma(p)$ to denote the state property of a state point, and $\sigma(U)$ to denote the state of the unit

$$\begin{aligned} \sigma(p_i) &\in S \\ \sigma(U) &= \langle \sigma(p_1), \sigma(p_2), ...\sigma(p_n) \rangle \in R = S^n \\ \sigma(U)(\nu(x)) &= \sigma(x) \qquad \forall x \in P \end{aligned}$$

Another property of a state point is the state currently requested for that point, through some message that is sent to the environment. We denote this requested state by $\varrho(p)$.

$$\begin{aligned} \varrho(p_i) &\in S \\ \varrho(U) = \langle \varrho(p_1), \varrho(p_2), ...\varrho(p_n) \rangle &\in V \end{aligned}$$

The set of requested states is restricted to V, the set of valid states.

Each state point is associated with an initial state which is not necessarily the same for all state points. Combined, they form the tuple defining the initial state for the unit.

$$\sigma_{init} = \langle s_{i_1}, s_{i_2}, ...s_{i_n} \rangle \in V$$

A *state transition function* τ defines the permissible transitions between any two different valid states in terms of transitions between expected states $E \subseteq R$ towards some valid goal state $r \in V$. Notice that E is, in general, a superset of the valid states as it may be necessary to reach one or more intermediary states $x \notin V$ on the progress between two valid states.

$$\tau : E \times V \mapsto E$$

The function τ maps the cartesian product of expected unit states and valid unit states to expected states, according to the following rules.

- $\tau(a, a) = a, \forall a \in V$
- $\tau(a, b) = a$ means that it is forbidden to reach b from a.
- $\tau(a, b) = b$, with $b \in V$, means that b can be reached in a simple, single-step transition.
- If $\tau(a, b) = c$, with $c \notin V$, then, to reach b, a transition to c must be made first, the successive step being defined by $\tau(c, b)$.

Example 3. For a very simple railway signal, we have $P = \{r, a, g\}$, $S = \{\text{off}, \text{on}\}$ and the set of valid states is $h = \langle \text{on}, \text{off}, \text{off} \rangle$, $w = \langle \text{off}, \text{on}, \text{off} \rangle$, $f = \langle \text{off}, \text{off}, \text{on} \rangle$. Then τ is defined by

$$\begin{aligned} \tau(h, w) &= w \\ \tau(h, f) &= f \\ \tau(w, h) &= h \\ \tau(f, h) &= h \\ \tau(w, f) &= h \\ \tau(f, w) &= h \end{aligned}$$

This means that any transition to and from h is possible in a single step, whereas the transitions between w and f first require a transition to h. Note that this is somewhat different from a definition that would rule out a direct transition between w and f by $\tau(w, f) = w$ and $\tau(f, w) = h$. ∎

Another function defines an alternative valid state for each valid state, to be assumed if that state cannot be achieved by a state transition.

$$\varphi : V \mapsto V$$

This alternative state is called "fall-back state". There is only one rule for the interpretation of this function:

- If $\varphi(a) = a$, no action is to be undertaken if a cannot be achieved.

Example 4. For the simple railway signal described in the previous example, function φ would be defined as

$$\varphi(x) = h \qquad \forall x \in \{h, w, f\}$$

The fall-back for h is h, so that a unit where h has been commanded will have $\varrho(r) = \text{on}$ even when $\sigma(r) = \text{off}$. ∎

Another relation may define a (partial) ordering between unit states.

$$\_ < \_ : S \times S$$

The relation is understood to mean "less permissive than".

3.4.3 Consistency Checks

The abstract model is based on sets and functions, which have to be instantiated if the model for some real unit is to be obtained. The actual sets and functions must meet several conditions.

State Sets

Sets have the intrinsic property that all of their elements are unique.

Although it would not invalidate the model, the existence of a state point that occurs only with a single state in all valid unit states would be an indication that something has been left out in the definition of the unit states.

$$\neg\, \exists\, p \in P \bullet \sigma(S_i)(\nu(p)) = \sigma(S_j)(\nu(p)) \quad \forall i \neq j$$

The Transition Function

There must be a transition to each valid state.

$$\forall v \in V \exists \tau(x, y) = v \qquad x \in E, y \in V$$

There must be a transition from each valid state.

$$\forall v \in V \exists \tau(v, y) = x \qquad x \in E, y \in V$$

If there is a transition into some temporary state, then there must be a transition from that temporary state, and both must have the same ultimate destination.

$$\forall e \in E \mid \tau(x, v) = e \bullet \exists \tau(e, v) = f \qquad x, e, f \in E, v \in V$$

The Fall-back Function

There must be a fall-back function defined for each valid state.

$$\forall v \in V \bullet \exists \varphi(v) = x \in V$$

There must be at least one fall-back that doesn't result in a (further) fall-back transition. This a necessary (but not sufficient) condition for fall-back to end up in a stable (if not valid) state.

$$\exists \varphi(x) = x \qquad x \in E$$

There must not be any cycles in the fall-back definitions (except for $\varphi(x) = x$).

3.4.4 Summary

The static model consists of a set of light points, a set of valid states (the aspects) and a set of temporary states. Given these states, a transition function and a fall-back function, there are some simple conditions that must be met for these definitions to be consistent.

A light point occurring in the same state in all (valid and temporary) states is an indication that a part of the required definitions has been omitted. (There is a case of a main signal that should always show {red}. But as the filament of its solitary light point fail, there must be another (fall-back) state: dark.)

There must be at least one transition *to* each valid state so that it can be reached, and there must be at least one transition *from* each valid state so that it can be left. (These conditions are necessary but not sufficient for all states being reachable; what is really required is that each state can be reached—directly or indirectly—from any other state.)

The definition of the transition function must be consistent, i.e., any transition from a valid state to another valid state that reaches a temporary state must have a (unique) continuation towards the goal state.

The fall-back function must be defined for all valid states. It is implied that the goal aspect's fall-back is applicable for all temporary states defined on the transition to that valid goal state.

There must not be any cycles in the fall-back definition, except for a fall-back from a single state (usually "dark") onto itself, which will terminate any further fall-back.

3.5 Dynamic Model

3.5.1 Informal Description

The state machine expects two kinds of inputs: *commands* requesting some aspect and *events* indicating the state of a light point.

As announced in the introductionary section, The model does not use all the valid and temporary states as defined by a signal's aspects and the temporary patterns. Instead, the automaton is based on four *operational states*, as explained in the next subsection. The combined state of the set of light points in their individual states (on, off,...) is kept as a (fact) object. Another object of the same type keeps track of the set of light point states that is to be reached next.

While the static model is concerned with describing the stable states and possible transitions between these states. Actual state switching, however, has to take into account that state transitions are not atomic so that there may be periods of time where the unit is in some transient, intermediary state which may neither be a valid state nor even an expected state. Therefore, the abstract automaton defined by a unit's transition and fall-back functions must be extended to handle transitions that are commanded in a transient state.

Another consequence is that the model must cope with state changes arriving from the external environment "one by one", as state notifications about individual state points.

Also, to model real-world implementations of s state switching unit, we'll have to deal with the possibility that some component of the unit breaks down, resulting in a spontaneous state transition to some arbitrary state which is usually not a valid state.

Finally, to enable the operative environment to react to state changes of a unit, the unit must report all state changes. Therefore, the dynamic model defines a protocol for state reporting.

3.5.2 Additional State Components

Operational States

The set of *operational states* comprises of the values *initializing, stable, changing* and *fall-back.* It is a property of the unit which we'll write as $\omega(U)$. An informal description of these individual states is given below.

- Initializing. The unit is, by definition, in its initial state but this hasn't been confirmed yet by messages from the environment. The unit ac-

cepts commands but does not report its state or faults until full information from the environment is available. A first *requested state* is associated with the initial state or a state received by the last command received in this state.

- Stable. The unit's state point set is in a valid state $s \in V$. This may be due to a command or a fall-back operation.
- Changing. The unit is performing a state change, and the state point set may be in any state $x \in R$, i.e., not necessarily a valid state. In addition to this current state, "changing" is also associated with the current goal state, to be reached next.
- Fall-back. Like "changing", except that the unit has determined that it cannot reach a commanded state s and is currently trying to reach $\varphi(s)$, the state assigned as a fall-back and replacing the commanded goal state.

Finally we note this proposition:

$$\omega(U) = \text{stable} \Rightarrow \sigma(U) = \varrho(U)$$

If the operational state is "stable", the current state must be equal to the requested goal state.

Additional State Point Properties

We define the state point property $\gamma(p)$ to reflect the next (intermediary or final) state to be reached. We can now state the condition

$$\gamma(U) = \varrho(U) = \sigma(U)$$

as a prerequisite for the unit reaching a commanded state.

We also introduce the state point property $\lambda(p)$ as being equal to the last stable state. Predicate $\lambda(p)$ is provided for reporting purposes.

Fault State

An additional state component represents the unit's *fault state*

$$F = \langle f_1, f_2, ...f_n \rangle \qquad f_i \in \mathcal{P}(S)$$

The tuple elements are values from the powerset of states of a state point. A tuple containing only empty sets, i.e.

$$\langle \{\}, \{\}, ...\{\} \rangle$$

represents the situation where there are no faults at all.

Some component f_i of the the fault state F may be extended whenever $\sigma(p_i) \neq \gamma(p_i)$ at the end of some state transition, i.e., when the next state could not be reached. Then, the fault state components are updated:

$$f_i' := f_i \cup \gamma(p_i) \qquad \forall i \in I_U \,|\, \sigma(p_i) \neq \gamma(p_i)$$

The fault state may be reduced whenever a transient or permanent state has been reached.

$$f_i' := f_i \setminus \sigma(p_i) \qquad \forall i \in I_U$$

3.5.3 Commands

A *command* $c = \langle r, b \rangle$ is an element of the cartesian product $V_U \times B$, where $r \in V$ is the state to reach and $b \in B = \{\text{false}, \text{true}\}$. If the second value is true the command overrides an already ongoing state transition.

The reaction to a command $c = \langle r, b \rangle$ depends on the current operational state and other properties of the unit, mostly the current state σ.[1]

Commands During Initialization

If the unit is still initializing, the commanded state is honoured without any consideration for intermediary states that might be defined by the function τ.

$$\begin{aligned} &\text{Rule command} - \text{init 1} \\ &\qquad \omega \;=\; \text{initializing} \\ &\Longrightarrow \\ &\qquad \varrho \;:=\; r \\ &\qquad \gamma \;:=\; r \end{aligned}$$

Commands in a Stable State

If the unit is in a valid and stable state $\sigma \in V$, a command $\langle r, b \rangle$ is accepted irrespective of b, but only if $\tau(\sigma, r)$ indicates that the commanded state r

[1] For better readability, function σ, ω, ϱ and γ will from now on be used without the argument U.

may be reached from the current state σ.

$$\begin{aligned}
&\text{Rule command} - \text{stable 1} \\
&\qquad\qquad \omega \;=\; \text{stable} \quad \wedge \\
&\qquad\qquad r \;\neq\; \sigma \quad \wedge \\
&\qquad \tau(\sigma, r) \;\neq\; \sigma \\
&\qquad\qquad \Longrightarrow \\
&\qquad\qquad \gamma \;:=\; \tau(\sigma, r) \\
&\qquad\qquad \varrho \;:=\; r \\
&\qquad\qquad \omega \;:=\; \text{changing}
\end{aligned}$$

Notice that the condition $r \neq \sigma$ avoids the initiation of a state transition if the unit is already in the commanded state.

If the unit is already stable and in the commanded state, no state transition has to be performed, but a status report has to be sent.

$$\begin{aligned}
&\text{Rule command} - \text{stable 2} \\
&\qquad\qquad \omega \;=\; \text{stable} \quad \wedge \\
&\qquad\qquad r \;=\; \sigma \\
&\qquad\qquad \Longrightarrow \\
&\qquad \text{report}(\sigma, \text{exec})
\end{aligned}$$

If the unit is in a valid, stable state $\sigma \in V$, a command is rejected (irrespective of b if the transition function does not permit a transition from σ to r.

$$\begin{aligned}
&\text{Rule command} - \text{stable 3} \\
&\qquad\qquad \omega \;=\; \text{stable} \quad \wedge \\
&\qquad\qquad r \;\neq\; \sigma \quad \wedge \\
&\qquad \tau(\sigma, r) \;=\; \sigma \\
&\qquad\qquad \Longrightarrow \\
&\qquad \text{reject}(r, \sigma)
\end{aligned}$$

Commands While Changing or During Fall-back

The handling of commands while the unit is in the "changing" or "fall-back" state depends on the override flag b in the command.

If the unit's operational state is "changing" or "fallback" and the command flag is true, the transition to the commanded state will be initiated

immediately. Notice that the function τ is not to be applied in this case, mainly because the current state need not be one of the valid states in V_U.

$$\begin{array}{lrcl}\text{Rule command} - \text{changing} - \text{fallback 1} & & & \\ & \omega & \in & \{\text{changing}, \text{fallback}\} \quad \wedge \\ & b & = & \text{true} \\ & \Longrightarrow & & \\ & \gamma & := & r \\ & \varrho & := & r\end{array}$$

If the unit's operational state is "changing" or "fallback" and the command flag is false, the command is refused.

$$\begin{array}{lrcl}\text{Rule command} - \text{changing} - \text{fallback 2} & & & \\ & \omega & \in & \{\text{changing}, \text{fallback}\} \quad \wedge \\ & b & = & \text{false} \\ & \Longrightarrow & & \\ & \text{refuse}(s, \omega) & & \end{array}$$

3.5.4 Events

An *event* indicates than one or more updates of the actual state of the unit have taken place and that the updated state should be re-evaluated. Notice that this definition does not make any assumptions whether a deviation from a stable state must be tolerated for some time or immediately reacted upon.

Initialization

When an event terminates the "initialization" state, we see whether we have reached the currently requested state. If so, we become stable.

$$\begin{array}{lrcl}\text{Rule event} - \text{init 1} & & & \\ & \omega & = & \text{initializing} \quad \wedge \\ & \sigma & = & \varrho \\ & \Longrightarrow & & \\ & \omega & := & \text{stable} \\ & \omega & := & \varrho \\ & \text{report}(\varrho) & & \end{array}$$

If the initialization state is terminated in a state other than the requested state we report a fault and initiate fall-back.

$$
\begin{aligned}
&\text{Rule event} - \text{init 2}\\
&\qquad \omega \;=\; \text{initializing} \quad \wedge\\
&\qquad \sigma \;\neq\; \varrho\\
&\qquad \Longrightarrow\\
&\text{fault}(\varrho, \sigma)\\
&\qquad \omega \;:=\; \text{fallback}\\
&\qquad \gamma \;:=\; \varphi(\varrho)\\
&\qquad \varrho \;:=\; \varphi(\varrho)
\end{aligned}
$$

Notice that the fall-back rule that applies when the destination state has been reached may apply immediately.

Stable

In the "stable" state, a deviation from the current actual state initiates a fall-back according to function φ.

$$
\begin{aligned}
&\text{Rule event} - \text{stable 1}\\
&\qquad \omega \;=\; \text{stable} \quad \wedge\\
&\qquad \sigma \;\neq\; \varrho \quad \wedge\\
&\qquad \varphi(\varrho) \;\neq\; \sigma\\
&\qquad \Longrightarrow\\
&\text{fault}(\varrho, \sigma)\\
&\qquad \omega \;:=\; \text{fallback}\\
&\qquad \gamma \;:=\; \varphi(\varrho)
\end{aligned}
$$

If the fall-back state for the current state isn't another state, we remain stable but report the new state and the fault.

$$
\begin{aligned}
&\text{Rule event} - \text{stable 2}\\
&\qquad \omega \;=\; \text{stable} \quad \wedge\\
&\qquad \sigma \;\neq\; \varrho \quad \wedge\\
&\qquad \varphi(\varrho) \;=\; \sigma\\
&\qquad \Longrightarrow\\
&\text{report}(\varrho)\\
&\text{fault}(\varrho, \sigma)
\end{aligned}
$$

For completeness' sake we observe that the arrival of an event that does not result in a deviation from the state can be ignored.

$$\begin{array}{rrcl} \text{Rule event} - \text{stable 3} & & & \\ & \omega & = & \text{stable} \quad \wedge \\ & \sigma & = & \varrho \\ & \Longrightarrow & & \end{array}$$

Changing

If the unit's state is "changing", we have to compare the current state with the present goal state to see whether we have reached this goal, which, in turn, could be the requested new state or just an intermediary state. The first scenario deals with reaching the final goal.

$$\begin{array}{rrcl} \text{Rule event} - \text{changing 1} & & & \\ & \omega & = & \text{changing} \quad \wedge \\ & \sigma & = & \gamma \quad \wedge \\ & \gamma & = & \varrho \\ & \Longrightarrow & & \\ & \omega & := & \text{stable} \\ & \lambda & := & \varrho \\ \text{report}(\varrho) & & & \end{array}$$

If we have reached an intermediary state, we continue on the path towards the requested state by updating the goal state.

$$\begin{array}{rrcl} \text{Rule event} - \text{changing 2} & & & \\ & \omega & = & \text{changing} \quad \wedge \\ & \sigma & = & \gamma \quad \wedge \\ & \gamma & \neq & \varrho \\ & \Longrightarrow & & \\ & \gamma & := & \tau(\sigma, \varrho) \end{array}$$

Not being able to reach a goal means that we have to fall back to the

fall-back state defined for the requested state.

$$
\begin{aligned}
&\text{Rule event} - \text{changing } 3 \\
&\qquad \omega \;=\; \text{changing} \quad \wedge \\
&\qquad \sigma \;\neq\; \gamma \\
&\qquad \Longrightarrow \\
&\text{fault}(\gamma, \sigma) \\
&\qquad \omega \;:=\; \text{fallback} \\
&\qquad \gamma \;:=\; \varphi(\varrho) \\
&\qquad \varrho \;:=\; \gamma
\end{aligned}
$$

Notice that the fall-back state may be terminated immediately if the required fall-back state happens to be the current state.

Fallback

If the unit state is "fallback" and we have reached the required fall-back state, the state becomes stable and we report the new status.

$$
\begin{aligned}
&\text{Rule event} - \text{fallback } 1 \\
&\qquad \omega \;=\; \text{fallback} \quad \wedge \\
&\qquad \sigma \;=\; \varrho \\
&\qquad \Longrightarrow \\
&\qquad \omega \;:=\; \text{stable} \\
&\qquad \lambda \;:=\; \varrho \\
&\text{report}(\varrho)
\end{aligned}
$$

Not being able to reach a goal means that we cannot reach the fall-back state. If we have a fall-back state for the fall-back state we're trying to reach, we continue with that.

$$
\begin{aligned}
&\text{Rule event} - \text{fallback } 2 \\
&\qquad \omega \;=\; \text{fallback} \quad \wedge \\
&\qquad \sigma \;\neq\; \gamma \quad \wedge \\
&\qquad \varphi(\gamma) \;\neq\; \gamma \\
&\qquad \Longrightarrow \\
&\text{fault}(\gamma, \sigma) \\
&\qquad \gamma \;:=\; \varphi(\gamma) \\
&\qquad \varrho \;:=\; \gamma
\end{aligned}
$$

In fall-back not being able to reach a goal means that we cannot reach the fall-back state. If we don't have a fall-back state for the fall-back state we're trying to reach, we have to become stable.

$$
\begin{array}{rcll}
\text{Rule event} - \text{fallback } 3 & & & \\
\omega & = & \text{fallback} & \wedge \\
\sigma & \neq & \gamma & \wedge \\
\varphi(\gamma) & = & \gamma & \\
\Longrightarrow & & & \\
\text{fault}(\gamma, \sigma) & & & \\
\omega & := & \text{stable} & \\
\lambda & := & \varrho &
\end{array}
$$

3.5.5 Proofs

Informal Introduction

If you have a set of boolean expressions where each of which should define a set of states represented by some variables you can convince yourself that you have covered all contingencies by straightforward application of boolean algebra, in combination with simple set operations and (when needed) elementary mathematics.

For example, here is a division of the set of all points of the plane with coordinates (x, y) by four boolean expressions:

$$
\begin{aligned}
x >= 0 \wedge y >= 0 \\
x >= 0 \wedge y < 0 \\
x < 0 \wedge y >= 0 \\
x < 0 \wedge y < 0
\end{aligned}
$$

Now, take the disjunction of these four terms and reduce the expression, keeping in mind that $x >= 0 \vee x < 0$ is always true for any real number and

using T for true:

$$\begin{aligned} x >= 0 \wedge y >= 0 \vee x >= 0 \wedge y < 0 \vee x < 0 \wedge y >= 0 \vee x < 0 \wedge y < 0 &= \\ x >= 0 \wedge (y >= 0 \vee y < 0) \vee x < 0 \wedge (y >= 0 \vee y < 0) &= \\ x >= 0 \wedge T \vee x < 0 \wedge T &= \\ x >= 0 \vee x < 0 &= \\ T & \end{aligned}$$

Equally important is the assertion that each combination is handled only once. This requires more work since you'll have to take the pairwise conjunction of these terms, for instance:

$$\begin{aligned} x >= 0 \wedge y >= 0 \wedge x >= 0 \wedge y < 0 &= \\ x >= 0 \wedge y >= 0 \wedge y < 0 &= \\ x >= 0 \wedge F &= \\ F & \end{aligned}$$

Coverage of a State

The rules in the previous sections divide and subdivide the unit state into as many categories as required for handling a command or an event according to the requirements. The question to be answered by this section is whether all possible states are covered by these rules.

We begin by taking the conditions for a command arriving in the stable

state ($\omega = \text{stable}$) and combining them in a disjunction.

$$\begin{aligned}
&\omega = \text{stable} \wedge r \neq \sigma \wedge \tau(\sigma, r) \neq \sigma \vee \\
&\omega = \text{stable} \wedge r = \sigma \vee \\
&\omega = \text{stable} \wedge r \neq \sigma \wedge \tau(\sigma, r) = \sigma \quad = \\
&\omega = \text{stable} \wedge \\
&(r \neq \sigma \wedge \tau(\sigma, r) \neq \sigma \vee \\
&r \neq \sigma \wedge \tau(\sigma, r) = \sigma \vee \\
&r = \sigma) \quad = \\
&\omega = \text{stable} \wedge \\
&(r \neq \sigma \wedge (\tau(\sigma, r) \neq \sigma \vee \tau(\sigma, r) = \sigma)) \vee \\
&r = \sigma) \quad = \\
&\omega = \text{stable} \wedge \\
&(r \neq \sigma \wedge r = \sigma) \quad = \\
&\omega = \text{stable} \qquad (3.1)
\end{aligned}$$

This means that the "stable" state is fully covered.

For the states "changing" and "fallback", we have two simple terms.

$$\begin{aligned}
&\omega \in \{\text{changing}, \text{fallback}\} \wedge b = \text{true} \vee \\
&\omega \in \{\text{changing}, \text{fallback}\} \wedge b = \text{false} \quad = \\
&\omega \in \{\text{changing}, \text{fallback}\} \wedge (b = \text{true} \vee b = \text{false}) \quad = \\
&\omega \in \{\text{changing}, \text{fallback}\} \qquad (3.2)
\end{aligned}$$

Observing that the only rule for a command during the initialization state has the simple condition $\omega = \text{initializing}$ we may now combine this with the results 3.1 and 3.2.

$$\begin{aligned}
&\omega = \text{initializing} \vee \\
&\omega = \text{stable} \vee \\
&\omega \in \{\text{changing}, \text{fallback}\} \quad = \\
&\omega \in \{\text{initializing}, \text{stable}, \text{changing}, \text{fallback}\} \quad = \quad \text{true}
\end{aligned}$$

Since the disjunction of all conditions yields true, we have shown that the arrival of a command is convered in all states.

Likewise we can combine the conditions for the rules of events arriving

during the initialization.

$$\begin{aligned}
&\omega = \text{initializing} \wedge \sigma = \varrho \vee \\
\omega = \text{initializing} \wedge \sigma \neq \varrho \quad &= \\
\omega = \text{initializing} \wedge (\sigma = \varrho \vee \sigma \neq \varrho) \quad &= \\
\omega = \text{initializing} \qquad &= \qquad (3.3)
\end{aligned}$$

For the "stable" state, we form a disjunction from three conditions.

$$\begin{aligned}
&\omega = \text{stable} \wedge \sigma \neq \varrho \wedge \varphi(\varrho) \neq \sigma \vee \\
&\omega = \text{stable} \wedge \sigma \neq \varrho \wedge \varphi(\varrho) = \sigma \vee \\
\omega = \text{stable} \wedge \sigma = \varrho \quad &= \\
\omega = \text{stable} \wedge \quad &\\
(\sigma \neq \varrho \wedge \varphi(\varrho) \neq \sigma \vee \quad &\\
\sigma \neq \varrho \wedge \varphi(\varrho) = \sigma \vee \quad &\\
\sigma = \varrho) \quad &= \\
\omega = \text{stable} \wedge \quad &\\
(\sigma \neq \varrho \wedge (\varphi(\varrho) \neq \sigma \vee \varphi(\varrho) = \sigma) \vee \quad &\\
\sigma = \varrho) \quad &= \\
\omega = \text{stable} \wedge \quad &\\
(\sigma \neq \varrho \vee \sigma = \varrho) \quad &= \\
\omega = \text{stable} \qquad &= \qquad (3.4)
\end{aligned}$$

Here is the same operation for the "changing" state.

$$\begin{aligned}
&\omega = \text{changing} \wedge \sigma = \gamma \wedge \gamma) = \varrho \vee \\
&\omega = \text{changing} \wedge \sigma = \gamma \wedge \gamma) \neq \varrho \vee \\
\omega = \text{changing} \wedge \sigma \neq \gamma \quad &= \\
\omega = \text{changing} \wedge \quad &\\
(\sigma = \gamma \wedge \gamma) = \varrho \vee \quad &\\
\sigma = \gamma \wedge \gamma) \neq \varrho \vee \quad &\\
\sigma \neq \gamma) \quad &= \\
\omega = \text{changing} \wedge \quad &\\
(\sigma = \gamma \wedge (\gamma) = \varrho \vee \gamma) \neq \varrho) \vee \quad &\\
\sigma \neq \gamma) \quad &= \\
\omega = \text{changing} \wedge \quad &\\
(\sigma = \gamma \vee \sigma \neq \gamma) \quad &= \\
\omega = \text{changing} \qquad &= \qquad (3.5)
\end{aligned}$$

And, finally, here is the "fallback" state.

$$\begin{aligned}
\omega = \text{fallback} \wedge \sigma = \varrho \vee \\
\omega = \text{fallback} \wedge \sigma \neq \gamma \wedge \varphi(\gamma) \neq \gamma \vee \\
\omega = \text{fallback} \wedge \sigma \neq \gamma \wedge \varphi(\gamma) = \gamma \quad &= \\
\omega = \text{fallback} \wedge \\
(\sigma = \varrho \vee \\
\sigma \neq \gamma \wedge \varphi(\gamma) \neq \gamma \vee \\
\sigma \neq \gamma \wedge \varphi(\gamma) = \gamma) \quad &= \\
\omega = \text{fallback} \wedge \\
(\sigma = \varrho \vee \\
\sigma \neq \gamma \wedge (\varphi(\gamma) \neq \gamma \vee \varphi(\gamma) = \gamma)) \quad &= \\
\omega = \text{fallback} \wedge \\
(\sigma = \varrho \vee \sigma \neq \gamma) \quad &= \\
\omega = \text{fallback} \qquad (3.6)
\end{aligned}$$

The conclusion that all possibilities have been covered is obtained by the trivial combination of the results 3.3, 3.4, 3.5 and 3.6.

$$\begin{aligned}
\omega = \text{initializing} \vee \omega = \text{stable} \vee \\
\omega = \text{changing} \vee \omega = \text{fallback} \quad &= \\
\omega \in \{\text{initializing}, \text{stable}, \text{changing}, \text{fallback}\} \quad &= \quad \text{true}
\end{aligned}$$

Thus, we have shown that the arrival of an event is covered in all states.

Unambiguity

It is equally important that the state subdivisions are disjoint. This can be shown in a similar way by pairing the conditions in conjunctions and observing that the result is false.

It is easy to see that any pair containing a clause $\omega = x$ in one condition and $\omega = y$, $x \neq y$, in the other condition yields false. Therefore, we can restrict the investigation to pairs asserting identical unit states.

For the conditions pertaining to a command arriving in the "stable" state we have these three pairs, where the result is immediately evident from two adverse terms.

$$\begin{aligned}
r \neq \sigma \wedge \\
\tau(\sigma, r) \neq \sigma \wedge \\
r = \sigma \quad &= \quad false
\end{aligned}$$

$$
\begin{aligned}
r \neq \sigma \wedge \\
\tau(\sigma, r) \neq \sigma \wedge \\
r \neq \sigma \wedge \\
\tau(\sigma, r) = \sigma \quad &= \quad false
\end{aligned}
$$

$$
\begin{aligned}
r = \sigma \wedge \\
r \neq \sigma \wedge \\
\tau(\sigma, r) = \sigma \quad &= \quad false
\end{aligned}
$$

For the states fall-back and changing, the rule conditions contain the adverse terms $b = \text{true}$ and $b = \text{false}$.

The conditions for handling an event in the initial state are also easily verified:

$$
\begin{aligned}
\sigma = \varrho \wedge \\
\sigma \neq \varrho \quad &= \quad \text{false}
\end{aligned}
$$

3.5.6 Status Reporting

A *status report* $r = \langle s, e \rangle$ is an element of the cartesian product $V \times E$, where $E = \{\text{exec}, \text{auto}\}$. The set value "exec" means that the state s has been reached due to a command requesting a state, and "auto" indicates that the state change was not due to some command.

3.5.7 Fault Reporting

A *fault report* consists of the current fault state F. It is to be sent for each actual change in its value.

3.5.8 Summary of the Dynamic Model

The dynamic model is based on four states of a signal unit: *initializing, stable, changing* and *fall-back*.

There are two categories of inputs: *commands* requesting certain aspect, and *events* indicating the change of state for a single light point.

The initialization state is required for bringing a signal unit up from an undefined state where no information is available on any of its light points. This state is associated with a certain valid state, typically "halt".

The "stable" state is assumed whenever a commanded aspect has been reached. It is left due to a command or an event indicating a broken filament.

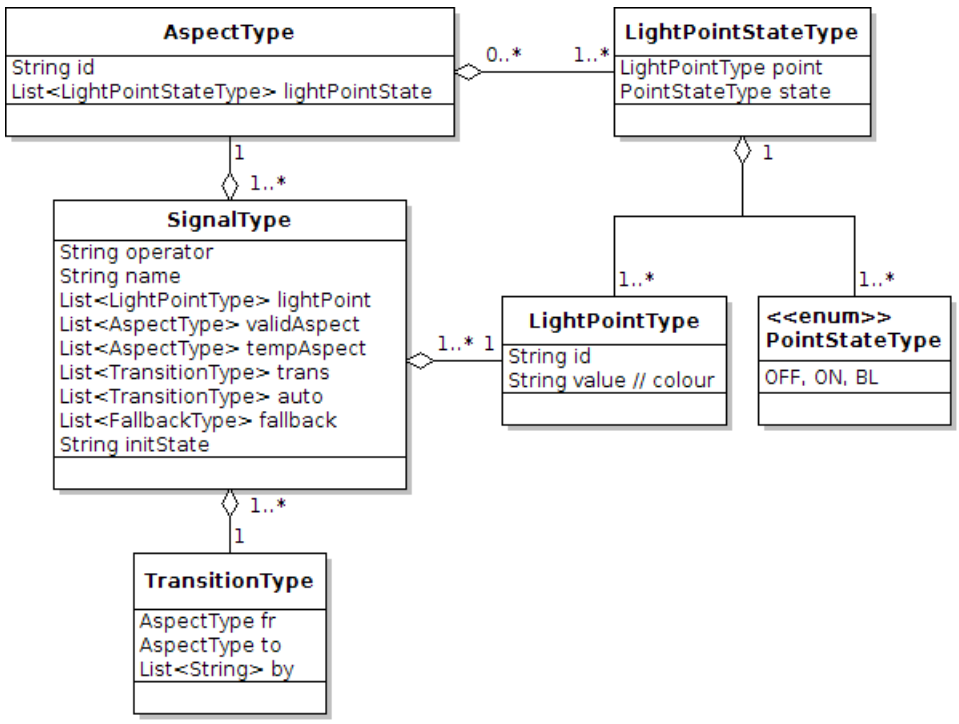

Figure 3.4: Signal and Aspect Type Data Model

For the "changing" state, the commanded aspect is set as the next goal. Events are duly noted, until the combined states of all light points indicate that the goal state has been reached, which returns the automaton into the "stable" state.

State "fall-back" is similar to changing, except that the handling of faults (any failure to reach a fall-back state) is somewhat different. If the fall-back state can be reached, the operational state reverts to "stable".

3.6 Glimpses of Code

3.6.1 The Static Data Model

The UML class diagram shown in Figure 3.4 illustrates the relationships between the essential types, i.e., `SignalType` with `AspectType` and `LightPointStateType` on the one hand and `TransitionType` on the other hand. These types are used for describing a signal with its aspects and transitional behaviour.

3.6.2 XML Data

The XML snippet given below shows the definitions for the Bulgarian main and distant signal. The list of light points is followed by the collection of valid aspects.

```
<Signal name="Main and distant signal" operator="BDZ"
        initState="h"
        graphic="BDZMainSignal">

  <LightPoint id="AMBER2">YELLOW</LightPoint>
  <LightPoint id="GREEN">GREEN</LightPoint>
  <LightPoint id="RED">RED</LightPoint>
  <LightPoint id="AMBER1">YELLOW</LightPoint>
  <LightPoint id="BAR">GREEN</LightPoint>
  <LightPoint id="WHITE">WHITE</LightPoint>

  <ValidAspect id="h"> <!-- halt -->
    <lightPointState point="RED"      state="ON"/>
  </ValidAspect>
  <ValidAspect id="mudu"> <!-- unlim./unlim. -->
    <lightPointState point="GREEN"    state="ON"/>
  </ValidAspect>
  <ValidAspect id="mud2"> <!-- unlim./speed 2 -->
    <lightPointState point="GREEN"    state="BL"/>
  </ValidAspect>
  <ValidAspect id="mud1">
    <lightPointState point="AMBER2"   state="BL"/>
  </ValidAspect>
  <ValidAspect id="mud0"> <!-- unlim./halt -->
    <lightPointState point="AMBER2"   state="ON"/>
  </ValidAspect>
  <ValidAspect id="m2du">
    <lightPointState point="GREEN"    state="ON"/>
    <lightPointState point="AMBER1"   state="ON"/>
    <lightPointState point="BAR"      state="ON"/>
  </ValidAspect>
  <ValidAspect id="m2d2">
    <lightPointState point="GREEN"    state="BL"/>
    <lightPointState point="AMBER1"   state="ON"/>
    <lightPointState point="BAR"      state="ON"/>
  </ValidAspect>
```

```
<ValidAspect id="m2d1">
  <lightPointState point="AMBER2"   state="BL"/>
  <lightPointState point="AMBER1"   state="ON"/>
  <lightPointState point="BAR"      state="ON"/>
</ValidAspect>
<ValidAspect id="m2d0">
  <lightPointState point="AMBER2"   state="ON"/>
  <lightPointState point="AMBER1"   state="ON"/>
  <lightPointState point="BAR"      state="ON"/>
</ValidAspect>
<ValidAspect id="m1du">
  <lightPointState point="GREEN"    state="ON"/>
  <lightPointState point="AMBER1"   state="ON"/>
</ValidAspect>
<ValidAspect id="m1d2">
  <lightPointState point="GREEN"    state="BL"/>
  <lightPointState point="AMBER1"   state="ON"/>
</ValidAspect>
<ValidAspect id="m1d1">
  <lightPointState point="AMBER2"   state="BL"/>
  <lightPointState point="AMBER1"   state="ON"/>
</ValidAspect>
<ValidAspect id="m1d0">
  <lightPointState point="AMBER2"   state="ON"/>
  <lightPointState point="AMBER1"   state="ON"/>
</ValidAspect>
<ValidAspect id="fs"> <!-- free for shunting -->
  <lightPointState point="WHITE"    state="ON"/>
</ValidAspect>
<ValidAspect id="fc"> <!-- on sight -->
  <lightPointState point="RED"      state="ON"/>
  <lightPointState point="WHITE"    state="BL"/>
</ValidAspect>

<!-- continued -->
```

Temporary aspects (not shown) are defined in the same way as valid aspects.

The next snippet contains some selected samples of transitions, from the "halt" state to some other state. The values of the attributes `@fr` and `@to` and of elements `by` are references of valid and temporary aspects, respectively.

The last transition does not require a temporary state.

```
  <!-- continuing -->
  <Trans fr="h" to="mudu">
    <by>mudu_t1</by>
  </Trans>
  <Trans fr="h" to="m2du">
    <by>m2du_t1</by>
    <by>m2du_t2</by>
  </Trans>
  <Trans fr="h" to="fs">
    <by>fs_t1</by>
  </Trans>
  <Trans fr="h" to="fc"/>
  <!-- continued -->
```

The signal definition is concluded with the mapping of the valid states to their fall-back states, as shown below.

```
  <!-- continuing -->
  <Fallback fail="h"    use="h"/>
  <Fallback fail="mudu" use="mud1"/>
  <Fallback fail="mud1" use="h"/>
  <Fallback fail="mud2" use="mud1"/>
  <Fallback fail="mud0" use="h"/>
  <Fallback fail="m2du" use="m2d1"/>
  <Fallback fail="m2d1" use="h"/>
  <Fallback fail="m2d2" use="m2d1"/>
  <Fallback fail="m2d0" use="h"/>
  <Fallback fail="m1du" use="m1d1"/>
  <Fallback fail="m1d1" use="h"/>
  <Fallback fail="m1d2" use="m1d1"/>
  <Fallback fail="m1d0" use="h"/>
  <Fallback fail="fs"   use="h"/>
  <Fallback fail="fc"   use="h"/>
</Signal>
```

3.6.3 Rules for Consistency Checks

Consistency check are advisable, since it is easy to overlook something while preparing the data by hand, and a data preparation tool may have a bug. The following consistency checks are implemented on static data items:

- All aspects must be unique.
- All transitions must be unique.
- There must be at least one transition to each valid state.
- There must be at least one transition from each valid state.
- If there is a transition into a temporary state, then there must be another one from it, with the same ultimate goal state.
- There must be a fallback definition for each valid aspect.

As an example, below is the rule checking the uniqueness of transitions.

```
rule "trans-unique"
when
  Transition( $fr: fr, $to: to, $by: by != null )
  Transition( fr == $fr, to == $to, by != $by )
then
  System.out.println( "duplicate Transition from " +
                      $fr.getName() + " to " +
                      $to.getName() );
end
```

3.6.4 The Dynamic Data Model

The dynamic data model somewhat different from the static data model, mainly because the static data model for a signal does not cater for the concurrent storage of various aspects, which is required for dealing with the dynamics of command and event handling. Moreover, it has to introduce classes for these inputs, which advance the state machine.

The UML class diagram shown in Figure 3.5 presents the simple class `SignalState`, which is essentially composed from 4 objects of class `Aspect` and a field storing the overall *operational* state. The type `Aspect` contains an identifier and two sets of light points, one for lit and another one for blinking lights. The aspects `sigma` (for the current aspect), `rho` (for the aspect requested by a command), `gamma` (the aspect to which the signal is currently "going") and `lambda` (the last previous aspect) can be considered as abstract refinements of the overall state.

The simple classes `Command`, `Change` and `Transition` (cf. Figure 3.6) complete the dynamic data model. The boolean field `forced` of `Command` is set when a command must be executed even when the signal's state is not

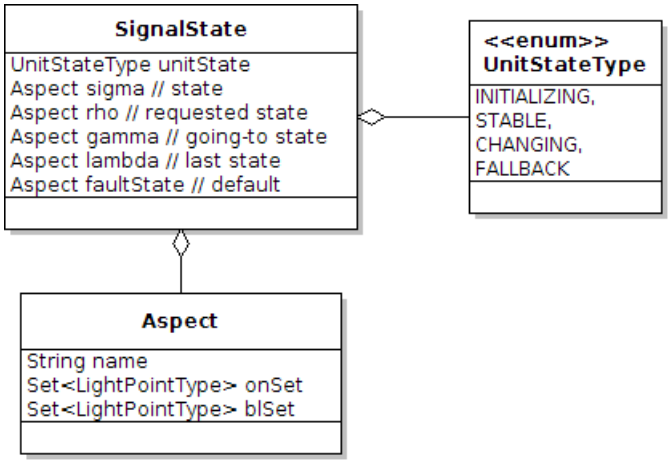

Figure 3.5: Dynamic Signal State

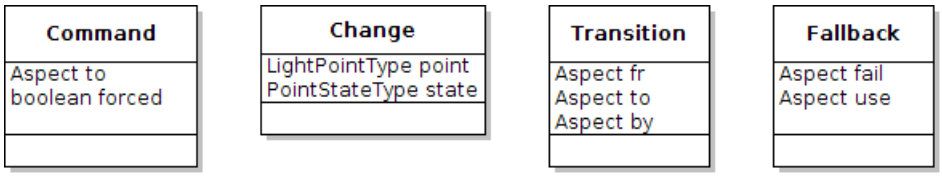

Figure 3.6: Command, Change and Transition

stable, as is usually done when the "halt" aspect is required.) Objects of class `Transition` define a single step in a transition between two aspects, i.e., the current valid or temporary) aspect, the ultimate goal and the very next (valid or temporary) aspect. `Fallback` facts associate a failing valid aspect with another one to be assumed instead.

Several transient facts are necessary for handling some technical issues in connection with defining the end of a state change (`Event`), faults (`NewFault`, `RepCheck`) and delays for realistic simulation (`DelayType`).

3.6.5 Selected Rules

The following rule accepts a command and initiates a transition from the current stable state to the requested aspect, making sure that an appropriate transition exists. (Otherwise the command would be refused and discarded by another rule.) The overall state and the state components `rho` and `gamma` need to be updated; the command fact can be retracted.

```
rule "command-stable-1"
when
```

```
  $cm: Command( $to: to )
  $ss: SignalState( unitState == UnitState.STABLE,
                    $sigma: sigma != $to )
       Transition( fr == $sigma, to == $to, $by: by )
then
  modify( $ss ){
    setGamma( $by ),
    setRho( $to ),
    setUnitState(  UnitState.CHANGING );
  }
  retract( $cm );
end
```

The next DRL snippet contains a rule reacting to an incoming change notification, i.e., the hardware interface has sensed that a light point has been lit.

```
rule "event-process-contents-on"
when
  $ch: Change( $point : point,
               $state : state == PointStateType.ON )
  $ss: SignalState( $sigma : sigma )
then
  retract( $ch );
  modify( $ss ){
    setSigma( $sigma )
  }
  theSignal.setAspect( $sigma ); // GUI
end
```

The third rule completes a transition, which can be due to a command or a fall-back. The necessary condition is that the cuurent state (`sigma`) equals the commanded state (`rho`), which is the same as the next state to be reached.

```
rule "event-changing-1"
when
  $ss: SignalState( unitState == UnitState.CHANGING ||
                                 == UnitState.FALLBACK,
                    $rho: rho, sigma == $rho, gamma == $rho )
then
  modify( $ss ){
```

```
    setUnitState( UnitState.STABLE ),
    setLambda( $rho );
  }
end
```

3.7 Examples

3.7.1 The Hungarian Main and Shunting Signal

The Hungarian combined main and shunting signal is defined by

$$\begin{aligned} S &= \{\text{off}, \text{on}, \text{bli}\} \\ U &= \{r, b3, a, b1, w\} \\ V &= \{h, s_0, s_1, s_2, s_3, f_s, f_o\} \\ h &= \langle \text{on}, \text{off}, \text{off}, \text{off}, \text{off} \rangle \\ s_0 &= \langle \text{off}, \text{on}, \text{on}, \text{off}, \text{off} \rangle \\ s_1 &= \langle \text{off}, \text{off}, \text{on}, \text{off}, \text{off} \rangle \\ s_2 &= \langle \text{off}, \text{off}, \text{on}, \text{on}, \text{off} \rangle \\ s_3 &= \langle \text{off}, \text{off}, \text{off}, \text{off}, \text{off} \rangle \\ f_s &= \langle \text{off}, \text{off}, \text{off}, \text{off}, \text{on} \rangle \\ f_o &= \langle \text{on}, \text{off}, \text{off}, \text{off}, \text{bli} \rangle \end{aligned}$$

The last two unit states are "free for shunting" and "on sight", respectively.

The following additional unit states are required for intermediary states when switching from h to s_0, s_1 or s_2:

$$\begin{aligned} h0a &= \langle \text{on}, \text{on}, \text{off}, \text{off}, \text{off} \rangle \\ h0b &= \langle \text{on}, \text{on}, \text{on}, \text{off}, \text{off} \rangle \\ h1 &= \langle \text{on}, \text{off}, \text{on}, \text{off}, \text{off} \rangle \\ h2a &= \langle \text{on}, \text{off}, \text{on}, \text{off}, \text{off} \rangle \\ h2b &= \langle \text{on}, \text{off}, \text{on}, \text{on}, \text{off} \rangle \\ hr &= \langle \text{on}, \text{off}, \text{off}, \text{off}, \text{on} \rangle \\ E &= V \cup \{h0a, h0b, h1, h2b, hr\} \end{aligned}$$

Note that $h1 = h2a$, and so we can omit $h2a$ from the list of states.

The transition function τ permits direct transitions to h from any aspect,

and from h to s_3 and f_o.

$$\begin{aligned} \tau(x,h) &= h \qquad \forall x \in V \\ \tau(h,s_3) &= s_3 \\ \tau(h,f_o) &= f_o \end{aligned}$$

Signalling from h to s_0, s_1, s_2 and f_s requires one or two intermediary steps.

$$\begin{aligned} \tau(h,s_0) &= h0a \\ \tau(h0a,s_0) &= h0b \\ \tau(h0b,s_0) &= s_0 \\ \tau(h,s_1) &= h1 \\ \tau(h1,s_1) &= s_1 \\ \tau(h,s_2) &= h2a \\ \tau(h2a,s_2) &= h2b \\ \tau(h2b,s_2) &= s_2 \\ \tau(h,f_s) &= hr \end{aligned}$$

The fall-back function is constant:

$$\varphi(x) = h \qquad \forall x \in V$$

3.7.2 The Bulgarian Main and Distant Signal

The Bulgarian combined main and distant signal is also used for permitting shunting movements. It is defined by

$$\begin{aligned}
S &= \{\text{off}, \text{on}, \text{bli}\} \\
U &= \{a2, g, r, a1, b, w\} \\
V &= \{h, s_0, s_1, s_2, s_3, f_s, f_o\} \\
h &= \langle \text{off}, \text{off}, \text{on}, \text{off}, \text{off}, \text{off} \rangle \\
m_u d_u &= \langle \text{off}, \text{on}, \text{off}, \text{off}, \text{off}, \text{off} \rangle \\
m_u d_2 &= \langle \text{off}, \text{bli}, \text{off}, \text{off}, \text{off}, \text{off} \rangle \\
m_u d_1 &= \langle \text{bli}, \text{off}, \text{off}, \text{off}, \text{off}, \text{off} \rangle \\
m_u d_0 &= \langle \text{on}, \text{off}, \text{off}, \text{off}, \text{off}, \text{off} \rangle \\
m_2 d_u &= \langle \text{off}, \text{on}, \text{off}, \text{on}, \text{on}, \text{off} \rangle \\
m_2 d_2 &= \langle \text{off}, \text{bli}, \text{off}, \text{on}, \text{on}, \text{off} \rangle \\
m_2 d_1 &= \langle \text{bli}, \text{off}, \text{off}, \text{on}, \text{on}, \text{off} \rangle \\
m_2 d_0 &= \langle \text{on}, \text{off}, \text{off}, \text{on}, \text{on}, \text{off} \rangle \\
m_1 d_u &= \langle \text{off}, \text{on}, \text{off}, \text{on}, \text{off}, \text{off} \rangle \\
m_1 d_2 &= \langle \text{off}, \text{bli}, \text{off}, \text{on}, \text{off}, \text{off} \rangle \\
m_1 d_1 &= \langle \text{bli}, \text{off}, \text{off}, \text{on}, \text{off}, \text{off} \rangle \\
m_1 d_0 &= \langle \text{on}, \text{off}, \text{off}, \text{on}, \text{off}, \text{off} \rangle \\
f_s &= \langle \text{off}, \text{off}, \text{off}, \text{off}, \text{off}, \text{on} \rangle \\
f_o &= \langle \text{off}, \text{off}, \text{on}, \text{off}, \text{off}, \text{bli} \rangle
\end{aligned}$$

The last two unit states are "free for shunting" and "on sight", respectively.

Temporary states are derived according to the requirement that no aspect that is more permissive than the requested aspect should be shown even when there is a lamp failure. This can be achieved by using one intermediary state when switching from halt (h) to all aspects except the ones where the main signal should signal "speed 2" ($m_2 d_u$ to $m_2 d_0$): here two intermediary states are required.

The transition function permits transitions from and to h for any aspect. Also, the distant signal permits transitions between its halt state and any other state, with the main signal remaining constant in any state except halt.

The fall-back function avoids halt in those cases where there is an aspect

that does not require the light points of the failing aspect.

$$\begin{aligned}
\varphi(m_u d_u) &= m_u d_1 \\
\varphi(m_u d_2) &= m_u d_1 \\
\varphi(m_2 d_u) &= m_2 d_1 \\
\varphi(m_2 d_2) &= m_2 d_1 \\
\varphi(m_1 d_u) &= m_1 d_1 \\
\varphi(m_1 d_2) &= m_1 d_1
\end{aligned}$$

For all other $x \in V$, we have $\varphi(x) = h$.

3.8 The Rule-Based Demonstration Kit

This section presents the rule-based implementation of the model. It uses the Drools Engine, but it should be noted that other rule engines would be equally well suited. An XML data file describes a signal; the transformation of its elements to facts is straightforward.

3.8.1 The Fact Types

The type `Aspect` contains an identifier and two sets of light points, one for lit and another one for blinking lights. Its subclasses `ValidAspect` and `TempAspect` do not define additional fields; they are used for restricting rule patterns to either subset.

Objects of class `Transition` define a single step in a transition between two aspects, i.e., the current (valid or temporary) aspect, the ultimate goal and the very next (valid or temporary) aspect. The `Fallback` facts associate a valid aspect, i.e., the one that cannot be achieved, with another one, i.e., the one to be assumed instead.

The dynamic state of the signal is contained in a fact of type `SignalState`.

Several transient facts deal with commands (`Command`), events (`Event`), faults (`NewFault`, `RepCheck`) and delays (`DelayType`).

Chapter 4

From XML to Facts

4.1 A Range of Scenarios

The two project studies presented in the preceding chapters have one things in common: XML is used to represent provisioning data.

But in one case there is a straightforward way of using the objects obtained by unmarshalling the DOM tree as facts in the application, and in the other case a transformation is required to create the objects for the dynamic model from the objects defined by the XML schema.

Laziness (allegedly being one of the three great virtues of a programmer) would urge us to use a single comprehensive model in any case, not requiring us to have this transition sitting between the static and the dynamic data model.

The following sections discuss several possible approaches.

4.2 Using the Dynamic Model for Static Data

Classes representing dynamic objects require fields for storing dynamic attributes. An XML definition can comprise dynamic fields in addition to the static ones that describe the immutable properties. But there are several caveats:

1. An XML schema (or any similar definition for XML data) is primarily an interface definition. Adding fields is in contradiction to this universally accepted usage.

2. The XML data file will not contain values for these additional fields; therefore they will have to be defined as optional if the XML definition should be used for validation.

3. An XML definition language like XML Schema is not too well equipped for defining particular data structures (beyond the ubiquitous sequence).

4. There may be more than one kind of processing required for a single XML data, which absolutely forbids this approach.

4.3 Deriving the XML Definition from Application Classes

Java annotations are a clever way of adding information to Java code that can go beyond processing. The Java Architecture for XML Binding (JAXB) is one standard technique, enabling programmers to define an XML structure on top of a set of Java classes, by adding various annotations to classes and properties.

There is even an annotation (`XmlTransient`) to prevent the mapping of a JavaBean property to XML representation. Moreover, the tool `schemagen` can create an XML schema from the annit class files.

However, there are two points to be considered:

1. Deriving the interface from application classes burdens this application with the responsibility for maintaining the XML definition. This is also somewhat in contradiction with the interfacing role XML is meant to fulfil.

2. There are several restrictions that must be observed when writing classes with JAXB annotations. Several of the JDK container classes do not have a straightforward XML binding, and workarounds tend to be tricky.

4.4 Data Modelling: Part of the Design

Especially the previous section has blatantly ignored that data modelling ought to be part of the design—not something that should be derived from application code.

UML, a widely used modelling language, features class diagrams, but apparently there is no way for relating a set of classes and class hierarchies to a data interface specification, such as an XML schema, and the usual notation doesn't leave room for relating one to the other.

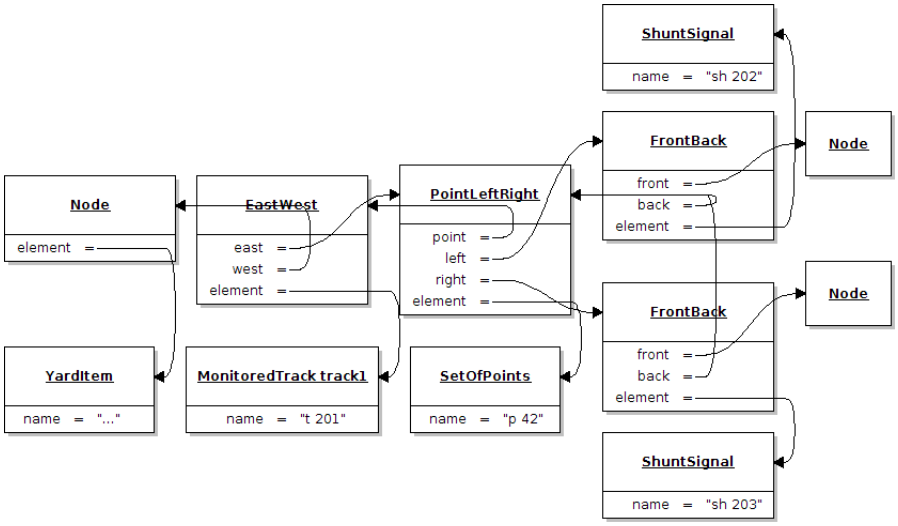

Figure 4.1: Shunting Yard: Topology and Elements

The approach that has worked best for me is to develop a design that optimally supports application needs. Then, from the resulting data model, develop a view that collects static (provisioning) data into a cross-section.

The UML object diagram in Figure 4.1 shows a design separating structural objects for linking railway interlocking elements from the actual objects destined to maintain element state. Rules (or any other business logic) can be kept separately, on the one hand for iterating over the topological connection and on the other hand for performing state transitions on element data.

For the static data describing a shunting yard, however, there is no need for this separation. Element data is conveniently collected according to element types, including references to neighbouring elements. Classes according to the classic *Builder* design pattern are responsible for creating the application objects from the ones containing the static provisioning data.