Domain-Specific Languages – Notation for Experts

Wolfgang Laun, Thales Austria GmbH

October 26, 2011

1 Fall In — Introduction

Drill commands is a widely known example of a *domain-specific language* (DSL), i.e., a relatively small set of sentences or phrases, with well defined meaning and, occasionally, variable parameters, intended for use with a specific group of persons, who have been trained to understand it. As a subset of some natural language, the command language deliberately relinquishes the power of the full language, but its very formal structure makes it easy to understand.

In connection with computer programming, a DSL is a programming language dedicated to a particular problem domain or a specific implementation technique, or both. More specifically, for decisioning systems, a DSL should provide a notation that lets you express *rules* in a way easily understood by experts for the rules' domain.

This paper presents three case studies from quite distinct applications. The first one demonstrates the separation of expert-defined logic from the general control flow of a Java program for graphic rendering of a symbol set for representing elements in varying states. The second one shows how a descriptive notation (ultimately even Java annotations) can be used for defining rules for data validation, and analyzes how to implement it using facts in a rule-based system. The third example outlines the development of a DSL, as close to natural language as possible, for the operating rules of an interlocking system.

2 Attention — Some Facts About DSLs

While everybody seems to agree that a DSL is the opposite of a general-purpose programming language, some other properties are not commonly acknowledged. A DSL...

1. requires, as a *formal language,* a definition (a grammar) and a translator into some target language that is "understood" by the executing system.

2. may not be able to express all that's possible with the underlying language.

3. can not be used by experts totally alien to computer programming (except in trivial cases).

4. is not necessarily based on a natural language.

5. may still require the skills of a good programmer, especially when designed to simplify some specific programming task.

6. causes, on first implementation, additional development effort, and *may* increase the maintenance effort.

Truth values	Conditions
Action selection	Actions

Figure 1: The four quadrants of a decision table

In short: *a DSL is neither a silver bullet nor a free lunch.*

So why do people use DSLs? The next sections present three reports of DSL usage, each of which provides part of the answer.

3 Report 1: Decision Tables

3.1 What is a "Decision Table"?

Classic decision tables associate truth value combinations with action selections. The basic structure is shown in Figure 1.

The upper half defines the rules for deciding between several alternatives, where exactly one must be selected. *Conditions* are boolean expressions. *Truth values* are represented by letters ('t' and 'f', 'y' or 'n') and the hyphen ('-') for "don't care". Thus, every column selects a number of truth value combinations, ranging from a single one (without a hyphen) to all possibilities, where all entries are hyphens. Ordering columns by increasing number of hyphens is essential for a correct evaluation. Rules with an equal number of "don't care" entries must have them in the same positions.

The lower half defines action statements, one per row. Whenever an action should be executed due to the selection of a rule, the intersection of row and column is marked with an 'x'; otherwise it is marked with a hyphen.

Example 3.1 presents the decision table for the logic controlling a light switch with sensors registering natural light and the state of the light bulb.

Example 3.1: Light Switch Logic

```
y n - - | daylight.isBright()
y n y - | switch.isOn()
y n n - | bulb.isLit()
========|=====================
- x - - | switch.setOn(true);
x - - - | switch.setOn(false);
- - x - | bulb.alarm();
```

An extended version of the decision table format permits discrete values (e.g., integers or enumerations) as entries of a single row in the upper left quadrant. The type of the expression must be compatible with the values, so that the boolean expression required for determining the rule column can be derived as a comparison between the expression and the value, which is replaced by a 'y'; thus, each discrete value results in a separate boolean condition row.

3.2 Use Case: Graphic Rendering

The graphical user interface for a signalman (of the Austrian Federal Railways) shows more than 20 different elements such as signals, tracks, switches, etc. For each element, the graphic representation must take several boolean status properties into account; their

number might be as low as 5, but typically it is between 10 and 20, and it tops 25 for a couple of complicated ones.

The operator's specification was provided by showing examples, featuring arbitrary status combinations. As an example, consider Figure 2, showing the ones provided for a switch, which has 20 boolean status values.

To be sure, the number of actually possible states requiring a graphic representation is always much smaller than the theoretical number 2^n for n states; nevertheless, a test run of programming the required logic (that was not planned as such) produced ample proof that the traditional approach is too taxing for many programmers. It was simply too difficult to distill the required logic from exemplary pictures and to hand-code the its intricacies.

This led to the idea of writing the logic that determines visibility, shape and colour of an element symbol's constituents as a decision table. Moreover, it was evident from the customer's specification that a decision table would be understood by both sides: railway experts are capable of reading these tables, even though they aren't programmers, and for programmers it should be a cinch.

3.3 The Decision Table Translator

With the GUI application being written in Java, it was thought best to provide a decision table translator with Java as the target language. To keep it simple, the translator was designed to locate a decision table embedded in a Java comment and delimited by keywords.

As we were using Eclipse as our IDE, the translator was provided as an Eclipse plugin. It is activated whenever the Java source file is saved. It locates embedded decision tables and expands them into Java code that is inserted immediately after the decision table. (The remainder of the previous expansion is, of course, discarded.) The expanded text is written to disk.

The algorithm for selecting a rule is based on the representations of y's and n's as bit sets. (Implementing them as `long` values is sufficient for 64 conditions.) Using the set value resulting from the current value of all conditions, the "rule", a decision table column, is determined; its ordinal number is used in a switch statement to select the branch where the x-marked actions of the corresponding rule are collected. The declarations of the supporting objects are also inserted by the translator, near the end of the Java source file containing the decision tables.

As an illustration of a decision table from the presented use case, Listing 3.1 shows the logic for representing the position of a switch. (Note that the actions use colours determined in another decision table, i.e., the one distinguishing between occupied and free and various route usage states.)

Listing 3.1: Desision table for switch position states

```
DecTabBegin SwitchPosition
Determine the position for ElementSwitch.
heartDashed is initialized to false.
leftShort and rightShort are initialized to true.
n n n n n n n - | mySwitch.getUndefined()
n y n n n - y - | mySwitch.getDisturbed()
y y n n n - - - | mySwitch.getForced()
- - y n n - - - | mySwitch.getMoving()
- - - y n y - - | mySwitch.getLeft()
```

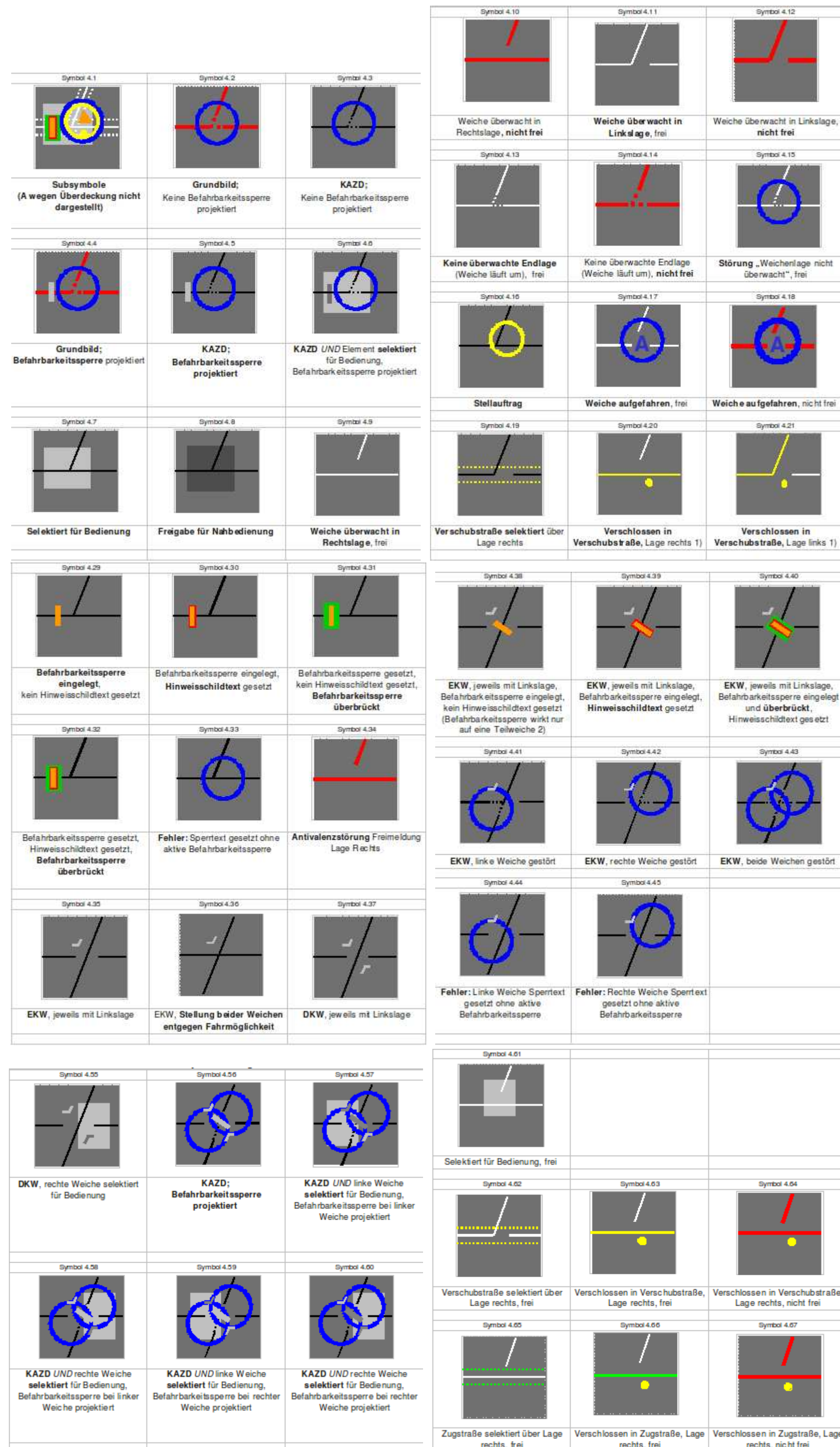

Figure 2: Selected Switch States.

```
- - - n y y - - | mySwitch.getRight()
================|========================
- - x - - x x x | heartDashed = true;
- - x - - x x x | heartCol = trackColor;
x x - - - x x x | dRing = Color.BLUE;
- - - - - x - x | isUndef = true;
x x - - - - - - | theA  = Color.BLUE;
- - - x - - - - | leftShort  = false;
- - - - x - - - | rightShort = false;
- - - x - - - - | rightColor = secondTrackColor;
- - - - x - - - | leftColor = secondTrackColor;
- - - - x - - - | rightColor = trackColor;
- - - x - - - - | leftColor = trackColor;
DecTabEnd SwitchPosition
```

3.4 Rating

Although not a DSL in the true sense of the word–it does not make use of any specific vocabulary–decision tables provide an excellelent means for solving the presented use case and similar ones. (In another application, decision tables were employed to implement sets of finite state machines.) Frequently enough, filling in a decision table shows that the logic can be divided into two or more separate tables which can be evaluated independently. Each rule in a completed table relates a set of states to an action sequence, which collectively guarantees full coverage of all states. Maintenance activities such as the addition of a new element property with corresponding changes in the graphic rendition turned out to be easy. The most important thing, however, is that the decision tables can be read by both programmers and railway experts.

4 Report 2: Defining Data Validation

4.1 Basic Data Validation

Frequently enough, the input data is not to be trusted. Consequently, a standard requirement is to add some logic ensuring that the data is *correct*. Assuming the usual structuring in records with distinct fields, this correctness may depend on

- individual fields (of a record) having an admissible value;
- the absence of contradictions by field values within a record;
- the absence of contradictions by field values between records.

There is virtually no limit for the complexity of data correctness rules, but there is a limited number of checks for fields of some simple type: numeric types, string, or enumeration. How can we implement such checks in a manageable way, especially where the user of an application should be able to control such checks?

4.2 Rules for Checking Field Values

One of the reasons Rule Based Systems (RBS) are praised is that rules separate "business logic" from the technical parts of an application. Considering that value checks are usually one facet of this business logic it makes sense to use rules for these checks as well; all the more so if the RBS is employed for the actual processing stage anyway.

The very simple approach is to write a rule for each and every field that warrants a check, as shown in Example 4.1.

Example 4.1: Range check of an integer value

```
rule "Check Item.code"
when
    $item: Item( $c: code < 1 || > 10 )
then
    System.out.println( "Invalid code " + $c + " in " + $item );
end
```

Clearly it is far from convenient to write dozens of similar rules; therefore a more convenient notation should be provided.

4.3 Rules for Value Checks

Consider the variable parts in the rule in Example 4.1:

- the class containing the field,
- the name of the field,
- the lower and upper bound,
- the error message text.

This data can be collected in an object of class `IntRangeCheck`, extending the abstract class `Check`. The base class deals with the class and the field, using reflection to determine the `java.lang.reflect.Field`. Class `IntRangeCheck` stores the bounds and implements method `isValid`, as shown in Example 4.2.

Example 4.2: Implementation of a range check

```
  public boolean isValid( Object object ) throws Exception {
    Object value = field.get( object );
    if( value == null ) return false;
    int intValue = (Integer)value;
    return lower <= intValue && intValue <= upper;
  }
```

Notice that checks ensuring that a value equals one out of several alternatives or that a string matches a regular expression can be represented by other subclasses of `Check`.

Inserting `IntRangeCheck` objects as facts we can now write a single rule that performs all integer range checks on all fields in all facts in the Working Memory.

Example 4.3: Integer range checks on Item

```
rule "Check all Item integer ranges"
when
    $item: Item()
    $cond: IntRangeCheck( clazz == ( Item.class ) )
    eval( ! $cond.isValid( $item ) )
then
    System.out.println( $cond.getError() + " " + $item.toString() );
end
```

Since the rule does not make any specific references to `Item` attributes and by observing that its class is readily available from the object, we can rewrite the rule to perform integer range checks on *arbitrary objects*. This is shown in Example 4.4.

Example 4.4: Integer range checks

```
rule "Check all integer ranges"
when
    $fact: Object()
    $cond: IntRangeCheck( clazz == ( $fact.getClass() ) )
    eval( ! $cond.isValid( $fact ) )
then
    System.out.println( $cond.getError() + " " + $fact.toString() );
end
```

Moreover, the rule doesn't refer to any properties of the subclass `IntRangeCheck` but only to field `clazz`, inherited from its base class. Thus, we can generalize the rule once more, as shown in Example 4.5.

Example 4.5: All checks

```
rule "All checks"
when
    $fact: Object()
    $cond: Check( clazz == ( $fact.getClass() ) )
    eval( ! $cond.isValid( $fact ) )
then
    System.out.println( $cond.getError() + " " + $fact.toString() );
end
```

Consequently, all checks on all fields on all facts are effected by a single rule, provided these checks have been defined by `Check` facts. This means that we use the fact matching capabilities of a RBS to achieve the simple task of combining each fact with each check definition for one of its fields.

Bringing this up in a talk on Domain Specific Languages might be considered a hoax or a mistake–after all, where is the DSL?

4.4 A Declarative DSL

What remains to be done and made available to the user, i.e., the person defining the checks, is a notation for these checks. Since the algorithm is provided with one of the subclasses of `Check`, defining the checks is a purely declarative task, by specifying class, field, and a pair of bounds, a list values, a regular expression or whatever else is required by the checking procedure.

Using XML is one way of representing such declarations. Class `Check` and its subclasses can be augmented with annotations from `javax.xml.bind.annotation` so that JAXB can be used for unmarshalling an XML file containing check definitions.

4.5 Rating

We have seen that a DSL providing metadata about data validity can be purely declarative. The actual decision whether a data item is valid or not is implied by the metadata type:

Property	`Signal`	`Track`	`Switch`	`Goal`	*Values*
Indication	yes	no	no	no	halt, clear
Lock	yes	no	yes	yes	true, false
Interdiction	no	yes	yes	no	true, false
Position	no	no	yes	no	left, right(,. . .)
Track circut	no	yes	yes	no	free, occupied
Fault	yes	no	yes	no	true, false

Table 1: Elements and Their Properties

range check, value list, regular expression, etc. The actual coding of the check is provided by implementation of the abstract method `isValid`. Notice, however, that this could also be implemented in a rule, according to Example 4.4, where the type of the `Check` subclass is used with the pattern selecting the definition of the check.

Looking at the rule in Example 4.5, we can see that the pattern matching facilities of a RBS aren't really put to the test. Assuming that an application can somehow keep track of the objects where validity checks are to be made, it is a simple programming exercise to find and match `Check` objects, and to call the appropriate `isValid` method.

If the responsibility for the validity checks does not rest with the user of the application it is certainly preferable to let software developers define constraining facets with the data definition, i.e., the class code itself. For this, Java provides annotations, and the aforementioned check definitions can equally well be written as such, using a set of annotation classes as a DSL.

5 Report 3: Operating Rules for a Signal Box

5.1 The Language of Requirements

Railway buffs may know it: the language used for talking about operating a signal box is peculiar, as it continues to use its quaint terms from obsolete technologies and due to tradition. Speaking a country's language well doesn't mean that you can fully understand the vernacular of its railway operator. This means that vendors developing software for railway operators must make extra efforts to come to grips with the operator's requirements.

Requirements that can be read and understood by both sides have to be in some natural language. This leaves the implementors with the burden of translating the words into code, a step that suffers from the lack of rigour of any natural language.

This is where a DSL using phrases from a natural language and capable of expressing operational rules would fill a gap. The project presented here demonstrates that this is indeed feasible. It uses Drools' framework for implementing a DSL and mapping it to rules in DRL, the Drools Rule Language.

5.2 The Data Model

The trackside elements are `Signal`, `Track` and `Switch`. A virtual element `Goal` is used alone or in combination with a `Signal` for marking a position where a train route ends. Table 1 shows the allocation of properties to these elements. Property values change due to orders from an operator or as a result of sensor data from surveillance circuits.

There are two forms of operator orders, represented by objects of class `OperatorOrder`: element orders and route orders. These orders must be validated and transformed into

internal commands of type `ElementCommand` and `RouteCommand`.

A series of listings present condensed versions of the model's classes and interfaces. Listing 5.1 contains the types for the command interface.

Listing 5.1: Operator and command classes

```
class Operator {
  String id;
  Operator( String id ){...}
}

class OperatorOrder {
  String code;
  String elId1;
  String elId2;
  String opId;
  OperatorOrder( String code, String elId,
                 String opId ){...}
  OperatorOrder( String code, String elId1, String elId2,
                 String opId ){...}
}

class ElementCommand {
  Element element;
  CommandCode code;
  Operator operator;
  ElementCommand( CommandCode code, Element element,
                  Operator operator ){...}
}

class RouteCommand extends ElementCommand {
  Element goal;
  RouteCommand( CommandCode code, Element start, Element goal,
                Operator operator ){...}
}

enum CommandCode {
  HALT      ( "H", 1 ),
  CLEAR     ( "C", 1 ),
  THROW_OVER( "T", 1 ),
  LOCK      ( "L", 1 ),
  UNLOCK    ( "U", 1 ),
  INTERDICT ( "I", 1 ),
  PERMIT    ( "P", 1 ),
  SET_ROUTE ( "R", 2 );

  String code;
  operands;
  CommandCode( String code, int operands ){...}
}
```

The class hierarchy for the representation of elements is based on the abstract class `Element` and uses the aforementioned interfaces in order to provide abstractions for elements with common properties. Condensed versions of all classes and interfaces are shown in Listing 5.2.

Listing 5.2: Element classes and interfaces

```
abstract class Element {
  private String id;
  protected Element( String id ){...}
}

enum OnOff {
    UNDEF, OFF, ON;
}

public interface Fault {
  public OnOff isFaulty();
  public void setFaulty( OnOff faulty );
}

public interface Interdiction {
  public OnOff isBarred();
  public void setBarred( OnOff value );
}

public interface Lock {
  public OnOff isLocked();
  public void setLocked( OnOff value );
}

public interface TrackCircuit {
  public OnOff isClear();
  public void setClear( OnOff value );
}

class Signal extends Element
  implements Lock, Fault {
  enum Indication {
    STOP, CLEAR;
  }
  Goal goal;
  Indication indication;
  OnOff locked;
  OnOff faulty;
  Signal( String id ){...}
}

class Switch extends Element
  implements TrackCircuit, Lock, Interdiction, Fault {
```

```
  enum Position {
    RIGHT, LEFT, UNDEFINED, FORCED;
  }
  OnOff clear;
  Position position;
  OnOff locked;
  OnOff barred;
  OnOff faulty;
  Switch( String id ){...}
}

class Track extends Element
  implements TrackCircuit, Interdiction {
  OnOff clear;
  OnOff barred;
  Track( String id ){...}
}

class Goal extends Element
  implements Lock {
  Signal signal;
  OnOff locked;
  Goal( String id ){...}
}
```

5.3 Developing the Railway DSL

5.3.1 Defining a DSL in Drools

Drools provides a generic DSL translator that uses regular expressions for detecting and translating DSL phrases into Drools Rule Language (DRL) clauses. Two sets of patterns are distinguished by the leading bracketed keyword: one for the condition and another one for the consequence. They are meant to be used between the keywords defining the rule structure, i.e., `when`, `then` and `end`.

```
[when]there is a car=$car: Car()
[then]print success=System.out.println("Success!");
```

Constraints can be added by following the convention to begin the DSL phrase and the matching pattern with a hyphen. The resulting text is inserted in the parentheses of the preceding pattern.

```
[when]- with colour green=colour == "green"
```

Parts of a regular expression can be set up as captures, so that the string occurring in this place can be interpolated into the replacement string.

```
[when]there is an? {thing}=${thing}: {thing}()
[when]- with colour {colour}=colour == "{colour}"
```

Below is a complete rule using this simple DSL:

```
rule "red bike"
when
```

```
    there is a bike
    - with colour red
then
    print success
end
```

Here is the result after the transformation:

```
rule "red bike"
when
    $bike: bike(colour == "red")
then
    System.out.println("Success!");
end
```

Replacing DSL phrases by regular expressions is applied line by line. For each line it is repeated until there is no match with a DSL phrase.

5.3.2 Processing Operator Orders

At this stage, we can develop the first part of the DSL, for writing the rules dealing with operator orders. The requirements are simple:

- An operator order must contain
  - a valid operator identification,
  - a valid command code,
  - one or two valid element identifications, according to the command code.

  Any error must be relayed to the operator.
- A correct operator order must be transformed into an internal element or route command.

The first check, written as a rule, should contain the condition

```
    $oc: OperatorOrder( $code: code, $opId: opId )
    not Operator( id == $opId )
```

Expressed in English, the condition could be read “when there is an operator order but no matching operator” and this is how we begin our DSL definition:

```
[when]There is an (?i:operator order)=
      $order: OperatorOrder( $code: code, $opId: opId )
[when]but no matching operator=
      not Operator( id == $opId )
```

The next requirement is a matching command code, which might be expressed by the following condition:

```
    $oc: OperatorOrder( $code: code, $opId: opId )
    Operator( id == $opId )
    not CommandCode( code == $code )
```

`CommandCode` was introduced in Listing 5.1. There is, of course, no reason why enum objects should not be inserted as facts—they provide an excellent means for asserting the correctness of fact data. For this rule, we have to add a couple of additional translations for DSL phrases:

```
[when]and a matching operator=
      $operator: Operator( id == $opId )
[when]and no matching command code=
      not CommandCode( code == $code )
```

Even at this stage, we can see that the number of required translations tends to explode. We need one for a missing operator and one for a matching operator, and then one for an invalid command code and one for a matching command code, and this will continue for similar situations. But there is a remedy: we can simply eliminate "there is a" and "and a", and translate the negative phrases "and no" and "but no" to the conditional element keyword "not" and use the remaining words for a translation to the appropriate pattern. Thus, our DSL definition is reduced to these entries:

```
[when](?:and)? [Tt]here is an?=
[when]and( with)? a=
[when](?i:and|but) no=not
[when](?i:operator order)=
      $order: OperatorOrder( $code: code,
                             $elId1: elId1,
                             $elId2: elId2,
                             $opId: opId )
[when]matching operator=$operator: Operator( id == $opId )
[when]matching command code=$cc: CommandCode( code == $code )
```

It's a good idea to provide bindings for all properties of the operator order. This lets us use the same operator order phrase for the next rule where we check for an incorrect number of element identifications.

```
    $order: OperatorOrder( $code: code, $id1: elId1, $id2: elId2,
                           $opId: opId )
    $operator: Operator( id == $opId )
    CommandCode( code == $code,
                 operands != ($id1 != null ? ($id2 != null ? 2 : 1) : 0) )
```

The restriction contains an expression determining the number of operands. We could create a unique phrase just for this `CommandCode` pattern, but this seems wasteful. Therefore we reuse the existing phrase and add a sub-phrase, just for the restriction.

```
[when]- with an incorrect number of elements=
      operands != ($id1 != null ? ($id2 != null ? 2 : 1) : 0) )
```

The complete conditions may now be written as

```
when
    there is an operator order
    and with a matching operator
    and no matching command code
then
```

```
when
    there is an operator order
    and with a matching operator
    and a matching command code
    - with an incorrect number of elements
then
```

The next rules must check that one or both elements exist. In English, you would say "the first element" or "the second element", and we could define translations for both. But we'll demonstrate another feature a translator might implement: the extraction of an integer from the text matched in a capture. Given the DSL entry

```
[when]matching {ord} element=
      $el{ord!num}: Element( id == $id{ord!num} )
```

we can write a set of conditions like these, where the number 1 is stripped from the trailing letters:

```
    when
        there is an operator order
        and with a matching operator
        and a matching command code
        but no matching 1st element
    then
```

Since the second element identification may be absent we must provide a clause ensuring that it isn't null, but otherwise we can reuse the phrases from the conditions for the first element.

```
[when]- with a {ord} element=elId{ord!num} != null
[when]- without a {ord} element=elId{ord!num} == null

    when
        there is an operator order
        - with a 2nd element
        and with a matching operator
        and a matching command code
        but no matching 2nd element
    then
```

The consequences for these checking rules are simple. First, a message has to be sent back to the operator and, second, the invalid operator order must be discarded. If there is no valid operator, we need a surrogate. The DSL definitions shown below are based on methods that should be implemented in class `Operator`.

```
[then][Ss]end message to( the)? operator {text}=
      $operator.message( {text} + ": " + $order );
[then][Ss]end message to( the)? main operator {text}=
      Operator.getMainOperator().message( {text} + ": " + $order );
[then][Dd]iscard( the)? {thing}=retract( ${thing} );
```

Only trivial additions to the DSL are necessary for completing the first set of rules with the rules performing the transformation of an operator order to an element or route command. The required definitions and the rules are given below.

```
[when]- with the correct number of elements=
      operands == ($id1 != null ? ($id2 != null ? 2 : 1) : 0) )
[then][Cc]reate the corresponding element command=
      insert( new ElementCommand( $cc, $el1, $operator ) );
[then][Cc]reate the corresponding route command=
      insert( new RouteCommand( $cc, $el1, $el2, $operator ) );

rule "transform element command"
when
    there is an operator order
    - without a 2nd element
    and with a matching operator
    and a matching command code
    - with the correct number of elements
    and a matching 1st element
then
    Discard the order
    Create the corresponding element command
end

rule "transform route command"
when
    there is an operator order
    - with a 2nd element
    and with a matching operator
    and a matching command code
    - with the correct number of elements
    and a matching 1st element
    and a matching 2nd element
then
    Discard the order
    Create the corresponding route command
end
```

5.3.3 Updating Element States

For element state changes we need a few very simple classes; these are shown in Listing 5.3. Class `SwitchState`, for instance, contains three fields with status attributes. This reflects a property of the surveillance circuitry, which reacts to any state change by sending the full range of an element's current state.

Listing 5.3: Classes representing element state changes

```
abstract class ElementState {
  String id;
  ElementState( String id ){...}
}

class SignalState extends ElementState {
  OnOff faulty;
  SignalState(String id, OnOff faulty){...}
```

```
}

class TrackState extends ElementState {
  OnOff clear;
  TrackState(String id, OnOff clear){...}
}

class SwitchState extends ElementState {
  OnOff clear;
  OnOff faulty;
  Position position;

  SwitchState( String id, OnOff clear, Position position,
               OnOff faulty ){...}
}
```

There are just three requirements for processing state changes:

- There must be an element whose type suits the kind of state change, identified by the `id` property.
- An element mismatch must be logged.
- The values in a correct state message replace the matching element's current status properties.

Two additional definitions are sufficent for the conditions. One is used for creating patterns matching any of the "state" message facts, and another one for the matching element. For the consequence, there is a definition for creating an error message and a cooperating pair of definitions for composing the "modify" statement. Note the convention for using a word such as "switch" in the DSL and its modification by the capture-transforming function `ucfirst` to fit the convention for Java class names, i.e., `SwitchState`.

```
[when]{type} state=
      $state: {type!ucfirst}State( $id: id )
[when]matching element=
      {type!ucfirst}( id == $id )
[then]Element state error {text}=
      Operator.getMainOperator().message( {text} + ": " + $state );
[then]update the element=
      modify( $element )\{ \}
[then]- setting(?: the)? {property} from the {entity} message=
      set{property!ucfirst}( ${entity}.get{property!ucfirst}() )
```

Note that the last entry has a leading hyphen. Within a consequence, this means that the resulting expanded text is inserted as another term in the body of the preceding "modify" statement.

The rules for checking and executing a `SwitchState` can now be written like this:

```
rule "switch state for unknown element"
when
    there is a switch state
    but no matching element
```

```
then
    Element state error "switch state for unknown element"
    discard the state
end

rule "switch state update"
when
    there is a switch state
    and a matching element
then
    update the element
    - setting faulty from the state message
    - setting clear from the state message
    - setting the position from the state message
    discard the state
end
```

The rules for `Track` and `Signal` are simple variations of the ones shown for `Switch`.

5.3.4 Element Command Processing

There are only two generic requirements for element command processing.

- A command must be valid for the addressed element.
- The element must not be in a state contradictory to the execution of the command.

Table 1 shows how properties are associated with element types. This suggests that the required checks can be done after successfully matching an element via the type—a class or an interface—sufficient to distinguish elements with the property associated with the command from all others, and then by testing the method returning the salient property.

All checks proceed according to a singular pattern, which can be written as shown below:

```
when
    there is an element command
    - with the code x-ify
    and the commanded element
    - is not a x-able element
then
    send failed command message "cannot x-ify element"
```

Given a method `execute(CommandCode code)`, a single trivial rule would be sufficient if we could be sure that all preconditions are met. To achieve this, all rules checking these conditions must be given a higher priority than the one executing the command. In our DSL, we use a third kind of definitions, i.e., the one for keywords. A keyword translation is performed independently from and preceding all other translation. Thus, a definition of keyword `precondition` permits us to distinguish rules checking preconditions from processing rules. The required DSL definitions are given below. (Note the regular expression after the colon in a capture: it restricts what can be matched by the capture.)

```
[keyword]precondition {title}=rule {title}\nsalience 100
```

```
[when](?i:element command)=
      $command: ElementCommand( $element: element,
                                $code: code,
                                $operator: operator )
[when]- with the code {code:\w+}=
      code == CommandCode.{code}
[when]or the code {code:\w+}=
      || code == CommandCode.{code}
[when](and)? the commanded element=
      Element( this == $element )
[when]- is not a {what} element=
      eval( ! ($element instanceof {what!ucfirst}) )

[then][Ss]end failed command message {text}=
      $operator.message( {text} + ": " + $element.getId() );

[then]execute command=
    modify( $element )\{ execute( $code ) \}
    retract( $command );
```

The single rule for checking that a "lock" or "unlock" command is given for a switch, signal or goal element can now be written as shown below, together with the simple rule for executing any command.

```
precondition "commmand: lock/unlock for switch, signal, goal"
when
    there is an element command
    - with the code LOCK or the code UNLOCK
    and the commanded element
    - is not a lock element
then
    send failed command message "cannot lock/unlock element"
end

rule "execute element command"
when
    there is an element command
then
    execute command
end
```

The "throw over" command for switches necessitates a more complex rule. Clearly, the switch must not be occupied, it may not be locked and, moreover, it must not be part of an established route. Since this rule includes facts for routes, we'll have to look at these first.

5.3.5 Route Command Processing

The main purpose of interlocking systems is the safe alignment of train routes, a sequence of elements that starts with some signal and ends at a goal, which might be associated with another signal or mark a point at the closer end of a line track. Figure 3 shows a track layout for a very small station. The six signals (red semicircles) next to the tracks 201,

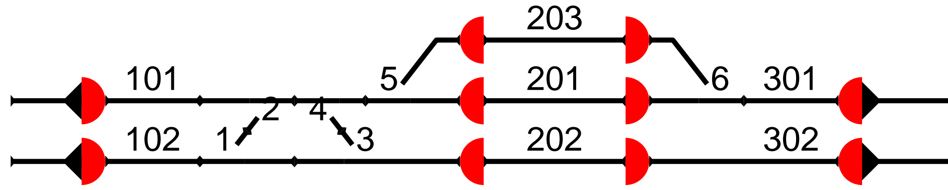

Figure 3: A simple track layout

202 and 203 may act as goals, as well as the four goal points, which are shown as black arrow tips in the direction of the train movement. Excluding detours, 18 simple routes are possible, for instance, the one via track 101, the switches 2, 4 and 3, and ending with track 202.

The data model for routes is shown in Listing 5.4. A `RouteDefinition` contains a list of all elements that are part of the route; for switches the required position is stored as well. Next, a `Route` is simply the combination of a route definition with a state. Classes `Token` and `SwitchToken` will be used for temporary facts that are attached to elements, marking them as being part of some route.

Listing 5.4: Classes for routes and their definition

```
class RouteDefinition {
  Map<Element,Position> positions;
  List<Element> elements;
  RouteDefinition( Signal start, Goal goal ){...}
}

class Route {
  enum State {
    SET, PROVING, DISSOLVING;
  }
  RouteDefinition definition;
  State state;
  public Route ( RouteDefinition definition ){...}
}

class Token {
  Route route;
  Element target;
  public Token( Route route, Element target ){...}
}

class SwitchToken extends Token {
  Position position;
  SwitchToken( Route route, Element element, Position position ){...}
}
```

The requirements for dealing with a `RouteCommand`[1] are manifold, starting with the simple check that the route is defined, comprising several checks against route elements

[1]For this discussion of a DSL development a considerably reduced set of requirements has been selected.

being in a forbidding state and, finally, ending with setting the route and clearing the start signal.

- A route must be defined, according to provisioned data.
- The route command must be refused if any of the following conditions is true:
  - The route goal is locked.
  - An interdiction is "on" for any route element.
  - A switch within the route is locked and in the adverse position.
  - Any route element is currently used by another route.
- An accepted route command must result in setting the route, i.e., all elements are now part of this route.
- Switches in the wrong position must be thrown over.
- The start signal is cleared for a route, which then enters the state "proving" as soon as all of the following conditions are true:
  - The start signal is not locked.
  - The start signal is not faulty.
  - All rail elements are free.
  - All switches are in the correct position.

Some additional DSL entry definitions are necessary for dealing with route commands and matching route definitions:

```
[when]set route command=
      $command: RouteCommand( $code: code == CommandCode.SET_ROUTE,
                              $start: start, $goal: goal,
                              $operator: operator )
[when]matching route definition=
      $def: RouteDefinition( start == $start, goal == $goal,
                             $elements: elements )
```

The definitions describing the inhibiting element states refer to properties of the elements in some `RouteDefinition`. Some new DSL entries are required for them, as shown below. The most interesting pattern deals with switches locked in the wrong positions. It uses `from $def.switches` to extract the `Switch` elements contained in the route definition, which makes them available for testing with the preceding pattern. Also noteworthy is the use of `Token`, which, when referring to an element within the commanded route, would indicate that the element is currently already part of another route. For the consequences, we can reuse "discard the command" but we need another entry for notifying the operator about a failed route command.

```
[when]locked goal=
      $badel: Goal( this == $goal,
                    locked == OnOff.ON )
[when]barred interdiction=
      $badel: Interdiction( this in ($elements),
                            barred == OnOff.ON )
```

```
[when]switch locked in the wrong position=
      $badel: Switch( locked == OnOff.ON,
                      position != ($def.getPositionOf($badel)) )
              from $def.switches
[when]element used in a(nother)? route=
      Token( $badel: target memberOf ($elements) )

[then][Ss]end invalid route message {text}=
      $operator.message( {text} + " for route: " +
          $start.getId() + " to " + $goal.getId() );

[then][Ss]end failed route message {text}=
      $operator.message( {text} + " on " +
           ((Element)$badel).getId() + " for route: " +
           $start.getId() + " to " + $goal.getId() );
```

Listing 5.5 shows most of the route checking rules, as they can be written with these additional DSL entries.

Listing 5.5: DSLR rules for checking routes

```
precondition "route: no definition"
when
    there is a set route command
    and no matching route definition
then
    send invalid route message "no such route"
    discard the command
end

precondition "refuse defined route if goal locked"
when
    there is a set route command
    and a matching route definition
    and a locked goal
then
    send failed route message "goal locked"
    discard the command
end

precondition "refuse route if switch locked in wrong position"
when
    there is a set route command
    and a matching route definition
    and a switch locked in the wrong position
then
    send failed route message "switch locked in wrong position"
    discard the command
end

precondition "refuse route if element already used by route"
```

```
when
    there is a set route command
    and a matching route definition
    and an element used in another route
then
    send failed route message "conflicting route"
    discard the command
end
```

Finally, we can design the rules for setting a route and commanding the required switch movements, and for monitoring the route so that the start signal can be cleared as soon as prevailing circumstance permit. The first one requires a new consequence action for establishing the route, which is done by inserting a suitable `Route` object and `Token` objects for marking its elements.

```
[then][Ee]stablish set route=
      Route route = new Route( $def );
      insert( route );
      for( Object element: $elements )\{
          Token token = ((Element)element).makeTokenForRoute( route );
          insert( token );
      \}

rule "establish route"
when
    there is a set route command
    and a matching route definition
then
    establish set route
    discard the command
end
```

Given that after the firing of rule "establish route" all switches are now marked with a `SwitchToken`, the rule for launching a "throw over" command for all switches in the position adverse to the required one is not too difficult, but none of the conditional elements after the pattern for `Route` and certainly not the consequence statement have been used before, and therefore we have to add another couple of DSL entries.

```
[when]free switch in this route in adverse position=
      SwitchToken( $target: target, route == $route, $reqPos: position )
      $switch: Switch( this == $target, clear == OnOff.ON,
                       $actPos: position )
      eval( $reqPos.isAdverseTo( $actPos ) )

[then]enter a throw-over command for the switch=
      insert( new ElementCommand( CommandCode.THROW_OVER, $switch,
                                  Operator.getAutoOperator() ) );

rule "command switches in route"
when
    There is a set route
    and a free switch in this route in adverse position
```

```
then
    enter a throw-over command for the switch
end
```

For writing the rule responsible for clearing the start signal we need a pattern matching a set route and a couple of additional conditions.

```
[when]and its start signal
      Signal( this == $start )

[when]and all of it's track circuits are free=
      not ( Token( route == $route, $target: target )
            and
            TrackCircuit( this == $target, clear != OnOff.ON ))

[when]and all switches are in required position=
      not ( SwitchToken( route == $route, $target: target )
            and
            $sw: Switch( this == $target,
                         eval( ! $route.isInPosition( $sw )))

[then]set the start signal to (?i:CLEAR)=
      modify( $start )\{ setIndication( Indication.CLEAR ) \}

[then]change the route state to proving=
      modify( $route )\{ setState( State.PROVING ) \}

rule "set signal to clear"
when
    there is a set route
    and its start signal
    - is not locked
    - is not faulty
    and all of it's track circuits are free
    and all switches are in required position
then
    set the start signal to clear
    change the route state to proving
end
```

More rules are necessary for dissolving a route due to a train movement, and for retracting a route by an operator command. This is left as an exercise for the reader.

5.4 Rating

The set of DSL rules presented here is quite impressive. Anyone understanding English and the project domain can read and understand the rules. Apart from a little basic know-how about rule interpretation by fact matching and firing and the way objects are represented by sets of properties no programming skills are required. This means that the primary goal of this project could be achieved.

As an additional benefit, the DSL is independent from the underlying Java code and the way constraints need to be written.

Some valuable insights were learned from this DSL development.

1. A DSL may not be able to express all that can be written in the underlying language. If, for instance, the translator is based on regular expressions it is not possible to write recursive structures. This, however, is not a handicap where the DSL is designed to cover a subset of the underlying language.

2. Complex conditional elements that cannot be composed using the generic DSL elements may still be provided by "tailored" DSL phrases. (As an ultimate resort, the DSL mechanism ought to let you add lines written in native rule language code, as Drools' DSL translator does.)

3. Developing and maintaining a DSL is not as simple as it might seem at first glance, especially if you want to create a reusable set of DSL phrases. Given the parsing algorithm, phrases must be sufficiently distinguishable from each other. As a remedy, consider developing more than one DSL and separate your DSL rules accordingly into different compilation units, each written in its own DSL.

4. DSL authors – programmers or domain experts – need good documentation instructing them about the way the individual DSL phrases are meant to be used. There may be built-in dependencies between a phrase representing a condition and phrases provided for the consequence, or condition phrases must be used in some specific order, etc.

5. A DSL may still leave ample opportunities for writing inefficient or downright wrong rules. Mentoring by an experienced rule programmer should mitigate this risk.

6 Fall Out — Conclusion

The presented triple of use cases for DSLs demonstrates that there is a wide range of "notations for experts." Common to all of them is that a *specific* language is better *understood* by domain experts. If generic parts of the DSL implementation can be reused, savings in development and maintenance effort are to be expected. In contrast, a high degree of specialization may not permit reuse; in this case the benefit might come from closing the gap between requirements and implementation.

In any case, a DSL should always be discussed as an option for the implementation of an application using RBS or, more widely, decisioning systems.