Event Processing in an Object-Oriented Rule-Based System

Wolfgang Laun

Thales Rail Signalling Solutions GesmbH
Wolfgang.Laun@thalesgroup.com, Scheydgasse 41
A-1210 Vienna, Austria

Abstract. This paper describes concepts being developed for processing events in a rule-based system, to be deployed in an embedded system that is part of a railway interlocking system, used as a safety-motivated secondary software channel. Events are either status notifications or operator commands, both of which operate on elements in a railway interlocking plant. It is shown that an object-oriented hierarchy of fact types results in a compact set of rules for processing both event types, illustrated by a prototype implementation.

Key words: railway interlocking, event processing, production rule system, object-orientedness

1 Introduction

Event processing by rule-based systems has gained increased attention, resulting in the addition of time-dependent conditional expressions and other time-related functions to such systems. This focus of this paper, however, is centered on the frequent observation that events (being viewed as transient facts) have to be correlated with a considerable variety of static facts representing "tangible" entities of the application domain, which, at first glance, results in a large number of rules pairing event types with entity types.

Being able to view more or less similar objects as carriers of identical property sets is one of the major advantages of object-oriented systems. It is the goal of this paper to show the benefit of this paradigm for event-processing production rule systems.

Section 2 provides some background information on the use of rules in a specific application domain, i.e., railway interlocking systems. Section 3 presents an analysis of the static entities of that domain, providing the groundwork for section 4, which discusses the actual implementation of event processing by rules.

A demonstration program provides ample proof that the proposed concepts permit a more compact set of rules, with all the resulting benefits.

2 Using Rules in a Railway Interlocking System

The central tasks of an interlocking system (IXL) are the setting of safe routes for train and shunting movements and the fail-safe operation of the interlocking plant's equipment. Signals, sets of points,[1] axle counters and other equipment are continuously monitored via interfaces that relay all state changes to the system's central controller and enact element commands such as changing a signal's aspect or throwing over a set of points.

Safety requirements must be met according to SIL 4, the highest level defined by EN 50128 ([2]). One of the software technology features used by Thales to achieve compliance with this requirement is to use a second software channel implemented as a rule-based production system, operating in parallel to the primary channel implemented according to the conventional procedural paradigm.

Current implementations of the Thales IXL LockTrac 6131 ([4]) use PAMELA ([1]), a rule-based system using the Rete algorithm ([3]), tightly coupled with the programming language CHILL ([6]). Both compiler implementations, based on an earlier version of the Recommendation Z.200, are not object-oriented, which has been identified as one major source for the considerable number of rules. Trackside equipment not only exhibits properties which can be fit into a natural class hierarchy; subsection 3.2 elaborates why alternative "views" via interfaces are desirable as well.

Research into object-oriented rule based systems such as Drools ([5]), implemented in Java, is intended to lay the foundation for an object-oriented approach in this domain. In addition to being used in a second software channel, rules are very well suited for a straightforward implementation of the railway operator's operational requirements for interlocking, which are typically formulated according to the when-condition-then-action pattern.

3 A Type System for Interlocking Elements

3.1 Generic Features

This section presents an object-oriented analysis of a characteristic set of interlocking elements as employed by a typical railway operator. Although elements may vary with respect to their relevant properties, analyzing these element groupings results in a Java type structure with the following characteristics:

- All element types are subtypes of a single abstract type, `ElemType`, with a set of attributes used for identification.
- The first level of subclasses provides a taxonomy according to the connection properties of the elements, joining them into an abstract graph with a maximum node degree of 4.

[1] U.S.: switch, turnout

– The decisive traits of the individual elements are determined by a composition of attributes that indicate certain capabilities (such as local operation) or reflect specific states, either induced by external events (e.g., being occupied by a vehicle) or resulting from the operation of the IXL itself (e.g., being locked).

Both, the joint possession of a topological pattern and the commonness of operational attribute sets, prepare the ground for the application of singular methods and rules. It is, therefore, the primary goal of an object-oriented analysis of any particular element group to identify these types, and correlate them and the operational requirements and restrictions.

The analysis is based on static and dynamic element properties, the latter being the target of status notification events, and all of them playing a role when processing command events.

Since an interlocking system may be seen as a representative of a wider class of production or service facilities, i.e., plants, the applicability of this analysis strategy should have a much wider scope.

3.2 The ÖBB Element Set

The Natural Type Hierarchy. According to the general strategy outlined in the preceding section, an analysis based on topological properties results in the structure presented in Figure 1. The uppermost level of subclasses below `ElemType` is given by the four abstract classes `FrontBackType`, `EastWestType`, `PointLeftRightType`, and `NelSerNwrSwlType`.[2] They classify elements according to their topological connections, where most element have two neighbours, sets of points have three, and intersections connect to four elements. Notice that although both tracks and signals are connected to two neighbours, a signal's orientation is essential. Further subclassing reflects operationally distinctive properties, e.g., a track may be on the line, or insulated, or not insulated (omitted in the figure). Connections not pointing to another element are due to dead-end tracks and "open" ends of line tracks. Here, a `NullType` element with a single connector is added, mainly to avoid the tedious null test.

It is, however, equally important to view element subsets according to their operational characteristics, resulting in a second, orthogonal type hierarchy.

Dynamic Element States. Dynamic operational attributes are conveniently represented as enumeration types. The low-level representation in the message protocol prescribed by the railway operator uses boolean values for atomic events. Operational rules, however, are based on *aggregated states* resulting from the combined consideration of $n > 1$ such values. Furthermore, the number of meaningful states that have to be distinguished is usually smaller than 2^n, requiring the mapping of all invalid combinations to a single "undefined" state.

[2] Points of the compass (NE,...SW) are intended to express opposites, not true orientations; l(eft) and r(ight) as viewed from the centre.

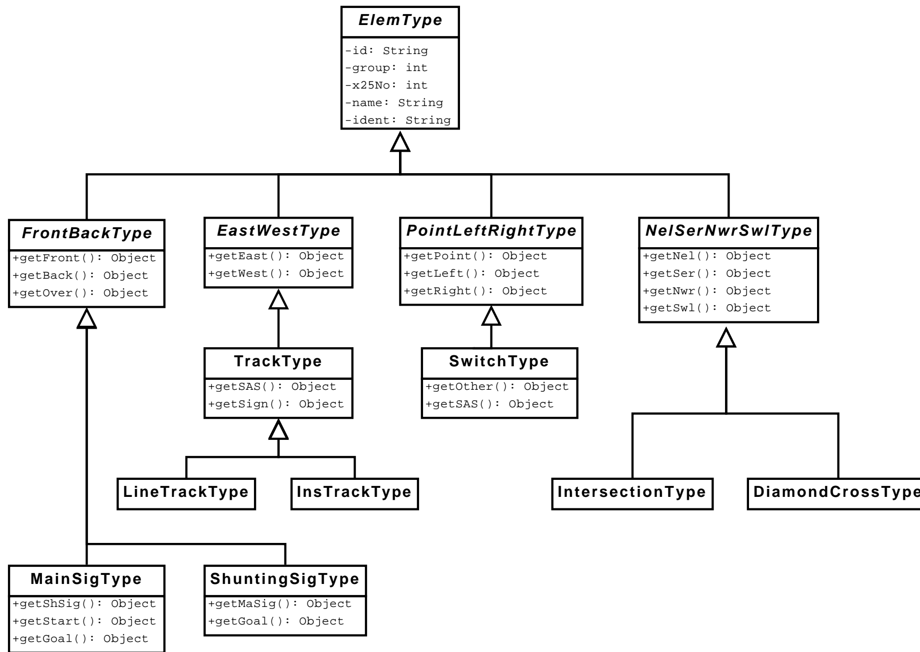

Fig. 1. Interlocking plant element type hierarchy (subset).

Also, almost all element categories feature a boolean status called “status not up-to-date” (with the German acronym “KAZD”), which overrules all operational states by forcing them to the undefined state. It is obvious that the formulation of operational rules would be quite cumbersome (and definitely not portable) if based on boolean values.

Aggregated status values have a straightforward application, namely when determining the *alternative status display* (ASD) for an element, a textual representation of an element’s state, intended for the operator of a signalling box. For this, an aggregated status is mapped to an enum constant from a single, comprehensive enum type identifying all “sentences” shown in the ASD dialog box.

Approximately 40 aggregated operational status types derive from the basic event messages defined in the ÖBB specification. They will result in as many interface definitions, used in the process of mapping boolean event patterns to aggregated events. A selection of typical and also exceptional examples is presented below.

A *single lock* is put on elements from several categories to make them temporarily unavailable for any interlocking activity. The enum class is defined as shown below, with the ASD enum constants being used as arguments for the enum constructor.

```
public enum SingleLock {
  UNLOCKED( AltStatus.SINGLE_NOT_LOCKED ),
```

```
    LOCKED( AltStatus.SINGLE_LOCKED ),
    UNDEFINED( AltStatus.SINGLE_LOCKED_KAZD );

    private AltStatus altStatus;
    SingleLock( AltStatus as ){ altStatus = as; }
    public AltStatus getAltStatus(){ return altStatus; }
    // ...
    public static SingleLock comp( Einzelsperre e ){
      if( e.getKazd() ) return UNDEFINED;
      return e.getEinzelsperre() ? LOCKED : UNLOCKED;
    }
}
```

Method `comp` is responsible for computing the aggregated status value from the pattern of boolean events from the interface `Einzelsperre` (German for "single lock"), which defines methods to access the boolean event values "KAZD" and "Einzelsperre". This interface is implemented by all event message types for elements featuring a single lock.

For the aggregated status in the element objects, a `SingleLock` attribute is included in all subtypes of `ElemType` implementing the interface `Locking`, which defines the getters and setters for `SingleLock`. It is convenient that some of these combined states occur jointly as this reduces the number of required interface definitions. The example shows that `SingleLock` always occurs together with `Interlock`.

```
public interface Locking extends AnyElement {
  public SingleLock getSingleLock();
  public void setSingleLock(SingleLock singleLock);
  public Interlock getInterlock();
  public void setInterlock(Interlock interlock);
}
```

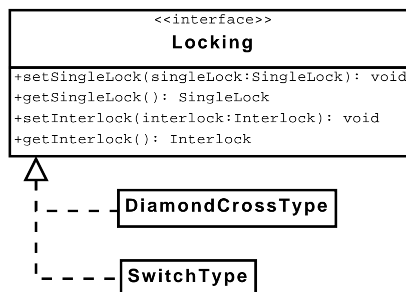

Fig. 2. Implementing classes of interface `Locking`. (In the full element set, three more classes implement this interface.)

The resulting relation between the interface and its implementing element classes is shown in Figure 2. About ten similar interfaces are defined as subinterfaces for `AnyElement`, which defines the attributes identifying an element. The most important subinterfaces are listed below.

- `AnySignal`: This interface combines getters and setters for the attributes `Aspect`, `LampDist`, and `SingleLock`, all of which are required for any kind of signal, i.e., main signal, shunting signal and protection signal.
- `AnyTrainSignal`: This interface extends `AnySignal` with the attributes representing states for signals providing train movement authority.
- `Locking`: This interface combines the attributes `Interlock`, according to an element's route interlock state, and `SingleLock`, indicating the single lock state.
- `Monitoring`: The attribute `FreeOccupied` is used with elements that are associated with an axle counter or track relay for monitoring the occupied condition of a section of running.
- `Moving`: The attribute `Position` mirrors the current state of orientation for sets of points and similar elements.

4 Event Processing

Given the fact class and interface hierarchy presented in the preceding section, we may now look at event processing itself.

4.1 Status Notifications

Status Notifications Processing Outline. The typical stages for processing a status notification are as follows:

1. The status notification message is inserted as a fact into Working Memory.
2. Matching the element addressed in the status notification with an element fact in Working Memory triggers a rule updating the element's status and retracting the status notification.
3. A low-priority rule would catch a status notification for an unknown element and handle it suitably.

Implementation by Rules. The basic representation of status notification messages is straightforward since most of them follow a uniform pattern that is partially implemented in the abstract base type `Message` and its (abstract) subclass `MsgElem`. With message attributes conveniently grouped into interfaces, the rules are very simple, matching a message type with the element it is addressed to. Below is the rule for processing a status notification for a track segment.

```
rule updateMg31
when
  $m : MsgMg31( $elNo : elNo )
```

```
  $e : InsTrackType( elNo == $elNo )
then
  modify( $e ){
    setFreeOccupied( FreeOccupied.comp( $m, true ) ),
    setInterdiction( Interdiction.comp( $m ) ),
    setRouteUsage( RouteUsage.comp( $m ) )
  }
  retract( $m );
end
```

4.2 Commands

Command events come in two groups: single element commands and route commands. We shall discuss a representative of each.

Command Processing Outline. There are four distinct steps for processing a command event:

1. The command is inserted as a fact into Working Memory. Ideally, the class representing the command message can be used for pattern matching; if not, an auxiliary class must provide a structure containing suitably transformed fields.
2. After a successful check of the command's validity, an internal fact supporting command processing is created. The original command may be retracted.
3. The command is processed by one or more operative rules. Complex commands, i.e., commands involving more that one element, typically require staged processing, with one stage selecting the relevant set of elements, the next one performing the checks and the last one applying the required operations to some or all of these elements.
4. In the last stage, any temporarily inserted facts must be discarded again, and a response is prepared and sent as a network message. Also, if the command didn't pass the initial tests, this stage provides rules for removing the command and sending a rejection.

Alternative Status Display. The Alternative Status Display (ASD) is a simple command, directed at a single element, determining its status according to the specification for "Method Safe Operation". This element status consists of a set of discrete values, each of which represents the state of a certain element property. Given that many properties occur with two or more elements, the computation of the ASD result *per element* would result in extensive code replications. Therefore, the concise approach of evaluating elements via suitably defined interfaces was adopted, even though this requires repeated rule activations and RHS executions, and the additional effort for collecting the results of these executions.

The temporary fact `ReplyASD` carries the standard element identification. Also, it features a field of type `java.util.Set` where the results of the individual

executions can be collected. Notice that a set is not ordered, which concurs with the indeterministic firing of the individual rules.

As a simple example, an ASD for a set of points requires the evaluation of the properties reflected by the attributes `freeOccupied`, `interlock` and `singleLock`, among others. Matching the `ReplyASD` with the identified element according to the interface `Monitoring` provides access to the selected element's `freeOccupied` attribute, whereas a match according to the interface `Locking` permits the evaluation of attributes `singleLock` and `interlock`.

```
rule getMonitoringASD
when
  $r : ReplyASD( $mg : group, $elNo : elNo, $ss : statusSet )
  $m : Monitoring( group == $mg, elNo == $elNo,
                   $fo : freeOccupied )
then
  $ss.add( $fo.getAltStatus() );
end

rule getLockingASD
when
  $r : ReplyASD( $mg : group, $elNo : elNo, $ss : statusSet )
  $m : Locking( group == $mg, elNo == $elNo,
                $sl : singleLock, $il : interlock )
then
  $ss.add( $sl.getAltStatus() );
  $ss.add( $il.getAltStatus() );
end
```

The computation of the resulting enum constant is invariably handled by the method `getAltStatus` of the enum type representing a property. Typically, this method simply returns the value from the overall enum type `ASDType` that corresponds to the local attribute value. (Refer to the code example showing enum class `SingleLock` on page 5.)

Route Command: Subsidiary Signal. The railway operator's specification for the command setting the subsidiary signal demands that the elements in the anticipated route of the train must meet a considerable set of conditions, according to element category. This route must be selected according to a predefined train route, which may be the regular route from start to goal, or one of the defined detours. The elements to be checked are the elements on the selected route. The information defining a route consists of the start element, the goal element, and the set of decision switches, entered via point, combined with the required direction.

Failing checks must be brought to the operator's attention. The required result consists of a list of element identifications combined with an indication of the noncompliant state. This list is sent as the regular reply to the ASD command.

The first stage of the execution of a route command has to establish the set of elements to be checked, based on the start and goal elements and the full set of decision switches. The auxiliary fact types used in this stage merit a closer look:

- A `Collector` fact is created which maintains a reference to the command and contains the list of elements and status values, and the set of general operator alerts.
- A `Movement` fact plays the role of a vehicle progressing from the track in front of the start signal to the goal element. It keeps track of the current position and references the `Collector` element.
- `SubSigRouteToken` and its subclasses (e.g., `SubSigRouteSwitchToken` for switches) are types for "token" facts created along the route, whenever the `Movement` fact matches with another element on the route. A token fact points to the element it belongs to and to the `Collector` fact.

Creating the route tokens proceeds by having rules match the `Movement` fact field pointing to the "current" element with a representative from one of the topological categories. Calling method `goAcross` of that element returns the next element, to replace the current element of the movement. A simple example, the single rule for advancing on track and signal elements which implement the Interface `GoAcross` is shown below. Somewhat more complex is the rule for traversing a set of points, from point, which must use the next direction from the route definition, kept in the field `startGoalConn` of the `Movement` object.

```
rule advanceGoAcross
when
  $m : Movement( $sgc : startGoalConn, $g : goal,
                 $p : previous, $c : current != $g,
                 $co : collector )
  $e : GoAcross( this == $c )
then
  insert( new SubSigRouteToken( $co, $sgc, $c ) );
  modify( $m ){
    setPrevious( $c ),
    setCurrent( $e.goAcross( $p ) )
  }
end

rule advanceFromPointToLeftRight
when
  $m : Movement( $sgc : startGoalConn, $go : goal,
                 $pr : previous, $cu : current != $go,
                 $si : switchIndex, $co : collector )
  $e : PointLeftRightType( this == $cu, $pt : point == $pr )
then
  LeftRight lr = $sgc.getDirList().get( $si );
```

```
    insert( new SubSigRouteSwitchToken( $co, $sgc, $cu, lr ) );
    modify( $m ){
      setPrevious( $cu ),
      setCurrent( $e.goAcross( $pr, leftRight ) ),
      setSwitchIndex( $si + 1 )
    }
end
```

Advancing the Movement fact terminates when the route's goal element is reached, where the Movement fact is retracted. The rule set checking elements with an attached token may now fire, indeterministically. Once again, the rules are mostly written with patterns referencing interfaces, thereby potentially matching with elements from a range of categories. As an example we present the rule that detects that some running section is occupied.

```
rule checkFreeOcc
when
  $t : SubSigRouteToken( $co : collector, $el : element )
  $m : Monitoring( this == $el,
                   freeOccupied != FreeOccupied.FREE )
then
  $co.getElemBoxStatusList().add(
      new ElemBoxStatus( $el, BoxStatus.NOT_FREE ) );
end
```

A low-priority rule concludes the first phase of processing, by sending a command reply from the data collected in the Collector fact, for the operator to acknowledge. A second processing phase will begin after the operator's confirmation. Any element update happening while the operator's confirmation is still expected sets the flag changed in the Collector fact to true, indicating that the command will be aborted. Finally, a low-priority rule cleans up by removing all SubSigRouteToken elements.

5 A Demonstration Program

The element set outlined in section 3 and the processing of the status notifications and commands described in section 4 were implemented as a Java program in combination with Drools.

The demo program loads, from an XML data file, a set of elements describing some specific shunting yard as facts into working memory. Incoming status notifications and commands are fed as event facts to the engine. Command responses (normally sent to an operator station) are here presented in a pop-up window, which must be acknowledged to let input processing proceed.

6 Conclusion and Further Work

Fact Types. The presented object and interface type structure guarantees simple, orthogonal rules. Considering a simple use case, we can compare the rules for extracting ASD results we find that the object-oriented approach needs 20 rules with distinct consequences, 16 of them with just two patterns and 4 requiring a third pattern. No if-statements are used in the consequences. When rules cannot be written with interfaces as fact types, a corresponding set of rules would require one rule per element type, i.e., 21 rules, but the consequences would have about as many if-statements and sections of duplicated code.

More complicated rule groups show even better savings. For iterating over the elements of a route, from the start signal to the goal element, we need just 5 orthogonal rules, whereas the conventional implementation takes 19 rules. Similar savings arise from the rules checking the conditions in a route. Asserting that all running sections are free requires either 7 rules (or as many if-statements) instead of just a single rule.

Event Processing in a Rules Engine. Event processing is done by asserting event messages as facts, firing after matching static facts. Complex events are handled conveniently using abstractions defined as interfaces, avoiding code replications.

Multi-stage processing of events involving several facts is done by following a pattern where elements are selected according to their relative position, with indeterministic firings of elements marked with appropriate tokens. The possibility of using subclasses also simplifies the handling of temporary tokens.

Further Work. Commands acting on some trackside equipment require *time supervision*, with timeouts resulting in an internally generated element. Also, a third category (in terms of persistence) of facts is required for representing routes between having been set and before being dissolved; event processing will then not only affect elements but also the semi-persistent route fact the element currently belongs to.

References

1. Barachini, F.: PAMELA: A Rule-Based AI Language for Process-Control Applications. IEA/AIE (Vol. 2), pp. 860–867 (1988)
2. DIN EN 50128; VDE 0831-128:2001-11: Bahnanwendungen – Telekommunikationstechnik, Signaltechnik und Datenverarbeitungssysteme – Software für Eisenbahnsteuerungs- und Überwachungssysteme. Deutsche Fassung EN 50128:2001
3. Forgy, C.L.: RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem, Artificial Intelligence, 19(1), pp. 21–37 (1982)
4. Fuß, W.: Tailored Solutions for Safety-Installations in the Loetschberg Tunnel – A Project with Importance for the Trans-European Rail Traffic, DATE'08 Proceedings, pp. 21–25 (2008)
5. JBoss: Drools 5, http://www.jboss.org/drools/

6. ITU: CHILL – The ITU-T Programming Language, CCITT/ISO/IEC International Standard ISO/IEC 9496, ITU Recommendation Z.200, Geneva (2003)