Configuration requirements from railway interlocking stations

(problem instance)

Andreas Falkner
Gerhard Fleischanderl

Siemens Austria, Program and Systems Engineering
Erdberger Laende 26, A-1030 Vienna, Austria
andreas.falkner@siemens.at / gerhard.fleischanderl@siemens.at

Abstract

Railway interlocks are a domain that provides large and complex configuration tasks. We describe a problem instance that focuses on software configuration, which is a distinguishing feature of the domain. Parameters for the operating system of a railway interlock have to be configured and re-configured.

1 Introduction

Configurators are already applied to different domains of industry products. For instance, telecommunication systems are among the products frequently treated with configurators [Fleischanderl *et al.*, 1998].

Railway interlocking stations are a large and complex domain for configurator applications. That may be one reason why only few papers on configurators in the railway domain were published. Configurations for interlocks usually have many elements and highly complex relationships. The configuration of a large interlocking station may comprise more than 50,000 objects. Furthermore, the configuration of software is an important part of an interlock configuration.

This paper briefly describes the domain of railway interlocks. The main part is the description of a problem instance for configurators. The relevant parts of an interlock configuration are presented with their objects, attributes, relationships, and constraints [Mittal and Frayman, 1989].

2 The domain

2.1 Railway interlocks

Railway interlocking stations are produced by few suppliers and are sold to industry customers, like railway operators or engineering companies that build railway stations.

Interlocks are used for monitoring and control of railway stations. The safe and secure operation of trains on a railway line depends on the correct and fail-safe functioning of the interlocks. A configurator has to provide the data for producing, assembling and parameterizing a customer-specific interlock.

Railway interlocks are not mass products. That is why interlocks are only partly designed for mass customization. There are still many small components and details to be chosen and parameterized for a specific interlock. These sub-systems have to be configured for an interlock:

- Hardware: racks, frames, cards, cables, wires.
- Software: data for parameterizing the operating system of the interlock.
- Communication equipment: hardware and software for the connection between interlocks.

2.2 Configuration elements

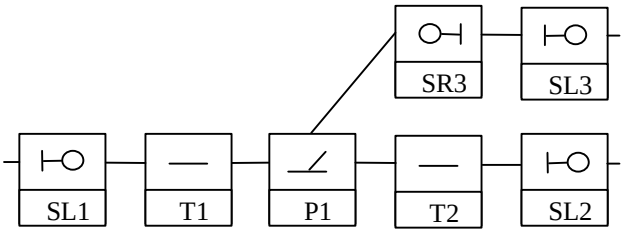
Our condensed example focuses on the configuration of software as a distinguishing feature of the domain.

A railway operating system, i.e. the real-time software of a railway interlocking station, comprises components for logic and control, for controllers of the field elements, for communication with neighbor stations and for the station operator's user interface.

The configuration of a specific station uses the information about the field elements (names, connections, speed limits induced by the environment like slopes, curves, etc.) and about the interfaces to neighbor stations and station operators. From there all the data (parameters) needed by the railway operating system have to be derived and partly adjusted by the configuration engineer.

2.3 Introductory example from the railway software domain

In order to support safe train operation, a minimal railway operating system needs the elements track, point (i.e. pair of points, or switch), signal, occupancy indicator and route. Tracks, points and signals are connected and form a topology, e.g.:



T1 and T2 are tracks. P1 is a point whose tip is directed to the left-hand side. SR3 is a signal whose foot is directed to the right-hand side and whose head is directed to the left-hand side. SL1, SL2 and SL3 are directed to the opposite direction.

Trains move along the topology. They can go in any direction, but they cannot go from one branch of a point to the other (only between tip and one branch). A signal is only visible to a train arriving from the foot side, e.g. SR3 is visible for trains arriving from the right-hand side. A signal shows the speed as an integer number. While moving the trains occupy tracks and points. Occupancy indicators detect whether a track or point is occupied or not.

A train is allowed to start only if the signal in front of it signals a speed > 0. Then it moves up to the next (visible) signal. A route is the possible path of a train from a start signal to a target signal, e.g. SL1-SL2 or SL1-SL3. There are no other routes in the picture: SL2 and SL3 are not connected via a valid path. For all other signal pairs, either the start or the target signal is not visible to the train.

In order to make a train start, the railway station operator sets up a route for it. Then the railway operating system switches the start signal to a speed less or equal to the speed restrictions of all tracks and branches on the route, only if no track or point in the route is occupied and some other conditions beyond this example hold.

3 Elements and constraints in a configuration

3.1 Types, ports and attributes

Trains, railway station operators and railway operating systems are handled at run-time and therefore are not included in the configuration. For the definition of the knowledge base we use the component-port terminology. We model the software in the same terminology as the hardware. A configuration yields parameters to be fed into the software for operation and user interfaces. The configuration does not generate procedure calls or source code for the software.

types={track,point,signal,route}.
For the sake of simplicity occupancy indicators are not included in this example. The abbreviation 'nb' stands for 'neighbor' and is used for port connections in the topology.

ports(track)={nb-left,nb-right}.
attributes(track)={name,index,max-speed}.
dom(track,name)=string. *dom*(track,index)=integer.
dom(track,max-speed)={5,10,15,...,300}.

ports(point)={nb-tip,nb-branch-left,nb-branch-right}.
attributes(point)={name,index,max-speed-branch-left,max-speed-branch-right,preferred-branch}.
dom(point,name)=string. *dom*(point,index)=integer.
dom(point,max-speed-branch-left)={5,10,15,...,300}.
dom(point,max-speed-branch-right)={5,10,15,...,300}.
dom(point,preferred-branch)={left,right}.

ports(signal)={nb-foot,nb-head}.
attributes(signal)={name,index}.
dom(signal,name)=string. *dom*(signal,index)=integer.

ports(route)={start-signal,target-signal,elem-1,elem-2,...}.
The elem-i are virtual connectors between a route and its elements in the path from start-signal to target-signal. For sake of readability the corresponding ports are not listed in signal, track and point.
attributes(route)={name,index,max-speed,is-preferred-route}.
dom(route,name)=string. *dom*(route,index)=integer.
dom(route,max-speed)={5,10,15,...,300}.
dom(route,is-preferred-route)={true,false}.

3.2 Topology and traversal constraints

Many constraints are defined on the topology. Simple ones restrict direct connections, e.g.:

- Only tracks, points and signals are allowed to be connected to one another.

Complex constraints represent dependencies of several distant elements, e.g.:

- For every route the elem-i-ports represent a traversal sequence, i.e. a sequence of neighbors starting at the head of the start-signal and ending at the foot of the target-signal. Traversal sequences are constructed by continuing 'on the other side' of the element, i.e. the other nb-port for tracks and signals. Coming from an nb-branch-port of a point traversal is done via the nb-tip. Coming from nb-tip there are two possible traversals: to the preferred-branch or to the other branch.
- In a complex topology different routes may exist between two signals. The points in the route where two routes start to diverge, are called decision points. A route is-preferred-route iff all its decision points are traversed from nb-tip to preferred-branch.
- For every route its max-speed is less or equal to the max-speed of each of its elements elem-i.

Typical configuration tasks for this example are:

- Manually add new elements and change connections, then check whether the topology is still valid.
- Given a topology, compute all routes and their ports and attributes (max-speed, is-preferred-route).
- After changes in the topology check and – if necessary – change the existing routes.

- The configuration engineer can do manual changes, e.g. decreasing max-speed, or deleting routes. The engineer shall get support to do the changes, e.g. constraint checks. His modifications shall not be overwritten by subsequent automatic extensions.

3.3 Constraints for sorting

In the railway operating system the data for elements are stored in arrays. Typically empty array elements are only allowed at the end of the array, i.e. indices must be used starting from the minimum value (i.e. 1) without gaps.

In a few cases the railway operating system requires particular sorting criteria. In most other cases simple sorting criteria are useful in order to achieve a repeatable numbering scheme (which is also important for comparing different outputs). For our example we assume alpha-numeric sorting along the names of the elements, e.g. SL1.index=1, SL2.index=2, SL3.index=3, SR3.index=4.

Whenever an element is created, the indices of all elements with an index greater or equal to the index of the new element have to be increased. The inverse action is required for deleting an element. Instead of rearranging them, the indices of all elements in the same array can be recomputed. The possibility of recomputation distinguishes software configuration from hardware configuration. Only the consistency among the indices must be preserved throughout reconfiguration.

If external interfaces are involved, re-sorting cannot be used, as an index once chosen and communicated to the interface partner must remain unchanged. Indices are then ordered along the creation time of the element. When an element is created, its index is set to the next available one. If an element is deleted and gaps are not allowed, the index of the element with the highest index has to be set to the deleted element's index.

3.4 Several instances for a real-world element

The railway operating system consists of several components, mainly the software for logic and control, and the interface to the elements in the field. As a consequence we have several types for a real-world element type and several instances (with different types) for a real-world element instance. We show this for the type point.

Typically we have a field element type, e.g. field-point, even if the railway operating system does not need it as a parameter. It is useful for encapsulating the input data for the configuration process.
ports(field-point)={nb-tip,nb-branch-left,nb-branch-right, logic-element}.
attributes(field-point)={name,max-speed-branch-left,max-speed-branch-right,preferred-branch,number-drives}.
dom(field-point,name)=string.
dom(field-point,max-speed-branch-left)= {5,10,15,...,300}.
dom(field-point,max-speed-branch-right)= {5,10,15,...,300}.
dom(field-point,preferred-branch)={left,right}.
dom(field-point,number-drives)={1,2,3}.

For parameterizing logic and control of a point we need a representation for it. The operating system will then use the logic point to store its logical state at run-time.
ports(logic-point)={field-element,controller-1,controller-2, controller-3}.
attributes(logic-point)={index,preferred-branch,number-controllers}.
dom(logic-point,index)=integer.
dom(logic-point,preferred-branch)={left,right}.
dom(logic-point,number-controllers)={1,2,3}.

For each point drive we need a controller.
ports(controller-for-point)={logic-element}.
attributes(controller-for-point)={index}.
dom(controller-for-point,index)=integer.

There are more types for representing a point on the station operator's user-interface (widget-for-point, menu-for-point, etc.) and for further software components. Having distinct types for representing a point in different software components is useful for separation of concerns.

We need constraints for synchronizing the types, e.g.:
- Existence constraints: A logic-point exists iff a field-point exists.
- Connections: A logic-point is connected to number-controllers controller-for-points.
- Equality or computation of attributes:
  For every logic-point L, L.number-controllers = F.number-drives iff L.field-element = F.

4 Summary and conclusion

Railway interlocks are a configuration domain that provides interesting and relevant tasks. A problem instance focusing on software configuration was presented.

We think that this paper is appropriate for a problem instance to be discussed at the workshop. It is suitable for presenting several important aspects of the application domain. After discussing the problem instance we can provide a longer description if the need for more details emerges.

We successfully implemented a configurator for all subsystems of a railway interlock. Currently we are working on a new configurator based on constraint satisfaction concepts. More will be published when the project's results are ready for presentation.

References

[Fleischanderl *et al.*, 1998] Gerhard Fleischanderl, Gerhard E. Friedrich, Alois Haselböck, Herwig Schreiner, and Markus Stumptner. Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems & their applications*, 13(4):59-68, July/Aug. 1998.

[Mittal and Frayman, 1989] Sanjay Mittal and Felix Frayman. Towards a generic model of configuration tasks. *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pp. 1395-1401, San Mateo, Cal., 1989, Morgan Kaufman Publishers.