Subject: Use railML to model parts of a large network
Posted by Thomas Langkamm on Wed, 08 Feb 2023 09:54:56 GMT
View Forum Message <> Reply to Message

Dear all, I wanted to discuss something that came up in the IS group but also applies to interlocking and visualization (and possibly to other schemes). The question is, how do we model part of a network? Unfortunately "just stick to the elements that you want to model" is not the correct answer. Right now we use a somewhat proprietary solution to this problem, but were wondering if this is something that railML in general should support.

Let me describe the use case first. We import a railML network as basis for our train control system, and I know two real-life situations where we need a part of a network.

(1) We're modelling importing a large network (say 50+ stations), but due to the size of the project we're doing this in stages, splitting a network in several parts. (One part could be the area controlled by one interlocking, for example, or perhaps one line.)

(2) There is construction work with new objects (tracks, switches, signals) and we model the part of the network that has changed.

In both cases we have the issue that we need to link up the newly imported file to the remaining parts in the network that we already have in our database. But we also need objects that span parts of the network like track routes, which may be only partially contained in the file. (Let's use track routes as benchmark for the rest of this post, but we could also talk about regions, lines, operational points and more.)

Consider the following track plan:

Assume that the red markers show the boundary of the area, and we're currently importing the part to the left. (The part to the right would be contained in a different track plan, and let's say we have already imported that one.) So left = track plan A (new), right = track plan B (old). We need to define the track routes starting at S01 and the ones ending in S04, all of which cross the border. Which means our "partial track plan" can't end at the red markers but must contain some extra elements, at the very least the part of the network containing the tracks to S05 and S07 (plus possible tracks required to define overlaps, flank conditions and flank overlaps).

Now we basically have two choices.
(a) Have overlapping track plans and full redundancy. That is, the part I showed here would be contained in both track plans A and B, including the definition of the track routes. Both track plans contain all objects and the full details of all objects in the affected area. (Meaning all the objects including the ones not displayed on the track plan like balises, axle counters, markerboards and whatnot.) There is no separation in "this part is defined in this track plan" and "this part should be defined somewhere else".

(b) Have overlapping track plans, but mark some objects as "border" objects and do not use all details. For example, track plan A would have netElements for the displayed tracks, but would

only contain the signals and tvdSections required to define the track routes. (For example, the platforms would be omitted as they are not relevant.) For the border objects, only very few properties would be set. For example, the signals would have an ID and a linear position, but none of the other 2 dozen attributes. And there is a clear definition which objects are border objects, and conversely we know which objects are defined in this plan. Which allows us to separate the network into disjoint sections, while maintaining only a minimum of "border" objects that are redundant.

Now, plan (a) would already be supported by railML. But there is two very important arguments against it, and the first one is this: Redundancies are bad. We all know that if we keep the same data in two or more places, there is always a risk that we update only one plan and have diverging plans. From a software standpoint, it would be undefined what happens if we have conflicting data because the same area is contained in two plans, but modelled differently. Now it is fairly clear that these border elements should be in an area as simple as possible. (So we'd never cut track plans in a station with 4 tracks, 10 parking tracks and 20 switches. Let's try to find an area with 1 or 2 tracks and no switches to make the cut.) But still, some of our track routes may be long and therefore the redundant part, modelled in both track plans, might be long. (Happens a lot in areas where we have two tracks, one for each direction, but some track routes going against the normal direction.) Which gives a lot of opportunities to make errors.

The second issue with the redundancy approach is: How would you detect if objects were deleted? Say some object O is in plan B and not plan A, and we're importing plan A. Was O removed (say a signal that got dismantled), or is it simply an object that is part of B and not modelled in A? (This sounds obvious if you see the track plan visually with a "left = A, right = B" marker. But it's not obvious at all for an algorithm that tries to decide which objects need to be deleted.

Therefore we currently use plan (b). Which is, unfortunately, not supported by railML. But it's fairly straightforward. All we need is to tag some objects as "border", and our import knows that these border object either exist in our database (in which case we can create the track routes) or they don't (in which case we'll throw a warning that we can't create the track routes because objects are missing). Our import also ignores everything but the ID from border objects, so there is no need to enter a gazillion properties. (We'll still try to make our cuts in "simple" areas where we have a minimum of tracks, track routes, switches and so on.)

What do you think? Would it make sense to include such concepts in railML, to better support the splitting of track plans?

File Attachments

1) borders.pdf, downloaded 101 times