Case Study 1:Railway Safety

Daniel Jackson

October 13, 2002

1.1 Motivation

A policy for controlling the motion of trains in a railway system is analyzed. Gates are placed on track segments to prevent trains for colliding. What criteria should be used to determine when gates should be closed?

This case study was designed for pedagogical purposes, to illustrate some fundamental modelling ideas in as small a model as possible. These are: the expression and analysis of a system with arbitrary topology; a separation of concerns into the requirement (the safety property to be established), the machine (the gates and their policy), and the environment (the tracks and trains); and a very partial description of the behaviour of the environment, so that the safety case should make as few assumptions as possible.

The separation of concerns follows the view of requirements developed more fully in Michael Jackson's theory of problem frames [1, 2].

This case study was inspired by a presentation on modelling San Francisco's BART railway, by Emmanuel Letier and Axel van Lamsweerde at a meeting of IFIP Working Group 2.9 on Requirements Engineering (Flims, Switzerland, February 2000). Their work was concerned with the automatic braking system and was focused on real-time event sequencing aspects. In contrast, this model was constructed to illustrate the use of Alloy to analyze a structural aspect.

[1] Michael Jackson. *Software Requirements and Specifications: A Lexicon of Principles, Practices and Prejudices*. Addison Wesley, 1995.

[2] Michael Jackson. *Problem Frames*.

1.2 Description

We view the railway topology is viewed as a collection of segments connected end-to-end. A segment is directional, and only admits traffic on one direction. There is a single gate at the end of the segment; when down, it is intended to prevent a train from leaving the segment. A piece of track that can be used in both directions is therefore viewed as two distinct segments.

The connectivity of segments is modelled by a relation `next` that maps a segment to its successors – the ones which a train can enter on leaving the segment. A relation `over-`

laps on segments models all the physical scenarios in which it would be unacceptable for trains to occupy a pair of segments: because they cross, meeting at a point, or because they come close enough for trains to collide, and so on. This relation is symmetric, not being interpreted as ordered: it would make no sense for a to overlap with b but for b not to overlap with a. It is not generally transitive.

Each train is assumed to occupy exactly one segment. That is, we don't model the introduction of new trains or retirement of old trains. Most unrealistically, we treat trains as points, assuming that a train crosses instantaneously from one segment to another, and never occupies more than one segment, or splits into several smaller trains.

Train motion is modelled by an operation in which some subset of trains on the railway move to successor segments. It is highly non-deterministic, because it does not constrain which trains move, nor for each train that moves, which successor it chooses. Since the safety of the system should rely as little as possible on assumptions about the environment, this non-determinism is desirable. Motion within a segment is not modelled; we pay no attention to timing concerns.

The basic gate policy is that (1) all predecessors of an occupied segment have closed gates (so that a train doesn't enter a segment and collide with a train already there); and (2) for any pair of segments that have the same or overlapping segments, there is at most one open gate (so that two trains occupying those segments don't move together and enter overlapping segments simultaneously).

Criterion (1) addresses a problem that doesn't arise in automobile traffic: that since trains have very long stopping distances, a mechanism is needed to prevent one train from approaching another too closely behind. It is unnecessarily stringent, since it does not permit a train to move to any successor of a segment when one successor is occupied. To weaken it, however, would require gates on *entry* to a segment, since the gate on exit from a segment cannot control which successor segment a train moves to.

Criterion (2) is a generalization of the traffic light rule that you mustn't show green in both directions at once.

1.3 Model Commentary

1.3: In an earlier version of the model, segments were connected by shared points. Unless the points have significance in their own right, a more abstract model that connects segments directly is more appropriate. Two relations characterize the topology: next, which maps each segment to (zero or more) successors, and overlaps, which maps each segment to the (zero or more) segments that it overlaps with. These relations are minimally constrained, since our aim is to account for as many topologies as possible. In developing a model of this sort, a good strategy is to start with no constraints, and

discover those that are necessary when safety violations are found by the analysis that correspond to unrealistic scenarios.

1.5: The overlaps relation is constrained to be symmetric. This is purely a modelling convenience. By viewing a segment as overlapping with itself, we obtain more uniform constraints. It is not necessary, for example, to say that two trains may not occupy the same segment, this case being subsumed by the constraint that they may not occupy overlapping segments.

1.6: The symmetry of the overlaps relation is, in contrast, justified by the notion being modelled, since overlapping is a phenomenon of a set of two segments, not an ordered pair. In a sense this constraint undoes the unwanted directionality of the binary relation. An alternative would be to model overlappings as atoms in their own right, declared with a signature and a relation mapping overlappings to sets of segments.

1.8: Trains are introduced as basic atoms without fields of their own. The property of being on a segment is time-dependent, and is therefore modelled as a field of the State signature.

1.9: The GateState signature models the set of global configurations of the gates. A gate state is associated, by the closed field, with a set of segments that are closed. This treatment is nice and abstract; it doesn't require us to allocate gates to segments first, and then determine which are closed (although in an implementation this is of course what you would do). Note that the state of a segment is modelled by its membership in the set of segments whose gates are closed. A novice might want to declare a signature corresponding to the different statuses of a gate (open and closed), and then define a mapping from gates to their statuses. This is clumsy; it introduces an unnecessary notion (gate status), and it tends to make constraints less succinct (with set comprehensions being introduced to provide the set of gates with a given status that is available immediately in the simpler treatment).

1.10: The TrainState signature models the set of configurations of trains: how they are placed on segments. Note that the underlying configuration of segments is not modelled explicitly by a signature, but by the nexts and overlaps relations of Seg. Since these are fixed, the model does not account for dynamic changes in toplogy. The on field of TrainState captures the train configurations, by mapping each train to the segment it sits on. The multiplicity symbol (the exclamation mark) on the right-hand side in the expression for on carries a lot of weight: it expresses the domain assumption that trains occupy at most one segment, and that they are never 'off track'. The occupied field is *defined* in terms of on. It is redundant, and included only for convenience. For a trainstate x, x.occupied gives the set of segments that have trains on them.

1.12: The safety criterion is recorded as a function, and not a fact, since we want to check that it follows from the policy, so it would make no sense to declare it to be true

from the outset. It is a property of a single train state: it is appropriate that its *span* (see 2) is as narrow as possible, and does not include the gate state, for example, which is part of the mechanism designed to achieve safety but not relevant to the essential safety concern. The constraint says that, for any pair of distinct trains t and t′, the segment that t occupies is not a member of the set of segments that are overlapped by the segment t′ occupies. Note that in the expression x.on[t′].overlaps, the [] operator binds more loosely than the dot operator, so it is parsed as ((x.on)[t′]).overlaps: first take the on relation of the train state x; find the segment associated with t′; then find the segments this segment overlaps with.

1.16: The transition relation describing what changes in state may actually occur is split into two functions. The first, MoveOK, is a constraint that describes in which gate and train states it is legal for a set of trains to move. The second, TrainsMove describes the effect on the train state of some set of trains moving. It is separated from MoveOK for methodological reasons. MoveOK records a social agreement: a driver may choose to violate it. TrainsMove, on the other hand, is a physical constraint: a driver may not choose to cross from a segment into a segment not connected to it. MoveOK says that a set of trains movers may move when the gates are in a state g and the trains are in a state x, so long none of the trains in movers are on segments whose gates are closed.

1.20: The function TrainsMove describes the effect of a set of trains movers moving from their segments to arbitrary successors. Note that the effect is described only in terms of the state of the trains, producing a change from x to x′, and is not constrained by the state of the gates. The constraint has two parts. The first (1.21) says that every train that moves ends up in the state x′ on a segment that is a successor of the segment it was in the previous state x. The second (1.22) says that the trains that don't move stay on the same segments. This function illustrates two key features of declarative specification: the ability to write very underdetermined descriptions (in this case, allowing any set of trains to move, and for each to select the successor segment to move to) without special constructs for non-determinism, and, arising from this, the need for frame conditions (in this case constraining the remaining trains to stay put). Making this description of train motion as loose as possible is crucial, since we want to check that the safety mechanism will work with as few assumptions about the environment as possible.

1.25: The function GatePolicy describes the safety mechanism, enforced as a policy on a gate state and a train state. It comprises two constraints. The first (1.26) is concerned with trains and gates: it says that the segments that are predecessors of those segments that are occupied by trains should have closed gates. In other words, a gate should be down when there is a train ahead. This is an unnecessarily stringent policy, since it does not permit a train to move to any successor of a segment when one successor is occupied. The second (1.27) is concerned with gates alone: it says that between any pair of segments that have an overlapping successor, at most one gate can not be closed. Note that, like the TrainsMove function, this constraint is also quite weak. An implementation

of a safety mechanism would be deterministic; it would presumably determine which gate to close according to many factors not modelled here (such as priority of trains, time of arrival, etc.). This weaker constraint is an attempt to capture properties that a whole class of mechanisms would share.

1.30: PolicyWorks asserts the safety claim. It says that if a move is permitted according to the rules on MoveOK, if the trains move according to the physical constraints of TrainsMove, if the safety mechanism described by GatePolicy is enforced, then a transition from a safe state will result in a state that is also safe. In other words, safety is preserved; the mechanism works.

1.39: The command CheckPolicy instructs the analyzer to evaluate the claim PolicyWorks for all situations involving at most 5 trains, 5 segments, 2 train states, and one gate state. No counterexample is found; the analysis is completed in about 5 seconds (Version 1.1, October 2002). We could increase the scope and consider more trains and segments, but it seems unlikely that it would reveal a problem. Note that considering more states is pointless, since any counterexample requires 2 train states and one gate state, those states named by bound variables.

```
1.1   module pedagogical/railway
1.2
1.3   sig Seg {next, overlaps: set Seg}
1.4
1.5   fact {all s: Seg | s in s.overlaps}
1.6   fact {all s, s': Seg | s in s'.overlaps = s' in s.overlaps}
1.7
1.8   sig Train {}
1.9   sig GateState {closed: set Seg}
1.10  sig TrainState {on: Train -! Seg, occupied: set Seg}{occupied = on[Train]}
1.11
1.12  fun Safe (x: TrainState) {
1.13     all disj t, t': Train | x.on[t] !in x.on[t'].overlaps
1.14     }
1.15
1.16  fun MoveOK (g: GateState, x: TrainState, movers: set Train) {
1.17     no x.on[movers] & g.closed
1.18     }
1.19
1.20  fun TrainsMove (x, x': TrainState, movers: set Train) {
1.21     all t: movers | x'.on[t] in x.on[t].next
1.22     all t: Train - movers | x'.on[t] = x.on[t]
1.23     }
1.24
```

```
1.25    fun GatePolicy (g: GateState, x: TrainState) {
1.26       x.occupied.overlaps.~next in g.closed
1.27       all s,s': Seg | some (s.next & s'.next.overlaps) = sole (s + s' - g.closed)
1.28       }
1.29
1.30    assert PolicyWorks {
1.31       all x, x': TrainState, g: GateState, movers: set Train |
1.32          {MoveOK (g, x, movers)
1.33          TrainsMove (x, x', movers)
1.34          GatePolicy (g, x)
1.35          Safe (x) }
1.36          = Safe (x')
1.37       }
1.38
1.39    CheckPolicy: check PolicyWorks for 5 but 2 TrainState, 1 GateState
```

1.4 Variations

1.4.1 A Faulty Gate Policy

As an example of a simple bug, suppose the safety mechanism were to omit one of the cases in which overlap is considered, so that the check tests only for a potential collision with a train in a succeeding segment, and not all those that overlap succeeding segments:

```
1.40    fun GatePolicy (g: GateState, x: TrainState) {
1.41       x.occupied.~next in g.closed
1.42       all s,s': Seg | some (s.next & s'.next.overlaps) = sole (s + s' - g.closed)
1.43       }
```

The conjecture PolicyWorks is now false. It has a counterexample in a smaller scope than the one specified in the command CheckPolicy. Executing the command

```
1.44    CheckPolicy': check PolicyWorks for 2 Train, 3 Seg, 2 TrainState, 1 GateState
```

produces the counterexample shown in the figure below, in which Train_0 on Seg_1 moves to Seg_2 which overlaps with Seg_0, which is occupied by Train_1.

1.4.2 Exploiting Relational Operators

In several places, we could make the model a bit more elegant by using relational operators rather than set-based operators.

In the GatePolicy function (1.25), for example, the second condition (1.27) could be written instead as

```
1.45    all s,s': Seg | some (s.next->s'.next) & overlaps = sole (s + s' - g.closed)
```

Here, the expression s.next->s'.next forms the set of all pairs of successors of s and s'; the condition some (s.next->s'.next) & overlaps tests whether any such pair also belongs to the overlap relation.

The properties of the overlap relation are generic properties of many relations, so it makes sense to factor them out as polymorphic functions. The following code illustrates this, as well as a way of writing the properties with relational operators rather than quantifiers.

```
1.46    fact {Reflexive(overlaps) && Symmetric(overlaps)}
1.47
1.48    fun Reflexive [t] (r: t - t) {iden[t] in r}
1.49    fun Symmetric [t] (r: t - t) {~r in r}
```