Train Composition and Decomposition: Domain and Requirements

Panagiotis Karras and Dines Bjørner
Computer Science and Engineering,
Informatics and Mathematical Modelling, Technical University of Denmark
Building 322, Richard Petersens Plads, DK–2800 Kgs. Lyngby, Denmark

{pan|db}@imm.dtu.dk

30 September, 2002

Abstract

The problem is to be tackled is as follows: There is a railway net. Trains travel from station to station. When leaving an intermediate station on a train journey, a train may have its direction reversed. Trains are composed from assemblies. An assembly is a sequence of carriages. At stations trains may have assemblies added (composed) to, or removed (decomposed) from the train. Station tracks may restrict train additions and removals to occur only at either the front, or at the back of a train. Given requirements for trains to provide suitable load (for example passenger) capacity along a journey with varying such demands, the problem is now to plan that trains, during their journeys, have suitable assemblies added to or removed from the train.

We present a standard informal narrative and a formal model of train composition and decomposition, of their planning and effectuation.

We relate this model to the rough sketch description provided by Dr. Leo Kroon of NLS Reisigers.

Contents

1 Brief Outline of the Formal Model

T:1, F:1

The end result of our model at this stage is schedule-generating function. The schedules this function generates are such, so that the desired rolling stock changes can be derived from the schedule itself. This function is producing a set of all allowed schedules which satisfy a given passenger statistics:

gen_Sched: Stat1 × RulReg × RS → Sched-**set**
gen_Sched(st,rr,rs) ≡ {sch | sch: Sched • fitting(reform_stat(st),sch)}

F:2

The satisfiability of a given statistics by a schedule is defined by a fitting function:

fitting: Stat1 × Sched → **Bool**

The statistics as entered into the fitting function are in a well formed format. In this format, only the number of passengers travelling between consecutive stations at given times is given. A general non-well-formed statistics may be transformed into a well-formed one through a function made for that purpose:

reform_stat: Stat1′ → Stat1

F:3

Crucial for the definition of this reformation function is the ability to sum over all possible 'trips' within a statistics of which a 'trip' between two given consecutive stations is a part. For that purpose we use a recursive summing function:

sum_of_nums: Stat1′ × Trip-**set** → Stat_Number

F:4

As well as a function that returns the desired set of trips within a statistics for a given trip, such that the given trip is part of every trip in that set:

super_trip_set: Stat1′ × Trip → Trip-**set**

F:5

Necessary for the definition of such a function is, among others, a function generating the set of stations that form the shortest path between two given stations:

connecting_set: Sta × Sta → Sta-**set**

F:6

The connecting set function is based on, among others, a path existence function, which, for two given stations, returns a true value when there exists a path between them within a network:

exists_path: Sta × Sta → **Bool**

The path existence function is defined recursively, while its basis is an observer function, which gives the set of neighbouring stations for a given station:

obs_SSS: Sta → Sta-**set**

It is axiomatically required that a station belongs to the unique same network as its neighbours. After this short excursion we may proceed to the presentation of the whole model. F:7

2 Introduction of the Formal Model

T:2, F:8

2.1 Railway Topology

We take as base concept for the railway net topology that of nets. From a railway net we can observe lines and stations. There are at least two stations and one line in a net. From a line we can observe the exactly two distinct stations it connects. From a station we can observe the set of one or more tracks (on which trains may halt). From a station we can observe the set of those lines from which the station can be reached. From a station we can observe the
F:9 set of lines that can be reached from that station. From a track of a station we can observe the lines from which the track can be reached. From a track of a station we can observe those lines that can be reached from that track. Given a track and a pair of incoming, respectively outgoing lines, we can observe whether a train, in order to pass from the incoming to the outgoing line will be reverse or not. From a route we can observe the ordered list of stations that it contains, including at least two stations. Given a route and a station, we can observe whether a train following this route will undergo a reversal at this station or not.

type

Net, Lin, Sta, Tra,
Route = Sta×Tra×(Lin×Sta×Tra)*

value

obs_Stas: Net → Sta-**set**,
obs_Lins: Net → Lin-**set**,

obs_SS: Lin → Sta-**set**,
obs_LS: Sta → Lin-**set**,
obs_SSS: Sta → Sta-**set**,

obs_Tras: Sta → Tra-**set**,

obs_in_Lins: Sta×Tra → Lin-**set**,
obs_out_Lins: Sta×Tra → Lin-**set**,

is_Line_Reversal: Lin × Tra × Lin → **Bool**,

obs_RStas: Route → Sta*,

is_RReversal: Route × Sta → **Bool**,

exists_path: Sta × Sta → **Bool**
exists_path(s1,s2) ≡
(s2 ∈ obs_SSS(s1)
  ∨
  (∃ s3: Sta •
    s3 ∈ obs_SSS(s1) ∧

```
        exists_path(s3,s2) )
  ),

  next_station: Route × Sta → Sta
  next_station((FS,FT,LSTlist),s) as s′
   post
     ( ∃ idx,idx′: Nat, l,l′: Lin, t,t′: Tra •
         idx′ = idx + 1 ∧
         (l,s,t) = LSTlist(idx) ∧
         (l′,s′,t′) = LSTlist(idx′)
     )
   pre
     (s = FS) ∨
     ( ∃ idx: Nat, l: Lin, t: Tra •
         (l,s,t) = LSTlist(idx)
     )
```

axiom

```
  ∀ n : Net • card obs_Stas(n) ≥ 2 ∧
                          card obs_Lins(n) ≥ 1,

  ∀ s: Sta • ∃! n: Net • s ∈ obs_Stas(n),
  ∀ l: Lin • ∃! n: Net • l ∈ obs_Lins(n),
  ∀ t: Tra • ∃! s: Sta • t ∈ obs_Tras(s),

  ∀ s: Sta, l: Lin •
   ( l ∈ obs_LS(s) ⇒
     ( ∃! n: Net •
      ( l ∈ obs_Lins(n) ∧ s ∈ obs_Stas(n) ))),

  ∀ ℓ: Lin •
   obs_SS(ℓ) =
     { s | s: Sta • ℓ∈ obs_LS(s) }
   ∧
   card obs_SS(ℓ) = 2,

  ∀ s: Sta •
   obs_SSS(s) =
     { s′ | s′: Sta •
         ∃ ℓ: Lin •
           {s,s′} ⊆ obs_SS(ℓ) },

  ∀ s : Sta • obs_Tras(s) ≠ {},
```

∀ s : Sta •
 ∃ t: Tra •
  obs_in_Lins(s,t) ≠ {} ∧
  (∀ l : Lin •
   l ∈ obs_in_Lins(s,t) ⇒
   s ∈ obs_SS(l)),

∀ s : Sta •
 ∃ t: Tra •
  obs_out_Lins(s,t) ≠{} ∧
  (∀ l : Lin •
   l ∈ obs_out_Lins(s,t) ⇒
   s ∈ obs_SS(l)),

∀ t : Tra, s : Sta, ℓ: Lin •
 ℓ∈ obs_in_Lins(s,t) ⇒
  t ∈ obs_Tras(s),

∀ t : Tra, s : Sta, ℓ: Lin •
 ℓ∈ obs_out_Lins(s,t) ⇒
  t ∈ obs_Tras(s),

∀ r : Route • **len** obs_RStas(r) > 1

F:10

2.2 Trains, Assemblies, Carriages, and Rolling Stock

F:11

From a train we can observe the ordered list of assemblies that it contains, including at least one assembly. From an assembly we can observe the ordered list of carriages that it contains, including at least one carriage. Given two trains, we can observe whether they constitute a reversal of each other, in case they are comparable. Given a route, we can observe the next station within it. We define equivalence and identity relationships between trains.

type

Train, Assem, Carrg

value

obs_Assems: Train → Assem*,
obs_Carrgs: Assem → Carrg*,

is_TReversal: Train × Train $\xrightarrow{\sim}$ **Bool**,
next_state_in_route: Route × Sta × Train → Train × Sta,

equivalent_trains: Train × Train → **Bool**

equivalent_trains(t1,t2) ≡
(∀ c: Carrg •
  (∃ asm: Assem •
      asm ∈ obs_Assems(t1) ∧
      c ∈ obs_Carrgs(asm)
)
  ≡
  (∃ asm′: Assem •
      asm′ ∈ obs_Assems(t2) ∧
      c ∈ obs_Carrgs(asm′)
)
),

identical_trains: Train × Train → **Bool**
identical_trains(t1,t2) ≡
(∀ c: Carrg, idx1,idx2: **Nat** •
  (∃ asm: Assem •
      asm = obs_Assems(t1)(idx1) ∧
      c = obs_Carrgs(asm)(idx2)
)
  ≡
  (∃ asm′: Assem •
      asm′ = obs_Assems(t2)(idx1) ∧
      c = obs_Carrgs(asm′)(idx2)
)
)

axiom

∀ t : Train • **len** obs_Assems(t) > 0,
∀ a : Assem • **len** obs_Carrgs(a) > 0,

∀ r: Route, s: Sta, t: Train •
 let
   (t′,s′) = next_state_in_route(r,s,t)
 in
   is_RReversal(r,s,t) ⇒ is_TReversal(t,t′)
 end,

∀ r: Route, s: Sta, t: Train •
 let
   (t′,s′) = next_state_in_route(r,s,t)
 in
   s′ = next_station(r,s)
 end

F:12

2.3 Journeys, Time Tables and Schedules

F:13

From a schedule we can observe get the set of journeys described in it. From a schedule and a train number we can observe the journey the train with this number makes according to this schedule. From a journey we can observe the list of stations visited in it, including at least two stations. Given a schedule, a train number and a station, we can observe the set of pairs of arrival and departure times for the train with this number respectively to and from that station according to this time table. Given a schedule, a train number, a station and a time, we can observe the characteristics of the train that appears in that station bearing this train number at that time by (according to) this schedule. We formulate well-formedness conditions for journeys and schedules. An assembly map is a pairing of assembly types to their numbers within a train configuration. For mathematical reasons that will be apparent in the following section, we define a well-formed assembly map so that its domain includes all existing assembly types, with possible zero values as numbers of assemblies. We define a function that gives the expected travelling time between two stations.

type

TrainNum, TrainChars, Platform, AssemType, OClock,
TimeDur == null|_,
TravelTime = TimeDur,
StopTime = TimeDur,
Interval = TimeDur,
FirstTime = OClock,
LastTime = OClock,
Num_of_Assems = **Nat**,
Num_of_Passengers = **Nat**,

SV = Sta×OClock×OClock×Platform×TrainChars,
Journ′ = SV*,
Journ = {|j: Journ′ • wf_journ(j) |},

TrSer = TrainNum*,

Sched = TrainNum $\xrightarrow[m]{}$ Journ,

AssemMap′ = AssemType $\xrightarrow{m}$ Num_of_Assems,
AssemMap = {|am: AssemMap′ • wf_assem_map(am) |}

value

obs_Assemblies: TrainChars → AssemMap,
obs_Num_of_Passengers: AssemType → Num_of_Passengers,

```
journs_in_sched: Sched → Journ-set
journs_in_sched(sch) ≡ rng sch,

journ_of_num: Sched × TrainNum → Journ
journ_of_num(sch,tn) ≡ sch(tn)
pre
 tn ∈ dom sch,

journ_stas: Journ → Sta-set
journ_stas(j) as station_set
post
 ( ∀ s: Sta •
      ( ∃ idx: Nat, dt,at: OClock,
                  p: Platform, tc: TrainChars •
          j(idx) = (s,dt,at,p,tc)
      )
    ≡
      s ∈ station_set
 ),

times: Sched × TrainNum × Sta → (OClock × OClock)-set
times(sch,tn,s) as times_set
post
 ( ∀ at,dt: OClock •
      (at,dt) ∈ times_set
    ≡
      ( ∃ idx: Nat, p: Platform, tc: TrainChars •
          (s,at,dt,p,tc) = sch(tn)(idx)
      )
 )
pre
 tn ∈ dom sch ∧
 s ∈ journ_stas(sch(tn)),

trainchars: Sched × TrainNum × Sta × OClock → TrainChars
trainchars(sch,tn,s,t) as tc
post
 ( ∃ idx: Nat, dt: OClock, p: Platform •
      (s,t,dt,p,tc) = sch(tn)(idx)
 )
pre
```

tn ∈ **dom** sch ∧
s ∈ obs_JStas(sch(tn)),

obs_travelling_time: Sta × Sta → TravelTime,

>: OClock × OClock → **Bool**,
>: TimeDur × TimeDur → **Bool**,

≥: OClock × OClock → **Bool**,
≥: TimeDur × TimeDur → **Bool**,

∗: **Nat** × Interval → Interval,
+: OClock × TimeDur → OClock,
+: TimeDur × TimeDur → TimeDur,
−: OClock × OClock → TimeDur,

wf_journ: Journ → **Bool**
wf_journ(j) ≡
(∀ idx1,idx2: **Nat** •
  idx1 < **len** j ∧ idx2 < **len** j ⇒
  let
    (s1,at1,dt1,p1,tc1) = j(idx1),
    (s2,at2,dt2,p2,tc2) = j(idx2)
  in
    s2 ∈ obs_SSS(s1) ∧
    dt2 > at2 ∧ at2 > dt1 ∧ dt1 > at1 ∧
    at2 − dt1 ≥ obs_travelling_time(s1,s2)
  end
),

wf_schedule: Sched → **Bool**
wf_schedule(sch) ≡
(∀ j: Journ •
  j ∈ **rng** sch ⇒ wf_journ(j)
),

wf_assem_map: AssemMap → **Bool**
wf_assem_map(am) ≡
(∀ at: AssemType •
  at ∈ **dom** am
)

axiom

∀ sch : Sched • journs_in_sched(sch) ≠ {},

∀ j : Journ • **card** journ_stas(j) > 1,

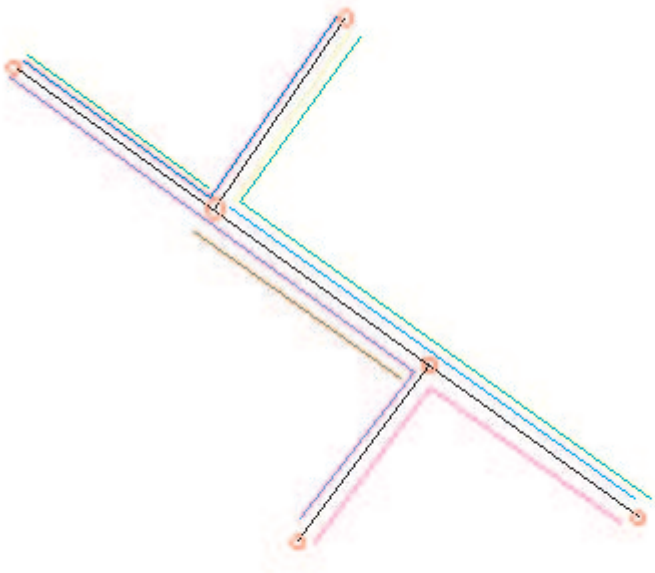
∀ s: Sta • obs_travelling_time(s,s) = null

F:14

2.4 Passenger Statistics

F:15

A Statistics is a mapping from pairs of time values to maps of couples of Stations mapped to the Number of passangers commuting between those two stations in the time interval between those two time values. We overload the addition and subtraction operators so that these may be used for the composition and decomposition of trains, respectively, expressed as adding and subtracting of assemply maps. Given a journey and a station within this journey, we may derive the rolling stock to be added and to be taken out of the train making that journey in that station. A real world statistics given in a general form, in terms of numbers of commuters between pairs of stations and departure and arrival times, is transformed into a specific statistics as needed in our model, in terms of number of commuters between consecutive stations only.



type

Stat_Number = **Nat**,
Trip = (OClock × OClock) × (Sta × Sta),

Stat1′ = (OClock×OClock) $\overrightarrow{m}$ ((Sta×Sta) $\overrightarrow{m}$ Number),
Stat1 = {|s: Stat1 • consecutive(s) |}

value

+: AssemMap × AssemMap → AssemMap
am1 + am2 **as** am3
post
(∀ at: AssemType •
am3(at) = am1(at) + am2(at)
),

−: AssemMap × AssemMap → AssemMap
am1 − am2 **as** am3
post
(∀ at: AssemType •
am3(at) = am1(at) − am2(at)
)
pre
(∀ at: AssemType •
am1(at) ≥ am2(at)
),

orthogonal: AssemMap × AssemMap → **Bool**
orthogonal(am1,am2) ≡
(∀ at: AssemType •
am1(at)∗am2(at) = 0
),

train_change: Journ × Sta → TrainChars × TrainChars
train_change(j,s) **as** (tc0,tc1)
post
(∃ idx: **Nat**, s0: Sta, at,at0,dt,dt0: OClock,
p,p0: Platform •
(s,at,dt,p,tc1) = j(idx) ∧
(s0,at0,dt0,p0,tc0) = j(idx − 1)
)
pre
s ∈ journ_stas(j),

implement_change: TrainChars × TrainChars → AssemMap × AssemMap
implement_change(tc1,tc2) **as** (out_assem_map,in_assem_map)

```
 post
   let
    ante_assem_map = obs_Assemblies(tc1),
    post_assem_map = obs_Assemblies(tc2)
   in
    post_assem_map = ante_assem_map + in_assem_map − out_assem_map
    ∧
    orthogonal(in_assem_map,out_assem_map)
   end,


comp_decomp: Journ × Sta → AssemMap × AssemMap
comp_decomp(j,s) ≡
 implement_change(train_change(j,s)),


consecutive: Stat1′ → Bool
consecutive(st) ≡
( ∀ dst,ast: Sta, num: Stat_Number, dt,at: OClock •
    (dt,at) ∈ dom st ∧
    (dst,ast) ∈ dom st(dt,at) ∧
    num = st(dt,at)(dst,ast)
    ⇒
    ast ∈ obs_SSS(dst) ∧ at − dt ≥ obs_travelling_time(dst,ast)
),


connecting_set: Sta × Sta → Sta-set
connecting_set(s1,s2) as connecting_set
post
 ( ∀ s: Sta •
      exists_path(s1,s) ∧ exists_path(s,s2) ∧
      obs_travelling_time(s1,s) + obs_travelling_time(s,s2) =
      obs_travelling_time(s1,s2)
      ≡
      s ∈ connecting_set
 )
pre
 exists_path(s1,s2),


super_trip_set: Stat1′ × Trip → Trip-set
super_trip_set(st,((dt,at),(dst,ast))) as super_trip_set
post
 ( ∀ dst′,ast′: Sta, dt′,at′: OClock •
      (dt′,at′) ∈ dom st ∧
      (dst′,ast′) ∈ dom st(dt′,at′) ∧
```

```
      {dst,ast} ⊆ connecting_set(dst′,ast′) ∧
      dt ≥ dt′ + obs_travelling_time(dst,dst′) ∧
      at′ ≥ at + obs_travelling_time(ast′,ast)
      ≡
      ((dt′,at′),(dst′,ast′)) ∈ super_trip_set
 ),


sum_of_nums: Stat1′ × Trip-set → Stat_Number
sum_of_nums(st,super_trip_set) ≡
 case card super_trip_set of
   0 → 0,
   _ →
    let
      dst: Sta, ast: Sta, dt: OClock, at: OClock •
      ((dt,at),(dst,ast)) ∈ super_trip_set
    in
      st(dt,at)(dst,ast) +
      sum_of_nums(st,super_trip_set\{((dt,at),(dst,ast))})
    end
 end,


reform_stat: Stat1′ → Stat1
reform_stat(st′) as st
post
 ( ∀ dst,ast: Sta, dt,at: OClock •
      (dt,at) ∈ dom st ∧
      (dst,ast) ∈ dom st(dt,at)
      ⇒
      st(dt,at)(dst,ast) =
      sum_of_nums(st′,super_trip_set(st′,((dt,at),(dst,ast))))
 )
pre
( ∀ dst,ast: Sta, dt,at: OClock •
    (dt,at) ∈ dom st′ ∧
    (dst,ast) ∈ dom st′(dt,at)
    ⇒
    at − dt ≥ obs_travelling_time(dst,ast)
)
```

F:16

3 Planning

T:3, F:17

3.1 Planning Function

Given a statistics and a set of rules and regulations we can generate a set of scedules for the given statistics under the given rules and regulations. The generated schedules satisfy the given statistics. A recursive function computes the sum of passenger load that a given configuration of train characteristics may take.

type

RulReg

value

```
passengers_sum: AssemMap → Nat
passengers_sum(assem_map) ≡
 case card dom assem_map of
   0 → 0,
   _ →
    let
      assem_type: AssemType •
      assem_type ∈ dom assem_map
    in
      assem_map(assem_type)∗obs_Num_of_Passengers(assem_type) +
      passengers_sum(assem_map\{assem_type})
    end
 end,


fitting: Stat1 × Sched → Bool
fitting(st,sch) ≡
( ∀ dt,at: OClock, dst,ast: Sta, num: Stat_Number •
    (dt,at) ∈ dom st ∧
    (dst,ast) ∈ dom st(dt,at) ∧
    num = st(dt,at)(dst,ast)
    ⇒
    ( ∃ j: Journ •
       j ∈ rng sch ∧
       (∃ idx1,idx2: Nat, at1,dt2: OClock,
                  p1,p2: Platform, tc1,tc2: TrainChars •
          idx1 ≤ len j ∧ idx2 ≤ len j ∧
          (dst,at1,dt,p1,tc1) = j(idx1) ∧
          (ast,at,dt2,p2,tc2) = j(idx2) ∧
          passengers_sum(obs_Assemblies(tc1)) ≥ num
```

)
)
),

gen_Sched: Stat1 × RulReg → Sched-**set**
gen_Sched(st,rr) ≡ {sch | sch: Sched • fitting(reform_stat(st),sch)}

F:18