Towards safe capacity in the railway domain – An experiment in Timed-CSP*

Yoshinao Isobe[×], Hoang Nga Nguyen[+], Markus Roggenbach[+]
[×]AIST, Japan [+]Swansea University, UK

December 10, 2012

Abstract

Railways need to be safe and, at the same time, shall offer high capacity. While the notion of safety is well understood in the railway domain, the meaning of capacity is clear only on an intuitive level. We show here, using the process algebra Timed Csp, how to treat capacity in a rigorous way. Our modelling approach builds on an established modelling technique for safety alone, provides an integrated view on safety as well as capacity, and offers proof support in terms of (untimed) model checking.

1 Introduction

Overcoming the constraints on railway capacity caused by nodes (stations and junctions) on the rail network is one of the most pressing challenges to the rail industry. In 2007, the UK governmental White Paper "Delivering a Sustainable Railway" [oT07] states: *"Rails biggest contribution to tackling global warming comes from increasing its capacity."* High capacity, however, is but one design aim within the railway domain. Railways are safety-critical systems. Their malfunction could lead to death or serious injury to people, loss or severe damage to equipment, or to environmental harm. This work, in cooperation with our industrial partner Invensys Rail, aims to develop an integrated view of rail networks, within which capacity can be investigated without compromising safety.

The process algebra Csp [Hoa85, Ros98, Ros10] has successfully been applied to modelling, analysing and verifying railways for safety aspects, see e.g. [SWD97, Win02]. Solely concerned with safety, however, these approaches have ignored the aspect of time. And yet the capacity of a rail network node is highly dependent on time: Moving a point or moving a train through a node takes time, sighting and braking distance are functions of time. Thus, rather than using Csp, we apply Timed Csp [Sch00, OW03] in order to achieve an integrated view

*Acknowledging support by the company Invensys and the EPSRC/RSSB research grant *SafeCap*.

on safety and capacity. While, e.g., [SWD97, Win02] model safety within Csp, to the best of our knowledge we are the first to consider railway capacity in Timed Csp or a related formalism.

Of the various capacity notions within the railway domain, we deal here with the so-called theoretical line capacity. "Theoretical capacity" denotes the capacity that in principal can be scheduled (as opposed to the capacity actually used). "Line capacity" stands for a setting, which considers only trains of the same characteristic (e.g. all trains have the same breaking behaviour and the same maximal speed) that all take the same path through a network. It is future work to capture the more complex notion of network capacity (the number of trains that can operate on a rail network in a given time period).

In Csp and Timed Csp, the verification of properties such as safety and capacity requires (1) to formulate the property of interest as a process *Prop* (2) to model the system under consideration as a process *Sys* and (3) to prove a refinement statement $Prop \rightsquigarrow Sys$. For Csp, tools such as the model checker FDR and the interactive theorem prover Csp-Prover [IR, IR05] can prove such refinement statements. For Timed Csp, the Timed Csp simulator [DGR11] can provide first insights into the system, the solver Horae [DHSZ06] can prove refinements, and, finally, for a restricted sub-language of Timed Csp, refinement statements over Timed Csp can equivalently translated into Csp, i.e., the Csp proof support can be re-used for Timed Csp.

Our paper is organized as follows: First, we discuss basic railway concepts in terms of a simple example and motivate the question of capacity. Then, we review railway modelling for safety in Csp following the approach by [Win02]. Next, the language Timed Csp and the idea of timed traces is introduced. In Section 5, we describe how to extend [Win02]'s modelling in order to capture the timing of events on a railway. Given such a timed behaviour, we can ask what capacity it has: in a first step, we define capacity as a function on timed traces; in a second step, we make this notion of capacity visible via a non-interfering observer process, which runs in parallel to the railway system. This observer process enables us to encode our capacity definition as a Timed Csp refinement, see Section 7. Using results from [OW03] and [HMP92], the Timed Csp refinement statements for safety and capacity can be translated into (untimed) Csp. We conclude our paper by applying these results to our original example.

2 A simple railway

Figure 1 shows a simple track plan, namely *the single line*. The single line consists of two routes R_1 and R_2, which are protected by two-aspect signals S_1 and S_2, respectively. Each route is divided further into tracks. R_1 comprises of two tracks AE and AF while R_2 consists of AG and AH. There are two special tracks, namely *Entry* and *Exit*, at both ends of the single line from which trains enter and exit the line, respectively. Trains can "appear" on the *Entry* track, travel along the routes R_1 and R_2 towards the *Exit* track, where they then "vanish". The signals S_1 and S_2 show either "proceed" or "halt". Train drivers

are only allowed to enter a route, say R_1, when the protecting signal, in our example S_1, shows "proceed". Otherwise, the train driver has to stop and to wait for a "proceed".

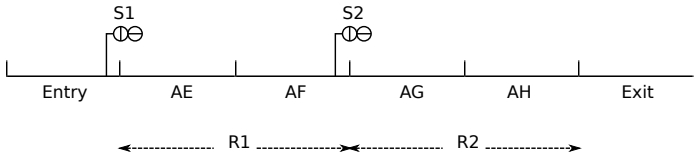

Figure 1: The track plan of *the single line*

Railway signals are controlled by a so-called interlocking. This interlocking shall guarantee safety. In general, train movements are considered safe if there is no collision (two trains on the same track at the same time) and no derailment (no point moves while there is a train moving over or trains are too fast while passing a point). As our single line has no point, it suffices for our paper, to consider collision freedom only.

The interlocking gathers inputs from the physical railway such as train locations with respect to tracks and sends out commands to control signal aspects and point positions. To this end, it implements so-called control tables.

Under current UK regulations, the following control tables are prescribed for the signals S_1 and S_2, which we will consider as Scenario 1:

$CT_1 - S_1$	Clear
Proceed	AE, AF, AG

$CT_1 - S_2$	Clear
Proceed	AG, AH

Signal S_1 only shows proceed when tracks AE, AF and AG all are clear. The track AG is called "overlap". The rationale behind this rule is that – even when the driver sees signal S_1 too late, e.g., just when she is passing it – she will always be able to stop the train before entering track AH.

Scenario 2 makes the assumption that all trains are equipped with an Automatic Train Protection (ATP) system [KR01]. ATP ensures that trains brake when needed. Thanks to ATP, trains are guaranteed to stop at or before signal S_1. Therefore the overlap AG can be removed from the clear part of control table of signal S_1

$CT_2 - S_1$	Clear
Proceed	AE, AF

while the control table of signal S_2 remains the same. Under current UK regulations, Scenario 2 is not allowed.

In the railway domain, capacity is regarded as an elusive concept which is not easy to define and measure. In general, it can be described as below:

> Capacity determines the maximum number of trains that would be able to operate on a given railway infrastructure, during a specific time interval, given the operational conditions [ABI+08].

Coming back to our single line example, the general view in railway industry, shared by our industrial partner Invensys, is the following: Removing overlaps such as track AH from control tables increases capacity. The scientific questions are: Can safety still be guaranteed? How can the expected effect be measured? Based on these results, the political question would be: Is the gained capacity increase enough to start changing regulations?

3 Modelling railways for safety in CSP

The process algebra Csp [Hoa85, Ros98, Ros10] provides a well established formalism to describe concurrent systems. While there is still ongoing research on foundations, Csp has found many applications in industry, e.g., in Train Controllers, Avionics, Security Protocols.

Csp describes reactive systems in terms of abstract, discrete events such as "train 12 enters track 1". The events of a system are collected in an *alphabet of communications* Σ. All these communications are atomic. In Csp terminology, a reactive system is a *process*. Such a process can be *Stop*, the system that does not do anything. Another basic process is *Skip*, which is a system that first performs the termination event $\checkmark$ (pronounced as tick) and then behaves like *Stop*. Given an event, say a, and a process, say P, one can form the new process $a \rightarrow P$ which first engages in a and then behaves like P. Csp offers two choice operators between processes: with internal choice $P \sqcap Q$, the newly formed process chooses between P and Q – with external choice $P \,\Box\, Q$, the environment chooses if P or Q is to be performed. Sequential composition of two processes P and Q passes control: $P;\ Q$ first behaves like P, should P terminate, it behaves like Q. Csp offers various operators to combine processes in parallel: $P \,|||\, Q$ runs the processes P and Q independent of each other; $P \,|[\, X \,]|\, Q$ requires them to cooperate on the events in the set X. There also is hiding $P \setminus X$ of events and renaming of events $f(P)$ using a function f on the events. Recursion offers the possibility to describe non terminating systems.

[Win02] describes how to model railway systems in Csp for proving their safety. The first step is to formalise the track plan as a graph using a functional language for data description:

```
datatype TrackIDs = Entry | Exit | AE | AF | AG | AH
next(t) = if (t==Entry) then {AE} else if (t==AE) ...
datatype SignalIDs = S1 | S2
signalhome(s) = if (s==S1) then Entry
                           else if (s==S2) then AF else AF
signalend(s) = if (s==S1) then AG
                          else if (s==S2) then AH else AH
```

Then, trains are modelled as processes. The state of a train is characterized by its identifier *id*, by the position of its *front* and its *rear*. Trains can move along the track plan only:

```
BehaveTrain(id.front.rear) =
([]t:next(front)@
    (front == rear &
         Moveff.front.t -> BehaveTrain(id.t.rear)))
    []
    (front != rear &
         Moverr.rear.front -> BehaveTrain(id.front.front)
)
```

In the beginning, conceptually we assume all trains to be on the *Entry* track:

```
Trains = ||| id : TrainIDs @ TrainBehave(id.Entry.Entry)
```

Finally, control tables are modelled: When the front of a train enters the protected area, the signal state becomes *Red* indicating "halt". Similarly, when the rear of a train leaves the protected area, the signal state becomes *Green* indicating "proceed".

```
SignalBehave(id.aspect) =
(aspect == Green &
 [] n : next(signalhome(id)) @
  moveff.signalhome(id).n -> SignalBehave(id.Red))
[]
(aspect == Red &
 [] n : next(signalend(id)) @
  moverr.signalend(id).n -> SignalBehave(id.Green))
```

In the beginning, all signals show *Green*:

```
Signals = ||| id : SignalIDs @ SignalBehave(id.Green)
```

Finally, the whole train system comprises trains and signals, which interact through a set of synchronized events:

```
TrainSystem =
Trains
[|
union(Union(
{{ moveff.signalhome(id).n |
   n<- next(signalhome(id)) } | id <- SignalIDs }),
Union(
{{ moverr.signalend(id).n |
   n<- next(signalend(id)) } | id <- SignalIDs }))
|]
Signals
```

We formalise the property "no collision" closely to [Win02]. In our example of the single line railway, a collision only occurs if the front of a train moves into an occupied track. Relatively to a set *Occ* of currently occupied tracks, the process *SafeMove* models all collision free train movement:

```
SafeMove(Occ) =
[] c : Occ @
  [] n : next(c) @ (not member(n,Occ) or n==Exit & moveff.c.n
                     -> SafeMove(union(Occ,{n}))
                    []
                    member(n,Occ) & moverr.c.n
                     -> SafeMove(union(diff(Occ,{c}),{Entry})))
```

Our model is safe iff it contains only safe moves. This can equivalently be formulated in Csp as the refinement statement over the traces of the respective processes:

$$\mathit{SafeMove}(\mathit{Entry}) \sqsubseteq_{\mathit{Traces}} \mathit{TrainSystem}$$

4 Timed CSP and timed traces

Time is an integral aspect of computer systems. It is essential for modelling a system's performance, but may also effect its safety or security. Timed Csp [Sch00] conservatively extends the process algebra Csp by timed primitives, where real numbers ≥ 0 model how time passes with reference to a single, conceptually global clock.

Syntactically, the core extension of Csp to Timed Csp is astonishingly small. There are only three new operators:

- $a@u \to P(u)$ – Time u of how long the event a has been delayed,
- $P \triangleright^d Q$ – timeout after d time units, and
- $P \triangle_e Q$ – interrupt after e time units.

Based on these, Timed Csp adds many operators as syntactic sugar. Most prominent are

- $\mathit{Wait}\ d = \mathit{Stop} \triangleright^d \mathit{Skip}$ – the process, which waits for d time units before it terminates – and
- $a \xrightarrow{d} P = a \to (\mathit{Stop} \triangleright^d P)$ – a delayed event prefix, which first performs a, then waits for d time units, before it behaves as P.

In Timed Csp, systems perform *timed events* $(r, e) \in \mathbb{R}_{\geq 0} \times \Sigma : r$ is the time at which event e occurs. Events are instantaneous, i.e., they do not take any time. The execution of a system leads to a *timed trace* which consists of all timed events that the system has performed. Such a timed trace $t = \langle (r_1, e_1), \ldots, (r_n, e_n) \rangle$ has the properties:

- t is ordered, i.e., $\forall\, 1 \leq i < j : r_i \leq r_j$, and
- only the last element of t can be the termination symbol ✓, i.e., $\forall\, 1 \leq i < n : e_i \neq \checkmark$.

The set of all timed traces is denoted as TT. The set of all timed traces of a Timed-CSP process P is denoted as $\mathcal{T}_{\mathbb{R}}[\![P]\!]$ which is a subset of TT. Given two Timed Csp processes P and Q, we say that Q refines to P in terms of timed traces, denoted by $Q \sqsubseteq_{TT} P$, iff $\mathcal{T}_{\mathbb{R}}[\![Q]\!] \supseteq \mathcal{T}_{\mathbb{R}}[\![P]\!]$. Furthermore, we define $\mathcal{T}_{\mathbb{Z}}[\![P]\!]$ to be the set of timed traces with integer time stamps only, that is

$$\mathcal{T}_{\mathbb{Z}}[\![P]\!] = \{\langle (r_1, e_1), \ldots, (r_n, e_n)\rangle \in \mathcal{T}_{\mathbb{R}}[\![P]\!] \mid \forall\, i : r_i \in \mathbb{Z}\}$$

Then, $Q \sqsubseteq^{\mathbb{Z}}_{TT} Q$, iff $\mathcal{T}_{\mathbb{Z}}[\![Q]\!] \supseteq \mathcal{T}_{\mathbb{Z}}[\![P]\!]$.

The empty time trace is written as $\langle\rangle$. Given a non-empty timed trace t, its visible beginning and ending time stamps are defined as follows respectively:

$$\begin{aligned} \mathit{begintime}(\langle (r_1, e_1), \ldots, (r_n, e_n)\rangle) &= r_1 \\ \mathit{endtime}(\langle (r_1, e_1), \ldots, (r_n, e_n)\rangle) &= r_n \end{aligned}$$

Then, the visible duration of a finite timed trace is defined as the difference of its beginning and ending times when the trace is not empty or 0 otherwise:

$$\mathit{duration}(t) = \begin{cases} \mathit{endtime}(t) - \mathit{begintime}(t) & \text{if } t \neq \langle\rangle \\ 0 & \text{otherwise} \end{cases}$$

Let us now recall some notations over timed traces. Given two timed traces t_1 and t_2, $t_1 \frown t_2$ denotes their concatenation[1]. Given a timed trace t, $\#t$ denotes the number of timed events occurring within t. Given a set of events A, $t \upharpoonright A$ denotes the projection of t onto A, which is the subsequence of t which consists of only events from A. Then, $t \downarrow A = \#(t \upharpoonright A)$ is the number of timed events from A in t.

With regards to recursion in Timed Csp, we use the setting defined in [OW03]: they show that the timed refusal traces with divergence model is a complete partial order and that every n-ary timed CSP operators can be interpreted as a continuous function. This avoids using Schneider's theory of timed-guarded recursion. Many of the recursive processes that we consider in the following are not timed-guarded, however, have a well-defined semantics thanks to [OW03].

5 Modelling timed behaviours of railway system

In the following, we make the assumptions concerning time in railways: We assume signalling to be instantaneous. In the real world, the cycle time of an interlocking is in the area of two seconds. This time is neglectable compared to the time a train needs to move from one track to another. Slightly more critical is our second assumption: For the moment being, we assume that trains accelerate and brake immediately. This is clearly an over simplification to be remedied in future work.

[1] Should the time stamps not match, then concatenation is not defined.

In order to model time, we enrich [Win02]'s model of a track plan. To this end, we associate each track with its length, denoted as $tracklength(t)$. Simplifying again, we say that the *tracklength* of a track t is identical with the minimal time that it takes a train to travel along t.

The second change to [Win02]'s model is that each train is associated with a train length, denoted by $trainlength(i)$. Again, this "length" stands for the minimal amount of time it takes a train to travel along an distance equal to its physical length[2]. Whenever performing an event *moveff*, the train has to wait for at least $trainlength(i)$ amount of time before being able to perform *moverr*; then it has to wait $(tracklength(t) - trainlength(i))$ amount of time before being able to perform the next *moveff*. The following Timed Csp code summarizes these changes:

```
TrainBehave(id.front.rear) =
if (front == rear and front == Exit) then Train(id)
else
(front == rear and front != Exit &
  [] n : next(front) @ moveff.front.n
     -{trainlength(id)}-> TrainBehave(id.n.rear))
 []
 (front != rear &
    moverr.rear.front
     -> (if (front==Exit) then SKIP
                      else WAIT
                           (tracklength(front)-trainlength(id)) ;
         TrainBehave(id.front.n))
```

The process *TrainBehave* defines now how the train behaves under timing conditions:

- The train can move the front if and only if both parts (front and end) of the train are on the same track. Then, it needs to wait for at least $trainlength(id)$ amount of time before any further events.
- It can move the back if and only if both of its parts are on different tracks. Then, it needs to wait for $tracklength(n) - trainlength(id)$ amount of time before any further events.

Thanks to our assumptions, all other processes remain as described in Section 3.

In Timed Csp, the process *TrainSystem* semantically defines the set

$$\mathcal{T}_{\mathbb{R}}[\![TrainSystem]\!]$$

of timed traces each of which corresponds to a possible movement of trains in the railway under the condition: The movements of the front and the rear of each train must trigger the required delays before the next behaviours of the train.

[2]We assume that $trainlength(i)$ is much smaller than $tracklength(t)$ for any train i and track t.

6 Modelling railway capacity

We first develop a semantic concept of capacity based on timed traces only. Then, we make capacity visible via a non-interfering observer process written in Timed Csp.

6.1 Capacity semantically

In this section, we present a measure of railway capacity which is compliant with the quotation in Section 2 and compatible with existing analytical methods such as [OR96]. Informally speaking, we want to count the number of trains appearing and operating within the railway. This number clearly depends on when we start counting and how long we observe. Thus, we speak of the *observation window* characterised by a starting time and a duration. There are two kinds of trains that we can observe in such a window: those trains, which are already present at starting time, and those trains, which appear in the window while it is open. For our notion of capacity, we take the duration as a parameter and maximise over all starting points.

In the previous section, we model the single line railway where the semantics of the process *TrainSystem* comprises the set of all possible timed traces describing the movement of trains in the railway. In other words, it captures all possible scenarios of trains operating in the railway which allows us to formally define the railway capacity in terms of timed traces.

Timed traces of train movements

As trains enter, travel a long and leave a railway, their movements can be recorded in a timed trace which includes the movement events of the front and the rear of trains from one track to the next. For the sake of simplicity, we assume that no train is in the railway initially.

While dealing with railway capacity, we are particularly interested in events in a timed trace which describe the entering and leaving of trains. In our single line example, $moveff.Entry.AE$ indicates the entering of a train into the railway, and $moverr.AH.Exit$ the leaving of a train out of the railway. In general, we define *Entering* and *Leaving* as the sets of timed events which indicate the entering and leaving of trains, respectively.

Storage characteristic

It is necessary to keep track the number of those trains, which are already in the railway, before the observation window starts. To this end, we study the history from the system start to the begin of the observation window in the form of a trace s. The number of trains in the system after s is determined by taking the number of trains entering the railway in s reduced by the number of trains leaving the railway in s.

Mathematically, we define a function *storage* as follows:

$$storage : TT \to \mathbb{N}$$

where

$$storage(s) = s \downarrow Entering - s \downarrow Leaving$$

Increase characteristic

From the start of the observation window until its end, we count the number of trains entering the railway. To this end, we study the history from the observation window start to the end of the observation window in form of a trace s. The number of trains entering the system after the begin of the observation window is determined by counting number of occurrence of timed events from *Entering* in s. Such a function is defined as follows:

$$increase : TT \to \mathbb{N}$$

where

$$increase(s) = s \downarrow Entering$$

Capacity

Finally, we formalise the measurement of capacity of a railway network with respect to a given length δ of an observation window. It is the maximal number of trains operating within δ over any possible train movements in $\mathcal{T}_{\mathbb{R}}[\![TrainSystem]\!]$. We define the capacity with respect to δ as follows:

$$\begin{array}{l} capacity(\delta) = \\ \max_{t \in \mathcal{T}_{\mathbb{R}}[\![TrainSystem]\!]} \{ storage(s_1) + increase(s_2) \mid \exists\, s_1, s_2, s_3 \in TT : \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad t = s_1 \frown s_2 \frown s_3 \text{ and} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad duration(s_2) \leq \delta \} \end{array}$$

The idea behind this definition is that, for each possible timed trace (or movement) of trains in $\mathcal{T}_{\mathbb{R}}[\![TrainSystem]\!]$, we associate different observations for measuring railway capacity depending on when the observations are starting – denoted by the prefix s_1 – and the length of the observations – denoted by s_2. The obvious condition for s_2 is that its duration in time must be δ. The number of trains on the railway before the observation starts is determined by $storage(s_1)$. Then, during the observation, we count new trains entering the railway within s_2 by $increase(s_2)$. Then, such an observation gives us the capacity as $storage(s_1) + increase(s_2)$. Then, the capacity of the railway is the maximal number of trains observed for any observation in any timed trace in $\mathcal{T}_{\mathbb{R}}[\![TrainSystem]\!]$. Furthermore, since the railway capacity is determined by using the function max, we can relax the condition on the length of the observations to be no more than δ, rather than to be exactly δ. Without such a relaxation, the definition of $capacity(\delta)$ would look much more cumbersome.

6.2 Capacity in Timed CSP

We realise the modelling of railway capacity in Timed Csp by defining an observer process which synchronises with *TrainSystem* over events indicating the entering and leaving of trains with respect to the railway. The definition of such a process will match with the definition of railway capacity which is introduced in the previous section. In particular, it is set up mainly by two processes. The first process plays its role before the observation starts. That is to keep track of the number of trains are already running on the railway. The second process, then, is responsible for counting the number of trains entering the railway during the observation.

Storage is the name given to the process which is responsible for keeping track the number of trains on the railway before the start of the observation window. It coincides with the notion of storage characteristic which is defined in the previous section. It contains events corresponding to that of a train entering and leaving which then increases or decreases the number of trains on the railway, respectively. The process is defined as below:

```
Storage(n) =
        ([] n1 : next(Entry) @ moveff.Entry.n1  -> Storage(n+1))
         []
        ([] n1 : pre(Exit)   @ moverr.n1.Exit  -> Storage(n-1))
         []
         startObs?delta -> Increase(n,0,delta)
```

In addition, the process can also decide to start the observation window through the event *startObs*. As soon as *startObs* is issued, the observation for counting capacity is started by means of the second process, namely $Increase(n, 0, delta)$ where:

- n is the number of trains which are on the railway already,
- 0 is the duration since the observation started, and
- *delta* is the size of the observation windows.

Because only entering trains are significant during the observation period, the number of trains is increased as soon as such an event occurs during. Otherwise, we simply keep this number unchanged. The process *Increase* is defined as follows:

```
Increase(n,d,delta) =
d<=delta &
        ([] n1 : next(Entry) @
            moveff.Entry.n1 @ u ->
                if d+u<=delta then Increase(n+1,d+u,delta)
                              else Infocap(n))
         []
        ([] n1 : pre(Exit) @
```

```
            moverr.n1.Exit @ u ->
                if d+u<=delta then Increase(n,d+u,delta)
                              else Infocap(n))

Infocap(n) =
         infocap.n -> EndCapObserver
         []
        ([] n1 : next(Entry) @
            moveff.Entry.n1 -> Infocap(n))
         []
        ([] n1 : pre(Exit) @
            moverr.n1.Exit ->  Infocap(n))

EndCapObserver =
        ([] n1 : next(Entry) @
            moveff.Entry.n1 -> EndCapObserver)
         []
        ([] n1 : pre(Exit) @
            moverr.n1.Exit ->  EndCapObserver)
```

After the observation period, events of trains entering or leaving do not change the record held in the first parameter of *Increase*. We also define this process such that it informs the recorded capacity by issuing an event *infocap.n* after the observation completes. The two auxiliary processes *Infocap* and *EndCapObserver* simply enable the issue of the event *infocap.n* and avoid falling into a deadlock, respectively.

We formalise that the observation window is controlled by a controller who has the power to decide when it starts, and later receives the result through the event *infocap*. This process is defined as follows:

```
Controller(delta) = startObs.delta -> infocap?n -> Stop
```

Since we assume that the railway is empty at the beginning, we define the observer process which starts the assumption that no train in the railway as follows:

Capacity incorporates with the process *TrainsSystem* by synchronising events of trains entering and leaving. This task is done through the use of the interface parallel operator:

```
TrainSystemWithCapacity =
    (TrainSystem
     [|
       union(
       { moveff.Entry.n | n <- next(Entry) },
       { moverr.n.Exit | n <- pre(Exit) })
     |]
     Storage(0))
     [| {|startObs, infocap|} |] Controller(delta)
```

We say that a process does not interfere another when being put in parallel if it always offers every event which is synchronised between two processes. Then, we have the following result for the coupling of *TrainSystem* and *Storage* in the definition of the process *TrainSystemWithCapacity*:

Theorem 1. *Storage*(0) *does not interfere TrainSystem*

The proof is straightforward by using the definition of processes *Storage*, *Increase*, *Infocap* and *EndCapObserver* where every event in $\{moveff.Entry.n \mid n \in next(Entry)\} \cup \{moverr.n.Exit \mid n \in pre(Exit)\}$ is alway ready to engage.

Theorem 2. *Given the length* δ *of the observation window,* $capacity(\delta) = n$ *iff*

- *for any* $n' \leq n$*, there exists a timed trace* $t \in \mathcal{T}_{\mathbb{R}}[\![TrainSystemWithCapacity]\!]$ *such that* $(r, infocap.n') \in t$ *for some* $r \in \mathbb{R}$*, and*
- *for any* $n' > n$*, there is no timed trace* $t \in \mathcal{T}_{\mathbb{R}}[\![TrainSystemWithCapacity]\!]$ *such that* $(r, infocap.n') \in t$ *for some* $r \in \mathbb{R}$*.*

Proof. Let us provide here a sketch proof of the lemma.

($\Rightarrow$) : Since $capacity(\delta) = n$, according to the definition of $capacity(\delta)$, there exists a timed trace $t \in \mathcal{T}_{\mathbb{R}}[\![TrainSystem]\!]$ such that $t = s_1 \frown s_2$ where $duration(s_2) \leq \delta$ and $storage(s_1) + increase(s_2) = n$. Thus, $t' = s_1 \frown \langle(begintime(s_2), startObs.\delta)\rangle \frown s_2 \frown \langle(endtime(s_2), infocap.n)\rangle \in \mathcal{T}_{\mathbb{R}}[\![TrainSystemWithCapacity]\!]$. Moreover, we can also select a suitable point of time in t to insert the event $startObs.\delta$ such that it later can issue the event $infocap.n'$ for any $n' < n$.

Furthermore, let us assume that $(r, infocap.n') \in t$ for some $n' > n$ and $t \in \mathcal{T}_{\mathbb{R}}[\![TrainSystemWithCapacity]\!]$ with $t = s_1 \frown \langle(r_1, startObs.\delta)\rangle \frown s_2 \frown s_3 \frown \langle(r, infocap.n')\rangle$ where $endtime(s_2) - r_1 \leq \delta$ and $storage(s_1) + increase(s_2) = n'$. Then, we can construct a timed trace $t' = s_1 \frown s_2$ which is in $\mathcal{T}_{\mathbb{R}}[\![TrainSystemWithCapacity]\!]$, however, this means $capacity(\delta) \geq n'$ which is a contradiction.

($\Leftarrow$) : We must have a timed trace $t \in \mathcal{T}_{\mathbb{R}}[\![TrainSystemWithCapacity]\!]$ which have the form of $t = s_1 \frown \langle(r_1, startObs.\delta)\rangle \frown s_2 \frown s_3 \frown \langle(r, infocap.n)\rangle$ where $duration(s_2) \leq \delta$ and $storage(s_1) + increase(s_2) = n$. However, this also means $s_1 \frown s_2 \in \mathcal{T}_{\mathbb{R}}[\![TrainSystem]\!]$, which implies that $capacity(\delta) \geq n$. We now show that $capacity(\delta)$ cannot be greater than n. Assume that $capacity(\delta) = n'$ for some $n' > n$. Then, there must be a trace $t = s_1 \frown s_2$ where $duration(s_2) \leq \delta$ and $storage(s_1) + increase(s_2) = n'$. However, we also have that $t' = s_1 \frown \langle(begintime(s_2), startObs.\delta)\rangle \frown s_2 \frown \langle(endtime(s_2), infocap.n')\rangle \in \mathcal{T}_{\mathbb{R}}[\![TrainSystemWithCapacity]\!]$ which is a contradiction.

□

7 Proving capacity and safety

As pointed out in [OW03], for simple systems and certain properties, it is possible to check refinement relations over Timed Csp using the model checker FDR. To this end, the processes involved are translated into a special variant of Csp, namely Tock-Csp. Often, this translation can be achieved via straight-forward coding tricks [Oua02]. Semantically, this transformation is justified thanks to a result presented in [OW03]: Given two Timed Csp processes P and Q, where $\mathcal{T}_{\mathbb{R}}[\![Q]\!]$ is closed under inverse digitisation, then $Q \sqsubseteq_{TT} P$ iff $Q \sqsubseteq^{\mathbb{Z}}_{TT} P$.

In our encoding, the process *TrainSystemCapacity* models the system of trains running on the single line railway example, coupled with an observer who counts the number of train operating in the railway in a given period of time *delta*. If the observer issues an event $infocap.n$, it means the railway is able to allow at least n trains operating within *delta*. We formalise the property where a railway can allow at most n trains operating, which only concerns with events issued by the observer ($startObs, infocap.n$), as follows:

```
CapacityFrom(n) = |~| n' : {0..n} @ startObs -> infocap.n' -> Stop
```

Then, we have the following result, which can be proved by means of Theorem 2 and the definition of $CapacityFrom(n)$:

Theorem 3. *Given a length δ of observation, $capacity(\delta) = n$ iff*

- *for all $k \geq n$ holds:*
  $CapacityFrom(k) \sqsubseteq_{TT} TrainSystemCapacity \setminus MoveEvents$*, and*
- *for all $0 \leq l < n$ holds:*
  $CapacityFrom(l) \not\sqsubseteq_{TT} TrainSystemCapacity \setminus MoveEvents$*.*

where $MoveEvents = \{moveff.x.y \mid x, y \in Tracks\} \cup \{moverr.x.y \mid x, y \in Tracks\}$

Furthermore, since the definition of the process does not include any time expression, $CapacityFrom(n)$ is a qualitative property, which, according to [HMP92], is closed under inverse digitisation. Therefore, instead of checking the refinement relation $\sqsubseteq_{TT}$, we can translate the processes into Tock CSP and check the refinement relation $\sqsubseteq^{\mathbb{Z}}_{TT}$. Such a check is well supported in FDR.

For safety, the same argument as for $CapacityFrom(n)$ holds: the process *SafeMove* is closed under inverse digitisation. Thus, again we can replace $\sqsubseteq_{TT}$ by $\sqsubseteq^{\mathbb{Z}}_{TT}$.

8 Studying safety and capacity of the single line

In this section, we present the results provided by the model checker FDR concerning capacity and safety. These give a formal argument that capacity increases when the signalling rule is changed from Scenario 1 to Scenario 2.

As pointed out in the previous section, prior to the usage of FDR, we are required to translate our formal model from Timed CSP into CSP in which time is modelled by the special event *tock*. The detail of the encoding and its translation into CSP with tock is listed in Appendix A and B. In our case, the translation from Timed CSP into CSP with tock expands the model up to roughly 33% in terms of size.

In the experiment, we make the following assumptions about the single line example:

- $\mathit{trainlength}(id) = 1$ for every train id,
- $\mathit{tracklength}(tr) = 3$ for all tracks tr, and
- the length of the observation window is fixed to 30 units of time.

In both scenarios FDR establishes safety. However, FDR shows different results concerning capacity. The following table summaries the result provided by FDR on capacity:

	1	2	3	4	5	6	7	8	9	10	Running time
Scenario 1	x	x	x	x	✓	✓	✓	✓	✓	✓	7 s
Scenario 2	x	x	x	x	x	x	✓	✓	✓	✓	15 s

Each row in the table provides the result for each scenario, while each column n expresses if the refinement relation

$$\mathit{Capacity}(n) \sqsubseteq^{\mathbb{Z}}_{TT} \mathit{TrainSystemCapacity}\ \mathit{MoveEvents}$$

holds, denoted by "✓", or not, denoted by "x". The last column shows how long FDR spends roughly for running all checks needed for safety and capacity of each scenario (on a machine with a 2 GHz 64 bit processor equipped with a 4GByte memory). Note that even in this simple example the running time of FDR doubles due to the increased number of trains that can be observed.

The table shows that, on the observation window of length 30 units of time, Scenario 1 offers the capacity of 5 trains. By removing the overlap as in Scenario 2, while the safety is still guaranteed to hold, the capacity is improved to 7 trains.

9 Summary and Future Work

We have provided a formal definition of line capacity based on the timed traces that one can observe in a natural, timed model of railway systems. This definition can equivalently be characterized as a refinement statement in Timed Csp. Re-using the safety formulation by [Win02], this allows us to study both, safety and capacity, in one formal model in Timed Csp.

As the refinements for safety and capacity only require the checking of a qualitative property, both refinement statements can be discharged by translation to untimed Csp. This approach has the advantage that one can re-use the established model checker FDR. For more complex examples, e.g., with long

delays or involving delays of different sizes, the translational approach is expected to become inefficient. Here, dedicated proof support, e.g., in the form of a Timed Csp-Prover will become necessary.

Concerning our single line example, we could show that our definition of line capacity works in the expected way: Scenario 2 has higher capacity than Scenario 1. The translational approach worked in the single line example: Both scenarios are safe; in both scenarios we could – after some guess work – determine the capacity.

It is future work to test our reasoning on capacity on more realistic examples concerning track length, speed of trains, breaking curves etc. Furthermore, it will be necessary to calibrate our notion of capacity by comparing its numerical value with behaviour observed in the real world. Another aspect is the development of better and more automated proof support for reasoning on line capacity. Finally, we intend to develop our definition further, so that it also captures the more complex notion of network capacity.

Acknowledgement The authors would like to thank Simon Chadwick and Dominic Taylor from the company Invensys for their encouraging feedback and Erwin R. Catesbeiana (Jr) for pointing out that immobility is the enemy of capacity.

References

[ABI+08] M. Abril, F. Barber, L. Ingolotti, MA Salido, P. Tormos, and A. Lova. An assessment of railway capacity. *Transportation Research Part E: Logistics and Transportation Review*, 44(5):774–806, 2008.

[DGR11] Marc Dragon, Andy Gimblett, and Markus Roggenbach. A Simulator for Timed CSP. In *AVoCS'11*. Technical Report. Newcastle University, 2011.

[DHSZ06] Jin Song Dong, Ping Hao, Jun Sun, and Xian Zhang. A reasoning method for Timed CSP based on constraint solving. In *ICFEM'06*, LNCS 4260. Springer, 2006.

[HMP92] T. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? *Automata, Languages and Programming*, pages 545–558, 1992.

[Hoa85] Tony Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[IR] Yoshinao Isobe and Markus Roggenbach. Webpage on CSP-Prover. `http://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html`.

[IR05] Yoshinao Isobe and Markus Roggenbach. A generic theorem prover of CSP refinement. In *TACAS 2005*, LNCS 3440. Springer, 2005.

[KR01] D. Kerr and T. Rowbotham. *Introduction To Railway Signalling.* Institution of Railway Signal Engineers, 2001.

[OR96] UIC 405 OR. Links between railway infrastructure capacity and the quality of operations. *International Union of Railways Leaflet*, 1996.

[oT07] Department of Transport. Delivering a sustainable railway, 2007.

[Oua02] J. Ouaknine. Digitisation and full abstraction for dense-time model checking. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 23–37, 2002.

[OW03] J. Ouaknine and J. Worrell. Timed csp= closed timed ε-automata. *Nordic Journal of Computing*, 10:1–35, 2003.

[Ros98] Bill Roscoe. *The theory and practice of concurrency.* Prentice Hall, 1998.

[Ros10] Bill Roscoe. *Understanding Concurrent Systems.* Springer, 2010.

[Sch00] Steve Schneider. *Concurrent and Real-time systems.* Wiley, 2000.

[SWD97] A. Simpson, J. Woodcock, and J. Davies. The mechanical verification of solid-state interlocking geographic data. In *Formal Methods Pacific 97*, pages 223–243. Springer, 1997.

[Win02] Kirsten Winter. Model checking railway interlocking systems. *Australian Computer Science Communications*, 24(1), 2002.

A Timed CSP code for the model of the single line

We list the code for encoding the model of the single line in Timed CSP in the following. The signalling rule here is for the case of Scenario 1.

```
-- track plan
datatype TrackIDs = Entry | Exit |
           AE | AF | AG | AH
datatype SignalIDs = S1 | S2
datatype Aspect = Green | Red

TrainIDs = {1,2,3,4,5,6,7,8,9,10}

-- topo
trainlength(t) =
if (member(t,TrainIDs)) then 1
                        else 0

tracklength(t) =
if (member(t,TrackIDs)) then 3 else 0

next(t) =
if (t==Entry) then {AE} else
if (t==AE) then {AF} else
if (t==AF) then {AG} else
if (t==AG) then {AH} else
if (t==AH) then {Exit} else {}

pre(t) = { t1 | t1 <- TrackIDs, member(t,next(t1)) }

signalhome(s) =
if (s==S1) then Entry else
if (s==S2) then AF else AF

signalend(s) =
if (s==S1) then AG else
if (s==S2) then AH else AH

-- define some constants
maxlenOfObs = 20
lenOfObs = 10
maxcap   = 10


-- define types, channels
channel moveff, moverr : TrackIDs.TrackIDs
channel infocap  : {0..maxcap}
channel startObs : {0..maxlenOfObs}
```

```
-- define the trains process
Train(id) = TrainBehave(id.Entry.Entry)

TrainBehave(id.front.rear) =
if (front == rear and front == Exit) then Train(id)
else
((front == rear and front != Exit &
  [] n : next(front) @ moveff.front.n
     -{trainlength(id)}-> TrainBehave(id.n.rear))
 []
 (front != rear &
  moverr.rear.front
     -> if (front==Exit) then SKIP
                         else
                          WAIT(tracklength(front)-trainlength(id)) ;
                          TrainBehave(id.front.front)))

Trains =
||| id : TrainIDs @ Train(id)

-- define signal process
SignalBehave(id.aspect) =
(aspect == Green &
 [] n : next(signalhome(id)) @
  moveff.signalhome(id).n -> SignalBehave(id.Red))
[]
(aspect == Red &
 [] n : next(signalend(id)) @
  moverr.signalend(id).n -> SignalBehave(id.Green))

Signals = ||| id : SignalIDs @ SignalBehave(id.Green)

-- combine trains and signals
TrainSystem =
Trains
[|
union(Union(
{{ moveff.signalhome(id).n |
   n<- next(signalhome(id)) } | id <- SignalIDs }),
Union(
{{ moverr.signalend(id).n |
   n<- next(signalend(id)) } | id <- SignalIDs }))
|]
Signals

-- define processes for

Storage(n) =
       (([] n1 : next(Entry) @ moveff.Entry.n1  -> Storage(n+1))
```

```
         []
        ([] n1 : pre(Exit)   @ moverr.n1.Exit  -> Storage(n-1))
         []
         startObs?delta -> Increase(n,0,delta))


Increase(n,d,delta) =
d<=delta &
       ([] n1 : next(Entry) @ moveff.Entry.n1 @ u ->
              if d+u<=maxdur then Increase(n+1,d+u,delta)
                             else Infocap(n)
        []
       ([] n1 : pre(Exit) @ moverr.n1.Exit @ u ->
              Increase(n,d+u,delta)))

Infocap(n) =
        infocap.n -> EndCapObs
        []
       ([] n1 : next(Entry) @ moveff.Entry.n1 -> Infocap(n))
        []
       ([] n1 : pre(Exit) @ moveff.n1.Exit -> Infocap(n))

EndCapObs =
       ([] n1 : next(Entry) @ moveff.Entry.n1 -> EndCapObs)
        []
       ([] n1 : pre(Exit) @ moveff.n1.Exit -> EndCapObs)

ObsController(delta) = startObs.delta -> infocap?n -> Stop

TrainSystemWithCapacity(delta) =
(TrainSystem
[|
union(
{ moveff.Entry.n | n <- next(Entry) },
{ moverr.n.Exit | n <- pre(Exit) })
|]
Storage(0))
[| {| startObs, infocap |} |] ObsController(delta)

SafeMove(Occ) =
[] c : Occ @
   [] n : next(c) @ (not member(n,Occ) or n==Exit &
                     moveff.c.n -> SafeMove(union(Occ,{n}))
                     []
                     member(n,Occ) & moverr.c.n ->
                        SafeMove(union(diff(Occ,{c}),{Entry})))

assert TrainSystem :[ deadlock free [F] ]
assert SafeMove({Entry}) [TT= TrainSystem
```

```
CapacityFrom(n,delta) = |~| n' : {0..n} @
       startObs.delta -> infocap.n' -> Stop

assert CapacityFrom(0,lenOfObs) [TT=
       TrainSystemWithCapacity(lenOfObs) \ {| moveff,moverr |}
assert CapacityFrom(1,lenOfObs) [TT=
       TrainSystemWithCapacity(lenOfObs) \ {| moveff,moverr |}
assert CapacityFrom(2,lenOfObs) [TT=
       TrainSystemWithCapacity(lenOfObs) \ {| moveff,moverr |}
assert CapacityFrom(3,lenOfObs) [TT=
       TrainSystemWithCapacity(lenOfObs) \ {| moveff,moverr |}
assert CapacityFrom(4,lenOfObs) [TT=
       TrainSystemWithCapacity(lenOfObs) \ {| moveff,moverr |}
assert CapacityFrom(5,lenOfObs) [TT=
       TrainSystemWithCapacity(lenOfObs) \ {| moveff,moverr |}
assert CapacityFrom(6,lenOfObs) [TT=
       TrainSystemWithCapacity(lenOfObs) \ {| moveff,moverr |}
assert CapacityFrom(7,lenOfObs) [TT=
       TrainSystemWithCapacity(lenOfObs) \ {| moveff,moverr |}
assert CapacityFrom(8,lenOfObs) [TT=
       TrainSystemWithCapacity(lenOfObs) \ {| moveff,moverr |}
```

It is also worth noticing that the encoding for the case of the Scenario 2 can be obtained simply by revising the output of $signalend(S1)$ to AF.

B The code in CSP with tock for the single line

We list here the translated code from Timed CSP into CSP with tock which can be tested using FDR. Once again, the code is for the case of Scenario 1. One can obtain the case of Scenario 2 by applying the same trick as remarked for the code in Timed CSP.

```
-- track plan
datatype TrackIDs = Entry | Exit |
           AE | AF | AG | AH
datatype SignalIDs = S1 | S2
datatype Aspect = Green | Red

TrainIDs = {1,2,3,4,5,6,7,8,9,10}

-- topo
trainlength(t) =
if (member(t,TrainIDs)) then 1
                        else 0

tracklength(t) =
if t==AE or t==AG then 3 else
if t==AF or t==AH then 3 else 0
```

```
next(t) =
if (t==Entry) then {AE} else
if (t==AE) then {AF} else
if (t==AF) then {AG} else
if (t==AG) then {AH} else
if (t==AH) then {Exit} else {}

pre(t) = { t1 | t1 <- TrackIDs, member(t,next(t1)) }

signalhome(s) =
if (s==S1) then Entry else
if (s==S2) then AF else AF

signalend(s) =
if (s==S1) then AG else
if (s==S2) then AH else AH

-- define some constants
delta  = 30
maxcap = 20
maxdur = 35


-- define types, channels
channel moveff, moverr : TrackIDs.TrackIDs
channel infocap  : {0..maxcap}
channel startObs : {0..delta}
channel tock

tocks(n,P) =
0 <= n and n<=maxdur & if n==0 then P else tock -> tocks(n-1,P)

-- define the trains process
Train(id) = TrainBehave(id.Entry.Entry)

TrainBehave(id.front.rear) =
if (front == rear and front == Exit) then Train(id)
else
((front == rear and front != Exit &
  [] n : next(front) @ moveff.front.n
     -> tocks(trainlength(id),TrainBehave(id.n.rear)))
 []
 (front != rear &
  moverr.rear.front
     -> if (front==Exit) then TrainBehave(id.front.front)
                         else tocks(tracklength(front)-
                                    trainlength(id),
                                 TrainBehave(id.front.front)))
 []
 (tock -> TrainBehave(id.front.rear)))
```

```
Trains =
[| {| tock |} |] id : TrainIDs @ Train(id)

-- define signal process
SignalBehave(id.aspect) =
(aspect == Green &
 [] n : next(signalhome(id)) @
  moveff.signalhome(id).n -> SignalBehave(id.Red))
[]
(aspect == Red &
 [] n : next(signalend(id)) @
  moverr.signalend(id).n -> SignalBehave(id.Green))
[]
(tock -> SignalBehave(id.aspect))

Signals = [| {| tock |} |] id : SignalIDs @ SignalBehave(id.Green)

-- combine trains and signals
TrainSystem =
Trains
[|
union(union(Union(
{{ moveff.signalhome(id).n |
   n<- next(signalhome(id)) } | id <- SignalIDs }),
Union(
{{ moverr.signalend(id).n |
   n<- next(signalend(id)) } | id <- SignalIDs })),
{| tock |})
|]
Signals

-- define processes for observing capacity

Storage(n) =
0<=n and n<=maxcap &
       (([] n1 : next(Entry) @ moveff.Entry.n1  -> Storage(n+1))
         []
        ([] n1 : pre(Exit)   @ moverr.n1.Exit  -> Storage(n-1))
         []
         startObs?delta -> Increase(n,0)
         []
        (tock -> Storage(n)))


Increase(n,d) =
0<=n and n<=maxcap and
0<=d and d<=delta+1 &
if (d<=delta) then
      (([] n1 : next(Entry) @ moveff.Entry.n1  -> Increase(n+1,d))
```

```
        []
       ([] n1 : pre(Exit) @ moverr.n1.Exit -> Increase(n,d))
        []
       (tock -> Increase(n,d+1)))
else Infocap(n)

Infocap(n) =
0<=n and n<=maxcap &
      ((infocap.n -> EndCapObs)
        []
       ([] n1 : next(Entry) @ moveff.Entry.n1 -> Infocap(n))
        []
       ([] n1 : pre(Exit) @ moverr.n1.Exit -> Infocap(n))
        []
       (tock -> Infocap(n)))

EndCapObs =
      (([] n1 : next(Entry) @ moveff.Entry.n1 -> EndCapObs)
        []
       ([] n1 : pre(Exit) @ moverr.n1.Exit -> EndCapObs)
        []
       (tock -> EndCapObs))

ObsController =
       (startObs.delta -> ObsController1)
       []
       (tock -> ObsController)

ObsController1 =
       (infocap?n -> ObsController2)
       []
       (tock -> ObsController1)

ObsController2 = tock -> ObsController2

TrainSystemWithCapacity =
(TrainSystem
[|
union(union(
{ moveff.Entry.n | n <- next(Entry) },
{ moverr.n.Exit | n <- pre(Exit) }),
{| tock |})
|]
Storage(0))
[| {| startObs, infocap, tock |} |] ObsController

SafeMove(Occ) =
[] c : Occ @
   [] n : next(c) @ (not member(n,Occ) or n==Exit &
                     moveff.c.n -> SafeMove(union(Occ,{n}))
```

```
                    []
                    member(n,Occ) & moverr.c.n ->
                       SafeMove(union(diff(Occ,{c}),{Entry})))
[] tock -> SafeMove(Occ)


assert TrainSystem :[ deadlock free [F] ]
assert TrainSystemWithCapacity :[ deadlock free [F] ]

assert SafeMove({Entry}) [T=
           TrainSystem :[ tau priority over ]:   { tock }

CapacityFrom(n) =
0<=n and n<=maxcap &
|~| n' : {0..n} @ CapacityFrom1(n')

CapacityFrom1(n) =
0<=n and n<=maxcap &
((startObs.delta -> CapacityFrom2(n))
 []
 (tock -> CapacityFrom1(n)))

CapacityFrom2(n) =
0<=n and n<=maxcap &
((infocap.n->CapacityFrom3)
 []
 (tock -> CapacityFrom2(n)))

CapacityFrom3 = tock -> CapacityFrom3



assert CapacityFrom(0) [T=
           TrainSystemWithCapacity \ {| moveff,moverr |}
           :[ tau priority over ]: {tock}
assert CapacityFrom(1) [T=
           TrainSystemWithCapacity \ {| moveff,moverr |}
           :[ tau priority over ]: {tock}
assert CapacityFrom(2) [T=
           TrainSystemWithCapacity \ {| moveff,moverr |}
           :[ tau priority over ]: {tock}
assert CapacityFrom(3) [T=
           TrainSystemWithCapacity \ {| moveff,moverr |}
           :[ tau priority over ]: {tock}
assert CapacityFrom(4) [T=
           TrainSystemWithCapacity \ {| moveff,moverr |}
           :[ tau priority over ]: {tock}
assert CapacityFrom(5) [T=
           TrainSystemWithCapacity \ {| moveff,moverr |}
           :[ tau priority over ]: {tock}
```

```
assert CapacityFrom(6) [T=
          TrainSystemWithCapacity \ {| moveff,moverr |}
          :[ tau priority over ]: {tock}
assert CapacityFrom(7) [T=
          TrainSystemWithCapacity \ {| moveff,moverr |}
          :[ tau priority over ]: {tock}
assert CapacityFrom(8) [T=
          TrainSystemWithCapacity \ {| moveff,moverr |}
          :[ tau priority over ]: {tock}
assert CapacityFrom(9) [T=
          TrainSystemWithCapacity \ {| moveff,moverr |}
          :[ tau priority over ]: {tock}
assert CapacityFrom(10) [T=
          TrainSystemWithCapacity \ {| moveff,moverr |}
          :[ tau priority over ]: {tock}
```