Specification of Radio Based Railway Crossings with the Combination of CSP, OZ, and DC *

Jochen Hoenicke

Fachbereich Informatik
Universität Oldenburg
26111 Oldenburg, Germany

hoenicke@informatik.uni-oldenburg.de

Abstract. We use a combination of three techniques for the specification of processes, data and time: CSP, Object-Z and Duration Calculus. Whereas the combination of CSP and Object-Z is well established by the work of C. Fischer [2, 3], the integration with Duration Calculus is new. The combination is used to specify parts of a novel case study on radio controlled railway crossings.

1 Introduction

Complex computing systems exhibit various behavioural aspects such as communication between components, state transformation inside components, and real-time constraints on the communications and state changes. Formal specification techniques for such systems have to be able to describe all these aspects. Unfortunately, a single specification technique that is well suited for all these aspects is yet not available. Instead one finds various specialised techniques that are very good at describing individual aspects of system behaviour. This observation has led to research into the combination and semantic integration of specification techniques. In this paper we use the combination of three well researched specification techniques: CSP [6, 7], Object-Z [13, 11] and Duration Calculus [14, 5].

The combination, CSP-OZ-DC, is applied to specify the case study of radio based railway crossings [8][1]. The main issue is that a train secures crossings and switches points via radio based messages without going through a central signal box, see Fig. 1. In this paper we will concentrate on the specification only. The purpose is to illustrate the powers of the combination CSP-OZ-DC to specify complex systems.

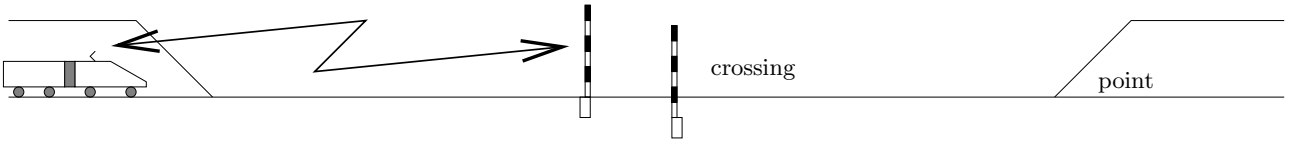

Fig. 1. Case study: Radio controlled railway crossings

The paper is organised as follows. Section 2 gives an introduction to the case study of radio controlled railway crossing. Section 3 introduces the main constructs of CSP-OZ-DC with some examples from the case study. Section 4 applies the new specification language to the case study. Finally, we conclude with section 5.

* This research is partially supported by the DFG under grant Ol/98-2.

[1] This case study is part of the priority research program "Integration of specification techniques with applications in engineering" of the German Research Council (DFG) (`http://tfs.cs.tu-berlin.de/projekte/indspec/SPP/index.html`).

2 Case Study

The main issue in the case study is to remotely operate points and crossings via radio based communication while keeping the safety standard. Figure 2 surveys the controller architecture we want to specify in this case study. The diagram shows several components connected by communication channels. In the centre of the diagram is the *train controller* whose purposes are to limit the speed of the train, decide when it is time to switch points and secure crossings, and make sure that the train does not enter them too early.

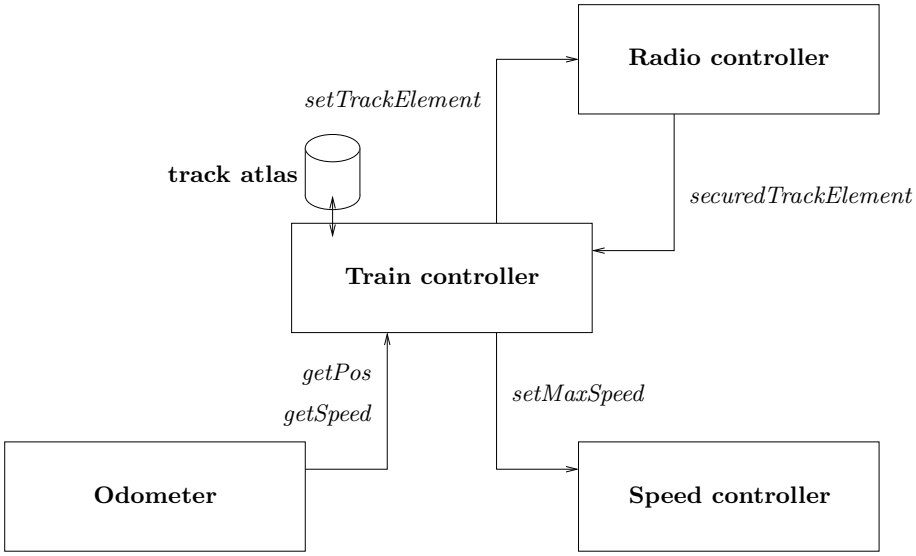

Fig. 2. Controller architecture

The *odometer* keeps track of the speed and position of the train. The position is measured by various means, e. g. counting rotations of wheels. Distributed over the track, there are *balises*, which are devices with a unique identifier that can be activated and read out by a passing train. With the help of these balises the odometer can determine the absolute position of the train. The *speed controller* supervises the speed and makes sure that it does not exceed the limit set by the train controller, otherwise it automatically slows down the train. When the speed limit is set to zero, the train will break until it comes to a safe halt. The communication with crossings is done by the *radio controller*. As said above, the communication medium is radio based. Special care has to be taken, because radio transmissions are inherently unsafe. The safety must still be established under the assumption that no message can be transferred.

Most work is done by the *train controller* so in this paper we will concentrate on this component. To specify this component several aspects must be handled, as described in the following. The train controller communicates with other components, e. g. a radio controller, that takes care of the radio protocol. Besides the communication aspects there are also data aspects in the case study: The train controller maintains a representation of the track, the track atlas. It further has to remember which crossings were notified and which are secure. Last but not least there are also real-time aspects: The train must continuously supervise its maximum speed. If it approaches a crossing it should secure it at the right moment, not too early and not too late. It is also safety critical that it breaks in time if the booms of the crossing do not close.

Each of these aspects can be expressed in CSP, OZ or DC respectively. In the next section we will introduce these languages and show in which way they are combined.

3 The Combination CSP-OZ-DC

In the last section we have seen that three aspects are important to specify the train controller. The first aspect is communication. Most of the communications are initiated by the train controller itself, e. g. the train controller decides when it is time to secure a track element. But there are also communications initiated externally, e. g. the signal that is sent when a crossing affirms that it is safe. These communications can be naturally modelled with CSP.

As an example we can model the loop supervising the speed in CSP by the following recursive equation:

$$\begin{aligned} SuperviseSpeed \stackrel{c}{=} & \; getSpeed \rightarrow getPos \\ & \rightarrow calcMaxSpeed \rightarrow setMaxSpeed \rightarrow SuperviseSpeed \end{aligned}$$

The symbol $\stackrel{c}{=}$ is used instead of an ordinary equals symbol to distinguish between CSP process equations and Z equations. The process specifies that the four events *getSpeed*, *getPos*, *calcMaxSpeed* and *setMaxSpeed* are communicated in this order. Afterwards the process calls itself again to allow for another cycle. For simplicity communication values are ignored here. This process can work in parallel with other processes, which is noted in CSP like this:

$$\texttt{main} \stackrel{c}{=} SuperviseSpeed \parallel OtherParallelProcess$$

Besides communication there are data aspects in the train controller. The *track atlas* contains a data base with the crossings and points. The train has to read this data base and also need to keep its own data structures to remember which track elements have already been switched and which have affirmed their safety. Specifying data structures and data bases is easily done with Object-Z (OZ). Starting from basic types *Identifier* and *Position* we can define the track elements (crossings and points) by the Z schema.

TrackElement

$id : Identifier$
$pos : Position$

This schema declares a new data type *TrackElement*. Each element of this type consists of several components listed inside the schema box. Each track element has a unique identifier *id*. There is also a position associated with each track element, which is the position at which the train must stop if it cannot secure it.

The track atlas contains information about track elements and the maximum speed for each track segment. It is also represented by a Z schema as follows. The *StaticProfile* is for storing the maximum speed and is not of interest here. The type seq *TrackElement* denotes finite sequences of *TrackElement*s.

TrackAtlas

$staticprof : StaticProfile$
$elems : \text{seq } TrackElement$

The data aspect has to interact with the communication aspect. The combination CSP-OZ [2, 3] provides a simple mechanism to bind a communication to a OZ-schema describing the effect to the data base. The following schema connects an action with the (internal) event *clearDangerPosition*. This event is not externally visible but is used to link the CSP and OZ parts. Whenever this event is triggered by a CSP process a position should be removed from the set of danger positions. This is done by writing a Z-schema with the name com_*clearDangerPosition* specifying the operation associated with that communication event.

com_*clearDangerPosition*

$$\Delta(dangerPositions)$$
$$id? : Identifier$$

$$\exists_1 elem : \operatorname{ran} trackatlas.elems \mid elem.id = id? \bullet$$
$$dangerPositions' = dangerPositions \setminus \{elem.pos\}$$

The Δ in the first line of this schema declares that this operation changes *dangerPositions*. The next line declares a parameter *id*, decorated with ? to signify that *id* is an input parameter. Notice that this naming convention of Z corresponds nicely with the naming conventions of CSP: the output of *id* along channel *clearDangerPosition* synchronises with the input of *id* in the Z schema. In Z the transformation of a state is expressed by a relation between the state before the operation and the state after the operation. The second state is distinguished from the first one by decorating it with a prime. The predicate relating the two states is given below the horizontal line. In this case we require that there is a track element *elem* with the identifier *id*? and that its position *elem.pos* is removed from the set of danger positions. If there is no element for that identifier the Z part blocks the communication.

To maintain safety, the train has to supervise the track repeatedly and must set the speed limit in time. This requires real-time constraints. Another aspect where real-time is important is the securing of crossings. If the train secures them too early, the traffic is unnecessarily blocked. If the train secures them too late, there is not enough time to close the gate before the train reaches the crossing.

For specifying real-time constrains, we use Duration Calculus (DC). As an example consider the following DC formula which states that the next *setMaxSpeed* communication must occur after at most one second:

$$\lceil count(setMaxSpeed) = n \rceil \xrightarrow{1s} \lceil count(setMaxSpeed) > n \rceil$$

The brackets $\lceil \cdot \rceil$ express that the enclosed predicate should hold all over a given time interval. A formula of the form $P \xrightarrow{t} Q$ states that whenever we have a time interval of length t where P holds it must be followed by an interval where Q holds. In DC all observations must have a duration in order to be visible. CSP events, however, happen at a single point in time, so we cannot observe them directly. Instead we count the number of times they did occur and reason about these values. The above formula states that if the number of *setMaxSpeed* events stays stable for one second, then the event has to occur afterwards so that *count*(*setMaxSpeed*) increases.

The basic building block in our combined formalism CSP-OZ-DC is a class. Its syntax is very similar to CSP-OZ [2, 3], only the DC part is new, see Fig. 3. First, the communication

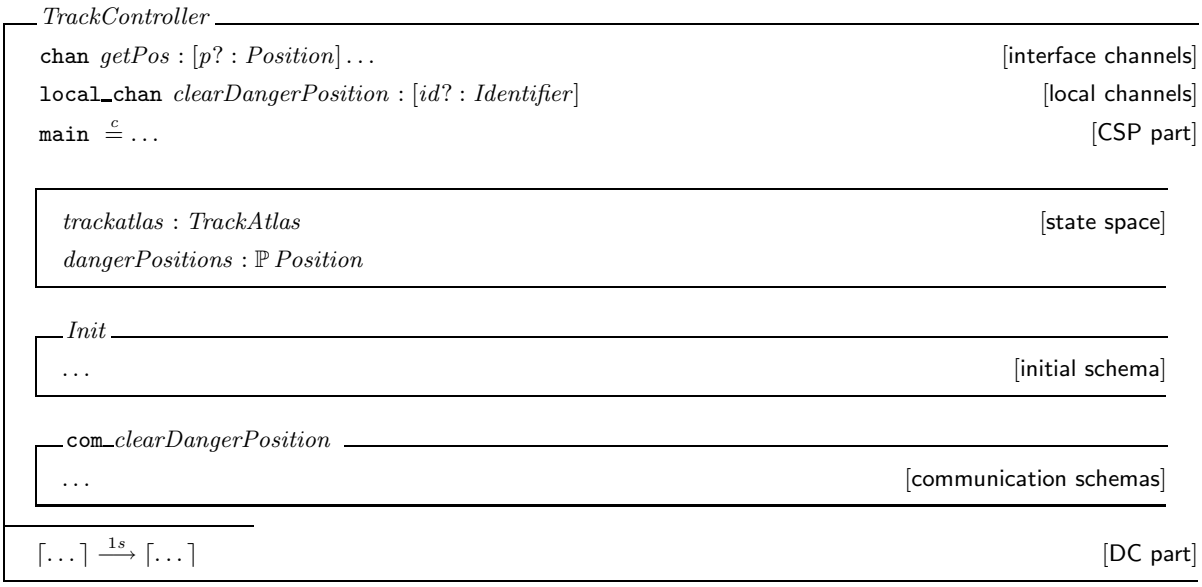

Fig. 3. Class in CSP-OZ-DC

channels of the class are declared. Every channel has a type which restricts the values that it can communicate. There are also local channels that are visible only inside the class and that are used by the CSP, Z, and DC parts for interaction. Second, the CSP part follows; it is given by a system of (recursive) process equations. Third, the Z part is given which itself consists of the state space, the Init schema and communication schemas. For each communication event a corresponding communication schema specifies in which way the state should be changed when the event occurs. Finally, below a horizontal line the DC part is stated.

Classes can be combined into larger specifications by CSP operators like parallel composition, hiding and renaming.

4 Applying CSP-OZ-DC to the Case Study

In Sect. 3 we have already introduced the case study of radio controlled railway crossing. In this section we want to take a closer look at the train controller, especially how it calculates the maximum speed. One central idea is the *braking curve*, see Fig. 4, which is a function that gives for each position on the track the maximum admissible speed. The braking curve consists of two parts: a static profile, a step function giving the admissible speed for each track segment, and a dynamic profile which takes care of unsafe crossings and of the braking characteristics of the train.

Before we go into the details of the braking curve, we first declare the basic data types in our case study. *Identifier* and *Direction* are abstract types, *Position* and *Speed* are represented by real numbers.

$$[Identifier, Direction]$$
$$Position == \mathbb{R}$$
$$Speed == \mathbb{R}$$
$$StaticProfile == \mathrm{seq}(Position \times Speed)$$

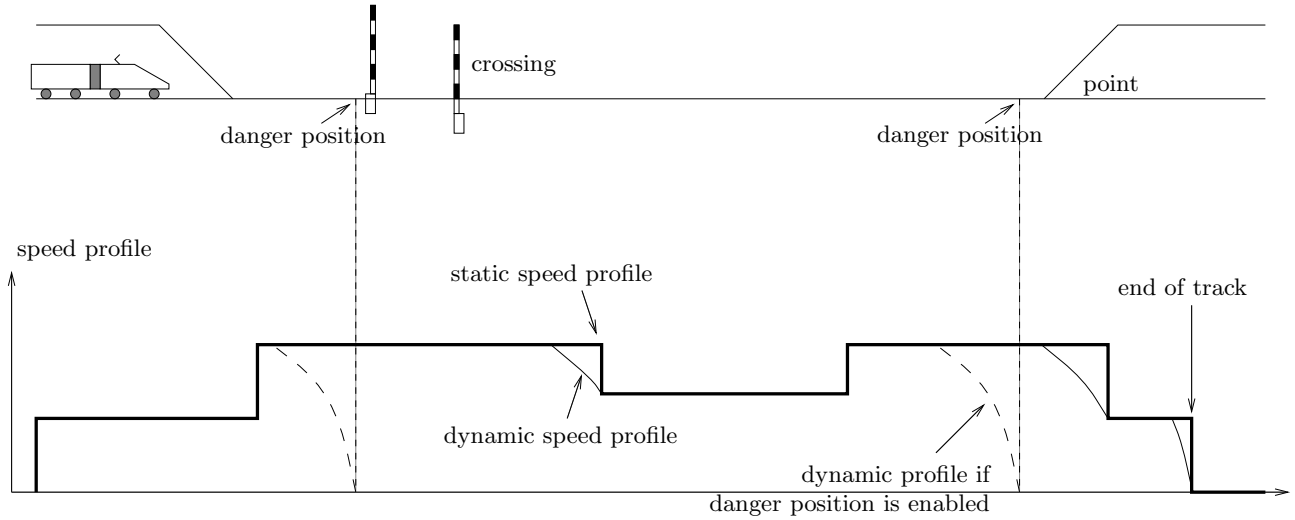

Fig. 4. Braking curve [4]

As said above the *StaticProfile* is a step function. It is represented here as a finite sequence of speed changes, each consisting of a position and a corresponding maximum speed. After such a change the speed remains constant until the position of the next change is reached.

For each crossing that has not affirmed its safety, a danger position in front of the crossing is set and in the dynamic profile the corresponding position gets a maximum speed of zero. To take care of the braking characteristic, there is a fixed function *brakingDist* that gives for each speed the maximum distance the train needs to get to a safe halt. In Z this is a monotone function from speed to position (distance).

$$brakingDist : Speed \rightarrow Position$$

$$\forall\, s, s' : Speed \mid s \leq s' \bullet brakingDist(s) \leq brakingDist(s')$$

With this function it is possible to calculate the dynamic profile. It is specified as a function *calcProfile* taking a static profile and a set of danger positions as arguments and returning the dynamic speed profile as a function from position to maximum admissible speed.

$$calcProfile : StaticProfile \times \mathbb{P}\, Position \rightarrow (Position \rightarrow Speed)$$

To keep a record of the track elements to which we already sent the setting command, we need another set *notifiedElements*. As soon as we choose to notify an element we add its identifier to this set.

$$notifiedElems : \mathbb{P}\, Identifier$$

In Sect. 3 we have already introduced *TrackElement* and *TrackAtlas*. Here we give the full definition. A track element has a unique identifier *id* and can be either a crossing or a point. The associated danger position is stored in *pos*. The component *setpos* gives the position where the train should send the set command. The last field *dir* is only meaningful for points and specifies the direction in which it should be switched.

TrackElement

$id : Identifier$
$type : \{crossing, point\}$
$pos : Position$
$setpos : Position$
$dir : Direction$

The track atlas contains the static profile as well as a sequence of track elements.

TrackAtlas

$staticprof : StaticProfile$
$elems : \text{seq } TrackElement$

Using these types we now specify the *TrainController* as a CSP-OZ-DC class. The external interface of this class was already depicted in Fig. 2. Notice, however, that here the values communicated over these channels are specified using Z schema types. The local channels *clearDangerPosition*, *insecureTrackElement* and *calcMaxSpeed* are internal communication channels used to link the CSP and Z part. The event *setDangerPosition* is triggered by the DC part and received by the Z part.

TrainController

$\texttt{chan}\ getPos : [p? : Position];\ getSpeed : [s? : Speed]$
$\texttt{chan}\ setMaxSpeed : [s! : Speed]$
$\texttt{chan}\ setTrackElement : [id! : Identifier;\ dir! : Direction]$
$\texttt{chan}\ securedTrackElement : [id? : Identifier]$
$\texttt{local\_chan}\ insecureTrackElement : [id : Identifier]$
$\texttt{local\_chan}\ setDangerPosition, clearDangerPosition : [id : Identifier]$
$\texttt{local\_chan}\ calcMaxSpeed : [maxs : Speed]$

$$\texttt{main} \stackrel{c}{=} SuperviseSpeed \parallel SecuredNotifier \parallel SuperviseTrack$$

$$\begin{aligned} SuperviseSpeed \stackrel{c}{=}\ & getSpeed?speed \rightarrow getPos?pos \\ & \rightarrow calcMaxSpeed?maxs \\ & \rightarrow setMaxSpeed!maxs \rightarrow SuperviseSpeed \end{aligned}$$

$$\begin{aligned} SecuredNotifier \stackrel{c}{=}\ & securedTrackElement?id \\ & \rightarrow clearDangerPosition!id \rightarrow SecuredNotifier \end{aligned}$$

$$\begin{aligned} SuperviseTrack \stackrel{c}{=}\ & insecureTrackElement?id \\ & \rightarrow setTrackElement!id?dir \rightarrow SuperviseTrack \end{aligned}$$

The CSP process *SuperviseSpeed* already occurred in Sect. 3, but here we also show the communicated data. There are two other CSP processes running in parallel. One is *SecuredNotifier*, which handles the secured signal received by the radio controller and clears the associated danger position. The other is *SuperviseTrack*, which decides which track element should be secured and sends the setting command.

The next part of the *TrainController* class is its state space. It consists of several components, which were already introduced before. The variables *position* and *speed* always hold the last position and speed, that were queried from the Odometer. The dynamic Profile *dynProf* is special, because it is calculated from the other state variables and this is represented by

the formula in the state schema below the horizontal line. This formula is also called a *class invariant*.

$$
\begin{array}{|l}
\hline
trackatlas : TrackAtlas \\
dangerPositions : \mathbb{P}\, Position \\
dynProf : Position \rightarrow Speed \\
notifiedElems : \mathbb{P}\, Identifier \\
position : Position \\
speed : Speed \\
\hline
dynProf = calcProfile(trackatlas.staticprof, dangerPositions) \\
\hline
\end{array}
$$

Init

$$
\begin{array}{|l}
\hline
dangerPositions = \{elem : \operatorname{ran} trackatlas.elems \bullet elem.pos\} \\
\hline
\end{array}
$$

Everytime we query the speed or position from the odometer the corresponding variables in the state space are automatically updated. This is easily modelled here by communication schemas. Notice that $position'$ denotes the new value of the variable position, while its old value *position* is ignored.

com_*getPosition*

$$
\begin{array}{|l}
\hline
\Delta(position) \\
pos? : Position \\
\hline
position' = pos? \\
\hline
\end{array}
$$

com_*getSpeed*

$$
\begin{array}{|l}
\hline
\Delta(speed) \\
speed? : Speed \\
\hline
speed' = speed? \\
\hline
\end{array}
$$

The operations *setDangerPosition* and *clearDangerPosition* update the set of danger positions. The only difficulty here is that they both take an *id* as input, while *dangerPositions* is a set of positions. So this schema has to use the trackatlas to look up the track element record by its *id*. Notice that this schema only sets the new value of *dangerPositions*, but implicitly *dynProf* also gets a new value through the state invariant.

com_*setDangerPosition*

$$
\begin{array}{|l}
\hline
\Delta(dangerPositions, dynProf) \\
id? : Identifier \\
\hline
\exists\, elem : \operatorname{ran} trackatlas.elems \mid elem.id = id? \bullet \\
\quad dangerPositions' = dangerPositions \cup \{elem.pos\} \\
\hline
\end{array}
$$

com_*clearDangerPosition*

$$
\begin{array}{|l}
\hline
\Delta(dangerPositions, dynProf) \\
id? : Identifier \\
\hline
\exists\, elem : \operatorname{ran} trackatlas.elems \mid elem.id = id? \bullet \\
\quad dangerPositions' = dangerPositions \setminus \{elem.pos\} \\
\hline
\end{array}
$$

The operation *insecureTrackElement* selects a track element from the track atlas that should be immediately notified. The CSP part will then call *setTrackElement* for the chosen element,

but the OZ part has to deliver the direction argument *dir*, which the CSP part does not know. In the DC part of the class we make sure that these two operations get executed whenever there is a track element that should be notified.

com_*insecureTrackElement*

$\Delta(notifiedElements)$
$id! : identifier$

$\exists\, toNotify : \{elem : \mathrm{ran}\, trackatlas.elems \mid elem.id \notin notifiedElements$
$\qquad \wedge\ elem.setpos \leq position\} \bullet id! = toNotify.id$
$notifiedElements' = notifiedElements \cup id!$

com_*setTrackElement*

$id? : Identifier$
$dir! : direction$

$dir! = \mu\, elem : trackAtlas.elems \mid elem.id = id! \bullet elem.dir$

The operation *calcMaxSpeed* basically looks up the maximum speed for the current position in the dynamic speed profile. But to make the train controller safe it must look a short time into the future. This is done by *reactDistance*, which returns the maximum distance the train can pass within its reaction time. We do not give a definition for this function here.

com_*calcMaxSpeed*

$maxs! : Speed$

$\mathtt{let}\ endp == position + reactDistance(speed) \bullet$
$\quad maxs! = min\ dynProf(\!|\ [position, endp]\ |\!)$

$\lceil count(setMaxSpeed) = n \rceil \xrightarrow{1s} \lceil count(setMaxSpeed) > n \rceil$
$\lceil enabled(insecureTrackElement) \wedge count(insecureTrackElement) = n \rceil$
$\quad \xrightarrow{100ms} \lceil count(insecureTrackElement) > n \rceil$
$\lceil enabled(setTrackElement) \wedge count(setTrackElement) = n \rceil$
$\quad \xrightarrow{100ms} \lceil count(setTrackElement) > n \rceil$
$\lceil enabled(clearDangerPosition) \wedge count(clearDangerPosition) = n \rceil$
$\quad \xrightarrow{100ms} \lceil count(clearDangerPosition) > n \rceil$
$\lceil count(setTrackElement.id) = n \rceil;\ \lceil count(setTrackElement.id) > n$
$\quad \wedge\ count(setDangerPosition.id) = m \rceil \xrightarrow{300s} \lceil count(setDangerPosition.id) > m \rceil$

The first DC formula was already presented in Sect. 3. The second DC formula gives an example of the *enabled* predicate. The operation *insecureTrackElement* is enabled by the OZ part when its precondition is true which means that there exists a track element to be notified. Whenever it is enabled the event should be triggered after $100ms$. The third formula is similar, but here the *enabled* predicate is true whenever the CSP part is ready to execute the event, i.e. when *insecureTrackElement* was just triggered. The fourth is an analogous formula and makes sure that *clearDangerPosition* is executed shortly after the *securedTrackElement* was received.

The last DC formula specifies that the danger position for a secured track element should be set again five minutes after the *setTrackElement* event was issued. Normally the train should

have passed the corresponding track element by that time, otherwise the train must consider it as unsafe again.

The overall specification of the control system is given by the parallel composition of the classes corresponding to Fig. 2, of which we have exhibited the class *TrainController*:

$$Spec = TrainController \parallel RadioController \parallel Odometer \parallel SpeedController$$

5 Conclusion

One of our guidelines for combining specification techniques is *refinement compositionality*, i.e. refinement of the parts should imply refinement of the whole. For example, our specification of the case study makes use of intervals and functions over the real numbers, so it is not directly implementable. It is possible though, to refine the data part with a Z refinement to get only primitive types and sequences. For the untimed combination CSP-OZ Fischer [3] has proven data refinement in OZ indeed implies process refinement of the whole CSP-OZ class.

Related work. Closest to our approach is Real-Time Object-Z [12]. Classes in this combination look similar to ours but lack the CSP and DC part. Another related work is TCOZ, a combination of Timed CSP [1] with Object-Z [9, 10]. Obviously, DC is not involved in this combination. So the constructs of Timed CSP are used to specify time dependencies between communications.

Perspectives. Future work will explore transformations for restructuring and refining specifications of the combined language and tool support. We can of course reuse the support available for the individual techniques due to refinement compositionality. However, a topic of ongoing research are verification techniques for properties of the combination.

References

1. J. Davies. *Specification and Proof in Real-Time CSP.* Cambridge University Press, 1993.
2. C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, volume 2, pages 423–438. Chapman & Hall, 1997.
3. C. Fischer. *Combination and Implementation of Processes and Data: From CSP-OZ to Java.* PhD thesis, Bericht Nr. 2/2000, University of Oldenburg, April 2000.
4. J.-T. Gayen. Zugfahrten im FFB. Technical report, Institut für Eisenbahnwesen und Verkehrssicherung, TU Braunschweig, http://ivev8.ivev.bau.tu-bs.de/forschung/dfg-spp/zugfahrt/, 1999. in German.
5. M.R. Hansen and C. Zhou. Duration calculus: Logical foundations. *Formal Aspects of Computing*, 9:283–330, 1997.
6. C.A.R. Hoare. Communicating sequential processes. *CACM*, 21:666–677, 1978.
7. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
8. L. Janssen and E. Schnieder. Referenzfallstudie Bahnübergang – Referenzfallstudie im Bereich Verkehrsleittechnik des DFG-SPP Softwarespezifikation. Technical report, Institut für Regelungs- und Automatisierungstechnik, TU Braunschweig, http://www.ifra.ing.tu-bs.de/m33/spezi/, 1999. in German.
9. B.P. Mahony and J.S. Dong. Blending Object-Z and Timed CSP: an introduction to TCOZ. In K. Futatsugi, R. Kemmerer, and K. Torii, editors, *The 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104. IEEE Computer Society Press, 1998.
10. B.P. Mahony and J.S. Dong. Sensors and actuators in TCOZ. In J.M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 – Formal Methods*, volume 1709 of *LNCS*, pages 1166–1185. Springer, 1999.
11. G. Smith. *The Object-Z Specification Language.* Kluwer Academic Publisher, 2000.
12. G. Smith and I. Hayes. Towards real-time Object-Z. In K. Araki, A. Galloway, and K. Taguchi, editors, *Integrated Formal Methods (IFM 99)*, LNCS, pages 49–65. Springer, 1999.
13. J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall International Series in Computer Science, 2nd edition, 1992.
14. C. Zhou, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.