Combining Specification Techniques for Processes, Data and Time [⋆]

Jochen Hoenicke and Ernst-Rüdiger Olderog

Fachbereich Informatik, Universität Oldenburg
26111 Oldenburg, Germany
Fax: +49-441-798-2965

{hoenicke,olderog}@informatik.uni-oldenburg.de

Abstract. We present a new combination CSP-OZ-DC of three well researched formal techniques for the specification of processes, data and time: CSP [17], Object-Z [36], and Duration Calculus [40]. The emphasis is on a smooth integration of the underlying semantic models and its use for verifying properties of CSP-OZ-DC specifications by a combined application of the model-checkers FDR [29] for CSP and UPPAAL [1] for Timed Automata. This approach is applied to part of a case study on radio controlled railway crossings.

Keywords: CSP, Object-Z, Duration Calculus, transformational semantic, real-time processes, model-checking, FDR, UPPAAL

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 combine three well researched specification techniques: CSP, Object-Z and Duration Calculus.

Communicating Sequential Processes (CSP) were originally introduced by Hoare in [16] and developed further in [17]. The central concepts of CSP are synchronous communication via channels between different processes, parallel composition and hiding of internal communication. For CSP a rich mathematical theory comprising operational, denotational and algebraic semantics with consistency proofs has been developed [30]. Tool support comes through the FDR model-checker [29]. The name stands for Failure Divergence Refinement and refers to the standard semantic model of CSP, the failures divergence model, and its notion of process refinement.

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

Z was introduced in the early 80's in Oxford by Abrial as a set-theoretic and predicate language for the specification of data, state spaces and state transformations. The first systematic description of Z is [38]. Since then the language has been published extensively (e.g. [39]) and used in many case studies and industrial projects. In particular, Z schemas and the schema calculus enable a structured way of presenting large state spaces and their transformation. *Object-Z* is an object-oriented extension of Z [36]. It comprises the concepts of classes, inheritance and instantiation. Z and Object-Z come with the concept of data refinement. For Z there exist proof systems for establishing properties of specifications and refinements such as Z/EVES [31] or HOL-Z [19]. For Object-Z type checkers exist. Verification support is less developed except for an extension of HOL-Z [32].

Duration Calculus (DC for short) originated during the ProCoS (Provably Correct Systems) project [13] as a new logic and calculus for specifying the behaviour of real-time systems [40, 12]. It is based on the notion of an observable obs interpreted as a time dependent function $obs_{\mathcal{I}} : Time \rightarrow D$ for some data domain D. A real-time system is described by a set of such observables. This links up well to the mathematical basis found in classical dynamic systems theory [20] and enables extensions to cover hybrid systems. Duration Calculus was inspired by the work on interval temporal logic [23, 24] and thus specifies interval-based properties of observables. Its name stems from its ability to specify the *duration* of certain states in a given interval using the integral. By choosing the right set of observables, real-time systems can be described at various levels of abstraction [28, 26, 33, 4]. Verification support for the general DC is provided by [35, 14] using theorem provers, and for a more specialised application of DC by [6] using a translation into timed automata for model-checking with UPPAAL [1].

It is well known that a consistent combination of different specification techniques is difficult [18]. Very popular is currently UML, the Unified Modeling Language [2]. It collects all the widespread specification techniques for object-oriented systems in one language. There is even an extension UML-RT [34] intended to cover real-time systems. However, a closer examination shows that this extension is just able to deal with reactive systems. A problem with UML is the so far missing semantic basis for this huge language. It is still a topic of ongoing research to provide a semantics for suitable subsets of UML.

We believe that the best chances for a well founded combination are with specification techniques that are well researched individually. An example of a clear combination of two specification techniques is CSP-OZ [8, 9]. In this paper we extend CSP-OZ by the aspect of continuous real-time. This is done by combining it in a suitable way with DC. The resulting specification language we call CSP-OZ-DC. The paper is organised as follows. Section 2 introduces the main constructs of CSP-OZ-DC. Section 3 describes the semantics of the combination. Section 4 shows how this semantics can be utilized for a partially automatic verification of properties of CSP-OZ-DC specifications, and applies this approach to an example of a radio controlled railway crossing. Finally, we conclude with section 5.

2 The Combination CSP-OZ-DC

In this section we introduce the new combined formalism with some examples taken from a case study of radio controlled railway crossings[1], see Fig. 1. The main issue in this study is to remotely operate points and crossings via radio based communication while keeping the safety standard.

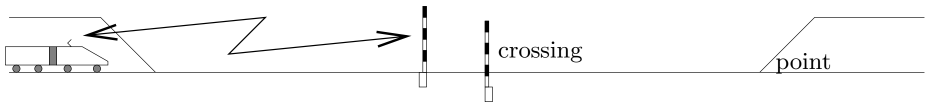

Fig. 1. Case study: radio controlled railway crossings

Fig. 2 surveys the controller architecture for a small part of this case study dealing with the safety of a level crossing. The diagram shows several components connected by communication channels. We discuss here the *cross controller* whose purpose is to secure the crossing upon requests issued by trains via the *radio controller*. We consider a multi-track level crossing where a request can be made for each track with the *set* communication. A request can be withdrawn at any time via the *clear* communication. The cross controller starts its securing cycle when at least one request was given. It continues through that cycle even if the request is withdrawn at a later time. When the crossing is secured it can communicate *secured* events to all requested tracks. When the train passes the crossing a *wheel counter* will notice that and trigger the *passed* communication. When no more requests are pending the crossing can be released.

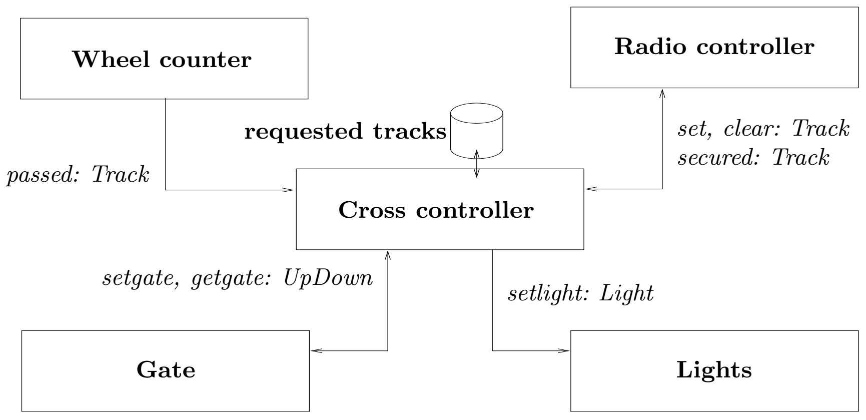

Fig. 2. Controller architecture

[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`).

To specify the cross controller several concepts must be handled, as described in the following. The cross controller communicates with other components, e. g. the *radio controller*. The order of these communications can be easily specified with CSP using mutually recursive process equations. The main equation is distinguished by the process identifier `main`. For the cross controller we have

$$
\begin{aligned}
\texttt{main} &\overset{c}{=} \mathit{assigned} \rightarrow \mathit{setlight}!\mathit{yellow} \rightarrow \mathit{setlight}!\mathit{red} \\
&\quad \rightarrow \mathit{setgate}!\mathit{down} \rightarrow \mathit{getgate.down} \rightarrow \mathit{Secure} \\
\mathit{Secure} &\overset{c}{=} \mathit{secured}?t \rightarrow \mathit{Secure} \\
&\quad \Box\ \mathit{free} \rightarrow \mathit{setgate}!\mathit{up} \rightarrow \mathit{setlight}!\mathit{off} \rightarrow \mathit{getgate.up} \\
&\qquad \rightarrow \mathit{wait} \rightarrow \texttt{main}
\end{aligned}
$$

The symbol $\overset{c}{=}$ is used instead of an ordinary equals symbol to distinguish between CSP process equations and Z equations. The communication *assigned* will be local and enabled only when a request for the crossing is pending (see Fig. 3). The crossing is secured in three steps: switch on first the yellow and then the red light, and afterwards close the gate. Next the process waits for a confirmation *getgate.down* form the *gate* that it is indeed closed. Then the crossing has reached a secure state and its further behaviour is modelled by the process *Secure*. Here it waits until all requests are either cleared or the corresponding train has passed the gate. Afterwards the gate opens again and the lights are switched off.

The cross controller should work for multi-track crossings. It therefore needs to remember the set of *requested tracks*. Handling such data and state information can be easily done with Object-Z (OZ). The state space is denoted by an unnamed schema:

$$\boxed{\; r : \mathbb{P}\ \mathit{Track} \;}$$

The initial state is described by an `Init` schema like this:

```
┌─ Init ──────────────────────
│ r = ∅
└─────────────────────────────
```

When a communication event like *set* is received from the radio controller, the Z state needs to be updated. In CSP-OZ it is very easy to link a data operation to a communication by writing a Z-schema with the name com_*set* specifying the operation associated with that communication event:

```
┌─ com_set ───────────────────
│ Δ(r)
│ t? : Track
├──────────
│ t? ∉ r ∧ r′ = r ∪ {t?}
└─────────────────────────────
```

The Δ in the first line of this schema declares that this operation may (only) change r. The next line declares a parameter t, decorated with ? to signify that

t is an input parameter. Notice that this naming convention of Z corresponds nicely with the naming conventions of CSP: the output of t along channel *set* synchronises with the input of t in the Z schema. In Z a state transformation is expressed by a predicate relating the state before and the state after the transformation. The second state is distinguished from the first one by decorating it with a prime. In this case the predicate states that the element $t?$ is added to the set r of requested tracks.

For embedded controllers another important aspect are real-time constraints. In our case study we want communication events to occur within certain time bounds. On the other hand, some events must not occur too early. This means we need timed progress and stability constraints. For specifying such real-time constrains, we use the Duration Calculus (DC). In DC *state assertions* P describe time dependent properties of observables $obs : Time \rightarrow D$. *Duration terms* describe interval-based real values. The name of the calculus stems from terms of the form $\int P$ measuring the *duration* of a state assertion P, i.e. the accumulated time that P holds in the considered interval. The simplest duration term is the symbol ℓ abbreviating $\int 1$ and thus denoting the *length* of the given interval. *Duration formulae* F, G describe interval-based properties. For example, $\lceil P \rceil$ abbreviates $\int P = \ell \wedge \ell > 0$ and thus specifies that P holds (almost) everywhere on a non-point interval. Sequential behaviour is modelled by the *chop* operator "$;$": the formula $F\,;\ G$ specifies that first F and then G holds. The formula $\Diamond F$ abbreviates *true*; F; *true* and thus expresses that on some subinterval F holds. The dual $\Box F$ abbreviates $\neg\Diamond\neg F$ and thus states that F holds on all subintervals.

A subset of the DC are the so-called *implementables* due to [27], which make use of the following idioms where $t \in Time$:

$$\begin{array}{llll} F \longrightarrow \lceil P \rceil & == & \Box\neg(F;\ \lceil \neg P \rceil) & \text{[followed-by]} \\ F \xrightarrow{t} \lceil P \rceil & == & (F \wedge \ell = t) \longrightarrow \lceil P \rceil & \text{[leads-to]} \\ F \xrightarrow{\leq t} \lceil P \rceil & == & (F \wedge \ell \leq t) \longrightarrow \lceil P \rceil & \text{[up-to]} \end{array}$$

Intuitively, $F \longrightarrow \lceil P \rceil$ expresses that whenever a pattern given by the formula F is observed, it will be "followed by" an interval where P holds. In the "leads-to" form the pattern is required to have a length t and in the "up-to" form it is bounded by a length "up to" t.

In this paper we also consider variants of the above formulae where we check an event ev by counting its number of occurrences:

$$F \xrightarrow[ev]{t} G == F \wedge \lceil ct(ev) = n \rceil \xrightarrow{t} G \wedge \lceil ct(ev) > n \rceil$$

For the cross controller we require for example the progress constraint

$$\lceil en(assigned) \rceil \xrightarrow[assigned]{1} \lceil \text{true} \rceil$$

stating that whenever the communication *assigned* is enabled it has to occur within 1 second. As an example for a stability constraint consider the DC formula

$$\lceil \neg\ en(setlight.red) \rceil;\ \lceil en(setlight.red) \rceil \xrightarrow{\leq 4} \lceil en(setlight.red) \rceil$$

stating that the *setlight.red* communication should stay enabled for at least 4 seconds before it can actually occur.

The basic building block in our combined formalism CSP-OZ-DC is a class. Its syntax is as in CSP-OZ [8, 9] except for the new DC part: see Fig. 3 for the complete specification of the *CrossController* class. First, the communication channels of the class are declared. Every channel has a type restricting the values that it can communicate. There are also local channels that are visible only inside the class and used for the interaction of the CSP, Z and DC parts. 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 specifying how the state changes when the corresponding communication event occurs. Finally, below a horizontal line the DC part is stated.

To describe architectures as in Fig. 2 classes can be combined into larger specifications by CSP operators like parallel composition, hiding and renaming.

3 Semantics

Each class of a CSP-OZ-DC specification denotes a time dependent process. In this section we describe how to define this process in a transformational way.

3.1 Semantics of the constituents

We begin by recalling the semantic domains of the constituent specification techniques. The standard semantics of untimed CSP is the $\mathcal{FD}$-semantics based on failures and divergence [30]. A *failure* is a pair (s, X) consisting of a finite sequence or *trace* $s \in \operatorname{seq} Comm$ over a set $Comm$ of communications and a so-called *refusal set* $X \in \mathbb{P}\, Comm$. Intuitively, a failure (s, X) describes that after engaging in the trace s the process can refuse to engage in any of the communications in X. Refusal sets allow us to make fine distinctions between different nondeterministic process behaviour; they are essential for obtaining a compositional definition of parallel composition in the CSP setting of synchronous communication when we want to observe deadlocks. Formally, we define the sets

$$\begin{aligned}&Traces == \operatorname{seq} Comm \quad \text{and} \quad Refusals == \mathbb{P}\, Comm,\\ &Failures == Traces \times Refusals.\end{aligned}$$

A *divergence* is a trace after which the process can engage in an infinite sequence of internal actions. The *$\mathcal{FD}$-semantics* of CSP is then given by two mappings

$$\mathcal{F} : \mathrm{CSP} \to \mathbb{P}\, Failures \quad \text{and} \quad \mathcal{D} : \mathrm{CSP} \to \mathbb{P}\, Traces.$$

For a CSP process P we write $\mathcal{FD}[\![P]\!] = (\mathcal{F}[\![P]\!]\ ,\ \mathcal{D}[\![P]\!])$. Certain well-formedness conditions relate the values of $\mathcal{F}$ and $\mathcal{D}$ (see [30], p.192). The $\mathcal{FD}$-semantics

$Track == 0..1$
$Color ::= off \mid yellow \mid red$
$UpDown ::= up \mid down$

$CrossController$

$\mathtt{chan}\ set, clear, passed, secured : [t? : Track]$
$\mathtt{chan}\ setlight : [color! : Color]$
$\mathtt{chan}\ setgate : [status! : UpDown]$
$\mathtt{chan}\ getgate : [status? : UpDown]$
$\mathtt{local\_chan}\ assigned, free, wait$

$$\begin{aligned}
\mathtt{main} &\overset{c}{=} assigned \rightarrow setlight!yellow \rightarrow setlight!red \\
&\quad \rightarrow setgate!down \rightarrow getgate.down \rightarrow Secure \\
Secure &\overset{c}{=} secured?t \rightarrow Secure \\
&\quad \Box\ free \rightarrow setgate!up \rightarrow setlight!off \rightarrow getgate.up \\
&\quad \rightarrow wait \rightarrow \mathtt{main}
\end{aligned}$$

$r : \mathbb{P}\ Track$

$\mathtt{Init}$
$r = \varnothing$

$\mathtt{com\_}secured$
$t? : Track$
$t? \in r$

$\mathtt{com\_}set$
$\Delta(r)$
$t? : Track$
$t? \notin r$
$r' = r \cup \{t?\}$

$\mathtt{com\_}clear$
$\Delta(r)$
$t? : Track$
$t? \in r$
$r' = r \setminus \{t?\}$

$\mathtt{com\_}passed$
$\Delta(r)$
$t? : Track$
$t? \in r$
$r' = r \setminus \{t?\}$

$\mathtt{com\_}assigned$
$r \neq \varnothing$

$\mathtt{com\_}free$
$r = \varnothing$

$$(\lceil en(assigned) \rceil \xrightarrow[assigned]{1} \lceil \mathrm{true} \rceil) \quad \wedge \quad (\lceil en(free) \rceil \xrightarrow[free]{1} \lceil \mathrm{true} \rceil)$$
$$\lceil en(setlight.yellow) \rceil \xrightarrow[setlight.yellow]{1} \lceil \mathrm{true} \rceil$$
$$\lceil \neg\ en(setlight.red) \rceil ;\ \lceil en(setlight.red) \rceil \xrightarrow{\leq 4} \lceil en(setlight.red) \rceil$$
$$\lceil en(setlight.red) \rceil \xrightarrow[setlight.red]{5} \lceil \mathrm{true} \rceil$$
$$\lceil \neg\ en(setlight.off) \rceil ;\ \lceil en(setlight.off) \rceil \xrightarrow{\leq 1} \lceil en(setlight.off) \rceil$$
$$\lceil en(setlight.off) \rceil \xrightarrow[setlight.off]{2} \lceil \mathrm{true} \rceil$$
$$\lceil \neg\ en(setgate.down) \rceil ;\ \lceil en(setgate.down) \rceil \xrightarrow{\leq 2} \lceil en(getgate.down) \rceil$$
$$\lceil en(setgate.down) \rceil \xrightarrow[setgate.down]{3} \lceil \mathrm{true} \rceil$$
$$\lceil en(setgate.up) \rceil \xrightarrow[setgate.up]{1} \lceil \mathrm{true} \rceil$$
$$\lceil \neg\ en(wait) \rceil ;\ \lceil en(wait) \rceil \xrightarrow{\leq 30} \lceil en(wait) \rceil$$
$$\lceil en(wait) \rceil \xrightarrow[wait]{31} \lceil \mathrm{true} \rceil$$

Fig. 3. A multi-track level crossing

induces a notion of *process refinement* denoted by $\sqsubseteq_{\mathcal{FD}}$. For CSP processes P and Q this relation is defined as follows:

$$P \sqsubseteq_{\mathcal{FD}} Q \text{ iff } \mathcal{F}[\![P]\!] \supseteq \mathcal{F}[\![Q]\!] \text{ and } \mathcal{D}[\![P]\!] \supseteq \mathcal{D}[\![Q]\!]$$

Intuitively, $P \sqsubseteq_{\mathcal{FD}} Q$ means that Q refines P, i.e. Q is more deterministic and more defined than P.

Instead of the negative information of refusal sets one can also use positive information about the future process behaviour in terms of so-called *acceptance sets*. For a trace s an acceptance set $A \in \mathbb{P}\,Comm$ describes a set of communications that are possible after s. The set of all initial communications after s is the largest acceptance set after s. Acceptance sets are due to Hennessy and De Nicola [15, 25] who developed an approach to testing of processes that resulted in a process model equivalent to the failures divergence model but with acceptance sets instead of refusal sets. Acceptance sets satisfy certain closure properties (see [15], p.77). For example, they are closed under union. Formally, let

$$Acceptances == \mathbb{P}\,Comm$$

and $\mathcal{A}$ be the process semantics

$$\mathcal{A} : \mathrm{CSP} \rightarrow \mathbb{P}(Traces \times Acceptances)$$

based on acceptance sets instead of refusal sets. $\mathcal{AD}$-semantics is the process semantics based on $\mathcal{A}$ and $\mathcal{D}$. We write $\mathcal{AD}[\![P]\!] = (\mathcal{A}[\![P]\!]\ ,\ \mathcal{D}[\![P]\!])$ for a CSP process P. Then the following proposition on process refinement can be proved:

Proposition 1. *$P \sqsubseteq_{\mathcal{FD}} Q$ iff $\mathcal{A}[\![P]\!] \supseteq \mathcal{A}[\![Q]\!]$ and $\mathcal{D}[\![P]\!] \supseteq \mathcal{D}[\![Q]\!]$*

Thus we do not lose any process information by taking acceptance sets instead of refusal sets. Since for our approach to verification will be based on acceptance sets, we shall represent here the semantics of untimed CSP on $\mathcal{A}$ and $\mathcal{D}$.

Object-Z (OZ) describes state spaces as collections of typed variables, say x of type D_x, and their possible transformation with the help of action predicates $A(x, x')$, for example $x' \geq x + 1$, where the decorated version x' represents the value of x after the transformation. The language comes with the usual notion of *data refinement* [39].

Duration Calculus (DC) specifies properties of observables *obs* interpreted as *finitely varying* functions of the form $obs_{\mathcal{I}} : Time \rightarrow D$ for a continuous time domain *Time* and a data domain D. Finitely varying means that $obs_{\mathcal{I}}$ can assume only finitely many different values within a finite time interval [12]. When modelling real-time systems in DC, refinement boils down to logical implication.

3.2 Untimed semantics of CSP-OZ classes

The untimed semantics of the combination CSP-OZ is defined in [8, 9]. The idea is that each CSP-OZ class denotes a process in the semantic model of CSP. This is achieved by transforming the Z part of such a class into a CSP process that runs in parallel and communicates with the CSP part of the class.

Consider a CSP-OZ class

U

I	[interface]
L	[local channels]
P	[CSP part]
Z	[Z part]

also written horizontally as $U \mathrel{\widehat{=}} \texttt{spec}\ I\ L\ P\ Z\ \texttt{end}$ with a Z part of the form

Z

st	[state space]
$Init(st)$	[initial condition]
$...com\_c(st, in?, out!, st')...$	
	[one communication schema for each c in I or L]

where the notation $com\_c(st, in?, out!, st')$ indicates that this communication schema relates the state st to the successor state st' and has input parameters $in?$ and output parameters $out!$.

The Z part of the class is transformed into a CSP process $ZMain$ defined by the following system of (parametrised) recursive equations for $ZPart$ using (indexed) CSP operators for internal choice ($\sqcap$) and alternative composition ($\Box$):

$$\begin{aligned} ZMain &= \bigsqcap_{st \text{ with } Init(st)} ZPart(st) \\ ZPart(st) &= \Box_{\substack{c\ in\ I\ or\ L;\ \ in? : Inputs(c) \\ \text{with } \exists\, out! : Outputs(c);\ st' \bullet com\_c(st, in?, out!, st')}} \ \bigsqcap_{\substack{out! : Outputs(c);\ st' \\ \text{with } com\_c(st, in?, out!, st')}} c.in?.out! \to ZPart(st') \end{aligned}$$

Informally, $ZMain$ can start in any state st satisfying $Init(st)$. Then $ZPart(st)$ is ready for every communication event $c.in?.out!$ along a channel c in I or L where for the input values $in?$ the communication schema $com\_c(st, in?, out!, st')$ is satisfiable for some output values $out!$ and successor state st'. For given input values $in?$ any such $out!$ and st' can be internally chosen to yield $c.in?.out!$ and the next recursive call $ZPart(st')$. Thus input and output along channels c are modelled by a subtle interplay of the CSP alternative and choice.

$ZMain$ runs in parallel with the explicit CSP process P of the class:

$$proc_U = P\ [|\ Events(I \cup L)\ |]\ ZMain$$

Here the parallel composition synchronises on all events in I and L. In [8, 9] the semantics of the class U is then defined by

$$\mathcal{FD}[\![U]\!] = \mathcal{FD}[\![proc_U \setminus Events(L)]\!]$$

where all events along local channels L are hidden. Hiding in untimed CSP makes communications occur autonomously without delay. Thus hiding can cause nondeterminism and divergence.

By the above process semantics of CSP-OZ, the refinement notion $\sqsubseteq_{\mathcal{FD}}$ is immediately available for CSP-OZ. One of our guidelines for combining specification techniques is *refinement compositionality*, i.e. refinement of the parts should imply refinement of the whole. For CSP-OZ this is shown in [9]:

Theorem 1. *Process refinement $P_1 \sqsubseteq_{\mathcal{FD}} P_2$ implies refinement in CSP-OZ:*

$$\texttt{spec}\ I\ L\ P_1\ Z\ \texttt{end} \quad \sqsubseteq_{\mathcal{FD}} \quad \texttt{spec}\ I\ L\ P_2\ Z\ \texttt{end}$$

Data refinement $Z_1 \sqsubseteq_R Z_2$ for a refinement relation R implies refinement in CSP-OZ:

$$\texttt{spec}\ I\ L\ P\ Z_1\ \texttt{end} \quad \sqsubseteq_{\mathcal{FD}} \quad \texttt{spec}\ I\ L\ P\ Z_2\ \texttt{end}$$

3.3 Timed semantics of CSP-OZ-DC classes

The semantic idea of the combination CSP-OZ-DC is that each class denotes a timed process. To this end, we lift the semantics of CSP and OZ onto the level of time dependent observables. In the timed setting the behaviour of internal actions has to be studied carefully. We distinguish between internal τ actions inherited form the untimed CSP setting and internal *wait* actions induced by hiding communications with a certain timing behaviour. Whereas internal τ actions do not take time and can thus be eliminated in accordance with the $\mathcal{FD}$-semantics, possibly inducing nondeterminism or divergence, internal *wait* actions let time pass before the next visible communication can occur. Whereas an infinite sequence of τ actions is equivalent to divergence, an infinite sequence of wait actions is equivalent to deadlock.

For simplicity, we do not consider the case where the untimed part diverges. Thus the semantics of CSP-OZ-DC will associate with each specification of the combined language a timed process consisting of a set of time dependent traces and time dependent acceptances:

$$\mathcal{A}_{Time} : \text{CSP-OZ-DC} \rightarrow \mathbb{P}((Time \rightarrow Traces) \times (Time \rightarrow Acceptances))$$

For a CSP-OZ-DC specification S its semantics $\mathcal{A}_{Time}[\![S]\!]$ will be described by a DC formula in the observables tr and Acc interpreted as finitely varying functions

$$tr_{\mathcal{I}} : Time \rightarrow Traces \quad \text{and} \quad Acc_{\mathcal{I}} : Time \rightarrow Acceptances.$$

This DC formula denotes the set of all interpretations of tr and Acc that make the formula true; thus it will be identified with $\mathcal{A}_{Time}[\![S]\!]$.

We explain the details first for a CSP-OZ-DC class C, which augments the untimed CSP-OZ class U by an additional timing part T expressed in DC:

$$\begin{array}{|l r}
\multicolumn{2}{l}{C} \\ \hline
U & \text{[untimed components]} \\ \hline
T & \text{[DC part]} \\ \hline
\end{array}$$

We shall also expand C horizontally into

$$C \mathrel{\widehat{=}} \texttt{spec}\ I\ L\ P\ Z\ T\ \texttt{end}.$$

The semantics of C is obtained by taking the CSP process $proc_U$ defined for the CSP-OZ class U but interpreting it in the setting of the time dependent observables tr and Acc, and then conjoining it with the time dependent restrictions expressed in the DC part T. Since $proc_U$ is still an untimed process, its semantics in terms of tr and Acc will allow any time dependent behaviour. More precisely, given the untimed acceptance semantics of $proc_U$ assumed to be divergence free,

$$\mathcal{A}[\![proc_U]\!] : \mathbb{P}(\mathit{Traces} \times \mathit{Acceptances}) \quad \text{with} \quad \mathcal{D}[\![proc_U]\!] = \varnothing,$$

we define its timed semantics as the DC formula

$$\mathcal{A}_{\mathit{Time}}[\![proc_U]\!] \Leftrightarrow \mathcal{F}_U \wedge \mathcal{F}_1 \wedge \mathcal{F}_2 \wedge \mathcal{F}_3$$

in the observables tr and Acc with subformulae $\mathcal{F}_U, \mathcal{F}_1 - \mathcal{F}_3$ given as follows:

$$\mathcal{F}_U : \Box \lceil (tr, Acc) \in \mathcal{A}[\![proc_U]\!] \rceil$$

requires that the values of the observables tr and Acc are taken from the untimed acceptance semantics of $proc_U$.

$$\mathcal{F}_1 : \lceil\rceil \vee \lceil tr = \langle\rangle \rceil;\ \mathit{true}$$

requires that initially the trace is empty.

$$\mathcal{F}_2 : \Box \forall h, h' \bullet (h \neq h' \wedge \lceil tr = h \rceil;\ \lceil tr = h' \rceil) \Rightarrow \exists c, v \bullet h' = h \frown \langle c.v \rangle$$

requires that the trace can only grow and that one communication event occurs at a time. The modality $\Box$ quantifies over all subintervals of a given time interval, and ; is the chop operator of interval temporal logic used in DC [40, 12]. The subformula $\lceil tr = h \rceil;\ \lceil tr = h' \rceil$ holds in any time interval where on a first non-point interval tr assumes the value h and on a second non-point interval the value h'. By $\mathcal{F}_2$, h' can differ from h only by one communication event. Together with the restriction to finite variability, we thus require that only finitely many communication events occur within a finite time interval and that one communication event occurs at a time. Consequently in our semantics a non-zero time passes between successive events. Finally,

$$\begin{aligned}\mathcal{F}_3 : \Box \forall h, c, v \bullet (\lceil tr = h \rceil;\ \lceil tr = h \frown \langle c.v \rangle \rceil \Rightarrow \\ (\lceil tr = h \rceil \wedge (\mathit{true};\ \lceil c.v \in Acc \rceil));\ \lceil tr = h \frown \langle c.v \rangle \rceil)\end{aligned}$$

requires that every communication $c.v$ can occur only with prior appearance in an acceptance set.

Only the DC part T can actually restrict this behaviour in a time dependent manner. To this end, T has limited access to the observables tr and Acc via the expressions $ct(X)$ and $en(X)$ where X is a set of communication events. By definition,

$$\begin{array}{|l} ct : \mathbb{P}\, Comm \rightarrow \mathbb{N} \\ \hline \forall X : \mathbb{P}\, Comm \bullet ct(X) = \#(tr \rhd X) \end{array}$$

Thus $ct(X)$ counts the number of occurrences of events from X in the trace tr. Next

$$\begin{array}{|l} en : \mathbb{P}\, Comm \rightarrow \mathbb{B} \\ \hline \forall X : \mathbb{P}\, Comm \bullet en(X) \Leftrightarrow X \subseteq Acc \end{array}$$

Thus $en(X)$ records whether all events from X can be accepted next. It is for this definition of enabledness that acceptance sets are easier to use than refusals. This motivated our choice of the semantic representation. For a single communication event $c.v$ we write $ct(c.v)$ and $en(c.v)$ instead of $ct(\{c.v\})$ and $en(\{c.v\})$. Using these expressions we can specify timing constraints for the visible communications.

Altogether the semantics of the timed class C is given by the formula

$$\mathcal{A}_{Time}[\![C]\!] \Leftrightarrow \texttt{hide}\ L \bullet (\mathcal{F}_U \wedge \mathcal{F}_1 \wedge \mathcal{F}_2 \wedge \mathcal{F}_3 \wedge T)$$

where all communications along the local channels in L are hidden. For a DC formula F in the observables tr and Acc we define

$$\texttt{hide}\ L \bullet F \Leftrightarrow \exists\, tr_0, Acc_0 \bullet ((\lceil\rceil \vee \lceil tr = squash(tr_0 \unrhd L) \wedge Acc = Acc_0 \setminus L\rceil) \wedge F[tr_0/tr, Acc_0/Acc])$$

Thus $\texttt{hide}\ L \bullet F$ is a DC formula in the observables tr and Acc, and their values are linked via the substitution $F[tr_0/tr, Acc_0/Acc]$ to the original values of these observables in F. It describes the timed semantics of the CSP hiding operator.

3.4 Timed Semantics of System Specifications

System specifications S are obtained by combining class specifications with the CSP operators for parallel composition, hiding and renaming. Thus a typical specification could be of the form

$$S = (C_1[R_1] \parallel C_2[R_2]) \setminus L.$$

The parallel composition $\parallel$ can be modelled by an *alphabetised parallel* ${}_A\|_B$ where A and B are the sets of interface events of $C_1[R_1]$ and $C_2[R_2]$. For DC formulae F_1 and F_2 in the observables tr and Acc the semantics of the parallel composition can be expressed, similarly to [33], by the following DC formula:

$$\begin{aligned}
F_1 \;{}_A\|_B\; F_2 \Leftrightarrow \exists\, tr_1, tr_2, Acc_0, Acc_1, Acc_2 \bullet \\
&((\lceil\rceil \vee \lceil tr \in \mathrm{seq}(A \cup B) \wedge tr \upharpoonright A = tr_1 \wedge tr \upharpoonright B = tr_2 \wedge \\
&\quad Acc_0 \cap (A \cup B) = \varnothing \wedge \\
&\quad Acc = (Acc_1 \cap Acc_2 \cap A \cap B) \cup \\
&\qquad (Acc_1 \setminus B) \cup (Acc_2 \setminus A)\rceil) \wedge \\
&F_1[tr_1/tr, Acc_1/Acc] \wedge F_2[tr_2/tr, Acc_2/Acc])
\end{aligned}$$

Hiding, denoted by $\backslash L$, is used to make the communication events in L internal. Semantically, hiding is defined using the operator `hide` $L \bullet F$ introduced above. Renaming, denoted by $[R]$ for a relation R between events, is used to rename communication events. The semantic definition is straightforward.

The *refinement* relation between classes or between specifications of the same interface is modelled by (reverse) logical implication in the semantic domain: a class C_2 *refines* a class C_1, abbreviated by

$$C_1 \sqsubseteq C_2, \text{ if } \mathcal{A}_{Time}[\![C_2]\!] \Rightarrow \mathcal{A}_{Time}[\![C_1]\!]$$

holds. We show that *refinement compositionality* holds also for CSP-OZ-DC.

Theorem 2. *(a) Process refinement $P_1 \sqsubseteq_{\mathcal{FD}} P_2$ implies refinement in CSP-OZ-DC:* `spec` $I\ L\ P_1\ Z\ T$ `end` $\sqsubseteq$ `spec` $I\ L\ P_2\ Z\ T$ `end`

(b) Data refinement $Z_1 \sqsubseteq_R Z_2$ for a refinement relation R implies refinement in CSP-OZ-DC: `spec` $I\ L\ P\ Z_1\ T$ `end` $\sqsubseteq$ `spec` $I\ L\ P\ Z_2\ T$ `end`

(c) Time constraint refinement $T_2 \Rightarrow T_1$ implies refinement in CSP-OZ-DC: `spec` $I\ L\ P\ Z\ T_1$ `end` $\sqsubseteq$ `spec` $I\ L\ P\ Z\ T_2$ `end`

Proof. Statements (a) and (b) are immediate consequences of Theorem 1 and the monotonicity of $\mathcal{F}_U$ w.r.t. refinements of the untimed class U. Statement (c) follows from the conjunctive form of $\mathcal{A}_{Time}[\![C]\!]$. □

By this theorem, it is possible to reuse verification techniques for the components of a CSP-OZ-DC specification to prove refinement results for the whole specification. However, when the desired property of the whole specification depends on the semantic interplay of the components, more sophisticated verification techniques are needed. In the following we develop one such a technique.

4 Verification

We exploit the above style of semantics for a partially automatic verification of properties of CSP-OZ-DC specifications that satisfy the following restrictions: the CSP part represents a finite-state process, the OZ data types are finite, and the DC part obeys certain patterns described below. Then the idea is as follows. Given a class $C \mathrel{\widehat=}$ `spec` $I\ L\ P\ Z\ T$ `end` we proceed in four steps:

(1) Represent the untimed process $U = \texttt{spec}\ I\ L\ P\ Z\ \texttt{end}$ in FDR-CSP, the input language of the FDR model-checker [29, 11] for CSP.
(2) Use the FDR model-checker to output a transition system TS_U for U with acceptance sets.
(3) Transform this transition system TS_U into a timed automaton $\mathcal{A}_C$ representing all the timing restrictions of the DC semantics.
(4) Verify properties of the class C by applying the model-checker UPPAAL [1] to $\mathcal{A}_C$.

Step(1) follows an approach of [10]. While steps (1) and (4) currently require user interaction, steps (2) and (3) proceed fully automatic.

The DC patterns for timing restrictions that can be handled in step (3) are new variants of the DC implementables [27] introduced next. An event set X appearing as a subscript of the chop operator or the followed-by operator (cf. section 2) indicates that an event from X happens at the corresponding chop point. Formally:

$$F \underset{X}{;} G == (F \wedge \lceil ct(X) = n \rceil);\ (G \wedge \lceil ct(X) > n \rceil)$$
$$F \xrightarrow[X]{t} G == (F \wedge \lceil ct(X) = n \rceil) \xrightarrow{t} (G \wedge \lceil ct(X) > n \rceil)$$

The following formula states that while a stability constraint applies, events from the set X *must not* happen:

$$F \xrightarrow[/X]{\le t} G == (F \wedge \lceil ct(X) = n \rceil) \xrightarrow{\le t} (G \wedge \lceil ct(X) = n \rceil)$$

A tool for step (3) developed by C. Ohler supports the following DC patterns:

$$\lceil P \rceil \underset{X}{;} \lceil Q \rceil \xrightarrow[Y]{t} \lceil R \rceil \qquad \text{[chop-leads-to]}$$
$$\lceil P \rceil \underset{X}{;} \lceil Q \rceil \xrightarrow[/Y]{\le t} \lceil R \rceil \qquad \text{[chop-up-to]}$$
$$\lceil Q \rceil \xrightarrow[X]{t} \lceil R \rceil \qquad \text{[leads-to]}$$
$$\lceil Q \rceil \xrightarrow[/Y]{\le t} \lceil R \rceil \qquad \text{[up-to]}$$

Here $t \in \mathit{Time}$ and P, Q, R are state assertions. The event sets X, Y are optional and can be omitted. Also the upper bound t in the *up-to* formulas can be omitted.

The tool implements an algorithm that applies given DC formulae of the above patterns one after the other to transform the transition system produced in step (2) into a timed automaton. As an example we show in Fig. 4 the pseudo code for the leads-to pattern. In step `1` the algorithm adds a new clock to measure the time the transition system stayed in a Q-state without executing an event from X. While being in a Q-state this clock must not grow beyond t because otherwise the DC formula would be violated. Therefore we add in step `2` a corresponding state invariant to all Q-states. The clock needs to be reset when a Q-state is entered from outside (step `3.a`) or when an event from X occurs

```
Pattern: $\lceil Q \rceil \xrightarrow[X]{t} \lceil R \rceil$
1. Introduce new clock $c$
2. To all states $s \in Q$ add invariant $c \leq t$
3. For each transition $tr : s \xrightarrow{ev} s'$ do:
   a. if $s \notin Q,\ s' \in Q$
        add reset $c := 0$ to $tr$
   b. if $s \in Q,\ s' \in Q,\ ev \in X$
        add reset $c := 0$ to $tr$
   c. if $s \in Q,\ (s' \notin R \vee ev \notin X)$
        add guard $c < t$ to $tr$
```

Fig. 4. Algorithm for the leads-to pattern

and the control stays in Q (step `3.b`). All outgoing events that do not lead into an R-state or do not communicate an event from X must happen before time t has elapsed. Therefore a corresponding guard is added in step `3.c`.

Besides enriching the transition system generated by FDR our tool also adds a timed *supervisor* automaton running in parallel. The supervisor serves two purposes: first, it ensures that – in agreement with the DC semantics defined in section 3.3 – a non zero time passes between successive events, and second, it hides the local channels that should not be visible to other processes.

4.1 Case Study

We now apply the above verification procedure to the case study introduced in section 2. The result of the manual step (1) is given in Fig. 5. It shows the FDR-CSP specification using the input language of the FDR model-checker and representing the CSP and Z part of the combined cross controller specification in Fig. 3. The representation of the constant declaration, the channel declaration and the CSP part is straightforward. Only at a few locations the syntax needs to be adapted to FDR. The transformation of the Z part into a CSP process *ZPart* was described in section 3.2: *ZPart* takes the complete Z state, here r, as a parameter. It offers an external choice over all communications that had corresponding Z schemas in the original specification. For readability we applied some simplifications to this part. The CSP and Z part are put in parallel and synchronize over their common alphabet.

The next two steps are performed automatically by our tool. In step (2) it uses the FDR model-checker to create a compact finite transition system from the specification of Fig. 5. The result is sketched in Fig. 6. The graph contains 40 states and 160 transitions. To make it more readable not all transitions are labelled. The whole graph is shaped as a cycle that corresponds to the cyclic behaviour of securing the crossing and releasing it.

Every CSP state is expanded into four substates corresponding to the four possible values of the variable r in the Z part. A closer view of the top left part of the graph is given in Fig. 7. The *passed* transitions are still omitted; they are in the same places as the *clear* transitions. Note that the initial state (the top most

```
-- Constants
Track  = {0..1}
datatype Color  = off | yellow | red
datatype UpDown = up | down

-- Channels
channel set, clear, secured, passed : Track
channel setlight                    : Color
channel setgate, getgate            : UpDown
channel assigned, free, wait

-- Class CrossController
CrossController =
  let
    -- CSP part
    main  = assigned -> setlight!yellow -> setlight!red
            -> setgate!down -> getgate!down -> Secure
    Secure = secured?t -> Secure
          [] free -> setgate!up -> setlight!off -> getgate!up
             -> wait -> main

    -- Z part
    ZPart(r) =
         ([] t : diff(Track, r) @ set.t -> ZPart(union(r, {t})))
      [] ([] t : r @ clear.t -> ZPart(diff(r, {t})))
      [] ([] t : r @ passed.t -> ZPart(diff(r, {t})))
      [] ([] t : r @ secured.t -> ZPart(r))
      [] r != {} & assigned -> ZPart(r)
      [] r == {} & free -> ZPart(r)

  within  --  Put CSP and Z part in parallel
    main [| {| secured, assigned, free |} |] ZPart({})
```

Fig. 5. FDR-CSP specification for the cross controller

in the detailed graph) has no outgoing *assigned* transition. This is because the Z part blocks this transition when r is empty. Since all transitions are deterministic in our case study, every state has only one acceptance set, which contains all outgoing events.

In step (3) the DC formulae are applied one after the other. For each DC formula a new clock is introduced and new guards and resets are added to the transition. The tool starts with the first DC formula

$$\lceil en(assigned) \rceil \xrightarrow[assigned]{1} \lceil \text{true} \rceil \ .$$

This is an instance of the leads-to pattern. The states where $en(assigned)$ holds are only the three states in Fig. 7 that have an outgoing *assigned* transition.

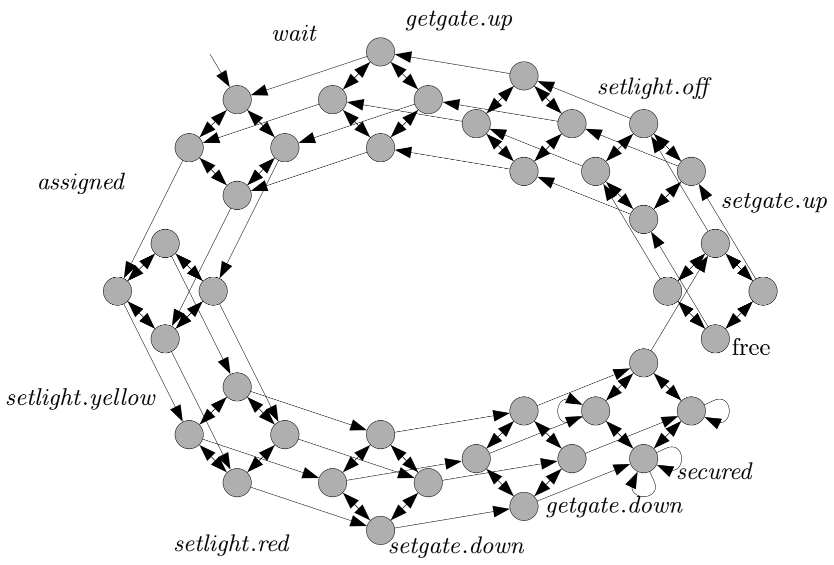

Fig. 6. Transition system generated by FDR

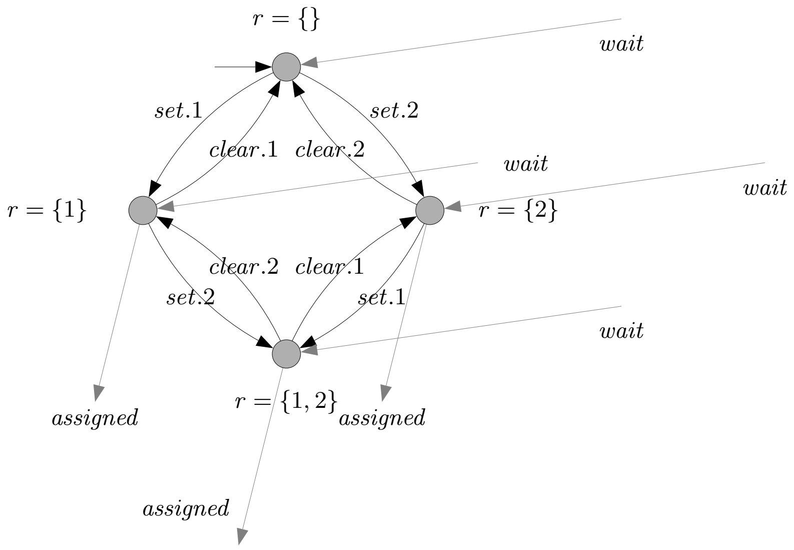

Fig. 7. Detailed view of the transition system

Thus only the transitions depicted in the detailed view are changed. Applying the algorithm in Fig. 4 yields the timed automaton depicted in Fig. 8.

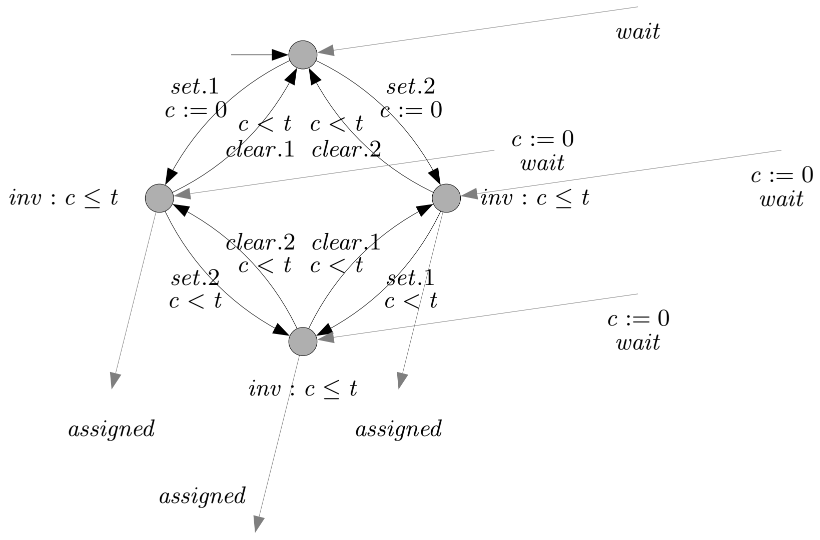

Fig. 8. Timed automaton resulting from the first DC formula

After applying all DC formulae step (3) terminates with a timed automaton representing the complete CSP-OZ-DC class. Altogether the algorithm adds 12 clocks and a lot of resets and guards. So the complete graph is not easily readable but we can verify that certain properties hold with the model-checker UPPAAL.

4.2 Model-Checking

We consider the following real-time property: Whenever a train requests a track and it does not clear the request or passes the crossing it can get a *secured* communication within a certain time t. We wish to determine the exact value of t experimentally.

To verify this property we build a test automaton. This is a small timed automaton that communicates with the cross controller over some of the channels we defined in our CSP-OZ-DC specification. As the identity of the track does not matter, we assume that our test automaton deals with track 0. Therefore we link it over the *set*.0, *clear*.0, *passed*.0 and *secured*.0 communication events. We instruct the tool to hide all other communications, so that they can occur at any time.

The test automaton is given in Fig. 9. In its initial state *idle* the automaton is able to communicate any event. When communicating *set*.0 it resets a clock *c_waiting* and switches to the *busy* state. In this state it waits until it can communicate *secured*.0. Then it returns to the *idle* state. This automaton is put in parallel with the cross controller.

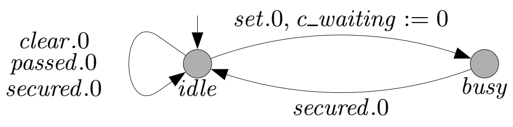

Fig. 9. Test automaton

We ask UPPAAL whether for all reachable states the test automaton is in the *idle* state or its clock is smaller than t, where t is an integer constant. This query can be expressed in the UPPAAL syntax as follows:

$$\mathit{Property}_t \;\Leftrightarrow\; \mathtt{A}\;\Box\;\mathit{TestAutomaton.idle}\;\;\mathtt{or}\;\;\mathit{c\_waiting} < t$$

Applying this to the cross controller, UPPAAL quickly generates a counter example showing that the property does not hold. The trace contains the CSP events so it is possible to compare it to the CSP-OZ-DC specification. It turns out that the property is violated because we do not have enough progress conditions.

We need the assumptions that the gate will actually close or open within a certain time bound, say 15 seconds, and that the train will actually receive the *secured* communication within a certain time. Note that these are assumptions about the *environment* of the controller. However, for simplicity we add them to our specification. The formulae we add are:

$$\lceil en(getgate.down)\rceil \;\xrightarrow[getgate.down]{15}\; \lceil \mathrm{true}\rceil$$
$$\lceil en(getgate.up)\rceil \;\xrightarrow[getgate.up]{15}\; \lceil \mathrm{true}\rceil$$
$$\lceil en(secured.0)\rceil \;\xrightarrow[secured.0]{1}\; \lceil \mathrm{true}\rceil$$

Now we can run UPPAAL again to check $\mathit{Property}_t$ for different values of t. If we choose a value for t smaller than 75 UPPAAL finds a counter example within a few seconds. For $t = 75$ the property is satisfied.

4.3 Experimental Results

The table in Fig. 10 gives some timings for these steps. The times were measured on an UltraSPARC-II with 296 MHz. Steps (2) and (3) of the verification procedure are quite fast. For the cross controller step (2) needs less than a second, and step (3) needs 1.8 seconds to apply the twelve DC formulae from the original specification plus the three formulae from the environment to the transition system generated by step (2). Model-checking in step (4) is most time consuming when no counter example exists, this is the reason for the difference in the last two columns. For $t = 74$ there is a counter example, but for $t = 75$ UPPAAL has to investigate the complete state space.

Consider now a larger system with more tracks. Adding a track doubles the Z state space and thus the resulting automaton states, and it yields almost three times as many transitions. As shown in Fig. 10, the steps take more time by a factor between two and three.

Number of tracks	step (2)	step (3)	step (4) with $t = 74$	step (4) with $t = 75$
2	0.3	1.8	16.4	1662
3	0.4	3.6	43.5	4379
4	0.6	10.4	101.5	10979

Fig. 10. Verification time (in seconds).

5 Conclusion

Related work. Closest to our way of combining specification techniques is Real-Time Object-Z [37]. Classes in this combination look similar to ours but lack the CSP and DC part. As we have seen in the case study, the CSP part is convenient for specifying sequencing constraints on the communications events. Furthermore, CSP offers parallel composition and hiding that can well be used for the structuring of larger CSP-OZ-DC specifications. In Real-Time Object-Z the timing properties are specified in an interval-based set-theoretic notation [7]. We also use an interval-based approach but in terms of the well researched Duration Calculus [40, 12]. The semantics of Real-Time Object-Z is given in terms of time dependent traces [37] whereas we consider also time dependent acceptances due to the presence of CSP.

Another related work is TCOZ, a combination of Timed CSP [3] with Object-Z [21, 22]. Obviously, DC is not involved in this combination. So the constructs of Timed CSP are used to specify time dependencies between communications. Besides this difference, the semantic integration of CSP with Object-Z differs from ours. In TCOZ an Object-Z operation schema denotes a process whereas in CSP-OZ-DC it specifies the effect of a communication event on the state.

Verification. We have shown how to exploit the transformational semantics of CSP-OZ-DC for a partially automatic verification of properties of combined specifications. To this end, we have developed a novel, systematic transformation of CSP-OZ-DC classes into timed automata that can be model-checked by the UPPAAL tool. This poses the question whether the timed automata semantics produced by the algorithm described in section 4 is *equivalent* to the DC semantics of section 3. A proof of such an equivalence is left for future work. We notice, however, that similar equivalence proofs between timed automata and DC semantics are given in [5].

Perspectives. Automatic verification works only for finite data types in the Z part and certain patterns of timing constraints in the DC part. For infinite data and more general DC formula one will need interactive verification techniques.

In this paper the DC part restricts only the timing of the communications. In general one would also like to restrict the timed behaviour of the class state. To this end, we pursue the idea that the current state of the Z part is made observable by a special communication.

Acknowledgement. Christian Ohler implemented the algorithm transforming FDR transition systems and DC patterns into UPPAAL timed automata.

References

1. J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and Wang Yi. Uppaal – a tool suite for automatic verification of real-time systems. In R. Alur, T.A. Henzinger, and E.D. Sonntag, editors, *Hybrid Systems III – Verification and Control*, volume 1066 of *LNCS*, pages 232–243. Springer, 1997.
2. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Object Technology Series. Addison Wesley, 1999.
3. J. Davies and S. Schneider. A brief history of Timed CSP. *Theoretical Computer Science*, 138:243–271, 1995.
4. H. Dierks. PLC-Automata: A New Class of Implementable Real-Time Automata. *Theoretical Computer Science*, 253(1):61–93, 2001.
5. H. Dierks, A. Fehnker, A. Mader, and F.W. Vaandrager. Operational and Logical Semantics for Polling Real-Time Systems. In A.P. Ravn and H. Rischel, editors, *FTRTFT'98*, volume 1486 of *LNCS*, pages 29–40. Springer, 1998.
6. H. Dierks and J. Tapken. Modelling and verifying of a 'cash point service' using MOBY/PLC. *Formal Aspects of Computing*, 12:220–221, 2000.
7. C.J. Fidge, I.J. Hayes, A.P. Martin, and A.K. Wabenhorst. A set-theoretic model for real-time specification and reasoning. In J. Jeuring, editor, *Mathematics of Program Construction*, volume 1422 of *LNCS*, pages 188–206. Springer, 1998.
8. 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.
9. 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.
10. C. Fischer and H. Wehrheim. Model-checking CSP-OZ specifications with FDR. In K. Araki, A. Galloway, and K. Taguchi, editors, *Integrated Formal Methods*, pages 315–334. Springer, 1999.
11. Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR 2*, Dec. 1995.
12. M.R. Hansen and C. Zhou. Duration calculus: Logical foundations. *Formal Aspects of Computing*, 9:283–330, 1997.
13. J. He, C.A.R. Hoare, M. Fränzle, M. Müller-Olm, E.-R. Olderog, M. Schenke, M.R. Hansen, A.P. Ravn, and H. Rischel. Provably correct systems. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault Tolerant Systems*, volume 863 of *LNCS*, pages 288–335. Springer, 1994.
14. S. Heilmann. *Proof Support for Duration Calculus.* PhD thesis, Dept. Inform. Technology, Tech. Univ. Denmark, June 1999. Tech. Report IT-TR: 1999-030.
15. M. Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.
16. C.A.R. Hoare. Communicating sequential processes. *CACM*, 21:666–677, 1978.
17. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

18. C.A.R. Hoare and J. He. *Unifying Theories of Programming.* Prentice Hall, 1997.
19. Kolyang. *HOL-Z – An Integrated Formal Support Environment for Z in Isabelle/HOL.* PhD thesis, Univ. Bremen, 1997. Shaker Verlag, Aachen, 1999.
20. D.G. Luenberger. *Introduction to Dynamic Systems. Theory, Models & Applications.* Wiley, 1979.
21. 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.
22. 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.
23. B. Moszkowski. A temporal logic for multi-level reasoning about hardware. *IEEE Computer*, 18(2):10–19, 1985.
24. B. Moszkowski. *Executing Temporal Logic Programs.* Cambridge Univ. Press, 1986.
25. R. De Nicola and M. Hennessy. Testing equivalences of processes. *Theoretical Computer Science*, 34:83–133, 1983.
26. E.-R. Olderog, A. P. Ravn, and J. U. Skakkebæk. Refining system requirements to program specifications. In C. Heitmeyer and D. Mandrioli, editors, *Formal Methods for Real-Time Computing*, pages 107–134. Wiley, 1996.
27. A.P. Ravn. Design of embedded real-time computing systems. Technical Report ID-TR: 1995-170, Tech. Univ. Denmark, 1995. Thesis for Doctor of Technics.
28. A.P. Ravn, H. Rischel, and K.M. Hansen. Specifying and verifying requirements of real-time systems. *IEEE Trans. Software Engineering*, 19(1):41–55, 1993.
29. A.W. Roscoe. Model-checking CSP. In A.W. Roscoe, editor, *A Classical Mind — Essays in Honour of C.A.R.Hoare*, pages 353–378. Prentice-Hall, 1994.
30. A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1997.
31. M. Saaltink. The Z/EVES system. In J. Bowen, M. Hinchey, and D. Till, editors, *ZUM'97*, volume 1212 of *LNCS*, pages 72–88. Springer, 1997.
32. T. Santen. *A Mechanized Logical Model of Z and Object-Oriented Specification.* PhD thesis, Tech. Univ. Berlin, Juli 1999. Shaker Verlag, Aachen, 2000.
33. M. Schenke and E.-R. Olderog. Transformational design of real-time systems – Part 1: from requirements to program specifications. *Acta Inform.*, 36:1–65, 1999.
34. B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. Technical report, ObjecTime, 1998.
35. J.U. Skakkebæk. *A Verification Assistent for a Real-Time Logic.* PhD thesis, Dept. Comp. Sci., Tech. Univ. Denmark, Nov. 1994. Tech. Report ID-TR: 1994-150.
36. G. Smith. *The Object-Z Specification Language.* Kluwer Academic Publisher, 2000.
37. G. Smith and I. Hayes. Towards real-time Object-Z. In K. Araki, A. Galloway, and K. Taguchi, editors, *Integrated Formal Methods*, pages 49–65. Springer, 1999.
38. J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall International Series in Computer Science, 2nd edition, 1992.
39. J. Woodcock and J. Davies. *Using Z — Specification, Refinement, and Proof.* Prentice-Hall, 1996.
40. C. Zhou, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.