Nordic Journal of Computing 9(2002), 301–334

CSP-OZ-DC: A COMBINATION OF SPECIFICATION TECHNIQUES FOR PROCESSES, DATA AND TIME *

JOCHEN HOENICKE

ERNST-RÜDIGER OLDEROG
Department of Computing Science, University of Oldenburg
26111 Oldenburg, Germany
{hoenicke,olderog}@informatik.uni-oldenburg.de

Abstract. CSP-OZ-DC is a new combination of three well researched formal techniques for the specification of processes, data and time: CSP [Hoare 1985], Object-Z [Smith 2000], and Duration Calculus [Zhou *et al.* 1991]. This combination is illustrated by specifying the train controller of a case study on radio controlled railway crossings. The technical contribution of the paper is a smooth integration of the underlying semantic models and its use for verifying timing properties of CSP-OZ-DC specifications. This is done by combining the model-checkers FDR [Roscoe 1994] for CSP and UPPAAL [Bengtsson *et al.* 1997] for timed automata with a new tool $f2u$ that transforms FDR transition systems and certain patterns of Duration Calculus formulae into timed automata. This approach is illustrated by the example of a vending machine.

CR Classification: D.2.1, D.2.2, D.2.4, F.3.1, F.4.1

Key words: CSP, Object-Z, Duration Calculus, transformational semantics, 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 not yet 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 [1978, 1985]. The central concepts of CSP are synchronous communication via

*A preliminary version of this article, with different examples, appeared as the conference paper [Hoenicke and Olderog 2002] published by Springer-Verlag. This research was partially supported by the DFG under grant Ol/98-2.

Received March 20, 2002; accepted December 20, 2002

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 [Roscoe 1997]. Tool support comes through the FDR model-checker [Roscoe 1994]. 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.

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 [Spivey 1992]. Since then the language has been published extensively (e.g. [Woodcock and Davies 1996]) 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 [Smith 2000]. 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 [Saaltink 1997] or HOL-Z based on Isabelle [Kolyang 1997]. For Object-Z type checkers exist. Verification support is less developed except for an extension of HOL-Z [Smith *et al.* 2002].

Duration Calculus (DC for short) originated during the ProCoS (Provably Correct Systems) project [He *et al.* 1994] as a new logic and calculus by Zhou *et al.* [1991] and Hansen and Zhou [1997] for specifying the behaviour of real-time systems. It is based on the notion of an observable *obs* interpreted as a time dependent function $obs_I : Time \to D$ 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 [Luenberger 1979] and enables extensions to cover hybrid systems. Duration Calculus was inspired by the work of Moszkowski [1985, 1986] on interval temporal logic 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 [see Ravn *et al.* 1993, Olderog *et al.* 1996, Schenke and Olderog 1999, Dierks 2001]. Verification support for the general DC is provided by Skakkebæk [1994] and Heilmann [1999] using theorem provers, and for a more specialised application of DC by Dierks and Tapken [2000] using a translation into timed automata for model-checking with UPPAAL [Bengtsson *et al.* 1997].

It is well known that a consistent combination of different specification techniques is difficult [Hoare and He 1997]. Very popular is currently UML, the Unified Modeling Language [Booch *et al.* 1999]. It collects all the widespread specification techniques for object-oriented systems in one language. There is even an extension UML-RT [Selic and Rumbaugh 1998] 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 missing semantic basis for this huge language. It is still a topic of ongoing research to provide a semantics for subsets of UML.

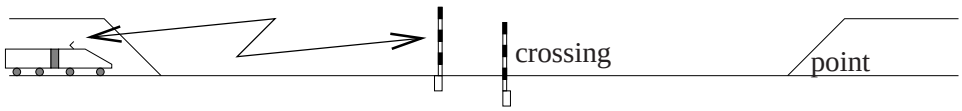


Fig. 1: Case study: Radio controlled railway crossings

We see the best chances for a well founded combination with specification techniques that are well researched individually. As guidelines for good combinations we propose the following:

- the strengths of the individual specifications techniques should be preserved,
- tools and verification methods should be reusable,
- the semantics of the combination should be easy to understand,
- different representations should be freely exchangeable, e.g. visual vs. textual.

An example of a clear combination of two specification techniques is CSP-OZ developed by Fischer [1997, 2000]. 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 employs it to specify a train controller for the safety of railway crossings. Section 5 shows how the semantics can be utilized for a partially automatic verification of properties of CSP-OZ-DC specifications, and applies this approach to a simple vending machine. Finally, we conclude with Section 6.

2. The Combination CSP-OZ-DC

In this section we introduce the new combined formalism by examples taken from a case study of radio controlled railway crossings which is part of the priority research program "Integration of specification techniques with applications in engineering" [1] of the German Research Council (DFG), 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. 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, to decide when it is time to switch points and secure crossings, and to make sure that the train does not enter them too early. The *odometer* keeps track of the speed and position of the train. The position is measured by various means, e. g. counting the rotations of the wheels. Buried in the track are so-called *balises*, which are devices with a unique identifier that can be read by the train.

[1] `http://tfs.cs.tu-berlin.de/projekte/indspec/SPP/index.html`

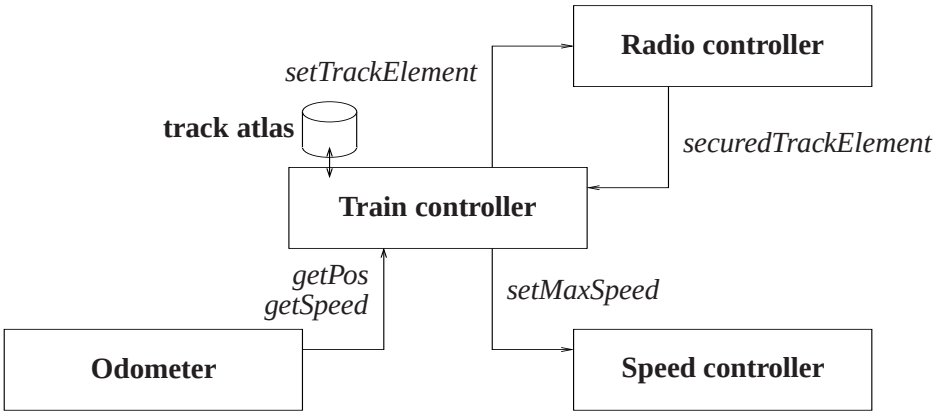


Fig. 2: Controller architecture

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 points and 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.

2.1 Using CSP

To specify the train controller component several concepts must be handled, as described in the following. The train controller communicates with other components, e. g. the radio controller. Most of these communications are initiated by the train controller itself, e. g. the train controller asks the odometer for the current speed and position and sets the new speed limit. But there are also communications initiated externally, e. g. *securedTrackElement*, which is sent when a crossing affirms that it is safe. These communications can be easily modelled with CSP.

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

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

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* (an internal communication) and *setMaxSpeed* are communicated in this order. For simplicity communication values are ignored

here. This process can then work in parallel with other processes, for instance the one handling the *securedTrackElement*:

$$\begin{aligned} \mathit{SecuredHandler} &\stackrel{c}{=} \mathit{securedTrackElement?id} \\ &\quad \rightarrow \mathit{clearDangerPosition!id} \rightarrow \mathit{SecuredHandler} \\ \texttt{main} &\stackrel{c}{=} \mathit{SuperviseSpeed} \parallel \mathit{SecuredHandler} \end{aligned}$$

Note that a value is communicated along the channels *securedTrackElement* and *clearDangerPosition*. This value will later be used by the Z part, see below. Receiving and sending of a value is indicated by the symbols ? and ! according to the CSP conventions.

2.2 Using Object-Z

The *track atlas* contains a data base with the crossings and points. The train has to manipulate this data base to remember which track elements have already been switched and which have affirmed their safety. Handling of 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

$$\begin{aligned} &\mathit{id} : \mathit{Identifier} \\ &\mathit{pos} : \mathit{Position} \\ &\ldots \end{aligned}$$

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 dots indicate that there is more information in the schema, e. g. to distinguish crossings from points and to know in which direction to switch a point.

The track atlas contains information about track elements, the maximum speed for each track segment, and all other information the train needs to know about the track. It is also represented by a Z schema as follows. The definition of *StaticProfile* is not of interest here and is given in Sect. 4. The type seq *TrackElement* denotes finite sequences of *TrackElement*s.

TrackAtlas

$$\begin{aligned} &\mathit{staticprof} : \mathit{StaticProfile} \\ &\mathit{elems} : \mathrm{seq}\, \mathit{TrackElement} \end{aligned}$$

For each crossing that is not yet secured the train controller keeps its associated positions in a set *dangerPositions*. Together with the track atlas this forms the state space of the train controller. The state space of a OZ class is denoted by an unnamed schema.

$$\begin{array}{|l} trackatlas : TrackAtlas \\ dangerPositions : \mathbb{P}\, Position \end{array}$$

Initially for every track element in the track atlas the corresponding danger position is set, to make sure that the train cannot pass it. In OZ the initial state is described by an *Init* schema like this:

Init

$$dangerPositions = \{elem : \mathrm{ran}\, trackatlas.elems \bullet elem.pos\}$$

The set on the right side contains elements of the form *elem.pos* where *elem* ranges over the elements in the sequence *trackatlas.elems*. The operator ran is used here to convert a sequence into the set of its elements.

When a value, i.e. position *id*, is received on channel *securedTrackElement*, the train controller has to remove this position from the set of danger positions. The handling of the communication event *clearDangerPosition*!*id* is done by the CSP process *SecuredHandler*, but we have to link this event with an update of the data base. This is done by writing a communication schema for *clearDangerPosition*, i.e. a Z-schema specifying the operation associated with that communication event.

com_*clearDangerPosition*

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

The prefix com_ of the schema name indicates that this is a communication scheme. It is possible to decompose communication schemas into enable and effect schemas [Fischer 2000] but we shall not pursue this here. The Δ in the first line of the schema declares that this operation may only change *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 all positions *elem.pos* are removed, where *elem* is an element from the sequence *trackatlas.elems* that has the identifier *id*?.

2.3 Using Duration Calculus

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). It is a interval-based real-time logic and calculus developed by Zhou *et al.* [1991]. It can be applied for specifying both high-level requirements and implementation-level details of real-time systems.

In DC *state assertions* P describe time dependent properties of observables $obs : Time \to D$. *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. A point interval is specified by $\lceil\rceil$, which abbreviates the formula $\ell = 0$. 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. For more details see [Hansen and Zhou 1997].

A subset of the DC is called *implementables* due to Ravn [1995], which make use of the following idioms where $t \in Time$:

$$F \longrightarrow \lceil P \rceil \;==\; \Box\neg(F ; \lceil\neg P\rceil) \qquad [\text{followed-by}]$$
$$F \xrightarrow{t} \lceil P \rceil \;==\; (F \wedge \ell = t) \longrightarrow \lceil P \rceil \qquad [\text{leads-to}]$$
$$F \xrightarrow{\leq t} \lceil P \rceil \;==\; (F \wedge \ell \leq t) \longrightarrow \lceil P \rceil \qquad [\text{up-to}]$$
$$F \longrightarrow_0 \lceil P \rceil \;==\; \neg(F ; \lceil\neg P\rceil) \qquad [\text{followed-by-initially}]$$
$$F \xrightarrow{\leq t}_0 \lceil P \rceil \;==\; (F \wedge \ell \leq t) \longrightarrow_0 \lceil P \rceil \qquad [\text{up-to-initially}]$$
$$F \xrightarrow{t}_0 \lceil P \rceil \;==\; (F \wedge \ell = t) \longrightarrow_0 \lceil P \rceil \qquad [\text{leads-to-initially}]$$

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. Note that the "leads-to" does not simply say that whenever F holds then t time units later $\lceil P \rceil$ holds, but it rather requires a *stability* of F for t time units before we can be certain that $\lceil P \rceil$ holds. The *initially* variants require the above behaviour only starting at time 0.

As an example consider the following DC formula which states that the next *setMaxSpeed* communication must occur after at most one second:

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

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. Therefore we count the number of times they occur and reason about this count. The above formula states that if the number of *setMaxSpeed* events stays stable

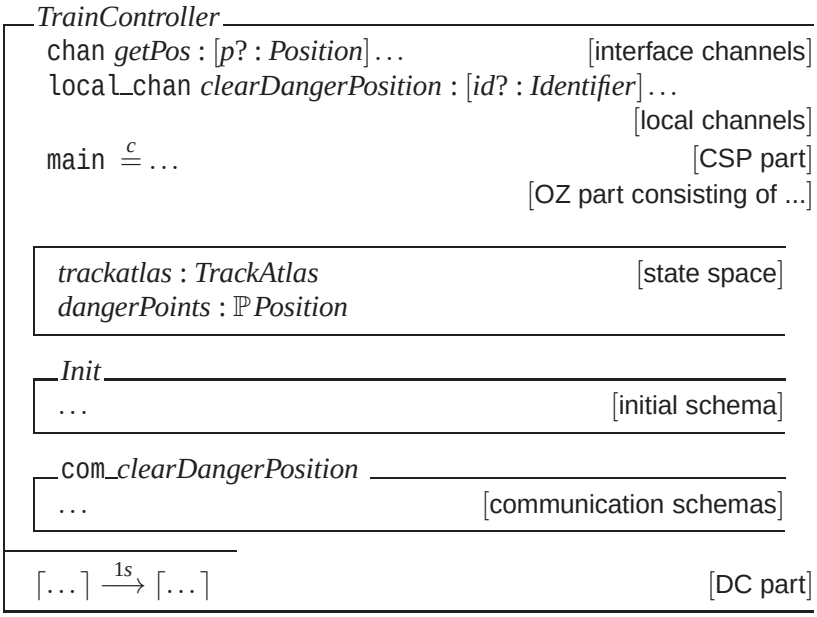


Fig. 3: Class in CSP-OZ-DC

for one second, then the event has to occur afterwards so that $ct(setMaxSpeed)$ increases.

2.4 Specifying Classes and Systems

The basic building block in our combined formalism CSP-OZ-DC is a class. Its syntax is as in CSP-OZ [see Fischer 1997, 2000], only the DC part is new, see Fig. 3. First, the communication 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 only visible inside the class and through which the CSP, OZ, and DC parts interact. Second, the CSP part follows; it is given by a system of (recursive) process equations. Third, the OZ part is given which itself consists of the state space, the Init schema and communication schemas specifying 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 the CSP structuring operators, i.e. parallel composition, renaming and hiding [Roscoe 1997]. This allows us to describe architectures like the one in Fig. 2.

3. Semantics

Each class of a CSP-OZ-DC specification denotes a time dependent process. Here 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 [Roscoe 1997]. 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

$$Traces == \operatorname{seq} Comm \;\text{ and }\; Refusals == \mathbb{P} Comm,$$
$$Failures == Traces \times Refusals.$$

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

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

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 Roscoe 1997, p.192]. The $\mathcal{FD}$-semantics 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. Fig. 4 exhibits the refusal and acceptance sets after the empty trace of the process $a \to \texttt{Stop} \;\Box\; b \to \texttt{Stop}$.

Acceptance sets are due to De Nicola and Hennessy [1983] and Hennessy [1988] 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 Hennessy 1988, p.77]. For example, they are closed under union.

Acceptance sets are discussed also in [Roscoe 1997, p. 278] as a means of normal form representation of CSP processes. However, in contrast to Hennessy and De

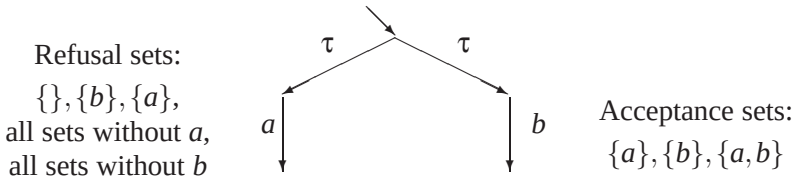


Fig. 4: Refusal and acceptance sets

Nicola's approach, there for each trace s only the *minimal* acceptance sets after s together with the set of all initial communications after s, which is the largest acceptance sets after s, are considered.

Formally, let

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

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

$$\mathcal{A} : \text{CSP} \to \mathbb{P}(Traces \times Acceptances)$$

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 holds:

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 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* [Woodcock and Davies 1996].

Duration Calculus (DC) specifies properties of observables *obs* interpreted as *finitely varying* functions of the form $obs_I : Time \to D$ for a continuous time domain *Time* and a data domain D (see Fig. 5). The concept of *finite variability* means that the function obs_I has at most finitely many discontinuity points in any finite time interval. As a consequence the integral (duration) operator of DC is well-defined and an induction rule for DC is valid [Hansen and Zhou 1997].

When modelling real-time systems in Duration Calculus, *refinement* is represented by logical implication $\Rightarrow$, i.e. for duration formulae F, G we say F *refines* G iff $F \Rightarrow G$. As examples we state two refinement laws for DC implementables due to Ravn [1995].

PROPOSITION 2. *Let P, Q be state assertions and $t, t' \in Time$ with $t \leq t'$.*

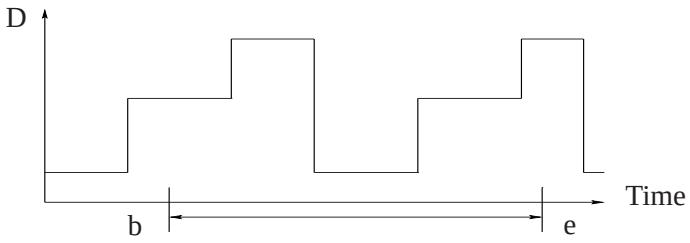


Fig. 5: Finitely varying function

(a) Decreasing the upper time bound of a reaction is a refinement, i.e. we have $\lceil P \rceil \xrightarrow{t} \lceil Q \rceil \Rightarrow \lceil P \rceil \xrightarrow{t'} \lceil Q \rceil$.

(b) Increasing the lower time bound of a stability is a refinement, i.e. we have $\lceil P \rceil \xrightarrow{\leq t'} \lceil Q \rceil \Rightarrow \lceil P \rceil \xrightarrow{\leq t} \lceil Q \rceil$.

3.2 Untimed semantics of CSP-OZ classes

The untimed semantics of the combination CSP-OZ is defined in Fischer [1997, 2000]. The idea is that each CSP-OZ class denotes a process in the semantic model of CSP. This is achieved by transforming the OZ 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	[OZ part]

also written horizontally as $U \mathrel{\widehat{=}} \texttt{spec}\ I\ L\ P\ Z\ \texttt{end}$ with an OZ part

Z

$st : State$	[state space]
$Init(st)$	[initial condition]
$\ldots \texttt{com\_}c(st, in?, out!, st') \ldots$	
	[one communication schema for each c in I or L]

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

The OZ part of the class is transformed into a CSP process *OZMain* defined by the following system of (parametrised) recursive equations for *OZpart* using the

(indexed) CSP operators for internal nondeterministic choice ($\sqcap$) and alternative composition ($\Box$):

$$\begin{aligned} \mathit{OZMain} &= \sqcap_{st}\ \mathit{OZPart}(st) \\ \mathit{OZPart}(st) &= \Box_{c,\ in?}\ \sqcap_{out!,\ st'}\ c.in?.out! \rightarrow \mathit{OZPart}(st') \end{aligned}$$

where *st* ranges over all states in *State* satisfying $\mathit{Init}(st)$. Thus the process *OZMain* can nondeterministically choose any state *st* satisfying $\mathit{Init}(st)$ to start with. Further on, *c* ranges over all channels declared in *I* or *L*, and *in*? ranges over the set $\mathit{Inputs}(c)$ such that the condition

$$\exists\, out! : \mathit{Outputs}(c);\ st' : \mathit{State}' \bullet \mathtt{com\_}c(st, in?, out!, st')$$

holds. Finally, for any chosen *c* and *in*?, the value *out*! ranges over the set $\mathit{Outputs}(c)$, and st' ranges over State' such that

$$\mathtt{com\_}c(st, in?, out!, st')$$

holds. So the $\mathit{OZPart}(st)$ 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 $\mathtt{com\_}c(st, in?, out!, st')$ is satisfiable for some output values *out*! and successor state st'. For given input values *in*? any such *out*! and st' can be nondeterministically chosen to yield $c.in?.out!$ and the next recursive call $\mathit{OZPart}(st')$. Thus input and output along channels *c* are modelled by a subtle interplay of the CSP alternative and nondeterministic choice.

OZMain runs in parallel with the explicit CSP process *P* of the class:

$$\mathit{proc}_U = P\ [|\ \mathit{Events}(I \cup L)\ |]\ \mathit{OZMain}$$

Here the parallel composition synchronises on all events in *I* and *L*. In [Fischer 1997, 2000] the semantics of the class *U* is then defined by

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

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. The following theorem is shown in [Fischer 2000]:

THEOREM 1. *(a) Process refinement $P_1 \sqsubseteq_{\mathcal{FD}} P_2$ implies refinement in CSP-OZ:* $\mathtt{spec}\ I\ L\ P_1\ Z\ \mathtt{end} \quad \sqsubseteq_{\mathcal{FD}} \quad \mathtt{spec}\ I\ L\ P_2\ Z\ \mathtt{end}$
(b) Data refinement $Z_1 \sqsubseteq_\rho Z_2$ for a refinement relation ρ implies refinement in CSP-OZ: $\mathtt{spec}\ I\ L\ P\ Z_1\ \mathtt{end} \quad \sqsubseteq_{\mathcal{FD}} \quad \mathtt{spec}\ I\ L\ P\ Z_2\ \mathtt{end}$

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 from 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 the simplicity of the subsequent exposition, we assume that the untimed part is divergence free, formally: $\mathcal{D}[\![proc_U]\!] = \varnothing$. Then 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:

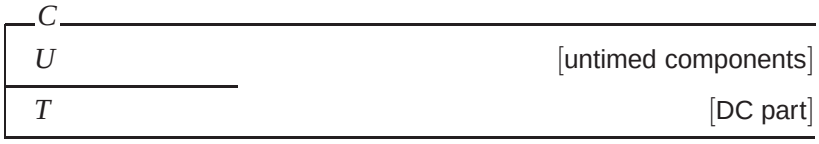
$$\mathcal{A}_{Time} : \text{CSP-OZ-DC} \to \mathbb{P}((Time \to Traces) \times (Time \to Acceptances))$$

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

$$tr_I : Time \to Traces \quad \text{and} \quad Acc_I : Time \to Acceptances.$$

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 a timing part T expressed in DC:



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 $proc_U$, which we assumed to be divergence free, i.e.

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

we define its timed semantics as the DC formula

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

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 $proc_U$.

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

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. Thus the subformula $\lceil tr = h\rceil;\ \lceil tr = h'\rceil$ holds in a given time interval if on a first non-point interval the trace *tr* assumes the value *h* and on a second non-point interval it assumes the value h'. By $\mathcal{F}_2$, h' can differ from *h* only by one communication event. Together with the restriction to finite variability (cf. subsection 3.1), we thus require that only finitely many communication events occur within any 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,

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

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,

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

In Z a finite sequence $tr = \langle a_1, ..., a_n\rangle$ is represented as a mapping of the form $\{1 \mapsto a_1, ..., n \mapsto a_n\}$. Then $tr \rhd X$ restricts the range of this mapping to those elements with $a_i \in X$. The operator # counts the remaining elements in this set. Thus $ct(X)$ counts the number of occurrences of events from *X* in the trace *tr*. Next

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

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

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

For a finite sequence $tr_0 = \langle a_1, ..., a_n \rangle = \{1 \mapsto a_1, ..., n \mapsto a_n\}$ the *range subtraction* $tr_0 \rhd L$ of Z removes all pairs $i \mapsto a_i$ with $a_i \in L$. This may result in a mapping which is not a sequence any more (due to missing indices), for instance $tr_0 \rhd \{a_1\} = \{2 \mapsto a_2, ..., n \mapsto a_n\}$. The operator *squash* transforms this mapping into a proper sequence, here $squash(tr_0 \rhd \{a_1\}) = \{1 \mapsto a_2, ..., n-1 \mapsto a_n\}$.

Thus tr is a trace resulting from tr_0 by removing all communications in L, and $\texttt{hide}\ L \bullet F$ is a DC formula in the observables tr and Acc with values 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, renaming and hiding. Thus a typical specification could be of the form

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

Renaming, denoted by the postfix operator $[R]$, where R is a binary relation between events, is used to change the names of communication events. The semantic definition is straightforward. *Hiding*, denoted by the postfix operator $\setminus L$, is used to make all communication events in set L internal. Semantically, hiding is defined using the operator $\texttt{hide}\ L \bullet F$ introduced above.

The *parallel composition* $C_1[R_1] \parallel C_2[R_2]$ is a special case of the *alphabetised parallel* $C_1[R_1]\ {}_A\|_B\ C_2[R_2]$ where A and B are the sets of interface events of the components $C_1[R_1]$ and $C_2[R_2]$. For DC formulae F_1 and F_2 in the observables tr and Acc the semantics of the alphabetised parallel can be expressed, similarly to [Schenke and Olderog 1999], by the following DC formula:

$$F_1\ {}_A\|_B\ F_2 \quad \Leftrightarrow \quad \exists tr_1, tr_2, Acc_1, Acc_2 \bullet$$

$$\begin{array}{c}((\lceil\rceil \vee \lceil tr \in \mathrm{seq}(A \cup B) \wedge tr \restriction A = tr_1 \wedge tr \restriction B = tr_2 \wedge \\ Acc = (A \cap B \cap Acc_1 \cap Acc_2) \cup \\ ((A \setminus B) \cap Acc_1) \cup \\ ((B \setminus A) \cap Acc_2)\rceil) \wedge \\ F_1[tr_1/tr, Acc_1/Acc] \wedge F_2[tr_2/tr, Acc_2/Acc])\end{array}$$

Intuitively, $F_1 \ {}_A\|_B \ F_2$ describes a combination where the left-hand process is allowed to engage in those events of A that satisfy F_1, the right-hand process is allowed to engage in those events of B that satisfy F_2, and both must synchronise on every event in the intersection $A \cap B$.

For traces this is formalised using the *filter* or *projection* function $\restriction$ of Z. If tr is a finite sequence, $tr \restriction A$ is the largest subsequence of tr containing only those elements that belong to the set A. Note that $tr \restriction A = squash(tr \rhd A)$ holds. For instance, if $tr = \langle a,a,b,c,d,d,e\rangle$ then $tr \restriction \{a,d\} = \langle a,a,d,d\rangle$. Thus $F_1 \ {}_A\|_B \ F_2$ describes all traces tr over the joint alphabet $A \cup B$ such that the projections $tr \restriction A$ and $tr \restriction B$ are consistent with the formulae F_1 and F_2. For the common acceptance set Acc three cases are distinguished: inside the synchronisation set $A \cap B$ of all events are taken that can be accepted by *both* F_1 and F_2, inside the set difference $A \setminus B$ all events are taken that can be accepted by F_1, and inside the set difference $B \setminus A$ all events are taken that can be accepted by F_2.

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

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

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:* $\texttt{spec}\ I\ L\ P_1\ Z\ T\ \texttt{end} \quad \sqsubseteq \quad \texttt{spec}\ I\ L\ P_2\ Z\ T\ \texttt{end}$

(b) Data refinement $Z_1 \sqsubseteq_\rho Z_2$ for a refinement relation ρ implies refinement in CSP-OZ-DC: $\texttt{spec}\ I\ L\ P\ Z_1\ T\ \texttt{end} \quad \sqsubseteq \quad \texttt{spec}\ I\ L\ P\ Z_2\ T\ \texttt{end}$

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

PROOF. Properties (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. For instance, for (a) let $U_i = \texttt{spec}\ I\ L\ P_i\ Z\ \texttt{end}$ and $C_i = \texttt{spec}\ I\ L\ P_i\ Z\ T\ \texttt{end}$ where $i = 1,2$. Then

	$P_1 \sqsubseteq_{\mathcal{FD}} P_2$	
implies	$U_1 \sqsubseteq_{\mathcal{FD}} U_2$	(by Theorem 1)
implies	$\mathcal{A}[\![proc_{U_2}]\!] \subseteq \mathcal{A}[\![proc_{U_1}]\!]$	(by Proposition 1)
implies	$\mathcal{F}_{U_2} \Rightarrow \mathcal{F}_{U_1}$	(by the monotonicity of $\mathcal{F}_U$)
implies	$\mathcal{A}_{\mathit{Time}}[\![C_2]\!] \Rightarrow \mathcal{A}_{\mathit{Time}}[\![C_1]\!]$	(by the form of $\mathcal{A}_{\mathit{Time}}[\![C]\!]$)
implies	$C_1 \sqsubseteq C_2$	(by the definition of $\sqsubseteq$)

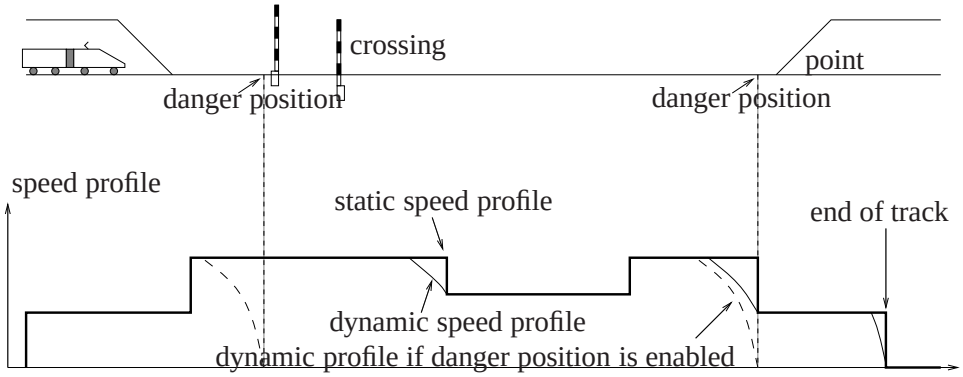


Fig. 6: Braking curve

Property (c) follows from the conjunctive form of the formula $\mathcal{A}_{Time}[\![C]\!]$. □

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 Section 5 we develop one such a technique.

4. Case Study

In Section 2 we have already introduced the case study of a 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. A central concept is the *braking curve*, see Fig. 6, 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.

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

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).

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

With this function it is possible to calculate the dynamic profile. The calculation is straightforward, so we do not go into details here. 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.

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

In Section 2 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
$$staticprof : StaticProfile$$
$$elems : \mathrm{seq}\, TrackElement$$

Using these types we now specify the *TrainController* as a CSP-OZ-DC class. We concentrate on the parts relevant for the braking curve, see Fig. 7. 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 *setDangerPosition*, *clearDangerPosition* and *calcMaxSpeed* are internal communication channels used to link the CSP and OZ part. The CSP processes already occurred in Section 2, but here we also show the communicated data.

The state space of the train controller consists of the *trackatlas*, a set of *dangerPositions* and the dynamic speed profile *dynProf*. The latter is calculated from the first two. In Object-Z this is represented by putting the dependent variable below a Δ-sign. This sign indicates that *dynProf* may change implicitly when *dangerPositions* changes. The formula defining *dynProf* in terms of the other variables is written below the horizontal line. This formula is also called a *class invariant*.

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 *reactDist* which calculates the maximum position the train may reach within its reaction time. The reaction time can be calculated from the first DC formula: the *setMaxSpeed* event occurs every second. Since the CSP part requires a *getPos* event between two *setMaxSpeed* event, the maximum time between *getPos* and *setMaxSpeed* is also a second. So at most two seconds after the last *getPos* event the *setMaxSpeed* is called a second time with updated values.

This is the reaction time of the train controller. To this time we can add the reaction time of the *SpeedController* to get the total reaction time. So assuming that the maximum speed of the train is given in a variable *maxSpeed* the maximum distance the train may pass in two seconds reaction time can be over-approximated and calculated by the following formula.

$$\begin{array}{|l} reactDist : Speed \to Position \\ \hline \forall s : Speed \bullet reactDist(s) = maxSpeed * reactTime \end{array}$$

The second DC formula gives an example of the *enable* predicate. It is used to check whether the *setTrackElement* communication is possible. This event is possible whenever the position in the *setpos* component of this track element was reached and the element was not notified yet. Whenever *setTrackElement* is enabled it should occur after at most one second.

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 (part of) the class *TrainController*:

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

5. Verification

We exploit the transformational semantics given in Section 3 for a partially automatic verification of properties of CSP-OZ-DC specifications satisfying 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. Our approach builds on existing tools and techniques:

TrainController

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

$$\texttt{main} \stackrel{c}{=} SuperviseSpeed \,||\, SecuredHandler$$
$$SuperviseSpeed \stackrel{c}{=} getSpeed?speed \rightarrow getPos?pos \rightarrow calcMaxSpeed!pos!speed?maxs \rightarrow setMaxSpeed!maxs \rightarrow SuperviseSpeed$$
$$SecuredHandler \stackrel{c}{=} securedTrackElement?id \rightarrow clearDangerPosition!id \rightarrow SecuredHandler$$

$trackatlas : TrackAtlas$
$dangerPositions : \mathbb{P}\, Position$
Δ
$dynProf : Position \rightarrow Speed$

$$dynProf = calcProfile(trackatlas.staticprof, dangerPositions)$$

Init

$$dangerPositions = \{elem : \mathrm{ran}\, trackatlas.elems \bullet elem.pos\}$$

$\texttt{com\_}setDangerPosition$

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

$$dangerPositions' = dangerPositions \cup \{elem : \mathrm{ran}\, trackatlas.elems \mid elem.id = id? \bullet elem.pos\}$$

$\texttt{com\_}clearDangerPosition$ [see Section 2]
$\texttt{com\_}setTrackElement$ [definition omitted here]

$\texttt{com\_}calcMaxSpeed$

$p? : Position$
$s? : Speed$
$maxs! : Speed$

$$\texttt{let}\ endp = p? + reactDist(s?) \bullet maxs! = min\, dynProf(|\, [p?, endp]\, |)$$

$$\lceil ct(setMaxSpeed) = n \rceil \xrightarrow{1s} \lceil ct(setMaxSpeed) > n \rceil$$
$$\lceil en(setTrackElement) \wedge ct(setTrackElement) = n \rceil \xrightarrow{1s} \lceil ct(setTrackElement) > n \rceil$$
$$(\lceil ct(setTrackElement.id) = n \rceil;\ (\ell = 300s \wedge \lceil ct(setTrackElement.id) > n \wedge ct(setDangerPoint.id) = m \rceil)) \longrightarrow \lceil ct(setDangerPoint.id) > m \rceil$$

Fig. 7: Class *TrainController*

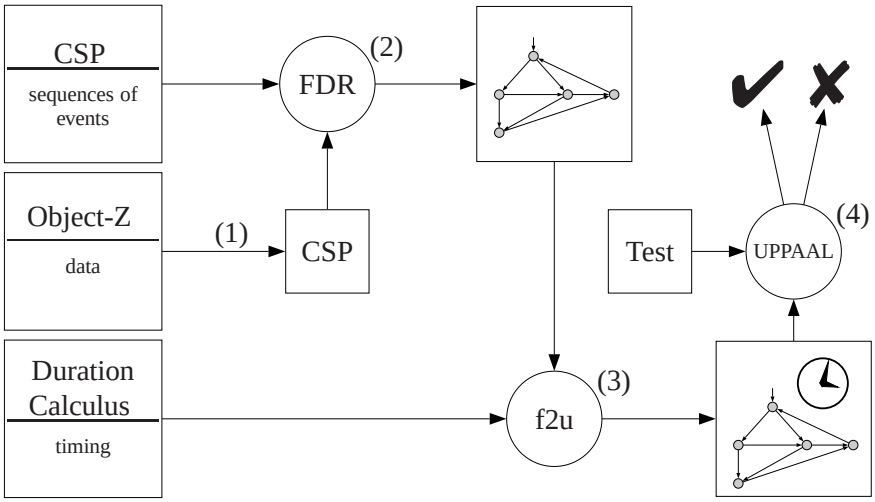


Fig. 8: Procedure for model-checking

- for dealing with CSP we use the FDR model-checker [see Roscoe 1994, Formal Systems (Europe) Ltd 1995],
- for dealing with Object-Z we use the transformation into CSP first described by Fischer and Wehrheim [1999],
- for dealing with timing properties we use the model-checker UPPAAL [Bengtsson *et al.* 1997] for timed automata.

Additionally, we use a new tool, called *f2u*, described below. The idea is as follows. Given a class $C \mathrel{\widehat{=}} \texttt{spec}\ I\ L\ P\ Z\ T\ \texttt{end}$ we proceed in four steps as illustrated in Fig. 8:

(1) Transform the untimed process $U = \texttt{spec}\ I\ L\ P\ Z\ \texttt{end}$ into FDR-CSP, the input language of the FDR model-checker for CSP.

(2) Apply the FDR model-checker to output a transition system TS_U for U with acceptance sets.

(3) Use our new tool, *f2u*, to transform this transition system TS_U into a timed automaton $\mathcal{A}_C$ representing all the timing restrictions of the DC part T of C.

(4) Verify properties of the class C by applying the model-checker UPPAAL to $\mathcal{A}_C$.

A direct automatic treatment of the DC part is not feasible because for DC only interactive verification support is available [see Skakkebæk 1994 and Heilmann 1999]. Also the model-checker of Tapken [2001] is not operating on DC but on phase automata. Hence the transformation step (3) is needed.

Although step (1) represents just the CSP process *OZMain* of Section 3, it requires user interaction to represent the state *st* and the communications schemas $\texttt{com\_}c(st, in?, out!, st')$ in FDR-CSP [see Fischer and Wehrheim 1999]. Steps (2)

and (3) proceed fully automatically. Step(4) requires again user interaction in the construction of certain test automata.

5.1 Transforming DC patterns

New is step (3). The DC patterns for timing restrictions that can be handled in this step are new variants of the DC implementables [Ravn 1995] 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:

$$F \underset{X}{;} G == (F \wedge \lceil ct(X) = n \rceil) \underset{X}{;} (G \wedge \lceil ct(X) > n \rceil)$$
$$F \xrightarrow[X]{t} G == (F \wedge \lceil ct(X) = n \rceil) \xrightarrow{t} (G \wedge \lceil ct(X) > n \rceil)$$
$$F \xrightarrow[X]{t}_0 G == (F \wedge \lceil ct(X) = n \rceil) \xrightarrow{t}_0 (G \wedge \lceil ct(X) > n \rceil)$$

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

$$F \xrightarrow[/X]{\leq t} G == (F \wedge \lceil ct(X) = n \rceil) \xrightarrow{\leq t} (G \wedge \lceil ct(X) = n \rceil)$$
$$F \xrightarrow[/X]{\leq t}_0 G == (F \wedge \lceil ct(X) = n \rceil) \xrightarrow{\leq t}_0 (G \wedge \lceil ct(X) = n \rceil)$$

The tool *f2u* supports the following DC patterns:

$$(\lceil P \rceil \underset{X}{;} \lceil Q \rceil) \xrightarrow[Y]{t} \lceil R \rceil \qquad \text{[chop-leads-to]}$$
$$(\lceil P \rceil \underset{X}{;} \lceil Q \rceil) \xrightarrow[/Y]{\leq t} \lceil R \rceil \qquad \text{[chop-up-to]}$$
$$\lceil Q \rceil \xrightarrow[X]{t} \lceil R \rceil \qquad \text{[leads-to]}$$
$$\lceil Q \rceil \xrightarrow[/Y]{\leq t} \lceil R \rceil \qquad \text{[up-to]}$$
$$\lceil Q \rceil \xrightarrow[X]{t}_0 \lceil R \rceil \qquad \text{[leads-to-initially]}$$
$$\lceil Q \rceil \xrightarrow[/Y]{\leq t}_0 \lceil R \rceil \qquad \text{[up-to-initially]}$$

Here $t \in \mathit{Time}$ and P, Q, R are state assertions. The event sets X, Y are optional and can be omitted.

The tool *f2u* 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. 9 the pseudo code for the leads-to pattern. The state assertions Q and R are represented by the set of states satisfying these assertions. 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 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.

5.2 An example

We illustrate the above verification procedure with the example of a simple coffee vending machine. The CSP-OZ-DC class describing this machine is given in Fig. 10. Money is represented by the natural numbers $\mathbb{N}$. As global constants we assume a finite amount of *Money* that can be handled by the *CoffeeMachine*, a finite set *Coin* of different kinds of coins that can be inserted into the *CoffeeMachine*, and a certain *price* for a cup of coffee. These assumption can be made more precise in a concrete application.

The automaton has five channels to communicate with its customer. The channel *in* is used by the customer to insert a coin. It takes the value of the coin as parameter. Similarly an *out* event is generated by the machine when a coin is returned. The other channels have no parameters and represent the events that the user presses the start *button*, gets a *cup*, and gets the *coffee*. The CSP part ensures that the events occur in correct order. First it accepts some incoming coins until the button is pressed. The external choice ensures that the process *Drink* is only called when user presses the button. When this happens the process *Drink* will signal the *cup* event, the *coffee* event, and continue with the process *Return*. This process will return coins until it is finished and then go back to the `main` process.

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. 9: Algorithm for the leads-to pattern

$Money : \mathbb{F}\,\mathbb{N}$
$Coin : \mathbb{F}\,\mathbb{N}$
$price : \mathbb{N}$

CoffeeMachine

chan $in : [coin? : Coin]$
chan $out : [coin! : Coin]$
chan $button, cup, coffee$
local_chan $finished$

$\texttt{main} \stackrel{c}{=} in?x \rightarrow \texttt{main} \;\Box\; Drink$
$Drink \stackrel{c}{=} button \rightarrow cup \rightarrow coffee \rightarrow Return$
$Return \stackrel{c}{=} out?y \rightarrow Return \;\Box\; finished \rightarrow \texttt{main}$

$m : Money$

Init

$m = 0$

com_*button*

$\Delta(m)$

$m \geq price$
$m' = m - price$

com_*in*

$\Delta(m)$
$coin? : Coin$

$m' = m + coin?$

com_*finished*

$\Delta()$

$m = 0$

com_*out*

$\Delta(m)$
$coin! : Coin$

$m' = m - coin!$

$(\lceil \text{true} \rceil \;;_{cup} \lceil en(coffee) \rceil) \xrightarrow{\leq 2} \lceil en(coffee) \rceil$
$\lceil \neg\, (en(button) \vee en(in)) \rceil \xrightarrow[finished]{15} \lceil \text{true} \rceil$

Fig. 10: Coffee vending machine

The CSP process does not care how much money was entered or returned, as this is better done in the OZ part. The only changeable data in this machine is the amount m of money that was inserted by the customer. It is initialized to zero by the Init schema. When a coin is inserted the value m increases by the value of that coin as specified by the com_*in* schema. Note that this schema is not enabled when the new value m' would exceed the maximum of the set *Money*. The machine will then refuse to take any more money.

The communication schema for the *button* event checks that enough money has been inserted and reduces *m* according to the price of the coffee. As long as not enough money has been inserted it will block the *button* event. The value *m* is also reduced when a coin is returned to the customer. Since *coin*! is an output parameter, its value is nondeterministically chosen by the Z schema as long as it does not exceed *m*. Again the schema refuses the *out* event when *m* is zero as it cannot be further reduced. The last schema com_*finished* checks whether *m* is zero.

Last but not least the class contains two timing constraints. The first constraint ensures that after the *cup* event it takes at least two seconds before the *coffee* event happens. This is expressed by the *chop-up-to* pattern introduced earlier. After the cup event the coffee event is enabled and formula requires that it stays enabled for at least two seconds. The expanded form of this constraint is

$$(\lceil ct(cup) = n \rceil ; \lceil ct(cup) > n \wedge en(coffee) \rceil) \xrightarrow{\leq 2} \lceil en(coffee) \rceil \ .$$

The second timing constraint ensures that the time interval in which the automaton is not responding is limited by 15 seconds. The automaton is responsive when it is ready to accept either a coin or a button can be pressed. The other events are not generated by the customer, but instead are driving actuators of the machine. The formula is a normal implementable for progress.

5.3 Applying the transformation steps

We now apply the four transformation steps to the CSP-OZ-DC specification in Fig. 10. The result of the manual step (1) is shown in Fig. 11. It shows the FDR-CSP specification using the input language for the FDR model-checker and representing the CSP-OZ part of the vending machine in Fig. 10. The constant declarations have now been instantiated. To keep the example small we restricted the maximum value of *Money* to 40, the set *Coin* to $\{10, 20\}$, and fixed the *price* at 20. The channel declaration and the CSP part are taken without modification. The OZ part is given as process that takes the complete state, here *m*, as a parameter. The CSP and the OZ part are then put in parallel synchronizing 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. 11. The result is displayed in Fig. 12. On the left-hand side are the states where the CSP part is still in its main process and the customer can insert money or press the button. The different nodes represent the different amounts of inserted money as indicated by the values of *m* in the graph. Note that the events which are blocked by the OZ part do not occur in the diagram. For example, the bottom most node, which represents the state where *m* has reached its maximum of 40, does not allow any *in* events. After the button is pressed the cup and coffee events are signalled and depending on the amount of inserted money the corresponding change is returned.

In the FDR model there is one state with an internal choice. In the diagram this is represented by labelling the node with two minimal acceptance sets. This

```
-- Constants
Money = {0..40}
Coin  = {10,20}
price = 20

-- Channels
channel in : Coin
channel out : Coin
channel button, coffee, cup, finished

-- Class CoffeeMachine
CoffeeMachine =
 let
   -- CSP part
   main  = in?x -> main [] Drink
   Drink = button -> cup -> coffee -> Return
   Return = out?y -> Return [] finished -> main

   -- OZ part
   OZPart(m) =
     ([] coin : {x| x<- Coin, member(m + x, Money)} @
         in.coin -> OZPart(m + coin))
     [] (let ReturnCoins = {x| x<- Coin, member(m - x, Money)}
         within ReturnCoins != {} &           -- Pre(out)
                |~| coin : ReturnCoins @
                    out.coin -> OZPart(m - coin))
     [] m >= price &                          -- Pre(button)
        button -> OZPart(m - price)
     [] m == 0 &                              -- Pre(finished)
        finished -> OZPart(m)

  within  --  Put CSP and OZ part in parallel
    main [| {| in, out, button, finished |} |] OZPart(0)
```

Fig. 11: FDR code for the Coffee Vending Machine

node represents the state where the customer inserts 40 and gets 20 back. The machine can choose whether it returns a 20 coin or a 10 coin (and a second 10 coin later). Semantically, each acceptance set represents a state of its own in which the automaton will only accept events from this set. By the closure properties of acceptance sets, there is – besides the minimal sets – also the full set containing all outgoing events as well as some intermediate sets that lie between the full set and a minimal set. For simplicity these sets are omitted from the diagram.

Since a timed automaton does not have the notion of acceptance sets, our tool

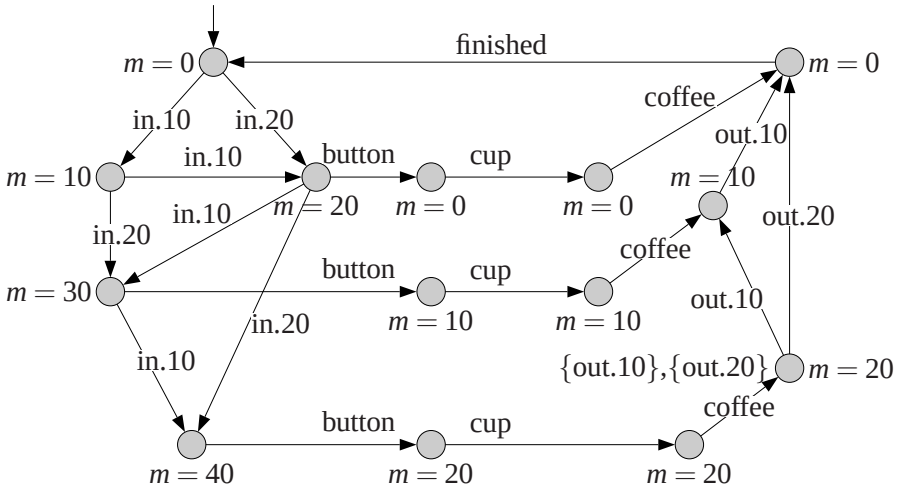


Fig. 12: Automaton generated by FDR

Fig. 13: Splitting nodes with more than one acceptance set

transforms it to make those sets more explicit. Every state having more than one acceptance set is split, so that each set belongs to exactly one state. The ingoing and outgoing transitions are copied as well. The next step is to remove outgoing transitions labelled with events that are not in the acceptance set. Afterwards the acceptance set of each state describes exactly the outgoing transitions of this state. This transformation process is illustrated in Fig. 13.

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 transitions. The resulting UPPAAL automaton is given in Fig. 14. In our example two clocks $c1$, $c2$ were introduced, one for each DC formula. The clock $c1$ measures the stability time for the first requirement. When a *cup* event is seen the *coffee* transition

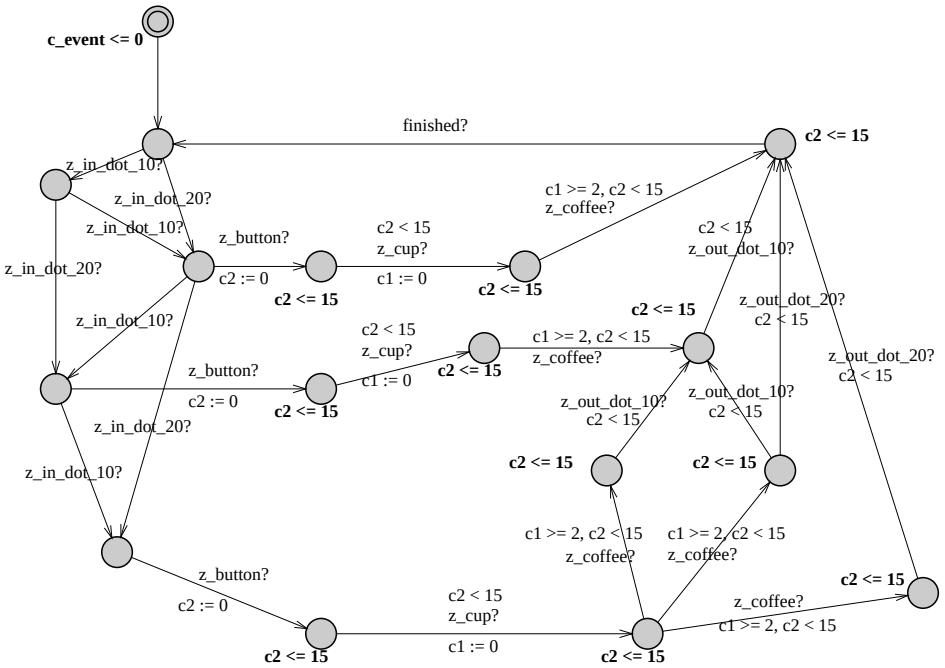


Fig. 14: UPPAAL Model of our CSP-OZ-DC class

should stay enabled for two seconds. The clock $c1$ is reset at all transitions that communicate the *cup* event and on the next *coffee* transition it is checked that the stability time has elapsed.

In general the algorithm first determines the set of states in which the *coffee* event is enabled. For each transition entering this set via a *cup* transition the clock $c1$ is reset and at each transition leaving the set the clock value is compared with 2. If there are transitions entering the state set without going through the *cup* event another copy of these states is created and the constraint is not checked for this copy.

For the progress constraint in the second formula the algorithm outlined in Fig. 9 is executed. First the clock $c2$ is added to the automaton. All states satisfying the condition $\neg\,(en(in) \vee en(button))$ are annotated with the invariant $c2 \leq 15$. All transitions that enter this set of states reset the clock. Finally all transitions from this set except for the *finished* transition are annotated with the guard $c2 < 15$.

The state with the double circle represents the starting state of the UPPAAL automaton. Its state invariant $c\_event \leq 0$ ensures that it will be left immediately to the real starting state of the FDR automaton. This construct is necessary when the real starting state is split as shown in Fig. 13 because UPPAAL needs a single starting state.

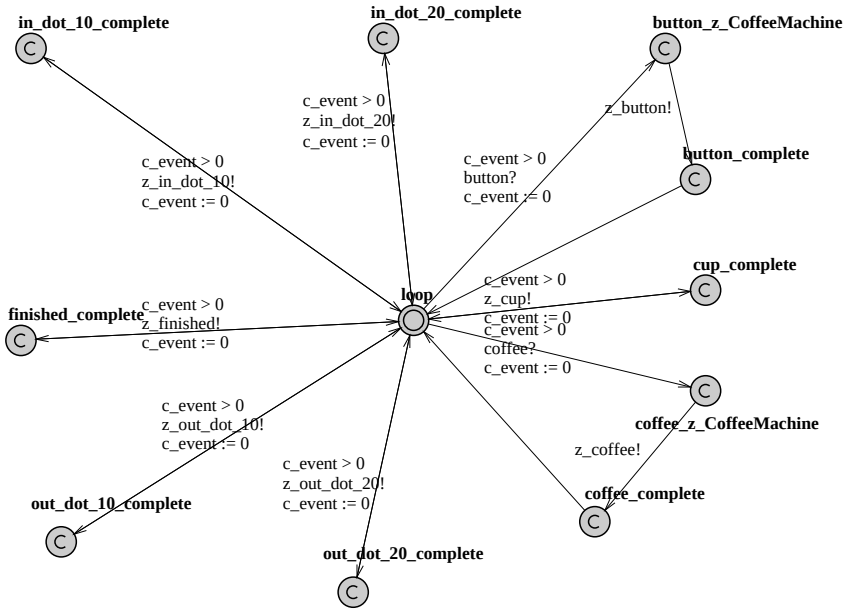


Fig. 15: Supervisor timed automaton

5.4 Model-Checking

As step (4) we consider the following real-time property of the vending machine: after pressing the *button* the customer will wait no longer than *t* seconds for the coffee. We wish to determine the exact value of *t* experimentally using UPPAAL.

The customer is represented by a small timed automaton *Test* with a clock *c_waiting* as shown in Fig. 16. It can be in one of two states, either *idle* or *waiting* for coffee. When the customer presses the *button* it changes to the *waiting* state and resets the clock *c_waiting*. This clock is used to measure the waiting time. When it gets the *coffee* it changes back to *idle* again.

The test automaton operates in parallel with the coffee machine as given in Fig. 14 and a *supervisor* automaton displayed in Fig. 15. The supervisor is the link between these two automata; it also handles the events not consumed by the test automaton. The supervisor starts in the double circled center state. When some event occurs one of the outer states annotated with C is entered. In UPPAAL this annotation marks so-called *committed* states that must be left immediately. Their purpose is to provide more information for the verification process. For instance, the external event *button* causes an internal event *z_button* to occur without any time passing in between. Another task of the supervisor is to ensure our condition that events cannot happen simultaneously (cf. subsection 3.3). To this end, it uses the clock *c_event* to check that time has passed ($c\_event > 0$) before the next event occurs, and resets this clock after each event.

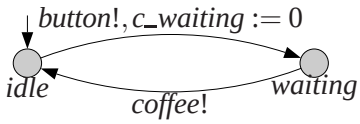


Fig. 16: Automaton representing a customer testing the behaviour

Number of coins	Max. value of m	Number of states	Step (2)	Step (3)	Step (4)
2	40	16	0.3	6.9	< 0.1
6	100	741	1.7	30.3	10.5
8	200	1822	3.9	180.6	62.1
8	400	4203	9.8	848.3	369.6

Fig. 17: Verification time (in seconds).

The whole system can be loaded into UPPAAL and the following property can be checked. It asserts that the customer is never waiting for his coffee for more than t seconds.

$$Prop_t \quad \Leftrightarrow \quad \mathrm{A}\,\Box\, Test.idle \vee c\_waiting \leq t$$

UPPAAL immediately returns an answer. It turns out that for $t = 15$ the property is satisfied, for smaller numbers it is not.

5.5 Experimental Results

The table in Fig. 17 gives some timings for the steps outlined at the start of this section. The times were measured on an UltraSPARC-II with 296 MHz. Step (2), which involves running FDR, is quite fast. Most time is spent in step (3), which is the addition of clocks to the automaton. It should be noted that our program has not been optimized.

We can easily change the OZ part to model systems of different size. The first line in the table is for the timed automaton shown in Fig. 14. Consider now larger systems with more coins and larger *Money* sets. As shown in Fig. 17, this yields automata with many more states. The tool scales better than quadratic to the number of states.

6. Conclusion

In the introduction we put forward guidelines for a good combination of specification techniques. The case study of the railway crossing was intended to show how the different aspects of the system can be specified conveniently in the dedicated specifications techniques for processes, data and time. The semantics of the combination was based on the principles of parallel composition and conjunction. This implied refinement compositionality and led us to reuse tools and verification

methods for a partially automatic verification. Another interesting consequence of our semantic integration of CSP, Object-Z and DC is that the representation of the individual specification techniques may be freely exchanged. For example, we may use *constraint diagrams* [Dietz 1996] to represent the DC formulas graphically and link to other development processes [Dierks and Tapken 2000].

Related work. The main idea of CSP-OZ-DC has been to combine three well-researched specification techniques in a constraint-oriented style. Closest to our approach to semantic integration is [Smith 2002] where Real-Time Object-Z [Smith and Hayes 1999] is integrated with CSP. The idea is that CSP operators serve to *combine* classes of Real-Time Object-Z, but in contrast to CSP-OZ-DC there is no CSP-part *inside* of classes. As we have seen in the examples, the CSP part is convenient for specifying sequencing constraints on the communications events.

In Real-Time Object-Z the timing properties are specified in an interval-based set-theoretic notation [Fidge *et al.* 1998]. We also use an interval-based approach but in terms of the Duration Calculus [see Zhou *et al.* 1991, Hansen and Zhou 1997]. To ensure that the CSP operators are well defined on Real-Time Object-Z classes, an $\mathcal{FD}$-semantics for these classes based on timed events is defined in [Smith 2002].

Another related work is TCOZ, a combination of Timed CSP [Davies and Schneider 1995] with Object-Z due to Mahony and Dong [1998, 1999]. As in CSP-OZ-DC the CSP operators are allowed both inside and outside of classes. However, here Timed CSP with its operational constructs like waits, timeouts and interrupts is used. By contrast, CSP-OZ-DC uses the predicates of DC to specify time dependencies between communications.

Verification. This paper goes beyond the above approaches by addressing verification. We have exploited the transformational semantics of CSP-OZ-DC for a partially automatic verification of timing properties of combined specifications. The core of the method is 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 of Section 5 is *equivalent* to the DC semantics of Section 3. A proof of such an equivalence is left for future work. However, similar equivalence proofs between timed automata and DC semantics are given in [Dierks *et al.* 1998].

Perspectives. Automatic verification works only for finite data types in the OZ 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 a separate (yet unpublished) case study we have applied the theorem prover KIV [Balser *et al.* 1999] to prove refinement of the OZ parts in a high-level specification by an implementation-level specification of a train controller for the radio controlled railway crossing.

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 OZ part is made observable by a special communication.

Acknowledgements

We would like to thank Christian Ohler, a research student at our group, for implementing the algorithm transforming FDR transition systems and DC patterns into UPPAAL timed automata. Detailed comments of the referees helped to improve the presentation.

References

BALSER, M., REIF, W., SCHELLHORN, G., AND STENZEL, K. 1999. KIV 3.0 for Provably Correct Systems. In *Applied Formal Methods – FM-Trends 98*, Volume 1641 of *LNCS*. Springer.

BENGTSSON, J., LARSEN, K.G., LARSSON, F., PETTERSSON, P., AND YI, WANG. 1997. Uppaal – a Tool Suite for Automatic Verification of Real-Time Systems. In *Hybrid Systems III – Verification and Control*, Volume 1066 of *LNCS*. Springer, 232–243.

BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. 1999. *The Unified Modeling Language User Guide*. Object Technology Series. Addison Wesley.

DAVIES, J. AND SCHNEIDER, S. 1995. A brief history of Timed CSP. *Theoretical Computer Science 138*, 243–271.

DE NICOLA, R. AND HENNESSY, M. 1983. Testing equivalences of processes. *Theoretical Computer Science 34*, 83–133.

DIERKS, H. 2001. PLC-Automata: A New Class of Implementable Real-Time Automata. *Theoretical Computer Science 253*, 1, 61–93.

DIERKS, H., FEHNKER, A., MADER, A., AND VAANDRAGER, F.W. 1998. Operational and Logical Semantics for Polling Real-Time Systems. In *FTRTFT'98*, Volume 1486 of *LNCS*. Springer, 29–40.

DIERKS, H. AND TAPKEN, J. 2000. Modelling and Verifying of a 'Cash Point Service' Using MOBY/PLC. *Formal Aspects of Computing 12*, 220–221.

DIETZ, C. 1996. Graphical formalization of real-time requirements. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Volume 1135 of *LNCS*. Springer, 366–385.

FIDGE, C.J., HAYES, I.J., MARTIN, A.P., AND WABENHORST, A.K. 1998. A Set-Theoretic Model for Real-Time Specification and Reasoning. In *Mathematics of Program Construction*, Volume 1422 of *LNCS*. Springer, 188–206.

FISCHER, C. 1997. CSP-OZ: A Combination of Object-Z and CSP. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, Volume 2. Chapman & Hall, 423–438.

FISCHER, C. 2000. *Combination and Implementation of Processes and Data: From CSP-OZ to Java*. PhD thesis, Bericht Nr. 2/2000, University of Oldenburg.

FISCHER, C. AND WEHRHEIM, H. 1999. Model-checking CSP-OZ specifications with FDR. In *Integrated Formal Methods (IFM 99)*. Springer, 315–334.

FORMAL SYSTEMS (EUROPE) LTD. 1995. *Failures-Divergence Refinement: FDR 2*.

HANSEN, M.R. AND ZHOU, C. 1997. Duration Calculus: Logical Foundations. *Formal Aspects of Computing 9*, 283–330.

HE, J., HOARE, C.A.R., FRÄNZLE, M., MÜLLER-OLM, M., OLDEROG, E.-R., SCHENKE, M., HANSEN, M.R., RAVN, A.P., AND RISCHEL, H. 1994. Provably correct systems. In *Formal Techniques in Real-Time and Fault Tolerant Systems*, Volume 863 of *LNCS*. Springer, 288–335.

HEILMANN, S. 1999. *Proof Support for Duration Calculus*. PhD thesis, Dept. Inform. Technology, Tech. Univ. Denmark.

HENNESSY, M. 1988. *Algebraic Theory of Processes*. MIT Press.

HOARE, C.A.R. 1978. Communicating Sequential Processes. *CACM 21*, 666–677.

HOARE, C.A.R. 1985. *Communicating Sequential Processes*. Prentice Hall.

HOARE, C.A.R. AND HE, J. 1997. *Unifying Theories of Programming*. Prentice Hall.

HOENICKE, J. AND OLDEROG, E.-R. 2002. Combining Specification Techniques for Processes, Data and Time. In *Integrated Formal Methods (IFM 2002)*, Volume 2335 of *LNCS*. Springer, 245–266.

KOLYANG. 1997. *HOL-Z — An Integrated Formal Support Environment for Z in Isabelle/HOL*. PhD thesis, Fachbereich Math. und Informatik, Univ. Bremen.

LUENBERGER, D.G. 1979. *Introduction to Dynamic Systems. Theory, Models & Applications*. Wiley.

MAHONY, B.P. AND DONG, J.S. 1998. Blending Object-Z and Timed CSP: an introduction to TCOZ. In *The 20th International Conference on Software Engineering (ICSE'98)*. IEEE Computer Society Press, 95–104.

MAHONY, B.P. AND DONG, J.S. 1999. Sensors and Actuators in TCOZ. In *FM'99 – Formal Methods*, Volume 1709 of *LNCS*. Springer, 1166–1185.

MOSZKOWSKI, B. 1985. A temporal logic for multi-level reasoning about hardware. *IEEE Computer 18*, 2, 10–19.

MOSZKOWSKI, B. 1986. *Executing Temporal Logic Programs*. Cambridge Univ. Press.

OLDEROG, E.-R., RAVN, A. P., AND SKAKKEBÆK, J. U. 1996. Refining system requirements to program specifications. In *Formal Methods for Real-Time Computing*. Wiley, 107–134.

RAVN, A.P. 1995. Design of Embedded Real-Time Computing Systems. Tech. Report ID-TR: 1995-170, Tech. Univ. Denmark.

RAVN, A.P., RISCHEL, H., AND HANSEN, K.M. 1993. Specifying and verifying requirements of real-time systems. *IEEE Trans. Software Engineering 19*, 1, 41–55.

ROSCOE, A.W. 1994. Model-checking CSP. In *A Classical Mind — Essays in Honour of C.A.R.Hoare*. Prentice-Hall, 353–378.

ROSCOE, A.W. 1997. *The Theory and Practice of Concurrency*. Prentice-Hall.

SAALTINK, M. 1997. The Z/EVES system. In *ZUM'97*, Volume 1212 of *LNCS*. Springer, 72–88.

SCHENKE, M. AND OLDEROG, E.-R. 1999. Transformational design of real-time systems – Part 1: from requirements to program specifications. *Acta Inform. 36*, 1–65.

SELIC, B. AND RUMBAUGH, J. 1998. Using UML for Modeling Complex Real-Time Systems. Tech. report, ObjecTime.

SKAKKEBÆK, J.U. 1994. *A Verification Assistent for a Real-Time Logic*. PhD thesis, Dept. Comp. Sci., Tech. Univ. Denmark.

SMITH, G. 2000. *The Object-Z Specification Language*. Kluwer Academic Publisher.

SMITH, G. 2002. An Integration of Real-Time Object-Z and CSP for Specifying Concurrent Real-Time Systems. In *Integrated Formal Methods (IFM 2002)*, Volume 2335 of *LNCS*. Springer, 267–285.

SMITH, G. AND HAYES, I. 1999. Towards Real-Time Object-Z. In *Integrated Formal Methods (IFM 99)*. Springer, 49–65.

SMITH, G., KAMMÜLLER, F., AND SANTEN, T. 2002. Encoding Object-Z in Isabelle/HOL. In *ZB 2002: Formal Specification and Development in Z and B*, Volume 2272 of *LNCS*. Springer, 82–99.

SPIVEY, J.M. 1992. *The Z Notation: A Reference Manual*, 2nd edition. Prentice-Hall International Series in Computer Science.

TAPKEN, J. 2001. *Model-Checking of Duration Calculus Specifications*. PhD thesis, Bericht Nr. 3/2001, University of Oldenburg.

WOODCOCK, J. AND DAVIES, J. 1996. *Using Z — Specification, Refinement, and Proof*. Prentice-Hall.

ZHOU, C., HOARE, C.A.R., AND RAVN, A.P. 1991. A calculus of durations. *Information Processing Letters 40*, 5, 269–276.