

AVACS – Automatic Verification and Analysis of
Complex Systems

REPORTS
of SFB/TR 14 AVACS

Editors: Board of SFB/TR 14 AVACS

From High-Level Verification to Real-Time
Scheduling: A Property-Preserving Integration

by
Johannes Faber Ingo Stierand

Publisher: Sonderforschungsbereich/Transregio 14 AVACS
(Automatic Verification and Analysis of Complex Systems)
Editors: Bernd Becker, Werner Damm, Martin Fränzle, Ernst-Rüdiger Olderog,
Andreas Podelski, Reinhard Wilhelm
ATRs (AVACS Technical Reports) are freely downloadable from www.avacs.org

Copyright © June 2007 by the author(s)
Author(s) contact: Ingo Stierand (Ingo.Stierand@Informatik.Uni-Oldenburg.DE).

From High-Level Verification to Real-Time Scheduling: A Property-Preserving Integration^{*}

Johannes Faber and Ingo Stierand

Department of Computing Science, University of Oldenburg, Germany
{johannes.faber|ingo.stierand}@informatik.uni-oldenburg.de

Abstract. In the design process of real-time systems, formal verification establishes global properties of high-level specifications while real-time scheduling analysis ensures that concrete realisations meet essential timing properties with respect to a given target platform. But a formal link between these phases is missing. It is unclear (1) whether timing assumptions that are required to verify properties of high-level specifications can actually be realised on a target platform and (2) whether verified properties remain valid for a schedulable task network. Our approach bridges this gap by guaranteeing that properties verified on specification level are preserved on the implementation level, and vice versa, schedulability results can be propagated back to the specification. To this end, we provide a property-preserving translation from a subclass of the high-level real-time language CSP-OZ-DC into Cyclic Timed Automata, a Timed Automata based task network formalism. We apply our method to a case study from the European Train Control System standard.

1 Introduction

Formal verification methods are widely used in the design process for complex real-time systems. Starting from formal requirements, high-level specification languages are an appropriate tool to specify systems during the *design phase* [25,18,2,22]. Formal verification methods, e.g., model checking, are applied to ensure that the specification complies with the requirements [1,4]. In the *implementation phase*, the specification is realised on a target platform, e.g., by automatic code generation. Real-time scheduling analysis [17,28,8] is applied to ensure that such a concrete implementation of the system is feasible on a hardware platform: it is verified that the underlying timing assumptions are met.

Thus, we have the situation that there are well-investigated methods on the specification level to model and verify real-time systems and likewise well-investigated methods considering realisations of systems on specific platforms. But often the formalisms of these disciplines are incompatible and a formal link between them is missing: real-time properties can be verified on specification

^{*} This work was partly supported by the German Research Council (DFG) under grant SFB/TR 14 AVACS. See <http://www.avacs.org> for more information.

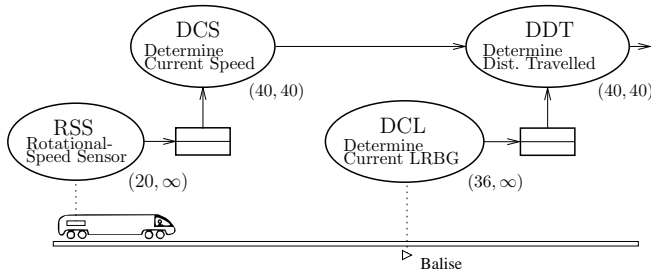


Fig. 1: Odometer unit in the ETCS

level but it is unclear (1) to what extent these results remain valid on implementation level, and (2) whether timing assumptions that are necessary to verify the specification can actually be established on a target platform.

Our approach bridges this gap by guaranteeing that properties verified on specification level are preserved on implementation level, and vice versa, by establishing the feasibility of the specification’s timing assumptions via existing real-time scheduling analysis techniques. To this end, we provide a translation from a subclass of the high-level specification language CSP-OZ-DC (COD) [12] into Cyclic Timed Automata (CTA) [26], a Timed Automata based task network model. The subset is chosen to match the expressiveness of the underlying task network model. Assuming a correct code generation for single COD tasks to executable code¹, we prove that our translation preserves real-time properties verified on specification level; if the translation of a COD specification results in a schedulable task network, our approach guarantees that timing assumptions of the specification, needed for high-level verification to ensure global properties, are met by the implementation. We illustrate our approach using an odometer unit as specified in the European Train Control System (ETCS) standard [5].

Our main contributions are (1) the definition of a COD subclass that can be translated into task networks, (2) a translation algorithm that generates task networks in terms of CTA from our COD subclass, (3) a proof that our translation results in a task network that—if it is schedulable—preserves properties of our high-level specification formalism, and (4) a small example from the ETCS. **Outline.** The remaining Sect. 1 gives an overview of the approach and reviews related work. In Sect. 2 we recall the underlying formalisms, COD and CTA. While in Sect. 3 a COD subclass for task specifications is introduced, Sect. 4 gives its translation to CTA and our example’s results. Section 5 concludes.

1.1 Motivating Example

We consider an odometer as specified in the European Train Control System (ETCS) standard as running example throughout the paper. In the final ETCS level, the trains’ positions are autonomously measured and reported. Hence, the system’s safety crucially depends on the odometer measuring position and speed

¹ This is not subject of this work.

of a train. The odometer computes a confidence interval taking the error of measurement into account. The scenario is sketched in Fig. 1. The train measures its movement using a *Rotational-Speed Sensor* (RSS) and stores the results in a shared resource. The speed is computed by a task *Determine Current Speed* (DCS). At irregular intervals, there are *balise groups* on the track, calibrating the position of the train. The odometer determines (DCL) the *last relevant balise group* (LRBG). Finally, the odometer uses these values to determine (DDT) position and a confidence interval specifying the assumed accuracy of the calculated position. A basic safety requirement is that the actual position of the train is not illegally outside the calculated confidence interval.

The first step in the design process is to specify the system using a high-level specification language. Systems like the odometer are determined by different aspects: control flow, data and state changes, and timing aspects. To cope with this diversity, [13] introduced COD, a combined language that integrates three well-investigated formalisms, CSP [11], Object-Z [25], and Duration Calculus (DC) [29]. We consider COD as a high-level language because it allows for modelling the identified dimensions in an *object-oriented, declarative, and compositional* way, incorporating complex and infinite data types—in contrast to the operational Timed Automata model. In particular, COD specifications can be verified against safety requirements given in terms of DC formulae [22,6].

Later in the design process, the specification has to be implemented. In order to verify the feasibility of the system on the designated hardware platform (in our case the train odometer), the specification is translated into a task network that can be handed over to scheduling analysis.

To ensure the overall system safety, verified properties have to be preserved for the implementation. That is we have to show, if (1) the COD specification fulfils a requirement ϕ and if (2) a task network for this specification is feasible on a certain machine M , then the property ϕ also holds for the implementation (cf. Fig. 2). In order to obtain

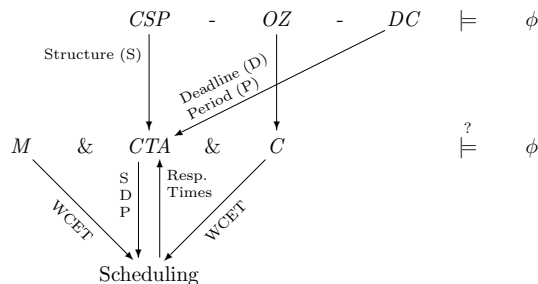


Fig. 2: Overview on the Approach

a common semantical basis for COD and task networks, we apply Cyclic Timed Automata (CTA) [26], a formalism to specify task networks in terms of Timed Automata [1]. The task network is generated from the control structure (CSP part) and the timing properties (DC part) of the specification. The data aspects (OZ part) constitute the executable code C for the tasks. We assume the generated code to comply with the OZ specification. Since a translation is not possible for arbitrary COD specifications, we provide a subclass that restricts the use of timing constraints and the control part. We prove that the translation is property-preserving by relating the trace semantics of COD and CTA.

1.2 Related Work

The work closest to ours is that of Burns and Lin [3]: They developed an engineering process that links different stages of the design process for real-time systems. Contrary to ours, their approach is not completely formal. Moreover, while they apply Timed Automata as underlying formalism, our approach benefits from the use of a high-level formalism with complex and infinite data types.

Model checking and scheduling analysis for scheduling client-server systems are combined in [15], but they only apply model checking on task network level—there is no established connection to high-level verification. Related work regarding correct implementations of high-level real-time specifications can be found, e.g., in [21,18,2,4]. For instance, [4] addresses code generation for real-time specifications in terms of PLC automata. But it is not checked whether specified timing assumptions can always be met. These approaches have in common that schedulability is not analysed.

Approaches to formal synthesis of real-time software considering scheduling are, e.g., [19,24,14]. All of these generate code from a formal Petri Net specification. In each case, they apply only specific static scheduling algorithms on single processor systems, in contrast to the CTA framework. Moreover, they do not prove that high-level properties are preserved.

Related work on Timed Automata based scheduling techniques can be found, e.g., in [7,8]: variants of Timed Automata are used to analyse fixed priority scheduled tasks. In [20] a framework for distributed systems has been presented. In contrast to the above, the CTA framework does not model the underlying scheduling algorithm as Timed Automata. Instead, it utilises efficient „traditional” scheduling analysis techniques such as [28] for the verification of schedulability, based on a compliance proof to the formalism of *demand bound functions* [27]. It bounds execution demands for the tasks in terms of functions over time intervals. Schedulability is expressed as a functional calculus problem.

2 Preliminaries

2.1 CSP-OZ-DC

CSP-OZ-DC (COD) was introduced in [13,12] and combines three formalisms, *Communicating Sequential Processes* [11], *Object-Z* [25], and *Duration Calculus* [29], allowing declarative specifications of a system’s control flow, data structures, and timing. Figure 3 displays a simple COD specification for our odometer example of Sect. 1.1. Each COD specification begins with an interface part (first line) declaring methods that are provided by the specification. In the following, we give an intuition of the individual parts of a COD specification.

CSP. The control flow of methods is described by CSP [11]. CSP is used to model parallel and sequential processes in terms of the events they communicate. Events always occur in instantaneous time. We denote the set of events **Events**. The CSP syntax is defined by the following BNF grammar: $P ::= \text{Stop} \mid \text{Skip} \mid$

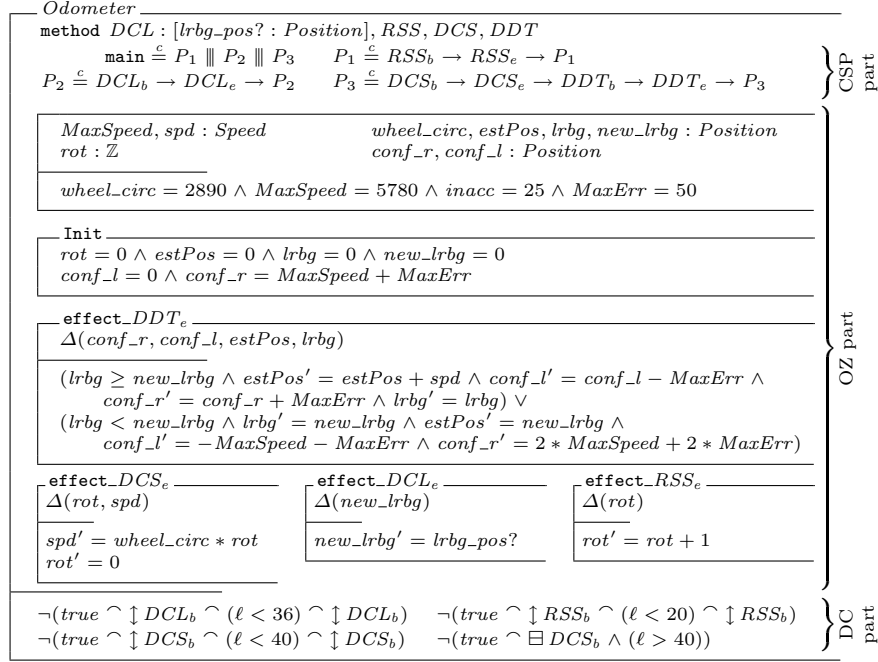


Fig. 3: COD specification for the odometer

$a \rightarrow P \mid P_1 \square P_2 \mid P_1 \parallel_A P_2 \mid P_1 \parallel P_2 \mid P_1 ; P_2 \mid X$. The **main** process in the odometer specification of Fig. 3 consists of an interleaving (\parallel) of three simple subprocesses. For instance, subprocess P_1 states that an RSS_b event, denoting the start of task RSS (cf. Fig. 1), is followed by a RSS_e event, denoting the task's completion. Then P_1 is restarted again. The remaining processes represent a diverging process (**Stop**), a terminating process (**Skip**), choice (\square), and sequential composition ($;$). Finally, X stands for a process identifier, which must be declared with an expression $X \stackrel{c}{=} P$.

OZ. Object-Z [25] is an object-oriented extension for Z. We use *OZ schemas* to define the state space over variables Var of a COD specification, and possible state changes. The first OZ schema in Fig. 3 defines the state space with invariants over the state variables, e.g., $wheel_circ$ contains a constant value for the train's wheel circumference. The **Init** schema defines constraints over the system's initial state. State changes are defined in a constraint-based declarative way by **effect** schemas. The schemas are associated to events from the CSP part such that a state change is always performed at the same time when a CSP event is triggered. For example, the effect of the *determine current speed* task, **effect** $_{DCS_e}$, changes the variables rot (counting the wheel rotations) and spd (the speed), which is indicated by the Δ expression. The unprimed variables in the constraints refer to the pre-state while the primed variant refers to the post-state; the new value of spd is computed by $wheel_circ * rot$, where rot is the old rotation value – the new value is set to zero. The schema **effect** $_{DDT_e}$ com-

putes the new estimated position $estPos$ and the confidence interval $(conf_l, conf_r)$ depending on the last relevant balise group $lrbg$.

DC. Timing properties are modelled with Duration Calculus (DC) [29], an interval based real-time logic using a dense time domain $\mathbb{T} := \mathbb{R}^+$. Since the full DC is undecidable, we restrict ourselves to counter-example traces [12]. The first DC formula in Fig. 3 states that there will never be an interval of length lower than 36 milliseconds ($\ell < 36$) between two DCL_b events (\uparrow). The second and third formulae are analogous. The last formula specifies an upper bound: there will be no interval of length greater than 40 ($\ell > 40$) without a DCS_b event (\boxminus).

Semantics. In [12], the operational semantics of COD specifications is given in terms of timed traces (originally defined via Phase Event Automata, a Timed Automata model [1] involving data aspects). A timed trace is an infinite sequence of configurations $\langle (e_1, \beta_1, t_1), (e_2, \beta_2, t_2), \dots \rangle$, with events e_i , variable valuations β_i , and points in time $t_i \in \mathbb{T}$, $t_i > 0$, for $i \in \mathbb{N}$. The semantics is compositional in that every part of the specification is translated separately; for a COD specification Z with the appropriate parts Z_{CSP} , Z_{OZ} , Z_{DC} the semantics is given by $\llbracket Z \rrbracket := \llbracket Z_{CSP} \rrbracket \cap \llbracket Z_{OZ} \rrbracket \cap \llbracket Z_{DC} \rrbracket$, where $\llbracket Z_{CSP} \rrbracket$ is the trace semantics given by the structured operational semantics of CSP [23], $\llbracket Z_{OZ} \rrbracket$ is defined s.t. variable valuations β are restricted according to the changes in the operation schemas of the OZ part, and $\llbracket Z_{DC} \rrbracket$ contains trace representations of the DC formulae’s interpretations. For details we refer to [12].

For the chosen subclass of DC, COD model checking is possible [22]. For a formula ϕ of this subclass we consider the set $\llbracket \phi \rrbracket$ of traces satisfying ϕ . We write $Z \models \phi$ iff $\llbracket Z \rrbracket \subseteq \llbracket \phi \rrbracket$.

2.2 Task Networks and Cyclic Timed Automata

The common model for scheduling analysis is a task network. Each task represents a piece of code, and has parameters assigned such as the execution time, depending on the code, and an activation pattern, like periodic or sporadic activations with some period P . The result of scheduling analysis are worst case response times for the tasks, depending on the underlying scheduling schema, e.g., fixed priority scheduling. The analysis results in a feasible task network if all response times are less or equal to deadlines assigned to each task.

The aim of CTA [26] is to combine a formal semantics for task networks with efficient scheduling analysis techniques [16]. The CTA framework comprises a number of Timed Automata templates, that can be instantiated and parameterised in order to constitute task networks. A CTA component consists of one or more Timed Automata. A set of interface automata define the visible input and output behaviour of the component. The internal timing behaviour is defined by a core automaton, which models, e.g., response times for tasks. Components are connected by synchronisation via their interface automata. Figure 4 depicts the component types used in this paper, except for *Triggers* and *Sinks*. They consist only of an interface automaton, and are used to model environmental tasks. *Execute* components are used to model tasks. The input interface automaton models

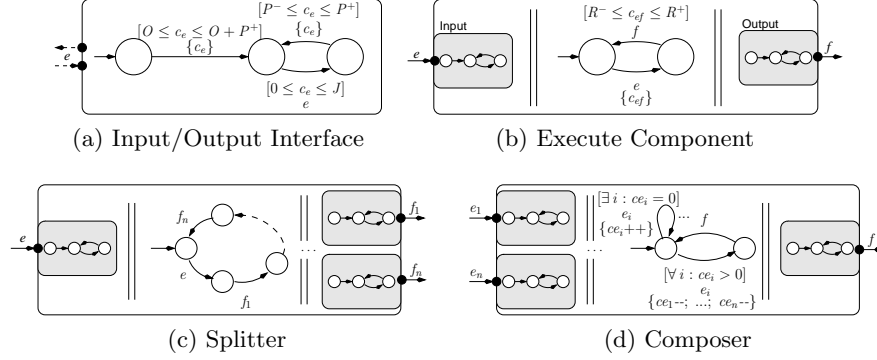


Fig. 4: CTA components

the activation behaviour, and the core automaton models the delay caused by the execution of the component. This delay is bounded by the minimum and maximum response times (R^- , R^+) as they are calculated by scheduling analysis. The output interface defines the resulting completion behaviour. *Splitter* and *Composer* are used to split and combine event flows.

Temporal behaviour of a CTA interface is defined by a tuple (P^-, P^+, J, O) , where P^- and P^+ are the minimal and maximal periods of the event stream, and J is the maximum jitter. Periodic activations are modelled by setting $P^- = P^+ = P$. Finally, the parameter O defines the start time of the component. Its value depends on the output behaviour of a preceding component the interface is connected to. In the following, we define CTA and their trace semantics.

Definition 1 (CTA component).

1. An interface automaton $In(e, P^-, P^+, J, O) / Out(e, P^-, P^+, J, O)$ is the automaton depicted in Fig. 4a. The semantics of In and Out is

$$\llbracket Out \rrbracket = \llbracket In \rrbracket = \{ \langle (e, t_0 + \delta_0), (e, t_1 + \delta_1), (e, t_2 + \delta_2), \dots \rangle \mid 0 \leq \delta_i \leq J \\ O \leq t_0 \leq O + P^+, t_{i-1} + P^- \leq t_i \leq t_{i-1} + P^+ \}.$$

If the context is obvious we use the abbreviations In_e and Out_e .

2. A trigger $Q(Out(e, P^-, P^+, J, O))$ is a automaton which semantics semantics $\llbracket Q \rrbracket$ is equal to $\llbracket Out(e, P^-, P^+, J, O) \rrbracket$.

3. A sink $S(In(e, P^-, P^+, J, O))$ is a automaton which semantics $\llbracket S \rrbracket$ is equal to $\llbracket In(e, P^-, P^+, J, O) \rrbracket$.

4. An execute component $E(In(e, P^-, P^+, J, O), f, R^-, R^+)$ is the automaton pictured in Fig. 4b. Its semantics $\llbracket E \rrbracket$ is

$$\llbracket E \rrbracket = \{ \langle (e, t_0 + \delta_0), (f, t_0 + \delta_0 + R_0), (e, t_1 + \delta_1), (f, t_1 + \delta_1 + R_1), \dots \rangle \\ \mid 0 \leq t_0 \leq O + P^+, t_{i-1} + P^- \leq t_i \leq t_{i-1} + P^+, \\ 0 \leq \delta_i \leq J, R^- \leq R_i \leq R^+ \}.$$

5. A splitter component $D(In_e, Out_{f_1}, \dots, Out_{f_n})$ is a automaton as depicted in Fig. 4c, with the following semantics:

$$\begin{aligned} \llbracket D \rrbracket = \{ & \langle (e, t_0), (f_1, t_0), \dots, (f_n, t_0), (e, t_1), (f_1, t_1), \dots \rangle \\ & \mid \langle (e, t_0), (e, t_1), \dots \rangle \in \llbracket In(e, P^-, P^+, J, O) \rrbracket \}. \end{aligned}$$

6. A composer $C(In_{e_1}, \dots, In_{e_n}, Out_f)$ is a automaton as depicted in Fig. 4d, with the following semantics:

$$\begin{aligned} \llbracket C \rrbracket = \{ & \langle (e_1, t_{1,0}), \dots, (e_n, t_{n,0}), (f, u_0), (e_1, t_{1,1}), \dots, (e_n, t_{n,1}), (f, u_1), \dots \rangle \\ & \mid \forall_{i=1, \dots, n} \langle (e_i, t_{i,0}), (e_i, t_{i,1}), \dots \rangle \in \llbracket In_{e_i} \rrbracket, \forall_{j \in \mathbb{N}} u_j = \max_{i=1, \dots, n} (t_{i,j}) \}. \end{aligned}$$

The definitions for In and Out are identical because the original definition for Timed Automata does not distinguish between sending and receiving events.

CTA components are combined to *CTA networks* by parallel composition. A CTA network is a tuple $(Events, A \subset CTA)$, where CTA is the set of all CTA components, and $Events$ is a set of common events. For each $e \in Events$ there are two unique CTA components $src, dest \in A$ with interfaces synchronising via e . The semantics of CTA is defined as $\llbracket CTA \rrbracket^- := \llbracket \llbracket X \in A \rrbracket X \rrbracket$.

Yet, task networks have no notion of data variables and valuations. Hence, in order to compare the traces of task networks and COD, we extend CTA traces $\langle (e_1, t_1), (e_2, t_2), \dots \rangle \in \llbracket CTA \rrbracket^-$ with valuations of variables: $\langle (e_1, \beta_1, t_1), (e_2, \beta_2, t_2), \dots \rangle \in \llbracket CTA \rrbracket$. Additionally, we define the implementation, feasibility, and correctness of an implementation.

Definition 2 (Implementations). *Let M be a machine, defined by a set of possible traces $\llbracket M \rrbracket$, and C be executable code, defined by $\llbracket C \rrbracket$. An implementation is then given by a task network CTA , code C for this tasks, and by M , written as (CTA, C, M) . We define its feasibility and correctness w.r.t. a formula φ :*

$$\begin{aligned} (CTA, C, M) \models \text{feasible} & \quad :\Leftrightarrow \llbracket CTA \rrbracket \cap \llbracket C \rrbracket \cap \llbracket M \rrbracket \neq \emptyset \\ (CTA, C, M) \models \varphi & \quad \quad \quad :\Leftrightarrow \llbracket CTA \rrbracket \cap \llbracket C \rrbracket \subseteq \llbracket \varphi \rrbracket \text{ and } (CTA, C, M) \models \text{feasible} \end{aligned}$$

3 Task Specifications

Since task networks have a simple and firm structure, it is not possible to translate arbitrary COD specifications into task networks. Hence, we introduce *Task Specifications*, a COD subclass that admits only those CSP and DC constructs that can be translated into task networks in a semantics preserving way. As our translation refers to initial and terminating events of CSP processes, we begin by defining functions yielding these events.

Definition 3. *The function $first$ returns all possible initial events occurring in a CSP process P . The functions $next(P, a)$ and $last(P, a)$ return the unique next and last event, respectively, or a default event.*

$$\begin{aligned} first(P) & := \begin{cases} \{e\} & \text{if } P \equiv e \rightarrow P' \\ first(P_1) & \text{if } P \equiv P_1 ; P_2 \\ first(P_1) \cup first(P_2) & \text{if } P \equiv P_1 \mid P_2 \text{ with } \mid \in \{\parallel, \square\} \end{cases} \\ next(P, a) & := \begin{cases} e & \text{if } P \equiv e \rightarrow P' \\ a & \text{else} \end{cases} \end{aligned}$$

$$\text{last}(P) := \begin{cases} e & \text{if } P \equiv e \rightarrow \text{Skip} \\ \text{last}(P') & \text{if } P \equiv e \rightarrow P' \text{ with } P' \neq \text{Skip} \\ \text{last}(P_1) \cup \text{last}(P_2) & \text{if } P \equiv P_1 \text{ op } P_2 \end{cases}$$

Now we proceed with the definition of the *Task Specifications* subclass.

Definition 4 (Task Specification). A Task Specification Z for a set of tasks \mathbb{T} is a COD specification with respective parts Z_{CSP} , Z_{OZ} , and Z_{DC} , with the following properties:

1. For every task $a \in \mathbb{T}$ the specification Z contains events $a_b, a_e \in \text{Events}$, denoting the begin and the end of task a . The OZ part Z_{OZ} must not impose a state change on the occurrence of a_b .
2. The CSP part Z_{CSP} complies with the BNF

$$\begin{aligned} \text{TaskProcDecl} &::= \text{LoopProcDecl} \mid \text{TaskProcDecl} \parallel \text{TaskProcDecl} \\ \text{LoopProcDecl} &::= \mu X : \text{TermProc} ; X \\ \text{TermProc} &::= [a_t \rightarrow] a_b \rightarrow a_e \rightarrow \text{Skip} \mid [a_t \rightarrow] a_b \rightarrow a_e \rightarrow \text{TermProc} \\ &\quad \mid \text{TermProc} ; \text{TermProc} \mid \text{TermProc} \parallel \text{TermProc} \\ &\quad \mid \text{TermProc} \square \text{TermProc}, \end{aligned}$$

for tasks $a \in \mathbb{T}$. The event a_t is a special trigger event needed for timing constraints on the tasks.

3. For tasks $a, \tilde{a}, \hat{a} \in \mathbb{T}$, with $a \in \text{first}(\text{main})$, and $P^-, P^+, J \in \mathbb{T} \setminus \{0\}$, every DC formulae in Z_{DC} is in the form of one of the following pattern:

$$\begin{aligned} \mathbf{DC}_{P^-}(P^-, a) &:= \neg(\text{true} \wedge \downarrow a_x \wedge P^- > \ell \wedge \downarrow a_x) \\ \mathbf{DC}_{P^+}(P^+, a) &:= \neg(\text{true} \wedge \boxplus a_b \wedge P^+ < \ell) \\ \mathbf{DC}_J(J, a) &:= \neg(\text{true} \wedge \downarrow a_t \wedge \boxplus a_b \wedge J < \ell) \end{aligned}$$

where $a_x = a_t$ if there is a formula $\mathbf{DC}_J(J, a)$ and $a_x = a_b$ otherwise. Moreover, in order to get a well-behaved task network, periods and jitters have to be unique for each loop process declaration, i.e., if there are DC formulae $\mathbf{DC}_J(P_1, a_1)$ and $\mathbf{DC}_J(P_2, a_2)$ for $\text{lpd} \in \text{LoopProcDecl}$ with $a_1, a_2 \in \text{first}(\text{lpd})$ we demand $P_1 = P_2$, and analogously for \mathbf{DC}_{P^-} and \mathbf{DC}_{P^+} .

The ideas behind this definition are as follows.

1. In COD state changes are defined using operation schemas, which define the tasks of the system. An essential difference between the task and the COD view is that state changes induced by a OZ schema do not consume any time. Time elapses just between state changes, whereas a task is basically characterised by its execution time. Obviously, it is not possible to produce a code generator that can reflect this instantaneous behaviour of a COD specification. To solve this issue, we integrate a more realistic notion of a task in COD, where the call of a method is separated from the time when the execution is performed.

In Def. 4, we use a straight-forward approach, where each task is associated with two events: The first event a_b reads input parameters and the state space, while the second event a_e writes the output and the modified state space. A code generator has to take this behaviour into account. The obvious drawback of using two events is the increased effort for high-level verification. Note that it is not possible to use the build-in *double event methods* of COD: the method execution cannot be interrupted by other tasks, which is essential for scheduling.

2. Obviously, we cannot construct a finite task network for every CSP process, because there are some cases where the operational semantics of a CSP process gets infinitely many states. Further, task networks have a periodic structure. Thus, Def. 4 gives a subclass of periodic CSP processes that can be translated. The idea is to allow only loop processes (*LoopProcDecl*) that are defined over terminating processes (*TermProc*), i.e., processes that end with the **Skip** process. The fix-point expression $\mu X : TermProc ; X$ denotes a process that behaves like *TermProc* and restarts again after terminating.

3. The temporal behaviour of tasks is determined by sporadic or periodic task activation with a possible jitter. Therefore, Def. 4 restricts the use of DC formulae to pattern describing periods and jitters of tasks. The formulae \mathbf{DC}_{P^-} and \mathbf{DC}_{P^+} define lower (P^-) and upper (P^+) time bounds for the occurrences of an event a_t —indicating the activation of a task a with jitter—or the occurrences of an event a_b for a task without jitter. Intuitively, the formula \mathbf{DC}_{P^-} states that two occurrences of events are separated by an interval of a length not less than P^- . The second pattern \mathbf{DC}_{P^+} formulates the upper period bound: there will never be an interval of length greater than P^+ without the appropriate event. The jitter J of a task a can be specified with a formula $\mathbf{DC}_J(J, a)$: after an event a_t an event a_b has to follow within an interval of length less or equal to J . No other DC formulae are allowed in COD task specifications. If we can prove a property on COD level without timing constraints, this property is preserved on task level for arbitrary periods, jitters, and deadlines. In our running example, the first three DC formulae have the \mathbf{DC}_{P^-} pattern, the last is a \mathbf{DC}_{P^+} formula. There is no \mathbf{DC}_J pattern, i.e., *initial* tasks are defined without jitter.

Definition 4 does not include parallel composition with synchronisation. This is to simplify the construction of the task networks. Since we only consider a trace semantics of CSP here, we can represent parallel composition with interleavings.

4 Translation

In this section, we introduce a property-preserving translation from the COD specification subclass defined above into task networks in terms of CTA and state the correctness.

We consider a task specification Z with a set T of tasks according to Def. 4. We translate only the CSP and the DC part into task network, because a task network abstracts from data and state changes. We inductively define the construction over the structure of the CSP part. On the level of *TaskProcessDecl*, every interleaved *LoopProcDecl* is translated separately. The entire task net-

Table 1: Mapping from *TermProc* to CTA components

CSP expression	CTA component
Termination	
Skip	$CTA(TP, P^-, P^+, J, O, a, e) = \emptyset$
Task execution $TP = [a_t \rightarrow] a_b \rightarrow a_e \rightarrow P$	Execute component $CTA(TP, P^-, P^+, J, O, a, e) =$ $\{E(In(a_b, P^-, P^+, J, O), next(P, a_e), R^-, R^+)\}$ $\cup CTA(P, P^-, P^+, J(a), O(a), next(P, a_e), e)$
Interleaving $TP = P_1 \parallel P_2$	Splitter, Composer component $CTA(TP, P^-, P^+, J, O, a, e) =$ $\{D(a, next(P_1, a), next(P_2, a))\}$ $\cup CTA(P_1, P^-, P^+, J, O, next(P_1, a), e)$ $\cup CTA(P_2, P^-, P^+, J, O, next(P_2, a), e)$ $\cup \{C(last(P_1), last(P_2), e)\}$
Sequence $TP = P_1 ; P_2$	Execute component $CTA(TP, P^-, P^+, J, O, a, e) =$ $CTA(P_1, P^-, P^+, J, O, a, f_1)$ $\cup \{E(In(last(P_1, f_1), P^-, P^+, J(P_1), O(P_1)),$ $next(P_2, f_2), 0, 0)\}$ $\cup CTA(P_2, P^-, P^+, J(P_1), O(P_1), next(P_2, f_2), e)$

work consists of the parallel composition of the task networks translated from the respective subprocesses. A *LoopProcDecl* consists of a terminating process *TermProc* that is recalled after termination. The task network construction maps each CSP expression occurring in *TermProc* to a CTA component: every task in \mathbb{T} is mapped to an execute component, interleaving is realised via splitter and composer components, and a sequence is represented by a dummy execute component. To this end, we define a mapping from *TermProc* to CTA components that depends on periods, jitters, the offset, and on input and output events. It is given in Tab. 1. With this, we formally define the translation.

Definition 5 (Translation of Task Specifications to CTA). *The translation from a Task Specifications Z with CSP part Z_{CSP} and DC part Z_{DC} to CTA components, i.e., $CTA : TaskProcDecl \rightarrow CTA$, is defined inductively over the structure of Z_{CSP} . For $TPD_1, TPD_2 \in TermProcDecl \setminus TermProc$ and $TP \in TermProc$:*

$$CTA(TPD_1 \parallel TPD_2) := CTA(TPD_1) \cup CTA(TPD_2)$$

$$CTA(\mu X : TP ; X) := \{Q(In_a), S(Out_e)\} \cup CTA(TP, P^-, P^+, J, 0, a, e),$$

where $a = next(TP, f_1)$, $e = last(TP, f_1)$, for new events f_1, f_2 , and Z_{DC} contains formulae $\mathbf{DC}_{P^-}(P^-, a)$, $\mathbf{DC}_{P^+}(P^+, a)$, and $\mathbf{DC}_J(J, a)$ or $J = 0$. The CTA components for *TermProc* processes are given by a mapping $CTA : TermProc \times \mathbb{T} \times \mathbb{T} \times \mathbb{T} \times \mathbb{T} \times \mathbf{Events} \times \mathbf{Events} \rightarrow CTA$ which is inductively defined in Tab. 1, where $J(a) = J + R^+(a) - R^-(a)$, $O(a) = O + R^-(a)$.

Currently, CTA do not support choice. Hence, we do not create CTA that reflect the behaviour of choice $P_1 \square P_2$. Instead, we apply the CTA generation and the scheduling analysis twice, for P_1 and P_2 , respectively.

4.1 Correctness

In order to show that the task network of a COD specification Z preserves properties on implementation level, we start by recapitulating and formalising the problem. First, we assume that Z satisfies a DC property φ , which is, e.g., proven by model checking [22]. We then apply our construction from Sect. 4 to obtain a task network CTA . Together with task code C , it realises the specification Z on some hardware M . The question is if (CTA, C, M) is a correct implementation of φ on a machine M (cf. Def. 2):

$$Z \models \varphi \quad \stackrel{?}{\Rightarrow} \quad (CTA, C, M) \models \varphi.$$

In order to prove this, we apply scheduling analysis and assume that the COD specification Z is feasible on M : $(CTA, C, M) \models \text{feasible}$. Since code generation is not subject of this work, we assume the existence of a compiler for the OZ part that produces correct executable code C for Z_{OZ} . This code is correct if it does not jeopardise the OZ specification, i.e., $\llbracket C \rrbracket \subseteq \llbracket Z_{OZ} \rrbracket$.

An issue we have to care about is that on CTA level different events can occur synchronously. The PEA semantics of COD yet demands that there is only one event per time point. For this reason, only properties that are *event robust* can be preserved from COD to task level. Event robustness means that a property does not distinguish between synchronous and sequential occurrences of events, where the events are separated by an infinitesimal amount of time.

Definition 6 (Event robustness). *Given events $a, b \in \text{Events}$, a property φ is called event robust iff $\exists 0 < \delta \forall 0 < \epsilon < \delta$ s.t.*

$$\begin{aligned} & \langle \dots, (a, t), (b, t + \epsilon), \dots \rangle \in \llbracket \varphi \rrbracket \Rightarrow \langle \dots, (a, t), (b, t), \dots \rangle \in \llbracket \varphi \rrbracket \\ \vee & \langle \dots, (a, t - \epsilon), (b, t), \dots \rangle \in \llbracket \varphi \rrbracket \Rightarrow \langle \dots, (a, t), (b, t), \dots \rangle \in \llbracket \varphi \rrbracket \end{aligned}$$

However, this is not a real restriction since properties depending on events that occur exactly at the same time are unrealistic and can never be implemented on real-world systems anyway. The DC properties that can be verified for COD specifications, i.e., DC test formulae from [22], are predominantly robust. Exceptions are for instance formulae like $((\exists a \wedge \ell > 1) \wedge (\exists b \wedge \ell \geq 1))$. The basic idea to consider robust systems is taken from [10,9].

We state the correctness of our construction—the main result of this work—in the following theorem.

Theorem 1 (Correctness). *We consider a COD model Z that shall be implemented on a machine M , a translation into a task network $CTA(Z) = CTA$, a translation of the OZ part Z_{OZ} into executable code C , and a event robust DC property φ , only restricting events in \mathbb{T} . Assume the translation of Z_{OZ} to be correct, i.e., $\llbracket C \rrbracket \subseteq \llbracket Z_{OZ} \rrbracket$. Then*

$$\left. \begin{array}{l} Z \models \varphi \\ (CTA, C, M) \models \text{feasible} \end{array} \right\} \Rightarrow (CTA, C, M) \models \varphi.$$

Proof. Let the parts of Z be given by Z_{CSP} , Z_{OZ} , and Z_{DC} , respectively. To prove this theorem, we apply the following property, establishing a relationship between the task networks of our construction and the COD specification.

$$[[CTA]]_{|\mathbb{T}} \subseteq [[Z_{CSP}]]_{|\mathbb{T}} \cap [[Z_{DC}]]_{|\mathbb{T}} \quad (1)$$

This property's correctness is proven in Lemma 3 that can be found in the appendix. The subscript $|\mathbb{T}$ denotes the restriction of traces to events of Z representing tasks. We know by definition that $[Z] = [Z_{CSP}] \cap [Z_{OZ}] \cap [Z_{DC}]$. Due to the precondition $Z \models \varphi$, we get $[Z] \subseteq [\varphi]$ and since φ restricts only events in \mathbb{T} also $[Z_{CSP}]_{|\mathbb{T}} \cap [Z_{OZ}] \cap [Z_{DC}]_{|\mathbb{T}} \subseteq [\varphi]$. We can conclude

$$\begin{aligned} & [[Z_{CSP}]]_{|\mathbb{T}} \cap [Z_{OZ}] \cap [Z_{DC}]_{|\mathbb{T}} \subseteq [\varphi] \\ \Rightarrow & [[Z_{CSP}]]_{|\mathbb{T}} \cap [C] \cap [Z_{DC}]_{|\mathbb{T}} \subseteq [\varphi] && \{[C] \subseteq [Z_{OZ}]\} \\ \Rightarrow & [[CTA]]_{|\mathbb{T}} \cap [C] \subseteq [\varphi] && \{(1)\} \\ \Rightarrow & [[CTA]] \cap [C] \subseteq [\varphi] && \{(*)\} \\ \Rightarrow & \emptyset \subset [[CTA]] \cap [C] \cap [M] \subseteq [\varphi] && \{\text{Feasibility}\} \\ \Rightarrow & (CTA, C, M) \models \varphi. && \{\text{Def. 2}\} \end{aligned}$$

The conclusion $(*)$ holds because φ only restricts events in \mathbb{T} and, particularly, no internal events of CTA . Further note that feasibility is necessary as (CTA, C, M) is only an implementation if its semantics is not empty (cf. Def. 2). \square

4.2 Results for the example

To demonstrate the application of our approach, we take up the odometer example of Sect. 1.1 and 2.1. We start by formulating the identified high-level requirements in terms of DC formulae: after execution of task DDT the actual train position is never beyond the calculated confidence interval, i.e., $\neg(\text{true} \wedge \downarrow DDT_e \wedge$

$[\text{actPos} < \text{estPos} + \text{conf}_l])$ and $\neg(\text{true} \wedge \uparrow DDT_e \wedge [\text{actPos} > \text{estPos} + \text{conf}_r])$). Note that the actual position is realised via a COD specification for the environment, which is not presented here. Using the techniques of [22], we verified these properties (assuming exclusive access to shared resources) for our case study model using the model checker ARMC. To pass from the specification to the implementation, we generated the task network according to Def. 5. We get the following CTA components for the processes P_1 and P_3 (cf. Fig. 3):

$$\begin{aligned} P_1 : & \quad Q(\text{Out}(RSS_b, 20, \infty, 0, O)), E(\text{In}(RSS_b, 20, \infty, 0, O), RSS_e, 2, 2), \\ & \quad S(\text{In}(RSS_e, 20, \infty, 0, 2)), \\ P_3 : & \quad Q(\text{Out}(DCS_b, 40, 40, 0, O)), E(\text{In}(DCS_b, 40, 40, 0, O), DDT_b, 11, 17), \\ & \quad E(\text{In}(DDT_b, 40, 40, 6, 11), DDT_e, 9, 28), S(\text{In}(DDT_e, 40, 40, 23, 20)) \end{aligned}$$

Task	WCET	P^-	P^+	R^-	R^+	J	O
RSS	2	20	∞	2	2	0	0
DCL	4	36	∞	4	6	0	0
DCS	11	40	40	11	17	0	0
DDT	9	40	40	9	28	6	11

Table 2: Scheduling results

In order to get the values for the response times (R^- , R^+), we applied scheduling analysis on the tasks. Since code generation is not subject of this work, we abandon worst case execution time (WCET) analysis in order to get execution times, and estimated the parameters as shown in Table 2. We assumed a fixed priority scheduling (tasks are shown in descending priority order). Thus, we could use the methods described for example in [17,28] to calculate the response times. All response times remain lower as the respective periods, so that the task network is feasible. Now, our correctness result from Theorem 1 automatically yields the feasibility of the specification on a given target platform, and—assuming a correct compiler and WCET analysis—the verified safety properties remain valid for implementations that comply with the task network structure.

5 Discussion and Future Work

In this paper, we have introduced a framework for connecting two stages of the formal design process for real-time systems: high-level verification on specifications and scheduling analysis for implementations. Without such an indispensable connection, it is undetermined whether a verified real-time specification can be implemented on a target platform and how verified properties can be preserved for the implementation. That is, safety is not guaranteed despite successful verification. To solve this issue, we have proposed a property-preserving translation from task specifications in terms of COD into task networks in terms of Cyclic Timed Automata. The limitation to a subclass of COD was necessary since COD can express much more timing assumptions and structures than scheduling analysis can deal with, due to its simple view on task networks with periods, jitters, and deadlines. However, it is worthwhile to use a formalism like COD because it enables declarative high-level system descriptions and it supports complex and infinite data types [12,22,6].

The current definition of the translation for the CSP part covers a big class of relevant periodic process structures. The extension of the control flow part to allow more complex structures is future work. In particular, a more sophisticated treatment of choices and synchronised parallel operators is desirable. Moreover, real-world systems are often not strict periodical but comprise mode changes. An integration of mode changes into the CSP part would be easy but scheduling must support mode changes, which is currently not the case for CTA. In addition, it is future work to provide rules for transforming general CSP processes (if possible) into processes matching our patterns from Def. 3.

Besides the fact that our method provides scheduling analysis for a high-level language, a second benefit is that schedulability results can be propagated back to the specification. The idea is that information of the scheduling analysis—particularly if not schedulable—can be used to merge tasks already at specification level to reduce communication costs. Scheduling analysis can then be repeated with the modified specification. A detailed elaboration of this iterative procedure remains future work.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. *Theo. Comp. Science* **126**(2) (1994) 183–235
2. Burmester, S., Giese, H., Hirsch, M., Schilling, D., Tichy, M.: The fujaba real-time tool suite: model-driven development of safety-critical, real-time systems. In: ICSE, ACM (2005) 670–671
3. Burns, A., Lin, T.: An engineering process for the verification of real-time systems. *Formal Aspects of Computing* **19**(1) (March 2007) 111–136
4. Dierks, H.: Time, abstraction and heuristics – automatic verification and planning of timed systems using abstraction and heuristics. *Habil. thesis, University of Oldenburg* (2005)
5. ERTMS User Group, UNISIG: ERTMS/ETCS System requirements specification. <http://www.aEIF.org/ccm/default.asp> (2002) Version 2.2.2.
6. Faber, J., Jacobs, S., Sofronie-Stokkermans, V.: Verifying CSP-OZ-DC specifications with complex data types and timing parameters. In: IFM. Volume 4519 of LNCS. (2007) To appear.
7. Fersman, E., Mokrushin, L., Petterson, P., Yi, W.: Schedulability Analysis using Two Clocks. In: TACAS. LNCS 2619, Springer (2003) 224–239
8. Fersman, E., Yi, W.: A Generic Approach to Schedulability Analysis of Real Time Tasks. *NJC* **11** (2004)
9. Fränzle, M.: Analysis of hybrid systems: An ounce of realism can save an infinity of states. In: CSL. Volume 1683 of LNCS., Springer (1999) 126–140
10. Gupta, V., Henzinger, T.A., Jagadeesan, R.: Robust timed automata. In Maler, O., ed.: HART. Volume 1201 of LNCS., Springer (1997) 331–345
11. Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice-Hall (1985)
12. Hoenicke, J.: *Combination of Processes, Data, and Time*. PhD thesis, University of Oldenburg, Germany (2006)
13. Hoenicke, J., Olderog, E.R.: CSP-OZ-DC: A combination of specification techniques for processes, data and time. *NJC* **9** (2002) 301–334
14. Hsiung, P.A., Lee, T.Y., Su, F.S.: Formal synthesis and code generation of real-time embedded software using time-extended quasi-static scheduling. In: APSEC, IEEE CS (2002) 395–404
15. Hsiung, P.A., Wang, F., Kuo, Y.S.: Scheduling system verification. In Cleaveland, R., ed.: TACAS. Volume 1579 of LNCS., Springer (1999) 19–33
16. Joseph, M., ed.: *Real-time systems*. Prentice Hall (1996)
17. Joseph, M., Pandya, P.: Finding Response Times in a Real-Time System. *BCS Computer Journal* **29**(5) (Oct. 1986) 390–395
18. Knapp, A., Merz, S., Rauh, C.: Model checking - timed UML state machines and collaborations. In: FTRTFT. Volume 2469 of LNCS., Springer (2002) 395–416
19. Lin, B.: Software synthesis of process-based concurrent programs. In: DAC, ACM (1998)
20. Madl, G., Abdelwahed, S., Schmidt, D.C.: Verifying Distributed Real-time Properties of Embedded Systems via Graph Transformations and Model Checking. *Real-Time Systems* **33** (2006)
21. Maler, O., Pnueli, A., Sifakis, J.: On the synthesis of discrete controllers for timed systems (an extended abstract). In: STACS. (1995) 229–242
22. Meyer, R., Faber, J., Rybalchenko, A.: Model checking duration calculus: A practical approach. In: ICTAC. Volume 4281 of LNCS., Springer (2006) 332–346
23. Roscoe, A.: *Theory and Practice of Concurrency*. Prentice Hall (1998)
24. Sgroi, M., Lavagno, L., Watanabe, Y., Sangiovanni-Vincentelli, A.: Synthesis of embedded software using free-choice petri nets. In: DAC, ACM (1999) 805–810
25. Smith, G.: *The Object Z Specification Language*. Kluwer Academic Publishers (2000)
26. Stierand, I., Metzner, A., Dierks, H.: Combining Timed Automata based Formal Specifications and Real-Time Scheduling. Submitted. <http://ca.informatik.uni-oldenburg.de/~ingos/cta.pdf>
27. Thiele, L., Chakraborty, S., Naedele, M.: Real-time Calculus for Scheduling Hard Real-Time Systems. In: ISCAS. (2000) 101–104
28. Tindell, K., Burns, A., Wellings, A.G.: Allocating Real-Time Tasks (An NP-Hard Problem made easy). *Real-Time Systems* **4**(2) (June 1992) 145–165
29. Zhou, C., Hansen, M.R.: *Duration Calculus*. Springer-Verlag (2004)

A Appendix for Referees

This appendix presents some definitions and the proof of Theorem 1 in detail.

A.1 CSP-OZ-DC

CSP. In COD, external and internal communications and the control flow of methods are described by CSP. CSP was introduced in [11], and is used to model parallel and sequential processes in terms of the events they communicate. Events always occur in instantaneous time. We denote the set of events \mathbf{Events} . The CSP syntax is defined by the following BNF grammar:

$$P ::= \mathbf{Stop} \mid \mathbf{Skip} \mid a \rightarrow P \mid P_1 \square P_2 \mid P_1 \parallel_A P_2 \mid P_1 \parallel P_2 \mid P_1 ; P_2 \mid X$$

The process \mathbf{Stop} denotes a diverging process, \mathbf{Skip} a terminating process. For an event $a \in \mathbf{Events}$, $a \rightarrow P$ denotes a process that communicates a and then behaves like P . The operator \square represent external represents external choice. The parallel composition of processes is described by \parallel_A and \parallel . The former is used for processes that synchronise on events $A \subseteq \mathbf{Events}$, and the latter for interleaved processes without any synchronisation. The operator $;$ denotes sequential composition of processes. Finally, X stands for a process identifier, which must be declared with an expression $X \stackrel{c}{=} P$.

For CSP a structured operational semantics (SOS) has been defined, represented as a labelled transition system $(Q, \mathbf{Events} \cup \{\tau, \checkmark\}, q_0, \longrightarrow)$, with a special event \checkmark representing the termination of a process, and an internal event τ that is not visible from outside. The states Q are the CSP processes, and $q_0 \in Q$ is the initial process. The transition relation \longrightarrow is defined inductively over the structure of CSP processes. Examples for rules defining \longrightarrow are:

$$\frac{}{a \rightarrow P \xrightarrow{a} P} \quad \frac{P_2 \xrightarrow{a} P'_2}{P_1 \parallel P_2 \xrightarrow{a} P_1 \parallel P'_2} \quad \frac{P \xrightarrow{\checkmark} P'}{P ; Q \xrightarrow{\tau} Q}$$

OZ. Object-Z [25] is an object-oriented extension for Z. We use OZ to define the state space over variables \mathbf{Var} of a COD specification and possible state changes. State changes are defined in a constraint-based declarative way by *operation schemas*. The schemas are associated to events from the CSP part such that a state change is always performed at the same time when a CSP event is triggered. As task analysis abstracts from the state space, it is not necessary to introduce the OZ part in more detail here. For details we refer to [12].

DC. Timing properties of a COD specification are modelled in the Duration Calculus (DC) part. The DC is an interval-based dense real-time logic [29]; we denote the time domain by $\mathbb{T} := \mathbb{R}^+$. The DC describes a state of a system at one point in time using so-called state expressions φ , formulae over time-dependent

state variables Var , and predicates p/n . For $\sim \in \{<, \leq, =, \geq, >\}$ and $t \in \mathbb{T}$ and state expressions φ the class of formulae ϑ is defined by:

$$\vartheta ::= \ell \sim t \mid \lceil \varphi \rceil \mid \neg \vartheta \mid \vartheta_1 \vee \vartheta_2 \mid \vartheta_1 \wedge \vartheta_2.$$

The length of an interval is denoted with ℓ , the operator \wedge splits an interval into two parts. The expression $\lceil \varphi \rceil$ states that a state expression φ holds almost everywhere on a given interval. The semantics of a state variable $v \in \text{Var}$ is given by an interpretation $I[v] : \mathbb{T} \rightarrow D(v)$, assigning values from the variable's data domain. The semantics of a predicate p/n is an interpretation $\hat{p} : D(v_1) \times \dots \times D(v_n) \rightarrow \{1, 0\}$. The semantics of DC formulae is an interpretation of the formulae on an interval $[b, e]$. With $b \leq e$; $b, e \in \mathbb{T}$ and $t \in \mathbb{T}$ we define the interpretation I of state expressions and formulae:

$$\begin{aligned} I[p(v_1, \dots, v_n)](t) &:= \hat{p}(I[v_1](t), \dots, I[v_n](t)) \\ I[\lceil \varphi \rceil](t) &:= 1 - I[\varphi](t) \\ I[\varphi_1 \vee \varphi_2](t) &:= \max\{I[\varphi_1](t), I[\varphi_2](t)\} \\ I[\ell \sim t][b, e] &:= \text{true iff } (e - b) \sim t \\ I[\lceil \varphi \rceil][b, e] &:= \text{true iff } \int_b^e I[\varphi](t) dt = (e - b), b < e \\ I[\vartheta_1 \wedge \vartheta_2][b, e] &:= \text{true iff there is an } m \in [b, e] \text{ s.t.} \\ &\quad I[\vartheta_1][b, m] = \text{true and } I[\vartheta_2][m, e] = \text{true} \end{aligned}$$

We demand finite variability for state expressions, so that $\int I[\varphi](t)$ is well defined. The semantics of \neg and \vee is as usual.

Events $e \in \text{Events}$ are realised by boolean state variables. The occurrence of an event e is denoted by $\uparrow e$, which is valid for a point in time t iff the state variable e changes its value at time t . Finally, to express that an event does not occur during an interval of length greater 0, we use $\Box e$.

Semantics. In [13,12], the operational semantics of COD specifications is given in terms of Phase Event Automata (PEA), a Timed Automata model [1] involving data aspects. Since we refer throughout this paper to the trace semantics of PEA only, we omit the formal definition, and relate the COD specification directly to the traces produced by its PEA semantics. A PEA trace is an infinite sequence of configurations $\langle (e_1, \beta_1, t_1), (e_2, \beta_2, t_2), \dots \rangle$, with events e_i , variable valuations β_i , and points in time $t_i \in \mathbb{T}$, $t_i > 0$, for $i \in \mathbb{N}$. Furthermore, we use a simplified variant of PEA traces, because we are not interested in program locations and clock valuations. With respect to satisfiability of DC properties, the results of [12] remain valid, as validity of DC properties only depends on events, variables valuations and timestamps.

Definition 7 (Trace semantics of COD). *Given a COD specification Z with the appropriate parts Z_{CSP} , Z_{OZ} , Z_{DC} , the semantics is defined compositionally by $\llbracket Z \rrbracket := \llbracket Z_{CSP} \rrbracket \cap \llbracket Z_{OZ} \rrbracket \cap \llbracket Z_{DC} \rrbracket$, with the following definitions for the parts of the specification:*

CSP. Traces of Z_{CSP} are given by

$$\begin{aligned} [Z_{CSP}] := \{ \langle (e_0, \beta_0, t_0), (e_1, \beta_1, t_1), \dots \rangle \mid \text{where} \\ q_0 \xrightarrow{\tau}^* \cdot \xrightarrow{e_0} \cdot \xrightarrow{\tau}^* \cdot \xrightarrow{e_1} \dots \text{ is a run of the SOS of } Z_{CSP} \}. \end{aligned}$$

OZ. The semantics of Z_{OZ} is given by traces $[Z_{OZ}]$, s.t. variable valuations β are restricted according to the changes in the operation schemas. Time is not restricted.

DC. Traces of Z_{DC} with an interpretation I are given by

$$\begin{aligned} [Z_{DC}] := \{ \langle (e_0, \beta_0, t_0), (e_1, \beta_1, t_1), \dots \rangle \mid \text{where} \\ I[v](t) = \beta_i(v) \text{ for almost all } t \in \text{Int}_i, \\ I[e](t) = I[e](t') \text{ for almost all } t, t' \in \text{Int}_i, \\ I[e](t) = I[e](t') \text{ if } e \neq e_i \text{ for almost all } t \in \text{Int}_i, t' \in \text{Int}_{i-1}, \\ I[e](t) = \neg I[e](t') \text{ if } e = e_i \text{ for almost all } t \in \text{Int}_i, t' \in \text{Int}_{i-1} \}, \end{aligned}$$

where $\text{Int}_i = [\sum_{j=0}^{i-1} t_j, \sum_{j=0}^i t_j]$, $e \in \text{Events}$, and $v \in \text{Var}$.

In [22] a subclass of DC is defined for which COD model-checking is possible. For a formula ϕ of this subclass we consider the set $[\phi]$ of traces satisfying ϕ . We write $Z \models \phi$ iff $[Z] \subseteq [\phi]$. We denote the set of all timed traces with Traces .

A.2 Correctness

To finish the proof of Theorem 1 we have to establish the desired relationship of CTA and COD: $[CTA] \subseteq [Z_{CSP}] \cap [Z_{DC}]$. We will prove this relationship in two steps. First, we show that the generated task network does not violate the structure of the CSP part. In a second step, we make use of the first step's results and prove that the traces of the task network also satisfy the timing constraints given by the DC part.

To this end, we start by defining the notion of *cyclic traces* and show in Lemma 1 the relationship between cyclic traces and CTA.

Definition 8 (Cyclic traces). A set T of traces $T \subseteq \text{Traces}$ is called cyclic over T' ending with event e for a trace $T' \subseteq \text{Traces}$, iff

$$T = \{ \langle w_0, (e, t_1), w_1, (e, t_2), w_2 \dots \rangle \mid w_i \in T' \}.$$

Analogously, it is called cyclic over T' starting with event e , iff

$$T = \{ \langle (e, t_1), w_1, (e, t_2), w_2, (e, t_3), w_3 \dots \rangle \mid w_i \in T' \}.$$

Lemma 1 (Concatenation of CTA). Assume CTA A_1 and A_2 with trace semantics $[A_1]$ and $[A_2]$ that are cyclic over A'_1 , ending with e , and A'_2 , starting with e , respectively. Additionally, the parallel composition of A_1 and A_2 exists

(i.e. they are schedulable, in particular, we demand that the accumulated response times are smaller than the period of the traces in A_1 : $\forall w \in [A'_1], v \in [A'_2] : J_w + R^+(w) + R^+(v) + R^+(e) < P^-$). Then the concatenation $A_1 \circ A_2$ is defined by $A_1 \circ A_2 := A_1 \parallel A_2$, and the traces of the concatenation are given by

$$[[A_1 \circ A_2]] = \{ \langle w_0, (e, t_1), \bar{w}_1, w_1, (e, t_2), \bar{w}_2, w_2, \dots \rangle \mid w_i \in [A'_1], \bar{w}_i \in [A'_2] \},$$

which is cyclic again, ending with e .

Proof. From the assumption that the A_i are cyclic over A'_i we know

$$\begin{aligned} [A_1] &= \{ \langle w_0, (e, t_1), w_1, (e, t_2), w_2, \dots \rangle \mid w_i \in [A'_1] \} \text{ and} \\ [A_2] &= \{ \langle (e, t_1), \bar{w}_1, (e, t_2), \bar{w}_2, (e, t_2), \dots \rangle \mid \bar{w}_i \in [A'_2] \} \end{aligned}$$

Since the A_i are schedulable, we know that there common traces exist. In particular, we only need to consider traces where the event e only occur at the same times t_i , otherwise the trace will not occur in the intersection of the traces of A_1 and A_2 . So all traces of the concatenation $A_1 \circ A_2$ have the same shape

$$\langle w_0, (e, t_1), v_1, (e, t_2), v_2, (e, t_3), v_3, \dots \rangle$$

and we only have to show that each of the v_i consists of $w_i \in [A'_1]$ and $\bar{w}_i \in [A'_2]$ that do not overlap. To this end, we recall that traces for an arbitrary trigger are given by

$$\begin{aligned} &\{ \langle (w, t_0 + \delta_0), (e, t_1 + \delta_1), \dots \rangle \mid \\ &O \leq t_0 \leq P^+, \quad t_i \in [t_{i-1} + P^-, t_{i-1} + P^+], \quad 0 \leq \delta_i \leq J \}. \end{aligned}$$

So, we consider traces from A_1 and A_2 with synchronous occurrences of e :

$$\begin{aligned} &\langle (w_0, t_0^w + \delta_0^w), (e, t_0 + \delta_0), (w_1, t_1^w + \delta_1^w), (e, t_0 + \delta_0), \dots \rangle \in [A_1] \\ &\langle (e, t_0 + \delta_0), (v_1, t_1^v + \delta_1^v), (e, t_0 + \delta_0), (v_2, t_2^v + \delta_2^v), \dots \rangle \in [A_2], \end{aligned} \quad (2)$$

where $w_i \in [A'_1]$ and $v_i \in [A'_2]$. For simplification, we abbreviate subtraces $w = \langle (a_0, t_0), \dots, (a_n, t_n) \rangle$ by (w, t_0) , because we are only interested in the time point at the beginning of a trace. Now, we show that for those traces in (2) we can conclude

$$\forall i \in \mathbb{N} : t_i^v + \delta_i^v + R^+(v_i) < t_i^w + \delta_i^w, \quad (3)$$

by which the v_i and the w_i are separated properly. First, we collect information about the traces from the periodicity of the w_i (the w_i occur with the same period as the event e). Hence, for the traces in (2) we get

$$t_0^w \leq P^+ \text{ and } t_i^w \in [t_{i-1}^w + P^-, t_{i-1}^w + P^+]. \quad (4)$$

The event e is determined by the same period, but with an offset given by the response time $R(w_i)$ of the w_i :

$$t_i + \delta_i \in [t_i^w + \delta_i^w + R^-(w_i), t_i^w + \delta_i^w + R^+(w_i)]. \quad (5)$$

Analogously, we determine the time points for the v_i and apply (5):

$$\begin{aligned} & t_i^v + \delta_i^v \in [t_{i-1} + \delta_{i-1} + R^-(e), t_{i-1} + \delta_{i-1} + R^+(e)] \\ \{(5)\} \Leftrightarrow & t_i^v + \delta_i^v \in [t_{i-1}^w + \delta_{i-1}^w + R^-(w_{i-1}) + R^-(e), \\ & t_{i-1}^w + \delta_{i-1}^w + R^+(w_{i-1}) + R^+(e)] \end{aligned} \quad (6)$$

Now, starting from the precondition, we can use our results in (4) and (6) to conclude (3).

$$\begin{aligned} & J + R^+(w_{i-1}) + R^+(e) + R^+(v_i) < P^- \\ \Rightarrow & t_{i-1}^w + J + R^+(w_{i-1}) + R^+(e) + R^+(v_i) < t_{i-1}^w + P^- \\ \{(4)\} \Rightarrow & t_{i-1}^w + J + R^+(w_{i-1}) + R^+(e) + R^+(v_i) < t_i^w \\ \Rightarrow & t_{i-1}^w + \delta_{i-1}^w + R^+(w_{i-1}) + R^+(e) + R^+(v_i) < t_i^w + \delta_i^w \\ \{(6)\} \Rightarrow & t_i^v + \delta_i^v + R^+(v_i) < t_i^w + \delta_i^w \end{aligned}$$

By this it is clear that the v_i and w_i in the traces of (2) do not overlap and every trace of the intersection of A_1 and A_2 has the form

$$\begin{aligned} & \langle (w_0, t_0^w + \delta_0^w), (e, t_0 + \delta_0), (v_1, t_1^v + \delta_1^v), \\ & (w_1, t_1^w + \delta_1^w), (e, t_0 + \delta_0), \dots \rangle \in [A_1 \parallel A_2], \end{aligned}$$

which is cyclic again per definition. \square

Since the CSP part always has the structure defined by the BNF in Def. 4, we show the compliance of the task network structure with the CSP part by induction over the structure of the CSP processes. To this end, the next Lemma 2 relates terminating processes *TermProc* to its CTA from the construction. The main problem here is that terminating CSP processes always produce inherently finite, terminating traces, whereas CTA always have periodic behaviour. Therefore, the main idea is to relate a CTA for a terminating process $CTA(TP)$ to a modified variant TP' representing the “periodic closure” of TP . That is, TP' behaves as TP but it restarts TP after terminating.

Lemma 2 (Control structure of *TermProc*). *Given a TermProc TP and a modified process $TP' := \mu X : TP ; X$, with a schedulable $CTA(TP)$ we can conclude the restriction of the CTA traces to events visible on COD level $[CTA(TP)]_{\Upsilon} \subseteq [TP']$ and, particularly, $[CTA(TP)]_{\Upsilon}$ is cyclic over $[TP]$.*

Proof. We show the proposition by induction over the structure of *TermProc*.

1. $a \rightarrow \text{Skip}$. The semantics of a CSP process in COD is defined by a labelled transition system represented as PEA [12]. Due to the stuttering invariance we can ignore all τ steps introduced by the translation of CSP to PEA. Then the (generalised) traces of the CSP process $\mu X : a \rightarrow \text{Skip} ; X$ are simply

$$\{ \langle (a, \cdot, \cdot), (a, \cdot, \cdot), (a, \cdot, \cdot), \dots \rangle \}. \quad (7)$$

We do not need to take the variable values β and the time points into account here, as they are not constraint by the CSP part and can be chosen arbitrarily. So we omit them completely in the following. The CTA of $a \rightarrow \mathbf{Skip}$ is given by $CTA(a \rightarrow \mathbf{Skip}, P, J, P, \cdot)$, which is the union of a trigger for a a execute automaton $E(In(a, P^-, P^+, J, O), e, R^+(a), R^-(a))$ and a sink for e , where e is a new event that is not visible in the trace semantics. Obviously, $CTA(a \rightarrow \mathbf{Skip})$ is a subset of (7) and it is cyclic over $\llbracket a \rightarrow \mathbf{Skip} \rrbracket = \{a\}$.

2. $a \rightarrow P$. The CTA for $a \rightarrow P$ are given by $CTA(a \rightarrow P, P^+, P^-, J, O, \cdot) := A_1 \parallel A_2$, where

$$A_1 := \{E(In(a, P^-, P^+, J, O), f, R^-(a), R^+(a))\} \quad (8)$$

$$A_2 := CTA(P, P^+, P^-, J, O + R^+(a) - R^-(a), f) \quad (9)$$

The traces for (8) are given by a set $\llbracket A_1 \rrbracket = \{\langle (a, \beta_1, t_1), (f, \beta'_1, t'_1), \dots \rangle\}$ according to [26]. It is cyclic, beginning with a . By induction hypothesis we know for the sub-process P that $\llbracket A_2 \rrbracket_{\top} \subseteq \llbracket \mu X : P ; X \rrbracket_{\top}$ which is cyclic over $\llbracket P \rrbracket_{\top}$, i.e. $\llbracket A_2 \rrbracket \subseteq \{\langle (f, \beta'_1, t'_1), w_1, (f, \beta'_2, t'_2), w_2, \dots \rangle \mid w_i \in \llbracket P \rrbracket\}$.

As the entire CTA for TP , namely $CTA(TP)$, is assumed to be schedulable, we know that all its sub CTA are also schedulable. So the preconditions for Lemma 1 are fulfilled and with application of this lemma we get traces of the concatenation of A_1 and A_2 :

$$\llbracket A_1 \circ A_2 \rrbracket \subseteq \{\langle (a, \beta_1, t_1), (f, \beta'_1, t'_1), w_1, (a, \beta_2, t_2), \dots \rangle \mid \langle w_i \rangle \in \llbracket P \rrbracket\}$$

and if we restrict these traces to visible events

$$\begin{aligned} \llbracket A_1 \circ A_2 \rrbracket_{\top} \subseteq \{\langle (a, \beta_1, t_1), w_1, (a, \beta_2, t_2), w_2, \dots \rangle \\ \mid \langle a, w_i \rangle \in \llbracket a \rightarrow P \rrbracket \text{ for } \langle w_i \rangle \in \llbracket P \rrbracket\}, \end{aligned}$$

which is cyclic over $\llbracket a \rightarrow P \rrbracket_{\top}$, beginning with a . Since $\llbracket CTA(a \rightarrow P) \rrbracket = \llbracket A_1 \parallel A_2 \rrbracket = \llbracket A_1 \circ A_2 \rrbracket$ we are finished.

3. $P_1 ; P_2$. The proof for this *TermProc* is analogous to the previous one. The CTA is given by $CTA(P_1 ; P_2, P^+, P^-, J, O, a) := A_1 \parallel A_2 \parallel A_3$ with

$$A_1 := CTA(P_1, P^+, P^-, J, O, a) \quad (10)$$

$$A_2 := \{E(In(e, P^-, P^+, J, O), f, R^-(e), R^+(e))\} \quad (11)$$

$$A_3 := CTA(P_2, P^+, P^-, J, O(P_1) + R^+(e) - R^-(e), f), \quad (12)$$

where e is an event such that $S(e, \cdot, \cdot, \cdot) \in CTA(P_1, P^+, P^-, J, O, a)$. We now make use of the induction hypothesis to get trace sets for A_1 and A_3 and the CTA semantics to get A_2 :

$$\llbracket A_1 \rrbracket_{\top} \subseteq \{\langle (w_1, (e, \beta'_1, t'_1), w_2, (e, \beta'_1, t'_1), \dots) \mid w_i \in \llbracket P_1 \rrbracket \rangle\} \quad (13)$$

$$\llbracket A_2 \rrbracket_{\top} \subseteq \{\langle (e, \beta'_1, t'_1), (f, \beta''_1, t''_1), (e, \beta'_2, t'_2), (f, \beta''_2, t''_2), \dots \rangle\} \quad (14)$$

$$\llbracket A_3 \rrbracket_{\top} \subseteq \{\langle (f, \beta''_1, t''_1), w'_1, (f, \beta''_1, t''_1), w'_2, \dots \rangle \mid w'_i \in \llbracket P_2 \rrbracket_{\top} \rangle\} \quad (15)$$

As those three sets comply with the requirements, we apply Lemma 1 twice and get the trace set

$$\begin{aligned} [A_1 \circ A_2 \circ A_3]_{|\mathbb{T}} \subseteq & \{ \langle w_1, (e, \beta'_1, t'_1), (f, \beta''_1, t''_1), w'_1, w_2, \dots \rangle \\ & | \langle w_1, (e, \beta'_1, t'_1), (f, \beta''_1, t''_1), w_2 \rangle \in [P_1; P_2]_{|\mathbb{T}} \\ & \text{for } \langle w_i \rangle \in [P_1], \langle w'_i \rangle \in [P_2]_{|\mathbb{T}} \}, \end{aligned}$$

assuming the visibility of e and f on COD level, otherwise we can just omit them in the last set.

4. $P_1 \parallel P_2$. The proof for the interleaving uses a parallelised variant of Lemma 1 – but the proof is identical to the proofs for the previous *TermProc* and, hence, omitted here.

□

Lemma 3 (CTA and PEA). *Given a COD model Z (consisting of the appropriate parts Z_{CSP}, Z_{OZ}, Z_{DC}) and a machine M , we assume a translation into task networks $CTA(Z)$, executable code for the tasks $C(Z)$. If we further assume the feasibility of these translations, $(CTA, C, M) \models \text{feasible}$, we can conclude*

$$[CTA(Z)]_{|\mathbb{T}} \subseteq [Z_{CSP}]_{|\mathbb{T}} \cap [Z_{DC}]_{|\mathbb{T}}.$$

Proof. The proof is in two steps. In a first step (a), we show that our translations preserves the control structure of Z for traces that are adjusted of invisible events not in \mathbb{T} , i.e. $[CTA(Z)]_{|\mathbb{T}} \subseteq [Z_{CSP}]_{|\mathbb{T}}$. We use this information to show in a second step (b) that the timing behaviour given by the COD model is also not violated, $[CTA(Z)]_{|\mathbb{T}} \subseteq [Z_{DC}]_{|\mathbb{T}}$.

(a) We know that Z_{CSP} is in the form of Def. 3, consisting in an interleaving of looping process declarations $TPD = LPD_1 \parallel \dots \parallel LPD_n$. For every *TermProc* in those process declarations we apply Lemma 2 and get

$$[CTA(LPD_i)]_{|\mathbb{T}} \subseteq [LPD_i]_{|\mathbb{T}}. \quad (16)$$

It remains to show that this relation also holds for the interleaving of looping process declarations. But this is clear since these interleavings are translated to a parallel composition of CTA without any synchronisation. Thus, on trace level we likewise have interleavings of traces. Therefore, we conclude

$$[CTA(\parallel_i LPD_i)]_{|\mathbb{T}} = \parallel_i [CTA(LPD_i)]_{|\mathbb{T}} \subseteq \parallel_i [LPD_i] = [\parallel_i LPD_i].$$

(b) We consider an arbitrary trace $\pi \in [CTA(Z)]_{|\mathbb{T}}$ and show that Z_{DC} also enables this trace π :

$$\pi \in [CTA(Z)]_{|\mathbb{T}} \Rightarrow \pi \in [Z_{DC}]_{|\mathbb{T}}.$$

We show this assumption for every formulae we allow for the DC part according to Def. 3. Examining $\mathbf{DC}_J(J, a)$, $\mathbf{DC}_{P+}(P, a_t)$, and $\mathbf{DC}_{P-}(P, a_t)$, we consider

that all of these refer to a_t , which is invisible on CTA level. So, for every $\pi \in \llbracket CTA(Z) \rrbracket_{\Gamma}$ we have to find a trace $\pi' \in \llbracket Z_{DC} \rrbracket$ such that

$$\pi'_{|\Gamma} = \pi. \quad (17)$$

Thus, let us assume a trace

$$\begin{aligned} \pi = \langle \dots, (a^0, \beta^0, t^0 + \delta^0), \dots, (a^1, \beta^1, t^1 + \delta^1), \dots, (a^2, \beta^2, t^2 + \delta_1), \dots \rangle \\ \in \llbracket CTA(Z) \rrbracket_{\Gamma} \end{aligned}$$

with no other occurrences of a . According to the CTA semantics we know that the time points fulfill the following criteria for the period and the jitter, $t_0 \leq P^+$, $t_{i+1} \in [t_i + P^-, t_i + P^+]$, $0 \leq \delta_i \leq J$ for $i \in \mathbb{N}$.

We now construct a modified trace π' with the desired property (17) (the dotted parts remain the same).

$$\begin{aligned} \pi' = \langle \dots, (a_t^0, \beta_t^0, t^0), \dots, (a^0, \beta^0, t^0 + \delta^0), \dots, \\ (a_t^1, \beta_t^1, t^1), \dots, (a^1, \beta^1, t^1 + \delta^1), \dots, \\ (a_t^2, \beta_t^2, t^2), \dots, (a^2, \beta^2, t^2 + \delta_1), \dots \rangle \end{aligned}$$

Obviously, property (17) is satisfied for π and π' . The only reason that this could not be a possible extension of π is that there are already other events b, β_b, t_i occurring at the same points in time t_i , where we inserted the a_t events. But because of the event robustness of the properties we want to check, we can assume that this is never the case. (Otherwise, we examine a slightly modified trace the events at the critical points in time are moved arbitrary small amount. Then, from the robustness it follows that all relevant properties also hold for π , confer Def. 6.) Additional, this trace π' has the necessary property that every a is preceded with a_t .

We now show $\pi' \in \llbracket Z_{DC} \rrbracket$. The new trace π' does not jeopardise $\mathbf{DC}_J(J, a) = \neg \diamond (\downarrow a_t \wedge \exists a \wedge J < \ell)$, because for every occurrence (a_t^i, β_t^i, t^i) there is $(a^i, \beta^i, t^i + \delta^i)$ in π' , where $t^i - (t^i + \delta^i) \leq J$. We see that

$$\mathbf{DC}_{P^-}(P^-, a_t) = \neg \diamond (\downarrow a_t \wedge P^- > \ell \wedge \downarrow a_t)$$

holds, since in π' for every (a_t^i, β_t^i, t^i) and $(a_t^{i+1}, \beta_t^{i+1}, t^{i+1})$ we have $t^{i+1} - t^i \geq P^-$. Analogously, we get $t^{i+1} - t^i \leq P^+$ and, particularly, $t^0 \leq P^+$. By this, $\mathbf{DC}_{P^+}(P^+, a_t) = \neg \diamond (\exists a_t \wedge P^+ < \ell)$ is also satisfied.

With these observations, we get a trace $\pi' \in \llbracket Z_{DC} \rrbracket$ for every $\pi \in \llbracket CTA(Z) \rrbracket_{\Gamma}$ such that property (17) holds. Hence, $\pi \in \llbracket Z_{DC} \rrbracket_{\Gamma}$. \square