DOI 10.1007/s00165-008-0082-7

Formal Aspects of Computing (2008) 20: 481–505

Formal Aspects of Computing

Model checking Duration Calculus: a practical approach*

Roland Meyer[1], Johannes Faber[1], Jochen Hoenicke[2], and Andrey Rybalchenko[3]

[1] Department für Informatik, Carl von Ossietzky Universität Oldenburg, 26111 Oldenburg, Germany.
E-mail: roland.meyer@informatik.uni-oldenburg.de; johannes.faber@informatik.uni-oldenburg.de
[2] Institut für Informatik, Albert-Ludwigs Universität Freiburg, 79110 Freiburg, Germany.
E-mail: hoenicke@informatik.uni-freiburg.de
[3] Max Planck Institute for Software Systems, 66123 Saarbrücken, Germany.
E-mail: rybal@mpi-sws.mpg.de

Abstract. Model checking of real-time systems against Duration Calculus (DC) specifications requires the translation of DC formulae into automata-based semantics. The existing algorithms provide a limited DC coverage and do not support compositional verification. We propose a translation algorithm that advances the applicability of model checking tools to realistic applications. Our algorithm significantly extends the subset of DC that can be checked automatically. The central part of the algorithm is the automatic decomposition of DC specifications into sub-properties that can be verified independently. The decomposition is based on a novel distributive law for DC. We implemented the algorithm in a tool chain for the automated verification of systems comprising data, communication, and real-time aspects. We applied the tool chain to verify safety properties in an industrial case study from the European Train Control System (ETCS).

Keywords: Model checking; Verification; Duration Calculus; Timed automata; Real-time systems; European Train Control System; Case study

1. Introduction

Verification of embedded hardware and software systems requires reasoning about data, communication, and real-time aspects. All three dimensions can be expressed by the Duration Calculus (DC) [ZH04]. This logic supports reasoning about variables that are interpreted over dense real-time intervals. Extensions of DC take infinite data types and communication events into account.

A prominent approach to the verification of temporal specifications consists of translating the problem into an automata-theoretic setting and then applying graph-based algorithms [VW86]. To apply this approach for model checking DC, we need an algorithm that converts DC formulae into automata. This is a difficult task and it has been shown in [ZHS93] that it cannot be solved in general. Translation algorithms into automata-based semantics

Correspondence and offprint requests to: R. Meyer, E-mail: roland.meyer@informatik.uni-oldenburg.de
* This work was partly supported by the German Research Council under the grants SFB/TR 14 AVACS and GRK 1076/1. This is an extended version of a paper that appeared in *Theoretical Aspects of Computing - ICTAC 2006* [MFR06].

are only known for restricted classes of DC [Rav94, BLR95, Pan02, Frä04, FH07]. These algorithms are either not compositional or forbid the use of negation. They do not consider infinite data domains and communication aspects. Support of parameterised models and DC specifications as well as variables over unbounded data domains is crucial for the applicability of the algorithm in our target domain of train control systems. Moreover, the existing translations suffer from an exponential blow-up of the resulting automata in the number of DC operators, which hinders the applicability of verification tools.

In this article, we identify a new class of DC formulae, called *test formulae*, that can be translated into timed automata, also referred to as test automata. Test formulae significantly extend the previously known classes (by strictly subsuming them or being incomparable). They take communication aspects and infinite data domains into account. Furthermore, we identify a normal form representation of test formulae that provides a basis for the compositional verification. Our normal form decomposes a formula into disjuncts that are translated independently. This allows for an efficient verification as it reduces the size of the automata. The normal form is realised with a new operator for the DC that permits a distributive law of linear complexity. The translation yields a characterisation of the model checking problem in terms of reachability. The reachability problem is decidable under a dense time interpretation for models with finite data domains.

We implemented our translation algorithm in a tool chain that verifies real-time models against properties specified using our class of test formulae. We successfully applied our tool chain to large-scale real-world problems and verified the emergency procedure of the European Train Control System (ETCS) [ERT02]. Our approach is the first that permits model checking of a comprehensive ETCS fragment considering communication, infinite data, and real-time. In our case study, we applied the abstraction refinement model checker ARMC [PR07], which supports parameters and infinite data types.

To summarise our contributions, we identify a novel class of DC formulae and give a translation algorithm into enhanced timed automata [AD94]. Since a direct translation leads to an exponential blow-up of the automata, we give a normal form for our novel class to decompose given properties. We demonstrate the applicability of our approach by verifying a case study of the European Train Control System.

This paper is an extended version of [MFR06]. It differs from the preliminary paper in the following aspects. We give the full and constructive proof for the normal form. We explain the translation from test formulae into a timed automata-based representation in detail and give hints for the correctness proof. This construction is also published in [Hoe06], but the presentation has been improved significantly. We prove the correctness of the overall model checking procedure. We provide our case study's full ETCS model that has only been sketched in the previous paper and a detailed explanation of the verification undertaken.

The paper is organised as follows. After a short introduction to our case study, we introduce the DC and the applied automaton model, Phase Event Automata (PEA), in Sect. 2. The class of test formulae, the new operator, and the normal form are presented in Sect. 3. Based upon these results, Sect. 4 gives the test automata semantics and proves the correctness of our model checking approach. The model of our case study and our model checking results are presented in Sect. 5. Section 6, reviewing related work and suggesting future investigations, concludes the paper.

1.1. Application domain: train control systems

The emerging ETCS is an international standard [ERT02] that shall replace national train control systems to ensure cross-border interoperability and improve railway safety as well as track utilisation. In the final ETCS implementation level, the existing national track side systems for detection of train speed, location, and integrity will not be used anymore. Instead, a radio block centre (RBC) controls the traffic in a well-defined area. It ascertains in cooperation with the ETCS units installed in the trains their speed and position values. RBCs and trains communicate over a GSM-R radio connection. To increase the possible traffic density, the ETCS employs a moving block principle, by which the RBC grants to a train the authorisation to move up to a position closely behind the preceding train (cf. Fig. 1). In our case study, we analyse the emergency handling: in case of an unusual event or a mishap, the train control system has to stop all trains safely. The desired property in our case study is that the trains will never collide.

Verification approaches for safety requirements of industrial systems like the ETCS have to consider the identified dimensions: data, communication, and real-time. In this paper, we present the first verification of an ETCS fragment where all these aspects are considered.

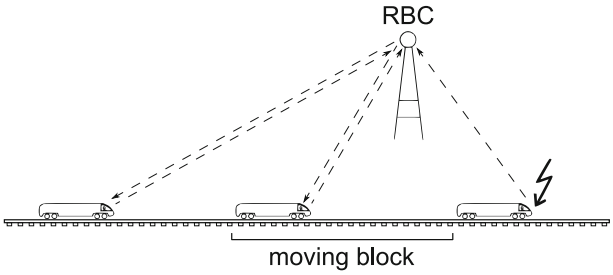

Fig. 1. Consecutive trains

2. Preliminaries

Since we translate DC formulae into Phase Event Automata (PEA), we review DC and PEA in this section.

2.1. Duration Calculus

Duration Calculus [ZH04] is an interval-based logic for the specification of real-time systems. We use dense real-time, $\mathbb{T}ime := \mathbb{R}_{\geq 0}$. To represent a system state at a point in time, DC uses *state expressions*. State expressions, denoted by φ, are quantifier-free first-order formulae over time-dependent variables, so-called *observables* ($X \in$) $SVar$. For every observable X there is a data domain $D(X)$. The semantics of an observable X is given by an *interpretation* $\mathcal{I}$ assigning a mapping $\mathcal{I}(X) : \mathbb{T}ime \to D(X)$ to the observable. Additionally, there are predicates $p_{/n}$ of arity $n \in \mathbb{N}$ with interpretations $\hat{p} : D(X_1) \times \ldots \times D(X_n) \to \mathbb{B}$.

The semantics of a state expression φ depends on the semantics of the observables. Given an interpretation $\mathcal{I}$ of the observables in φ, the semantics of φ is given by the mapping $\mathcal{I}[\![\varphi]\!] : \mathbb{T}ime \to \{0, 1\}$ as follows:

$$\begin{aligned} \mathcal{I}[\![p(X_1, \ldots, X_n)]\!](t) &:= 1 \text{ iff } \hat{p}(\mathcal{I}(X_1)(t), \ldots, \mathcal{I}(X_n)(t)) = tt \\ \mathcal{I}[\![\neg\varphi_1]\!](t) &:= 1 - \mathcal{I}[\![\varphi_1]\!](t) \\ \mathcal{I}[\![\varphi_1 \wedge \varphi_2]\!](t) &:= \mathcal{I}[\![\varphi_1]\!](t) \cdot \mathcal{I}[\![\varphi_2]\!](t). \end{aligned} \tag{1}$$

We require finite variability, i.e., for every predicate and every choice of observables the function in (1) has finitely many discontinuities on every finite interval. This ensures $\mathcal{I}[\![\varphi]\!]$ is (Riemann) integrable. The *class of DC formulae* ($F \in$) $\mathbb{F}orm$ is defined by

$$\mathbb{F}orm ::= \lceil\varphi\rceil \mid \ell \sim k \mid \neg\mathbb{F}orm \mid \mathbb{F}orm_1 \wedge \mathbb{F}orm_2 \mid \mathbb{F}orm_1 \,;\, \mathbb{F}orm_2 \mid \exists X : \mathbb{F}orm,$$

where $\sim \in \{\leq, <, =, >, \geq\}$, $k \in \mathbb{R}_{\geq 0}$. Given an interpretation $\mathcal{I}$ of the observables in state expressions, the semantics of a DC formula F evaluates the formula on a given finite interval:

$$\begin{array}{ll} \mathcal{I}, [b, e] \models \lceil\varphi\rceil & \text{iff } \int_b^e \mathcal{I}[\![\varphi]\!](t)\, dt = e - b \text{ and } e > b \\ \mathcal{I}, [b, e] \models \ell \sim k & \text{iff } (e - b) \sim k \\ \mathcal{I}, [b, e] \models \neg F & \text{iff } \mathcal{I}, [b, e] \not\models F \\ \mathcal{I}, [b, e] \models F_1 \wedge F_2 & \text{iff } \mathcal{I}, [b, e] \models F_1 \text{ and } \mathcal{I}, [b, e] \models F_2 \\ \mathcal{I}, [b, e] \models F_1 \,;\, F_2 & \text{iff there is } m \in [b, e] \text{ such that } \mathcal{I}, [b, m] \models F_1 \text{ and } \mathcal{I}, [m, e] \models F_2 \\ \mathcal{I}, [b, e] \models \exists X : F & \text{iff there is } \mathcal{I}' =_{\setminus X} \mathcal{I} \text{ such that } \mathcal{I}', [b, e] \models F. \end{array}$$

Two interpretations are *equal up to* X, $\mathcal{I}' =_{\setminus X} \mathcal{I}$, iff they coincide on all observables except X.

An interpretation $\mathcal{I}$ *satisfies a formula F from time zero*, $\mathcal{I} \models_0 F$, iff the formula holds on all intervals starting from zero, i.e., $\forall t \in \mathbb{R}_{\geq 0} : \mathcal{I}, [0, t] \models F$. Two formulae F_1, F_2 are *satisfiability equivalent* iff for any interpretation $\mathcal{I}$ it holds:

$$\exists t \in \mathbb{R}_{\geq 0} : \mathcal{I}, [0, t] \models F_1 \Leftrightarrow \exists t' \in \mathbb{R}_{\geq 0} : \mathcal{I}, [0, t'] \models F_2.$$

The definition of test formulae in Sect. 3 depends on the notion of events specifying changes in the values of Boolean observables (cf. transition formulae defined in [ZH04]). Let $\mathcal{E}$ be a Boolean observable. An *event* $\updownarrow\mathcal{E}$ is valid in a point interval $[t, t]$ iff the value of $\mathcal{E}$ changes at t. The formula $\not\updownarrow\mathcal{E}$ holds in $[t, t]$ iff the value of $\mathcal{E}$ does

not change at t. For an interval the *forbidden event* formula $\boxminus \mathcal{E}$ holds iff the value of $\mathcal{E}$ is constant in the given interval.

2.2. Phase Event Automata

Phase Event Automata (PEA) [HM05] are a class of timed automata [AD94] that describe the behaviour of state- and event-based systems. They serve as a bridge between the Duration Calculus and the model checker ARMC. A distinguished feature of PEA is their parallel composition that synchronises on both events and data variables. The *event and data variables* (denoted by A and V, respectively) directly correspond to the observables in DC. Furthermore, the automaton has the notion of *clocks* from timed automata. A clock is a special variable that is automatically incremented as time passes and that is not a DC observable. Let $\mathcal{L}(V)$ denote the set of first-order formulae over variables in V.

Definition 2.1 (**Phase Event Automaton**) A *Phase Event Automaton* is a tuple $\mathcal{A} = (P, V, A, C, E, s, I, P^0)$, where

- P is a finite set of *locations* with *initial locations* $P^0 \subseteq P$,
- V, A, C are finite sets of real-valued *state variables*, *events*, and real-valued *clocks*, respectively,
- $E \subseteq P \times \mathcal{L}(V \cup V' \cup A \cup C) \times \mathbb{P}(C) \times P$ is a set of *transitions*,
- $s : P \to \mathcal{L}(V)$ associates with each location a predicate that holds while the location is visited, and
- $I : P \to \mathcal{L}(C)$ associates with each location a *clock invariant*.

A tuple $(p_1, g, X, p_2) \in E$ represents a transition from p_1 to p_2 with a *guard* g over (possibly primed) variables, clocks, and events, and a set X of clocks that are reset. Primed variables v' denote the post-state of v whereas v always refers to the pre-state. In addition, we postulate the presence of a *stuttering transition* $(p, \bigwedge_{e \in A} \neg e \wedge \bigwedge_{v \in V} v' = v, \varnothing, p)$ for every location p. It allows the automaton to take a transition that does not change the current state.

The operational semantics of PEA is given by *runs*.

Definition 2.2 (**Run of a PEA**) A *run* of a PEA $\mathcal{A}$ is a (finite or infinite) sequence

$$\langle (p_0, \beta_0, \gamma_0, t_0, Y_0), (p_1, \beta_1, \gamma_1, t_1, Y_1), \dots \rangle,$$

with *locations* $p_i \in P$, *valuations of variables* β_i, *clock valuations* γ_i, *durations* $t_i > 0$, and *event sets* $Y_i \subseteq A$. Furthermore, we demand $p_0 \in P^0$, $\gamma_0(c) = 0$ for all clocks $c \in C$, $\beta_i \models s(p_i)$, and $\gamma_i + t_i \models I(p_i)$. For all transitions (p_i, g, X, p_{i+1}) we require $\beta_i, \beta'_{i+1}, \gamma_i + t_i, Y_i \models g$ and $\gamma_{i+1} = (\gamma_i + t_i)[X := 0]$. We denote the *set of all runs* of $\mathcal{A}$ by $\mathbf{Run}(\mathcal{A})$.

The intuition is that the automaton starts in p_0, the valuation of variables is given by β_0, and the valuation of clocks is given by γ_0. It stays in the location p_i for t_i seconds. During this time, the variables in V must not change their values that are given by β_i and that satisfy the state invariant $s(p_i)$. The values of the clocks when p_i is entered are given by γ_i. They are incremented by t_i when p_i is left. At that time the clock invariant $I(p_i)$ holds. With the transition from p_i to p_{i+1} the events in Y_i occur. When the transition is taken the guard g holds. Its unprimed variables are evaluated by β_i, the primed variables by β_{i+1}, and the clocks by $\gamma_i + t_i$. Furthermore, the clock valuation γ_{i+1} after the transition is computed from the valuation $\gamma_i + t_i$ by setting all clocks in X to zero.

PEA composed in parallel synchronise on common events and additionally on common variables. That is, a variable that occurs in both automata may only be changed if both automata agree. Clocks may not be shared between the automata. The automata can always step synchronously, because any automaton can always take its stuttering transitions.

Definition 2.3 (**Parallel composition**) The *parallel composition* of PEA $\mathcal{A}_1$ and $\mathcal{A}_2$ with $\mathcal{A}_l = (P_l, V_l, A_l, C_l, E_l, s_l, I_l, P^0_l)$ with *disjoint* clock sets, $C_1 \cap C_2 = \varnothing$, is given by

$$\mathcal{A}_1 \,||\, \mathcal{A}_2 := (P_1 \times P_2, V_1 \cup V_2, A_1 \cup A_2, C_1 \cup C_2, E, s_1 \wedge s_2, I_1 \wedge I_2, P^0_1 \times P^0_2),$$

where $((p_1, p_2), g_1 \wedge g_2, X_1 \cup X_2, (p'_1, p'_2)) \in E$ iff $(p_l, g_l, X_l, p'_l) \in E_l$ for $l = 1, 2$.

The expressiveness of PEA crucially depends on the constraints over the state variables: transition guards like $x' = x + 1$ directly allow for encoding counter machines into PEA. Thus, reachability is in general undecidable. However, if the data domain is finite and all transition guards and state invariants are decidable the reachability problem can be decided as follows. We compute the parallel product to get a single automaton. Then we build the following timed automaton [AD94]. For each location p of the PEA and each valuation β that satisfies the state invariant $s(p)$ we create a location (p, β) of the timed automaton. Its invariant is the clock invariant $I(p)$. For each transition $(p_1, g \wedge g_c, X, p_2)$ in the PEA, where g accesses only state and event variables and g_c only clock variables, and for each pair of valuations β_1 and β_2 such that $\beta_1 \cup \beta_2' \models g$ we introduce a transition labelled with g_c and reset set X between the locations (p_1, β_1) and (p_2, β_2) of the timed automaton. Then a location in the PEA is reachable if and only if one of its corresponding locations in the timed automaton is reachable. To decide the latter the algorithm in [AD94] can be used.

3. Test formulae

In this section, we introduce a novel subclass of DC we call *test formulae*, denoted by *Testform*. For test formulae we give an automata translation in Sect. 4. Applying the automata-theoretic approach [VW86], we can decide whether a system satisfies a negated test formula. Thus, test formulae may be interpreted as undesired system behaviour.

Test formulae extend the state of the art in DC model checking (cf. Sect. 6). In particular, our class contains formulae closed under Boolean operations. Hence, our automata translation has to handle negated DC formulae. Also, the construction sheds light upon the relationship between DC and automata-theoretic models.

We briefly discuss the structure of test formulae. To specify system states, we define *phase formulae* (in *Phase*). Phases may have a timing constraint restricting their duration ($\ell \sim k$), an invariant that holds while a system is in the phase ($\lceil \varphi \rceil$), and a set of events that are forbidden ($\boxminus \mathcal{E}$). From phases we construct *trace formulae* (in *Trace*) to specify system evolutions. Event specifications ($\updownarrow \mathcal{E}$ and $\not\updownarrow \mathcal{E}$, respectively) constrain the state changes of a system. Formulae in *Form* are Boolean combinations of traces. Therefore, our class permits the negation of traces, i.e., the negation of complex formulae containing the chop operator. *Test formulae* in *Testform* combine formulae in *Form* via disjunction, chop, and conjunction. This yields additional expressiveness, e.g., for specifying timing issues.

Definition 3.1 (***Testform***) The *class of test formulae Testform* is defined inductively:

$$\begin{aligned}
\mathit{Phase} &::= \ell > 0 \wedge \ell \sim k \mid \mathit{Phase} \wedge \lceil \varphi \rceil \mid \mathit{Phase} \wedge \boxminus \mathcal{E} \\
\mathit{Trace} &::= \mathit{Phase} \mid \updownarrow \mathcal{E} \mid \not\updownarrow \mathcal{E} \mid \mathit{Trace}_1 \,;\, \mathit{Trace}_2 \\
\mathit{Form} &::= \mathit{Trace} \mid \neg \mathit{Form} \mid \mathit{Form}_1 \wedge \mathit{Form}_2 \\
\mathit{Testform} &::= \mathit{Form} \mid \mathit{Testform}_1 \,;\, \mathit{Testform}_2 \mid \mathit{Testform}_1 \wedge \mathit{Testform}_2 \mid \mathit{Testform}_1 \vee \mathit{Testform}_2,
\end{aligned}$$

where $k \in \mathbb{R}_{>0}$, φ is a state expression, $\mathcal{E}$ is a Boolean observable, and $\sim \in \{=, \leq, <, >, \geq\}$. When $\ell > 0$ is the only time constraint, we use $\sim = =$. We demand the first element in a trace to be a phase.

For our case study, undesired behaviour is that the leading train sends an alert message, indicated by formula (2), but for longer than 6.5 time units neither the leading nor the following train applies the brakes, (3) and (4). Test formula (5) reflects the critical behaviour. Note that the required property is $\neg TF$:

$$\mathit{Warn} := \lceil \mathit{true} \rceil \,;\, \updownarrow \mathit{FrontTrain.send.Alert} \,;\, \lceil \mathit{true} \rceil \,;\, \updownarrow \mathit{RearTrain.receive.Warning} \tag{2}$$

$$\mathit{NoBrake}_1 := \boxminus \mathit{RearTrain.applyEB} \wedge \ell > 6.5 \tag{3}$$

$$\mathit{NoBrake}_2 := \boxminus \mathit{FrontTrain.applyEB} \wedge \ell > 6.5 \tag{4}$$

$$TF := \mathit{Warn} \,;\, (\mathit{NoBrake}_1 \wedge \mathit{NoBrake}_2) \,;\, \lceil \mathit{true} \rceil. \tag{5}$$

We argue that a logic based approach to verification is superior to approaches where undesired behaviour is expressed directly in terms of test automata. The benefit of DC is its conciseness. A negated trace comprising n phases requires in the worst case an automaton exponential in n. Thus, even for simple behaviour the modelling of test automata by hand is error-prone, a disadvantage the automated compilation overcomes.

3.1. Sync events

For arbitrary DC formulae F, G, H there is no distributive law between the chop operator and the conjunction, i.e., $F \,;\, (G \wedge H) \not\Leftrightarrow (F \,;\, G) \wedge (F \,;\, H)$. To recover some form of distributive law, we introduce *sync events* $\Updownarrow_{\mathcal{S}}$, i.e., distinguished events occurring only once. They can be used to uniquely identify a chop point. For sync events the following distributivity holds:

$$F \Updownarrow_{\mathcal{S}} (G \wedge H) \Leftrightarrow (F \Updownarrow_{\mathcal{S}} G) \wedge (F \Updownarrow_{\mathcal{S}} H). \tag{6}$$

Definition 3.2 (**Sync events**) Let F, G be DC formulae, $\mathcal{S}$ a Boolean observable not contained in F nor G. Let $\mathcal{I}$ be an interpretation, $b, e \in \mathbb{R}_{\geq 0}$, $b \leq e$. The *sync event* $F \Updownarrow_{\mathcal{S}} G$ is defined as follows:

$$\begin{aligned} \mathcal{I}, [b, e] \models F \Updownarrow_{\mathcal{S}} G :\Leftrightarrow \exists t \in [b, e] : (\mathcal{I}, [b, t] \models F) \wedge (\mathcal{I}, [t, e] \models G) \wedge \\ (\mathcal{I}, [t, t] \models \updownarrow \mathcal{S}) \wedge \left(\forall t' \in [0, t) \cup (t, \infty) : \mathcal{I}, [t', t'] \models \not\updownarrow \mathcal{S}\right). \end{aligned}$$

To introduce sync events to test formulae, Equivalence (7) in the following lemma allows the replacement of a chop operator with a fresh sync event not used in one of the formulae. Furthermore, an efficient distributivity between sync events and conjunctions is stated.

Lemma 3.3 (**Sync event introduction and linear distributivity**) Let $\mathcal{S}$ be a Boolean observable not contained in F, F_i, G, G_j. The following equivalences hold:

$$(F \wedge \ell > 0) \,;\, G \quad \Leftrightarrow \quad \exists \mathcal{S} : (F \Updownarrow_{\mathcal{S}} G) \tag{7}$$

$$\left(\bigwedge_{i=1}^{m} F_i\right) \Updownarrow_{\mathcal{S}} \left(\bigwedge_{j=1}^{n} G_j\right) \quad \Leftrightarrow \quad \bigwedge_{i=1}^{m} (F_i \Updownarrow_{\mathcal{S}} true) \wedge \bigwedge_{j=1}^{n} \left(\lceil true \rceil \Updownarrow_{\mathcal{S}} G_j\right). \tag{8}$$

By definition, events do not happen at time zero. Therefore, we require $\ell > 0$ in (7). Analogously, the phase before a sync event has a duration greater zero in (8), i.e., $\lceil true \rceil$ holds. The distributivity in Equivalence (8) results in $m + n$ conjuncted formulae compared to the distributivity in (6) resulting in $m * n$ formulae:

$$\bigwedge_{i=1}^{m} (F_i \Updownarrow_{\mathcal{S}} true) \wedge \bigwedge_{j=1}^{n} \left(\lceil true \rceil \Updownarrow_{\mathcal{S}} G_j\right) \Leftrightarrow \bigwedge_{i=1}^{m} \bigwedge_{j=1}^{n} \left(F_i \Updownarrow_{\mathcal{S}} G_j\right).$$

The introduction of sync events transforms a time-triggered real-time system specification using chopped formulae into an event-triggered specification with sync events replacing chops. Event-triggered system specifications allow for canonical operational semantics using labelled transitions whereas time-triggered specifications need some elaborate clock construction to represent the timing issues.

3.2. A normal form theorem for test formulae

The following normal form theorem shows that every test formula is equivalent with a disjunctive normal form (DNF) over traces. It thus decomposes the model checking problem to checking disjuncts. We comment on how this reduces the size of the test automata at the end of Sect. 4.

Theorem 3.4 (**Normal form theorem**) Every test formula is satisfiability equivalent with a formula of the form

$$\exists \mathcal{S}_{ijk} : \bigvee\nolimits_i \bigwedge\nolimits_j \mathcal{T}_{ij}, \tag{9}$$

$$\text{with} \quad \mathcal{T}_{ij} ::= Tr_{ij} \Updownarrow_{\mathcal{S}_{ij}} \lceil true \rceil \mid \lceil true \rceil \Updownarrow_{\mathcal{S}_{ij1}} Tr_{ij} \Updownarrow_{\mathcal{S}_{ij2}} \lceil true \rceil, \tag{10}$$

where Tr_{ij} are traces or negated traces and $\mathcal{S}_{ijk}$ are fresh Boolean observables.

We use the following lemma to prove the theorem. It states that every formula is equivalent with a formula which has almost the desired form.

Lemma 3.5 Every test formula is equivalent with a formula of the form (9) where a disjunct $\bigwedge_j \mathcal{T}_{ij}$ is 1. $\ell = 0$ or 2. of the form $\ell > 0 \wedge \bigwedge_j \mathcal{T}_{ij}$, where

$$\mathcal{T}_{ij} ::= Tr_{ij} \mid Tr_{ij} \Updownarrow_{\mathcal{S}} \lceil true \rceil \mid \lceil true \rceil \Updownarrow_{\mathcal{S}} Tr_{ij} \mid \lceil true \rceil \Updownarrow_{\mathcal{S}_1} Tr_{ij} \Updownarrow_{\mathcal{S}_2} \lceil true \rceil, \text{ with } Tr_{ij} \text{ (negated) traces.} \tag{11}$$

Proof We show the lemma by induction on the structure of test formulae. Throughout the proof we omit the indices in disjunctive normal forms. For the base case, we show that every formula in $F \in \mathit{Form}$ has an equivalent formula in the required form. Since F is a Boolean combination of traces, it has an equivalent representation in disjunctive normal form: $\bigvee \bigwedge Tr$, where Tr are traces or negated traces. This is equivalent with $\bigvee((\bigwedge Tr) \wedge (\ell > 0 \vee \ell = 0))$. Distributivity gives $\bigvee(((\bigwedge Tr) \wedge \ell > 0) \vee ((\bigwedge Tr) \wedge \ell = 0))$. Since a non-negated trace contains a phase with $\ell > 0$, it is not satisfiable on a point interval with $\ell = 0$. Therefore the second term, $(\bigwedge Tr) \wedge \ell = 0$, is only satisfiable if all Tr are negated traces. But in this case the term is equivalent with $\ell = 0$. This proves the base case.

For the induction step, we assume that $TF_l \in \mathit{Testform}$ is equivalent with $\exists \mathcal{S}_l : \bigvee \bigwedge \mathcal{T}_l$, for $l = 1, 2$. Consider the case $TF_1 \,;\, TF_2$. Applying the hypothesis to TF_1 gives $(\exists \mathcal{S}_1 : \bigvee \bigwedge \mathcal{T}_1) \,;\, TF_2$. Pulling the quantification outside yields $\exists \mathcal{S}_1 : ((\bigvee \bigwedge \mathcal{T}_1) ; TF_2)$. The distributivity of chop and disjunction leads to the formula $\exists \mathcal{S}_1 : \bigvee((\bigwedge \mathcal{T}_1) ; TF_2)$. We treat TF_2 analogously and get $\exists \mathcal{S}_1 : \exists \mathcal{S}_2 : \bigvee \bigvee((\bigwedge \mathcal{T}_1) \,;\, (\bigwedge \mathcal{T}_2))$.

We now show how to transform the chopped formula $(\bigwedge \mathcal{T}_1) \,;\, (\bigwedge \mathcal{T}_2)$ into the desired form. There are three cases. If $\bigwedge \mathcal{T}_1$ is $\ell = 0$, we can equivalently replace the chopped formula by $\bigwedge \mathcal{T}_2$ which is of the desired form by the induction hypothesis. Analogously in the case when $\bigwedge \mathcal{T}_2$ is $\ell = 0$. In the case both disjuncts are different from $\ell = 0$, both of them contain $\ell > 0$ by construction. This allows the introduction of a sync event. Thus $(\bigwedge \mathcal{T}_1) \,;\, (\bigwedge \mathcal{T}_2)$ is equivalent with $\exists \mathcal{S} : (\bigwedge \mathcal{T}_1) \Updownarrow_{\mathcal{S}} (\bigwedge \mathcal{T}_2)$. With the distributivity of sync events and conjunction, we have $\exists \mathcal{S} : \bigwedge(\mathcal{T}_1 \Updownarrow_{\mathcal{S}} \mathit{true}) \wedge \bigwedge(\lceil \mathit{true} \rceil \Updownarrow_{\mathcal{S}} \mathcal{T}_2)$. The existence of $\ell > 0$ in $(\bigwedge \mathcal{T}_2)$ allows us to replace the formulae $\mathcal{T}_1 \Updownarrow_{\mathcal{S}} \mathit{true}$ by $\mathcal{T}_1 \Updownarrow_{\mathcal{S}} \lceil \mathit{true} \rceil$. This yields $\exists \mathcal{S} : \bigwedge(\mathcal{T}_1 \Updownarrow_{\mathcal{S}} \lceil \mathit{true} \rceil) \wedge \bigwedge(\lceil \mathit{true} \rceil \Updownarrow_{\mathcal{S}} \mathcal{T}_2)$. Afterwards, we pull outside the existential quantification.

We now distinguish the cases for $\mathcal{T}$. If $\mathcal{T}$ is a trace or negated trace Tr, both $Tr \Updownarrow_{\mathcal{S}} \lceil \mathit{true} \rceil$ and $\lceil \mathit{true} \rceil \Updownarrow_{\mathcal{S}} Tr$ are of the form in (11). In the case $\mathcal{T} = \lceil \mathit{true} \rceil \Updownarrow_{\mathcal{S}'} Tr$, we have that $\lceil \mathit{true} \rceil \Updownarrow_{\mathcal{S}'} Tr \Updownarrow_{\mathcal{S}} \lceil \mathit{true} \rceil$ is of the form in (11). For $\lceil \mathit{true} \rceil \Updownarrow_{\mathcal{S}} \lceil \mathit{true} \rceil \Updownarrow_{\mathcal{S}'} Tr$ we apply the distributivity in Equivalence (8) which gives $(\lceil \mathit{true} \rceil \Updownarrow_{\mathcal{S}} \lceil \mathit{true} \rceil \Updownarrow_{\mathcal{S}'} \mathit{true}) \wedge (\lceil \mathit{true} \rceil \Updownarrow_{\mathcal{S}'} Tr)$. Again we argue that we can equivalently replace true by $\lceil \mathit{true} \rceil$. Both formulae, $\lceil \mathit{true} \rceil \Updownarrow_{\mathcal{S}} \lceil \mathit{true} \rceil \Updownarrow_{\mathcal{S}'} \lceil \mathit{true} \rceil$ and $\lceil \mathit{true} \rceil \Updownarrow_{\mathcal{S}'} Tr$, are of the desired form. The cases $Tr \Updownarrow_{\mathcal{S}} \lceil \mathit{true} \rceil$ and $\lceil \mathit{true} \rceil \Updownarrow_{\mathcal{S}_1} Tr \Updownarrow_{\mathcal{S}_2} \lceil \mathit{true} \rceil$ follow analogously.

In the case $TF_1 \wedge TF_2$, the hypothesis gives $(\exists \mathcal{S}_1 : \bigvee \bigwedge \mathcal{T}_1) \wedge (\exists \mathcal{S}_2 : \bigvee \bigwedge \mathcal{T}_2)$. We pull outside both quantifications and use the distributivity of disjunction and conjunction. This equivalently gives $\exists \mathcal{S}_1 : \exists \mathcal{S}_2 : \bigvee \bigvee((\bigwedge \mathcal{T}_1) \wedge (\bigwedge \mathcal{T}_2))$. By the hypothesis all $\mathcal{T}_i$ are of the form in (11) which proves the case.

In the case $TF_1 \vee TF_2$, we only pull outside the existential quantifiers. This concludes the proof. □

Proof of Theorem 3.4 Consider the test formula TF. It is satisfiability equivalent with $TF \,;\, \lceil \mathit{true} \rceil$. We replace TF with the equivalent formula from Lemma 3.5, i.e., a formula $\exists \mathcal{S} : \bigvee \bigwedge \mathcal{T}$ satisfying the requirements. This yields $(\exists \mathcal{S} : \bigvee \bigwedge \mathcal{T}) \,;\, \lceil \mathit{true} \rceil$. We pull the quantification outside and use the distributivity of chop and disjunction to equivalently get $\exists \mathcal{S} : \bigvee((\bigwedge \mathcal{T}) \,;\, \lceil \mathit{true} \rceil)$. In the case the disjunct is $\ell = 0$, the formula $\ell = 0 \,;\, \lceil \mathit{true} \rceil$ is equivalent with $\lceil \mathit{true} \rceil \,;\, \lceil \mathit{true} \rceil$ which is equivalent with $\exists \mathcal{S} : \lceil \mathit{true} \rceil \Updownarrow_{\mathcal{S}} \lceil \mathit{true} \rceil$ via (7). This yields a formula of the form (10). In the case we have a proper disjunct, a sync event is introduced giving $\exists \mathcal{S} : (\bigwedge \mathcal{T}) \Updownarrow_{\mathcal{S}} \lceil \mathit{true} \rceil$. We pull the existential quantification outside. Via Distributivity (8) the remaining formula is equivalent with $\bigwedge(\mathcal{T} \Updownarrow_{\mathcal{S}} \lceil \mathit{true} \rceil) \wedge (\lceil \mathit{true} \rceil \Updownarrow_{\mathcal{S}} \lceil \mathit{true} \rceil)$. A sync event implies a length of the first interval greater zero, thus we can remove the sub-formula $\lceil \mathit{true} \rceil \Updownarrow_{\mathcal{S}} \lceil \mathit{true} \rceil$. That all $\mathcal{T} \Updownarrow_{\mathcal{S}} \lceil \mathit{true} \rceil$ can be brought into the form in (10) follows from the proof of Lemma 3.5. □

The computation of the DNFs and the known distributivities may lead to an exponential blow up of TF. We tackle this problem by model checking all disjuncts separately. Distributivity (8) neither increases the number of (negated) traces nor the size of the product automata (cf. restriction, Sect. 4).

To continue the example above, we gain the normal form of formula (5) by introducing two sync events (7) and using the distributivity of sync events and conjunctions (8):

$$\begin{aligned} & \mathit{Warn} \,;\, (\mathit{NoBrake}_1 \wedge \mathit{NoBrake}_2) \,;\, \lceil \mathit{true} \rceil \\ \Leftrightarrow\ & \exists \mathcal{S}_0 : \exists \mathcal{S}_1 : (\mathit{Warn} \Updownarrow_{\mathcal{S}_0} \lceil \mathit{true} \rceil) \wedge \textstyle\bigwedge_{i=1,2} (\lceil \mathit{true} \rceil \Updownarrow_{\mathcal{S}_0} \mathit{NoBrake}_i \Updownarrow_{\mathcal{S}_1} \lceil \mathit{true} \rceil). \end{aligned}$$

4. Model checking with test automata

To define whether a PEA model of a system satisfies a test formula, we relate interpretations of observables and runs of PEA. Given Boolean observables $\mathcal{E}_1, \ldots, \mathcal{E}_n$ and observables $X_1, \ldots, X_m$, an interpretation $\mathcal{I}$ is said to *fit to a run r* iff

- the set of events in the PEA is included in the set of Boolean observables $\{\mathcal{E}_1, \ldots, \mathcal{E}_n\}$, the set of variables in the PEA is a subset of the observables $\{X_1, \ldots, X_m\}$,
- the observables used in the PEA are interpreted as imposed by the valuations in the run,
- a change in the interpretation of a Boolean observable $\mathcal{E}_i$ occurs at time t iff the PEA changes its state at time t and the variable is contained in the set of events, i. e., $\mathcal{E}_i \in Y$.

Every run of a PEA induces a fitting interpretation. *Satisfaction* of a formula by a PEA is defined over the interpretations fitting to the runs of the automaton.

Definition 4.1 A PEA $\mathcal{A}$ *satisfies* a DC formula F, denoted by $\mathcal{A} \models_0 F$, iff all interpretations $\mathcal{I}$ fitting to a run r satisfy the formula from time zero:

$$\mathcal{A} \models_0 F :\Leftrightarrow \forall \mathcal{I} : \forall r \in \mathbf{Run}(\mathcal{A}) : (\mathcal{I} \text{ fits to } r \Rightarrow \mathcal{I} \models_0 F).$$

With this notion of satisfaction, we establish the correctness of compositional verification for PEA in Theorem 4.2: local properties hold for the entire system. More precisely, if a property holds for an automaton $\mathcal{A}_1$, then it holds for any composition $\mathcal{A}_1 \parallel \mathcal{A}_2$.

Theorem 4.2 Given PEA $\mathcal{A}_1$, $\mathcal{A}_2$, and a DC formula F. If $\mathcal{A}_1 \models_0 F$ then $\mathcal{A}_1 \parallel \mathcal{A}_2 \models_0 F$.

Proof The automaton $\mathcal{A}_1 \parallel \mathcal{A}_2$ satisfies F iff every interpretation $\mathcal{I}$ that fits to a run r of $\mathcal{A}_1 \parallel \mathcal{A}_2$ satisfies F. Consider such an interpretation $\mathcal{I}$ which fits to the run r. Since r is a run of the composition $\mathcal{A}_1 \parallel \mathcal{A}_2$, it is in particular a run of $\mathcal{A}_1$. Thus, $\mathcal{I}$ is an interpretation that fits to a run of $\mathcal{A}_1$. Since $\mathcal{A}_1 \models_0 F$, $\mathcal{I}$ satisfies F from zero. □

4.1. Power set construction for traces

To begin with, we sketch a non-compositional construction that builds a *deterministic* PEA for a trace formula. A trace tr consists of several consecutive phases. Each phase can have a state expression φ_i, a time restriction denoted by $\ell \sim k_i$, $\sim \in \{\varnothing, <, \leq, \geq, >\}$, and a conjunction of forbidden events $\boxminus \mathcal{E}_1 \wedge \ldots \wedge \boxminus \mathcal{E}_n$. This set of events is denoted by $\mathcal{F}\mathcal{E}_i = \{\mathcal{E}_1, \ldots, \mathcal{E}_n\} \subseteq A$. Before the i-th phase there can be some event specification, $\phi(\mathcal{E}_1); \ldots; \phi(\mathcal{E}_m)$, where $\phi(\mathcal{E}_j) = \updownarrow \mathcal{E}_j$ or $\phi(\mathcal{E}_j) = \not\updownarrow \mathcal{E}_j$. This specification is equivalent with the conjunction $\phi(\mathcal{E}_1) \wedge \ldots \wedge \phi(\mathcal{E}_m)$, which is denoted by $\mathcal{E}_{entry,i}$. We say that the i-th phase *requires an entry event* if $\mathcal{E}_{entry,i}$ contains $\updownarrow \mathcal{E}_j$. The event specifications after the last phase are exit events of the trace and their conjunction is denoted by $\mathcal{E}_{exit}$. Thus our trace looks like this:

$$tr = \left(\lceil \varphi_1 \rceil \wedge \bigwedge_{\mathcal{E} \in \mathcal{F}\mathcal{E}_1} \boxminus \mathcal{E} \wedge 0 < \ell \sim k_1 \right); \mathcal{E}_{entry,2}; \ldots; \left(\lceil \varphi_n \rceil \wedge \bigwedge_{\mathcal{E} \in \mathcal{F}\mathcal{E}_n} \boxminus \mathcal{E} \wedge 0 < \ell \sim k_n \right); \mathcal{E}_{exit}.$$

It is straightforward to build a nondeterministic PEA by introducing a location for each phase of the trace that ensures $\lceil \varphi_i \rceil$ holds and no forbidden events occur. These locations are connected by transitions that check that the entry events occur. A location of the automaton is reached in a run if there is an interpretation that fits to the run and that satisfies the prefix of the trace ending with the corresponding phase. However, to allow the negation of traces we need to construct a deterministic PEA. The idea is similar to the power set construction of a deterministic finite automaton from a nondeterministic one. For finite automata the locations of the deterministic automaton are labelled with the sets of locations which the nondeterministic automaton can reach for a given word. In our setting, the locations of the nondeterministic automaton correspond to the phases of the trace. Thus, we label the locations with sets of phases.

With $Prefix(tr, i)$ we denote the prefix of the trace tr ending with the i-th phase. If the trace has a timing constraint $\ell \sim k_i$ on the i-th phase, we denote with $Prefix^{\leq}(tr, i)$ the prefix where this timing constraint is replaced with $\ell \leq k_i$. Note, that only the last timing constraint is replaced.

To simplify our presentation, we give the construction for traces containing only strict time bounds ($<$, $>$). In [Hoe06] the general construction is given in full detail. The locations P of the deterministic PEA are the sets in $\mathbb{P}(\{1, \ldots, n, 1^{>}, \ldots, n^{>}\})$, thus it may grow exponentially in the number of phases in the trace. The basic idea is to construct a deterministic automaton with the following properties. The location p the automaton visits at time t in a run contains those phases i of the trace for which the fitting interpretations in the interval $[0, t]$ satisfy $Prefix(tr, i)$. However, for phases with an upper bound, that bound only has to hold weakly, i. e., they

satisfy $Prefix^{\leq}(tr, i)$. Furthermore, the location contains $i^{>}$ for those phases i that have a lower bound and where $Prefix^{\leq}(tr, i) \wedge \neg\, Prefix(tr, i)$ holds, i. e., the minimum duration of phase i has not been reached. We also say that a phase i is *active* if the current location p contains either i or $i^{>}$. A location p of the automaton never contains i and $i^{>}$ simultaneously.

Particular care is needed for ensuring the length constraints on the phases. Since the time domain is dense, the automaton cannot remember all possible durations for which a phase may be active. The trick we use in our construction is to remember only the infimum or the supremum of the possible values. We do this depending on whether the phase has an upper or a lower bound. For each phase i with an *upper* bound, there is a clock c_i measuring the *infimum* of the amount of time that has been spent in the phase i. To be more precise, if the location p contains the phase i then for every time t when p is visited in a run to which an interpretation $\mathcal{I}$ fits the value of clock c_i is

$$\inf\left\{ t - t' \mid \mathcal{I}, [0, t'] \models Prefix(tr, i-1) \wedge \mathcal{I}, [t', t] \models \mathcal{E}_{entry,i} \,;\, \left(\lceil \varphi_i \rceil \wedge \bigwedge_{\mathcal{E} \in \mathcal{FE}_i} \boxminus \mathcal{E} \right) \right\}.$$

However, there is a special case when the infimum is zero. This is explained after the *seep* formula, below.
For each phase i with a *lower* bound, there is a clock c_i measuring the *supremum* of the amount of time that has been spent in this phase: if $i^{>} \in p$, then the value of clock c_i is

$$\sup\left\{ t - t' \mid \mathcal{I}, [0, t'] \models Prefix(tr, i-1) \wedge \mathcal{I}, [t', t] \models \mathcal{E}_{entry,i} \,;\, \left(\lceil \varphi_i \rceil \wedge \bigwedge_{\mathcal{E} \in \mathcal{FE}_i} \boxminus \mathcal{E} \right) \right\}.$$

Note, that this supremum is at most k_i, since $Prefix^{\leq}(tr, i)$ holds.

In Duration Calculus with dense time, the formula $\lceil \varphi_1 \wedge \varphi_2 \rceil \Rightarrow (\lceil \varphi_1 \rceil \,;\, \lceil \varphi_2 \rceil)$ holds. So if the latter trace is given and $\varphi_1 \wedge \varphi_2$ holds, the automaton should not only enter the $\lceil \varphi_1 \rceil$-phase but also the $\lceil \varphi_2 \rceil$-phase. Therefore, in the construction it is possible to enter two or more adjacent phases simultaneously. This happens if there is no entry event required for the second phase, there is no lower bound for the first phase, and the phase formulae φ_i for both phases hold. In this case, we say the automaton *seeps* through the first phase into the second phase. The following formula captures this notion and states that in location p seeping into phase i is possible if $i - 1 \in p$ and no entry event is required for phase i:

$$seep(i, p) = \begin{cases} false & \text{if } i \text{ requires an entry event,} \\ i - 1 \in p & \text{otherwise.} \end{cases}$$

If for some location p and a phase i with an upper bound $seep(i, p)$ (thus $i - 1 \in p$) and $i \in p$ hold, then the chop point between phase $i - 1$ and phase i can be set to an arbitrarily large time. This means, phase i is visited as late as possible. Thus the infimum

$$\inf\left\{ t - t' \mid \mathcal{I}, [0, t'] \models Prefix(tr, i-1) \wedge \mathcal{I}, [t', t] \models \mathcal{E}_{entry,i} \,;\, \left(\lceil \varphi_i \rceil \wedge \bigwedge_{\mathcal{E} \in \mathcal{FE}_i} \boxminus \mathcal{E} \right) \right\}$$

is zero as long as the automaton stays in location p. Therefore, there is no need to use the clock c_i in a location for which $seep(i, p)$ holds.

The following function *complete* denotes that the automaton has detected that $Prefix(tr, i)$ holds. This can only be the case if $i \in p$. For a phase with an upper bound $\ell < k_i$ we additionally require that $c_i < k_i$ or $seep(i, p)$ holds. In the latter case, the duration of the ith phase can be arbitrarily small.

$$complete(i, p) \Leftrightarrow \begin{cases} i \in p \wedge (seep(i, p) \vee c_i < k_i) & \text{iff } i \text{ has an upper bound,} \\ i \in p & \text{otherwise.} \end{cases}$$

There are three reasons a phase i is active after a transition of the automaton. Either the phase is entered from the previous phase, or the automaton seeps into the current phase, or the phase was already active and is kept

active. The following formulae denote the cases where a phase is entered or kept. In the formulae, i denotes the phase under consideration and p the previous location:

$$enter(i,p) = complete(i-1,p) \wedge \mathcal{E}_{entry,i},$$
$$keep(i,p) = (i^{>} \in p \vee complete(i,p)) \wedge \bigwedge_{\mathcal{E} \in \mathcal{FE}_i} \neg\, \mathcal{E}.$$

These functions serve to define the transitions in the automaton. The transition set consists of tuples (p, g, X, p') where p is the source, p' the destination of the transition, and X is the set of clocks that are reset. The formula g is

$$\bigwedge_{i \in \{1,\ldots,n\}} \left(\begin{array}{l} \big((i \in p' \vee i^{>} \in p') \Leftrightarrow (enter(i,p) \vee seep(i,p') \vee keep(i,p)) \wedge \varphi_i\big) \\ \wedge \left(c_i \in X \Leftrightarrow \begin{cases} (enter(i,p) \vee seep(i,p')) \wedge \varphi_i \wedge \neg\, keep(i,p) & \text{if } i \text{ has lower bound,} \\ \neg\, seep(i,p') \wedge & \\ ((seep(i,p) \wedge keep(i,p)) \vee enter(i,p)) \wedge \varphi_i & \text{if } i \text{ has upper bound,} \\ false & \text{otherwise,} \end{cases} \right) \\ \wedge \left(\begin{cases} i \in p' \Leftrightarrow keep(i,p) \wedge \varphi_i \wedge (i \in p \vee c_i \geq k_i) & \text{if } i \text{ has lower bound,} \\ i^{>} \notin p' & \text{otherwise.} \end{cases} \right) \end{array} \right)$$

The first line states that a phase i is active in the successor location ($i \in p'$ or $i^{>} \in p'$) iff *enter*, *seep*, or *keep* hold and the state expression φ_i is true. Note that seeping is possible if the previous phase $i-1$ is also in the *successor* location p', while for enter and keep the previous location p matters. The second to fourth lines define the different cases when a clock is reset. For phases with a lower bound, the clock should contain the supremum, therefore it is only reset if it is not possible to stay in the phase, i. e., $\neg$ *keep* holds. For phases with an upper bound, the clock should contain the infimum. We argued above that the clock value is ignored if $seep(i,p')$ holds. Therefore there is no need to reset the clock in this case. Otherwise, if $seep(i,p)$ holds for the *previous* location p, the infimum is zero at the time the transition is taken. Therefore, if the automaton stays in phase i ($keep(i,p)$ holds) the clock is reset. It is also reset if the phase i is entered, e. g., with an entry event. Finally, for a lower bound phase i, the automaton is in a location p' with $i \in p'$ only if the phase is active for at least the duration k_i. This is only the case if it was active before (denoted $keep(i,p)$) and $i \in p$ holds for the previous location, or the clock c_i has reached the bound k_i. Note that when evaluating the above formula for a given p, X, and p', most of the atoms simplify to *true* and *false*, respectively. In many cases the whole formula can be simplified to *true* or *false*.

The state invariant makes sure that for all active phases φ_i hold. Furthermore, if a phase i is not active but $seep(i,p)$ holds, then its state invariant must be violated, because otherwise it would be immediately activated. The clock invariant makes sure that $c_i \leq k_i$ for all phases $i \in p$ with an upper bound and all phases i with $i^{>} \in p$. This guarantees that $Prefix^{\leq}(tr, i)$ holds for these locations:

$$s(p) = \left(\bigwedge_{i \in p \vee i^{>} \in p} \varphi_i \right) \wedge \left(\bigwedge_{i, i^{>} \notin p \wedge seep(i,p)} \neg\, \varphi_i \right)$$
$$I(p) = \left(\bigwedge_{i \in p, \neg\, seep(i,p), i \text{ has upper bound}} c_i \leq k_i \right) \wedge \left(\bigwedge_{i^{>} \in p} c_i \leq k_i \right).$$

The initial locations of the automaton are (1) the locations labelled with $\{1, 2, \ldots, i\}$, where all phases do not have a lower bound and do not require entry events and (2) the location $\{1, \ldots, i-1, i^{>}\}$, where i has a lower bound, phases $1, \ldots, i-1$ have no lower bound, and all phases do not require entry events.

The PEA for the trace

$$\lceil true \rceil \,;\, \lceil \varphi_1 \rceil \,;\, 0 < \ell < k \,;\, \lceil \varphi_2 \rceil \tag{12}$$

is depicted in Fig. 2 in the rectangular area. The trace formula holds if there is a change from a state satisfying φ_1 to a state satisfying φ_2 in less than k time units. In the context of our case study, this can be a movement of the train that should not be possible due to its limited speed. In the diagram the locations p with $2 \in p$, $3 \notin p$ are omitted since their invariant $s(p)$ is *false*. The locations with $1 \notin p$ are also omitted since they are not reachable. Furthermore, most of the guards are identical to the invariant of the destination location so they are omitted, too.

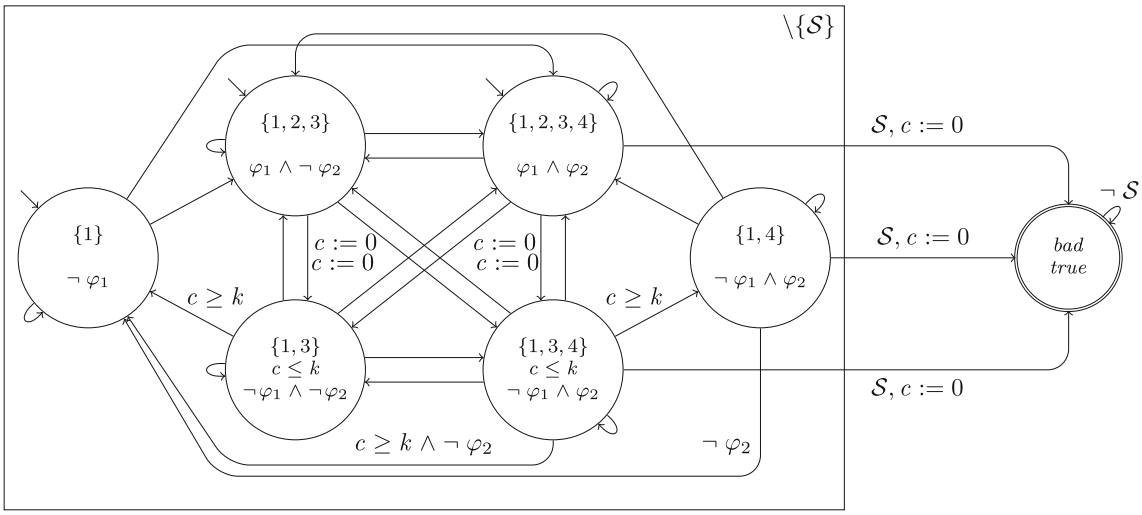

Fig. 2. The test automaton for $(\lceil true \rceil \,;\, \lceil \varphi_1 \rceil \,;\, 0 < \ell < k \,;\, \lceil \varphi_2 \rceil) \Updownarrow_{\mathcal{S}} \lceil true \rceil$

Lemma 4.3 (**Deterministic construction**) The automaton produced by the construction sketched above is deterministic, i. e., for a location p, clock valuation γ, event set Y, and state β, there is a unique successor p' and a unique clock reset set X, such that for the corresponding transition (p, g, X, p') the guard formula g holds.

Proof We exploit the shape of the guard formula g. It is a conjunction of biimplications, that define whether $i, i^{>} \in p'$ and whether $c_i \in X$. Care has to be taken since the right-hand-sides of the biimplications contain p' in the formula $seep(i, p')$. However, $seep(i, p')$ only depends on $i - 1 \in p'$, which can be computed in an earlier step. Therefore, it is possible to compute p' and X by starting with $i = 1$ and evaluating the truth values for $i \in p'$, $i^{>} \in p'$ and $i \in X$. Afterwards this is repeated for $i = 2$ and so on. It is easy to see that the p' and X computed this way correspond to the unique transition for which g is true. □

The interpretation satisfies the whole trace tr iff the last phase n is complete and all exit events occur. This is formalised as

$$complete_{tr}(p) \Leftrightarrow complete(n, p) \wedge \mathcal{E}_{exit}. \tag{13}$$

Note that $complete(n, p)$ is either $true$, $false$, or $c_n < k_n$.

Lemma 4.4 (**Power set construction**) For every trace tr there are a deterministic PEA $\mathcal{P}(tr)$ and a formula $complete_{tr}(p)$ for every location p in $\mathcal{P}(tr)$ such that for each interpretation $\mathcal{I}$ fitting to a run $\langle (p_0, \beta_0, \gamma_0, t_0, Y_0), (p_1, \beta_1, \gamma_1, t_1, Y_1), \ldots \rangle \in \mathbf{Run}(\mathcal{P}(tr))$

$$\mathcal{I}, \left[0, \sum_{j=0}^{k} t_j \right] \models tr \quad \text{if and only if} \quad \gamma_k + t_k, Y_k \models complete_{tr}(p_k).$$

The proof is given in [Hoe06].

4.2. Test automata

Test automata (TA) are PEA with a distinguished location, called the *bad state*. The runs of a TA are the runs of the underlying PEA. A run is said to be a *test run* iff it reaches the bad state. Reaching the bad state in the parallel composition of a system with a TA means that the system can exhibit the undesired behaviour specified in the test formula the TA is constructed for.

We define the TA semantics for the normal form of test formulae. Therefore, we require three operations on TA: parallel composition to express the conjunction, sequential composition to represent the formula structure in (10), and restriction to model sync events.

The *parallel composition* of test automata, $TA_1 \parallel TA_2$, takes the parallel composition of the underlying PEA and defines the bad state of the composed automaton to be the pair of the bad states in the component automata.

The *sequential composition* of two test automata, denoted by $TA_1 \bullet_{\mathcal{S},\gamma} TA_2$, means the second TA is started when the first one has accepted its formula. Since the acceptance of trace formulae may depend on clock valuations, we cannot use bad states to check the acceptance in the TA for traces. Instead, we use a guard function γ yielding a first-order formula for every location. Precisely, we define the sequential composition as follows. A transition between every location p in the first automaton and every initial location p' in the second automaton is inserted. The new transition demands $\mathcal{S} \wedge \gamma(p)$, where the event $\mathcal{S}$ represents the sync event in (10) and $\gamma(p)$ holds iff the test formula represented by the first TA is satisfied. The function γ is given by $complete_{tr}$ (respectively $\neg complete_{tr}$) as defined in (13). Furthermore, the transition resets all clocks.

Given a test automaton TA, the *restriction of TA to the event* $\mathcal{S}$, denoted by $TA \backslash \{\mathcal{S}\}$, is defined by TA with the guards of the transitions changed: if the guard does not contain $\mathcal{S}$ in TA, the requirement $\neg\mathcal{S}$ is added in $TA \backslash \{\mathcal{S}\}$, otherwise, the transition remains unchanged. The restriction operator is used to make the occurrences of sync events unique.

As an example, consider the formula $tr \Updownarrow_{\mathcal{S}} \lceil true \rceil$, where tr is the trace in (12). Figure 2 represents the corresponding TA: $\big(\mathcal{P}(tr) \bullet_{\mathcal{S}, complete_{tr}} \mathcal{P}(\lceil true \rceil)\big) \backslash \{\mathcal{S}\}$. The trace automaton $\mathcal{P}(tr)$ is connected with the automaton $\mathcal{P}(\lceil true \rceil)$, the composed automaton is restricted to the event $\mathcal{S}$. For $\mathcal{P}(tr)$ the restriction is only indicated. Moreover, transitions with guard *false* are removed.

4.3. A test automata semantics for test formulae

We now define a test automata semantics for test formulae, i.e., a mapping that assigns to each test formula (in normal form, cf. Theorem 3.4) TF a test automaton $\mathcal{P}(TF)$. The disjunction in the normal form is not lifted to automata level but model checking is done stepwise for all disjuncts until a satisfied disjunct is found.

Definition 4.5 (**Test automata semantics**) The test automata semantics for a test formula in normal form TF yields a PEA $\mathcal{P}(TF)$ defined as follows:

$$\begin{aligned}
\mathcal{P}(tr \Updownarrow_{\mathcal{S}} \lceil true \rceil) &:= \big(\mathcal{P}(tr) \bullet_{\mathcal{S}, complete_{tr}} \mathcal{P}(\lceil true \rceil)\big) \backslash \{\mathcal{S}\} \\
\mathcal{P}(\neg tr \Updownarrow_{\mathcal{S}} \lceil true \rceil) &:= \big(\mathcal{P}(tr) \bullet_{\mathcal{S}, \neg complete_{tr}} \mathcal{P}(\lceil true \rceil)\big) \backslash \{\mathcal{S}\} \\
\mathcal{P}(\lceil true \rceil \Updownarrow_{\mathcal{S}_1} Tr \Updownarrow_{\mathcal{S}_2} \lceil true \rceil) &:= \big[\big(\mathcal{P}(\lceil true \rceil) \bullet_{\mathcal{S}_1, true} \mathcal{P}(Tr \Updownarrow_{\mathcal{S}_2} \lceil true \rceil)\big) \backslash \{\mathcal{S}_2\}\big] \backslash \{\mathcal{S}_1\} \\
\mathcal{P}(\bigwedge_{i_1} \mathcal{T}_{i_1} \wedge \bigwedge_{i_2} \mathcal{T}_{i_2}) &:= \mathcal{P}(\bigwedge_{i_1} \mathcal{T}_{i_1}) \parallel \mathcal{P}(\bigwedge_{i_2} \mathcal{T}_{i_2}),
\end{aligned}$$

where tr is a trace, Tr a trace or a negated trace, $\mathcal{S}, \mathcal{S}_1, \mathcal{S}_2$ are Boolean observables, and $\mathcal{T}_{i_l}$ are of the form in Eq. (10).

A test formula is satisfied by an interpretation on an interval iff the bad state in the TA is reachable in a run the interpretation fits to. This characterisation is essential in the correctness proof of our model checking procedure.

Lemma 4.6 (**Characterisation of satisfaction with test automata**) Consider the normal form $\exists \mathcal{S}_{ijk} : \bigvee_i \bigwedge_j \mathcal{T}_{ij}$ of a test formula. Given an interpretation $\mathcal{I}$ and $t \in \mathbb{R}_{\geq 0}$, the following equivalence holds for every disjunct:

$$\mathcal{I}, [0, t] \models \textstyle\bigwedge_j \mathcal{T}_{ij} \Leftrightarrow \exists \text{ test run } r \in \mathbf{Run}(\mathcal{P}(\bigwedge_j \mathcal{T}_{ij})) : \mathcal{I} \text{ fits to } r \text{ and } r \text{ reaches the bad state at time } t.$$

Proof We prove the case $tr \Updownarrow_{\mathcal{S}} \lceil true \rceil$ and comment on the other base cases. Let $\mathcal{I}, [0, t] \models tr \Updownarrow_{\mathcal{S}} \lceil true \rceil$. By definition there is $t' \in]0, t[$ such that $\mathcal{I}, [0, t'] \models tr$ and $\mathcal{I}, [t', t'] \models true \Updownarrow_{\mathcal{S}} true$ and $\mathcal{I}, [t', t] \models \lceil true \rceil$. According to Lemma 4.3, $\mathcal{P}(tr)$ is deterministic. Thus, the interpretation $\mathcal{I}$ induces a run it fits to:

$$r' := \langle (p_0, \beta_0, \gamma_0, t_0, Y_0), (p_1, \beta_1, \gamma_1, t_1, Y_1), \ldots \rangle \in \mathbf{Run}(\mathcal{P}(tr)).$$

The closure of runs under stuttering steps allows us to assume the existence of $k, k_t \in \mathbb{N}_0$ with $k < k_t$ and $\Sigma_{i=0}^{k} t_i = t'$, $\Sigma_{i=0}^{k_t} t_i = t$. We construct a run r of the composed automaton. For all clocks c, the new evaluations are $\hat{\gamma}_j(c) := 0$, if $j = k + 1$ and $\hat{\gamma}_j(c) := \hat{\gamma}_{j-1} + t_{j-1}$, otherwise. The new run is

$$r := \langle (p_0, \beta_0, \gamma_0, t_0, Y_0), \ldots, (p_k, \beta_k, \gamma_k, t_k, Y_k \cup \{\mathcal{S}\}), (p_{true}, \beta_{k+1}, \hat{\gamma}_{k+1}, t_{k+1}, Y_{k+1}), \ldots \rangle.$$

We argue that $r \in \mathbf{Run}(\mathcal{P}(tr \Updownarrow_{\mathcal{S}} \lceil true \rceil))$. With Lemma 4.4 $\mathcal{I}, [0, t'] \models tr$ implies $\gamma_k + t_k, Y_k \models complete_{tr}(p_k)$. The sync event $\mathcal{S}$ does not occur in the run r' because it is fresh for $\mathcal{P}(tr)$. Therefore, in r the event $\mathcal{S}$ occurs

precisely when the transition is taken from p_k to p_{true}. This proves $r \in \mathbf{Run}(\mathcal{P}(tr \Updownarrow_{\mathcal{S}} \lceil true \rceil))$. From the fact that $\mathcal{I}$ fits to r' it follows immediately that $\mathcal{I}$ fits to r. We also note that the bad state is reached at time t.

To show the implication from right to left, we assume the existence of a run in $\mathbf{Run}(\mathcal{P}(tr \Updownarrow_{\mathcal{S}} \lceil true \rceil))$ the interpretation $\mathcal{I}$ fits to and that reaches the bad state p_{true} at time t. Let that run be

$$\langle (p_0, \beta_0, \gamma_0, t_0, Y_0), \ldots, (p_k, \beta_k, \gamma_k, t_k, Y_k), (p_{k+1}, \beta_{k+1}, \gamma_{k+1}, t_{k+1}, Y_{k+1}), \ldots \rangle,$$

where p_{k+1} is the first occurrence of p_{true} and $t = \Sigma_{i=0}^{j} t_i$, for $j > k$. The set of runs is prefix closed. Thus, $\langle (p_0, \beta_0, \gamma_0, t_0, Y_0), \ldots, (p_k, \beta_k, \gamma_k, t_k, Y_k) \rangle$ is a run of $\mathcal{P}(tr \Updownarrow_{\mathcal{S}} \lceil true \rceil)$ to which $\mathcal{I}$ fits. Since it uses locations and transitions in $\mathcal{P}(tr)$ only, it is a run of $\mathcal{P}(tr)$. From the transition from p_k to p_{true} it follows that $\gamma_k + t_k, Y_k \models complete_{tr}(p_k)$. Lemma 4.4 implies $\mathcal{I}, [0, t'] \models tr$ with $t' := \Sigma_{i=0}^{k} t_i$. By construction, exactly Y_k contains $\mathcal{S}$. Thus $\mathcal{I}, [t', t'] \models true \Updownarrow_{\mathcal{S}} true$. We conclude that $\mathcal{I}, [0, t] \models tr \Updownarrow_{\mathcal{S}} \lceil true \rceil$.

The proof for $\neg tr \Updownarrow_{\mathcal{S}} \lceil true \rceil$ and $\lceil true \rceil \Updownarrow_{\mathcal{S}_1} tr \Updownarrow_{\mathcal{S}_2} \lceil true \rceil$ are similar. The case $\lceil true \rceil \Updownarrow_{\mathcal{S}_1} \neg tr \Updownarrow_{\mathcal{S}_2} \lceil true \rceil$ contains a subtlety. The formula $\neg tr$ is satisfied by an empty interval. But the proof of Theorem 3.4 ensures that time elapses between both sync events. It is therefore correct that we do not construct a transition with both sync events from the initial phase to the bad state.

For the induction step, let $\bigwedge_{i_l} \mathcal{T}_{i_l}$, $l = 1, 2$, be two conjunctions of formulae in (10). We first prove the implication from left to right. By definition, $\mathcal{I}, [0, t] \models \bigwedge_{i_1} \mathcal{T}_{i_1} \wedge \bigwedge_{i_2} \mathcal{T}_{i_2}$ is equivalent with $\mathcal{I}, [0, t] \models \bigwedge_{i_1} \mathcal{T}_{i_1}$ and $\mathcal{I}, [0, t] \models \bigwedge_{i_2} \mathcal{T}_{i_2}$. By the hypothesis, there are automata $\mathcal{P}(\bigwedge_{i_l} \mathcal{T}_{i_l})$ and test runs

$$r_l := \left\langle (p_{0,l}, \beta_{0,l}, \gamma_{0,l}, t_{0,l}, Y_{0,l}), (p_{1,l}, \beta_{1,l}, \gamma_{1,l}, t_{1,l}, Y_{1,l}), \ldots \right\rangle \in \mathbf{Run}(\mathcal{P}(\textstyle\bigwedge_{i_l} \mathcal{T}_{i_l})).$$

Moreover, $\mathcal{I}$ fits to r_l and there are $k, k' \in \mathbb{N}_{>0}$ with $p_{k,1} = p_{bad,1}$, $p_{k',2} = p_{bad,2}$ and $\Sigma_{j=0}^{k} t_{j,1} = t = \Sigma_{j=0}^{k'} t_{j,2}$. We construct $r \in \mathcal{P}(\bigwedge_{i_1} \mathcal{T}_{i_1}) \parallel \mathcal{P}(\bigwedge_{i_2} \mathcal{T}_{i_2})$ that satisfies the requirements. The sets of runs in PEA are closed under stuttering. We introduce stuttering steps into r_1 and r_2 such that the transitions are taken synchronously. This gives the runs r'_1 and r'_2 to which $\mathcal{I}$ fits. We define

$$r := \left\langle \left((p_{0,1}, p_{0,2}), \beta'_{0,1} \cup \beta'_{0,2}, \gamma'_{0,1} \cup \gamma'_{0,2}, t'_{0,1}, Y'_{0,1} \cup Y'_{0,2} \right), \ldots \right\rangle.$$

The union of γ' functions is well defined due to the disjointness of the clocks. The union of β' functions is well defined because $\mathcal{I}$ fits to both r'_l. For the same reason, we have $\mathcal{E} \in Y'_{i,1}$ iff $\mathcal{E} \in Y'_{i,2}$. This shows $\mathcal{I}$ fits to r. It is straightforward to prove that r is a run of $\mathcal{P}(\bigwedge_{i_1} \mathcal{T}_{i_1}) \parallel \mathcal{P}(\bigwedge_{i_2} \mathcal{T}_{i_2})$ reaching $(p_{bad,1}, p_{bad,2})$ at time t.

To show the reverse direction, let there be a run $r \in \mathbf{Run}(\mathcal{P}(\bigwedge_{i_1} \mathcal{T}_{i_1}) \parallel \mathcal{P}(\bigwedge_{i_2} \mathcal{T}_{i_2}))$ such that $\mathcal{I}$ fits to r and r reaches the bad state $(p_{bad,1}, p_{bad,2})$ at moment t. Restricting the run to the variables and events of the component automata yields runs $r_l \in \mathbf{Run}(\mathcal{P}(\bigwedge_{i_l} \mathcal{T}_{i_l}))$. Moreover, $\mathcal{I}$ fits to r_l and the bad states $p_{bad,l}$ are reached at t. The hypothesis gives $\mathcal{I}, [0, t] \models \bigwedge_{i_1} \mathcal{T}_{i_1}$ and $\mathcal{I}, [0, t] \models \bigwedge_{i_2} \mathcal{T}_{i_2}$ which is $\mathcal{I}, [0, t] \models \bigwedge_{i_1} \mathcal{T}_{i_1} \wedge \bigwedge_{i_2} \mathcal{T}_{i_2}$. This concludes the proof. □

With Lemma 4.6 we can reduce the problem whether a PEA satisfies a negated test formula to a reachability question. The correctness of our model checking procedure is stated in the following theorem.

Theorem 4.7 (**Correctness of model checking**) Let TF be a test formula with the normal form $\exists \mathcal{S}_{ijk} : \bigvee_i \bigwedge_j \mathcal{T}_{ij}$. The model checking problem whether the negated test formula is satisfied by a PEA $\mathcal{A}$ can be characterised by reachability as follows:

$$\neg (\mathcal{A} \models_0 \neg TF) \Leftrightarrow \exists i : \exists r \in \mathbf{Run}(\mathcal{A} \parallel \mathcal{P}(\textstyle\bigwedge_j \mathcal{T}_{ij})) : r \text{ reaches a location } (p, p_{bad}),$$

where p is a location of $\mathcal{A}$ and p_{bad} is the bad state of $\mathcal{P}(\bigwedge_j \mathcal{T}_{ij})$.

Proof Consider the model checking problem $\mathcal{A} \models_0 \neg TF$ for a PEA $\mathcal{A}$ and a test formula TF. To begin with, we apply the definition of satisfaction for a formula and an automaton, Definition 4.1, and the definition of satisfaction from time zero, Sect. 2.1. Pulling the negation inwards yields the following equivalences:

$$\begin{aligned}
&\neg (\mathcal{A} \models_0 \neg TF) \\
\Leftrightarrow &\neg \forall \mathcal{I} : \forall r \in \mathbf{Run}(\mathcal{A}) : (\mathcal{I} \text{ fits to } r \Rightarrow \mathcal{I} \models_0 \neg TF) \\
\Leftrightarrow &\neg \forall \mathcal{I} : \forall r \in \mathbf{Run}(\mathcal{A}) : (\mathcal{I} \text{ fits to } r \Rightarrow (\forall t \in \mathbb{R}_{\geq 0} : \mathcal{I}, [0, t] \models \neg TF)) \\
\Leftrightarrow &\exists \mathcal{I} : \exists r \in \mathbf{Run}(\mathcal{A}) : \exists t \in \mathbb{R}_{\geq 0} : \mathcal{I} \text{ fits to } r \wedge \mathcal{I}, [0, t] \models TF.
\end{aligned}$$

According to Theorem 3.4, the normal form $\exists \mathcal{S}_{ijk} : \bigvee_i \bigwedge_j \mathcal{T}_{ij}$ is satisfiability equivalent with TF. The set of observables $\mathcal{S}_{ijk}$ can be assumed to be disjoint from the variables and events in $\mathcal{A}$. We replace TF by its normal form and apply the definition of quantification over observables. This continues the equivalence:

$$\Leftrightarrow \exists \mathcal{I} : \exists r \in \mathbf{Run}(\mathcal{A}) : \exists t' \in \mathbb{R}_{\geq 0} : \mathcal{I} \text{ fits to } r \wedge \mathcal{I}, [0, t'] \models \exists \mathcal{S}_{ijk} : \bigvee_i \bigwedge_j \mathcal{T}_{ij}$$
$$\Leftrightarrow \exists \mathcal{I} : \exists r \in \mathbf{Run}(\mathcal{A}) : \exists \mathcal{I}' =_{\setminus \mathcal{S}_{ijk}} \mathcal{I} : \exists t' \in \mathbb{R}_{\geq 0} : \mathcal{I} \text{ fits to } r \wedge \mathcal{I}', [0, t'] \models \bigvee_i \bigwedge_j \mathcal{T}_{ij}.$$

Since $\mathcal{S}_{ijk} \cap \mathcal{A} = \varnothing$, $\mathcal{I}'$ belongs to a run of $\mathcal{A}$ if and only if $\mathcal{I}$ does. This allows us to unify the quantifications over interpretations:

$$\Leftrightarrow \exists \mathcal{I}' : \exists r \in \mathbf{Run}(\mathcal{A}) : \exists t' \in \mathbb{R}_{\geq 0} : \mathcal{I}' \text{ fits to } r \wedge \mathcal{I}', [0, t'] \models \bigvee_i \bigwedge_j \mathcal{T}_{ij}$$
$$\Leftrightarrow \exists \mathcal{I}' : \exists r \in \mathbf{Run}(\mathcal{A}) : \exists t' \in \mathbb{R}_{\geq 0} : \exists i : \mathcal{I}' \text{ fits to } r \wedge \mathcal{I}', [0, t'] \models \bigwedge_j \mathcal{T}_{ij}.$$

Applying Lemma 4.6 allows us to replace the satisfaction problem for a disjunct by a reachability question for the corresponding test automaton:

$$\Leftrightarrow \exists \mathcal{I}' : \exists r \in \mathbf{Run}(\mathcal{A}) : \exists t' \in \mathbb{R}_{\geq 0} : \exists i : \mathcal{I}' \text{ fits to } r \wedge$$
$$\exists r' \in \mathbf{Run}(\mathcal{P}(\textstyle\bigwedge_j \mathcal{T}_{ij})) : \mathcal{I}' \text{ fits to } r' \wedge r' \text{ reaches the bad state } p_{bad} \text{ at } t'$$
$$\Leftrightarrow \exists \mathcal{I}' : \exists i : \exists r'' \in \mathbf{Run}(\mathcal{A} \parallel \mathcal{P}(\textstyle\bigwedge_j \mathcal{T}_{ij})) : \exists t' \in \mathbb{R}_{\geq 0} : \mathcal{I}' \text{ fits to } r'' \wedge r'' \text{ reaches a location } (p, p_{bad}) \text{ at } t'.$$

The last equivalence is proven in the induction step of Lemma 4.6. In the beginning of this section, we mentioned that every run induces an interpretation that fits to it. We therefore remove the quantification over the interpretation. Analogously, we remove the quantification over the point in time:

$$\Leftrightarrow \exists i : \exists r'' \in \mathbf{Run}(\mathcal{A} \parallel \mathcal{P}(\textstyle\bigwedge_j \mathcal{T}_{ij})) : r'' \text{ reaches a location } (p, p_{bad}).$$

This concludes the proof of the equivalence. □

Model checking can be done separately for all disjuncts and terminates as soon as the bad state is reachable in one of the disjuncts. The parallel composition $\mathcal{A} \parallel \mathcal{P}(\bigwedge_j \mathcal{T}_{ij})$ only needs to be computed for the evaluated disjuncts. A disjunct may consist of several conjuncted formulae. For model checking, a subset of these formulae may be chosen. Only if the bad state is reachable in the TA for the subset, further formulae have to be added. Model checking is repeated for the new set of formulae gained by this iterative procedure. If the bad state is not reachable for the subset, we know that it is not reachable for the whole disjunct. This incremental approximation can significantly reduce the TA size.

We briefly discuss the decidability of our model checking approach stated in Theorem 4.7. Consider a PEA $\mathcal{A}$ and a test formula TF. To decide $\mathcal{A} \models_0 \neg TF$, we need to compute the normal form $\exists \mathcal{S}_{ijk} : \bigvee_i \bigwedge_j \mathcal{T}_{ij}$ of TF, the test automata $\mathcal{P}(\bigwedge_j \mathcal{T}_{ij})$ of the disjuncts, and the composition $\mathcal{A} \parallel \mathcal{P}(\bigwedge_j \mathcal{T}_{ij})$. All procedures terminate. We then need to decide, whether the composition reaches the bad state in the test automaton.

Thus, our model checking problem is decidable if and only if the reachability problem in $\mathcal{A} \parallel \mathcal{P}(\bigwedge_j \mathcal{T}_{ij})$ is. As discussed in Sect. 2.2, the reachability problem is undecidable for general PEA. But, if we have finite data domains in $\mathcal{A}$ and effective predicates (1) on the transitions of $\mathcal{A}$ and (2) in TF then the reachability problem is decidable, even in the presence of parameters. We translate $\mathcal{A} \parallel \mathcal{P}(\bigwedge_j \mathcal{T}_{ij})$ into a timed automaton as explained in Sect. 2.2 and check reachability with the procedure in [AD94] or using more efficient implementations [UPP05]. Unfortunately, in our case study (cf. Sect. 5) we do not have finite data domains. Moreover, unfolding the states to all evaluations of state variables results in an exponential blow up of the state space. We discuss how we handle these problems in the following section.

4.4. Tool support

We implemented the graphical design tool Moby/PEA [HMF06] to develop PEA models. For complex scenarios, we use the high-level language CSP-OZ-DC [Hoe06] to create system models (cf. Sect. 5), which we then compile to PEA. Afterwards we check whether the PEA satisfies a given test formula. The verification process is outlined in Fig. 3.

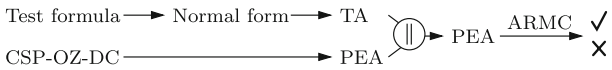

Fig. 3. Flow of the verification process

For transforming the test formula into a set of test automata, applying the algorithms of Sect. 3 and 4, we implemented a compiler [HMF06] that automatically computes the normal form and the corresponding test automata. In the next step, we compute the parallel composition of the test automata and the PEA of the model. Our tool generates outputs for the model checking tools Uppaal [UPP05] and ARMC [Ryb07, PR07]. Finally, we apply a model checker on the product automaton.

Our case study requires that the applied model checker is able to handle infinite numeric data domains and parameters. Unfortunately, for this class of inputs the model checker Uppaal is of limited use, because it does not meet this requirement. We use the model checker ARMC, which can analyse reachability properties of infinite state systems. Compared to other state-of-the-art model checkers, e.g., Blast [HJMM04] and SLAM [BMMR01], ARMC particularly suites the case study due to its specialised abstraction discovery algorithms for unbounded numeric data domains [RSS07].

We briefly sketch the abstract reachability algorithm implemented in ARMC, and refer to [PR07] for a detailed account. The input to ARMC is a control-flow graph whose edges are annotated by transition relations expressed as linear arithmetic constraints together with the specification of the initial and error location. ARMC automatically constructs a safe abstraction (i.e., an over-approximation) of the set of reachable states of the input PEA by means of predicate abstraction [GS97]. The abstraction is incrementally refined until the error location is not contained in the over-approximation or a counterexample connecting the initial and error locations is found.

The building blocks for computing the abstraction are predicates over PEA variables, events, and clocks. We use abstraction induced by a fixed finite set of predicates to ensure convergence of the reachability computation. Given such a set, the abstraction of a set of PEA states is the conjunction of all predicates that subsume the given set of states. The abstraction of the set of reachable states is computed iteratively, by interleaving symbolic unrolling of the PEA's transition relation, which computes a set of reachable states with its abstraction using predicates.

In this process, the right choice of predicates is crucial, since the resulting over-approximation needs to be sufficiently precise in order to prove the non-reachability of the error locations. The process of choosing the right abstraction is guided by spurious counterexamples that are found if the abstraction is not precise enough to verify the property [CGJ+00]. We apply the recent methodology for the extraction of new predicates from spurious counterexamples that is based on interpolation [McM03, HJMM04]. We designed and implemented an efficient algorithm for the computation of interpolants for arbitrary linear inequalities over rationals/reals [RSS07]. The existing implementations only support restricted classes of inequalities over at most two variables and do not handle strict inequalities, which are particularly needed for the verification of real-time systems. Our algorithm applies constraint solving techniques to compute interpolants. It is implemented using Linear Programming solvers in a black-box fashion, providing a basis for an efficient implementation.

5. Case study

In this section, we take up the case study from Sect. 1.1 to show the practicability of our approach in a realistic scenario. We pass from a declarative high-level description of our case study to a PEA representation for which we verify safety properties.

5.1. Case study scenario: emergency procedure in the ETCS

As introduced in Sect. 1.1, our goal is to model and analyse the treatment of emergency messages in the European Train Control System that is informally specified in [ERT02, ECS99]. Therefore, we consider two consecutive trains on a (to simplify matters) infinite track segment (cf. Fig. 1), defined by train positions: $\mathit{Position} := \mathbb{R}$. The trains measure their positions periodically and adjust their speed between a lower bound *TargetSpeed* and an upper bound *MaxSpeed*, such that the follower is always able to brake safely applying the service brakes. To this end, the *service brake intervention limit* (SBI) is calculated periodically: the last position the train has to apply the service brakes such that it remains possible to stop safely. If the first train detects an emergency it sends an

alert to the RBC which forwards this message within 5 time units. If necessary the follower immediately brakes. Otherwise, the emergency is indicated to the driver (within 1.5 time units) who has to acknowledge the warning (i.e., the driver takes responsibility for driving and the system is assumed to be safe) or to apply the emergency brakes. If the driver does not react the emergency brakes are applied automatically by the ETCS on-board unit within 5 time units. We use our verification technique to verify collision freedom of the trains.

5.2. Case study specification

To facilitate the verification of complex and heterogenous systems like the ETCS case study, one has to consider different aspects. That is, we have to cope with several communicating components (the trains and the RBC) with a complex control structure (driver reaction to incoming emergency warnings) and thus, with a large discrete state space. In addition, the state space becomes infinite due to infinite data types and a continuous time model. Moreover, the case study requires the transfer of messages including data values from infinite domains (real-valued train positions). Finally, since the case study is parametric, i.e., no explicit values are given for some constants, the system has to be verified in the presence of parameters with an infinite range.

To cope with the discussed aspects, we use the high-level language CSP-OZ-DC [HO02] as specification formalism. It integrates the well-investigated languages CSP [Hoa85] for specifying communication aspects and control flow, Object-Z [Smi00] for specifying data changes over infinite data types and parameters, and DC [ZH04] for the real-time dimension. CSP-OZ-DC is a declarative and object-oriented language with an operational semantics [Hoe06] in terms of PEA. Hence, we can apply the verification methodology introduced in this paper.

Our case study incorporates five components (Fig. 4) that can be modelled in CSP-OZ-DC with the application domain classes: *Train*, *RBC*, *Track*, *Driver*, and a communication layer *ComNetwork*, which is necessary to model the transfer times of messages between trains and the RBC. To simplify the verification we differentiate between the first and the second train on the track and consider two classes *FrontTrain* and *RearTrain*. We explain the class *RearTrain* in more detail here—the entire specification can be found in Appendix A. Figure 4 also shows connections between the classes. Messages are transferred between trains and the RBC via channels *send* and *receive*. A train's position is updated periodically via *updatePosition*. Additionally, there are channels to indicate an emergency situation to the driver and channels for acknowledgements and braking instructions from the driver.

The first part of a CSP-OZ-DC class is the interface declaring channels that are visible outside the class and local channels for class-internal use. The interface of *RearTrain* is defined by

RearTrain(*ID* : *TrainID*; *StartPosition* : *Position*)

`chan` send : [m! : Message, id : {ID}]	`local_chan` computeSBI : [loa?,sbi! : Position]
`chan` receive : [m? : Message, id : {ID}]	`local_chan` applySB, releaseSB
`chan` updatePosition : [id : {ID},pos! : Position]	`local_chan` getPosition : [pos! : Position]
`chan` indication	`local_chan` getSBI : [sbi! : Position]
`chan` getLOA : [id : {ID}, loa? : Position]	`local_chan` selectSpeed
`chan` driverAck, driverEB : [id : {ID}]	`local_chan` applyEB

The first line contains, besides the class name *RearTrain*, two formal parameters for the ID of the train and the initial position. Then, the class *RearTrain* defines channels that can be used for inter-class (e.g., *send* and *receive*) and internal communication. For example, there are local channels *applySB* and *releaseSB* for applying and releasing the train's service brakes. Channels may have parameters. The parameter *m* of *send* is of type *Message* ::= *Alert* | *Warning* | *Ack*. The message *Alert* represents an emergency alert that is sent to the RBC. The RBC notifies the second train using *Warning* and waits for an acknowledgement *Ack*. Note that CSP-OZ-DC admits the transfer of infinite data types, e.g., real-valued train positions as parameter of *updatePosition*.

The events induced by these channel declarations are structured by CSP processes [Hoa85] in the CSP part of the class.

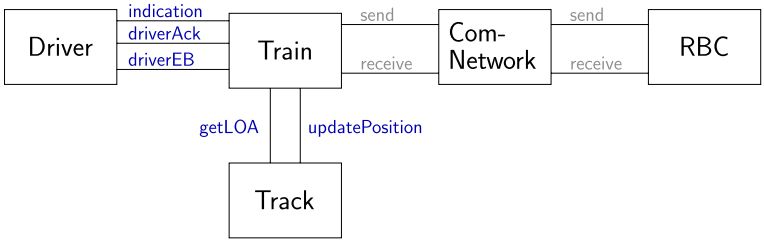

Fig. 4. Objects of the case study scenario

$$
\begin{aligned}
\texttt{main} &\stackrel{c}{=} \mathit{Running} \;|||\; \mathit{HandleEM} \\
\mathit{Running} &\stackrel{c}{=} \mathit{updatePosition.ID?pos} \rightarrow \mathit{getLOA.ID?loa} \rightarrow \mathit{computeSBI!loa?sbi} \rightarrow \\
&\quad \textbf{if } \mathit{sbi} \leq \mathit{pos} \\
&\quad \textbf{then } \mathit{applySB} \rightarrow \mathit{selectSpeed} \rightarrow \mathit{Running} \\
&\quad \textbf{else } \mathit{releaseSB} \rightarrow \mathit{selectSpeed} \rightarrow \mathit{Running} \\
\mathit{HandleEM} &\stackrel{c}{=} \mathit{receive.Warning.ID} \rightarrow \mathit{send.Ack.ID} \rightarrow \mathit{getPosition?pos} \rightarrow \mathit{getSBI?sbi} \rightarrow \\
&\quad \textbf{if } \mathit{pos} < \mathit{sbi} - \mathit{OffsetDistance} \\
&\quad \textbf{then } \mathit{indication.ID} \rightarrow (\mathit{driverAck.ID} \rightarrow \mathit{DriverResponsible} \\
&\qquad\qquad\qquad\qquad \Box\ \mathit{EmergencyStop} \\
&\qquad\qquad\qquad\qquad \Box\ \mathit{driverEB.ID} \rightarrow \mathit{EmergencyStop}) \\
&\quad \textbf{else } \mathit{EmergencyStop}
\end{aligned}
$$

The execution of the CSP part starts with the `main` process. The `main` process of *RearTrain* consists of an interleaving of the two subprocesses *Running* and *HandleEM*. The former represents the normal behaviour of a train, the latter the emergency procedure. The subprocess *Running* states that every *updatePosition* is followed by a *getLOA* event that gets the new limit of authority (LOA), a movement authorisation up to a given position. The next permitted event is *computeSBI* that takes the new LOA and computes the SBI position *sbi* (cf. Sect. 5.1) that is needed in the **if** statement. If the current position of the train is already beyond *sbi* the train has to brake down to *TargetSpeed*—the speed the train is allowed to run in the vicinity of the LOA. Otherwise, the train releases the brakes and selects a new speed value up to *MaxSpeed*.

The periodicity of *updatePosition* and *computeSBI* that is required by the informal case study description is realised by imposing time constraints to *updatePosition* in the DC part of the specification. Since there is always either an *applyEB* or a *releaseSB* event between two *updatePosition* events, the periodicity of *updatePosition* likewise guarantees an upper time bound on the occurrence of the brake event *applySB*.

When the RBC sends a *warning*, the train receives this message on the channel *receive* with the process *HandleEM*. The process uses the information about the train position and the SBI to decide if it is necessary to apply the emergency brakes immediately. The constant *OffsetDistance* represents a reaction distance originating from the periodic position updates.

In CSP-OZ-DC, data aspects are specified with Object-Z (OZ) [Smi00]. The OZ part consists of schemas describing the state space and data changes of a class.

$$
\begin{array}{|l}
\hline
\mathit{currentPosition}, \mathit{currentSpeed} : \mathit{Position} \\
\mathit{sbi}, \mathit{standstillEB} : \mathit{Position} \\
\mathit{brakingMode} : \mathit{BrakingMode} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\texttt{Init} \\
\hline
\mathit{brakingMode} = \mathit{None} \\
\mathit{TargetSpeed} < \mathit{currentSpeed} \leq \mathit{MaxSpeed} \\
\mathit{currentPosition} = \mathit{StartPosition} \\
\hline
\end{array}
$$

The OZ part of *RearTrain* includes a *state schema* defining the attributes of the class, e.g., the current position of the train (*currentPosition* of the real-valued type *Position*), the current braking mode (of type $\mathit{BrakingMode} ::= \mathit{None} \mid \mathit{ServiceBrake} \mid \mathit{EmergencyBrake}$), or the position (*standstillEB*) that will be reached by the train if applying the emergency brakes. To simplify the treatment of units, we measure speed in terms of $\frac{\mathit{Position}}{\mathit{updateBound}}$, where *updateBound* is the time between two position updates. The `Init` schema constrains the initial state of a CSP-OZ-DC class. Here, the train is initially not braking, the speed is between *TargetSpeed* and *MaxSpeed*, and the current position is set to *StartPosition*. We stress that the constants *TargetSpeed* and *MaxSpeed*, as well as

Length, *TargetSpeedDistance*, *OffsetDistance*, and *StopDistance* are parameters, i.e., we do not need to interpret all constants. Instead, it suffices to specify conditions that restrict the values adequately, e.g., the parameter *Length* is constrained by $Length > 0$.

Operation schemas define state changes that are performed at the same time when—in agreement with the CSP part of the class—the appropriate events occur. For instance, we associate with the event *updatePosition* (from the CSP part) the operation schema com_*updatePosition*.

com_*selectSpeed*

$\Delta(currentSpeed)$

---

if $brakingMode = None$ **then**
$TargetSpeed < currentSpeed' \leq MaxSpeed$
if $brakingMode = ServiceBrake$ **then**
$currentSpeed' = TargetSpeed$
if $brakingMode = EmergencyBrake$ **then**
$(currentSpeed' < TargetSpeed \ \wedge$
$currentPosition + currentSpeed' \leq standstillEB)$
$\vee \ (currentSpeed' = 0 \ \wedge$
$currentPosition + TargetSpeed > standstillEB)$

com_*updatePosition*

$\Delta(currentPosition)$
$pos! : Position$

$currentPosition' = currentPosition$
$+currentSpeed$
$pos! = currentPosition'$

The Δ expression indicates the state variables changed by an operation schema: in com_*updatePosition*, the new *currentPosition* is constrained by $currentPosition' = currentPosition + currentSpeed$, where the primed variable refers the new value of the variable (analogous to the definition of PEA in Sect. 2.2). Output variables of operation schemas like *pos*! are indicated by !, input variables by ?. The idea for constraining com_*selectSpeed* is as follows. If the train is not braking it selects a new speed value between *TargetSpeed* and *MaxSpeed*. If the service brakes are applied the new speed is set to *TargetSpeed*, and if the emergency brakes are applied the speed is set to a value below *TargetSpeed*. In the latter case, we take the *standstillEB* position into account, which is computed in com_*applyEB*. Furthermore, there are operation schemas (cf. Appendix A) for applying and releasing the service brakes (com_*applySB* and com_*releaseSB*), for selecting a new speed value (com_*selectSpeed*), and for applying the emergency brakes (com_*applyEB*).

Finally, in our case study scenario we have real-time constraints on the trains: positions have to be updated periodically and in case of an emergency the train controller has to react in time. Real-time constraints are described using counterexample-trace formulae, i.e., negated trace formulae according to Sect. 3:

$\neg(true \,;\, \updownarrow updatePosition \,;\, updateBound > \ell \,;\, \updownarrow updatePosition)$
$\neg(true \,;\, (\boxminus updatePosition.ID \wedge updateBound < \ell))$
$\neg(true \,;\, \updownarrow receive.Warning.ID \,;\, (\boxminus applyEB \wedge \boxminus indication \wedge 0.5 < \ell) \,;\, (\boxminus indication \wedge 1 < \ell))$
$\neg(true \,;\, \updownarrow indication \,;\, (\boxminus driverAck \wedge \boxminus applyEB \wedge 5 < \ell))$

The first two DC formulae define that *updatePosition* is a periodic event occurring every *updateBound* time units. The third DC formula demands that after receiving an emergency warning the emergency brakes are applied within 0.5 time units or the emergency is indicated to the driver within 1.5 time units. The driver has to acknowledge the indication or the emergency brakes are automatically applied within 5 time units, which is required by the last formula.

Remaining components. The components *LeadingTrain*, *RBC*, *Driver*, *ComNetwork*, and *Track* are simpler than the *RearTrain* class and we only summarise the ideas behind their specifications. The classes can be found in the appendix.

LeadingTrain. This is a simplified version of the *RearTrain* class. The CSP part only considers the running behaviour as well as the detection of emergencies. If an emergency is detected we assume the worst case, i.e., the train stops immediately.

ComNetwork. The communication layer receives and forwards messages between trains and the RBC.

RBC. The RBC receives emergency alerts from the first train and forwards a warning to the follower. In addition, a timing constraint demands that the warning will not be delayed.

Driver. After an emergency indication the driver chooses to acknowledge the indication or to apply the emergency brakes.

Track. The track represents the environment in our case study scenario. It periodically receives position updates from the trains and calculates a new LOA if requested (*getLOA*) by the *RearTrain*. Additionally, the *Track* class guarantees that the train positions are initially safe, i.e., the *StartPosition* and *StartSBI* variables are constrained in the `Init` schema such that no collision occurs.

Safety property. The desired safety property in our case study is that the trains will never collide if the driver has not taken the responsibility for driving. In the latter case, the system is assumed to be safe. For a setting with two trains, this can be expressed by the test formula

$$\neg((\ell > 0 \wedge \boxminus \mathit{driverAck})\,;\lceil \mathit{RearTrain.currentPosition} > \mathit{LeadingTrain.currentPosition} - \mathit{Length}\rceil). \tag{14}$$

Here, *driverAck*, *RearTrain.currentPosition*, and *LeadingTrain.currentPosition* are observables, i.e., their values change during system evolution. *Length* is the parameter from the case study, the value remains fixed.

Translation into PEA. The translation of CSP-OZ-DC specifications into PEA is compositional: every part of the specification is translated separately; the semantics of the entire specification is the parallel composition of the automata for every part: $\mathcal{A}(\mathit{CSP\text{-}OZ\text{-}DC}) = \mathcal{A}(\mathit{CSP}) \parallel \mathcal{A}(\mathit{OZ}) \parallel \mathcal{A}(\mathit{DC})$.

The PEA for the CSP part directly represents the structured operational semantics of CSP [Ros98] (with additional stuttering transitions at every location). The OZ part is translated into a PEA with two locations: one for setting the initial values of the state variables and one for calling operations while the system runs. The latter has the constraint of the state schema as invariant. It has one transition for every communication schema of the OZ part. The guard of this transition is given by the constraint defined in the corresponding communication schema. Finally, each formula of the DC part is translated into a deterministic PEA according to the power set construction in Sect. 4.1. From this automaton the locations p with $\mathit{complete}_{tr}(p) \neq \mathit{false}$ are removed to ensure that the system behaviour satisfies the negated trace formula.

5.3. Results

As explained in the previous section, our case study incorporates different dimensions of complex control flow, communications, real-time aspects, and parameters. Due to this complexity, the translation of the case study model into PEA results in 18 parallel components, 10 state variables, 15 parameters of channels, and 9 clocks. Note that the state space is infinite because of infinite ranges of variables and real-time constraints. Hence, this model is too large to verify the global safety property (14) in a single step. Therefore, we decompose it manually into smaller parts and verify local properties for the parallel components. These local properties always depend on less components of the case study model and, by this, they can be verified faster. That the local properties hold for the entire system is guaranteed by Theorem 4.2. We identify the following local properties:

1. Without emergency detection there is always a (safe) minimal distance between the trains.
2. After an emergency detection the follower starts braking within 8 time units.
3. Starting with the minimal distance of 1., the train still has a distance of *StopDistance* to the next danger point after 8 time units.
4. If the train applies the emergency brakes with a distance of *StopDistance* to the next danger point, it always stops before reaching the danger point.

Currently, one has to establish manually that the local properties imply the global property, e.g., by using traditional proof rules for the DC. It is ongoing work to support this step by automated methods. In the following, we illustrate how these local properties can be defined and verified using our technique.

1. Minimal distance without emergency. This characteristic can be specified by

$$\neg((\ell > 0 \wedge \boxminus \mathit{tf\_receiveFromTrain\_alert})\mathbf{;}\\ \lceil \mathit{Track.position}_1 > \mathit{Track.position}_0 - \mathit{Length} - \mathit{TargetSpeedDistance} - \mathit{StopDistance}\rceil). \tag{15}$$

Table 1. Experimental results (Athlon XP 2200+, 512 MB RAM)

Task	1	2	3	4	5	6	7	8
Property (15)	10	482	27	7	6	5	4 s	14 s
Property (16)	34	193	27	3	47	3	3 s	4 s
Property (17)	4.9 T	99 T	47	14	3.4 T	14	5 m	216 m
Property (18)	53	3.3 T	34	2	13	2	15 s	22 s
Property (19)	34	1.5 T	33	20	66	18	8 s	2 m
Property (20)	52	6.8 T	33	33	108	23	29s	19 m

T thousand units, *m* minutes, *s* seconds
1 Program locations, *2* transitions, *3* variables, *4* predicates generated by ARMC, *5* abstract states, *6* refinements loops performed by ARMC, *7* runtime for generating test automata and parallel product, *8* runtime for model checking

The *tf_receiveFromTrain_alert* is an auxiliary event representing *receiveFromTrain*'s occurrence with the train ID 1 and the message *Alert* as parameters. Like collision freedom this property depends on almost the entire model. For this reason, we first have to prove further sub-properties, e.g., that the second train updates its position correctly depending on the current SBI, or that without emergency the first train's speed is always greater than *TargetSpeed*. The latter is specified by the negated test formula

$$\neg((\ell > 0 \wedge \boxminus \mathit{detectEmergency}) \wedge (\lceil true \rceil \,;\, \lceil \mathit{LeadingTrain.currentSpeed} < \mathit{TargetSpeed} \rceil \,;\, \lceil true \rceil)). \tag{16}$$

According to Theorem 4.2, it suffices to take those PEA into account that influence property (16), i.e., in this case the PEA for the CSP and the OZ part of *LeadingTrain*. Using our verification method, this property can be established efficiently (cf. Property (16) in Table 1). Finally, we can show that property (15) holds under the assumption that those sub-properties are valid. The performance results for this verification task are listed as Property (15) in Table 1.

2. Braking initialised within 8 time units. The second property we want to verify is that the timing behaviour of our case study model is correct, i.e.,

$$\neg(\lceil true \rceil \,;\, \updownarrow \mathit{tf\_receiveFromTrain\_alert} \,;\, (\boxminus \mathit{driverAck} \wedge \boxminus \mathit{applyEB} \wedge 8 < \ell)). \tag{17}$$

This formula states that after an *tf_receiveFromTrain_alert* event the driver has to acknowledge the indication of the emergency or the emergency brakes have to be applied. To verify the formula we need to consider the PEA influencing the timing behaviour (again with Theorem 4.2 we get the correctness for the entire specification).

3. Minimal distance after 8 time units. We also have to show that the train moves in 8 time units a distance that is less than *TargetSpeedDistance*:

$$\neg(\lceil true \rceil \,;\, \lceil \mathit{Track.position}_1 \leq x \rceil \,;\, 0 < \ell < 8 \,;\, \lceil \mathit{Track.position}_1 \geq x + \mathit{TargetSpeedDistance} \rceil). \tag{18}$$

4. Standstill before danger point. Finally, we have to show that if the train applies its emergency brakes with a distance of *StopDistance* to the danger point then it will stop before reaching the preceding train. In a first step, we show that the train will not exceed its *standstillEB* position:

$$\neg(\lceil true \rceil \,;\, \updownarrow \mathit{applyEB} \,;\, \lceil true \rceil \,;\, \lceil \mathit{RearTrain.currentPosition} > \mathit{RearTrain.standstillEB} \rceil). \tag{19}$$

In a second step, we have to show that the *standstillEB* position is safe, i.e., it is never beyond the current position of the front train when the rear train brakes with a distance to the preceding train greater than *StopDistance*:

$$\begin{aligned}\neg(\lceil true \rceil \,;\, \lceil \mathit{RearTrain.currentPosition} < \mathit{LeadingTrain.currentPosition} - \mathit{Length} - \mathit{StopDistance} \rceil;\\ \updownarrow \mathit{applyEB} \,;\, \lceil \mathit{RearTrain.standstillEB} \geq \mathit{LeadingTrain.currentPosition} - \mathit{Length} \rceil).\end{aligned} \tag{20}$$

Table 1 shows our experimental results for the verification tasks. Columns (1)–(3) contain information about the size of the model part for the corresponding verification task, columns (4)–(6) contain data about the run of the model checker. The last two columns contain the run times for the generation of test automata and parallel product, and for the model checking. Particularly, the table illustrates that we can handle large-scale models with up to 99,000 program transitions and up to 47 variables (with infinite data types) in an order of 216 minutes. Hence, these results demonstrate that the model checking algorithm presented in this paper can deal with complex problems in large applications of realistic size.

6. Related and future work

Our class of test formulae is a proper generalisation of previously known classes. It is based on the class of counterexample-trace formulae [Hoe06] that correspond to negated traces. Counterexample-traces cover the class of DC implementables [Rav94, Hoe06]. Non-negated traces with phases of exact length, i.e., $\ell = k$ bound, are covered by *Testform*. With this observation our class forms a proper superset of $\{\lceil\varphi\rceil, \ell < k, \ell = k, \ell > k\}$-formulae that have exactly one outermost negation [Frä04]. We conjecture that the classes of constraint diagrams used for model checking timed automata in [DL02] form proper subsets of *Testform*. We have not yet compared the expressiveness of our class with the results in [ABBL03].

In [FH07] a positive subclass of DC containing the integral operator, denoted by DC_{ub}, is translated into Linearly Multi-Priced Timed Automata. The translation is fully compositional, i.e., for every DC operator there is a corresponding operator on the automata. The construction yields a decision algorithm for model checking timed automata against negated DC_{ub} formulae. The classes DC_{ub} and *Testform* are incomparable. We deal with negated traces but do not handle accumulated durations, while DC_{ub} forbids the use of negation.

For positive Duration Interval Logic formulae (DIL$^+$ formulae) a translation into Integration Automata (IA) is given in [BLR95]. DIL$^+$ formulae are covered by *Testform*, because they correspond to traces that contain phases of exact length. To give IA semantics to negated formulae, the authors of [BLR95] show that the negation of a strongly overlap free DIL$^+$ formula has a congruent DIL$^+$ formula. Since our translation for negated traces does not require overlap freeness, it covers a strictly larger class of negated formulae.

Pandya proves the decidability of Interval Duration Logic with located constraints (LIDL–) by translation into event recording timed automata [Pan02]. Located constraints require disjoint phases, a condition our construction does not impose. In contrast, LIDL– is closed under negation even for phases with exact length.

The idea of sync events is closely related to the theory of nominals. In a DC extended with nominals [Han06], intervals are identified uniquely by names. Similarly, sync events identify chop points. In [KP05] phases in the QDDC are equipped with fresh observables to identify chop points. This yields decomposition results similar to ours. The benefit of our work is the integration of sync events with the operators of DC.

Our case study complements the work of [FJSS07] as it incorporates a more complex control structure. The case study in [FJSS07] considers only a single class but an arbitrary number of trains. Related work on ETCS case studies like [ZH05, HJU05] focuses on the stochastic examination of the communication reliability and models components like the train and the RBC in an abstract way without considering data aspects. A hybrid view on speed control in the ETCS is presented in [Pla07].

We currently develop an algorithm for the verification of DC liveness properties. We follow the automata-theoretic approach to temporal verification [Var91], which requires checking (fair) termination of the system composed with the test automaton. Recent advances in automated liveness checking using abstraction refinement [CPR05, PR05] can provide the necessary model checking support.

Our test formulae admit free function symbols in state expressions, but they are not supported by the underlying reachability checker. We currently extend the abstraction refinement procedure of ARMC to support the combined theory of linear arithmetic and free function symbols, following the constraint-based algorithm [RSS07]. This will allow one to verify more realistic models, where the RBC keeps a list of trains that are currently on its track segment.

In addition, enhancing our decomposition techniques for properties is ongoing work. They allow for compositional verification of inherently parallel systems like the ETCS. A prerequisite for compositional verification is the identification of those system parts necessary to prove the property. Slicing techniques [Brü07] may lead to solutions to this problem.

Acknowledgements

We thank Ernst-Rüdiger Olderog for fruitful discussions. Andrey Rybalchenko is supported in part by Microsoft Research through the European Fellowship Programme.

Appendix A: CSP-OZ-DC model of the train case study

RearTrain(*ID* : *TrainID*; *StartPosition*, *StartSBI* : *Position*)

```
chan send : [m! : Message, id : {ID}]
chan receive : [m? : Message, id : {ID}]
chan updatePosition : [id : {ID},pos! : Position]
chan indication
chan getLOA : [id : {ID}, loa? : Position]
chan driverAck, driverEB : [id : {ID}]
local_chan computeSBI : [loa?,sbi! : Position]
local_chan applySB, releaseSB
local_chan getPosition : [pos! : Position]
local_chan getSBI : [sbi! : Position]
local_chan selectSpeed
local_chan applyEB
```

$$
\begin{array}{rcl}
\texttt{main} & \stackrel{c}{=} & \mathit{Running} \;|||\; \mathit{HandleEM} \\
\mathit{Running} & \stackrel{c}{=} & \mathit{updatePosition.ID?pos} \rightarrow \mathit{getLOA.ID?loa} \rightarrow \mathit{computeSBI!loa?sbi} \rightarrow \\
 & & \quad \textbf{if } \mathit{sbi} \leq \mathit{pos} \\
 & & \quad \textbf{then } \mathit{applySB} \rightarrow \mathit{selectSpeed} \rightarrow \mathit{Running} \\
 & & \quad \textbf{else } \mathit{releaseSB} \rightarrow \mathit{selectSpeed} \rightarrow \mathit{Running} \\
\mathit{HandleEM} & \stackrel{c}{=} & \mathit{receive.Warning.ID} \rightarrow \mathit{send.Ack.ID} \rightarrow \mathit{getPosition?pos} \rightarrow \mathit{getSBI?sbi} \rightarrow \\
 & & \quad \textbf{if } \mathit{pos} < \mathit{sbi} - \mathit{OffsetDistance} \\
 & & \quad \textbf{then } \mathit{indication.ID} \rightarrow (\mathit{driverAck.ID} \rightarrow \mathit{DriverResponsible} \\
 & & \qquad\qquad \Box\ \mathit{EmergencyStop} \\
 & & \qquad\qquad \Box\ \mathit{driverEB.ID} \rightarrow \mathit{EmergencyStop}) \\
 & & \quad \textbf{else } \mathit{EmergencyStop} \\
\mathit{EmergencyStop} & \stackrel{c}{=} & \mathit{applyEB} \rightarrow \texttt{Skip} \\
\mathit{DriverResponsible} & \stackrel{c}{=} & \texttt{Skip}
\end{array}
$$

$$
\begin{array}{l}
\mathit{sbi} : \mathit{Position} \\
\mathit{standstillEB} : \mathit{Position} \\
\mathit{currentPosition} : \mathit{Position} \\
\mathit{currentSpeed} : \mathit{Speed} \\
\mathit{brakingMode} : \mathit{BrakingMode}
\end{array}
$$

Init

$$
\begin{array}{l}
\mathit{brakingMode} = \mathit{None} \\
\mathit{TargetSpeed} < \mathit{currentSpeed} \leq \mathit{MaxSpeed} \\
\mathit{currentPosition} = \mathit{StartPosition} \\
\mathit{position} = \mathit{StartPosition} \\
\mathit{sbi} = \mathit{StartSBI}
\end{array}
$$

com_*applySB*

$$
\begin{array}{l}
\Delta(\mathit{brakingMode}) \\
\hline
\textbf{if } \mathit{brakingMode} = \mathit{None} \textbf{ then} \\
\quad \mathit{brakingMode}' = \mathit{ServiceBrake} \\
\textbf{else} \\
\quad \mathit{brakingMode}' = \mathit{brakingMode}
\end{array}
$$

com_*releaseSB*

$$
\begin{array}{l}
\Delta(\mathit{brakingMode}) \\
\hline
\textbf{if } \mathit{brakingMode} = \mathit{ServiceBrake} \textbf{ then} \\
\quad \mathit{brakingMode}' = \mathit{None} \\
\textbf{else} \\
\quad \mathit{brakingMode}' = \mathit{brakingMode}
\end{array}
$$

com_*selectSpeed*

$$
\begin{array}{l}
\Delta(\mathit{currentSpeed}) \\
\hline
\textbf{if } \mathit{brakingMode} = \mathit{None} \textbf{ then} \\
\quad \mathit{TargetSpeed} < \mathit{currentSpeed}' \leq \mathit{MaxSpeed} \\
\textbf{if } \mathit{brakingMode} = \mathit{ServiceBrake} \textbf{ then} \\
\quad \mathit{currentSpeed}' = \mathit{TargetSpeed} \\
\textbf{if } \mathit{brakingMode} = \mathit{EmergencyBrake} \textbf{ then} \\
\quad \mathit{currentSpeed}' < \mathit{TargetSpeed} \ \wedge \\
\quad \mathit{currentPosition} + \mathit{currentSpeed}' \leq \mathit{standstillEB} \\
\quad \vee \\
\quad \mathit{currentSpeed}' = 0 \ \wedge \\
\quad \mathit{currentPosition} + \mathit{TargetSpeed} > \mathit{standstillEB}
\end{array}
$$

com_*applyEB*

$$
\begin{array}{l}
\Delta(\mathit{brakingMode}, \mathit{standstillEB}) \\
\hline
\mathit{brakingMode}' = \mathit{EmergencyBrake} \\
\textbf{if } \mathit{brakingMode} = \mathit{EmergencyBrake} \textbf{ then} \\
\quad \mathit{standstillEB}' = \mathit{standstillEB} \\
\textbf{else} \\
\quad \textbf{if } \mathit{currentSpeed} = \mathit{TargetSpeed} \textbf{ then} \\
\quad\quad \mathit{standstillEB}' = \mathit{currentPosition} + \mathit{StopDistance} \\
\quad \textbf{else} \\
\quad\quad \mathit{standstillEB}' = \mathit{currentPosition} \\
\quad\quad\quad + \mathit{TargetSpeedDistance} + \mathit{StopDistance}
\end{array}
$$

com_*updatePosition*

$$
\begin{array}{l}
\Delta(\mathit{currentPosition}) \\
\mathit{pos}! : \mathit{Position} \\
\hline
\mathit{currentPosition}' = \mathit{currentPosition} + \mathit{currentSpeed} \\
\mathit{pos}! = \mathit{currentPosition}'
\end{array}
$$

com_*computeSBI*

$$
\begin{array}{l}
\Delta(\mathit{sbi}) \\
\mathit{loa}?, \mathit{sbi}! : \mathit{Position} \\
\hline
\mathit{sbi}' = \mathit{loa}? - \mathit{TargetSpeedDistance} - \mathit{StopDistance} \\
\quad\quad - \mathit{MaxSpeed} \\
\mathit{sbi}! = \mathit{sbi}'
\end{array}
$$

com_*getPosition*

$$
\begin{array}{l}
\mathit{pos}! : \mathit{Position} \\
\hline
\mathit{pos}! = \mathit{currentPosition}
\end{array}
$$

com_*getSBI*

$$
\begin{array}{l}
\mathit{sbi}! : \mathit{Position} \\
\hline
\mathit{sbi}! = \mathit{sbi}
\end{array}
$$

$$
\begin{array}{l}
\neg(\mathit{true} \,;\, \updownarrow \mathit{updatePosition} \,;\, \mathit{updateBound} > \ell \,;\, \updownarrow \mathit{updatePosition}) \\
\neg(\mathit{true} \,;\, (\boxminus \mathit{updatePosition.ID} \wedge \mathit{updateBound} < \ell)) \\
\neg(\mathit{true} \,;\, \updownarrow \mathit{receive.Warning.ID} \,;\, (\boxminus \mathit{applyEB} \wedge \boxminus \mathit{indication} \wedge 0.5 < \ell) \,;\, (\boxminus \mathit{indication} \wedge 1 < \ell)) \\
\neg(\mathit{true} \,;\, \updownarrow \mathit{indication} \,;\, (\boxminus \mathit{driverAck} \wedge \boxminus \mathit{applyEB} \wedge 5 < \ell))
\end{array}
$$

LeadingTrain(*ID* : *TrainID*;
StartPosition : *Position*)
chan*send* : [*m*! : *Message*, *id* : {*ID*}]
chan*receive* : [*m*? : *Message*, *id* : {*ID*}]
chan*updatePosition* : [*id* : {*ID*}, *pos*! : *Position*]
chan*detectEmergency*
local_chan*selectSpeed*
local_chan*applyEB*
main $\stackrel{c}{=}$ *updatePosition.ID* →
(*selectSpeed* → *Running*
□ *detectEmergency*
→ *send.Alert.ID*
→ *applyEB*
→ *selectSpeed* → main)

currentPosition : *Position*
currentSpeed : *Speed*
brakingMode : *BrakingMode*

Init
brakingMode = *None*
TargetSpeed ≤ *currentSpeed* ≤ *MaxSpeed*
position = *StartPosition*

com_*updatePosition*
Δ(*currentPosition*)
pos! : *Position*
currentPosition′ = *currentPosition*
+*currentSpeed*
pos! = *currentPosition*′

com_*applyEB*
Δ(*brakingMode*)
brakingMode′ = *EmergencyBrake*

com_*selectSpeed*
Δ(*currentSpeed*)
if *brakingMode* = *None* **then**
TargetSpeed ≤ *currentSpeed*′(≤ *MaxSpeed*)
else
currentSpeed′ = 0

¬(*true* ; ↕ *updatePosition* ; *updateBound* > ℓ
; ↕ *updatePosition*)
¬(*true* ; (⊟ *updatePosition.ID*
∧ *updateBound* < ℓ))

RBC
chan*send* : [*m*! : *Message*, *id*! : *TrainID*]
chan*receive* : [*m*? : *Message*, *id*? : *TrainID*]
main $\stackrel{c}{=}$ *Idle*
Idle $\stackrel{c}{=}$ *receive.Alert* → *HandleEM*
HandleEM $\stackrel{c}{=}$ *send.Warning*!*id*
→ *receive.Ack.id* → *Idle*

¬(*true* ; ↕ *receive.Alert* ; (⊟ *send.Warning* ∧ 0.5 < ℓ))

Driver(*ID* : *TrainID*)
chan*indication* : [*id* : {*ID*}]
chan*driverAck*, *driverEB* : [*id* : {*ID*}]
main $\stackrel{c}{=}$ *indication.ID* → *HandleEM*
HandleEM $\stackrel{c}{=}$ main
⊓ *driverAck.ID* → main
⊓ *driverEB.ID* → main

ComNetwork
chan *sendToTrain*, *sendToRBC*
: [*m*? : *Message*, *id*? : *TrainID*]
chan *receiveFromTrain*, *receiveFromRBC*
: [*m*! : *Message*, *id*! : *TrainID*]
main $\stackrel{c}{=}$ *receiveFromTrain*?*m*.*id*
→ *sendToRBC.m.id* → main
□ *receiveFromRBC*?*m*?*id*
→ *sendToTrain.m.id* → main

¬(*true* ; ↕ *receiveFromTrain.id*;
(⊟ *sendToRBC.id* ∧ 0.5 < ℓ))
¬(*true* ; ↕ *receiveFromRBC.id*;
(⊟ *sendToTrain.id* ∧ 0.5 < ℓ))

Track(*StartPosition*$_0$, *StartPosition*$_1$,
StartSBI$_1$: *Position*)
chan*updatePosition* : [*id*? : *TrainID*,
pos? : *Position*]
chan*getLOA* : [*id*? : *TrainID*, *pos*! : *Position*]

position$_0$: *Position*
position$_1$: *Position*

Init
position$_0$ = *StartPosition*$_0$
position$_1$ = *StartPosition*$_1$
StartSBI$_1$ = *StartPosition*$_0$
− *Length* − *MaxSpeed*
− *TargetSpeedDistance* − *StopDistance*
StartPosition$_1$ < *StartSBI*$_1$

com_*updatePosition*
Δ(*position*$_0$, *position*$_1$)
id? : *TrainID*
pos? : *Position*
if *id*? = 0 **then**
position′$_0$ = *pos*? ∧ *position*′$_1$ = *position*$_1$
else
position′$_1$ = *pos*? ∧ *position*′$_0$ = *position*$_0$

com_*getLOA*
loa! : *Position*
loa! = *position*$_0$ − *Length*

Length, *TargetSpeed*, *MaxSpeed*, *OffsetDistance*,
TargetSpeedDistance, *StopDistance* : *Position*

0 < *Length* ∧ 0 < *TargetSpeed* < *MaxSpeed* ∧
0 < *OffsetDistance*
0 < *StopDistance* < *TargetSpeedDistance*

For abbreviated synchronisation alphabets and class instatiations

$$A == updatePosition, getLOA$$
$$B == [send \mapsto receiveFromTrain, receive \mapsto sendToTrain]$$
$$C == [receiveFromRBC \mapsto send, sendToRBC \mapsto receive]$$
$$D == driverAck, driverEB, indication$$
$$LT == LeadingTrain(0, StartPosition_0)$$
$$RT == RearTrain(1, StartPosition_1) \underset{D}{\parallel} Driver(1)$$
$$Track == Track(StartPosition_0, StartPosition_1, StartSBI_1)$$
$$Trains == LT \;|||\; RT$$

the full case study model is defined by the CSP expression

$$Trains \underset{A}{\parallel} Track \underset{B}{\parallel} ComNetwork \underset{C}{\parallel} RBC.$$

Note that for synchronisation with the *ComNetwork* (communication alphabets B and C) the *linked parallel* operator [Ros98] is used. This is, instead of directly synchronising on the channels *send* and *receive* between the trains and the RBC, we map *send* and *receive* to *receiveFromTrain* and *sendToTrain*.

References

[ABBL03] Aceto L, Bouyer P, Burgueño A, Larsen KG (2003) The power of reachability testing for timed automata. Theor Comput Sci 300(1–3):411–475

[AD94] Alur R, Dill DL (1994) A theory of timed automata. Theor Comput Sci 126(2):183–235

[BLR95] Bouajjani A, Lakhnech Y, Robbana R (1995) From duration calculus to linear hybrid automata. In: Wolper P (ed.) CAV, LNCS, vol 939. Springer, Heidelberg, pp 196–210

[BMMR01] Ball T, Majumdar R, Millstein T, Rajamani S (2001) Automatic predicate abstraction of C programs. In: PLDI, volume 36 of ACM SIGPLAN Notices. ACM Press, New York, pp 203–213

[Brü07] Brückner I (2007) Slicing Concurrent Real-Time System Specifications for Verification. In: Davies J, Gibbons J (eds) Integrated Formal Methods, LNCS, vol 4591. Springer, Heidelberg, pp 54–74

[CGJ+00] Clarke EM, Grumberg O, Jha S, Lu Y, Veith H (2000) Counterexample-guided abstraction refinement. In: Emerson EA, Sistla AP (eds) CAV, LNCS, vol 1855. Springer, Heidelberg, pp 154–169

[CPR05] Cook B, Podelski A, Rybalchenko A (2005) Abstraction refinement for termination. In: SAS, LNCS, vol 3672. Springer, Heidelberg, pp 87–101

[DL02] Dierks H, Lettrari M (2002) Constructing test automata from graphical real-time requirements. In: Damm W, Olderog E-R (eds) FTRTFT, LNCS, vol 2469. Springer, Heidelberg, pp 433–453

[ECS99] ECSAG. ERTMS/ETCS Functional requirements specification (1999)

[ERT02] ERTMS User Group, UNISIG. ERTMS/ETCS System requirements specification (2002)

[FH07] Fränzle M, Hansen MR (2007) Deciding an interval logic with accumulated durations. In: TACAS, LNCS, vol 4424. Springer, Heidelberg, pp 201–215

[FJSS07] Faber J, Jacobs S, Sofronie-Stokkermans V (2007) Verifying CSP-OZ-DC specifications with complex data types and timing parameters. In: Davies J, Gibbons J (eds) Integrated Formal Methods. LNCS, vol 4591. Springer, Heidelberg, pp 233–252

[Frä04] Fränzle M (2004) Model-checking dense-time duration calculus. Formal Asp Comput 16(2):121–139

[GS97] Graf S, Saidi H (1997) Construction of abstract state graphs with PVS. In: Grumberg O (ed.) CAV, vol 1254. Springer, Heidelberg, pages 72–83

[Han06] Hansen M (2006) DC with nominals. Personal communication, March (2006)

[HJMM04] Henzinger TA, Jhala R, Majumdar R, McMillan KL (2004) Abstractions from proofs. In: Jones ND, Leroy X (eds) POPL. ACM Press, New York, pp 232–244

[HJU05] Hermanns H, Jansen DN, Usenko YS (2005) From StoCharts to MoDeST: a comparative reliability analysis of train radio communications. In: WOSP. ACM Press, New York, pp 13–23

[HM05] Hoenicke J, Maier P (2005) Model-checking of specifications integrating processes, data and time. In: Fitzgerald JS, Hayes IJ, Tarlecki A (eds) FM, LNCS, vol 3582. Springer, Heidelberg, pp 465–480

[HMF06] Hoenicke J, Meyer R, Faber J (2006) PEA toolkit home page. http://csd.informatik.uni-oldenburg.de/projects/epea.html

[HO02] Hoenicke J, Olderog ER (2002) CSP-OZ-DC: A combination of specification techniques for processes, data and time. Nordic J Comput 9

[Hoa85] Hoare CAR (1985) Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs

[Hoe06] Hoenicke J (2006) Combination of Processes, Data, and Time. Ph.D. thesis, University of Oldenburg

[KP05] Krishna SN, Pandya PK (2005) Modal strength reduction in quantified discrete duration calculus. In: Ramanujam R, Sen S (eds) FSTTCS, LNCS, vol 3821. Springer, Heidelberg, pp 444–456

[McM03] McMillan KL (2003) Interpolation and SAT-based model checking. In: Hunt WA Jr, Somenzi F (eds) CAV, LNCS, vol 2725. Springer, Heidelberg, pp 1–13

[MFR06] Meyer R, Faber J, Rybalchenko A (2006) Model checking duration calculus: A practical approach. In: Barkaoui K, Cavalcanti A, Cerone A (eds) ICTAC, LNCS, vol 4281. Springer, Heidelberg, pp 332–346

[Pan02] Pandya PK (2002) Interval duration logic: Expressiveness and decidability. ENTCS 65(6)

[Pla07] Platzer A (2007) Differential dynamic logic for verifying parametric hybrid systems. In: Olivetti N (ed.) TABLEAUX, LNCS, vol 4548. Springer, Heidelberg, pp 216–232

[PR05] Podelski A, Rybalchenko A (2005) Transition predicate abstraction and fair termination. In: POPL. ACM Press, New York, pp 132–144

[PR07] Podelski A, Rybalchenko A (2007) ARMC: the logical choice for software model checking with abstraction refinement. In: PADL, LNCS, vol 4281. Springer, Heidelberg, pp 245–259

[Rav94] Ravn AP (1994) Design of Embedded Real-Time Computing Systems. Ph.D. thesis, Technical University of Denmark

[Ros98] Roscoe AW (1998) Theory and Practice of Concurrency. Prentice Hall, Englewood Cliffs

[RSS07] Rybalchenko A, Sofronie-Stokkermans V (2007) Constraint solving for interpolation. In: VMCAI, LNCS, vol 4349. Springer, Heidelberg, pp. 346–362
[Ryb07] Rybalchenko A (2007) ARMC. http://www.mpi-sws.mpg.de/~rybal/armc
[Smi00] Smith G (2000) The Object-Z Specification Language. Kluwer, Dordrecht
[UPP05] Uppaal home page. University of Aalborg and University of Uppsala. http://www.uppaal.com, 1995–2005
[Var91] Vardi MY (1991) Verification of concurrent programs: The automata-theoretic framework. Ann Pure Appl Logic 51(1–2): 79–98
[VW86] Vardi MY, Wolper P (1986) An automata-theoretic approach to automatic program verification. In: LICS. IEEE Computer Society, pp 332–344
[ZH04] Zhou C, Hansen MR (2004) Duration Calculus. Springer, Heidelberg
[ZH05] Zimmermann A, Hommel G (2005) Towards modeling and evaluation of ETCS real-time communication and operation. J Syst Softw 77(1):47–54
[ZHS93] Zhou C, Hansen MR, Sestoft P (1993) Decidability and undecidability results for duration calculus. In: Enjalbert P, Finkel A, Wagner KW (eds) STACS, LNCS, vol 665. Springer, Heidelberg, pp 58–68

Received 2 May 2007
Accepted in revised form 3 April 2008 by K. Barkaoui, M. Broy, A. Cavalcanti and A. Cerone
Published online 6 May 2008