Model Checking Duration Calculus: A Practical Approach[⋆]

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

[1] Carl-von-Ossietzky-Universität Oldenburg
[2] Ecole Polytechnique Fédérale de Lausanne
[3] Max-Planck-Institut Informatik Saarbrücken

Abstract. Model checking of real-time systems with respect to Duration Calculus (DC) specifications requires the translation of DC formulae into automata-based semantics. This task is difficult to automate. 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 real world applications. Our algorithm significantly extends the subset of DC that can be handled. It decomposes DC specifications into sub-properties that can be verified independently. The decomposition bases on a novel distributive law for DC. We implemented the algorithm as part of our tool chain for the automated verification of systems comprising data, communication, and real-time aspects. Our translation facilitated a successful application of the tool chain on an industrial case study from the European Train Control System (ETCS).

1 Introduction

Verification of embedded hardware and software systems requires reasoning about data, communication, and real-time aspects. Duration Calculus (DC) represents these dimensions in one formalism. As a fundamental concept, it offers the use of data variables with possibly infinite data domains that are interpreted over dense real-time intervals.

To apply the automata theoretic approach of Vardi and Wolper [VW86] for model checking DC, we need to translate 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 are known for restricted classes of DC only [Rav94, BLR95, Pan02, Frä04]. But they are not compositional and consider neither infinite data domains nor communication.

We identify a new class of DC formulae, called test formulae, that can be translated into automata, also referred to as test automata in this paper. Test

⋆ This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS) and the Graduate School "TrustSoft" (GRK 1076/1). See `http://www.avacs.org` and `http://www.trustsoft.org`.

formulae (1) significantly extend the previously known classes and (2) take communication aspects and infinite data domains into account. Our expressiveness results suggest that the new class is among the richest for which satisfiability with respect to an automaton is decidable under a dense time interpretation.

Translations of DC suffer from an exponential blow up of the resulting automata in the number of operators. To overcome this problem, we provide an algorithm that decomposes a formula into sub-formulae that are translated independently. It allows for an efficient verification as it reduces the size of the automata. The decomposition is realised using a new operator for the DC that permits a distributive law of linear complexity.

We implemented our translation algorithm as part of a tool chain and provide evidence that it can handle industrial problems. We verify the emergency treatment of the European Train Control System (ETCS) [ERT02]. Our approach is the first that permits model checking of a comprehensive ETCS fragment considering communication, data, and real-time. Therefore, we bridge the gap between theoretical results and their practical applications. Due to our model's parameters and infinite data types, we apply the abstraction refinement model checker ARMC [Ryb06].

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. The normal form is realised using a new distributive law of linear complexity based on a new operator for the DC. We implemented the algorithm and applied our tool chain to verify real-time properties of the ETCS case study.

The paper is organised as follows. After a short introduction to our case study, we recall 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 states its correctness. The case study and our model checking results are sketched in Sect. 5. Section 6, reviewing related work and suggesting future investigations, concludes the paper.

1.1 Motivating Example

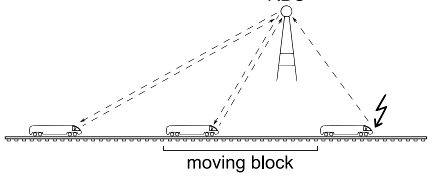


Fig. 1. Consecutive trains.

The emerging European Train Control System (ETCS) is an international standard [ERT02] that shall replace national train control systems to ensure cross-border interoperability and to improve railway safety as well as track utilisation. In the final ETCS implementation level, the existing national trackside systems for detection of train speed, location, and integrity will not be used anymore. Instead, data values required for a moving train are ascertained in cooperation of the train's on-board ETCS unit with a radio block centre (RBC) that controls the traffic in a well-defined area and grants movement authorities to trains. RBCs and trains communicate over a GSM-R radio connection. To increase the possible traffic density, the ETCS employs the moving block principle, by which the movement authorities are always given 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 accident, the train control system has to stop all trains safely. The main 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. It is the first time, a fragment of the ETCS is verified considering all of these aspects.

2 Preliminaries

Since we translate DC formulae into phase event automata (PEA), we review the 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.

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

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. Consider $\sim \in \{\leq, <, =, >, \geq\}$, $k \in \mathbb{R}_{\geq 0}$. The class of DC formulae $(F \in)\ \mathbb{F}orm$ is defined by

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

Given an interpretation $\mathcal{I}$ of the observables in state expressions, the semantics of a DC formula F is a mapping evaluating the formula on a given finite interval.

$$\mathcal{I}[\![\lceil \varphi \rceil]\!][b, e] := tt \text{ iff } \textstyle\int_b^e \mathcal{I}[\![\varphi]\!](t)\, dt = e - b \text{ and } e > b$$

$$\begin{aligned}
\mathcal{I}[\![\ell \sim k]\!][b,e] &:= \mathit{tt} \text{ iff } (e-b) \sim k\\
\mathcal{I}[\![\neg F]\!][b,e] &:= \mathit{tt} \text{ iff } \mathcal{I}[\![F]\!][b,e] = \mathit{ff}\\
\mathcal{I}[\![F_1 \wedge F_2]\!][b,e] &:= \mathit{tt} \text{ iff } \mathcal{I}[\![F_1]\!][b,e] = \mathit{tt} \text{ and } \mathcal{I}[\![F_2]\!][b,e] = \mathit{tt}\\
\mathcal{I}[\![F_1 \,\mathbf{;}\, F_2]\!][b,e] &:= \mathit{tt} \text{ iff there is } m \in [b,e] \text{ such that}\\
&\qquad\quad \mathcal{I}[\![F_1]\!][b,m] = \mathit{tt} \text{ and } \mathcal{I}[\![F_2]\!][m,e] = \mathit{tt}\\
\mathcal{I}[\![\exists\, X : F]\!][b,e] &:= \mathit{tt} \text{ iff there is } \mathcal{I}' =_{\setminus X} \mathcal{I} \text{ such that } \mathcal{I}'[\![F]\!][b,e] = \mathit{tt}.
\end{aligned}$$

Two interpretations are equal up to X, $\mathcal{I}' =_{\setminus X} \mathcal{I}$, if they coincide on all observables except X. The finite variability ensures that $\mathcal{I}[\![\varphi]\!]$ is integrable.
Two formulae F_1, F_2 are *satisfiability equivalent* iff for any interpretation $\mathcal{I}$ 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 Section 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 at time t iff the value of $\mathcal{E}$ changes at t. A *forbidden event* $\not\updownarrow\mathcal{E}$ holds at time t iff the value of $\mathcal{E}$ does not change at t. For an interval the *no event* formula $\boxminus\mathcal{E}$ holds iff the value of $\mathcal{E}$ is constant in the given interval.

2.2 Phase Event Automata

PEA [HM05] are a class of timed automata [AD94] that synchronise on both events and data variables. Let $\mathcal{L}(V)$ be the set of first-order formulae over variables in V.

Definition 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 phases with initial phases* $P^0 \subseteq P$*,*
- 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 phase a predicate that holds during the phase, and*
- $I : P \to \mathcal{L}(C)$ *associates with each phase a clock invariant.*

An edge (p_1, g, X, p_2) *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 to be 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 edge $(p, \bigwedge_{e\in A} \neg e \wedge \bigwedge_{v\in V} v' = v, \varnothing, p)$ *for every phase* p*.*

The operational semantics of PEA is given by infinite sequences of configurations and events, called runs.

Definition 2 (Run of a PEA). *A* run *of a PEA $\mathcal{A}$ is a sequence*

$$\big\langle (p_0, \beta_0, \eta_0), t_0, Y_0, (p_1, \beta_1, \eta_1), t_1, Y_1, \ldots \big\rangle,$$

with phases $p_i \in P$, entry event sets $Y_i \subseteq A$, valuations of variables β_i and primed variables β_i', where $\beta_i(v) = \beta_i'(v')$, clock valuations η_i, and points in time $t_i > 0$. Furthermore, we demand $p_0 \in P^0, \eta_0(c) = 0$ for all clocks $c \in C$, $\beta_i \models s(p_i)$, and $\eta_i + t_i \models I(p_i)$. For all transitions (p_i, g, X, p_{i+1}) we require $\beta_i, \beta_{i+1}', \eta_i + t_i, Y_i \models g$ and $\eta_{i+1} = (\eta_i + t_i)[X := 0]$. We denote the set of all runs of $\mathcal{A}$ by **Run**$(\mathcal{A})$*.*

PEA composed in parallel synchronise over common events and additionally over common variables. That is, a variable that occurs in both automata may only be changed if both automata agree.

Definition 3 (Parallel Composition). *The parallel composition of PEA $\mathcal{A}_1$ and $\mathcal{A}_2$ with $\mathcal{A}_i = (P_i, V_i, A_i, C_i, E_i, s_i, I_i, P_i^0)$ is given by*

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

where $((p_1, p_2), g_1 \wedge g_2, X_1 \cup X_2, (p_1', p_2')) \in E$ iff $(p_i, g_i, X_i, p_i') \in E_i$ with $i = 1, 2$.

This parallel product allows for compositional verification, because once a safety property is proven for an arbitrary subset of parallel components, it is also true for the entire system.

3 Test Formulae

In this section, we introduce the DC subclass of test formulae, denoted by *Testform.* For test formulae we construct test automata in Sect. 4. Applying the automata theoretic approach [VW86, ABBL03], we can automatically decide whether a system satisfies a negated test formula. Thus, test formulae may be interpreted as undesired system behaviour.

We use so-called trace formulae to specify system executions. The class *Testform* is built up from trace formulae and admits a restricted use of negation.

Definition 4 (*Testform*). *The formula class Testform is defined inductively:*

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

where $k \in \mathbb{R}_{>0}$, φ is a state expression, $\mathcal{E}$ is a Boolean observable, and $\sim \in \{\varnothing, \leq, <, >, \geq\}$. We use $\sim = \varnothing$ to indicate $\ell > 0$ is the only time bound. We impose the condition that the first element of a trace always is a phase.

In our running example, undesired behaviour is that the leading train sends an alert message, indicated by formula (2), but for longer than five time units neither the leading nor the following train applies the brakes, stated in formula (3), with $i = 1, 2$. Test formula (4) reflects the critical behaviour:

$$\begin{aligned}
\mathit{Warn} :=& \lceil \mathit{true} \rceil \,;\, \updownarrow \mathit{Train}_1\mathit{ToRBC\_Alert} \,;\, \lceil \mathit{true} \rceil; \\
& \updownarrow \mathit{RBCToTrain}_1\mathit{\_Warn}_1 \,;\, \lceil \mathit{true} \rceil \,;\, \updownarrow \mathit{RBCToTrain}_2\mathit{\_Warn}_2 && (2)\\
\mathit{NoBrake}_i :=& \boxminus \mathit{ApplyEmergencyBrake}_i \wedge \ell > 5 && (3)\\
\mathit{TF} :=& \mathit{Warn} \,;\, (\mathit{NoBrake}_1 \wedge \mathit{NoBrake}_2) \,;\, \lceil \mathit{true} \rceil. && (4)
\end{aligned}$$

A different approach would express the undesired behaviour directly in terms of test automata. The benefit of DC is its conciseness. A negated DC trace comprising n phases requires in the worst case a test automaton of size 4^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 $\underset{\mathcal{S}}{\Updownarrow}$ i.e., distinguished events occurring only once. They can be used to uniquely identify a chop point. For sync events the following distributivity holds:

$$F \underset{\mathcal{S}}{\Updownarrow} (G \wedge H) \Leftrightarrow \left(F \underset{\mathcal{S}}{\Updownarrow} G \right) \wedge \left(F \underset{\mathcal{S}}{\Updownarrow} H \right). \qquad (5)$$

Definition 5 (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 *$F \underset{\mathcal{S}}{\Updownarrow} G$ is defined as follows:*

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

To introduce sync events to the class of test formulae, equivalence (6) 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 1 (Sync Event Introduction and Linear Distributivity). *Let $\mathcal{S}$ be a Boolean observable not contained in F, F_i, G, G_j, $1 \leq i \leq m, 1 \leq j \leq n$, $m, n \in \mathbb{N}$. The following equivalences hold:*

$$(F \wedge \ell > 0) \,;\, G \Leftrightarrow \exists \mathcal{S} : \left(F \underset{\mathcal{S}}{\Updownarrow} G \right) \qquad (6)$$

$$\left(\bigwedge_{i=1}^{m} F_i\right) \underset{\mathcal{S}}{\Updownarrow} \left(\bigwedge_{j=1}^{n} G_j\right) \Leftrightarrow \bigwedge_{i=1}^{m} \left(F_i \underset{\mathcal{S}}{\Updownarrow} true\right) \wedge \bigwedge_{j=1}^{n} \left(\lceil true \rceil \underset{\mathcal{S}}{\Updownarrow} G_j\right) . \tag{7}$$

We know that the *true* phase before a sync event has a duration greater zero, i.e., $\lceil true \rceil$ holds, because events cannot happen at time zero. The distributivity in equivalence (7) results in $m + n + 1$ conjuncted formulae compared to the distributivity in (5) resulting in $m * n$ formulae:

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

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

Our normal form is a disjunctive normal form (DNF) over traces.

Theorem 1 (Normal Form Theorem). *Every test formula is satisfiability equivalent with a formula of the form*

$$\exists \mathcal{S}_{ijk} : \bigvee\nolimits_i \bigwedge\nolimits_j \mathcal{T}_{ij}, \tag{8}$$

$$\textit{with } \mathcal{T}_{ij} ::= Tr_{ij} \underset{\mathcal{S}_{ij}}{\Updownarrow} \lceil true \rceil \mid \lceil true \rceil \underset{\mathcal{S}_{ij1}}{\Updownarrow} Tr_{ij} \underset{\mathcal{S}_{ij2}}{\Updownarrow} \lceil true \rceil, \tag{9}$$

where Tr_{ij} are (negated) traces, $k = 1, 2$, and $\mathcal{S}_{ijk}$ are fresh Boolean observables.

For the construction of the normal form, we assume the given test formula *TF* to end with a $\lceil true \rceil$ phase (cf. satisfiability equivalence, Sect. 2.1). We then replace every *Form* formula inside *TF* with its DNF. To obtain the outermost disjunctions in (8), we apply the known distributivities for disjunction and chop/conjunction to the resulting formula. We end with chop separated conjunctions of (negated) traces. For all these chops, we introduce sync events (6) and use distributivity (7).

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 (7) neither increases the number of (negated) traces nor the size of the product automata (cf. restriction, Sect. 4).

For example, we gain the normal form of formula (4) by introducing two sync events (6) and using the distributivity of sync events and conjunctions (7):

$$\begin{aligned} &Warn \,;\, (NoBrake_1 \wedge NoBrake_2) \,;\, \lceil true \rceil \\ \Leftrightarrow\; &\exists \mathcal{S}_0 : \exists \mathcal{S}_1 : Warn \underset{\mathcal{S}_0}{\Updownarrow} \lceil true \rceil \wedge \bigwedge\nolimits_{i=1,2} \lceil true \rceil \underset{\mathcal{S}_0}{\Updownarrow} NoBrake_i \underset{\mathcal{S}_1}{\Updownarrow} \lceil true \rceil . \end{aligned}$$

4 Model Checking with Test Automata

To define whether a PEA model of a system satisfies a test formula, we need to clarify the meaning of satisfiability of a DC formula with respect to a PEA (cf. Definitions 1 and 2). Given Boolean observables $\mathcal{E}_1 \dots, \mathcal{E}_n$ and observables $X_1, \dots, X_m$, an interpretation $\mathcal{I}$ is said to *fit to a run* r iff

– the set of events A in the PEA can be identified with the set of interpreted Boolean observables, $A = \{\mathcal{E}_1 \dots, \mathcal{E}_n\}$, the set of variables in the PEA is identical with the set of interpreted observables, $V = \{X_1 \dots, X_m\}$,
– 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, $\mathcal{E}_i \in Y$.

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

Definition 6. *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:*

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

4.1 Test Automata

Test automata (TA) are PEA with a distinguished state, 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 the system with a TA means that the system can exhibit the undesired behaviour specified in the test formula the TA is constructed for.

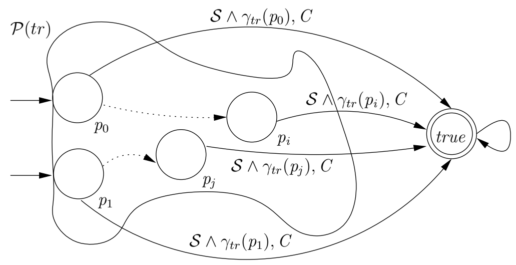
We define the TA semantics for the normal form of test formulae. Therefore, we require three operations on TA: parallel composition to express the conjunction, sequential composition to represent the formula structure in (9), and restriction to model sync events.

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 as the pair of the bad states of the original automata.

The *sequential composition* of two test automata, denoted by $TA_1 \underset{\mathcal{S},\gamma}{\bullet} TA_2$, means the second TA is started when the first one has accepted its formula. Since the acceptance of trace formulae depends 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 state. We define the sequential composition as follows. A transition between every state in the first automaton and every initial state in the second automaton is inserted. The new transitions demand an event $\mathcal{S}$ representing the sync events in (9). Furthermore, they require a guard that holds iff the test formula represented by the first TA is satisfied. The guard is given by the function γ. All clocks in both automata are reset when the first TA is left.

As an example, consider the formula $tr \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil$ for a trace tr. Figure 2 represents the structure of the TA for the sequential composition $\mathcal{P}(tr) \underset{\mathcal{S},\gamma_{tr}}{\bullet} \mathcal{P}(\lceil true \rceil)$ connecting the trace automaton $\mathcal{P}(tr)$ with the automaton $\mathcal{P}(\lceil true \rceil)$.

Given a test automaton TA, the *restriction of TA to the event* $\mathcal{S}$, denoted by $TA \setminus \{\mathcal{S}\}$, 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 $TA \setminus \{\mathcal{S}\}$, otherwise, the transition remains unchanged. The restriction operator is used to make the occurrences of sync events unique.



$\mathcal{P}(tr)$
$\mathcal{S} \wedge \gamma_{tr}(p_0), C$
$\mathcal{S} \wedge \gamma_{tr}(p_i), C$
p_0
p_i
true
$\mathcal{S} \wedge \gamma_{tr}(p_j), C$
p_j
p_1
$\mathcal{S} \wedge \gamma_{tr}(p_1), C$

Fig. 2. Sequential composition (illustration).

4.2 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 1) TF a test automaton $\mathcal{P}(TF)$. To begin with, we sketch the following non-compositional TA construction for traces. A trace tr consists of several subsequent phases. A state in the TA $\mathcal{P}(tr)$ represents a subset of these phases combined with a set of flags. For each phase p in this set all runs leading to the state accept the prefix of the trace up to p. The flags indicate the bound types ($\varnothing, \leq, <, >, \geq$) that need to be used for every phase in this state. Given a state in the TA, a successor state is computed for every possible event set, clock and variable valuation. This results in a deterministic automaton, that may grow exponentially in the number of phases inside the trace. The phase p is accepted, if the given valuation and event set satisfy the guard function $\gamma_{tr,p}$ of this phase. The successor state contains the next phase in the trace. In Definition 7, the function γ_{tr} is the guard function of the last phase in the trace. Details of the construction and the guard function can be found in [Hoe06].

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 7 (Test Automata Semantics). *Let tr be a trace and $\mathcal{S}, \mathcal{S}_1, \mathcal{S}_2$ be Boolean observables. The test automata semantics for a test formula in normal form TF yields a PEA $\mathcal{P}(TF)$ defined as follows:*

$$\mathcal{P}(tr \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil) := \big(\mathcal{P}(tr) \underset{\mathcal{S},\gamma_{tr}}{\bullet} \mathcal{P}(\lceil true \rceil)\big) \setminus \{\mathcal{S}\}$$

$$\mathcal{P}(\neg tr \underset{\mathcal{S}}{\Updownarrow} \lceil true \rceil) := \big(\mathcal{P}(tr) \underset{\mathcal{S},\neg\gamma_{tr}}{\bullet} \mathcal{P}(\lceil true \rceil)\big) \setminus \{\mathcal{S}\}$$

$$\begin{aligned}\mathcal{P}(\lceil true \rceil \underset{\mathcal{S}_1}{\Updownarrow} Tr \underset{\mathcal{S}_2}{\Updownarrow} \lceil true \rceil) :=& \Big[\big(\mathcal{P}(\lceil true \rceil) \underset{\mathcal{S}_1, true}{\bullet} \mathcal{P}(Tr \underset{\mathcal{S}_2}{\Updownarrow} \lceil true \rceil)\big) \setminus \{\mathcal{S}_2\}\Big] \setminus \{\mathcal{S}_1\} \\ \mathcal{P}(TF_1 \wedge TF_2) :=& \mathcal{P}(TF_1) \parallel \mathcal{P}(TF_2),\end{aligned}$$

where Tr is a (negated) trace, TF_1 and TF_2 are in the form of (9). The function γ_{tr} guarantees that the trace tr is accepted.

Figure 3 shows the TA semantics for the formula $\lceil true \rceil \underset{\mathcal{S}_0}{\Updownarrow} NoBrake_1 \underset{\mathcal{S}_1}{\Updownarrow} \lceil true \rceil$, simplified by removing a transition with guard $false$ from state 2 to state 4. State 2 and state 3 represent the states of the trace automaton $\mathcal{P}(NoBrake_1)$.

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.

Lemma 2 (Characterisation of Satisfiability with Test Automata). *Consider the normal form $\bigvee_i \bigwedge_j \mathcal{T}_{ij}$ of a test formula. Given an interpretation $\mathcal{I}$ and $t \in \mathbb{R}_{\geq 0}$, the following equivalence holds for every disjunct:*

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

With Lemma 2 we can reduce the problem whether a PEA satisfies a negated test formula to a reachability question. The correctness of our semantics with respect to model checking is stated in the following theorem.

Theorem 2 (Model Checking Theorem). *Let TF be a test formula with the normal form $\bigvee_i \bigwedge_j \mathcal{T}_{ij}$. The question whether the negated test formula is satisfied by a PEA $\mathcal{A}$ can be decided as follows:*

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

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

The decidability of the reachability problem depends on the constraints over the state variables of the PEA.

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. If the bad state is reachable in the TA for the subset, further formulae may 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.

5 Case Study: Real-Time Aspects of the ETCS

In this section we take up the case study of Sect. 1.1 for the experimental evaluation of our verification method.

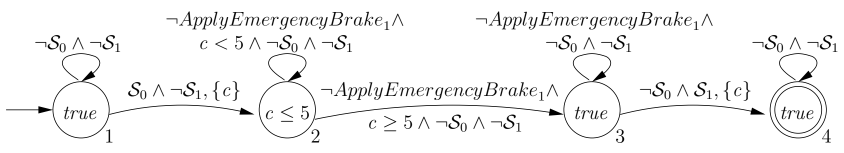
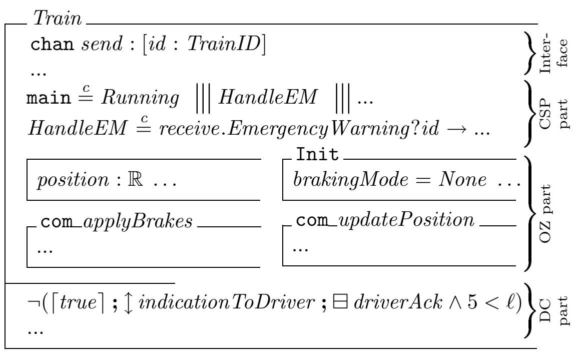

$\neg\mathcal{S}_0 \wedge \neg\mathcal{S}_1$
$\neg ApplyEmergencyBrake_1 \wedge$ $c < 5 \wedge \neg\mathcal{S}_0 \wedge \neg\mathcal{S}_1$
$\neg ApplyEmergencyBrake_1 \wedge$ $\neg\mathcal{S}_0 \wedge \neg\mathcal{S}_1$
$\neg\mathcal{S}_0 \wedge \neg\mathcal{S}_1$
$true$ 1
$\mathcal{S}_0 \wedge \neg\mathcal{S}_1, \{c\}$
$c \leq 5$ 2
$\neg ApplyEmergencyBrake_1 \wedge$ $c \geq 5 \wedge \neg\mathcal{S}_0 \wedge \neg\mathcal{S}_1$
$true$ 3
$\neg\mathcal{S}_0 \wedge \mathcal{S}_1, \{c\}$
$true$ 4

Fig. 3. Test automaton for $\lceil true \rceil \underset{\mathcal{S}_0}{\Updownarrow} NoBrake_1 \underset{\mathcal{S}_1}{\Updownarrow} \lceil true \rceil$.

Complex systems like the ETCS consist of several components running in parallel, by the communications between these components, by internal data and state changes, and by real-time aspects. We use the declarative formal language CSP-OZ-DC [HO02] to model our case study. CSP-OZ-DC integrates the well-investigated languages CSP [Hoa85], Object-Z [Smi00], and DC [ZH04] into a unified formalism. CSP-OZ-DC is given an operational semantics [Hoe06] in terms of PEA.

Our case study incorporates five different components that can be modelled with CSP-OZ-DC in an object-oriented way using classes: *Train*, *RBC*, *Track*, *Driver*, and a communication layer *ComNetwork*, which is necessary to model the transfer times of messages between trains and RBC. Every CSP-OZ-DC class comprises an interface part (Fig. 4) defining channels that can be used for the inter-class communication.


Train

chan $send : [id : TrainID]$ ... } Inter-face

main $\overset{c}{=} Running \;|||\; HandleEM \;|||\; ...$
$HandleEM \overset{c}{=} receive.EmergencyWarning?id \rightarrow ...$ } CSP part

$position : \mathbb{R} \ldots$

Init: $brakingMode = None \ldots$

com_*applyBrakes*: ...

com_*updatePosition*: ... } OZ part

$\neg(\lceil true \rceil \,;\, \updownarrow indicationToDriver \,;\, \boxminus driverAck \wedge 5 < \ell)$
... } DC part

Fig. 4. Exemplary train class.

The external and internal *communications* of parallel components are described with Communicating Sequential Processes (CSP) [Hoa85]. These processes communicate over channels (or events) that facilitate the transfer of data values, e.g., the main process of a train comprises the interleaving of three subprocesses. When the RBC sends an emergency warning, the train receives this message on the channel *receive* with the process *HandleEM* (Fig. 4).

Data aspects are specified with the object-oriented specification language Object-Z (OZ) [Smi00]. The OZ part consists of schemas describing data changes of a class. For instance, the OZ part of *Train* (Fig. 4) includes the state schema defining attributes of the class, e.g., *position*, the Init-schema defining that initially the train is not braking, and operation schemas, e.g., com_*applyBrakes* defining data changes that are performed at the same time when—in agreement with the CSP part of the class—the event *applyBrakes* occurs. In particular, our case study comprises, besides the real-time aspects, infinite data types, e.g., the positions, that are modelled as reals. Furthermore, the values of such infinite data types are also transferred via channels to other classes. Another important property of the data handling in CSP-OZ-DC is the use of parameters, i.e., we do not need to interpret all constants. Instead, it suffices to specify conditions that restrict the values adequately. In our case study, we have a parameter for the length of trains and the only condition we need is $\mathit{length} > 0$.

Real-time constraints are described using the logic DC [ZH04]. Since the full DC is too expressive for automatic verification, we only use counterexample-trace formulae, i.e., negated trace formulae according to Sect. 3.

The operational semantics of CSP-OZ-DC is given in terms of PEA, which can handle infinite data types and parameters. It is compositional in the sense that every part (CSP, OZ, and DC) of every component is translated into a single PEA, and the whole specification is translated into the parallel product of all these automata. For details we refer to [HM05, Hoe06].

The desired safety property in our case study is that the trains will never collide. For a setting with two trains, this can be expressed in the DC formula

$$\neg(\lceil \mathit{true} \rceil \mathbin{;} \lceil \mathit{position}_1 > \mathit{position}_0 - \mathit{length}_0 \rceil), \tag{10}$$

where $\mathit{position}_0$ is the position of the first train with length length_0. The variable $\mathit{position}_1$ represents the position of the subsequent train.

5.1 Tool Support

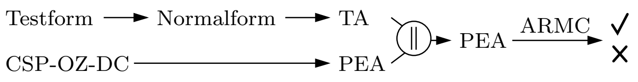


Fig. 5. Flow of the verification process.

In order to verify whether a CSP-OZ-DC model satisfies a test formula, we execute the following steps (cf. Fig. 5). We translate the model into PEA according to its semantics. The translation of the DC part is automated. To develop PEA for the CSP and the OZ part the graphical design tool Moby/PEA [HMF06] is available. The DC test formula is transformed into a set of test automata (TA), applying the algorithm introduced in Sect. 3 and 4. To this end, we implemented a compiler (available on [HMF06]) that automatically computes the normal form and the corresponding test automata semantics. In a next step, we compute the parallel composition of the test automata and the PEA of the model. Our tool generates outputs in the Uppaal [UUP05] and ARMC [Ryb06] supported formats. Finally, we apply a model checker on the product automaton. For our case study, Uppaal is of limited use, because it can neither cope with infinite data domains nor parameters.

We use the abstraction refinement model checker ARMC [Ryb06] for infinite state systems to prove the unreachability of bad states in PEA. We implemented a new abstraction refinement method in ARMC that allows us to handle large input PEA from the case study. ARMC automatically constructs a safe abstraction of the input PEA. The abstraction is defined by predicates over PEA variables, events, and clocks, and computed in the standard way [GS97]. The process of

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

Task	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
Running	178	6.1T	31	46	347	22	25s	26m
Running (decomp. 1)	8	150	20	11	11	8	2.5s	7.5s
Running (decomp. 2)	20	899	22	8	32	8	4.0s	21.5
Running (decomp. 3)	48	1.2T	27	13	93	10	5.9s	45s
Running (decomp. 4)	48	1.7T	27	11	70	7	6.3s	47.5s
Delivery	122	18T	20	41	2.2T	32	50s	86m
Delivery (decomp. 1)	14	366	14	9	29	8	2.7s	13.9s
Delivery (decomp. 2)	17	173	10	25	17	17	2.2s	1.9s
Delivery (decomp. 3)	12	71	9	12	9	9	1.9s	0.7s
Delivery (decomp. 4)	17	156	12	25	19	17	2.2s	2.6s
Delivery (decomp. 5)	7	28	4	3	5	3	1.6s	0.1s
Braking 1	44	240	17	45	44	3	3s	5.1s
Braking 2	172	1.6T	33	63	88	59	9s	35.3s

(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

T : thousand units
m : minutes   s : seconds

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 [HJMM04, McM03]. We designed and implemented an efficient algorithm for the computation of interpolants for linear arithmetic over rationals/reals based on linear programming, which are particularly needed for the verification of real-time systems.

5.2 Results

The model of the case study is too large to verify the global safety property (10) in a single step. Therefore, we decompose the model manually into smaller parts and verify local properties for the parallel components. The semantics of CSP-OZ-DC ensures that local properties hold for the entire system (cf. Sect. 2.2).

Table 1 shows our experimental results for a range of verification tasks. For instance, we consider the running behaviour of the train in isolation and verify (10) on the assumption that the first train does not apply the emergency brakes. To this end, we take only those PEA into account that influence the running behaviour, i.e., the automata for the subprocess *Running* (Fig. 4) together with the automata for the OZ and the DC part. The performance results of applying our model checking approach to this verification task are listed as "Running" in Tab. 1. The other entries (decomp. 1 – decomp. 4) contain our results for a further (manual) decomposition of "Running" into smaller tasks that allows for a more efficient verification. For the "Delivery" task (and also the decomposed variants) we verify that messages like an emergency message between train and RBC are not delivered too late. Bringing together the verification tasks and showing that they imply (10) for the entire model is subject of ongoing work.

The table illustrates that we can handle up to 18000 program transitions and up to 33 variables (with potentially infinite data types) in an order of 86 min. Hence, these results demonstrate that our new algorithm implemented in our tool chain can deal with problems in real world applications.

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].

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 can be identified uniquely using their 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 the full DC.

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.

We currently work on model checking DC liveness properties with the automata theoretic approach. In addition, enhancing our decomposition techniques is ongoing work. They allow for compositional verification of inherently parallel systems like the ETCS.

References

[ABBL03] L. Aceto, P. Bouyer, A. Burgueño, and K. G. Larsen. The power of reachability testing for timed automata. *Theoretical Computer Science*, 300(1-3):411–475, 2003.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[BLR95] A. Bouajjani, Y. Lakhnech, and R. Robbana. From duration calculus to linear hybrid automata. In *CAV*, volume 939 of *LNCS*, pages 196–210. Springer-Verlag, 1995.

[CGJ$^+$00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, volume 1855 of *LNCS*, pages 154–169. Springer-Verlag, 2000.

[DL02] H. Dierks and M. Lettrari. Constructing test automata from graphical real-time requirements. In *FTRTFT*, volume 2469 of *LNCS*, pages 433–453. Springer-Verlag, 2002.

[ERT02] ERTMS User Group, UNISIG. ERTMS/ETCS System requirements specification. http://www.aeif.org/ccm/default.asp, 2002. Version 2.2.2.

[Frä04] M. Fränzle. Model-checking dense-time duration calculus. *Formal Aspects of Computing*, 16(2):121–139, 2004.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254, pages 72–83. Springer-Verlag, 1997.

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

[HJMM04] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244. ACM Press, 2004.

[HJU05] H. Hermanns, D. N. Jansen, and Y. S. Usenko. From StoCharts to MoDeST: a comparative reliability analysis of train radio communications. In *WOSP*, pages 13–23. ACM Press, 2005.

[HM05] J. Hoenicke and P. Maier. Model-checking of specifications integrating processes, data and time. In *FM 2005*, volume 3582 of *LNCS*, pages 465–480. Springer-Verlag, 2005.

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

[HO02] J. Hoenicke and E.-R. Olderog. CSP-OZ-DC: A combination of specification techniques for processes, data and time. *NJC*, 9, 2002.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hoe06] J. Hoenicke. *Combination of Processes, Data, and Time*. PhD thesis, University of Oldenburg, Germany, 2006. To appear.

[KP05] S. N. Krishna and P. K. Pandya. Modal strength reduction in quantified discrete duration calculus. In *FSTTCS*, volume 3821 of *LNCS*, pages 444–456. Springer-Verlag, 2005.

[McM03] K. L. McMillan. Interpolation and SAT-based model checking. In *CAV*, volume 2725 of *LNCS*, pages 1–13. Springer-Verlag, 2003.

[Pan02] P. K. Pandya. Interval duration logic: Expressiveness and decidability. *ENTCS*, 65(6), 2002.

[Rav94] A. P. Ravn. *Design of Embedded Real-Time Computing Systems*. PhD thesis, Technical University of Denmark, 1994.

[Ryb06] A. Rybalchenko. ARMC. http://www.mpi-inf.mpg.de/~rybal/armc, 2006.

[Smi00] G. Smith. *The Object-Z Specification Language*. Kluwer, 2000.

[UUP05] Uppaal home page. University of Aalborg and University of Uppsala, http://www.uppaal.com, 1995-2005.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS*, pages 332–344, 1986.

[ZH04] C. Zhou and M. R. Hansen. *Duration Calculus*. Springer-Verlag, 2004.

[ZH05] A. Zimmermann and G. Hommel. Towards modeling and evaluation of ETCS real-time communication and operation. *JSS*, 77(1):47–54, 2005.

[ZHS93] C. Zhou, M. R. Hansen, and P. Sestoft. Decidability and undecidability results for duration calculus. In *STACS*, volume 665 of *LNCS*, pages 58–68. Springer-Verlag, 1993.