Automatic Verification of Parametric Specifications with Complex Topologies*

Johannes Faber[1], Carsten Ihlemann[2], Swen Jacobs[3], and Viorica Sofronie-Stokkermans[2]

[1] Department of Computing Science, University of Oldenburg, Germany
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[3] École Polytechnique Fédérale de Lausanne, Switzerland

Abstract. The focus of this paper is on reducing the complexity in verification by exploiting modularity at various levels: in specification, in verification, and structurally. For specifications, we use the modular language CSP-OZ-DC, which allows us to decouple verification tasks concerning data from those concerning durations. At the verification level, we exploit modularity in theorem proving for rich data structures and use this for invariant checking. At the structural level, we analyze possibilities for modular verification of systems consisting of various components which interact. We illustrate these ideas by automatically verifying safety properties of a case study from the European Train Control System standard, which extends previous examples by comprising a complex track topology with lists of track segments and trains with different routes.

1 Introduction

Parametric real-time systems arise in a natural way in a wide range of applications, including controllers for systems of cars, trains, and planes. Since many such systems are safety-critical, there is great interest in methods for ensuring that they are safe. In order to verify such systems, one needs (i) suitable formalizations and (ii) efficient verification techniques. In this paper we analyze both aspects. Our main focus throughout the paper will be on reducing complexity by exploiting modularity at various levels: in the specification, in verification, and also structurally. The main contributions of the paper are:

(1) We exploit modularity at the specification level. In Section 2, we use the modular language CSP-OZ-DC (COD), which allows us to separately specify processes (as Communicating Sequential Processes, CSP), data (using Object-Z, OZ) and time (using the Duration Calculus, DC).
(2) We exploit modularity in verification (Sections 3 and 4).
   - First, we consider transition constraint systems (TCSs) that can be automatically obtained from the COD specification, and address verification tasks such as invariant checking. We show that for pointer data structures, we can obtain decision procedures for these verification tasks.

* This paper is an extended version of [8].

- Then we apply these methods to a detailed case study, additionally analyzing situations in which the use of COD specifications allows us to decouple verification tasks concerning data (OZ) from verification tasks concerning durations (DC). For systems with a parametric number of components, this allows us to impose (and verify) conditions on the single components which guarantee safety of the overall complex system.

(3) We also use modularity at a structural level. In Section 5, we use results from [24] to obtain possibilities for modular verification of systems with complex topologies by decomposing them into subsystems with simpler topologies.
(4) We describe a tool chain which translates a graphical UML version of the CSP-OZ-DC specification into TCSs, and automatically verifies the specification using our prover H-PILoT and other existing tools (Section 6).
(5) We illustrate the ideas on a running example taken from the European Train Control System standard (a system with a complex topology and a parametric number of components—modeled using pointer data structures and parametric constraints), and present a way of fully automatizing verification (for given safety invariants) using our tool chain.

The results were first presented in [8]. This paper extends [8] with a detailed description of the case study and full proofs.

Related work. Model-based development and verification of railway control systems with a complex track topology are analyzed in [10]. The systems are described in a domain-specific language and translated into SystemC code that is verified using bounded model checking. Neither verification of systems with a parametric number of components nor pointer structures are examined there.

In existing work on the verification of parametric systems often only few aspects of parametricity are studied together. [21] addresses the verification of temporal properties for hybrid systems (in particular also fragments of the ETCS as case study) but only supports parametricity in the data domain. [2] presents a method for the verification of a parametric number of timed automata with real-valued clocks, while in [5] only finite-state processes are considered. In [3], regular model checking for a parametric number of homogeneous linear processes and systems operating on queues or stacks is presented. There is also work on the analysis of safety properties for parametrized systems with an arbitrary number of processes operating on unbounded integer variables [1,7,16]. In contrast to ours, these methods sacrifice completeness by using either an over-approximation of the transition relation or abstractions of the state space. We, on the other hand, offer complete methods (based on decision procedures for data structures) for problems such as invariant checking and bounded model checking.

Motivating example. Consider a system of trains on a complex track topology as depicted in Fig. 1, and a radio block center (RBC) that has information about track segments and trains, like e.g. length, occupying train and allowed maximal speed for segments, and current position, segment and speed for trains. We identify situations in which safety of the system with complex track topology is

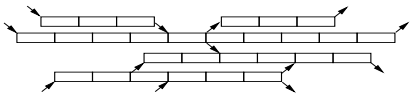


Fig. 1. Complex Track Topology

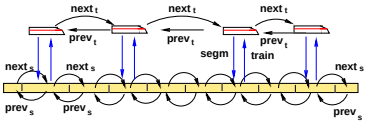


Fig. 2. Linear Track Topology

a consequence of safety of systems with linear track topology. Such modular verification possibilities allow us to consider the verification of a simplified version of this example, consisting of a *linear track* (representing a concrete route in the track topology), on which trains are allowed to enter or leave at given points. We model a general RBC controller for an area with a linear track topology and an arbitrary number of trains. For this, we use a theory of pointers with sorts t (for trains; next_t returns the next train on the track) and s (for segments; with next_s, prev_s describing the next/previous segment on the linear track). The link between trains and segments is described by appropriate functions train and segm (cf. Fig. 2). In addition, we integrate a simple timed train controller Train into the model. This allows us to certify that certain preconditions for the verification of the RBC are met by every train which satisfies the specification of Train, by reasoning on the timed and the untimed part of the system independently.

2 Modular specifications: CSP-OZ-DC

We start by presenting the specification language CSP-OZ-DC (COD) [12,11] which allows us to present in a modular way the control flow, data changes, and timing aspects of the systems we want to verify. We use *Communicating Sequential Processes* (CSP) to specify the control flow of a system using processes over events; *Object-Z* (OZ) for describing the state space and its change, and the *Duration Calculus* (DC) for modeling (dense) real-time constraints over durations of events. The operational semantics of COD is defined in [11] in terms of a timed automata model. For details on CSP-OZ-DC and its semantics, we refer to [12,11,9]. Our benefits from using COD are twofold:

- COD is compositional in the sense that it suffices to prove safety properties for the separate components to prove safety of the entire system [11]. This makes it possible to use different verification techniques for different parts of the specification, e.g. for control structure and timing properties.
- We benefit from high-level tool support given by Syspect[1], a UML editor for a dedicated UML profile [20] proposed to formally model real-time systems. It has a semantics in terms of COD. Thus, Syspect serves as an easy-to-use front-end to formal real-time specifications, with a graphical user interface.

2.1 Example: Systems of trains on linear tracks

To illustrate the ideas, we present some aspects of the case study mentioned in Section 1. We exploit the benefits of COD in (i) the specification of a complex

[1] `http://csd.informatik.uni-oldenburg.de/~syspect/`

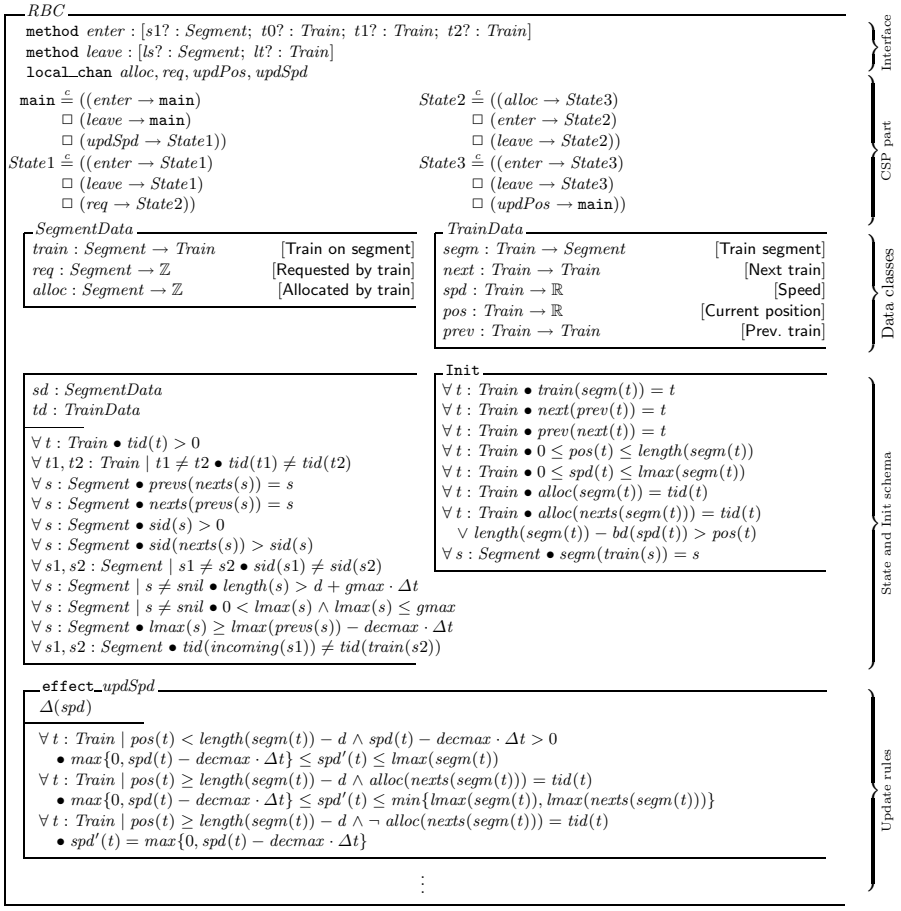


Fig. 3. Excerpt of the RBC controller as COD specification. The full specification can be found in the Appendix.

RBC controller; (ii) the specification of a controller for individual trains; and (iii) composing such specifications. Even though space does not allow us to present all the details, we present aspects of the example which cannot be considered with other formalisms, and show how to cope in a natural way with parametricity. More details of the case study are presented in Section 4.1 and the full case study model is given in the Appendices A and B. Figure 3 gives an exemplary overview of a COD specification and we explain its several parts in the following.

CSP part. The processes and their interdependency are specified using the CSP specification language. The RBC system passes repeatedly through four phases, modeled by events with corresponding COD schemata *updSpd* (*speed update*), *req* (*request update*), *alloc* (*allocation update*), and *updPos* (*position update*).

CSP:

$\texttt{main} \stackrel{c}{=} ((enter \rightarrow \texttt{main}) \;\Box\; (leave \rightarrow \texttt{main}) \;\Box\; (updSpd \rightarrow State1))$

$State1 \stackrel{c}{=} ((enter \rightarrow State1) \;\Box\; (leave \rightarrow State1) \;\Box\; (req \rightarrow State2))$

$State2 \stackrel{c}{=} ((alloc \rightarrow State3) \;\Box\; (enter \rightarrow State2) \;\Box\; (leave \rightarrow State2))$

$State3 \stackrel{c}{=} ((enter \rightarrow State3) \;\Box\; (leave \rightarrow State3) \;\Box\; (updPos \rightarrow \texttt{main}))$

The *speed update* models the fact that every train chooses its speed according to its knowledge about itself and its track segment as well as the next track segment. The *request update* models how trains send a request for permission to enter the next segment when they come close to the end of their current segment. The *allocation update* models how the RBC may either grant these requests by allocating track segments to trains that have made a request, or allocate segments to trains that are not currently on the route and want to enter. The *position update* models how all trains report their current positions to the RBC, which in turn de-allocates segments that have been left and gives movement authorities to the trains. Between any of these four updates, we can have trains *leaving* or *entering* the track at specific segments using the events *leave* and *enter*. The effects of these updates are defined in the OZ part.

OZ part. The OZ part of the specification consists of data classes, axioms, the `Init` schema, and update rules.

Data classes. The data classes declare function symbols that can change their values during runs of the system, and are used in the OZ part of the specification.

SegmentData

$train : Segment \rightarrow Train$	[Train on segment]
$req : Segment \rightarrow \mathbb{Z}$	[Requested by train]
$alloc : Segment \rightarrow \mathbb{Z}$	[Allocated by train]

TrainData

$segm : Train \rightarrow Segment$	[Train segment]
$next : Train \rightarrow Train$	[Next train]
$spd : Train \rightarrow \mathbb{R}$	[Speed]
$pos : Train \rightarrow \mathbb{R}$	[Current position]
$prev : Train \rightarrow Train$	[Prev. train]

Axioms. The axiomatic part defines properties of the data structures and system parameters which do not change during an execution of the system: $gmax : \mathbb{R}$ (the global maximum speed), $decmax : \mathbb{R}$ (the maximum deceleration of trains), $d : \mathbb{R}$ (a safety distance between trains), and $bd : \mathbb{R} \rightarrow \mathbb{R}$ (mapping the speed of a train to a safe approximation of the corresponding braking distance). We specify properties of those parameters, among which an important one is $d \geq bd(gmax) + gmax \cdot \Delta t$ stating that the safety distance d to the end of the segment is greater than the braking distance of a train at maximal speed $gmax$ plus a further safety margin (distance for driving Δt time units at speed $gmax$). Furthermore, unique, non-negative ids for trains (sort *Train*) and track segments (sort *Segment*) are defined. The route is modeled as a doubly-linked list[2] of track segments, where every segment has additional properties specified by the constraints in the state schema.

E.g., *sid* is increasing along the *nexts* pointer, the *length* of a segment is bounded from below in terms of d and $gmax$, and the difference between local maximal speeds on neighboring segments is bounded by $decmax$. Finally, we have a function *incoming*; its value $incoming(s)$ for a track segment s is either a train

[2] Note that we use relatively loose axiomatizations of the list structures for both trains and segments, also allowing for disjoint families of linear, possibly infinite lists.

which wants to enter the segment s from outside the current route, or *tnil* if there is no such train. Although the values of *incoming* can change during an execution, we consider the constraint (*) (the last of the axioms on the right) as a property of our environment that always holds. Apart from that, *incoming* may change arbitrarily and is not explicitly updated. Note that *Train* and *Segment* are pointer sorts with a special null element (*tnil* and *snil*, respectively), and all constraints implicitly only hold for non-null elements. So, constraint (*) actually means

$$\forall\, s1, s2 : \mathit{Segment} \mid s1 \neq \mathit{snil} \neq s2 \wedge \mathit{incoming}(s1) \neq \mathit{tnil} \wedge \mathit{train}(s2) \neq \mathit{tnil} \bullet \mathit{tid}(\mathit{incoming}(s1)) \neq \mathit{tid}(\mathit{train}(s2))$$

$\forall\, t : \mathit{Train} \bullet \mathit{tid}(t) > 0$
$\forall\, t1, t2 : \mathit{Train} \mid t1 \neq t2 \bullet \mathit{tid}(t1) \neq \mathit{tid}(t2)$
$\forall\, s : \mathit{Segment} \bullet \mathit{prevs}(\mathit{nexts}(s)) = s$
$\forall\, s : \mathit{Segment} \bullet \mathit{nexts}(\mathit{prevs}(s)) = s$
$\forall\, s : \mathit{Segment} \bullet \mathit{sid}(s) > 0$
$\forall\, s : \mathit{Segment} \bullet \mathit{sid}(\mathit{nexts}(s)) > \mathit{sid}(s)$
$\forall\, s1, s2 : \mathit{Segment} \mid s1 \neq s2 \bullet \mathit{sid}(s1) \neq \mathit{sid}(s2)$
$\forall\, s : \mathit{Segment} \mid s \neq \mathit{snil} \bullet \mathit{length}(s) > d + \mathit{gmax} \cdot \Delta t$
$\forall\, s : \mathit{Segment} \mid s \neq \mathit{snil} \bullet 0 < \mathit{lmax}(s) \wedge \mathit{lmax}(s) \leq \mathit{gmax}$
$\forall\, s : \mathit{Segment} \bullet \mathit{lmax}(s) \geq \mathit{lmax}(\mathit{prevs}(s)) - \mathit{decmax} \cdot \Delta t$
$\forall\, s1, s2 : \mathit{Segment} \bullet \mathit{tid}(\mathit{incoming}(s1)) \neq \mathit{tid}(\mathit{train}(s2))$ (*)

Init schema. The Init *schema* describes the initial state of the system. It essentially states that trains are arranged in a doubly-linked list, that all trains are initially placed correctly on the track segments and that all trains respect their speed limits.

Init
$\forall\, t : \mathit{Train} \bullet \mathit{train}(\mathit{segm}(t)) = t$
$\forall\, t : \mathit{Train} \bullet \mathit{next}(\mathit{prev}(t)) = t$
$\forall\, t : \mathit{Train} \bullet \mathit{prev}(\mathit{next}(t)) = t$
$\forall\, t : \mathit{Train} \bullet 0 \leq \mathit{pos}(t) \leq \mathit{length}(\mathit{segm}(t))$
$\forall\, t : \mathit{Train} \bullet 0 \leq \mathit{spd}(t) \leq \mathit{lmax}(\mathit{segm}(t))$
$\forall\, t : \mathit{Train} \bullet \mathit{alloc}(\mathit{segm}(t)) = \mathit{tid}(t)$
$\forall\, t : \mathit{Train} \bullet \mathit{alloc}(\mathit{nexts}(\mathit{segm}(t))) = \mathit{tid}(t)$
$\quad \vee\ \mathit{length}(\mathit{segm}(t)) - \mathit{bd}(\mathit{spd}(t)) > \mathit{pos}(t)$
$\forall\, s : \mathit{Segment} \bullet \mathit{segm}(\mathit{train}(s)) = s$

Update rules. Updates of the state space, that are executed when the corresponding event from the CSP part is performed, are specified with *effect schemata*. The schema for *updSpd*, for instance, consists of three rules, distinguishing (i) trains whose distance to the end of the segment is greater than the safety distance d (the first two lines of the constraint), (ii) trains that are beyond the safety distance near the end of the segment, and for which the next segment is allocated, and (iii) trains that are near the end of the segment without an allocation. In case (i), the train can choose an arbitrary speed below the maximal speed of the current segment. In case (ii), the train needs to brake if the speed limit of the next segment is below the current limit. In case (iii), the train needs to brake so that it safely stops before reaching the end of the segment.

effect_*updSpd*
$\Delta(\mathit{spd})$

$\forall\, t : \mathit{Train} \mid \mathit{pos}(t) < \mathit{length}(\mathit{segm}(t)) - d \wedge \mathit{spd}(t) - \mathit{decmax} \cdot \Delta t > 0$
$\quad \bullet\ \mathit{max}\{0, \mathit{spd}(t) - \mathit{decmax} \cdot \Delta t\} \leq \mathit{spd}'(t) \leq \mathit{lmax}(\mathit{segm}(t))$
$\forall\, t : \mathit{Train} \mid \mathit{pos}(t) \geq \mathit{length}(\mathit{segm}(t)) - d \wedge \mathit{alloc}(\mathit{nexts}(\mathit{segm}(t))) = \mathit{tid}(t)$
$\quad \bullet\ \mathit{max}\{0, \mathit{spd}(t) - \mathit{decmax} \cdot \Delta t\} \leq \mathit{spd}'(t) \leq \mathit{min}\{\mathit{lmax}(\mathit{segm}(t)), \mathit{lmax}(\mathit{nexts}(\mathit{segm}(t)))\}$
$\forall\, t : \mathit{Train} \mid \mathit{pos}(t) \geq \mathit{length}(\mathit{segm}(t)) - d \wedge \neg\, \mathit{alloc}(\mathit{nexts}(\mathit{segm}(t))) = \mathit{tid}(t)$
$\quad \bullet\ \mathit{spd}'(t) = \mathit{max}\{0, \mathit{spd}(t) - \mathit{decmax} \cdot \Delta t\}$

Timed train controller. In the DC part of a specification, real-time constraints are specified: A second, timed controller Train (for one train only) interacts with the RBC controller, which is presented in the overview of the case study in Figure 4. The train controller Train consists of three timed components running

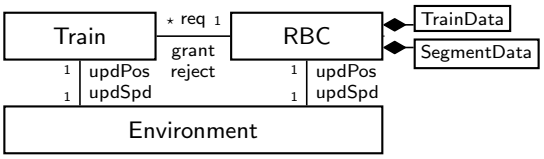


Fig. 4. Structural overview

in parallel. The first updates the train's position. This component contains e.g. the DC formula

$$\neg(true \frown \updownarrow updPos \frown (\ell < \Delta t) \frown \updownarrow updPos \frown true),$$

that specifies a lower time bound Δt on *updPos* events. The second component checks periodically whether the train is beyond the safety distance to the end of the segment. Then, it starts braking within a short reaction time. The third component requests an extension of the movement authority from the RBC, which may be granted or rejected. The train controller is explained in more detail in Section 4.2.

3 Modular verification

In this section, we introduce an invariant checking approach for the verification of safety properties of COD specifications, and present decidability results for local theory extensions that imply decidability of the invariant checking problem for a large class of parameterized systems.

Formally, our approach works on a *transition constraint system* (TCS) obtained from the COD specification by an automatic translation (see [9]) which is guaranteed to capture the defined semantics of COD (as defined in [11]).

Definition 1. *The tuple* $T = (V, \Sigma, (\mathsf{Init}), (\mathsf{Update}))$ *is a* transition constraint system*, which specifies: the variables* (V) *and function symbols* (Σ) *whose values may change over time; a formula* (Init) *specifying the properties of initial states; and a formula* (Update) *which specifies the transition relation in terms of the values of variables* $x \in V$ *and function symbols* $f \in \Sigma$ *before a transition and their values (denoted* x', f'*) after the transition.*

In addition to the TCS, we obtain a *background theory* $\mathcal{T}$ from the specification, describing properties of the used data structures and system parameters that do not change over time. Typically, $\mathcal{T}$ consists of a family of standard theories (like the theory of real numbers), axiomatizations for data structures, and constraints on system parameters. In what follows $\phi \models_{\mathcal{T}} \psi$ denotes logical entailment and means that every model of the theory $\mathcal{T}$ which is a model of ϕ is also a model for ψ. We denote false by $\bot$, so $\phi \models_{\mathcal{T}} \bot$ means that ϕ is unsatisfiable w.r.t. $\mathcal{T}$.

3.1 Verification problems

We consider the problem of *invariant checking* of safety properties.[3] To show that a safety property, represented as a formula (Safe), is an invariant of a TCS T (for a given background theory $\mathcal{T}$), we need to identify an *inductive invariant* (Inv) which strengthens (Safe), i.e., we need to prove that

(1) $(\mathsf{Inv}) \models_{\mathcal{T}} (\mathsf{Safe})$,
(2) $(\mathsf{Init}) \models_{\mathcal{T}} (\mathsf{Inv})$, and
(3) $(\mathsf{Inv}) \wedge (\mathsf{Update}) \models_{\mathcal{T}} (\mathsf{Inv}')$, where (Inv') results from (Inv) by replacing each $x \in V$ by x' and each $f \in \Sigma$ by f'.

Lemma 2. *If* (Safe), (Inv), (Init) *and* (Update) *belong to a class of formulae for which the entailment problems w.r.t.* $\mathcal{T}$ *above are decidable then the problem of checking that* (Inv) *is an invariant of* T *which strengthens* (Safe) *is decidable.*

We use this result in a verification-design loop as follows: We start from a specification written in COD. We use a translation to TCS and check whether a certain formula (Inv) (usually a safety property) is an inductive invariant.
(i) If invariance can be proved, safety of the system is guaranteed.
(ii) If invariance cannot be proved, we have the following possibilities:

1. Use a specialized prover to construct a counterexample (model in which the property (Inv) is not an invariant) which can be used to find errors in the specification and/or to strengthen the invariant[4].
2. Using results in [25] we can often derive additional (weakest) constraints on the parameters which guarantee that Inv is an invariant.

Of course, the decidability results for the theories used in the description of a system can be also used for checking consistency of the specification.

If a TCS models a system with a parametric number of components, the formulae in problems (1)–(3) may contain universal quantifiers (to describe properties of all components), hence standard SMT methods – which are only complete for ground formulae – do not yield decision procedures. In particular, in cases (ii)(1–2) and for consistency checks we need possibilities of reliably detecting satisfiability of sets of universally quantified formulae for which standard SMT solvers cannot be used. We now present situations in which this is possible.

3.2 Modularity in automated reasoning: Decision procedures

We identify classes of theories for which invariant checking (and bounded model checking) is decidable. Let $\mathcal{T}_0$ be a theory with signature $\Pi = (S_0, \Sigma_0, \mathsf{Pred})$, where S_0 is a set of sorts, and Σ_0 and Pred are sets of function resp. predicate symbols. We consider extensions of $\mathcal{T}_0$ with new function symbols in a set Σ, whose properties are axiomatized by a set $\mathcal{K}$ of clauses.

[3] We can address bounded model checking problems in a similar way, cf. [15,9,13].
[4] This last step is the only part which is not fully automated. For future work we plan to investigate possibilities of automated invariant generation or strengthening.

Local theory extensions. We are interested in theory extensions in which for every set G of ground clauses we can effectively determine a finite (preferably small) set of instances of the axioms $\mathcal{K}$ sufficient for checking satisfiability of G without loss of completeness. If G is a set of Π^c-clauses (where Π^c is the extension of Π with constants in a set Σ_c), we denote by $\mathsf{st}(\mathcal{K}, G)$ the set of ground terms starting with a Σ-function symbol occurring in $\mathcal{K}$ or G, and by $\mathcal{K}[G]$ the set of instances of $\mathcal{K}$ in which the terms starting with Σ-functions are in $\mathsf{st}(\mathcal{K}, G)$. $\mathcal{T}_0 \cup \mathcal{K}$ is a *local extension* of $\mathcal{T}_0$ [23] if the following condition holds:

(Loc) For every set G of ground clauses, $G \models_{\mathcal{T}_0 \cup \mathcal{K}} \bot$ iff $\mathcal{K}[G] \cup G \models_{\mathcal{T}_0^\Sigma} \bot$

where $\mathcal{T}_0^\Sigma$ is the extension of $\mathcal{T}_0$ with the free functions in Σ. We can define *stable locality* (SLoc) in which we use the set $\mathcal{K}^{[G]}$ of instances of $\mathcal{K}$ in which the variables below Σ-functions are instantiated with terms in $\mathsf{st}(\mathcal{K}, G)$. In local theory extensions, sound and complete hierarchical reasoning is possible.

Theorem 3 ([23]). *With the notations introduced above, if $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}$ satisfies condition* ((S)Loc) *then the following are equivalent to $G \models_{\mathcal{T}_0 \cup \mathcal{K}} \bot$:*

(1) $\mathcal{K}*[G] \cup G \models_{\mathcal{T}_0^\Sigma} \bot$ ($\mathcal{K}*[G]$ is $\mathcal{K}[G]$ for local; $\mathcal{K}^{[G]}$ for stably local extensions).
(2) $\mathcal{K}_0 \cup G_0 \cup D \models_{\mathcal{T}_0^\Sigma} \bot$, where $\mathcal{K}_0 \cup G_0 \cup D$ is obtained from $\mathcal{K}*[G] \cup G$ by introducing (bottom-up) new constants c_t for subterms $t = f(g_1, \ldots, g_n)$ with $f \in \Sigma$, g_i ground $\Sigma_0 \cup \Sigma_c$-terms; replacing the terms with the corresponding constants; and adding the definitions $c_t \approx t$ to the set D.
(3) $\mathcal{K}_0 \cup G_0 \cup N_0 \models_{\mathcal{T}_0} \bot$, where

$$N_0 = \{ \bigwedge_{i=1}^{n} c_i \approx d_i \rightarrow c = d \mid f(c_1, \ldots, c_n) \approx c, f(d_1, \ldots, d_n) \approx d \in D \}.$$

The hierarchical reduction method is implemented in the system H-PILoT [14].

Corollary 4 ([23]). *If the theory extension $\mathcal{T}_0 \subseteq \mathcal{T}_1 = \mathcal{T}_0 \cup \mathcal{K}$ satisfies* ((S)Loc). *then satisfiability of sets of ground clauses G w.r.t. $\mathcal{T}_1$ is decidable if $\mathcal{K}*[G]$ is finite and $\mathcal{K}_0 \cup G_0 \cup N_0$ belongs to a decidable fragment $\mathcal{F}$ of $\mathcal{T}_0$. Since the size of $\mathcal{K}_0 \cup G_0 \cup N_0$ is polynomial in the size of G (for a given $\mathcal{K}$), locality allows us to express the complexity of the ground satisfiability problem w.r.t. $\mathcal{T}_1$ as a function of the complexity of the satisfiability of $\mathcal{F}$-formulae w.r.t. $\mathcal{T}_0$.*

3.3 Examples of local theory extensions

We are interested in reasoning efficiently about data structures and about updates of data structures. We here give examples of such theories.

Update axioms. In [13] we show that for any TCS with background theory $\mathcal{T}$ with signature $\Pi = (S_0, \Sigma_0 \cup \Sigma, \mathsf{Pred})$ (possibly also containing axiomatizations of the properties of the functions in Σ), all update rules $\mathsf{Update}(\Sigma, \Sigma')$ which describe how the values of the Σ-functions change, depending on a set

$\{\phi_i \mid i \in I\}$ of mutually exclusive conditions, i.e. update rules described by conjunctions of formulae with mutually exclusive guards $\bigwedge_{f' \in \Sigma'} \mathsf{Def}_{f'}$ define local theory extensions – where $\mathsf{Def}_{f'}$ is the conjunction of formulae of the form:

$$\phi_1(\overline{x}) \rightarrow s_1(\overline{x}) \leq f'(x) \leq t_1(\overline{x})$$
$$\ldots$$
$$\phi_n(\overline{x}) \rightarrow s_n(\overline{x}) \leq f'(x) \leq t_n(\overline{x})$$

with s_i, t_i are $\Sigma_0 \cup \Sigma$-terms, and ϕ_i are Π-formulae with the property that $\phi_i \wedge \phi_j \models_{\mathcal{T}} \bot$ for $i \neq j$ and $\phi_i \models_{\mathcal{T}} s_i \leq t_i$.

This locality property follows as a consequence of the following result:

Theorem 5 ([13]). *Let $\mathcal{T}_0$ be a base theory with signature $\Pi_0 = (S_0, \Sigma_0, \mathsf{Pred})$. Assume that:*

(i) $\{\phi_i \mid i \in I\}$ are formulae over the base signature Π_0 such that $\phi_i(\overline{x}) \wedge \phi_j(\overline{x}) \models_{\mathcal{T}_0} \bot$ for $i \neq j$, and
(ii) s_i, t_i are (possibly equal) Σ_0-terms such that $\mathcal{T}_0 \models \forall \overline{x}(\phi_i(\overline{x}) \rightarrow s_i(\overline{x}) \leq t_i(\overline{x}))$ for all $i \in I$.

Let Σ' be a family of function symbols which is disjoint from Σ_0. Then the extension of $\mathcal{T}_0$ with axioms of the form $\bigwedge_{f \in \Sigma'} \mathsf{Def}_f$ is local, where:

$$\mathsf{Def}_f \qquad \bigwedge_{i \in I} (\forall \overline{x}(\phi_i(\overline{x}) \rightarrow s_i(\overline{x}) \leq f(\overline{x}) \leq t_i(\overline{x}))).$$

Data structures. Numerous locality results for data structures exist, e.g. for fragments of the theories of arrays [6,13], and pointers [18,13]. As an illustration – since the model we used in the running example involves a theory of linked data structures – we now present a slight extension of the fragment of the theory of pointers studied in [18,13], which is useful for modeling the track topologies and successions of trains on these tracks. We consider a set of pointer sorts $\mathsf{P} = \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}$ and a scalar sort s.[5] Let $(\Sigma_s, \mathsf{Pred}_s)$ be a scalar signature, and let Σ_P be a set of function symbols with arguments of pointer sort consisting of sets $\Sigma_{\overline{\mathsf{p}} \rightarrow \mathsf{s}}$ (the family of functions of arity $\overline{\mathsf{p}} \rightarrow \mathsf{s}$), and $\Sigma_{\overline{\mathsf{p}} \rightarrow \mathsf{p}}$ (the family of functions of arity $\overline{\mathsf{p}} \rightarrow \mathsf{p}_i$). (Here $\overline{\mathsf{p}}$ is a tuple $\mathsf{p}_{i_1} \ldots \mathsf{p}_{i_k}$ with $k \geq 0$.) We assume that for every pointer sort $\mathsf{p} \in \mathsf{P}$, Σ_P contains a constant null_p of sort p.

Example 6. The fact that we also allow scalar fields with more than one argument is very useful because it allows, for instance, to model certain relationships between different nodes. Examples of such scalar fields could be:

- $\mathsf{distance}(p, q)$ associates with non-null p, q of pointer type a real number;
- $\mathsf{reachable}(p, q)$ associates with non-null p, q of pointer type a boolean value (true (1) if q is reachable from p using the next functions, false (0) otherwise).

Let $\Sigma = \Sigma_P \cup \Sigma_s$. In addition to allowing several pointer types and functions of arbitrary arity, we loosen some of the restrictions imposed in [18,13].

[5] We assume that we only have one scalar sort for simplicity of presentation; the scalar theory can itself be an extension or combination of theories.

Definition 7. *An* extended pointer clause *is a formula of form* $\forall\, \bar{p}.\ (\mathcal{E} \vee \varphi)(\bar{p})$, *where* $\bar{p}$ *is a set of pointer variables including all free variables of* $\mathcal{E}$ *and* φ, *and:*

(1) $\mathcal{E}$ *consists of disjunctions of pointer equalities, and has the property that for every term* $t = f(t_1, \ldots, t_k)$ *with* $f \in \Sigma_P$ *occurring in* $\mathcal{E} \vee \varphi$, $\mathcal{E}$ *contains an atom of the form* $t' = \mathsf{null_p}$ *for every proper subterm (of sort* p*)* t' *of* t*;*
(2) φ *is an* arbitrary *formula of sort* s.

$\mathcal{E}$ *and* φ *may additionally contain free scalar and pointer constants, and* φ *may contain additional quantified variables of sort* s.

Theorem 8. *Let* $\Sigma = \Sigma_P \cup \Sigma_s$ *be a signature as defined before. Let* $\mathcal{T}_s$ *be a theory of scalars with signature* Σ_s. *Let* Φ *be a set of* Σ*-extended pointer clauses. Then, for every set* G *of ground clauses over an extension* Σ_c *of* Σ *with constants in a countable set* c *the following are equivalent:*

(1) G *is unsatisfiable w.r.t.* $\Phi \cup \mathcal{T}_s$*;*
(2) $\Phi^{[G]} \cup G$ *is an unsatisfiable set of clauses in the disjoint combination* $\mathcal{T}_s \cup \mathcal{E}\mathcal{Q}_P$ *of* $\mathcal{T}_s$ *and* $\mathcal{E}\mathcal{Q}_P$*, the many-sorted theory of pure equality over pointer sorts,*

where $\Phi^{[G]}$ *consists of all instances of* Φ *in which the universally quantified variables of pointer type occurring in* Φ *are replaced by ground terms of pointer type in the set* $\mathsf{st}(\Phi, G)$ *of all ground terms of sort* p *occurring in* Φ *or in* G.

The proof is similar to that in [13]. H-PILoT can be used as a decision procedure for this theory of pointers – if the theory of scalars is decidable – and for any extension of this theory with function updates in the fragment in Theorem 5.

Example 9. Let $P = \{\mathsf{s}(egment), \mathsf{t}(rain)\}$, and let $\mathsf{next_t}, \mathsf{prev_t} : \mathsf{t} \to \mathsf{t}$, $\mathsf{next_s}, \mathsf{prev_s} : \mathsf{s} \to \mathsf{s}$, and $\mathsf{train} : \mathsf{s} \to \mathsf{t}$, $\mathsf{segm} : \mathsf{t} \to \mathsf{s}$, and functions of scalar sort as listed at the beginning of Section 2.1. All axioms describing the background theory and the initial state in Section 2.1 are expressed by extended pointer clauses.

The following formula expressing a property of reachability of trains can be expressed as a pointer clause:

$$\forall\, p, q (p \neq \mathsf{null}_t \wedge q \neq \mathsf{null}_t \wedge \mathsf{next}_t(q) \neq \mathsf{null}_t \to (\mathsf{reachable}(p, q) \to \mathsf{reachable}(p, \mathsf{next}_t(q)))).$$

Decidability for verification. A direct consequence of Theorem 3 and Corollary 4 is the following decidability result for invariant checking:

Corollary 10 ([25]). *Let* T *be the transition constraint system and* $\mathcal{T}$ *be the background theory associated with a specification. If the update rules* Update *and the invariant property* Inv *can be expressed as sets of clauses which define a chain of local theory extensions* $\mathcal{T} \subseteq \mathcal{T} \cup \mathsf{Inv}(\overline{x}, \overline{f}) \subseteq \mathcal{T} \cup \mathsf{Inv}(\overline{x}, \overline{f}) \cup \mathsf{Update}(\overline{x}, \overline{x}', \overline{f}, \overline{f}')$ *then checking whether a formula is an invariant is decidable.*

In this case we can use H-PILoT as a decision procedure (and also to construct a model in which the property Inv is not an invariant). We can also use results from [25] to derive additional (weakest) constraints on the parameters which guarantee that Inv is an invariant.

4 Verification of the case study

Now, we introduce in more detail the example from Section 1, which is a more complex version of the RBC verification case study than those presented in previous work [15,9]. We demonstrate how it can be verified by using a combination of the invariant checking approach presented in Section 3.1 for the RBC specification and a model-checking approach for timing properties (introduced in [19]) for the COD specification for a single train Train. This combination is necessary because we want to prove safety of discrete updates with a parametric number of trains for the RBC component, and real-time safety properties for the train controller Train. Among other things, the specification of the RBC assumes that the train controllers always react in time to make the train brake before reaching a critical position.

Using the modularity of COD, we can separately use the invariant checking approach to verify the RBC for a parametric number of trains, and the approach for model-checking DC formulae to verify that every train satisfies the timing assumptions made in the RBC specification.

Moreover, in the new case study we model a complex track topology, consisting of an arbitrary number of interconnected track segments (see Figure 1), which can be occupied by trains. In this Section, we prove safety for an arbitrary route in such a topology, including the possibility of trains leaving or entering the route. In Section 5 we show that every complex track topology without cycles and with the property that at every crossing point only two paths come together can be decomposed into a family of linear track systems such that the safety of the whole system follows from the safety of the linear track systems.

4.1 Verification of the RBC

In the following we describe in detail the example from Section 1, and show how we can verify safety properties of the system.

Background theory. We model the route as a doubly-linked list of track segments (sort s), and trains on this route as a doubly-linked list of train objects (sort t). Both sorts contain a special null element (null_s and null_t, respectively). The following constant and function symbols are used to model properties of track segments and trains which do not change during an execution of the system:

- $gmax : \mathbb{R}$ (the global maximum speed),
- $maxDecl : \mathbb{R}$ (the maximum deceleration of trains),
- $\Delta t : \mathbb{R}$ (the time between position updates),
- $d : \mathbb{R}$ (a safety distance between trains),
- $\mathsf{next}_\mathsf{s} : \mathsf{s} \to \mathsf{s}$ (pointer to the next segment),
- $\mathsf{prev}_\mathsf{s} : \mathsf{s} \to \mathsf{s}$ (pointer to the previous segment),
- $\mathsf{id}_\mathsf{s} : \mathsf{s} \to \mathbb{N}$ (a unique identifier, increasing along next_s),
- $\mathsf{max} : \mathsf{s} \to \mathbb{R}$ (the maximum allowed speed on the segment),
- $\mathsf{length} : \mathsf{s} \to \mathbb{R}$ (the length of the segment),

- $\mathsf{id_t} : \mathsf{t} \to \mathbb{N}$ (a unique identifier for each train),
- $\mathsf{prio} : \mathbb{N} \to \mathbb{N}$ (mapping train ids to unique priorities),
- and $\mathsf{bd} : \mathbb{R} \to \mathbb{R}$ (mapping the speed of a train to a safe approximation of the corresponding braking distance).

On these symbols, we impose the following axioms:

$$
\begin{aligned}
(\mathsf{Def_{next_s}}) &:= \forall\, s : && s \neq \mathsf{null_s} \ \wedge\ \mathsf{prev_s}(s) \neq \mathsf{null_s} \to \mathsf{next_s}(\mathsf{prev_s}(s)) = s \\
(\mathsf{Def_{prev_s}}) &:= \forall\, s : && s \neq \mathsf{null_s} \ \wedge\ \mathsf{next_s}(s) \neq \mathsf{null_s} \to \mathsf{prev_s}(\mathsf{next_s}(s)) = s \\
(\mathsf{Def_{id_s}}) &:= \forall\, s : && s \neq \mathsf{null_s} \ \wedge\ \mathsf{next_s}(s) \neq \mathsf{null_s} \to \mathsf{id_s}(s) < \mathsf{id_s}(\mathsf{next_s}(s)) \\
& && \mathsf{id_s}(\mathsf{null_s}) = 0 \\
(\mathsf{Def_{max}}) &:= \forall\, s : && s \neq \mathsf{null_s} \ \wedge\ 0 < \mathsf{max}(s) \leq gmax \\
& \forall\, s : && s \neq \mathsf{null_s} \ \wedge\ \mathsf{next_s}(s) \neq \mathsf{null_s} \\
& && \to \mathsf{max}(\mathsf{next_s}(s)) \geq \mathsf{max}(s) - maxDecl \\
(\mathsf{Def_{length}}) &:= \forall\, s : && s \neq \mathsf{null_s} \ \wedge\ \mathsf{length}(s) > d + gmax \cdot \Delta t \\
(\mathsf{Def_{id_t}}) &:= \forall\, t_1, t_2 : && t_1 \neq \mathsf{null_t} \neq t_2 \ \wedge\ t_1 \neq t_2 \to \mathsf{id_t}(t_1) \neq \mathsf{id_t}(t_2) \\
& && \mathsf{id_t}(\mathsf{null_t}) = 0 \\
(\mathsf{Def_{prio}}) &:= \forall\, x_1, x_2 : x_1 \neq x_2 \to \mathsf{prio}(x_1) \neq \mathsf{prio}(x_2)
\end{aligned}
$$

$$
\begin{aligned}
(\mathsf{Def_{bd}}) &:= \forall\, x : bd(x) = x^2/(2 \cdot maxDecl) \\
(\mathsf{Def_{d}}) &:= \quad d > \mathsf{bd}(gmax) + gmax \cdot \Delta t
\end{aligned}
$$

Additionally, we have a function $\mathsf{incoming} : \mathsf{s} \to \mathsf{t}$, the value of which is either a train which wants to enter the given segment from outside the current route, or $\mathsf{null_t}$ if there is no such train. Although the valuation of incoming can change during an execution, we consider it as a property of our environment and will not explicitly update it. Instead, we assume that the following always holds:

$$
\begin{aligned}
(\mathsf{Def_{incoming}}) := \forall\, s_1, s_2 : {} & s_1 \neq \mathsf{null_s} \neq s_2 \ \wedge\ \mathsf{incoming}(s_1) \neq \mathsf{null_t} \ \wedge\ train(s_2) \neq \mathsf{null_t} \\
& \to \mathsf{id_t}(\mathsf{incoming}(s_1)) \neq \mathsf{id_t}(\mathsf{train}(s_2)).
\end{aligned}
$$

Our background theory $\mathcal{T}$ then consists of $\mathbb{R}$ and $\mathbb{N}$, extended with the constant and function symbols above, as well as the given axioms.

System variables and function symbols. The set of system variables V consists only of $\mathsf{pc} : \mathbb{N}$, a program counter that is introduced in the translation process from COD to TCS (it takes values between 1 and 4, corresponding to Figure 5. The set Σ of function symbols which may change their valuation over time consists of:

- $\mathsf{train} : \mathsf{s} \to \mathsf{t}$ (pointer to the train which occupies the track segment; $\mathsf{null_t}$ if segment is not occupied)
- $\mathsf{alloc} : \mathsf{s} \to \mathbb{N}$ (either 0 or the $\mathsf{id_t}$ of a train which is allowed to occupy the segment),
- $\mathsf{req} : \mathsf{s} \to \mathbb{N}$ (either 0 or the $\mathsf{id_t}$ of a train on the previous segment which has requested to enter this segment),

- $\mathsf{incoming} : \mathsf{s} \to \mathsf{s}$ (either $\mathsf{null_t}$ or a train which wants to enter this track segment from outside the current route)
- $\mathsf{next_t} : \mathsf{t} \to \mathsf{t}$ (pointer to the next train, $\mathsf{null_t}$ if there is no next),
- $\mathsf{prev_t} : \mathsf{t} \to \mathsf{t}$ (pointer to the previous train, $\mathsf{null_t}$ if there is no previous),
- $\mathsf{segm} : \mathsf{t} \to \mathsf{s}$ (pointer to the track segment of the train, $\mathsf{null_s}$ if train is not on current route),
- $\mathsf{pos} : \mathsf{t} \to \mathbb{R}$ (position of the train relative to its track segment), and
- $\mathsf{spd} : \mathsf{t} \to \mathbb{R}$ (current speed of the train).

System behavior. As shown in Figure 5, the RBC system passes repeatedly through four phases, modeled by the sets of update rules (Speed), (Request), (Alloc) and (Position). The *speed update* models the fact that every train chooses its speed according to its knowledge about itself and its track segment as well as the next track segment. The *request update* models how trains send a request for permission to enter the next segment when they come close to the end of their current segment. The *allocation update* models how the RBC may either grant these requests by allocating track segments to trains that have made a request, or allocate segments to trains that are not currently on the route and want to enter. The *position update* models how (after Δt time units) all trains report their current positions to the RBC, which in turn de-allocates segments that have been left and gives movement authorities to the trains.

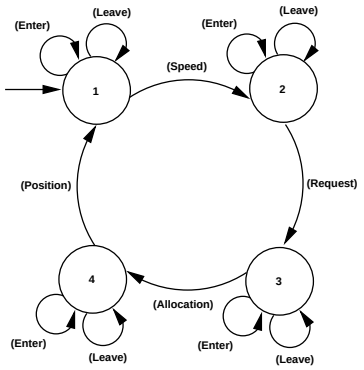


Fig. 5. Transition Constraint System for the RBC Case Study

Between any of these four updates, we can have trains *leaving* or *entering* the track at specific segments. This is modeled in the sets of update rules (Enter) and (Leave).

In the following, we give the initial state and a part of the update rules of the system.

Initial state. Axioms describing the initial state (and later on, the invariants) are *extended pointer clauses* (cf. Section 3.3, Definition 7) and implicitly contain the required definedness constraints $t' = \mathsf{null_p}$. We define (Init) as the conjunction of these constraints:

$$(\mathsf{Init_{pc}}) := \quad \mathsf{pc} = 1 \qquad (\mathsf{Init_{segm}}) := \forall\, s : \mathsf{segm}(\mathsf{train}(s)) = s$$

$$(\mathsf{Init_{next_t}}) := \forall\, t : \mathsf{next_t}(\mathsf{prev_t}(t)) = t \qquad (\mathsf{Init_{train}}) := \forall\, t : \mathsf{train}(\mathsf{segm}(t)) = t$$

$$(\mathsf{Init_{prev_t}}) := \forall\, t : \mathsf{prev_t}(\mathsf{next_t}(t)) = t \qquad (\mathsf{Init_{pos}}) := \forall\, t : 0 \le \mathsf{pos}(t) \le \mathsf{length}(\mathsf{segm}(t))$$

$$(\mathsf{Init_{alloc}}) := \forall\, t : \mathsf{alloc}(\mathsf{segm}(t)) = \mathsf{id_t}(t) \quad (\mathsf{Init_{spd}}) := \forall\, t : 0 \le \mathsf{spd}(t) \le \mathsf{max}(\mathsf{segm}(t))$$

$$(\mathsf{Init_{bd}}) := \forall\, t : \mathsf{pc} = 1 \ \wedge\ \mathsf{alloc}(\mathsf{next_s}(\mathsf{segm}(t))) \neq \mathsf{id_t}(t) \\ \rightarrow \mathsf{length}(\mathsf{segm}(t)) - \mathsf{bd}(\mathsf{spd}(t)) > \mathsf{pos}(t)$$

System updates. We define the transition relation of the system as

$$(\mathsf{Update}) = (\mathsf{Speed}) \vee (\mathsf{Request}) \vee (\mathsf{Alloc}) \vee (\mathsf{Position}) \vee (\mathsf{Enter}) \vee (\mathsf{Leave}),$$

$$
\begin{array}{l}
\quad \mathsf{pc} = 1 \ \wedge \ \mathsf{pc}' = 2 \\
\forall\, t : \mathsf{pos}(t) < \mathsf{length}(\mathsf{segm}(t)) - d \\
\quad \rightarrow max\{0,\ \mathsf{spd}(t) - maxDecl \cdot \Delta t\} \leq \mathsf{spd}'(t) \leq \mathsf{max}(\mathsf{segm}(t)) \\
\forall\, t : \mathsf{pos}(t) \geq \mathsf{length}(\mathsf{segm}(t)) - d \ \wedge \ \mathsf{alloc}(\mathsf{next}_s(\mathsf{segm}(t))) = \mathsf{id}_\mathsf{t}(t) \\
\quad \rightarrow max\{0,\ \mathsf{spd}(t) - maxDecl \cdot \Delta t\} \leq \mathsf{spd}'(t) \\
\qquad \wedge \ \mathsf{spd}'(t) \leq min\{\mathsf{max}(\mathsf{segm}(t)), \mathsf{max}(\mathsf{next}_\mathsf{s}(\mathsf{segm}(t)))\} \\
\forall\, t : \mathsf{pos}(t) \geq \mathsf{length}(\mathsf{segm}(t)) - d \ \wedge \ \mathsf{alloc}(\mathsf{next}_\mathsf{s}(\mathsf{segm}(t))) \neq \mathsf{id}_\mathsf{t}(t) \\
\quad \rightarrow \mathsf{spd}'(t) = max\{0,\ \mathsf{spd}(t) - maxDecl \cdot \Delta t\}
\end{array}
$$

Fig. 6. (Speed) Update

where each of the disjuncts models one of the updates mentioned above. We give in detail the (Speed) and (Enter) updates in Figures 6 and 7, respectively. The full system specification can be found in Appendix A. Variables and function symbols which are not mentioned in the update rules remain unchanged. In (Enter), t_0, t_1, t_2 and s_1 are fresh constants (t_1 enters segment s_1, t_0 and t_2 are either null_t or represent the previous and next trains of t_1 after the update).

Verification. The verification problems for the RBC are satisfiability problems containing universally quantified formulae, hence cannot be decided by standard methods of reasoning in combinations of theories. Instead, we use the hierarchical reasoning approach from Section 3.2. To this end, we partition our axioms into different extension levels:

- Axiom ($\mathsf{Def}_\mathsf{prio}$) together with an approximation D' of (Def_bd) by a step-function[6] define a local extension $\mathbb{R} \cup \mathbb{N} \cup \{(\mathsf{Def}_\mathsf{prio}), D'\}$ of the combined theory of $\mathbb{R}$ and $\mathbb{N}$.
- This theory is then extended again, by all the remaining axioms in the background theory (resulting in the theory $\mathcal{T}$) and either an invariant (Inv) or the axioms (Init) for the initial state. This is a local pointer extension (Theorem 8).
- Finally, we can extend the resulting theory again by the update rules (Update). This is another local extension (by Theorem 5).

We can decide satisfiability of any set of ground clauses G modulo the resulting theories by repeatedly using the reduction from Section 3.3, using the proper instantiation $\mathcal{K} * [G]$ of the axioms in the current extension level.

Safety properties. As safety property for the RBC we want to prove that we never have two trains on the same segment:

$$(\mathsf{Safe}) := \forall\, t_1, t_2 : \mathit{Train}.\ t_1 \neq t_2 \rightarrow id_s(segm(t_1)) \neq id_s(segm(t_2)).$$

To this end, we need to find a formula (Inv) such that we can prove

[6] This approximation is necessary to accommodate for the lack of SMT solvers that can handle non-linear arithmetic constraints. Similarly, for our tests we needed to fix $\Delta t = 1$ to avoid non-linear constraints.

$$
\begin{array}{l}
\quad s_1 \neq \mathsf{null_s} \ \wedge\ t_1 \neq \mathsf{null_t} \ \wedge\ \mathsf{next_t}(t_1) = \mathsf{null_t} \ \wedge\ \mathsf{prev_t}(t_1) = \mathsf{null_t} \\
\quad \mathsf{incoming}(s_1) = t_1 \ \wedge\ \mathsf{alloc}(s_1) = \mathsf{id_t}(t_1) \\
\quad \mathsf{segm}(t_0) = \mathsf{null_s} \rightarrow t_0 = \mathsf{null_t} \\
\quad t_0 \neq \mathsf{null_t} \rightarrow \mathsf{id_s}(\mathsf{segm}(t_0)) < \mathsf{id_s}(s_1) \\
\forall\, t.\ t \neq t_0 \ \wedge\ \mathsf{id_s}(\mathsf{segm}(t)) < \mathsf{id_s}(s_1) \rightarrow \mathsf{id_s}(\mathsf{segm}(t)) < \mathsf{id_s}(\mathsf{segm}(t_0)) \\
\quad t_0 \neq \mathsf{null_t} \rightarrow \mathsf{next_t}(t_0) = t_2 \\
\quad \mathsf{segm}(t_2) = \mathsf{null_s} \rightarrow t_2 = \mathsf{null_t} \\
\quad t_2 \neq \mathsf{null_t} \rightarrow \mathsf{id_s}(\mathsf{segm}(t_2)) > \mathsf{id_s}(s_1) \\
\forall\, t.\ t \neq t_2 \ \wedge\ \mathsf{id_s}(\mathsf{segm}(t)) > \mathsf{id_s}(s_1) \rightarrow \mathsf{id_s}(\mathsf{segm}(t)) > \mathsf{id_s}(\mathsf{segm}(t_2)) \\
\quad t_2 \neq \mathsf{null_t} \rightarrow \mathsf{prev_t}(t_2) = t_0 \\
\quad pc1' = pc1 \\
\forall\, t.\ t_0 = t_2 \ \wedge\ t \neq t_1 \rightarrow \mathsf{prev_t}'(t) = \mathsf{prev_t}(t) \ \wedge\ \mathsf{next_t}'(t) = \mathsf{next_t}(t) \\
\qquad\qquad\qquad\qquad \wedge\ \mathsf{prev_t}'(t_1) = \mathsf{null_t} \ \wedge\ \mathsf{next_t}'(t_1) = \mathsf{null_t} \\
\forall\, t.\ t_0 \neq t_2 \ \wedge\ t \neq t_0 \ \wedge\ t \neq t_1 \ \wedge\ t \neq t_2 \\
\quad \rightarrow \mathsf{next_t}'(t) = \mathsf{next_t}(t) \ \wedge\ \mathsf{prev_t}'(t) = \mathsf{prev_t}(t) \\
\quad t_0 \neq t_2 \ \wedge\ t_0 \neq \mathsf{null_t} \rightarrow \mathsf{next_t}'(t_0) = t_1 \\
\quad t_0 \neq t_2 \rightarrow \mathsf{prev_t}'(t_0) = \mathsf{prev_t}(t_0) \ \wedge\ \mathsf{next_t}'(t_1) = t_2 \\
\qquad\qquad \wedge\ \mathsf{prev_t}'(t_1) = t_0 \ \wedge\ \mathsf{next_t}'(t_2) = \mathsf{next_t}(t_2) \\
\quad t_0 \neq t_2 \ \wedge\ t_2 \neq \mathsf{null_t} \rightarrow \mathsf{prev_t}'(t_2) = t_1 \\
\forall\, t.\ t \neq t_1 \rightarrow \mathsf{segm}'(t) = \mathsf{segm}(t) \ \wedge\ \mathsf{segm}'(t_1) = s_1 \\
\forall\, t.\ t \neq t_1 \rightarrow spd'(t) = spd(t) \ \wedge\ 0 \leq spd'(t_1) \leq lmax(s_1) \\
\forall\, t.\ t \neq t_1 \rightarrow pos'(t) = pos(t) \ \wedge\ pos'(t_1) = 0 \\
\forall\, s.\ s \neq s_1 \rightarrow train'(s) = train(s) \ \wedge\ train'(s_1) = t_1 \\
\forall\, s.\ alloc'(s) = alloc(s) \\
\forall\, s.\ req'(s) = req(s)
\end{array}
$$

Fig. 7. (Enter) Update

(1) $(\mathsf{Inv}) \cup \neg(\mathsf{Safe}) \models_{\mathcal{T}} \bot$,
(2) $(\mathsf{Init}) \cup \neg(\mathsf{Inv}) \models_{\mathcal{T}} \bot$, and
(3) $(\mathsf{Inv}) \cup (\mathsf{Update}) \cup \neg(\mathsf{Inv}') \models_{\mathcal{T}} \bot$,

where (Update) is the update formula associated with the transition relation obtained by translating the COD specification into TCS [11,9], and (Init) consists of the constraints in the `Init` schema. The background theory $\mathcal{T}$ is obtained from the state schema of the OZ part of the specification: it is the combination of the theories of real numbers and integers, together with function and constant symbols satisfying the constraints given in the state schema.

Calling H-PILoT on problem (3) with (Inv) = (Safe) shows us that (Safe) is not inductive over all transitions. Since we expect the updates to preserve the well-formedness properties in (Init), we tried to use this as our invariant, but with the same result. However, inspection of counterexamples provided by H-PILoT allowed us to identify the following additional constraints needed to make the invariant inductive:

$$
\begin{array}{l}
(\mathsf{Ind_1}) := \forall\, t : Train.\ pc \neq 1 \wedge alloc(nexts(segm(t))) \neq tid(t) \\
\qquad\qquad\qquad\qquad \rightarrow length(segm(t)) - bd(spd(t)) > pos(t) + spd(t) \cdot \Delta t
\end{array}
$$

$$(\mathsf{Ind}_2) := \forall\, t : \mathit{Train}.\ \mathit{pc} \neq 1 \wedge \mathit{pos}(t) \geq \mathit{length}(\mathit{segm}(t)) - d \\ \rightarrow \mathit{spd}(t) \leq \mathit{lmax}(\mathit{nexts}(\mathit{segm}(t)))$$

Thus, define (Inv) as the conjunction (Init) $\wedge$ (Ind_1) $\wedge$ (Ind_2). Now, all of the verification tasks above can automatically be proved using Syspect and H-PILoT, in case (3) after splitting the problem into a number of sub-problems. To ensure that our system is not trivially safe because of inconsistent assumptions, we also check for consistency of $\mathcal{T}$, (Inv) and (Update). Since by Theorem 5 all the update rules in the RBC specification define local theory extensions, and the axioms specifying properties of the data types are extended pointer clauses, by Corollary 10 we obtain the following decidability result.

Corollary 11. *Checking properties (1)–(3) is decidable for all formulae* Inv *expressed as sets of extended pointer clauses with the property that the scalar part belongs to a decidable fragment of the theory of scalars.*

Topological invariants. We also considered certain topological invariants of the system – e.g. that if a train t is inserted between trains t_1, and t_2, the next and prev links are adjusted properly, and if a train leaves a track then its next_t and prev_t links become null. We also checked that if certain reachability conditions – modeled using a binary transitive function reachable with Boolean output which is updated when trains enter or leave the line track – are satisfied before an insertion/removal of trains then they are satisfied also after. We cannot include these examples in detail here; they will be presented in a separate paper.

4.2 Verification of the timed train controller

As pictured in Figure 4, the RBC component described in the previous section is not considered in isolation but in combination with an arbitrary number of trains that are controlled by the RBC. In this section. we detail the basic ideas of the COD component describing the timed controller for the trains.

COD model of the timed train controller. A train is modeled with a class *Train*. The control structure of a train is, as usual for COD, specified with a CSP process:

$$\begin{aligned} \mathit{main} &\stackrel{c}{=} \mathit{REQUEST} \,|||\, \mathit{CHECK} \,|||\, \mathit{DRIVE} \\ \mathit{REQUEST} &\stackrel{c}{=} \mathit{req} \rightarrow \\ &\quad ((\mathit{grant} \rightarrow \mathit{REQUEST}) \,\Box\, (\mathit{reject} \rightarrow \mathit{REQUEST})) \\ \mathit{CHECK} &\stackrel{c}{=} \mathit{check} \rightarrow ((\mathit{brake} \rightarrow \mathit{CHECK}) \,\Box\, (\mathit{release} \rightarrow \mathit{CHECK})) \\ \mathit{DRIVE} &\stackrel{c}{=} \mathit{updSpd} \rightarrow \\ &\quad ((\mathit{enterNewSegm} \rightarrow \mathit{DRIVE}) \,\Box\, (\mathit{updPos} \rightarrow \mathit{DRIVE})) \end{aligned}$$

This process is also illustrated in Figure 8 as an equivalent state machine and it consists of three sub-processes *REQUEST*, *CHECK*, and *DRIVE* running in

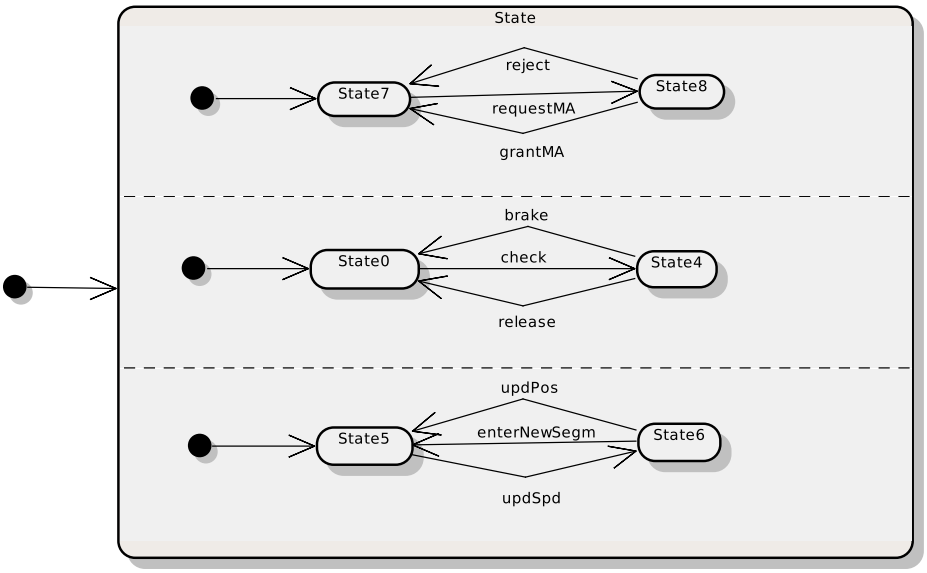


Fig. 8. Control structure of the train controller as state machine

parallel. The first requests extensions of movement authorities, i.e., if the train approaches a new segment it may send such a request to the RBC. The RBC grants or rejects an extension according to the *alloc* attribute from the RBC controller, the data structure that is used to store assignments from segments to trains. The *CHECK* process periodically checks if the train is already beyond a certain danger position *dp* (corresponding to the value of d in the RBC controller) indicating that the train has to apply the brakes in order to stop safely before the end of the segment. The brakes are applied or released according to the result of this check. Finally, the process *DRIVE* updates the speed and position values of the trains. Usually, every event *updSpd* is directly followed by an *updPos* event. But if a new segment is entered the position update is performed by the event *enterNewSegm* because then the position—which is relative to the segment—is reset. In our case study, we made the simplifying assumption that the computed speed and position values are immediately propagated to the RBC controller and used for its calculations (cf. Figure 4).

The timing constraints for the train component are given in the DC part of the specification:

$$\neg(\mathit{true} \frown (\boxminus \mathit{check} \wedge \ell > 0.5\Delta t) \frown \mathit{true}) \tag{1}$$

$$\neg(\mathit{true} \frown \updownarrow \mathit{updPos} \frown (\ell < \Delta t) \frown \updownarrow \mathit{updPos} \frown \mathit{true}) \tag{2}$$

$$\neg(\mathit{true} \frown \updownarrow \mathit{updSpd} \frown \boxminus \mathit{updPos} \frown \updownarrow \mathit{brake} \frown \mathit{true}) \tag{3}$$

The DC formulae are given in terms of interval-based counterexample trace formulae [11], a DC dialect that can be translated into a timed automata model.

DC formula (1) specifies that there is no interval (the $\frown$-operator divides traces into intervals) with a length greater than $0.5\Delta t$ (ℓ always refers to the length of the interval) where no *check* event occurs ($\boxminus$ specifies non-occurrence of events). Formulated positively: at most $0.5\Delta t$ time passes between two check events.

Formula (2) states that at least Δt time passes between two position updates, which corresponds to the time between position updates in the RBC controller. The last formula (3), that is not a real-time formula, demands that a speed and position update cycle is not interfered by a brake event.[7] For the safety of the system it is important that the checks whether to apply the brakes or not are executed often enough. This is ensured by the time constraints (1) and (2).

The data part of the train controller basically reflects the speed and position updates of the RBC controller, but restricted to one train whereas the RBC controller needs to maintain a list of speed and position values. Moreover, while the RBC considers all possible behaviors of trains, the behavior of the single train controller is divided into control decisions and updates according to its control structure described above. For instance, the speed update of the train is given by the schema

`effect_`*updateSpeed*

$\Delta(curSpd)$

$$
\begin{array}{l}
(brakesApplied \leq 0 \land allowedSpd \leq curSpd + incmax) \Rightarrow curSpd' = allowedSpd \\
(brakesApplied \leq 0 \land allowedSpd > curSpd + incmax) \Rightarrow curSpd' = curSpd + incmax \cdot \Delta t \\
(brakesApplied \geq 1 \land curSpd - decmax > 0) \Rightarrow curSpd' = curSpd - decmax \cdot \Delta t \\
(brakesApplied \geq 1 \land curSpd - decmax \leq 0) \Rightarrow curSpd' = 0
\end{array}
$$

This schema sets the speed value corresponding to the speed updates of the RBC controller in Section 2.1 on page 6 except that the decision whether the brakes are applied are made in the *CHECK* cycle which sets the *brakesApplied* variable to the correct value.[8] In addition, when increasing the speed, instead of setting the speed value to an arbitrary value below the maximum speed, the train controller takes the maximal acceleration *incmax* into account. Analogously, the position update corresponds to the position update of the RBC controller reduced to a single train but we use an additional event *enterNewSegment* for the case that the segment is to be changed. The full train controller can be found in Appendix B.

Verification. Using the model-checking approach from [19] (see Section 6), we can automatically prove real-time properties of COD specifications. In this case, we use the approach only on the train controller part *Train*. We show that the safety distance d and the braking distance bd postulated in the RBC controller model can actually be achieved by trains that comply with the train

[7] By this, the brake event is delayed to the next cycle such that *updPos* always refers correctly to the values computed for the current cycle.

[8] The *min* and *max* operators were resolved because the used model checker does not support them.

specification, i.e., trains that update the current speed values and apply their brakes according to the parallel cycles *CHECK* and *DRIVE* from the previous section. That is, we prove that (for an arbitrary train) the train position *curPos* is never beyond its movement authority *ma*:

$$(\mathsf{Safe_T}) := \neg\Diamond(curPos > ma).$$

This formula does not contain real-time constraints but depends on the timing properties specified in the train component.

4.3 Safety of the overall system

The safety property for trains ($\mathsf{Safe_T}$) implies that train controllers satisfying the specification also satisfy the timing assumptions made implicitly in the RBC controller. Compositionality of COD guarantees [11] that it is sufficient to verify these components separately. Thus, by additionally proving that (Inv) is a safety invariant of the RBC, we have shown that the system consisting of a combination of the RBC controller and arbitrarily many train controllers is safe.

5 Modular verification for complex track topologies

We now consider a complex track as described in Figure 1. Assume that the track can be modeled as a directed graph $G = (V, E)$ with the following properties:

(i) The graph G is acyclic (the rail track does not contain cycles);
(ii) The in-degree of every node is at most 2 (at every point at which two lines meet, at most two linear tracks are merged).

Every directed acyclic graph $G = (V, E)$ can be written as the union of the family of all its connected linear subgraphs $\{G_i = (V_i, E_i) \mid i \in I\}$.

Lemma 12. *Let k be an arbitrary (but fixed) stricly positive integer constant. A finite directed graph $G = (V, E)$ is acyclic iff there exists a map $v : V \to \mathbb{Z}$ with the property that for every $(x_1, x_2) \in E$, $v(x_1) \leq v(x_2) - k$.*

Proof: Let w be a weight function that assigns the weight $-k$ to each edge of G. Clearly, G is acyclic iff there exists no cycle of negative weight in the weighted graph (G, w). Let $(G', w') = (V \cup \{x\}, E \cup E', w')$ be a new weighted graph obtained from G by adding a new vertex $x \notin V$, and for every vertex $y \in V$, a new edge (y, x) with weigth 0. The new weighted graph has no cycles of negative weight iff G is acyclic. Let $\mathcal{S} = \{x_i \leq x_j - k \mid (x_i, x_j) \in E\}$ be a system of inequations in the variables V. The following are equivalent:

(1) G (and hence also G') has no cycle.
(2) (G, w) (and hence also (G', w')) has no cycle of negative weight.
(3) The system $\mathcal{S}$ of inequations has a solution $v : V \to \mathbb{Z}$.

The equivalence of (1) and (2) was discussed before. We prove that (2) implies (3). Assume that (G, w) (and hence also (G', w')) has no cycle of negative weight. Then there exists a shortest path between any two nodes, and a solution for $\mathcal{S}$ is $v(x_i) = \delta(x_i, x)$, where $\delta(x_i, x)$ is the length of the shortest path from x_i to x in G'. Indeed, let $(x_1, x_2) \in E$. Then the weight of the shortest path from x_1 to x is possibly smaller than the sum of the weights of edge (x_1, x_2) and the weight of the shortest path from x_2 to x. Thus,

$$\begin{aligned} v(x_1) &= \delta(x_1, x) \\ &\leq w(x_1, x_2) + \delta(x_2, x) \\ &= -k + \delta(x_2, x) \\ &= v(x_2) - k, \end{aligned}$$

so v is a solution of $\mathcal{S}$. To prove that (3) implies (1), assume that $\mathcal{S}$ has a solution and there exists a cycle $(x_1, x_2, \ldots, x_n)$ with $(x_i, x_{i+1}) \in E$ and $x_n = x_1$ in G. Then for all $1 \leq i \leq n$, $v(x_i) \leq v(x_{i+1}) - k$. Thus, $0 = v(x_1) - v(x_1) = \sum_{i=1}^{n}(v(x_i) - v(x_{i+1})) \leq -n * k$. Absurd. □

Theorem 13. *For every track topology satisfying conditions (i) and (ii) above we can find a decomposition* $\mathcal{L} = \{\mathsf{ltrack}_i \mid i \in I\}$ *into linear tracks such that if* $(x, y) \in E$ *then* $y = \mathsf{next}_s^{\mathsf{ltrack}_i}(x)$ *for some* $i \in I$ *and for every* $\mathsf{ltrack} \in \mathcal{L}$ *identifiers are increasing w.r.t.* $\mathsf{next}_s^{\mathsf{ltrack}}$.

Proof: Consider a track topology satisfying conditions (i) and (ii). Let V be the set of all segments on the track topology and let $G = (V, E)$ be the directed graph modeling the immediate successor relationships between segments on the track. With G we associate the following system of inequalities $\mathcal{S} = \{x_i \leq x_j - k \mid (x_i, x_j) \in E\}$, where k is an arbitrary, strictly positive constant. By Lemma 12, $G = (V, E)$ is acyclic iff $\mathcal{S}$ has a solution $v : V \to \mathbb{Z}$. We can use this solution for labeling the nodes. Then for every linear track, the labeling is strictly monotone w.r.t. the next field (for ensuring this it is sufficient to take $k = 1$; we can ensure the injectivity of the labelling by choosing k to be the total number of segments on the complex track system). □

Our goal is now to prove that we can guaranteeing the safety of the whole system if we can prove that all controllers for the linear tracks are safe. This result is a consequence of results on modular verification of complex systems presented in [24]. A direct, informal argument for this result can be given as follows. Assume that we start from a safe state with an admissible change and the resulting state is not safe. Then there exists a linear track ltrack for which one of the conditions in the invariant Inv does not hold in the new state after the transition. We can analyze all possibilities using a case distinction depending on the type of update which takes place on the linear track ltrack:

– The position, speed and request updates are not problematic: they refer to one track only (and are the same on subtracks) so the safety of such update

rules for the individual tracks implies safety in the complex system and vice-versa. If the new state after the update is not safe, then an unsafe state is reached after the update on one linear track. Contradiction.
- The allocation updates are uniform: if a train wants to enter the track and another train has requested the segment the requested track is always allocated to the train with higher priority. ($\mathsf{incoming}^{\mathsf{ltrack}}(s)$ can be regarded as a "disjunction" of all $\mathsf{request}^{\mathsf{ltrack}'}(s)$ over all tracks $\mathsf{ltrack}' \neq \mathsf{ltrack}$; there can be at most one such ltrack' because of condition (ii)). If the new state after the update is not safe, then the outcome of an allocation update on one of the linear tracks is not safe, which contradicts the fact that allocation updates on all linear tracks preserve the safety property.
- We now consider enter and leave updates, which involve two tracks. Since for every train/segment we have corresponding $\mathsf{next}_t^{\mathsf{ltrack}}/\mathsf{next}_s^{\mathsf{ltrack}}$ fields for every linear track ltrack, the safety of the updates rules for individual tracks implies that "global" satefy is guaranteed as well.

Thus in all cases – since (when seen from the perspective of a single track ltrack) the initial safe state of the whole system restricts to a safe state of track ltrack and the transition restricts to a (possibly empty) transition on ltrack – it would follow that the specification of the controller of at least one linear track ltrack does not satisfy the corresponding safety property. But this contradicts our assumption.

We now present the sketch of a more formal proof which uses the ideas in [24].

5.1 Definition of the composition of the linear track controllers

Let $\mathcal{L} = \{\mathsf{ltrack}_i \mid i \in I\}$ be a family of linear tracks with the property that for every $\mathsf{ltrack} \in \mathcal{L}$ identifiers are increasing w.r.t. $\mathsf{next}_s^{\mathsf{ltrack}}$.

We assume that for each linear track $\mathsf{ltrack} \in \mathcal{L}$ we have one controller RBC^{ltrack} which uses the control protocol described in Section 2.1, where we label the functions describing the train and segment succession using indices (e.g. we use $\mathsf{next}_t^{\mathsf{ltrack}}, \mathsf{prev}_t^{\mathsf{ltrack}}$ for the successor/predecessor of a train on ltrack, and $\mathsf{next}_s^{\mathsf{ltrack}}$, $\mathsf{prev}_s^{\mathsf{ltrack}}$ for the successor/predecessor of a segment on ltrack. Assume that these controllers are compatible on their common parts, i.e.

(1) if two tracks $\mathsf{track}_1, \mathsf{track}_2$ have a common subtrack track_3 then the corresponding fields agree, i.e. whenever $s, \mathsf{next}_s^{\mathsf{track}_i}(s)$ are on track_3, $\mathsf{next}_s^{\mathsf{track}_1}(s) = \mathsf{next}_s^{\mathsf{track}_2}(s) = \mathsf{next}_s^{\mathsf{track}_3}(s)$ (and the same for prev_s, and also for $\mathsf{next}_t, \mathsf{prev}_t$ on the corresponding tracks);
(2) the update rules are compatible for trains jointly controlled. [9]

We will show that under these conditions, proving safety for the complex track can be reduced to checking safety of linear train tracks with incoming and outgoing trains.

[9] We also assume that all priorities of the trains on the complex track are different.

5.2 States

A state s of the system obtained by the interconnection of the linear controllers in $\mathcal{L}$ is a structure:

$$(P_{\mathsf{t}}, P_{\mathsf{s}}, \mathbb{R}, \mathbb{Z}, \{\mathsf{next}_t^{\mathsf{ltrack}}, \mathsf{prev}_t^{\mathsf{ltrack}}, \mathsf{next}_s^{\mathsf{ltrack}}, \mathsf{prev}_s^{\mathsf{ltrack}}\}_{\mathsf{track}\in\mathcal{L}} \cup \{\mathsf{segm}, \mathsf{train}, \mathsf{pos}, ...\})$$

satisfying the axioms Def in Section 4.1 (seen as a family of copies of these axioms with the respective functions indexed by the tracks in $\mathcal{L}$), and with the additional condition that if a train/segment is not on a track ltrack then the $\mathsf{next}_t^{\mathsf{ltrack}}, \mathsf{prev}_t^{\mathsf{ltrack}}$, resp. $\mathsf{next}_s^{\mathsf{ltrack}}, \mathsf{prev}_s^{\mathsf{ltrack}}$ fields are null.

Compatibility of states. We now establish links between "local" states, referring to one track only, and "global" states, referring to the whole track system.

Restriction. If considering only functions and variables which refer to one linear track $\mathsf{ltrack} \in \mathcal{L}$ in the complex track topology, every structure s representing a state of the system of trains on the complex track topology restricts to a structure:

$$s_{\mathsf{ltrack}} = (P_{\mathsf{t}}, P_{\mathsf{s}}, \mathbb{R}, \mathbb{Z}, \{\mathsf{next}_t^{\mathsf{ltrack}}; \mathsf{prev}_t^{\mathsf{ltrack}}; \mathsf{next}_s^{\mathsf{ltrack}}; \mathsf{prev}_s^{\mathsf{ltrack}}; \mathsf{segm}, \mathsf{train}, \mathsf{pos}, ...\})$$

Glueing. Any family $\{s_{\mathsf{ltrack}_i} \mid i \in I\}$ of states on the component tracks which agree on the common sub-tracks can be "glued together" to a structure s by putting together all the functions corresponding to the single tracks. The fact that the new structure is well defined follows from the compatibility of the family $\{s_{\mathsf{ltrack}_i} \mid i \in I\}$ of states on the common sub-tracks.

Lemma 14. *A state s of the system is a model $(P_{\mathsf{t}}, P_{\mathsf{s}}, \mathbb{R}, \mathbb{Z}, \{\mathsf{next}_t^{\mathsf{ltrack}}, \mathsf{prev}_t^{\mathsf{ltrack}}, \mathsf{next}_s^{\mathsf{ltrack}}, \mathsf{prev}_s^{\mathsf{ltrack}}\}_{\mathsf{track}\in\mathcal{L}} \cup \{\mathsf{segm}, \mathsf{train}, \mathsf{pos}, ...\})$ the axioms Def in Section 4.1, where all the functions relativized to tracks are compatible on common subtracks. The following hold:*

(a) Every state s of the system of trains on the complex track restricts to a state s_{ltrack} of the system of trains on its component linear track.
(b) Any family $\{s_{\mathsf{ltrack}_i} \mid i \in I\}$ of states on the component tracks which agree on the common sub-tracks can be "glued together" to a state s of the system of trains on the complex track topology.

Proof: (a) follows from the way the restriction of a state of the whole track systems to one track is defined. Clearly, if s satisfies the axioms Def in Sect. 4.1 (seen as a family of copies of these axioms with the respective functions indexed by the tracks in $\mathcal{L}$), then also s_{ltrack} satisfies the specific instances of the axioms Def referring to the track ltrack.

(b) From any family $\{s_{\mathsf{ltrack}_i} \mid i \in I\}$ of states on the component tracks which agree on the common sub-tracks we can construct, as explained above, a structure s in which all the operations for the component tracks are considered together. As $s_{\mathsf{ltrack}_i}, i \in I$ satisfy the axioms Def referring to the track ltrack_i, it immediately follows that s satisfies the axioms Def in Sect. 4.1 seen as a family of copies of these axioms with the functions indexed by the tracks in $\mathcal{L}$.

Initial states and safe states. Both properties (a) and (b) above hold if we only consider *initial* states (i.e. states satisfying the initial conditions) and *safe* states (i.e. states satisfying the safety conditions in the invariant Inv). This can be seen by simply analyzing the formulae in (Init) and (Inv).

Similar properties hold for parallel actions and for transitions.

5.3 Parallel events

The events in the complex system are sets of independent events in the component systems with a linear track. Here we assume, for the sake of simplicity, that on every linear track only one event can take place at a time. We also assume that if two linear tracks have a common part, and in each of them an event of the same type e_{track_i} (position update, speed update, allocation update, ...) changes the state of the common part, then the events are synchronized, and the changes on the common track segment coincide. [10]

Compatibility of parallel events. By definition and by the requirement that in the global system only one event can take place at a time, the restriction of an event to the variables known on a linear component of the system is a (single) event for that linear track. [11] Due to the assumption that similar events on tracks which overlap synchronize, it follows that if we have a family of compatible events for the linear tracks they "glue together" to an event for the complex track.

5.4 Transitions

Assume that we are in a state s, and a transition t (parallel composition of various independent transitions on the component tracks) takes place in this state in the complex track system. Let s_{ltrack} be the part of state s noticed on track ltrack, and t_{ltrack} the changes in the transition which refer to changes on track ltrack. Seen from the perspective of the track ltrack, transition ltrack changes the state s_{ltrack} to a state s'_{ltrack}. Since we assumed that the controllers synchronize on the common tracks (in the sense that the events produce compatible changes), we can show that the new states $\{s'_{\mathsf{ltrack}} \mid \mathsf{ltrack} \in \mathcal{F}\}$ have the property that their reducts on subtracks agree, and they can be "glued together" to a state of the whole system. We can see this by analyzing the various types of transitions.

(i) The position, speed and request updates are not problematic, since they are fully synchronized on any connected component of the track topology.

[10] It follows therefore that on all segments on a connected component of the complex topology the events "position update", "speed update", "request", and "allocation" take place at the same time. These events can take place independently for any two disjoint connected components.

[11] We can extend the formalization by allowing parallel events on linear tracks, but for the sake of simplicity we do not do so now.

(The formulae which define speed updates are expressed as upper and lower bounds for the new speed; the position update depends on speed. This ensures compatibility.)

(ii) The allocation updates are uniform: if a train wants to enter the track on segment s and another train has requested the segment s, s is allocated to the train with higher priority. (As pointed out when giving the intuitive justification, $\mathsf{in}^{\mathsf{ltrack}}(s)$ can be regarded as a "disjunction" of all $\mathsf{req}^{\mathsf{ltrack}'}(s)$ over all tracks $\mathsf{ltrack}' \neq \mathsf{ltrack}$ and by assumption that the in-degree of every node is at most 2 made at the beginning of this section, it can be at most one such ltrack'). Therefore their effects are compatible.

(iii) The enter and leave updates involve two tracks only, which share at least the segment on which entering/leaving takes place. By analyzing the special form of these updates we show that compatibility of the resulting states is guaranteed.

This compatibility of changes is used for defining transitions of (parallel) events.

5.5 Modular verification

We now prove that safety can be checked modularly. Assume that the individual controllers for the linear tracks are safe.

Safety of initial states. By the remarks above, any initial state s of the complex system is obtained by glueing together certain initial states $\{s_{\mathsf{track}_i} \mid i \in I\}$ of the linear track subsystems. These states are all safe if and only if the global state s is safe.

Invariant checking. Assume now that we are in a safe state s. Let e be an event in the complex system, and let s' be the resulting state. We know therefore that for every $\mathsf{ltrack} \in \mathcal{F}$, s'_{ltrack} is a safe state. We proved that safe states "glue together" to a safe state. Thus, s' is safe.

Theorem 15. *Consider a complex track topology satisfying conditions (i)–(ii) above. Let $\mathcal{L} = \{\mathsf{ltrack}_i \mid i \in I\}$ be its decomposition into a finite family of finite linear tracks such that for all $\mathsf{ltrack}_1, \mathsf{ltrack}_2 \in \mathcal{L}$, $\mathcal{L}$ contains all their common maximal linear subtracks. Assume that the tracks $\mathsf{ltrack}_i \in \mathcal{L}$ (with increasing segment identifiers w.r.t. $\mathsf{next}_s^{\mathsf{ltrack}}$) are controlled by controllers RBC^{ltrack_i} using the protocols in Section 2.1 which synchronize on common subtracks. Then we can guarantee safety of the control protocol for the controller of the complex track obtained by interconnecting all linear track controllers $\{RBC^{\mathsf{ltrack}_i} \mid i \in I\}$.*

6 From specification to verification

For the practical application of verification techniques tool support is essential. For this reason, we introduce a full tool chain in this section for automatically checking the invariance of safety properties starting from a given specification and give some experimental results for our RBC case study.

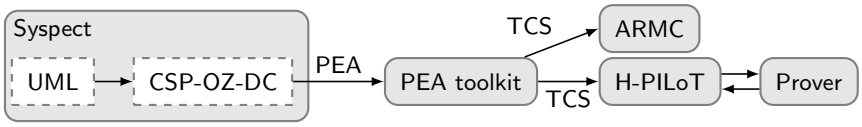


Fig. 9. Tool chain

Tool chain. The tool chain is sketched in Fig. 9. In order to capture the systems we want to verify, we use the COD front-end Syspect (cf. Section 2). [11] defines the semantics of COD in terms of a timed automata model called Phase Event Automata (PEA). A translation from PEA into TCS is given in [11], which is implemented in the PEA toolkit[12] and used by Syspect.

Given an invariance property, a Syspect model can directly be exported into a TCS in the syntax of H-PILoT. If the specification's background theory consists of chains of local theory extensions, the user needs to specify via input dialog (i) that the pointer extension of H-PILoT is to be used; (ii) which level of extension is used for each function symbol of the specification. With this information, our tool chain can verify the invariance of a safety condition fully automatically by checking its invariance for each transition update (cf. Section 3.1). Therefore, for each update, Syspect exports a file that is handed over to H-PILoT. The invariance of a safety condition is proven if H-PILoT detects the unsatisfiability of each verification task. Otherwise, H-PILoT generates a model violating the invariance of the desired property, which may be used to fix the problems in the specification.

In addition, the PEA toolkit also supports output of TCS into the input language of the abstraction refinement model checker ARMC [22], which we used to verify correctness of the timed train controller from our example.

Experimental results. Table 1 gives experimental results for checking the RBC controller[13]. The table lists execution times for the involved tools: (sys) contains the times needed by Syspect and the PEA toolkit to write the TCS, (hpi) the time of H-PILoT to compute the reduction and to check satisfiability with Yices as back-end, (yic) the time of Yices to check the proof tasks without reductions by H-PILoT. Due to some semantics-preserving transformations during the translation process the resulting TCS consists of 46 transitions. Since our invariant (Inv) is too complex to be handled by the clausifier of H-PILoT, we check the invariant for every transition in two parts yielding 92 proof obligations. In addition, results for the most extensive proof obligation are stated: one part of the speed update. Further, we performed consistency tests to ensure that the system is not blocked by a false precondition.

Table 1. Results

	(sys)	(hpi)	(yic)
(Inv) *unsat*			
Part 1	11s	72s	52s
Part 2	11s	124s	131s
speed update	11s	8s	45s
(Safe) *sat*	9s	8s	t.o.
Consistency	13s	3s	(U) 2s

(obtained on: AMD64, dual-core 2 GHz, 4 GB RAM)

[12] `http://csd.informatik.uni-oldenburg.de/projects/epea.html`

[13] Note that even though our proof methods fully support parametric specifications, we instantiated some of the parameters for the experiments because the underlying provers Yices and ARMC do not support non-linear constraints.

The table shows that the time to compute the TCS is insignificant and that the overall time to verify all transition updates with Yices and H-PILoT does not differ much. On the speed update H-PILoT was 5 times faster than Yices alone. During the development of the case study H-PILoT helped us finding the correct transition invariants by providing models for satisfiable transitions. The table lists our tests with the verification of condition (Safe), which is not inductive over all transitions (cf. Section 3): here, H-PILoT was able to provide a model after 8s whereas Yices detected unsatisfiability for 17 problems, returned "unknown" for 28, and timed out once (listed as (t.o) in the table). For the consistency check H-PILoT was able to provide a model after 3s, whereas Yices answered "unknown" (listed as (U)).

In addition, we used ARMC to verify the property (Safe_T) of the timed train controller. The full TCS for this proof tasks comprises 8 parallel components, more than 3300 transitions, and 28 real-valued variables and clocks (so it is an infinite state system). For this reason, the verification took 26 hours (on a standard desktop computer).

7 Conclusion

We augmented existing techniques for the verification of real-time systems to cope with rich data structures like pointers. We identified a decidable fragment of the theory of pointers, and used it to model systems of trains on linear tracks with incoming and outgoing trains. We then proved that certain types of complex track systems can be decomposed into linear tracks, and that proving safety of train controllers for such complex systems can be reduced to proving safety of controllers for linear tracks. We implemented our approach in a new tool chain taking high-level specifications in terms of COD as input. To uniformly specify processes, data and time, [17,4,26] use similar combined specification formalisms. We preferred COD due to its strict separation of control, data, and time, and its compositionality (cf. Section 2), which is essential for automatic verification. There is also sophisticated tool support given by Syspect and the PEA toolkit. Using this tool chain we automatically verified safety properties of a complex case study, closing the gap between a formal high-level language and the proposed verification method for TCS. We plan to extend the case study to consider emergency messages (like in [9]), possibly coupled with updates in the track topology, or updates of priorities. Concerning the track topology, we are experimenting with more complex axiomatizations (e.g. for connectedness properties) that are not in the pointer fragment presented in Section 3.3; we already proved various locality results. We also plan to study possibilities of automated invariant generation in such parametric systems.

Acknowledgments. Many thanks to Werner Damm and Ernst-Rüdiger Olderog for their helpful comments.

References

1. Abdulla, P.A., Delzanno, G., Rezine, A.: Approximated parameterized verification of infinite-state processes with global conditions. Form. Method Syst. Des. 34(2), 126–156 (2009)
2. Abdulla, P.A., Jonsson, B.: Verifying networks of timed processes. In: Steffen, B. (ed.) TACAS'98. LNCS, vol. 1384, pp. 298–312. Springer, Heidelberg (1998)
3. Abdulla, P.A., Jonsson, B., Nilsson, M., Saksena, M.: A survey of regular model checking. In: Gardner, P., Yoshida, N. (eds.) CONCUR'04. LNCS, vol. 3170, pp. 35–48. Springer, Heidelberg (2004)
4. Abrial, J.R., Mussat, L.: Introducing dynamic constraints in B. In: Bert, D. (ed.) B'98. LNCS, vol. 1393, pp. 83–128. Springer, Heidelberg (1998)
5. Arons, T., Pnueli, A., Ruah, S., Xu, J., Zuck, L.D.: Parameterized verification with automatically computed inductive assertions. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV'01. LNCS, vol. 2102, pp. 221–234. Springer, Heidelberg (2001)
6. Bradley, A., Manna, Z., Sipma, H.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI'06. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2006)
7. Clarke, E.M., Talupur, M., Veith, H.: Environment abstraction for parameterized verification. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI'06. LNCS, vol. 3855, pp. 126–141. Springer, Heidelberg (2006)
8. Faber, J., Ihlemann, C., Jacobs, S., Sofronie-Stokkermans, V.: Automatic verification of parametric specifications with complex topologies. In: Méry, D. and Merz, S. (eds.) IFM'10, LNCS, vol. 6396, pp. 152–167, Springer, Heidelberg (2010).
9. Faber, J., Jacobs, S., Sofronie-Stokkermans, V.: Verifying CSP-OZ-DC specifications with complex data types and timing parameters. In: Davies, J., Gibbons, J. (eds.) IFM'07. LNCS, vol. 4591, pp. 233–252. Springer, Heidelberg (2007)
10. Haxthausen, A.E., Peleska, J.: A domain-oriented, model-based approach for construction and verification of railway control systems. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. LNCS, vol. 4700, pp. 320–348. Springer, Heidelberg (2007)
11. Hoenicke, J.: Combination of Processes, Data, and Time. Ph.D. thesis, University of Oldenburg, Germany (2006)
12. Hoenicke, J., Olderog, E.R.: CSP-OZ-DC: A combination of specification techniques for processes, data and time. Nordic J. Comput. 9(4), 301–334 (2002)
13. Ihlemann, C., Jacobs, S., Sofronie-Stokkermans, V.: On local reasoning in verification. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS'08. LNCS, vol. 4963, pp. 265–281. Springer, Heidelberg (2008)
14. Ihlemann, C., Sofronie-Stokkermans, V.: System description: H-PILoT. In: Schmidt, R.A. (ed.) CADE'09. LNCS, vol. 5663, pp. 131–139. Springer, Heidelberg (2009)
15. Jacobs, S., Sofronie-Stokkermans, V.: Applications of hierarchic reasoning in the verification of complex systems. ENTCS 174(8), 39–54 (2007)
16. Lahiri, S.K., Bryant, R.E.: Indexed predicate discovery for unbounded system verification. In: Alur, R., Peled, D.A. (eds.) CAV'04. LNCS, vol. 3114, pp. 135–147. Springer, Heidelberg (2004)
17. Mahony, B.P., Dong, J.S.: Blending Object-Z and timed CSP: An introduction to TCOZ. In: ICSE'98. pp. 95–104 (1998)
18. McPeak, S., Necula, G.: Data structure specifications via local equality axioms. In: Etessami, K., Rajamani, S.K. (eds.) CAV'05. LNCS, vol. 3576, pp. 476–490 (2005)

19. Meyer, R., Faber, J., Hoenicke, J., Rybalchenko, A.: Model checking duration calculus: A practical approach. Form. Asp. Comput. 20(4–5), 481–505 (2008)
20. Möller, M., Olderog, E.R., Rasch, H., Wehrheim, H.: Integrating a formal method into a software engineering process with UML and Java. Form. Asp. Comput. 20, 161–204 (2008)
21. Platzer, A., Quesel, J.D.: European train control system: A case study in formal verification. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM'09. LNCS, vol. 5885, pp. 246–265. Springer, Heidelberg (2009)
22. Podelski, A., Rybalchenko, A.: ARMC: The logical choice for software model checking with abstraction refinement. In: Hanus, M. (ed.) PADL'07. LNCS, vol. 4354, pp. 245–259. Springer, Heidelberg (2007)
23. Sofronie-Stokkermans, V.: Hierarchic reasoning in local theory extensions. In: Nieuwenhuis, R. (ed.) CADE'05. LNCS, vol. 3632, pp. 219–234. Springer, Heidelberg (2005)
24. Sofronie-Stokkermans, V.: Sheaves and geometric logic and applications to modular verification of complex systems. ENTCS 230, 161–187 (2009)
25. Sofronie-Stokkermans, V.: Hierarchical reasoning for the verification of parametric systems. In: Giesl, J., Hähnle, R. (eds.) IJCAR'10. LNAI, vol. 6173, pp. 171–187. Springer, Heidelberg (2010)
26. Woodcock, J.C.P., Cavalcanti, A.L.C.: A concurrent language for refinement. In: Butterfield, A., Strong, G., Pahl, C. (ed.) IWFM'01. BCS Elec. Works. Comp. (2001)

A Update rules of the case study

Speed Update. (Speed) is given by the following axioms:

$$\mathsf{pc} = 1 \ \wedge \ \mathsf{pc}' = 2$$

$$\forall\, t : \mathsf{pos}(t) < \mathsf{length}(\mathsf{segm}(t)) - d \\ \rightarrow max\{0,\ \mathsf{spd}(t) - maxDecl \cdot \Delta_t\} \leq \mathsf{spd}'(t) \leq \mathsf{max}(\mathsf{segm}(t))$$

$$\forall\, t : \mathsf{pos}(t) \geq \mathsf{length}(\mathsf{segm}(t)) - d \ \wedge \ \mathsf{alloc}(\mathsf{next}_s(\mathsf{segm}(t))) = \mathsf{id}_\mathsf{t}(t) \\ \rightarrow max\{0,\ \mathsf{spd}(t) - maxDecl \cdot \Delta_t\} \\ \leq \mathsf{spd}'(t) \leq min\{\mathsf{max}(\mathsf{segm}(t)), \mathsf{max}(\mathsf{next}_\mathsf{s}(\mathsf{segm}(t)))\}$$

$$\forall\, t : \mathsf{pos}(t) \geq \mathsf{length}(\mathsf{segm}(t)) - d \ \wedge \ \mathsf{alloc}(\mathsf{next}_\mathsf{s}(\mathsf{segm}(t))) \neq \mathsf{id}_\mathsf{t}(t) \\ \rightarrow \mathsf{spd}'(t) = max\{0,\ \mathsf{spd}(t) - maxDecl \cdot \Delta_t\}$$

Request Update. (Request) is given by the following axioms:

$$\mathsf{pc} = 2 \ \wedge \ \mathsf{pc}' = 3$$

$$\forall\, s : \mathsf{train}(\mathsf{prev}_\mathsf{s}(s)) = \mathsf{null} \rightarrow \mathsf{req}'(s) = \mathsf{null}$$

$$\forall\, s : \mathsf{train}(\mathsf{prev}_\mathsf{s}(s)) \neq \mathsf{null} \ \wedge \ \mathsf{length}(\mathsf{prev}_\mathsf{s}(s)) - \mathsf{pos}(\mathsf{train}(\mathsf{prev}_\mathsf{s}(s))) \leq d \\ \rightarrow \mathsf{req}'(s) = \mathsf{train}(\mathsf{prev}_\mathsf{s}(s))$$

$$\forall\, s : \mathsf{train}(\mathsf{prev}_\mathsf{s}(s)) \neq \mathsf{null} \ \wedge \ \mathsf{length}(\mathsf{prev}_\mathsf{s}(s)) - \mathsf{pos}(\mathsf{train}(\mathsf{prev}_\mathsf{s}(s))) > d \\ \rightarrow \mathsf{req}'(s) = \mathsf{null}$$

Leave Update. The following axioms model the (Leave) update, where train t_1 from segment s_1 wants to leave the route given by our list of track segments:

$$s_1 \neq \mathsf{null}_1 \ \wedge \ t_1 \neq \mathsf{null}_2 \ \wedge \ \mathsf{train}(s_1) = t_1 \ \wedge \ \mathsf{pc}' = \mathsf{pc}$$

$$\forall\, t : t \neq t_1 \ \wedge \ \mathsf{next}_\mathsf{t}(t) = t_1 \rightarrow \mathsf{next}_\mathsf{t}{}'(t) = \mathsf{next}_\mathsf{t}(t_1)$$

$$\forall\, t : t \neq t_1 \ \wedge \ \mathsf{next}_\mathsf{t}(t) \neq t_1 \rightarrow \mathsf{next}_\mathsf{t}{}'(t) = \mathsf{next}_\mathsf{t}(t)$$

$$\forall\, t : t \neq t_1 \ \wedge \ \mathsf{prev}_\mathsf{t}(t) = t_1 \rightarrow \mathsf{prev}_\mathsf{t}{}'(t) = \mathsf{prev}_\mathsf{t}(t_1)$$

$$\forall\, t : t \neq t_1 \ \wedge \ \mathsf{prev}_\mathsf{t}(t) \neq t_1 \rightarrow \mathsf{prev}_\mathsf{t}{}'(t) = \mathsf{prev}_\mathsf{t}(t)$$

$$\forall\, t : t = t_1 \rightarrow \mathsf{next}_\mathsf{t}{}'(t) = \mathsf{null}_\mathsf{t}$$

$$\forall\, t : t = t_1 \rightarrow \mathsf{prev}_\mathsf{t}{}'(t) = \mathsf{null}_\mathsf{t}$$

$$\forall\, t : t \neq t_1 \rightarrow \mathsf{segm}'(t) = \mathsf{segm}(t)$$

$$\forall\, t : t = t_1 \rightarrow \mathsf{segm}'(t) = \mathsf{null}_1$$

$$\forall\, s : s \neq s_1 \rightarrow \mathsf{train}'(s) = \mathsf{train}(s)$$

$$\forall\, s : s = s_1 \rightarrow \mathsf{train}'(s) = \mathsf{null}_2$$

$$\forall\, s : s \neq s_1 \rightarrow \mathsf{alloc}'(s) = \mathsf{alloc}(s)$$

$$\forall\, s : s = s_1 \rightarrow \mathsf{alloc}'(s) = 0$$

Allocation Update. (Allocation) is given by the following axioms:

$$\mathsf{pc} = 3 \ \wedge \ \mathsf{pc}' = 4$$

$$\forall\, s : \mathsf{alloc}(s) \neq 0 \rightarrow \mathsf{alloc}'(s) = \mathsf{alloc}(s)$$

$$\forall\, s : \mathsf{alloc}(s) = 0 \ \wedge \ \mathsf{incoming}(s) = \mathsf{null} \rightarrow \mathsf{alloc}'(s) = \mathsf{alloc}(s)$$

$$\forall\, s : \mathsf{alloc}(s) = 0 \ \wedge \ \mathsf{incoming}(s) \neq \mathsf{null} \ \wedge \ \mathsf{req}(s) = 0 \rightarrow \mathsf{alloc}'(s) = \mathsf{id_t}(\mathsf{incoming}(s))$$

$$\forall\, s : \mathsf{alloc}(s) = 0 \ \wedge \ \mathsf{incoming}(s) \neq \mathsf{null} \ \wedge \ \mathsf{req}(s) \neq 0 \ \wedge \ \mathsf{prio}(\mathsf{id_t}(\mathsf{incoming}(s))) < \mathsf{prio}(\mathsf{req}(s)) \rightarrow \mathsf{alloc}'(s) = \mathsf{req}(s)$$

$$\forall\, s : \mathsf{alloc}(s) = 0 \ \wedge \ \mathsf{incoming}(s) \neq \mathsf{null} \ \wedge \ \mathsf{req}(s) \neq 0 \ \wedge \ \mathsf{prio}(\mathsf{id_t}(\mathsf{incoming}(s))) \geq \mathsf{prio}(\mathsf{req}(s)) \rightarrow \mathsf{alloc}'(s) = \mathsf{id_t}(\mathsf{incoming}(s))$$

Position Update. (Position) is given by the following axioms:

$$\mathsf{pc} = 2 \ \wedge \ \mathsf{pc}' = 1$$

$$\forall\, t : \mathsf{pos}(t) + \mathsf{spd}(t) \cdot \Delta t \leq \mathsf{length}(\mathsf{segm}(t)) \rightarrow \mathsf{pos}'(t) = \mathsf{pos}(t) + \mathsf{spd}(t) \cdot \Delta t$$

$$\forall\, t : \mathsf{pos}(t) + \mathsf{spd}(t) \cdot \Delta t > \mathsf{length}(\mathsf{segm}(t)) \rightarrow \mathsf{pos}'(t) = (\mathsf{pos}(t) + \mathsf{spd}(t) \cdot \Delta t) - \mathsf{length}(\mathsf{segm}(t))$$

$$\forall\, t : \mathsf{segm}(t) = \mathsf{null}_1 \rightarrow \mathsf{segm}'(t) = \mathsf{null}_1$$

$$\forall\, t : \mathsf{segm}(t) \neq \mathsf{null}_1 \ \wedge \ \mathsf{pos}(t) + \mathsf{spd}(t) \cdot \Delta t \leq \mathsf{length}(\mathsf{segm}(t)) \rightarrow \mathsf{segm}'(t) = \mathsf{segm}(t)$$

$$\forall\, t : \mathsf{segm}(t) \neq \mathsf{null}_1 \ \wedge \ \mathsf{pos}(t) + \mathsf{spd}(t) \cdot \Delta t > \mathsf{length}(\mathsf{segm}(t)) \rightarrow \mathsf{segm}'(t) = \mathsf{next_s}(\mathsf{segm}(t))$$

$$\forall\, s : \mathsf{train}(s) = \mathsf{null}_2 \ \wedge \ \mathsf{prev_s}(s) = \mathsf{null}_1 \rightarrow \mathsf{train}'(s) = \mathsf{null}_2$$

$$\forall\, s : \mathsf{train}(s) = \mathsf{null}_2 \ \wedge \ \mathsf{prev_s}(s) \neq \mathsf{null}_1 \ \wedge \ \mathsf{train}(\mathsf{prev_s}(s)) = \mathsf{null}_2 \rightarrow \mathsf{train}'(s) = \mathsf{null}_2$$

$$\forall\, s : \mathsf{train}(s) = \mathsf{null}_2 \ \wedge \ \mathsf{prev_s}(s) \neq \mathsf{null}_1 \ \wedge \ \mathsf{train}(\mathsf{prev_s}(s)) \neq \mathsf{null}_2 \ \wedge \ \mathsf{pos}(\mathsf{train}(\mathsf{prev_s}(s))) + \mathsf{spd}(\mathsf{train}(\mathsf{prev_s}(s))) \cdot \Delta t > \mathsf{length}(\mathsf{prev_s}(s)) \rightarrow \mathsf{train}'(s) = \mathsf{train}(\mathsf{prev_s}(s))$$

$$\forall\, s : \mathsf{train}(s) = \mathsf{null}_2 \ \wedge \ \mathsf{prev_s}(s) \neq \mathsf{null}_1 \ \wedge \ \mathsf{train}(\mathsf{prev_s}(s)) \neq \mathsf{null}_2 \ \wedge \ \mathsf{pos}(\mathsf{train}(\mathsf{prev_s}(s))) + \mathsf{spd}(\mathsf{train}(\mathsf{prev_s}(s))) \cdot \Delta t \leq \mathsf{length}(\mathsf{prev_s}(s)) \rightarrow \mathsf{train}'(s) = \mathsf{null}_2$$

$$\forall\, s : \mathsf{train}(s) \neq \mathsf{null}_2 \ \wedge \ \mathsf{pos}(\mathsf{train}(s)) + \mathsf{spd}(\mathsf{train}(s)) \cdot \Delta t > \mathsf{length}(s) \rightarrow \mathsf{train}'(s) = \mathsf{null}_2$$

$$\forall\, s : \mathsf{train}(s) \neq \mathsf{null}_2 \ \wedge \ \mathsf{pos}(\mathsf{train}(s)) + \mathsf{spd}(\mathsf{train}(s)) \cdot \Delta t \leq \mathsf{length}(s) \rightarrow \mathsf{train}'(s) = \mathsf{train}(s)$$

$$\forall\, s : \mathsf{train}(s) = \mathsf{null}_2 \rightarrow \mathsf{alloc}'(s) = \mathsf{alloc}(s)$$

$$\forall\, s : \mathsf{train}(s) \neq \mathsf{null}_2 \ \wedge \ \mathsf{pos}(\mathsf{train}(s)) + \mathsf{spd}(\mathsf{train}(s)) \cdot \Delta t \leq \mathsf{length}(s) \rightarrow \mathsf{alloc}'(s) = \mathsf{alloc}(s)$$

$$\forall\, s : \mathsf{train}(s) \neq \mathsf{null}_2 \ \wedge \ \mathsf{pos}(\mathsf{train}(s)) + \mathsf{spd}(\mathsf{train}(s)) \cdot \Delta t > \mathsf{length}(s) \rightarrow \mathsf{alloc}'(s) = 0$$

Enter Update. The following axioms model the (Enter) update. Train t_1 wants to enter at track segment s_1. t_0 is the closest train on a segment smaller than s_1, t_2 is the closest train on a segment larger than s_1 (if such trains exist, otherwise $\mathsf{null_t}$):

$$
\begin{array}{l}
\quad s_1 \neq \mathsf{null_s} \ \wedge\ t_1 \neq \mathsf{null_t} \ \wedge\ \mathsf{next_t}(t_1) = \mathsf{null_t} \ \wedge\ \mathsf{prev_t}(t_1) = \mathsf{null_t} \\
\quad \mathsf{incoming}(s_1) = t_1 \ \wedge\ \mathsf{alloc}(s_1) = \mathsf{id_t}(t_1) \\
\quad \mathsf{segm}(t_0) = \mathsf{null_s} \rightarrow t_0 = \mathsf{null_t} \\
\quad t_0 \neq \mathsf{null_t} \rightarrow \mathsf{id_s}(\mathsf{segm}(t_0)) < \mathsf{id_s}(s_1) \\
\forall\, t.\ t \neq t_0 \ \wedge\ \mathsf{id_s}(\mathsf{segm}(t)) < \mathsf{id_s}(s_1) \rightarrow \mathsf{id_s}(\mathsf{segm}(t)) < \mathsf{id_s}(\mathsf{segm}(t_0)) \\
\quad t_0 \neq \mathsf{null_t} \rightarrow \mathsf{next_t}(t_0) = t_2 \\
\quad \mathsf{segm}(t_2) = \mathsf{null_s} \rightarrow t_2 = \mathsf{null_t} \\
\quad t_2 \neq \mathsf{null_t} \rightarrow \mathsf{id_s}(\mathsf{segm}(t_2)) > \mathsf{id_s}(s_1) \\
\forall\, t.\ t \neq t_2 \ \wedge\ \mathsf{id_s}(\mathsf{segm}(t)) > \mathsf{id_s}(s_1) \rightarrow \mathsf{id_s}(\mathsf{segm}(t)) > \mathsf{id_s}(\mathsf{segm}(t_2)) \\
\quad t_2 \neq \mathsf{null_t} \rightarrow \mathsf{prev_t}(t_2) = t_0 \\
\quad pc1' = pc1 \\
\forall\, t.\ t_0 = t_2 \ \wedge\ t \neq t_1 \rightarrow \mathsf{prev_t}'(t) = \mathsf{prev_t}(t) \ \wedge\ \mathsf{next_t}'(t) = \mathsf{next_t}(t) \\
\qquad\qquad\qquad\qquad \wedge\ \mathsf{prev_t}'(t_1) = \mathsf{null_t} \ \wedge\ \mathsf{next_t}'(t_1) = \mathsf{null_t} \\
\forall\, t.\ t_0 \neq t_2 \ \wedge\ t \neq t_0 \ \wedge\ t \neq t_1 \ \wedge\ t \neq t_2 \\
\quad \rightarrow \mathsf{next_t}'(t) = \mathsf{next_t}(t) \ \wedge\ \mathsf{prev_t}'(t) = \mathsf{prev_t}(t) \\
\quad t_0 \neq t_2 \ \wedge\ t_0 \neq \mathsf{null_t} \rightarrow \mathsf{next_t}'(t_0) = t_1 \\
\quad t_0 \neq t_2 \rightarrow \mathsf{prev_t}'(t_0) = \mathsf{prev_t}(t_0) \ \wedge\ \mathsf{next_t}'(t_1) = t_2 \\
\qquad\qquad \wedge\ \mathsf{prev_t}'(t_1) = t_0 \ \wedge\ \mathsf{next_t}'(t_2) = \mathsf{next_t}(t_2) \\
\quad t_0 \neq t_2 \ \wedge\ t_2 \neq \mathsf{null_t} \rightarrow \mathsf{prev_t}'(t_2) = t_1 \\
\forall\, t.\ t \neq t_1 \rightarrow \mathsf{segm}'(t) = \mathsf{segm}(t) \ \wedge\ \mathsf{segm}'(t_1) = s_1 \\
\forall\, t.\ t \neq t_1 \rightarrow \mathit{spd}'(t) = \mathit{spd}(t) \ \wedge\ 0 \leq \mathit{spd}'(t_1) \leq \mathit{lmax}(s_1) \\
\forall\, t.\ t \neq t_1 \rightarrow \mathit{pos}'(t) = \mathit{pos}(t) \ \wedge\ \mathit{pos}'(t_1) = 0 \\
\forall\, s.\ s \neq s_1 \rightarrow \mathit{train}'(s) = \mathit{train}(s) \ \wedge\ \mathit{train}'(s_1) = t_1 \\
\forall\, s.\ \mathit{alloc}'(s) = \mathit{alloc}(s) \\
\forall\, s.\ \mathit{req}'(s) = \mathit{req}(s)
\end{array}
$$

B CSP-OZ-DC specification of the case study

In this section, we present the entire CSP-OZ-DC model of our case study as it is generated by Syspect (we adjusted some of the line breaks to increase readability). Figure 4 gives a structural overview of the components of the case study, which are listed as COD specifications in the following (starting with some axiomatic definitions, followed by the RBC controller, and finally the train controller is listed).

In addition to the components of Fig. 4, there is a further component $RBC-Com$, representing a communication interface between Train component and RBC. It is basically used to simplify presentation and verification of the RBC controller because it encapsulates the communication of data that otherwise would have to be considered within the RBC controller. In particular, we make the simplifying assumption that the *getData* event of the $RBC-Com$ component is only possible if the *alloc* variable assigns the next segment to the current train. Then *getData* returns the length and allowed maximal speed for the next segment. The operation *getData* does not change any value of the RBC.

Note that even though our proof methods fully support parametric specifications, it were necessary to instantiate some of the parameters for the experiments because the underlying provers yices and ARMC do not support non-linear constraints.

B.1 Axiomatic definitions

$$[Train, Segment]$$

$$d, decmax, gmax, \Delta t : \mathbb{R}$$
$$decmax = 2$$
$$gmax \leq 30$$
$$d \geq bd(gmax) + gmax$$

$$tnil : Train$$

$$snil : Segment$$

$$prio : \mathbb{Z} \rightarrow \mathbb{Z}$$
$$\forall i, j : \mathbb{Z} \mid prio(i) = prio(j) \bullet i = j$$

$$nexts, prevs : Segment \rightarrow Segment$$
$$\forall s : Segment \bullet nexts(prevs(s)) = s$$
$$\forall s : Segment \bullet prevs(nexts(s)) = s$$

$$length, lmax : Segment \rightarrow \mathbb{R}$$
$$\forall s : Segment \bullet length(s) > d + gmax \cdot \Delta t$$
$$\forall s : Segment \bullet 0 < lmax(s) \wedge lmax(s) \leq gmax$$
$$\forall s : Segment \bullet lmax(s) \geq lmax(prevs(s)) - decmax$$

$$sid : Segment \rightarrow \mathbb{Z}$$
$$\forall s1, s2 : Segment \mid s1 \neq s2 \bullet sid(s1) \neq sid(s2)$$
$$\forall s : Segment \bullet sid(s) > 0$$
$$sid(snil) = 0$$
$$\forall s : Segment \bullet sid(nexts(s)) > sid(s)$$

$$tid : Train \rightarrow \mathbb{Z}$$
$$\forall t1, t2 : Train \mid t1 \neq t2 \bullet tid(t1) \neq tid(t2)$$
$$\forall t : Train \bullet tid(t) > 0$$
$$tid(tnil) = 0$$

$$incoming : Segment \rightarrow Train$$

$bd : \mathbb{R} \rightarrow \mathbb{R}$

$\forall r : \mathbb{R} \mid r \leq 2 \bullet bd(r) = 1$
$\forall r : \mathbb{R} \mid r \leq 4 \land r > 2 \bullet bd(r) = 4$
$\forall r : \mathbb{R} \mid r \leq 6 \land r > 4 \bullet bd(r) = 9$
$\forall r : \mathbb{R} \mid r \leq 8 \land r > 6 \bullet bd(r) = 16$
$\forall r : \mathbb{R} \mid r \leq 10 \land r > 8 \bullet bd(r) = 25$
$\forall r : \mathbb{R} \mid r \leq 12 \land r > 10 \bullet bd(r) = 36$
$\forall r : \mathbb{R} \mid r \leq 14 \land r > 12 \bullet bd(r) = 49$
$\forall r : \mathbb{R} \mid r \leq 16 \land r > 14 \bullet bd(r) = 64$
$\forall r : \mathbb{R} \mid r \leq 18 \land r > 16 \bullet bd(r) = 81$
$\forall r : \mathbb{R} \mid r \leq 20 \land r > 18 \bullet bd(r) = 100$
$\forall r : \mathbb{R} \mid r \leq 22 \land r > 20 \bullet bd(r) = 121$
$\forall r : \mathbb{R} \mid r \leq 24 \land r > 22 \bullet bd(r) = 144$
$\forall r : \mathbb{R} \mid r \leq 26 \land r > 24 \bullet bd(r) = 169$
$\forall r : \mathbb{R} \mid r \leq 28 \land r > 26 \bullet bd(r) = 196$
$\forall r : \mathbb{R} \mid r \leq 30 \land r > 28 \bullet bd(r) = 225$

SegmentData

$train : Segment \rightarrow Train$
$req : Segment \rightarrow \mathbb{Z}$
$alloc : Segment \rightarrow \mathbb{Z}$

TrainData

$segm : Train \rightarrow Segment$
$next : Train \rightarrow Train$
$spd : Train \rightarrow \mathbb{R}$
$pos : Train \rightarrow \mathbb{R}$
$prev : Train \rightarrow Train$

B.2 RBC controller

RBC − OZ

`method` $enter : [s1? : Segment;\ t0? : Train;\ t1? : Train;\ t2? : Train]$
`method` $leave : [ls? : Segment;\ lt? : Train]$
`local_chan` $allocation$
`local_chan` $request$
`local_chan` $updatePosition$
`local_chan` $updateSpeed$

$$
\begin{aligned}
InitState &\stackrel{c}{=} ((enter \rightarrow InitState) \\
&\quad \Box\ (leave \rightarrow InitState) \\
&\quad \Box\ (updateSpeed \rightarrow State1)) \\
State1 &\stackrel{c}{=} ((enter \rightarrow State1) \\
&\quad \Box\ (leave \rightarrow State1) \\
&\quad \Box\ (request \rightarrow State2)) \\
State2 &\stackrel{c}{=} ((allocation \rightarrow State3) \\
&\quad \Box\ (enter \rightarrow State2) \\
&\quad \Box\ (leave \rightarrow State2)) \\
State3 &\stackrel{c}{=} ((enter \rightarrow State3) \\
&\quad \Box\ (leave \rightarrow State3) \\
&\quad \Box\ (updatePosition \rightarrow InitState)) \\
\texttt{main} &\stackrel{c}{=} InitState
\end{aligned}
$$

$sd : SegmentData$
$td : TrainData$

`Init`

$\forall t : Train \bullet alloc(nexts(segm(t))) = tid(t)$
$\quad \lor\ length(segm(t)) - bd(spd(t)) > pos(t)$
$\forall s : Segment \bullet segm(train(s)) = s$
$\forall t : Train \bullet train(segm(t)) = t$
$\forall t : Train \bullet next(prev(t)) = t$
$\forall t : Train \bullet prev(next(t)) = t$
$\forall t : Train \bullet 0 \leq pos(t)$
$\forall t : Train \bullet pos(t) \leq length(segm(t))$
$\forall t : Train \bullet 0 \leq spd(t)$
$\forall t : Train \bullet spd(t) \leq lmax(segm(t))$
$\forall t : Train \bullet alloc(segm(t)) = tid(t)$

effect_*updateSpeed*

$\Delta(spd)$

$\forall\, t : Train \mid pos(t) < length(segm(t)) - d \wedge spd(t) - decmax \cdot \Delta t > 0$
$\quad \bullet\ max\{0, spd(t) - decmax \cdot \Delta t\} \leq spd'(t) \leq lmax(segm(t))$
$\forall\, t : Train \mid pos(t) \geq length(segm(t)) - d \wedge alloc(nexts(segm(t))) = tid(t)$
$\quad \bullet\ max\{0, spd(t) - decmax \cdot \Delta t\} \leq spd'(t) \leq min\{lmax(segm(t)), lmax(nexts(segm(t)))\}$
$\forall\, t : Train \mid pos(t) \geq length(segm(t)) - d \wedge \neg\, alloc(nexts(segm(t))) = tid(t)$
$\quad \bullet\ spd'(t) = max\{0, spd(t) - decmax \cdot \Delta t\}$

effect_*leave*

$\Delta(segm, next, train, prev, alloc)$
$ls? : Segment;\ lt? : Train$

$ls? \neq snil$
$lt? \neq tnil$
$train(ls?) = lt?$
$\forall\, t : Train \mid t \neq lt? \wedge next(t) \neq lt? \bullet next'(t) = next(t)$
$\forall\, t : Train \mid t \neq lt? \wedge next(t) = lt? \bullet next'(t) = next(lt?)$
$\forall\, t : Train \mid t \neq lt? \wedge prev(t) \neq lt? \bullet prev'(t) = prev(t)$
$\forall\, t : Train \mid t \neq lt? \wedge prev(t) = lt? \bullet prev'(t) = prev(lt?)$
$next'(lt?) = tnil$
$prev'(lt?) = tnil$
$\forall\, t : Train \mid t \neq lt? \bullet segm'(t) = segm(t)$
$segm'(lt?) = snil$
$\forall\, s : Segment \mid s \neq ls? \bullet train'(s) = train(s)$
$\forall\, s : Segment \mid s \neq ls? \bullet alloc'(s) = alloc(s)$
$train'(ls?) = tnil$
$alloc'(ls?) = 0$

effect_*request*

$\Delta(req)$

$\forall\, s : Segment \mid train(prevs(s)) = tnil \bullet req'(s) = 0$
$\forall\, s : Segment \mid length(prevs(s)) - pos(train(prevs(s))) \leq d$
$\quad \wedge\ train(prevs(s)) \neq tnil \bullet req'(s) = tid(train(prevs(s)))$
$\forall\, s : Segment \mid length(prevs(s)) - pos(train(prevs(s))) > d$
$\quad \wedge\ train(prevs(s)) \neq tnil \bullet req'(s) = 0$

effect_*allocation*

$\Delta(alloc)$

$\forall\, s : Segment \mid alloc(s) \neq 0 \bullet alloc'(s) = alloc(s)$
$\forall\, s : Segment \mid alloc(s) = 0 \wedge incoming(s) = tnil \bullet alloc'(s) = req(s)$
$\forall\, s : Segment \mid alloc(s) = 0 \wedge req(s) = 0 \wedge incoming(s) \neq tnil$
$\quad \bullet\ alloc'(s) = tid(incoming(s))$
$\forall\, s : Segment \mid alloc(s) = 0 \wedge prio(tid(incoming(s))) < prio(req(s))$
$\quad \wedge\ incoming(s) \neq tnil \wedge req(s) \neq 0 \bullet alloc'(s) = req(s)$
$\forall\, s : Segment \mid alloc(s) = 0 \wedge prio(tid(incoming(s))) \geq prio(req(s))$
$\quad \wedge\ incoming(s) \neq tnil \wedge req(s) \neq 0 \bullet alloc'(s) = tid(incoming(s))$

effect_*updatePosition*

$\Delta(segm, train, pos, alloc)$

$\forall\, t : Train \mid pos(t) + spd(t) \cdot \Delta t \leq length(segm(t)) \bullet pos'(t) = pos(t) + spd(t) \cdot \Delta t$
$\forall\, t : Train \mid pos(t) + spd(t) \cdot \Delta t > length(segm(t))$
$\quad \bullet\ pos'(t) = (pos(t) + spd(t) \cdot \Delta t) - length(segm(t))$
$\forall\, t : Train \mid pos(t) + spd(t) \cdot \Delta t \leq length(segm(t)) \land segm(t) \neq snil$
$\quad \bullet\ segm'(t) = segm(t)$
$\forall\, t : Train \mid pos(t) + spd(t) \cdot \Delta t > length(segm(t)) \land segm(t) \neq snil$
$\quad \bullet\ segm'(t) = nexts(segm(t))$
$\forall\, t : Train \mid segm(t) = snil \bullet segm'(t) = snil$
$\forall\, s : Segment \mid train(s) = tnil \land prevs(s) = snil \bullet train'(s) = tnil$
$\forall\, s : Segment \mid train(s) = tnil \land prevs(s) \neq snil \land train(prevs(s)) = tnil$
$\quad \bullet\ train'(s) = tnil$
$\forall\, s : Segment \mid train(s) = tnil \land prevs(s) \neq snil \land train(prevs(s)) \neq tnil$
$\quad \land\ pos(train(prevs(s))) + spd(train(prevs(s))) \cdot \Delta t > length(prevs(s))$
$\quad \bullet\ train'(s) = train(prevs(s))$
$\forall\, s : Segment \mid train(s) = tnil \land prevs(s) \neq snil \land train(prevs(s)) \neq tnil$
$\quad \land\ pos(train(prevs(s))) + spd(train(prevs(s))) \cdot \Delta t \leq length(prevs(s))$
$\quad \bullet\ train'(s) = tnil$
$\forall\, s : Segment \mid train(s) \neq tnil \land pos(train(s)) + spd(train(s)) \cdot \Delta t > length(s)$
$\quad \bullet\ train'(s) = tnil$
$\forall\, s : Segment \mid train(s) \neq tnil \land pos(train(s)) + spd(train(s)) \cdot \Delta t \leq length(s)$
$\quad \bullet\ train'(s) = train(s)$
$\forall\, s : Segment \mid train(s) \neq tnil \land pos(train(s)) + spd(train(s)) \cdot \Delta t > length(s)$
$\quad \bullet\ alloc'(s) = 0$
$\forall\, s : Segment \mid train(s) \neq tnil \land pos(train(s)) + spd(train(s)) \cdot \Delta t \leq length(s)$
$\quad \bullet\ alloc'(s) = alloc(s)$
$\forall\, s : Segment \mid train(s) = tnil \bullet alloc'(s) = alloc(s)$

effect_*enter*

$\Delta(segm, next, train, spd, pos, prev)$
$s1? : Segment;\ t0? : Train;\ t1? : Train;\ t2? : Train$

$\forall\, s1, s2 : Segment \bullet tid(incoming(s1)) \neq tid(train(s2))$
$s1? \neq snil$
$t1? \neq tnil$
$next(t1?) = tnil$
$prev(t1?) = tnil$
$incoming(s1?) = t1?$
$alloc(s1?) = tid(t1?)$
$segm(t0?) = snil \Rightarrow t0? = tnil$
$t0? \neq tnil \Rightarrow sid(segm(t0?)) < sid(s1?)$
$\forall\, t : Train \mid t \neq t0? \land sid(segm(t)) < sid(s1?) \bullet sid(segm(t)) < sid(segm(t0?))$
$segm(t2?) = snil \Rightarrow t2? = tnil$
$t2? \neq tnil \Rightarrow sid(segm(t2?)) > sid(s1?)$
$\forall\, t : Train \mid t \neq t2? \land sid(segm(t)) > sid(s1?) \bullet sid(segm(t)) > sid(segm(t2?))$
$t0? \neq tnil \Rightarrow next(t0?) = t2?$
$t2? \neq tnil \Rightarrow prev(t2?) = t0?$
$t0? = t2? \Rightarrow next'(t1?) = tnil$
$t0? = t2? \Rightarrow prev'(t1?) = tnil$
$\forall\, t : Train \mid t \neq t1? \land t0? = t2? \bullet next'(t) = next(t)$
$\forall\, t : Train \mid t \neq t1? \land t0? = t2? \bullet prev'(t) = prev(t)$
$t0? \neq tnil \land t0? \neq t2? \Rightarrow next'(t0?) = t1?$
$t0? \neq t2? \Rightarrow prev'(t0?) = prev(t0?)$
$t0? \neq t2? \Rightarrow next'(t1?) = t2?$
$t0? \neq t2? \Rightarrow prev'(t1?) = t0?$
$t0? \neq t2? \Rightarrow next'(t2?) = next(t2?)$
$t0? \neq t2? \land t2? \neq tnil \Rightarrow prev'(t2?) = t1?$
$\forall\, t : Train \mid t0? \neq t2? \land t \neq t0? \land t \neq t1? \land t \neq t2? \bullet next'(t) = next(t)$
$\forall\, t : Train \mid t0? \neq t2? \land t \neq t0? \land t \neq t1? \land t \neq t2? \bullet prev'(t) = prev(t)$
$\forall\, t : Train \mid t \neq t1? \bullet segm'(t) = segm(t)$
$\forall\, t : Train \mid t \neq t1? \bullet spd'(t) = spd(t)$
$\forall\, t : Train \mid t \neq t1? \bullet pos'(t) = pos(t)$
$segm'(t1?) = s1?$
$0 \leq spd'(t1?)$
$spd'(t1?) \leq lmax(s1?)$
$pos'(t1?) = 0$
$\forall\, s : Segment \mid s \neq s1? \bullet train'(s) = train(s)$
$train'(s1?) = t1?$

B.3 Train controller and communication interface

RBC − Com

$\texttt{chan}\ grantMA : [ma2! : \mathbb{R};\ rid2! : \mathbb{Z};\ spd2! : \mathbb{R}]$
$\texttt{chan}\ reject : [rid1! : \mathbb{Z}]$
$\texttt{chan}\ requestMA : [rid? : \mathbb{Z};\ spd? : \mathbb{R}]$
$\texttt{local\_chan}\ getData : [ma3? : \mathbb{R};\ rid3! : \mathbb{Z};\ spd3? : \mathbb{R}]$

$State10 \stackrel{c}{=} ((getData \rightarrow State11)$
$\quad \Box\ (reject \rightarrow main))$
$State11 \stackrel{c}{=} (grantMA \rightarrow main)$
$main \stackrel{c}{=} (requestMA \rightarrow State10)$

$old\_spd\_param : \mathbb{R}$
$new\_spd, new\_ma : \mathbb{R}$
$rid : \mathbb{Z}$

enable_*reject*
$rid1! : \mathbb{Z}$

effect_*reject*
$\Delta(rid)$
$rid1! : \mathbb{Z}$

$rid1! = rid$
$rid' = 0$

enable_*getData*
$ma3? : \mathbb{R};\ rid3! : \mathbb{Z};\ spd3? : \mathbb{R}$

effect_*getData*
$\Delta(new\_spd, new\_ma)$
$ma3? : \mathbb{R};\ rid3! : \mathbb{Z};\ spd3? : \mathbb{R}$

$new\_ma' = ma3?$
$new\_spd' = spd3?$
$rid3! = rid$
$new\_ma' > bdmax + gmax + gmax$
$0 < new\_spd' \leq gmax$
$new\_spd' \geq old\_spd\_param - decmax$

enable_*grantMA*
$ma2! : \mathbb{R};\ rid2! : \mathbb{Z};\ spd2! : \mathbb{R}$

effect_*grantMA*
$\Delta(rid)$
$ma2! : \mathbb{R};\ rid2! : \mathbb{Z};\ spd2! : \mathbb{R}$

$rid2! = rid$
$rid' = 0$
$ma2! = new\_ma$
$spd2! = new\_spd$

enable_*requestMA*
$rid? : \mathbb{Z};\ spd? : \mathbb{R}$

effect_*requestMA*
$\Delta(old\_spd\_param, rid)$
$rid? : \mathbb{Z};\ spd? : \mathbb{R}$

$rid' = rid?$
$old\_spd\_param' = spd?$

Train

```
method updateSpeed
method updatePosition
method grantMA : [ma2? : ℝ; rid2? : ℤ; spd2? : ℝ]
method reject : [rid1? : ℤ]
method requestMA : [rid! : ℤ; spd! : ℝ]
local_chan brake
local_chan check
local_chan enterNewSegm
local_chan release
```

$$
\begin{aligned}
InitialState1 &\stackrel{c}{=} (check \rightarrow State4)\\
InitialState2 &\stackrel{c}{=} (updateSpeed \rightarrow State6)\\
InitialState3 &\stackrel{c}{=} (requestMA \rightarrow State8)\\
State4 &\stackrel{c}{=} ((brake \rightarrow InitialState1)\\
&\quad \Box\ (release \rightarrow InitialState1))\\
State6 &\stackrel{c}{=} ((enterNewSegm \rightarrow InitialState2)\\
&\quad \Box\ (getPos \rightarrow InitialState2))\\
State8 &\stackrel{c}{=} ((grantMA \rightarrow InitialState3)\\
&\quad \Box\ (reject \rightarrow InitialState3))\\
main &\stackrel{c}{=} ((InitialState1\\
&\quad |||\ InitialState2\\
&\quad |||\ InitialState3) \stackrel{o}{_9}\ \texttt{Stop})
\end{aligned}
$$

$$
\begin{array}{l}
nextAllowedSpd : \mathbb{R}\\
curPos : \mathbb{R}\\
nextMa : \mathbb{R}\\
allowedSpd : \mathbb{R}\\
applyBrakes : \mathbb{R}\\
decmax : \mathbb{R}\\
bdmax : \mathbb{R}\\
ma : \mathbb{R}\\
brakesApplied : \mathbb{R}\\
curSpd : \mathbb{R}\\
gmax : \mathbb{R}\\
dp : \mathbb{R}\\
incmax : \mathbb{R}\\
selfid : \mathbb{Z}\\
\hline
decmax = 2\\
0 < incmax < decmax\\
0 < allowedSpd \leq gmax \leq 30\\
bdmax \geq 225\\
selfid > 0
\end{array}
$$

Init

$$
\begin{array}{l}
dp > curPos\\
curPos > 0\\
0 \leq curSpd \leq allowedSpd\\
applyBrakes = 0\\
brakesApplied = 0\\
ma - dp \geq bdmax + gmax\\
nextAllowedSpd = 0\\
nextMa = ma
\end{array}
$$

enable_*reject*

$$
\begin{array}{l}
rid1? : \mathbb{Z}\\
\hline
selfid = rid1?
\end{array}
$$

effect_*reject*

$$
rid1? : \mathbb{Z}
$$

effect_*check*

$\Delta(applyBrakes)$

$(curPos \geq dp \land allowedSpd > nextAllowedSpd) \Rightarrow applyBrakes' = 1$
$(curPos \geq dp \land allowedSpd \leq nextAllowedSpd) \Rightarrow applyBrakes' = 0$
$curPos < dp \Rightarrow applyBrakes' = 0$

enable_*enterNewSegm*

$curPos + curSpd \geq ma \land nextAllowedSpd > 0$

effect_*enterNewSegm*

$\Delta(nextAllowedSpd, ma, curPos, allowedSpd, dp)$

$curPos' = curPos + curSpd - ma$
$ma' = nextMa$
$dp' = ma' - bdmax - gmax$
$allowedSpd' = nextAllowedSpd$
$nextAllowedSpd' = 0$

enable_*grantMA*

$ma2? : \mathbb{R};\ rid2? : \mathbb{Z};\ spd2? : \mathbb{R}$

$rid2? = selfid$

effect_*grantMA*

$\Delta(nextAllowedSpd, nextMa)$
$ma2? : \mathbb{R};\ rid2? : \mathbb{Z};\ spd2? : \mathbb{R}$

$nextAllowedSpd' = spd2?$
$nextMa' = ma2?$

enable_*requestMA*

$rid! : \mathbb{Z};\ spd! : \mathbb{R}$

effect_*requestMA*

$rid! : \mathbb{Z};\ spd! : \mathbb{R}$

$rid! = selfid$
$spd! = allowedSpd$

enable_*updatePosition*

$curPos + curSpd < ma \lor nextAllowedSpd \leq 0$

effect_*updatePosition*

$\Delta(curPos)$

$curPos' = curPos + curSpd$

enable_*release*

$applyBrakes \leq 0$

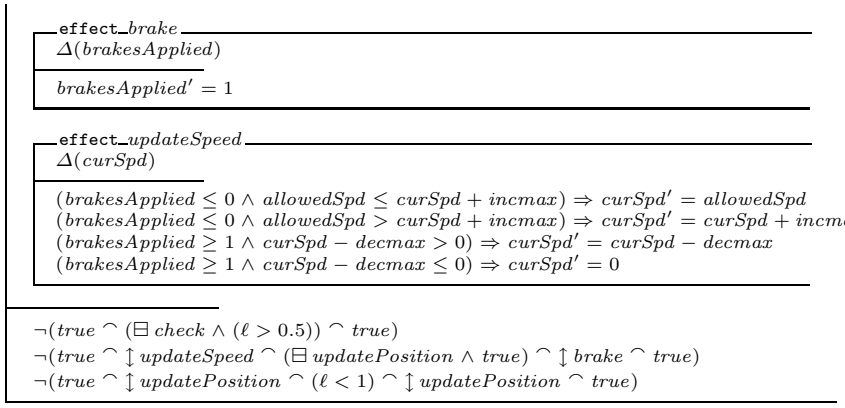
effect_*release*

$\Delta(brakesApplied)$

$brakesApplied' = 0$

enable_*brake*

$applyBrakes \geq 1$


effect_brake
Δ(brakesApplied)
brakesApplied′ = 1
effect_updateSpeed
Δ(curSpd)
(brakesApplied ≤ 0 ∧ allowedSpd ≤ curSpd + incmax) ⇒ curSpd′ = allowedSpd
(brakesApplied ≤ 0 ∧ allowedSpd > curSpd + incmax) ⇒ curSpd′ = curSpd + incmax
(brakesApplied ≥ 1 ∧ curSpd − decmax > 0) ⇒ curSpd′ = curSpd − decmax
(brakesApplied ≥ 1 ∧ curSpd − decmax ≤ 0) ⇒ curSpd′ = 0
¬(true ⌢ (⊟ check ∧ (ℓ > 0.5)) ⌢ true)
¬(true ⌢ ↕ updateSpeed ⌢ (⊟ updatePosition ∧ true) ⌢ ↕ brake ⌢ true)
¬(true ⌢ ↕ updatePosition ⌢ (ℓ < 1) ⌢ ↕ updatePosition ⌢ true)