Automatic Verification of Parametric Specifications with Complex Topologies

Johannes Faber, Carsten Ihlemann, Swen Jacobs, Viorica Sofronie-Stokkermans

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 Sect. 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 (Sect. 3).
   - 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 work was partly supported by the German Research Council (DFG) under grant SFB/TR 14 AVACS. See `http://www.avacs.org` for more information.

- Then we analyze 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 Sect. 4, 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 (Sect. 5).
(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.

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 will show under which situations safety of the system with complex track topology is 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

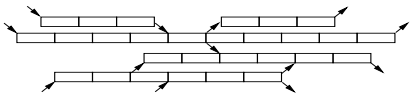

Fig. 1. Complex Track Topology

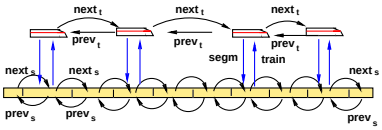

Fig. 2. Linear Track Topology

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 integrated a simple timed train controller Train into the model. This allowed 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[4], 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 Sect. 1 (the full case study is presented in [8]). We exploit the benefits of COD in (i) the specification of a complex 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 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.

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

CSP part. The processes and their interdependency is 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:

`method` *enter* : [*s1?* : *Segment*; *t0?* : *Train*; *t1?* : *Train*; *t2?* : *Train*]
`method` *leave* : [*ls?* : *Segment*; *lt?* : *Train*]
`local_chan` *alloc*, *req*, *updPos*, *updSpd*

`main` $\stackrel{c}{=}$ ((*updSpd*→*State1*) □(*leave*→`main`) □(*enter*→`main`))
State1 $\stackrel{c}{=}$ ((*req*→*State2*) □(*leave*→*State1*) □(*enter*→*State1*))
State2 $\stackrel{c}{=}$ ((*alloc*→*State3*) □(*leave*→*State2*) □(*enter*→*State2*))
State3 $\stackrel{c}{=}$ ((*updPos*→`main`) □(*leave*→*State3*) □(*enter*→*State3*))

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

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

TrainData

$segm : Train \to Segment$	[Train segment]
$next : Train \to Train$	[Next train]
$spd : Train \to \mathbb{R}$	[Speed]
$pos : Train \to \mathbb{R}$	[Current position]
$prev : Train \to Train$	[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 $bd : \mathbb{R} \to \mathbb{R}$ (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[5] of track segments, where every segment has additional properties specified by the constraints in the state schema.

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

E.g., *sid* is increasing along the *nexts* pointer, the *length* of a segment is bounded from below in terms of d and $\mathit{gmax} \cdot \Delta t$, and the difference between local maximal speeds on neighboring segments is bounded by $\mathit{decmax} \cdot \Delta t$. Finally, we have a function *incoming*, the value of which is either a train which wants to enter the given segment from outside the current route, or *tnil* if there is no such train. Although the valuation of *incoming* can change during an execution, we consider the constraint (*) 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

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

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 such that it safely stops before reaching the end of the segment.

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

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

[5] 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.

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 Fig. 3. The train controller Train consists of three timed components running in parallel. The first updates the train's position. This component contains e.g. the DC formula

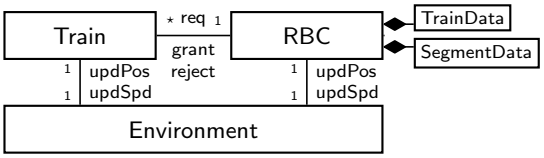

Fig. 3. Structural overview

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

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 full train controller can be found in [8].

3 Modular Verification

In this section, we combine two approaches for the verification of safety properties of COD specifications:

- We introduce the invariant checking approach and present decidability results for local theory extensions that imply decidability of the invariant checking problem for a large class of parameterized systems.
- We illustrate how we can combine this invariant checking for the RBC specification with a method for model checking of real-time properties (introduced in [19]) for the COD specification for a single train Train.

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.[6] 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* $(\mathsf{Inv}), (\mathsf{Init})$ *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 *(resp.* T *satisfies the property* (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[7].
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, for (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.

[6] We can address bounded model checking problems in a similar way, cf. [15,9,13].
[7] This last step is the only part which is not fully automatized. 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 $\mathcal{T}_0 \subseteq \mathcal{T}_1$ 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 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, define local theory extensions.

Theorem 5 ([13]). *Assume that $\{\phi_i \mid i \in I\}$ are formulae over the base signature such that $\phi_i(\overline{x}) \wedge \phi_j(\overline{x}) \models_{\mathcal{T}_0} \bot$ for $i \neq j$, and that s_i, t_i are (possibly equal)*

terms over the signature Σ 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 $i \in I$. Then the extension of $\mathcal{T}_0$ with axioms of the form $\mathsf{Def}(f)$ is local.

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

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.[8] 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].

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 $t' = \mathsf{null}_\mathsf{p}$ 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 $\mathcal{T}_s \cup \mathcal{EQ}_P$ of $\mathcal{T}_s$ and $\mathcal{EQ}_P$, the many-sorted theory of pure equality over pointer sorts,

[8] 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.

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 Thm. 5.

Example 9. Let $P = \{\mathsf{sg}(segment), \mathsf{t}(train)\}$, and let $\mathsf{next_t}, \mathsf{prev_t} : \mathsf{t} \to \mathsf{t}$, and $\mathsf{next_s}, \mathsf{prev_s} : \mathsf{sg} \to \mathsf{sg}$, and $\mathsf{train} : \mathsf{sg} \to \mathsf{t}$, $\mathsf{segm} : \mathsf{t} \to \mathsf{sg}$, and functions of scalar sort as listed at the beginning of Sect. 2.1. All axioms describing the background theory and the initial state in Sect. 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 Thm. 3 and Cor. 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.

3.4 Example: Verification of the Case Study

We demonstrate how the example from Sect. 1 can be verified by using a combination of the invariant checking approach presented in Sect. 3.1 and a model checking approach for timing properties. This combination is necessary because the example contains both the RBC component with its discrete updates, and the train controller Train with real-time safety properties. 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.

Verification of the RBC. 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 Sect. 3.2.

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

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

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

(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:

$$\begin{aligned}(\mathsf{Ind}_1) := \forall\, t : \mathit{Train}.\ & \mathit{pc} \neq \mathit{InitState} \wedge \mathit{alloc}(\mathit{nexts}(\mathit{segm}(t))) \neq \mathit{tid}(t) \\ & \rightarrow \mathit{length}(\mathit{segm}(t)) - \mathit{bd}(\mathit{spd}(t)) > \mathit{pos}(t) + \mathit{spd}(t) \cdot \Delta t \\ (\mathsf{Ind}_2) := \forall\, t : \mathit{Train}.\ & \mathit{pc} \neq \mathit{InitState} \wedge \mathit{pos}(t) \geq \mathit{length}(\mathit{segm}(t)) - d \\ & \rightarrow \mathit{spd}(t) \leq \mathit{lmax}(\mathit{nexts}(\mathit{segm}(t)))\end{aligned}$$

The program counter *pc* is introduced in the translation process from COD to TCS and we use the constraint $\mathit{pc} \neq \mathit{InitState}$ to indicate that the system is not in its initial location. Thus, define (Inv) as the conjunction $(\mathsf{Init}) \wedge (\mathsf{Ind}_1) \wedge (\mathsf{Ind}_2)$. 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 Thm. 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 Cor. 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.

Verification of the timed train controller. Using the model checking approach from [19], we can automatically prove real-time properties of COD specifications. In this case, we use the approach only on the train controller part Train (Fig. 3). 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 specification. That is, we prove that (for an arbitrary train) the train position $curPos$ is never beyond its movement authority ma:

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

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.

4 Modular Verification for Complex Track Topologies

We now consider a complex track as described in Fig. 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).

Theorem 12. *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.* $\mathsf{next}_s^{\mathsf{ltrack}}$.

We assume that for each linear track ltrack we have one controller RBC^{ltrack} which uses the control protocol described in Sect. 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 $\mathsf{track_3}$ then the corresponding fields agree, i.e. whenever $s, \mathsf{next}_s^{\mathsf{track_i}}(s)$ are on $\mathsf{track_3}$, $\mathsf{next}_s^{\mathsf{track_1}}(s)$=$\mathsf{next}_s^{\mathsf{track_2}}(s)$=$\mathsf{next}_s^{\mathsf{track_3}}(s)$ (the same for prev_s, and for $\mathsf{next}_t, \mathsf{prev}_t$ on the corresponding tracks); (2) the update rules are compatible for trains jointly controlled.[9] Under these conditions,

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

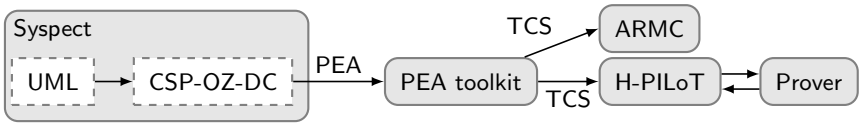

Fig. 4. Tool chain

proving safety for the complex track can be reduced to checking safety of linear train tracks with incoming and outgoing trains (for details cf. [8]).

Lemma 13. *A state s of the system is a model* $(P_\mathsf{t}, P_\mathsf{s}, \mathbb{R}, \mathbb{Z}, \{\mathsf{next}^{\mathsf{ltrack}}, \mathsf{prev}^{\mathsf{ltrack}}, \mathsf{next}_s^{\mathsf{ltrack}}, \mathsf{prev}_s^{\mathsf{ltrack}}\}_{\mathsf{track}\in\mathcal{L}} \cup \{\mathsf{segm}, \mathsf{train}, \mathsf{pos}, ...\})$*, 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.

(a) and (b) also hold if we consider initial *states (i.e. states satisfying the initial conditions) and* safe *states (i.e. states satisfying the safety conditions in the invariant* Inv*). Similar properties hold for parallel actions and for transitions.*

Theorem 14. *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 Sect. 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 $\{RBC^{\mathsf{ltrack}_i} \mid i \in I\}$.*

5 From Specification to Verification

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

Tool chain. The tool chain is sketched in Fig. 4. In order to capture the systems we want to verify, we use the COD front-end Syspect (cf. Sect. 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[10] 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

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

(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 invariance of a safety condition fully automatically by checking its invariance for each transition update (cf. Sect. 3.1). Therefore, for each update, Syspect exports a file that is handed over to H-PILoT. The safety invariance 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.[11] 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 tests to ensure that the specifications are consistent.

	(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)

Table 1. Results

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. Sect. 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).

[11] 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.

6 Conclusion

We augmented existing techniques for the verification of real-time systems to cope with rich data structures like pointer structures. 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. Sect. 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 also 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 Sect. 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, Ernst-Rüdiger Olderog and the anonymous referees 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. Reports of SFB/TR 14 AVACS No. 66, SFB/TR 14 AVACS (2010), www.avacs.org
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)