Combining Look-ahead Design-time and Run-time Control-synthesis for Graph Transformation Systems

He Xu, Sven Schneider(✉), and Holger Giese

Hasso Plattner Institute, University of Potsdam, Potsdam, Germany
{he.xu,sven.schneider,holger.giese}@hpi.de

Abstract. The correct operation of safety-critical cyber-physical systems is crucial. However, such systems often feature a large variability of start configurations, an intractably large state space, a high degree of uncertainty, or inherently unsafe behavior. A model of the expected system behavior starting in the current state can be used by look-ahead controllers to derive control decisions to avoid paths to safety violations when possible. However, the computational effort for deriving and analyzing the future system behavior is exponential in the look-ahead.
In this paper, we employ Graph Transformation Systems (GTSs) for the modeling of expected system behavior. We then combine design-time and run-time control synthesis based on Supervisory Control Theory (SCT) achieving an exponential cost-reduction for a given controller look-ahead. For a fixed required reaction time of controllers, much longer look-aheads may therefore be employed. To illustrate and evaluate our approach, we consider a system where shuttles must avoid collisions with ambulances at level crossings.

Keywords: cyber-physical systems, self-adaptive systems, supervisory control, model-predictive control, runtime verification, bounded model checking

1 Introduction

Cyber-physical systems in which software components operate in a physical environment often encompass complex concurrent behavior. The development or synthesis of such control software achieving a given set of goals while also ensuring the satisfaction of a given safety-specification is crucial. In model-predictive control, a model of the expected system behavior is employed to obtain look-ahead controllers. Such controllers derive control decisions based on the set of all behavior sequences of a chosen look-ahead length starting in the current state. However, the set of such behavior sequences is exponential in the look-ahead length limiting the look-ahead to values allowing admissible reaction times.

As a running example, we consider a variation of the *RailCab* system from [38, 30]. In this system, shuttles navigate on a large-scale track topology, which intersects with a road topology at level crossings. Ambulances, which can be

D. Beyer and A. Cavalcanti (Eds.): FASE 2024, LNCS 14573, pp. 77–100, 2024.
https://doi.org/10.1007/978-3-031-57259-3_4

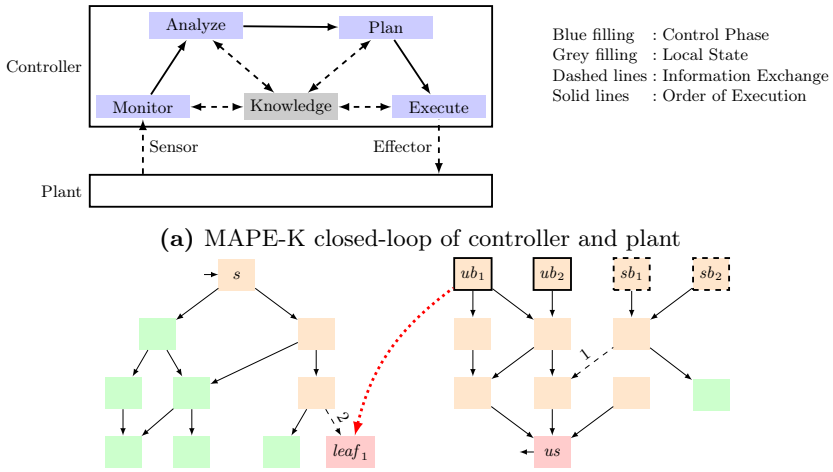
monitored by shuttles with a certain degree of uncertainty, navigate on the road topology and may traverse level crossings. The shuttle control to be derived, must avoid collisions with ambulances when possible by adjusting the speed of the shuttle taking potential ambulance behavior into account. To focus on our approach and to simplify our presentation, we reduce the possible number of steps of actors in the system model by employing a small topology fragment with one level crossing, a single shuttle, and one ambulance.

Besides run-time efficiency, controller synthesis approaches for cyber-physical systems must solve an array of further problems. *P1 (Sets of Start States):* The start state of the system is often not precisely known requiring the consideration of a large or even infinite set of start states. These start states may differ in rigid components but also in the number, the state, and the interconnection of active components. For our running example, the underlying rigid topology and the location of shuttles and ambulances on this topology may vary greatly. *P2 (State space explosion):* Even when selecting a single start state, the state space of the system is often intractably large or even infinite because all steps of all components must be captured in the system model. *P3 (Uncertainty):* The uncontrolled part of the system can often not be modeled faithfully at design time due to uncertainty. For example, uncertainty arises due to behavioral or configuration adaptation as well as from unknown, unreliable, or unpredictable components/actors (such as humans) performing additional steps that cannot be foreseen at design time or fail to perform such steps [45]. *P4 (Unsafe Systems):* Avoidance of unsafe states is not always feasible due to uncertainty or in contexts where unsafe states cannot be avoided by control at all.

For the modeling of the expected future system behavior, we employ Graph Transformation Systems (GTSs), which can be used when system states can be captured by graphs and when the steps of the involved components can be captured using local graph modifications. In the past, various GTS-variants have been developed and employed for the modeling, design, and analysis of such systems in an abundance of publications such as [19, 20, 21, 18, 29, 22, 30, 49, 48, 33] focusing on different system aspects and requirements.

To accommodate for these problems (discussed in more detail in the subsequent section), we propose a model-driven approach based on GTSs and the MAPE-K control framework where we employ a sliding window technique considering actor-specific state fragments to reduce the computational effort (problems P1 and P2) and combine design-time control synthesis with run-time control synthesis as a look-ahead extension technique to efficiently obtain best-effort control (to tackle problems P3 and P4). Both, at design-time and run-time, we employ an extension of Supervisory Control Theory (SCT) with priorities for the synthesis of controllers where the uncontrolled system is modeled using an extension of GTSs with controllability notions.

This paper is structured as follows. In section 2, we discuss our conceptual approach in the context of the MAPE-K framework including the sliding window technique. In section 3, we consider related work. In section 4, we present our extension of SCT with priorities. In section 5, we integrate controllability



(a) MAPE-K closed-loop of controller and plant

(b) Runtime Model. A Bounded Forward State Space from the current system state s (left). A Bounded Backward State Space leading to the unsafe state us (right). The safe boundary $\{sb_1, sb_2\}$ from which us can be avoided by preventing step 1. The unsafe boundary $\{ub_1, ub_2\}$ from which us cannot be avoided. The state $leaf_1$ contains ub_1 (indicated by the dotted arrow) leading to the prevention of step 2.

Fig. 1. Overview of MAPE-K-based approach

notions into the GTS framework and present our running example. In section 6, we discuss control synthesis at design-time. In section 7, we discuss control synthesis at run-time based on the design-time results. In section 8, we evaluate our approach for a larger case study. Finally, in section 9, we conclude the paper and provide an outlook on future work.

2 MAPE-K Closed-Loop Approach

Software being executed in a cyber-physical system on a device often follows (at least implicitly) the MAPE-K closed-loop design [53, 1] depicted in Figure 1a developed for systems with a high degree of complexity, uncertainty, and dynamicity. Such software interacts with its context in that system via sensors and effectors and keeps a Runtime Model (RTM) to store its local state across its looped executions. It executes (a) the monitoring phase to react to sensor information by updating the RTM accordingly, (b) the analysis phase to determine the impact of the most recent events on its options to achieve its control goals, (c) the planning phase to derive a control plan satisfying suitable quality standards, and (d) the execution phase to send events to the effectors to implement the steps of the derived control plan. Ideally, such a MAPE-K control architecture adapts to unexpected situations at run-time in an ad-hoc manner.

In our approach, the RTM (see Figure 1b) contains (a) a Bounded Forward State Space (BFSS) from the current system state s (derived and maintained

at run-time) and (b) a Bounded Backward State Space (BBSS) from unsafe states *us* (derived at design-time). Both of theses state spaces are (similarly to bounded model checking [50]) derived from the GTS capturing the expected system behavior. Moreover, the RTM contains the controllers derived from these two state spaces, which capture for each depicted state the exiting steps that the shuttle may perform. At run-time the controller obtained from the BFSS and the BBSS are combined by attempting to identify boundary graphs of the BBSS in the leaf states of the BFSS. For a BFSS and BBSS of depth n and k, this combination grants an effective look-ahead of $n+k$ to the controller. Clearly, the look-ahead should be maximized (taking other aspects such as required response time into account) to provide the controller synthesis procedure with as much information as possible to avoid the execution of overly conservative behavior (such as unnecessarily slowing down the shuttle). Not employing a BBSS only constructing a BFSS of depth $n+k$ to achieve the same look-ahead $n+k$ would be exponentially more expensive and, moreover, this additional cost would be incurred at run-time whereas at least the BBSS is obtained in our approach at design-time rendering its cost of construction negligible.

In our approach, the four MAPE phases are as follows.

- *Monitor phase:* when the controller is informed via its sensors about a state change from the BFSS root s to state s', it selects s' as the new root of the BFSS. Unless the step to s' was not expected due to uncertainty, s' is already one of the successors of s contained in the BFSS.
- *Analysis phase:* States of the BFSS unreachable from s' are removed and the GTS model is used to re-extend the BFSS to the chosen depth n. To identify states to be avoided, all leaf states of the BFSS are checked for occurrences of unsafe boundary states of the BBSS. Finally, the run-time controller is then adjusted to the modified BFSS by selecting steps to be prevented that would lead to the states to be avoided.
- *Planning phase:* The controller can then plan the execution of any controllable step exiting the new root state s' of the BFSS (in the running example, these steps are the steps of the shuttle) or let the plant perform the next step.[1]
- *Execute phase:* If a step has been selected in the planning phase, this step is send for execution to the corresponding effector (in the running example, a hardware controller of the shuttle will receive and implement such a signal).

The worst-case controller response time depends on the time required for (a) the full reconstruction of the BFSS and the corresponding controller synthesis thereon (upon an occurrence of an unexpected step) and (b) the identification of leaf states of the BFSS containing unsafe boundary states of the BBSS. The

[1] The absence of such controllable steps does not indicate a problem as the controller may just not need to change the behavior of the agent (e.g., the shuttle may already be driving at the desired speed) but, in the considered time-abstract setting, the absence of any step implies that no control strategy guaranteeing the avoidance of unsafe states could be obtained. In this case, fallback behavior such as not modeled emergency maneuvers or decisions by the environment on uncontrollable events may still result in the avoidance of unsafe states.

usage of the BBSS exponentially reduces the computational effort for (a) as discussed but, regarding (b), it also requires that the leaf states of the BFSS need to be checked against a potentially large number of unsafe boundary states instead of only the unsafe states. In our evaluation in section 8, we measure and further discuss these effects for a considered case study.

As mentioned in the introduction already, we employ a sliding window approach reducing the size of the BFSS and BBSS to be constructed. Instead of assuming that each agent maintains a perspective on the entire system state, we adopt the technique from [30] where, in a compositional approach, agent-specific scopes are used. On the one hand, this greatly reduces the number of steps (and thereby the size of the BFSS and BBSS) as only a small number of agents will be typically in the view range of an agent. On the other hand, a smaller view range may result (closely related to the look-ahead) in an overly conservative controller behavior. Besides mitigating the effect of state space explosion, this sliding window approach has the additional advantage that start states must only be determined for each actor individually and not globally. Intuitively, each system step must be followed by suitable postprocessing to update the reached state to the view range of the actor. These postprocessing steps are part of the system model and therefore define changes in the context of the agent to which the controller must suitably respond. In our evaluation in section 8, we further discuss this sliding window technique as we abstract from it in our running example to focus on controller synthesis via BFSS and BBSS.

3 Related Work

Model checking [2] is often inadequate for complex systems due to the state space explosion problem and uncertainty. Bounded Model Checking (BMC) [50, 24, 25] has been devised to reduce analysis costs providing, however, weaker guarantees and no support for uncertainty.

When formal fully-automatic verification is infeasible, Runtime Verification also called Runtime Monitoring [28] is an approach for monitoring the system's states and steps at run-time for notable behavior such as violations of invariants that require a manual or automatic response. However, without look-ahead capabilities, potential near-future unsafe states cannot be detected. Therefore, some RV approaches such as [45, 23, 15, 32, 52, 16] integrate a behavioral model describing expected future evolutions of the system. In [45], the expected future evolutions of a Timed Automata (TA) are analyzed at run-time using BMC. In [15], Deterministic Timed Markov Chains modeling the system are analyzed at design-time to obtain expressions on step-probabilities that will become available at run-time to make probability-maximizing decisions at run-time by evaluating the expressions at run-time instead of performing computationally expensive analysis. In [23], a run-time statistical model checking component has been integrated into a self-adaptive system. However, these approaches also rely on BMC and thereby suffer from state space explosion and in some cases such as [45, 15] also from being unable to react to uncertain events.

The approach of k-induction [26, 11] that has been adopted for variants of GTSs in [47, 48, 3] establishes state invariants by symbolically applying GT rules backwards from unsafe states to accumulate context capturing why and how the symbolic violation could be reached. This approach is thereby a symbolic version of backward BMC. We use a similar approach in this paper tackling the problem of a large number of undesirable backward steps constructed by k-induction.

A combination of forward and backward BMC similar to our approach for the analysis of Hybrid Automata in [54] applies depth first search forward and backward in parallel to find paths to unsafe states for Hybrid Automata with complex state space structure.

SCT as established in [40, 41, 39] for capturing, analyzing, and synthesizing supervisory control when the controllers, the plants, and their closed loops are given by regular languages over events (see also [27, 46] for an in-depth introduction and a discussion of derived approaches) has to our knowledge not been combined with event-priorities. However, priorities have been used to combine supervised modules preventing blocking situations in [6, 7]. Also, approaches in the Model Predictive Control domain (see [51] for a survey) employ models to predict the future system behavior as in our approach but focus usually on continuous time systems minimizing costs as in [4, 5] and have not been combined with SCT to the best of our knowledge. Besides the approach to distinguish between controllable and uncontrollable events as customary in SCT, other approaches of identifying actions of different actors and capturing interactions among such actors in the GT domain include [9] but also SCT for TA (related to [45] above) has been considered in [43, 42]. [35, 36, 34, 33] where a safety constraint has already been violated due to uncertainty or adversarial effects requiring the derivation and execution of recovery mechanisms.

4 Priority-aware Supervisory Control Theory

We recall SCT as introduced in the seminal work of Ramadge and Wonham [40, 41, 39] in which the closed loop is given by the event-synchronizing composition of controller and plant. To provide the essentials of this approach in our notation and to extend this approach with the concept of event priorities, we introduce a variant of Labeled Transition Systems (LTSs) extending finite automata thereby capturing regular languages over an event alphabet as considered in standard SCT. In such an LTS, events are grouped into controllable and uncontrollable events (cf. the MAPE-K closed-loop in Figure 1a), which are executed by the controller (e.g., signals to effectors) and the plant (e.g., signals from sensors). The controller may restrict the execution of controllable events in the closed-loop.

We aim at controller synthesis such that event-prevention ensures that the closed-loop avoids undesirable states (this notion is formalized below as non-blockingness) and no steps executing uncontrollable events have been prevented at the model level (this notion is formalized below as controllability) while not preventing event executions unnecessarily to retain the highest possible

degree of freedom for further control steps.[2] We equip events with a priority as motivated in the next section by our running example: steps executing (un)controllable events are then only enabled when no steps executing higher-priority (un)controllable events are enabled (i.e., priorities are checked within the two groups of controllable and uncontrollable events separately).

Definition 1 (Labeled Transition System (LTS)). *A Labeled Transition System (LTS) Γ contains the following components.*

- $\mathsf{states}(\Gamma)$ *contains all states and its subsets* $\mathsf{start}(\Gamma)$, $\mathsf{safe}(\Gamma)$, *and* $\mathsf{unsafe}(\Gamma)$ *contain the start, safe, and unsafe states.*
- $\mathsf{events}(\Gamma)$ *contains the controllable and uncontrollable events* $\mathsf{eventsC}(\Gamma)$ *and* $\mathsf{eventsUC}(\Gamma)$.
- $\mathsf{prio}(\Gamma) : \mathsf{events}(\Gamma) \rightarrow \mathbf{N}$ *assigns a priority to each event.*
- $\mathsf{steps}(\Gamma) \subseteq \mathsf{states}(\Gamma) \times \mathsf{events}(\Gamma) \times \mathsf{states}(\Gamma)$ *is a set of event-labelled steps.*

Moreover, Γ_1 is a sub-LTS of Γ_2, written $\Gamma_1 \leq \Gamma_2$, when the components of Γ_1 are contained in the corresponding components of Γ_2 and the reversed LTS $\mathsf{rev}(\Gamma)$ *is obtained by reversing* $\mathsf{steps}(\Gamma)$ *and swapping* $\mathsf{start}(\Gamma)$ *and* $\mathsf{unsafe}(\Gamma)$.

The priority-resolved LTS is obtained by omitting all controllable/uncontrollable steps disabled by higher-priority controllable/uncontrollable steps. Only the paths through this priority-resolved LTS can actually be observed.

Definition 2 (Priority-resolved LTS). *For an LTS Γ and a set of events E, $\Gamma' = \mathsf{resPrio}(\Gamma, E)$ is the largest sub-LTS of Γ such that for all $(s, e_1, s_1) \in \mathsf{steps}(\Gamma')$ with $e_1 \in E$ there is no $(s, e_2, s_2) \in \mathsf{steps}(\Gamma')$ with $e_2 \in E$ and $\mathsf{prio}(\Gamma')(e_2) > \mathsf{prio}(\Gamma')(e_1)$. Then, the priority-resolved LTS of Γ is given by* $\mathsf{resPrio}(\Gamma) = \mathsf{resPrio}(\mathsf{resPrio}(\Gamma, \mathsf{eventsUC}(\Gamma)), \mathsf{eventsC}(\Gamma))$.[3]

A controller Γ_C to be synthesized for a given plant Γ_P is a sub-LTS of Γ_P and, hence, the event-synchronizing closed loop of Γ_C and Γ_P is just Γ_C.

The notion of controllability requires that the controller cannot prevent uncontrollable events that the plant can execute.

Definition 3 (Controllability). *A plant Γ_P and a controller $\Gamma_C \leq \Gamma_P$ satisfy controllability, if every path π of* $\mathsf{resPrio}(\Gamma_C)$ *that can be extended by* $\mathsf{resPrio}(\Gamma_P)$ *with a step executing an uncontrollable event $u \in \mathsf{eventsUC}(\Gamma_P)$ can be extended by* $\mathsf{resPrio}(\Gamma_C)$ *with a step executing u as well.*

The notion of non-blockingness requires the liveness property that the closed loop may eventually reach a safe state from any of its states. In our approach, we define unsafe states as those violating a state invariant and safe states as those not having paths to any unsafe states.

Definition 4 (Non-blockingness). *A plant Γ_P and a controller $\Gamma_C \leq \Gamma_P$ satisfy non-blockingness, if every path π of* $\mathsf{resPrio}(\Gamma_C)$ *can be extended to a state in* $\mathsf{safe}(\Gamma_P)$.

[2] Note that controllers can only force certain events in a given state in this framework when all events executable from that state are controllable (differing from, e.g., [55]).

[3] Note that, in general, $\mathsf{resPrio}(\Gamma) \neq \mathsf{resPrio}(\Gamma, \mathsf{events}(\Gamma))$.

For the case of controllers and plants generating regular languages considered here, admissible controllers satisfying controllability and non-blockingness are closed under arbitrary unions [40, 41, 39, 27, 46]. Desired controllers are therefore defined as those admissible controllers that result in the largest closed loops in terms of sets of executable event sequences. Admissible controllers are also closed under arbitrary union in the presence of event priorities because the union of controllers will result in a controller that favors the highest priority steps from any of the controllers and, moreover, LTSs are memoryless (beyond their current state) implying that choosing higher priority steps from different controllers can not lead to states not traversable using any of the controllers. However, only the priority resolved versions of synthesized controllers for which the classic results from [40, 41, 39, 27, 46] readily apply are to be used anyway.

Following SCT, the first controller candidate is the plant LTS Γ. This candidate is then incrementally refined by preventing events enforcing controllability and non-blockingness least-restrictively until an admissible controller $\mathsf{control}(\Gamma)$ is obtained (closedness under arbitrary union also implies that the order in which violations of controllability and non-blockingness are resolved is insignificant). Note that this fixed-point procedure supports also cyclic LTSs in general (in which, as usual, loops may delay the visiting of safe states indefinitely as opposed to [55]). To handle the case with priorities, we resolve priorities among uncontrollable events before applying the fixed-point procedure and resolving priorities of remaining controllable steps afterwards to obtain the priority-aware controller $\mathsf{pControl}(\Gamma)$.

Definition 5 (Priority-Aware Controller). *An LTS Γ induces the LTS $\Gamma' = \mathsf{control}(\Gamma)$ by adapting Γ as follows:*[4]

- $\mathsf{steps}(\Gamma')$ *is the largest subset of* $\mathsf{steps}(\Gamma)$ *such that for each* $(s, e_1, s_1) \in \mathsf{steps}(\Gamma')$ *(non-blockingness) there is some path from* s_1 *to a state in* $\mathsf{safe}(\Gamma')$ *using steps in* $\mathsf{steps}(\Gamma')$ *and (controllability) when* $(s_1, u_2, s_2) \in \mathsf{steps}(\Gamma)$ *is a step using an uncontrollable event* u_2 *from* $\mathsf{eventsUC}(\Gamma)$ *then* (s_1, u_2, s_2) *is also a step in* $\mathsf{steps}(\Gamma')$.

Moreover, $\mathsf{pControl}(\Gamma) = \mathsf{resPrio}(\mathsf{control}(\mathsf{resPrio}(\Gamma, \mathsf{eventsUC}(\Gamma))), \mathsf{eventsC}(\Gamma))$ *is the priority-aware controller for Γ.*

As an example for controller synthesis, consider the LTS in Figure 2 representing an uncontrolled plant and the priority-aware controller synthesized for it.[5] First, to resolve blocking at s_4, the controllable priority 2 event c_2 from s_0 is prevented enabling the priority 1 event c_1 from s_0. Second, to resolve blocking at s_3, the uncontrollable event uc_3 from s_1 is prevented. Third, to resolve non-controllability at s_1, the controllable priority 1 event c_1 from s_0 is prevented enabling the priority 0 event uc_1 from s_0. The resulting controller will only contain the path from s_0 to s_2 executing the event uc_1. Note that maintaining the steps of all priorities in the LTS simplifies controller synthesis since the effect of preventing controllable events (such as c_2 and c_1) becomes apparent immediately without

[4] For brevity, we omit here the removal of unreachable states from Γ'.

[5] When resolving priorities among uncontrollable events and later among controllable events no steps are removed in this example.

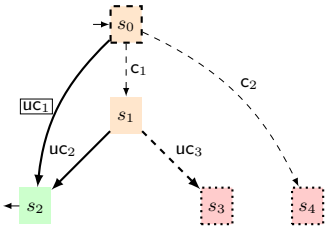


Fig. 2. Example of controllability and non-blockingness. The unsafe states $\{s_3, s_4\}$ are given in red with dotted border, the safe state s_2 is given in green with exiting arrow symbol, the remaining orange states have paths to unsafe states, the start state s_0 has an entering arrow symbol, the bold steps execute the uncontrollable events uc_i, the non-bold steps execute the controllable events c_i, the dashed steps have been prevented, the event c_2 has priority 2, the event c_1 has priority 1, the other events have priority 0, and only the boxed event uc_1 can be executed since the steps executing $\{\mathsf{c}_1, \mathsf{c}_2\}$ have been prevented.

the need to derive such steps intermittently for then enabled steps (e.g., only the step executing c_2 was enabled initially due to its priority) decoupling LTS generation and control synthesis.

Note that $\mathsf{control}(\mathsf{resPrio}(\Gamma)) \neq \mathsf{resPrio}(\mathsf{control}(\Gamma))$ in general because first resolving the priorities restricts the possible controllers to be synthesized. For example, first resolving priorities in Figure 2 would remove the step with the event uc_1, which would otherwise be the only remaining step.

5 Control-oriented Graph Transformation

We first introduce control-oriented GTSs before discussing the modeling of our running example using this formalism.

To ease presentation, we employ the simple class of typed directed graphs (short graphs) (see [12, 13, 14] for details). In our running example, we employ the type graph TG from Figure 3a, which can be understood to be a simple UML class diagram, and graphs, which can be understood to be simple UML object diagrams. In visualizations of graphs such as Figure 3b, types of nodes are indicated by their names (i.e., S_i and T_i are nodes of type Shuttle and Track), names of edges are omitted, types of edges are only given when required to avoid ambiguity (the only edge types with equal source and target node types are *fast*, *slow*, and *halt*). We denote monomorphisms (monos) from graph H to graph H' mapping nodes and edges injectively by $f : H \hookrightarrow H'$.

To introduce control-oriented GTSs, we first introduce GT rules used to derive GT steps between graphs. A Graph Transformation (GT) rule ρ consists of two monos $\ell : K \hookrightarrow L$ and $r : K \hookrightarrow R$ describing the removal and addition of elements and a set N of monos $n_i : L \hookrightarrow N_i$ of Negative Application Conditions

(NACs) describing forbidden extensions of L.[6] We use the abbreviation $\mathsf{lhs}(\rho) = L$ later on. In visualizations of GT rules (see Figure 3), we use an integrated notation in which L, K, and R are given in a single graph where graph elements marked with $\ominus$ are from $L - K$ and will be deleted, graph elements marked with $\oplus$ are from $R - K$ and will be created, and where all other graph elements are in K and will be preserved. When NACs are present, they are given on the left side of the $\triangleright$ symbol. For example, consider the GT rule in Figure 3c which preserves the ambulance and shuttle nodes A_1 and S_1, removes the edge from S_1 to A_1, creates an edge from A_1 to S_1, and is only applicable when A_1 has no edge to some road node R_1.

We now introduce our novel notion of control-oriented GTSs. Such a GTS S contains a set $\mathsf{start}(S)$ of start graphs, a set $\mathsf{unsafe}(S)$ of unsafe graphs representing violations of invariants, a set $\mathsf{rules}(S)$ of GT rules with the subsets of controllable and uncontrollable GT rules $\mathsf{rulesC}(S)$ and $\mathsf{rulesUC}(S)$, and a mapping $\mathsf{prio}(S)$ assigning a natural number as a priority to each GT rule. Note that, similarly as in our presentation of SCT in section 4, we assign priorities to GT rules and group them into controllable/uncontrollable GT rules capturing which steps can/cannot be prevented by the controller to be synthesized.

GT steps $G \Rightarrow_\sigma G'$ from a graph G to a graph G' are labeled with a pair $\sigma = (\rho, m)$ consisting of a GT rule ρ and a match $m : \mathsf{lhs}(\rho) \hookrightarrow G$ identifying an occurrence of $\mathsf{lhs}(\rho)$ in G. The match m must satisfy the requirement that there is no NAC $n_i : \mathsf{lhs}(\rho) \hookrightarrow N_i$ contained in ρ for which some $m'_i : N_i \hookrightarrow G$ satisfying $m'_i \circ n_i = m$ exists. The graph G' is then constructed from G via the usual Double Pushout (DPO) diagram (see [12, 13, 14] for a details).

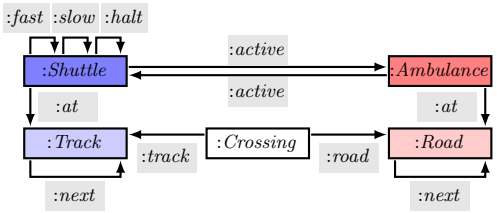
A GTS induces a forward LTS by deriving GT steps from already included graphs and adds these steps as well as their target states in the resulting LTS. Note that we merely propagate the priorities of the GT rules into the constructed LTS instead of enforcing them by excluding lower-priority steps when higher-priority steps are present.

Definition 6 (Forward LTS of a GTS, BFSS). *A GTS S induces the unique LTS $\Gamma = [\![S]\!]$ as follows:*

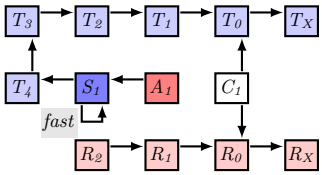
- $\mathsf{states}(\Gamma)$ *contains* $\mathsf{start}(\Gamma)$ *and the target states of all steps in* $\mathsf{steps}(\Gamma)$.
- $\mathsf{start}(\Gamma)$ *contains the graphs from* $\mathsf{start}(S)$.
- $\mathsf{safe}(\Gamma) \subseteq \mathsf{states}(\Gamma)$ *contains the graphs from which* $\mathsf{unsafe}(\Gamma)$ *can't be reached.*
- $\mathsf{unsafe}(\Gamma) \subseteq \mathsf{states}(\Gamma)$ *contains the graphs G into which a mono $t : H \hookrightarrow G$ from some graph $H \in \mathsf{unsafe}(S)$ exists.*
- $\mathsf{eventsC}(\Gamma)$ *and* $\mathsf{eventsUC}(\Gamma)$ *contain the step labels $\sigma = (\rho, m)$ of the steps in* $\mathsf{steps}(\Gamma)$ *where $\rho \in \mathsf{rulesC}(S)$ and $\rho \in \mathsf{rulesUC}(S)$.*
- $\mathsf{prio}(\Gamma)(\rho, m) = \mathsf{prio}(S)(\rho)$ *assigns the priority of the used GT rule ρ.*
- $\mathsf{steps}(\Gamma)$ *is the least relation containing all GT steps* from *states in* $\mathsf{states}(\Gamma)$.

Moreover, the BFSS of depth n, denoted $[\![S]\!]_n$, is the largest sub-LTS of $[\![S]\!]$ in which all paths starting in $\mathsf{start}(\Gamma)$ *through distinct states have length $\leq n$.*

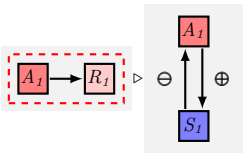
[6] Our approach is orthogonal to the use of more expressive notions of application conditions such as nested graph conditions [18, 14, 10].



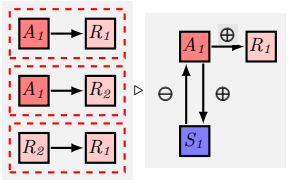
(a) Type graph



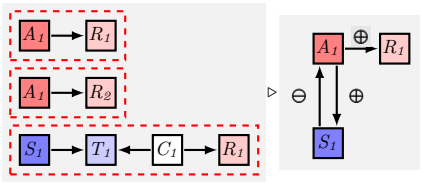
(b) Current graph



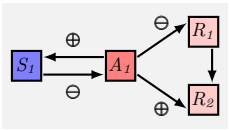
(c) GT rule ρ_{acp} for postponing ambulance creation.



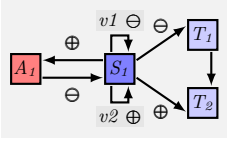
(d) GT rule ρ_{ace} for expected ambulance creation at the farthest road segment from the crossing.



(e) GT rule ρ_{acu} for unexpected ambulance creation at some road segment (not on the crossing when there is a shuttle already).

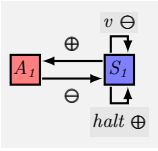


(f) GT rule ρ_{a} moving the ambulance to the next road.



$(v1, v2) \in \{$
$(fast, fast),$
$(slow, fast),$
$(fast, slow),$
$(slow, slow),$
$(halt, slow)\}$

(g) GT rules ρ_{ff}, ρ_{sf}, ρ_{fs}, ρ_{ss}, and ρ_{hs} resulting in a fast or slow shutle on the next track.



$v \in \{slow, halt\}$

(h) GT rules ρ_{sh} and ρ_{hh} resulting in a halted shuttle on the same track.

GT rules	controllable?	priority	S_{FE}	S_{FU}	Figure
ρ_{acp}	no	0	yes	yes	Figure 3c
ρ_{ace}	no	0	yes	no	Figure 3d
ρ_{acu}	no	0	no	yes	Figure 3e
ρ_{a}	no	0	yes	yes	Figure 3f
ρ_{fs}, ρ_{ss}, ρ_{hs}	yes	1	yes	yes	Figure 3g
ρ_{ff}, ρ_{sf}	yes	2	yes	yes	Figure 3g
ρ_{sh} ρ_{hh}	yes	0	yes	yes	Figure 3h

(i) Overview of the GT rules used in the GTSs S_{FE} and S_{FU}.

Fig. 3. Details on the running example.

We now discuss the modeling of our running example, which is a simplification of the case study considered in our evaluation in section 8. We model shuttles driving on a track topology where subsequent tracks are connected using *next* edges as in Figure 3b. The driving speed of each shuttle is either fast, slow, or halt (as marked using *fast*, *slow*, or *halt* loops). Level crossings (where track and road topology intersect) are indicated by the node type *Crossing* and are connected to the corresponding track and road segments. Ambulances may appear and drive on the road topology including the level crossings.

The graph in Figure 3b represents the current view of the shuttle on the system state. The ambulance A_1 is not yet connected to a road meaning that it can be ignored by the shuttle at this point. Ambulance and shuttle perform steps alternatingly by switching the directed edge between them in each step to ensure a certain level of fairness since the system would otherwise be fundamentally unsafe as the shuttle could not rule out collisions anymore. The edge from the ambulance to the shuttle indicates that the shuttle will perform the next step.

Shuttles may maintain their speed (events ff, ss, and hh) or switch between fast and slow (events fs and sf) as well as between slow and halt (events sh and hs), modeling the stopping and acceleration distance. These seven driving speed transitions are controllable for the shuttle controller but all steps of ambulances are uncontrollable. To allow the shuttle to make timely control decisions, an ambulance detection mechanism informs the shuttle when ambulances are two roads ahead of an upcoming level crossing (i.e., an ambulance would be detected in Figure 3b when it enters the road R_2). We derive shuttle control assuming that this detection mechanism is reliable but analysis will reveal partial robustness against unreliability in situations where ambulances are detected first on the closer road segments R_1 or even R_0. Note that shuttle and ambulance performing steps alternatingly will result in violations of non-blockingness when the controller prevents all controllable steps of the shuttle in a given state, which is thereby implicitly excluded as well.

We use GT rule priorities to model that the shuttle prefers faster driving speeds over slower driving speeds. Therefore, without preventing any steps, the shuttle will maintain its fast speed.

We now discuss the GT rules used in these GTSs in more detail. Again, shuttle and ambulance steps alternate as implemented by switching the direction of the edge between them in every GT rule. When its the ambulances turn, the GT rules ρ_{ace}, ρ_{acu}, and ρ_{acp} are applicable when the ambulance has no edge to some road segment yet and the GT rule ρ_{a} is used otherwise. The GT rule ρ_{ace} models the expected creation of the ambulance by creating an edge from the ambulance to the road R_2 in Figure 3b (the three NACs check that A_1 is not yet on R_1, that A_1 is not yet on some other road, and that the matched road R_1 has no predecessor). The GT rule ρ_{acu} models the unexpected creation of the ambulance by creating an edge from the ambulance to an arbitrary road unless this road is at the level crossing with a shuttle being already located there as well (the three NACs check that A_1 is not yet on R_1, that A_1 is not yet on some other road, and that S_1 is not on a track connected by a crossing to R_1). The

GT rule ρ_{acp} models the case that the ambulance is not yet created meaning that ambulance detection is postponed (the NAC checks that the ambulance is not yet on a road). Lastly, the GT rule ρ_{a} models the moving of a detected ambulance to the next road segment (by removing the edge from A_1 to the current road segment R_1 and creating such an edge to the road segment R_2 reached). When its the shuttles turn, the GT rules ρ_{ff}, ρ_{fs}, ρ_{sf}, ρ_{ss}, ρ_{sh}, ρ_{hs}, and ρ_{hh} are used. The GT rules ρ_{sh} and ρ_{hh} do not move the shuttle to the next track while the other GT rules do so. Here, the movement of the shuttle is implemented as for the GT rule ρ_{a} by deleting and creating an edge and the driving speed transitions are encoded by deleting and creating the driving speed loop at the shuttle.

In our running example, we first consider the GTS S_{FE} with expected ambulance detection: for this GTS, we employ the graph from Figure 3b as start graph, use 10 of the 11 GT rules from Figure 3, split GT rules into controllable and uncontrollable GT rules, and employ priorities as listed in Figure 3i. In particular, when its the ambulances turn, each enabled GT rule has the same priority 0 making all steps derivable using the GT rules ρ_{ace} and ρ_{acp} viable. When its the shuttles turn, GT rules setting the speed to halt, slow, and fast have priorities 0, 1, and 2 favoring a faster driving speed. Also, the GT rules for slowing down or remaining halted (ρ_{fs}, ρ_{sh}, and ρ_{hh}) cannot be prevented as this would lead to a violation of non-blockingness as discussed. Additionally, we consider a second GTS S_{FU} in which ambulances are possibly detected closer or on the level crossing: this GTS differs from S_{FE} by replacing the GT rule ρ_{ace} with ρ_{acu} for detecting an ambulance, which may result in up to four steps detecting the ambulance on any of the four road segments.

In the considered GTSs, only a finite number of graphs can be reached and, in the remainder, we represent each graph using an element of $\{✘, 0, 1, 2, ✔\} \times \{0, 1, 2, 3, 4, ✔\} \times \{\mathsf{f}, \mathsf{s}, \mathsf{h}\} \times \{\mathsf{s}, \mathsf{a}\}$ where *(a)* ✘ means that the ambulance has not been detected yet, 0–2 is the distance of the ambulance to the crossing, and ✔ means that the ambulance has advanced beyond the crossing, *(b)* 0–4 is the distance of the shuttle to the crossing and ✔ means that the shuttle has advanced beyond the crossing, *(c)* f, s, and h is the driving speed of the shuttle, and *(d)* s or a means that the shuttle or the ambulance performs the next step. The start graph from Figure 3b is therefore represented by ✘4fs as the ambulance has not yet been detected, the shuttle is four tracks away from the level crossing, the shuttle is in fast driving speed, and the shuttle will perform the next step.

The 6 unsafe graphs in $\{0\} \times \{0\} \times \{\mathsf{f}, \mathsf{s}, \mathsf{h}\} \times \{\mathsf{s}, \mathsf{a}\}$ of the considered GTSs S_{FE} and S_{FU} all contain a shuttle and an ambulance on the level crossing but differ in the three possible driving speeds of the shuttle and the two cases of which entity performs the next step. While we specify the set of all unsafe states in our GTS by providing it explicitly, unsafe states could also be identified using advanced approaches such as nested graph conditions, Linear Temporal Logic [37], Computation Tree Logic [8, 2], or Metric Temporal Graph Logic [49].

The controller to be synthesized should force the shuttle to drive fast unless an ambulance is present, in which case the controller should ensure that the shuttle reaches the track T_1 with slow speed and then halts there until the ambu-

lance has passed the level crossing. The controller synthesized by our integrated approach results in this controller as discussed subsequently.

6 Design-time Control-synthesis

We now discuss design-time control synthesis based on *(a)* BBSS generation from unsafe states and *(b)* control synthesis based on SCT together resulting in an LTS with unsafe boundary to be avoided at run-time to avoid unsafe states and a safe boundary for which the LTS is a controller avoiding unsafe states.

For our running example, we start the BBSS generation using only two unsafe states $X_0 = \{\mathsf{00sa}, \mathsf{00fa}\}$ for presentation purposes. We depict the obtained BBSS in Figure 4, which is constructed by adding up to k steps backwards from X_0. From all additional states X_1, unsafe states in X_0 can be reached by construction; to derive viable alternative steps avoiding unsafe states, we include all missing forward steps from states in X_1 to additional states X_2. The states X_2 are by construction safe states (indicated by the exiting arrow symbol) of the resulting LTS from which unsafe states in X_0 cannot be reached (within k steps). The start states of the constructed backward LTS are the last states traversed on each backward path (indicated by the entering arrow symbol). These start states will be grouped into the safe and unsafe boundary in the next step.

We construct a controller from the BBSS given in Figure 4 by applying SCT. First, the two unsafe states 00sa and 00fa violate non-blockingness. To make these states unreachable, all five steps with one of them as a target are prevented resulting in a violation of non-blockingness at 01fs. To make this state unreachable, the step (11fa, a, 01fs) is prevented resulting in a violation of controllability at 11fa. To make this state unreachable, all three steps with 11fa as target are prevented. Due to event-priorities, only the boxed events can be actually executed. Intuitively, the depicted controller ensures that, in the presence of an ambulance approaching the upcoming level crossing, the shuttle will avoid collisions, e.g., by halting in state 01ha. When the ambulance is created unexpectedly closer to the crossing using ρ_{acu} in S_{FU}, the controller obtained here will fail since it would enter track T_1 with fast speed when no ambulance is detected reaching state ✘1fa and then not be able to halt in front of the level crossing when the ambulance is then unexpectedly detected on the level crossing in the next step reaching state 01fs.

Technically, we construct the BBSS for a given GTS relying on a secondary GTS called the *backward GTS*: We generate the BFSS for the backward GTS (according to Definition 6), reverse the obtained LTS (according to Definition 1), and then add the missing forward steps to safe states as explained above. For our running example, we employ the backward GTSs S_{BE} and S_{BU}, which can be obtained from their forward counterpart GTSs S_{FE} and S_{FU} by reversing their GT rules (see, e.g., [14, Lemma 3.14] for rule reversal based on the L operation) and switching the sets of unsafe and start graphs. The reason for using a backward GTS is a reduced size of the BBSS, since (not simply using rule reversal) modeling the backward GTS separately (while still ensuring that it agrees with

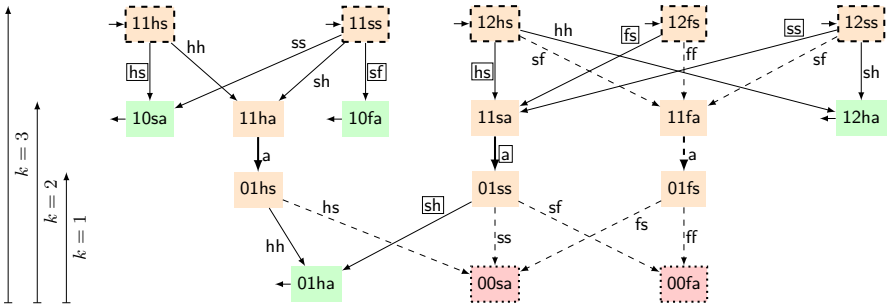


Fig. 4. Design-time controller synthesis based on BBSSs. We reuse the notation from Figure 2 for start states, unsafe states, safe states, potentially unsafe states, steps executing controllable/uncontrollable events, and prevented steps. The depicted BBSS of depth 3 and the resulting synthesized controller for the GTS S_{BE} (or the GTS S_{BU}) based for brevity on only two of the six unsafe states. The two unsafe states can be avoided resulting in an empty unsafe boundary.

the forward GTS as discussed in the next section) as in the case study considered in section 8 allows to enforce known system invariants (such as a minimum distance between level crossings or upper bounds of shuttles in certain areas) to reduce the number of derived steps.

Definition 7 (Backward LTS of a GTS, BBSS). *A (backward) GTS S induces the LTS $\Gamma = [\![S]\!]^{\mathsf{back}}$ by adapting $\Gamma' = \mathsf{rev}([\![S]\!])$ as follows:*
- $\mathsf{states}(\Gamma)$ *contains* $\mathsf{states}(\Gamma')$ *and the safe states* $\mathsf{safe}(\Gamma)$.
- $\mathsf{safe}(\Gamma)$ *contains the target graphs of all steps in* $\mathsf{steps}(\Gamma) - \mathsf{steps}(\Gamma')$.
- $\mathsf{steps}(\Gamma)$ *contains* $\mathsf{steps}(\Gamma')$ *and all GT steps* from *states in* $\mathsf{states}(\Gamma')$.

Moreover, the BBSS of depth k, denoted $[\![S]\!]^{\mathsf{back}}_k$, is the largest sub-LTS of $[\![S]\!]^{\mathsf{back}}$ in which all paths through distinct states ending in $\mathsf{unsafe}(\Gamma)$ *have length $\leq k$.*

We now apply the procedure pControl to the BBSS to derive the design-time controller. The unsafe boundary for which no suitable control could be derived is then given by all start states without an outgoing step and the safe boundary is given by the remaining start states (for which a controllable path to a safe state could be established).

Definition 8 (Design-time Controller). *If S is a (backward) GTS and $k \in \mathbf{N}$, then $\Gamma = \mathsf{pControl}([\![S]\!]^{\mathsf{back}}_k)$ is the design-time controller with unsafe boundary* $\mathsf{uBoundary}(S,k) = \{s \in \mathsf{start}(\Gamma) \mid \nexists (s,e,s') \in \mathsf{steps}(\Gamma)\}$.

The design-time controller for the BBSS in Figure 4 is constructed for $k = 3$ and has an empty unsafe boundary. However, when using $k = 2$ (removing the states in the first row and the safe states in the second row), we obtain a design-time controller with safe boundary $\{\mathsf{11ha}, \mathsf{11sa}\}$ and unsafe boundary $\{\mathsf{11fa}\}$.

As a further example, consider Figure 5 in which the uncontrollable event acu is used by the GTS S_{FU} for an unexpected shuttle detection leading to a non-empty unsafe boundary $\{\text{✘}\mathsf{1fa}\}$. In comparison, the controller obtained for S_{FE}

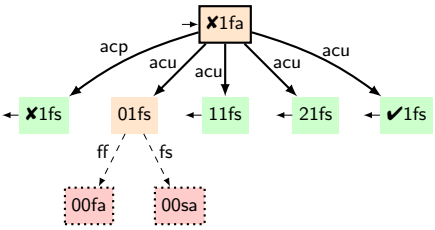


Fig. 5. Design-time controller synthesis with unexpected shuttle detection

not assuming unreliable ambulances detection as in the step $(\text{✘1fa}, \text{acu}, \text{01fs})$ is robust by also avoiding (according to Figure 4) the state 01fs preceding a collision in Figure 5. Moreover, this controller is robust against ambulances appearing unexpectedly directly on the crossing using the step $(\text{✘2fa}, \text{acu}, \text{02fs})$ unless the shuttle is already closer via step $(\text{✘1fa}, \text{acu}, \text{01fs})$. Also, when an ambulance appears one track ahead of the crossing, either no collision occurs (after step $(\text{✘2fa}, \text{acu}, \text{12fs})$) or the ambulance crashes into the shuttle (after step $(\text{✘1fa}, \text{acu}, \text{11fs})$).

7 Run-time Control-synthesis

At run-time, we employ a given (forward) GTS S_{FS} to derive the run-time controller as follows. First, we adapt S_{FS} into S'_{FS} by using the current state of the system as the unique start state and add $\mathsf{uBoundary}(S_{\mathsf{BS}}, k)$ to the set of unsafe states. Second, we construct the BFSS of depth n (which is assumed to be maintained throughout system execution as described in section 2) for S'_{FS}. Third, we apply SCT to obtain the least-restrictive controller.

Definition 9 (Run-time Controller). *If S is the GTS obtained from the forward GTS as the adjustment to the current system state and the unsafe boundary of the design-time controller and $n \in \mathbf{N}$, then $\Gamma = \mathsf{pControl}([\![S]\!]_n)$ is the run-time controller with leaf set* $\mathsf{leafs}(S, n) = \{s \in \mathsf{states}(\Gamma) \mid \nexists (s, e, s') \in \mathsf{steps}(\Gamma)\}$.

We now discuss in more detail how our run-time control synthesis obtains an effective look-ahead of $n + k$ steps towards unsafe states given by the n steps of Γ and the k steps of the design-time BBSS.[7] To this end, we first define a simulation relation to capture when a backward GTS such as S_{BE} and S_{BU} for our running example is correct w.r.t. a forward GTS such as S_{FE} and S_{FU} for our running example. Since we do not consider the step labels (containing the GT rules or matches applied in these steps), we can understand this simulation

[7] Our presentation also covers the special case where the backward GTS used at design-time is obtained by reversing the rules of the run-time GTS but also applies to backward GTSs that are designed for improved design-time efficiency and applicability (as mentioned before Definition 7, in relation to k-induction discussed in section 3, and as elaborated in section 8).

to be a weak simulation in which one step of the forward GTS is simulated (backwards) by the backward GTS using any number of GT steps.

Definition 10 (Simulation Relation for GTS-based LTSs). *Given two LTSs Γ and Γ' induced from GTSs according to Definition 6 and Definition 7. A set R of morphisms $f_1 : G'_1 \hookrightarrow G_1$ from states $G'_1 \in \mathsf{states}(\Gamma')$ to states $G_1 \in \mathsf{states}(\Gamma)$ is a simulation relation from Γ to Γ', if for every $(G_2, \sigma, G_1) \in \mathsf{steps}(\Gamma)$ capturing the forward GT span $(g_2 : D_1 \hookrightarrow G_2, g_1 : D_1 \hookrightarrow G_1)$ there is a sequence of GT steps $(G'_2, \sigma'_n, G'_{2,n-1}), \ldots (G'_{2,1}, \sigma'_1, G'_1) \in \mathsf{steps}(\Gamma')$ that can be combined (using an iterated E-concurrent GT rule, [12, Theorem 3.26]) into the backward GT span $(g'_2 : D'_1 \hookrightarrow G'_2, g'_1 : D'_1 \hookrightarrow G'_1)$ such that $d_1 : D'_1 \hookrightarrow D_1$ and $f_2 : G'_2 \hookrightarrow G_2$ exist satisfying $f_2 \in R$, $f_2 \circ g'_2 = g_2 \circ d_1$, and $f_1 \circ g'_1 = g_1 \circ d_1$.*

$$\begin{array}{ccccc} G_2 & \xleftarrow{g_2} & D_1 & \xrightarrow{g_1} & G_1 \\ \uparrow{\scriptstyle f_2} & = & \uparrow{\scriptstyle d_1} & = & \uparrow{\scriptstyle f_1} \\ G'_2 & \xleftarrow{g'_2} & D'_1 & \xrightarrow{g'_1} & G'_1 \end{array}$$

The following theorem then states that the existence of such a simulation relation R from the forward GTS to the backward GTS containing at least all embeddings of unsafe states V into the graphs reachable in the forward GTS within k steps is sufficient to ensure that any safety violation of the forward GTS within n to $n + k$ steps is detected by checking the states reachable by n steps in $[\![S_{\mathsf{FS}}]\!]_n$ against the start states of $[\![S_{\mathsf{BS}}]\!]_k^{\mathsf{back}}$. Note that Theorem 1 does not exclude spurious violation paths in terms of path pairs (π_1, π_2) that are not composable to a path π of S_{FS} due to application conditions in GT rules used in π_1 or π_2. Moreover, note that paths to unsafe states of length at most n steps are detected by constructing $[\![S_{\mathsf{FS}}]\!]_n$ already.

Theorem 1 (Violation Detection). *Given a forward GTS S_{FS}, a backward GTS S_{BS}, and an unsafe graph V contained in $\mathsf{unsafe}(S_{\mathsf{FS}})$ and $\mathsf{unsafe}(S_{\mathsf{BS}})$, every violation detected in $[\![S_{\mathsf{FS}}]\!]_{n+k}$ in terms of some path π of length $> n$ from $\mathsf{start}(S_{\mathsf{FS}})$ to a graph containing V is correspondingly detected by the combined technique using $[\![S_{\mathsf{FS}}]\!]_n$ and $[\![S_{\mathsf{BS}}]\!]_k^{\mathsf{back}}$ by two paths π_1 of length n from $\mathsf{start}(S_{\mathsf{FS}})$ to a graph containing B and π_2 of length $\leq k$ from some B' (for which some $b : B' \hookrightarrow B$ exists) to the graph V whenever there is a simulation relation R from $[\![S_{\mathsf{FS}}]\!]_k$ to $[\![S_{\mathsf{BS}}]\!]_k^{\mathsf{back}}$ containing every mono $f : V \hookrightarrow G$ into states G of $[\![S_{\mathsf{FS}}]\!]_k$.*

Proof (sketch). By induction on k, we derive the existence of an embedding of the last graph B of π_2 into the last graph of π_1 ensuring that steps in π reaching a violating graph can be mimicked backwards via the simulation relation.

This theorem thereby ensures that the system has an effective look-ahead of $n+k$ steps at run-time towards unsafe states allowing it to derive suitable control decisions to avoid such unsafe states (if possible for that effective look-ahead).

8 Evaluation

As a case study, we now consider a more complex variation of the running example, including additional track features such as junctions, explicit modeling

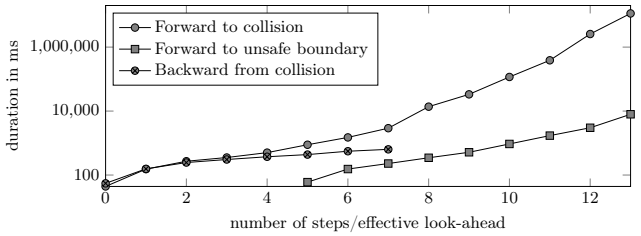


Fig. 6. Evaluation results. Look-ahead for "forward to collision", effective look-ahead for "forward to unsafe boundary", and depth of BBSS for "backward from collision".

of monitoring and signals (traffic lights for shuttles and ambulances). The used GTSs modeling this case study ensure that the sliding window perspective of the controlled shuttle is enforced by removing track and road segments behind the shuttle and enlarging the track/road topology forwards, potentially also including junctions, level crossing, and further components in a way to be expected by the shuttle. While we simply used the reversed rules for the backward GTSs in the running example, this would generate here for our case study, as for typical applications of the related approach of k-induction, a large number of unrealistic track topologies that would need to be singled out using other techniques such as structural constraints reducing the applicability and performance of our approach at design-time. Applying Theorem 1, we constructed a backward GTS with 31 GT rules by hand such that all steps of the forward GTS with 34 GT rules can be mimicked by at most two backward steps while minimizing the overapproximation of additional track topologies that are never reachable in the forward GTS. We used the tool Groove [17, 44] and provide the documented model files an explanation of our evaluation steps online.[8]

We evaluated the efficiency of our integrated approach in terms of consumed time by comparing it to the case where only a BFSS is constructed at run-time.[9] First, we use Groove to construct BFSSs of the forward GTS (for different bounds) thereby simulating the case where our approach is not used. Second, we use Groove to construct BBSSs of the backward GTS (for different bounds) also acquiring the unsafe boundary graphs thereby simulating the design-time aspect of our approach. Finally, we use Groove to construct the BFSS of the forward GTS (for different bounds) using the unsafe boundary graphs as target graphs (which means that the overhead of attempting to match the unsafe boundary graphs is included in our measurement) thereby simulating the run-time aspect of our approach. Generating the entire BFSS (for a given bound) instead of only adjusting it to the last observed step means that we consider the worst-case situation in which the entire BFSS is to be reconstructed due to, e.g., an unexpected step of the system. According to Figure 6 (forward to collision), the BFSS construction requires exponential run-time. In particular, collisions

[8] https://github.com/OpenAcademicProject/Running-Example-of-Railway-Transportation-System

[9] System: 64-bit Win10, Intel Core i7-6700HQ, 40GB RAM, Groove 5.8.1

are detected at depth 13 requiring 188 min, indicating that only using a BFSS may incur inacceptable costs at run-time. According to Figure 6 (backward to collision), the BBSS grows much slower compared to the BFSS because of *(a)* our usage of a separate backward GTS and *(b)* the restriction of considering paths that definitely lead to unsafe states. Hence, increasing the bound k for this BBSS is more advantageous compared to increasing the bound n for the BFSS in this scenario. Lastly, according to Figure 6 (forward to unsafe boundary), the first member of the unsafe boundary is found at run-time in the BFSS at depth 8 requiring 8 s with an effective look-ahead of 13 (as the depth 7 BBSS captures 5 forward steps of the forward GTS), which is 1423 times faster. We conclude from our evaluation that the goal of shifting computation time (and memory costs) from run-time to design-time is achieved by a factor of 1423 for the case study.

We note that applying our approach using a value $k > 0$ can increase the run-time cost. This would be the case when the forward/backward GTSs are constructed and the values of n and k are selected such that the time required for checking the leaf states of the run-time controller against the unsafe boundary of the design-time controller exceeds the time saved by generating at run-time a BFSS of depth n instead of $n + k$. This may be the case when, e.g., the BBSS contains a large number of infeasible paths (in the sense that the forward GTS cannot exhibit (instantiations of) them for the considered start states) resulting in an unsafe boundary containing a large number of states that can never be matched. While this issue did not arise for the case study considered here where run-time cost was decreased by a factor of 1423, this issue can be mitigated when it arises by employing assumed state invariants (capturing infeasibility of paths) to exclude states from the BBSS following the approaches in [47, 48, 3].

9 Conclusion and Future Work

In this paper, we presented a novel control-theoretic approach to run-time control for Graph Transformation Systems (GTSs) with priorities modeling large-scale systems with the threat of unexpected events. For the actor to be controller, we combine controllers synthesized at design-time and run-time with look-aheads n and k to obtain combined controllers with look-ahead $n + k$. An evaluation based on a shuttle transportation system shows a decrease of run-time computation cost by a factor of 1423 compared to using only run-time controllers with the same look-ahead suggesting that our approach successfully shifts a large amount of run-time computation cost to design-time. Moreover, we exemplified the robustness of the devised controlled system against unexpected events.

In the future, we will extend our approach to Interval Probabilistic Timed Graph Transformation Systems [31] to model cyber-physical systems and the steps of the contained actors more precisely, incorporate techniques to minimize checking time against unsafe boundary nodes, and combine k-induction with hand-coded backward GTSs to obtain small Bounded Backward State Spaces (BBSSs) that are correct w.r.t. the forward GTS by design.

References

[1] P. Arcaini, E. Riccobene, and P. Scandurra. "Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation". In: *10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015, Florence, Italy, May 18-19, 2015*. Ed. by P. Inverardi and B. R. Schmerl. IEEE Computer Society, 2015, pp. 13–23. DOI: 10.1109/SEAMS.2015.10.

[2] C. Baier and J. Katoen. *Principles of model checking*. MIT Press, 2008. ISBN: 978-0-262-02649-9.

[3] B. Becker and H. Giese. *Cyber-Physical Systems with Dynamic Structure: Towards Modeling and Verification of Inductive Invariants*. Tech. rep. 64. Hasso Plattner Institute, University of Potsdam, 2012. URL: https://nbn-resolving.org/urn:nbn:de:kobv:517-opus-62437.

[4] T. Brüdigam, V. Gaßmann, D. Wollherr, and M. Leibold. "Minimization of constraint violation probability in model predictive control". In: *International Journal of Robust and Nonlinear Control* 31.14 (2021), pp. 6740–6772. DOI: https://doi.org/10.1002/rnc.5636. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/rnc.5636.

[5] T. Brüdigam, J. Teutsch, D. Wollherr, M. Leibold, and M. Buss. "Probabilistic model predictive control for extended prediction horizons". In: *at - Automatisierungstechnik* 69.9 (2021), pp. 759–770. DOI: doi:10.1515/auto-2021-0025.

[6] Y.-L. Chen, S. Lafortune, and F. Lin. "Modular Supervisory Control with Priorities for Discrete Event Systems". In: *Proceedings of 1995 34th IEEE Conference on Decision and Control*. Vol. 1. 1995, pp. 409–415. DOI: 10.1109/CDC.1995.478832.

[7] Y. Chen, S. Lafortune, and F. Lin. "Resolving Feature Interactions Using Modular Supervisory Control with Priorities". In: *Feature Interactions in Telecommunications Networks IV, June 17-19, 1997, Montréal, Canada*. Ed. by P. Dini, R. Boutaba, and L. Logrippo. IOS Press, 1997, pp. 108–122.

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". In: *ACM Trans. Program. Lang. Syst.* 8.2 (1986), pp. 244–263. DOI: 10.1145/5397.5399.

[9] A. Corradini, L. Foss, and L. Ribeiro. "Graph Transformation with Dependencies for the Specification of Interactive Systems". In: *Recent Trends in Algebraic Development Techniques, 19th International Workshop, WADT 2008, Pisa, Italy, June 13-16, 2008, Revised Selected Papers*. Ed. by A. Corradini and U. Montanari. Vol. 5486. Lecture Notes in Computer Science. Springer, 2008, pp. 102–118. DOI: 10.1007/978-3-642-03429-9_8.

[10] B. Courcelle. "The Expression of Graph Properties and Graph Transformations in Monadic Second-Order Logic". In: *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. Ed. by G. Rozenberg. World Scientific, 1997, pp. 313–400. ISBN: 9810228848.

[11] A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer. "Software Verification Using k-Induction". In: *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*. Ed. by E. Yahav. Vol. 6887. Lecture Notes in Computer Science. Springer, 2011, pp. 351–368. ISBN: 978-3-642-23701-0. DOI: 10.1007/978-3-642-23702-7_26.

[12] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006. ISBN: 978-3-540-31187-4. DOI: 10.1007/3-540-31188-2.

[13] H. Ehrig, C. Ermel, U. Golas, and F. Hermann. *Graph and Model Transformation - General Framework and Applications*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2015. ISBN: 978-3-662-47979-7. DOI: 10.1007/978-3-662-47980-3.

[14] H. Ehrig, U. Golas, A. Habel, L. Lambers, and F. Orejas. "$\mathcal{M}$-adhesive transformation systems with nested application conditions. Part 1: parallelism, concurrency and amalgamation". In: *Mathematical Structures in Computer Science* 24.4 (2014). DOI: 10.1017/S0960129512000357.

[15] A. Filieri, C. Ghezzi, and G. Tamburrelli. "Run-time efficient probabilistic model checking". In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. Ed. by R. N. Taylor, H. C. Gall, and N. Medvidovic. ACM, 2011, pp. 341–350. DOI: 10.1145/1985793.1985840.

[16] S. Gerasimou, R. Calinescu, and A. Banks. "Efficient runtime quantitative verification using caching, lookahead, and nearly-optimal reconfiguration". In: *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, Proceedings, Hyderabad, India, June 2-3, 2014*. Ed. by G. Engels and N. Bencomo. ACM, 2014, pp. 115–124. DOI: 10.1145/2593929.2593932.

[17] GROOVE Team. *Graphs for Object-Oriented Verification (GROOVE)*. https://groove.cs.utwente.nl. University of Twente, 2011.

[18] A. Habel and K. Pennemann. "Correctness of high-level transformation systems relative to nested conditions". In: *Mathematical Structures in Computer Science* 19.2 (2009), pp. 245–296. DOI: 10.1017/S0960129508007202.

[19] R. Heckel. "Open graph transformation systems: a new approach to the compositional modelling of concurrent and reactive systems". PhD thesis. Technical University of Berlin, Germany, 1998. URL: https://d-nb.info/95713598X.

[20] R. Heckel, G. Engels, H. Ehrig, and G. Taentzer. "A View-based Approach to System Modeling Based on Open Graph Transformation Systems". In: *Handbook of Graph Grammars and Computing by Graph Transformation Volume 2: Applications, Languages and Tools*. Ed. by H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg. World Scientific, 1999, pp. 639–668. ISBN: 978-981-02-4020-2. DOI: 10.1142/9789812815149_0016.

[21] R. Heckel, G. Lajios, and S. Menge. "Stochastic Graph Transformation Systems". In: *Fundam. Inform.* 74.1 (2006), pp. 63–84. URL: https://content.iospress.com/articles/fundamenta-informaticae/fi74-1-04.

[22] R. Heckel and G. Taentzer. *Graph Transformation for Software Engineers - With Applications to Model-Based Development and Domain-Specific Language Engineering*. Springer, 2020. ISBN: 978-3-030-43915-6. DOI: 10.1007/978-3-030-43916-3.

[23] M. U. Iftikhar and D. Weyns. *Towards runtime statistical model checking for self-adaptive systems*. CW Reports CW693. Department of Computer Science, KU Leuven; Leuven, Belgium, Aug. 2016. URL: https://lirias.kuleuven.be/1656638.

[24] N. Jansen, C. Dehnert, B. L. Kaminski, J. Katoen, and L. Westhofen. "Bounded Model Checking for Probabilistic Programs". In: *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*. Ed. by C. Artho, A. Legay, and D. Peled. Vol. 9938. Lecture Notes in Computer Science. 2016, pp. 68–85. DOI: 10.1007/978-3-319-46520-3_5.

[25] J. Katoen. "The Probabilistic Model Checking Landscape". In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*. Ed. by M. Grohe, E. Koskinen, and N. Shankar. ACM, 2016, pp. 31–45. DOI: 10.1145/2933575.2934574.

[26] Z. Khasidashvili, K. Korovin, and D. Tsarkov. "EPR-based k-induction with Counterexample Guided Abstraction Refinement". In: *Global Conference on Artificial Intelligence, GCAI 2015, Tbilisi, Georgia, October 16-19, 2015*. Ed. by G. Gottlob, G. Sutcliffe, and A. Voronkov. Vol. 36. EPiC Series in Computing. EasyChair, 2015, pp. 137–150. DOI: 10.29007/scv7.

[27] R. Kumar and V. K. Garg. *Modeling and Control of Logical Discrete Event Systems*. 1st ed. Springer New York, NY, 1995. DOI: 10.1007/978-1-4615-2217-1.

[28] M. Leucker and C. Schallhart. "A brief account of runtime verification". In: *J. Log. Algebr. Program.* 78.5 (2009), pp. 293–303. DOI: 10.1016/j.jlap.2008.08.004.

[29] M. Maximova, H. Giese, and C. Krause. "Probabilistic timed graph transformation systems". In: *J. Log. Algebr. Meth. Program.* 101 (2018), pp. 110–131. DOI: 10.1016/j.jlamp.2018.09.003.

[30] M. Maximova, S. Schneider, and H. Giese. "Compositional Analysis of Probabilistic Timed Graph Transformation Systems". In: *Fundamental Approaches to Software Engineering - 24th International Conference, FASE 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*. Ed. by E. Guerra and M. Stoelinga. Vol. 12649. Lecture Notes in Computer Science. Springer, 2021, pp. 196–217. DOI: 10.1007/978-3-030-71500-7_10.

[31] M. Maximova, S. Schneider, and H. Giese. "Interval Probabilistic Timed Graph Transformation Systems". In: *Graph Transformation - 14th International Conference, ICGT 2021, Held as Part of STAF 2021, Virtual Event, June 24-25, 2021, Proceedings*. Ed. by F. Gadducci and T. Kehrer. Vol. 12741. Lecture Notes in Computer Science. Springer, 2021, pp. 221–239. DOI: 10.1007/978-3-030-78946-6_12.

[32] G. A. Moreno, J. Cámara, D. Garlan, and B. R. Schmerl. "Proactive self-adaptation under uncertainty: a probabilistic model checking approach". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. Ed. by E. D. Nitto, M. Harman, and P. Heymans. ACM, 2015, pp. 1–12. DOI: 10.1145/2786805.2786853.

[33] O. Özkan. "Decidability of Resilience for Well-Structured Graph Transformation Systems". In: *Graph Transformation - 15th International Conference, ICGT 2022, Held as Part of STAF 2022, Nantes, France, July 7-8, 2022, Proceedings*. Ed. by N. Behr and D. Strüber. Vol. 13349. Lecture Notes in Computer Science. Springer, 2022, pp. 38–57. DOI: 10.1007/978-3-031-09843-7_3.

[34] O. Özkan. "Infinite-state graph transformation systems under adverse conditions". In: *it Inf. Technol.* 63.5-6 (2021), pp. 311–320. DOI: 10.1515/itit-2021-0011.

[35] O. Özkan. "Modeling Adverse Conditions in the Framework of Graph Transformation Systems". In: *Proceedings of the Eleventh International Workshop on Graph Computation Models, GCM@STAF 2020, Online-Workshop, 24th June 2020*. Ed. by B. Hoffmann and M. Minas. Vol. 330. EPTCS. 2020, pp. 35–54. DOI: 10.4204/EPTCS.330.3.

[36] O. Özkan and N. Würdemann. "Resilience of Well-structured Graph Transformation Systems". In: *Proceedings Twelfth International Workshop on Graph Com-*

putational Models, GCM@STAF 2021, Online, 22nd June 2021. Ed. by B. Hoffmann and M. Minas. Vol. 350. EPTCS. 2021, pp. 69–88. DOI: 10.4204/EPTCS.350.5.

[37] A. Pnueli. "The Temporal Logic of Programs". In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32. URL: https://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4567914.

[38] RailCab Team. *RailCab Project*. https://www.hni.uni-paderborn.de/cim/projekte/railcab.

[39] P. J. G. Ramadge and W. M. Wonham. "On the Supremal Controllable Sublanguage of a Given Language". In: *SIAM Journal on Control and Optimization (SICON)* 25.3 (1987), pp. 637–659.

[40] P. J. G. Ramadge and W. M. Wonham. "On the Supremal Controllable Sublanguage of a given Language". In: *Decision and Control, 1984. The 23rd IEEE Conference on*. Vol. 23. 1984, pp. 1073–1080. DOI: 10.1109/CDC.1984.272178.

[41] P. J. G. Ramadge and W. M. Wonham. "Supervisory Control of a Class of Discrete Event Processes". English. In: *Analysis and Optimization of Systems*. Ed. by A. Bensoussan and J. Lions. Vol. 63. Lecture Notes in Control and Information Sciences. Springer Berlin Heidelberg, 1984, pp. 475–498. DOI: 10.1007/BFb0006306.

[42] A. Rashidinejad, P. van der Graaf, and M. A. Reniers. "Nonblocking Supervisory Control Synthesis of Timed Automata using Abstractions and Forcible Events". In: *16th International Conference on Control, Automation, Robotics and Vision, ICARCV 2020, Shenzhen, China, December 13-15, 2020*. IEEE, 2020, pp. 1033–1040. DOI: 10.1109/ICARCV50220.2020.9305312.

[43] A. Rashidinejad, M. A. Reniers, and M. Fabian. "Supervisory Control Synthesis of Timed Automata Using Forcible Events". In: *CoRR* abs/2102.09338 (2021). arXiv: 2102.09338. URL: https://arxiv.org/abs/2102.09338.

[44] A. Rensink. "The GROOVE simulator: A tool for state space generation". In: *Applications of Graph Transformations with Industrial Relevance: Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27-October 1, 2003, Revised Selected and Invited Papers 2*. Springer. 2004, pp. 479–485.

[45] J. Rinast. "An online model-checking framework for timed automata". PhD thesis. Hamburg University of Technology, 2015. URL: http://tubdok.tub.tuhh.de/handle/11420/1256.

[46] S. Schneider. "Deterministic pushdown automata as specifications for discrete event supervisory control in Isabelle". PhD thesis. Straße des 17. Juni 135, 10623 Berlin, Germany: Technische Universität Berlin, Dec. 2019. 286 pp. DOI: 10.14279/depositonce-9332. In press.

[47] S. Schneider, J. Dyck, and H. Giese. "Formal Verification of Invariants for Attributed Graph Transformation Systems Based on Nested Attributed Graph Conditions". In: *Graph Transformation - 13th International Conference, ICGT 2020, Held as Part of STAF 2020, Bergen, Norway, June 25-26, 2020, Proceedings*. Ed. by F. Gadducci and T. Kehrer. Vol. 12150. Lecture Notes in Computer Science. Springer, 2020, pp. 257–275. DOI: 10.1007/978-3-030-51372-6_15.

[48] S. Schneider, M. Maximova, and H. Giese. "Invariant Analysis for Multi-agent Graph Transformation Systems Using k-Induction". In: *Graph Transformation - 15th International Conference, ICGT 2022, Held as Part of STAF 2022, Nantes,*

France, July 7-8, 2022, Proceedings. Ed. by N. Behr and D. Strüber. Vol. 13349. Lecture Notes in Computer Science. Springer, 2022, pp. 173–192. DOI: 10.1007/978-3-031-09843-7_10.

[49] S. Schneider, M. Maximova, L. Sakizloglou, and H. Giese. "Formal testing of timed graph transformation systems using metric temporal graph logic". In: *Int. J. Softw. Tools Technol. Transf.* 23.3 (2021), pp. 411–488. DOI: 10.1007/s10009-020-00585-w.

[50] T. Schüle and K. Schneider. "Bounded model checking of infinite state systems". In: *Formal Methods Syst. Des.* 30.1 (2007), pp. 51–81. DOI: 10.1007/s10703-006-0019-9.

[51] M. Schwenzer, M. Ay, T. Bergs, and D. Abel. "Review on model predictive control: an engineering perspective". In: *The International Journal of Advanced Manufacturing Technology* 117.5 (Nov. 2021), pp. 1327–1349. ISSN: 1433-3015. DOI: 10.1007/s00170-021-07682-3.

[52] A. M. Sharifloo and A. Metzger. "Mcaas: Model checking in the cloud for assurances of adaptive systems". In: *Software Engineering for Self-Adaptive Systems III. Assurances*. Springer, 2017, pp. 137–153. DOI: 10.1007/978-3-319-74183-3_5.

[53] D. Weyns, B. R. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka. "On Patterns for Decentralized Control in Self-Adaptive Systems". In: *Software Engineering for Self-Adaptive Systems II - International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*. Ed. by R. de Lemos, H. Giese, H. A. Müller, and M. Shaw. Vol. 7475. Lecture Notes in Computer Science. Springer, 2010, pp. 76–107. DOI: 10.1007/978-3-642-35813-5_4.

[54] Y. Yang, L. Bu, and X. Li. "Forward and backward: Bounded model checking of linear hybrid automata from two directions". In: *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*. Ed. by G. Cabodi and S. Singh. IEEE, 2012, pp. 204–208. URL: https://ieeexplore.ieee.org/document/6462575/.

[55] R. Zhang, Z. Wang, and K. Cai. "N-Step Nonblocking Supervisory Control of Discrete-Event Systems". In: *2021 60th IEEE Conference on Decision and Control (CDC), Austin, TX, USA, December 14-17, 2021*. IEEE, 2021, pp. 339–344. DOI: 10.1109/CDC45484.2021.9683593.