AutoKnigge—Modeling, Evaluation and Verification of Cooperative Interacting Automobiles

Christian Kehl, Maximilian Kloock, Evgeny Kusmenko, Lutz Eckstein, Bassam Alrifaee, Stefan Kowalewski, and Bernhard Rumpe

Abstract The development of cooperative driving functions to optimize traffic systems shows high potential to improve individual autonomous driving systems with respect to topics like traffic flow, vehicle safety and user comfort. The core concept of the presented solutions is the Local Traffic System (LTS). Following the messages defined in European Telecommunications Standards Institute (ETSI) Intelligent Transport Systems (ITS) G5 for Vehicle-to-everything (V2X) cooperation we introduce concepts and implementations to intelligently group vehicles based on the exchanged V2X data with respect to the individual vehicle capability for cooperation. Based on the determined grouping, we present algorithms for cooperative trajectory planning. We develop a verification method for the cooperatively planned trajectories

C. Kehl, M. Kloock, E. Kusmenko—These authors contributed equally.

C. Kehl (✉) · L. Eckstein
Institute for Automotive Engineering (ika), RWTH Aachen University, Aachen, Germany
e-mail: christian.kehl@ika.rwth-aachen.de

L. Eckstein
e-mail: lutz.eckstein@ika.rwth-aachen.de

M. Kloock · S. Kowalewski
Chair of Embedded Software, RWTH Aachen University, Aachen, Germany
e-mail: kloock@embedded.rwth-aachen.de

S. Kowalewski
e-mail: kowalewski@embedded.rwth-aachen.de

E. Kusmenko · B. Rumpe
Chair of Software Engineering, RWTH Aachen University, Aachen, Germany
e-mail: kusmenko@se-rwth.de

B. Rumpe
e-mail: rumpe@se-rwth.de

B. Alrifaee
Department of Aerospace Engineering, University of the Bundeswehr Munich, Neubiberg, Germany
e-mail: bassam.alrifaee@unibw.de

C. Stiller et al. (eds.), *Cooperatively Interacting Vehicles*,
https://doi.org/10.1007/978-3-031-60494-2_13

within a LTS. The verification guarantees collision avoidance and deadlock-freeness in real-time. Finally we introduce a model language based on MontiArc to enable a systematic representation and description of the presented concepts for grouping, cooperation and interaction.

1 Introduction

Rapid technological advancements in the area of automated driving functions in recent years make large-scale deployment of SAE Level 4 and 5 [45] automated vehicles likely in the next few years. While technological progress is mainly limited to the development of vehicle-specific automation functions, the development of cooperative automation functions for the optimization of traffic systems already shows high potential to significantly improve current topics of concern such as traffic flow, vehicle safety and user comfort.

Current Vehicle-to-everything (V2X) systems show a beacon-like behavior without a direct sender or receiver and are rather designed to transmit one-time events to alert other traffic participants. The next logical step towards the development of cooperatively interacting vehicles requires a significant extension of existing V2X systems at all levels. The extension of these systems from a one-time event-based communication to a continuous data exchange for the execution of cooperatively interacting algorithms [4, 28, 39], raises questions regarding the grouping of the involved road users [29], reliability vehicle communication [32], the type of information exchanged, the underlying algorithms as well as the basic model description of these systems. Methods that present cooperative trajectory planning of vehicles in different scenarios are, e.g., the works in [31, 33, 34]. These works focus on the applicability of cooperative trajectory planning in intersections, pose control, and vehicle racing.

The core concept of the solutions presented in the following is the Local Traffic System [7]. Local Traffic Systems can be understood as cooperating C-ITS subsystems as defined in European Telecommunications Standards Institute (ETSI) Intelligent Transport Systems (ITS) G5. Based on this concept, different approaches for the detection of the corresponding traffic scenarios, the formation of Local Traffic Systems as well as their evaluation are presented. Within these systems the cooperation takes place. In the context of this work, the cooperative trajectory planning, as well as a real-time verification of the cooperatively planned trajectories are presented. The verification guarantees the absence of collisions and deadlocks for the trajectories of all vehicles in one or multiple LTS. Finally, a model language based on MontiArc is presented for the systematic representation and description of the presented concepts for grouping, cooperation and interaction.

2 Learning-Based and Vehicle Capability-Aware Architecture for Clustering of Cooperative Interacting Automobiles

One of the central aspects within the overall process for cooperation and interaction of vehicles is the clustering of traffic participants relevant for cooperation. The formation of these clusters for the purpose of cooperation inevitably leads to the following questions: When is cooperation and interaction between traffic participants useful? What kind of vehicle data must be exchanged before and during cooperation? How can relevant traffic participants be identified?

In order to group the corresponding traffic participants, this work takes up the concept of Local Traffic Systems (LTS) [7] and develops it further. Local Traffic Systems are defined as a grouping of road users for the purpose of information exchange as well as cooperation. The cooperation take place exclusively within the LTS.

Previous work [8] in the area of Local Traffic Systems has been based on a single evaluation function. This evaluation function consists of various normalized distance metrics such as the distance between individual vehicles, the derivative of the distance function, the direction of travel, etc. The position information is based on a predefined road graph that must be known to all road users. The nodes of the road graph represent different points within the traffic network and have a distance of a few meters. The edges of the road graph represent the roads themselves. Road properties such as the maximum permitted speed are assigned to the edges. The individual distance metrics are then normalized and multiplied by a developer defined weighting factor. The now normalized and weighted metrics are finally added to an overall evaluation function. The objective is to minimize the evaluation function. The LTS configuration with the most minimized evaluation function is considered as an optimal solution. The information exchanged here to determine the individual metrics is already based on current standardizations such as the Cooperative Awareness Message (CAM) [13] and are extended when necessary. The vehicle data is exchanged cyclically. After the LTS formation, the cooperation takes place through data exchange within the system.

However, this approach has several disadvantages. The recurring calculation of the entire LTS configuration leads to an enormously high computing load, which makes a calculation in real time almost impossible. In [8] therefore a greedy algorithm is recommended, which makes only small changes at the past configuration in each time step without recalculating the total configuration. Additionally, the number of permitted LTS participants is limited to a maximum of 5-10 participants. This serves on the one hand to reduce the total computation time, and on the other hand to reduce the amount of information exchanged within the LTS in order to prevent an overload of the available bandwidth.

The necessity for a common road graph model shared by all vehicles represents a considerable limitation. The formation based on a road graph is here not limited by the mere necessity of the graph itself, but by the required correspondence of the graph between all road users. It is already apparent today that predefined map data will play a decisive role in the implementation of autonomous driving functions

[47]. However, they often take on a supporting role for localization [41]. Due to the frequent changes in the road network and the resulting inaccurate data, possible cooperation approaches should be map-independent. Furthermore, planning based on the road graph limits the accuracy of LTS formation to the accuracy of the existing map material because all positions are defined relative to the underlying graph. This poses a problem especially for cooperative maneuvers when the required vehicle distance is below the minimal accuracy level defined by the road graph.

Another disadvantage is the decoupling of exchanged vehicle data, LTS generation, the underlying cooperation algorithms, and the current driving situation. The permanently high amount of exchanged vehicle data leads to an unnecessarily high utilization of the available V2X data rate in the vehicle. IEEE 802.11p and LTE V2X can support data rate of up to 27 Mbit s^{-1} and 28.8 Mbit s^{-1} [44]. The lack of a link to the current vehicle situation and the underlying algorithms not only makes it difficult to prioritize individual LTS systems, but also ignores the influence of the current driving situation on LTS generation. The following section is intended to give a better impression of the resulting problems and derive additional requirements for improvements.

2.1 Requirements for an Extended LTS Architecture

Using selected examples, this section attempts to provide insight into the motivation for extending the previous approach and to derive possible requirements for an extended architecture. The goal is to preserve the general concept while identifying concept limitations and avoiding the disadvantages identified in the course of previous work.

The question of when a Local Traffic System should be formed at all and which road users should participate in it is closely linked to the respective traffic scenario. Possible traffic scenarios are shown in Fig. 4. In the following an exemplary traffic scenario of a roundabout with five vehicles is displayed in Fig. 1. The planned routes of the vehicles are marked in color along the center of the lane. The drawn rectangles indicate the possible LTS groupings for the scenario at hand. Vehicles can be grouped based on their respective vehicle state relative to other road users as well as relative to their surroundings. This can be based on various vehicle data such as the spatial proximity to the next vehicle, the overlap of the planned trajectories, the general overlap of the planned routes, or the spatial proximity of the vehicle under consideration relative to a relevant traffic node such as an intersection or roundabout.

One of the central problems here is the influence of cooperation, or cooperation capability, as well as the driving situation on LTS generation itself. The scenario shown in Fig. 2 illustrates the problem. A fast moving vehicle is approaching a slow moving vehicle on the right lane. In order to avoid heavy braking of the right vehicle behind, a lane change to the left lane is attempted. In general, two possible LTS are conceivable in this situation. One LTS consisting of the rear two vehicles to coordinate the lane change and one overall LTS consisting of all vehicles. However, if the rear

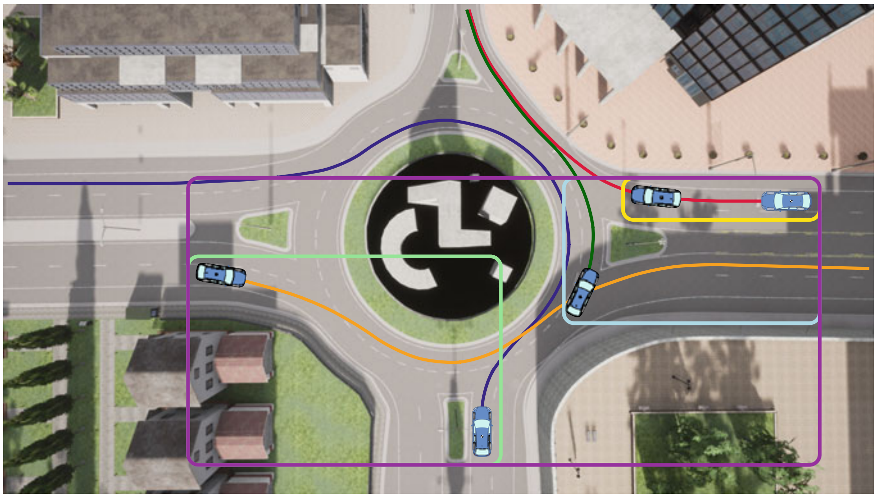


Fig. 1 Possible local traffic systems—roundabout scenario

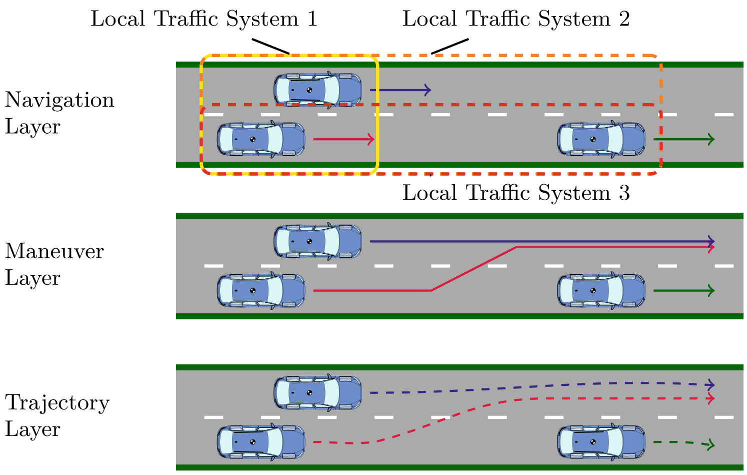


Fig. 2 Motion planning LTS layers

vehicle does not have the ability to change lanes cooperatively, the formation of a third LTS from the two right vehicles for the purpose of speed adaptation is necessary. A downstream cooperation without consideration of the vehicle capabilities leads to an incorrect LTS formation.

If we now extend the given scenario as shown in Fig. 3, assuming the ability to change lanes, another problem becomes apparent. In order to enable a lane change of the right vehicle, in principle three vehicles would have to slow down their speed, which is unfavorable from the point of view of a global optimization. However, a

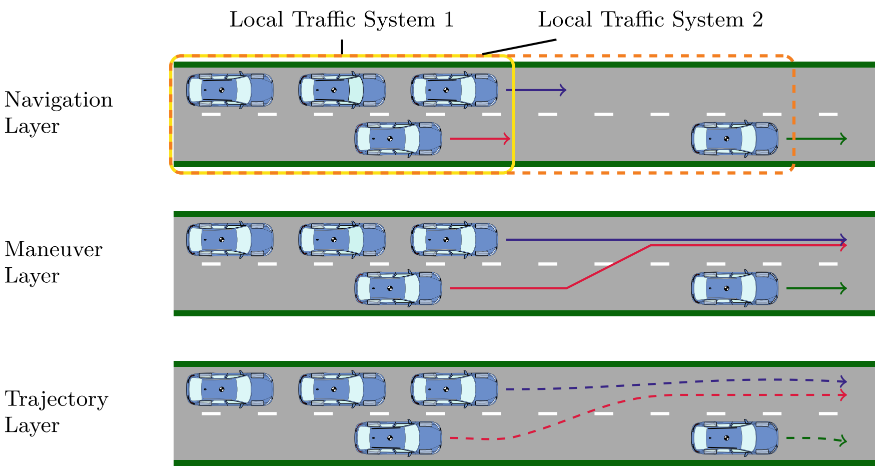


Fig. 3 Motion planning LTS layers conflict

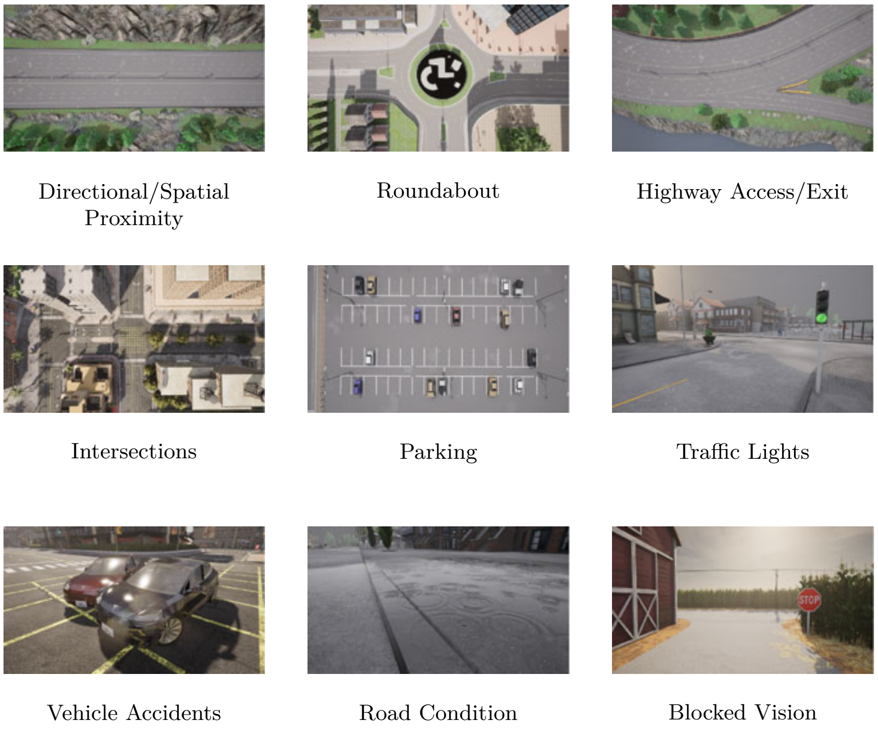


Fig. 4 Possible LTS scenarios

human actor would possibly prefer the light braking of several vehicles to the strong braking maneuver of a single vehicle. The respective driving situation therefore also has a decisive influence on the vehicle grouping here. The simple static weighting of different distance metrics is highly unlikely to meet this requirement.

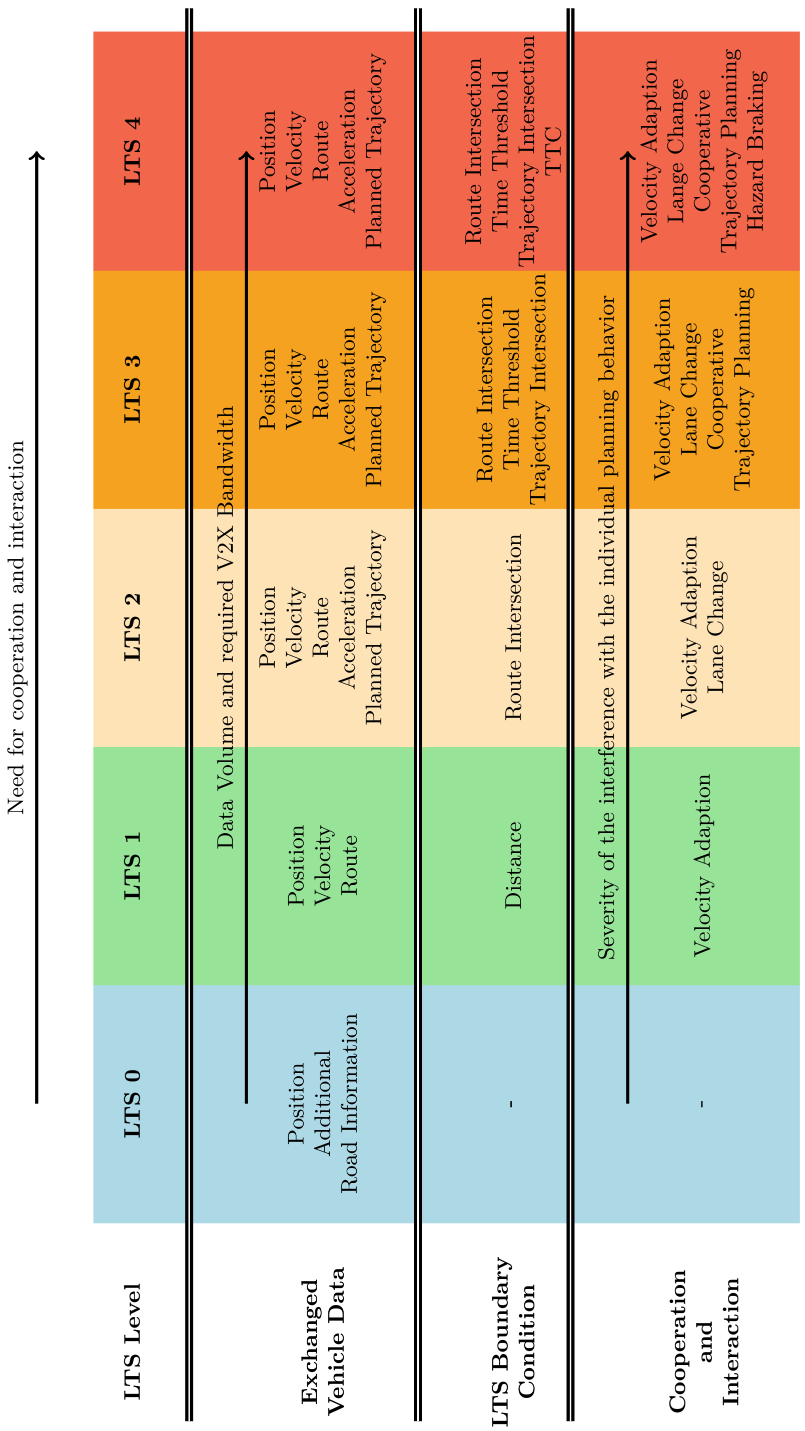
The examples described above show that the selection of LTS participants is complex and a wide variety of conflict situations can occur within a single scenario. These are strongly dependent on the direct driver environment and require a high level of algorithmic understanding of the driving situation which, as shown in Fig. 3, cannot be solved solely within the cooperation algorithm but has direct repercussions on the LTS formation. The incorrect too narrow selection of the LTS makes an optimal solution impossible.

In addition to the wrong selection of the LTS participants, there is also the possibility of potential target conflicts between different LTSs. If a vehicle is simultaneously a member of several competing LTSs, it is necessary to establish a prioritization between the individual systems.

2.2 *Extended LTS Architecture*

The disadvantages and problems of the previous concept described in the previous sections are to be solved by an extension of the architecture. The basic principles and advantages of the previous approach are to be preserved.

Figure 5 describes the novel approach to the clustering of vehicles. The most obvious difference is the division into different LTS levels between level 0 and level 4. The individual levels represent an increasing urgency in the need for cooperation between the road users and allow prioritization between individual LTS. Systems with a higher level are always given priority. In case of identical levels, no cooperation is performed. The system waits for escalation to higher levels. If several systems reach the highest level at the same time, cooperation between all traffic participants is required. The traffic systems are merged into a larger system. Each LTS level is associated with a specific set of exchanged vehicle data, boundary conditions, and available cooperation algorithms. At the beginning, every vehicle that has not yet been assigned to a specific group is at level 0. No active cooperation takes place here. Only simple awareness based information, like the current vehicle position or additional road information, like emergency warnings, are exchanged. This also provides a way to integrate passive road users unable to participate in a cooperative effort such as pedestrians, cyclists or infrastructure components like traffic lights. Each level is assigned a cooperation algorithm in addition to the vehicle data and associated boundary conditions. The LTS level is increased if the exchanged vehicle data exceeds the level specific boundary conditions. The type of cooperation algorithm increases according to the intensity of the intervention in the longitudinal and lateral control of the vehicle. The vehicle data required for the cooperation must not exceed the scope of the data exchange planned for the level. The amount of data exchanged increases here because more complex cooperation maneuvers usually require a larger pool of


Need for cooperation and interaction
Data Volume and required V2X Bandwidth
Severity of the interference with the individual planning behavior

LTS Level	**LTS 0**	**LTS 1**	**LTS 2**	**LTS 3**	**LTS 4**
Exchanged Vehicle Data	Position Additional Road Information	Position Velocity Route	Position Velocity Route Acceleration Planned Trajectory	Position Velocity Route Acceleration Planned Trajectory	Position Velocity Route Acceleration Planned Trajectory
LTS Boundary Condition	-	Distance	Route Intersection	Route Intersection Time Threshold Trajectory Intersection	Route Intersection Time Threshold Trajectory Intersection TTC
Cooperation and Interaction	-	Velocity Adaption	Velocity Adaption Lane Change	Velocity Adaption Lane Change Cooperative Trajectory Planning	Velocity Adaption Lange Change Cooperative Trajectory Planning Hazard Braking

Fig. 5 LTS structure

data. The data exchanged is roughly based on the data specified by the ETSI ITS G5 standard. Large parts of the described ETSI ITS G5 functionalities are still in an early stage of development at the time of this work and are therefore susceptible to possible changes. ETSI ITS G5 defines Cooperative Intelligent Transport Systems (C-ITS) as ITS subsystems such as people, vehicles, roadside units that exchange information or cooperate with each other to improve driving safety, traffic guidance or driving experience. The cooperation capabilities to be realized are referred to as services.

A general distinction is made between three categories of services. Cooperative Awareness Services [12–14] define the lowest level and describe the exchange of simple status information such as position and speed for the purpose of simple warnings. Cooperative Perception Services [17, 18] describe the second type of information exchange on top of status data. Within this service, other traffic participants are not only warned but also enabled to perform more complex functions such as Cooperative Adaptive Cruise Control. Cooperative Maneuver Coordination Services [15, 16] describe the highest level of cooperation. In addition to status and observation data road users can share their intention in order to allow cooperation in complex driving situations. These include scenarios like platooning or cooperative lane changes. The approach to LTS education presented here is oriented along the escalating nature of these services in terms of user interaction and user collaboration. In addition to minimizing intervention in the longitudinal and lateral control of the vehicle to increase user comfort, this also reduces the required bandwidth. Instead of exchanging all driving information periodically, only the information required for cooperation at the current level is exchanged. Further development and replacement of individual cooperation algorithms is possible without adaptation to the overall system.

For further development of the described architecture, a stimulative implementation approach is used. The developed framework is structured according to the diagram in Fig. 6. The CARLA simulator [10] serves as the basic foundation. The CARLA simulator is an open source driving simulator providing a virtual environment to simulate different driving scenarios and test autonomous driving functions in a virtual environment. The simulator is using a server/client concept. While the server is responsible for the simulation itself, the client controls the simulator by reading and writing data from and to the simulation using a TCP/IP. The client exposes the provided functions using an API to control traffic generation, pedestrian behavior, weather, sensors, maps and much more. During this project the provided Python API is used by the simulator interface to expose relevant functionality to the other components of the Autoknigge Framework. All components are connected using a ROS2 Communication Layer. This applies to messages controlling the simulator itself as well as messages exchanged between simulated vehicles. ROS2 uses the Data Distribution Service (DDS) which is also part of the Automotive Open System Architecture (AUTOSAR) Adaptive Platform. DDS is a middleware specified by the Object Management Group for data-centric communication in distributed systems. Based on the ROS Communication Layer there are higher level components like the World component. The World acts as a central repository for all data relevant to the simulation, such as vehicle positions, velocities, planned routes and trajectories, LTS

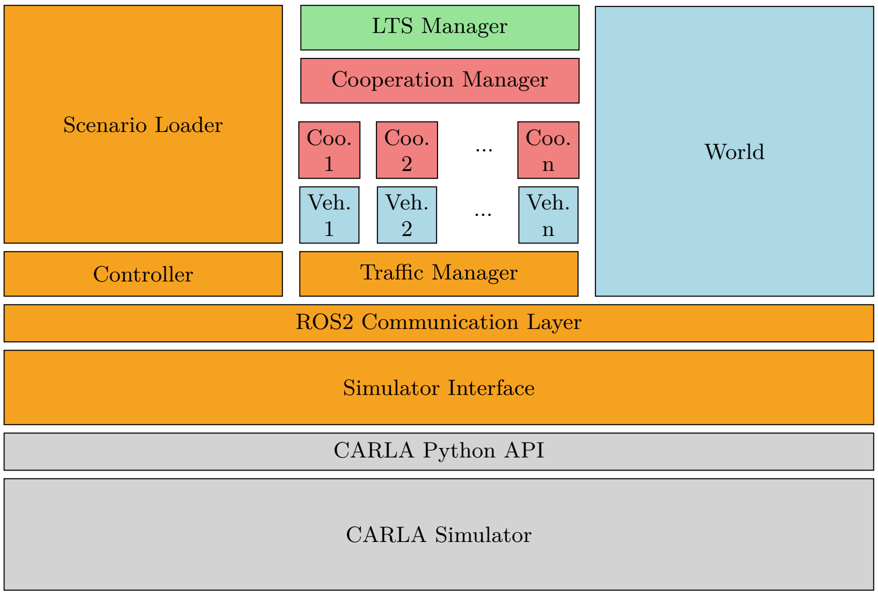


Fig. 6 Autoknigge architecture (Cooperation (Coo.), Vehicle (Veh.))

allocations, etc. The World component is also the central repository for all data relevant to the simulation. Unlike the data stored in the individual vehicle components, all data is available here. The controller is responsible for controlling the simulation itself and provides functions for selecting the map, adding vehicles and people. The controller is in turn used by the Scenario Loader to load various traffic scenarios. The Traffic Manager is used to abstract the management and communication of individual vehicles in the simulation. Here, for example, a distance-based forwarding of V2X messages takes place in order to simulate a range limitation of the vehicle messages. The number of managed vehicles is determined by the vehicles currently present in the simulation. Each vehicle component can theoretically contain its own Cooperation Manager to start and stop cooperation maneuvers. For simplicity in the context of the simulation, all vehicles currently share a Cooperation Manager. This also applies to the LTS Manager which assigns vehicles to the individual LTS.

The architecture described so far still lacks a concrete cooperation algorithm. Therefore Sect. 2.3 presents the implementation of a method for cooperative velocity adaptation for LTS level 1 systems.

2.3 Cooperative Velocity Adaption Algorithm

Cooperative adjustment of vehicle velocity represents one of the most minimally invasive forms of active cooperation, as it only interferes with the longitudinal control of the vehicle. By intervening in the vehicle's longitudinal control at an early stage, it is often possible to resolve conflict situations without impairing the vehicle occupants' sense of comfort due to strong longitudinal or lateral acceleration. The cooperative speed adaptation algorithm presented in the following is designed as a constraint optimization problem. The relevant boundary conditions are formulated as hard constraints and soft constraints. Hard constraints are unbreakable rules that must be fulfilled in any case, otherwise the result is not considered as a valid solution of the problem. Soft constraints represent less strict constraints and are understood as a kind of optimization parameter to distinguish several valid solutions in their quality. The problem is formulated in the form of a model and then passed to a solver. The concrete algorithm uses the Google OrTools CP-Sat Solver [42]. Due to the limitations of the solver, all variables and parameters of the model are formulated as integer values. Input and output values that are represented as floating point numbers are appropriately scaled by the algorithm beforehand.

The algorithm expects for each vehicle j two position arrays specifying the planned x,y-trajectories for each timestep i as well as additional parameters like the allowed minimum speed $v_{j,min}$, the maximum speed $v_{j,max}$. In addition, limits for the permitted longitudinal acceleration a_{max} as well as a minimum time gap $t_{gap,min}$ to ensure collision avoidance need to be defined. The variables are defined for each vehicle j involved.

The algorithm defines a vehicle velocity variable $v_{j,i}$ as well as a resulting timestamp $t_{j,i}$ for each vehicle position $p_{j,i} = (x_{j,i}, y_{j,i})$. The maximum acceleration is used to determine the permitted velocity change $v_{j,\Delta,max,i}$ for each distance step $d_{j,i}$. The timestamp $t_{j,i+1}$ is automatically calculated in the solver model using the distance $d_{j,i}$ between the position $p_{j,i}$ and $p_{j,i+1}$ as well as the velocity $v_{j,i}$ determined by the solver. To avoid a collision the model requires the time gap between two timestamps of two vehicles to be greater than the predefined time gap $t_{gap,min}$ threshold if the positions are closer than d_{thres}. The algorithm currently does not take into account the actual vehicle geometry. Therefore, the position distance value d_{thres} must be chosen sufficiently large. At every position the calculated velocity $v_{j,i}$ must be between $v_{j,min}$ and $v_{j,max}$ to be considered as valid result. As an optimizable soft constraint, the algorithm determines the maximum total duration of the maneuver as the time at which the last vehicle reaches the last target position in the planned trajectory.

The goal of the optimization is to minimize the total maneuvering time while taking into account the constraints described above.

As an example the algorithm is applied to the intersection scenario presented in Fig. 7. The vehicles are each positioned 10 m from the center of the intersection. The given speed is chosen for both vehicles so that the trajectories intersect at the same time and place. The usage of the presented algorithm with a spatial resolution of 1 m results in an optimal solution shown in Fig. 8. Compared to the individual use of the

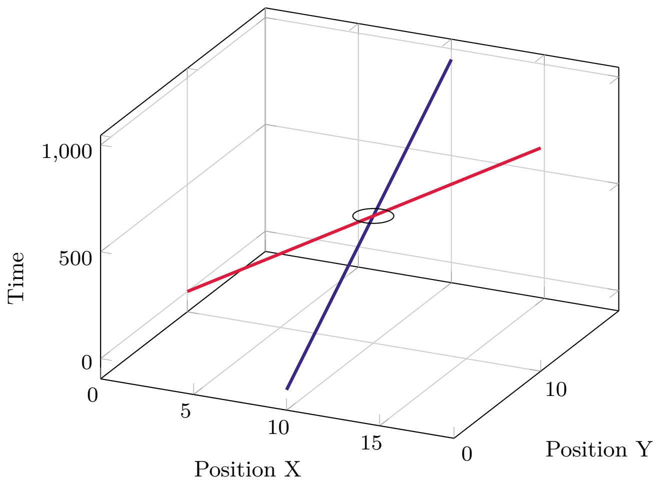


Fig. 7 Intersection scenario—conflict

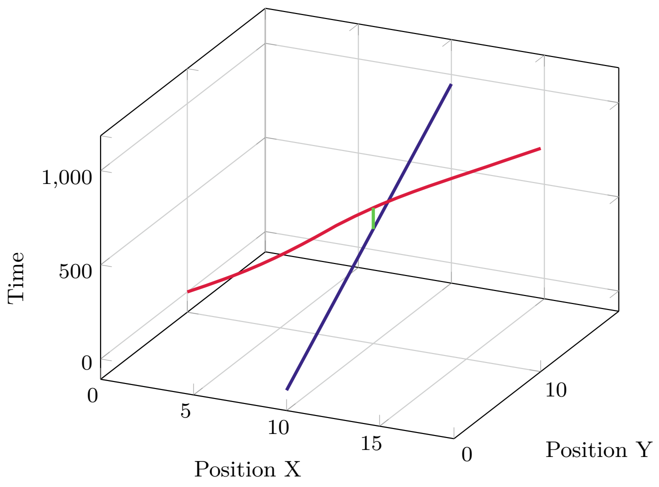


Fig. 8 Intersection scenario—conflict resolution by velocity adaption

intersection by each vehicle, the travel time for the braking vehicle is increased by approximately 13.5 %. The specified time gap is marked in green. The labeling of the displayed time axis does not correspond to the actual time in seconds but represents the direct integer solution value of the solver.

The formulation as a constraint optimization problem offers several advantages. On the one hand, the solver is able to capture the problem completely and detect conflicting constraints or prove the unsolvability of the problem at an early stage. The LTS system can thus detect the unsolvability of the cooperation task at an early

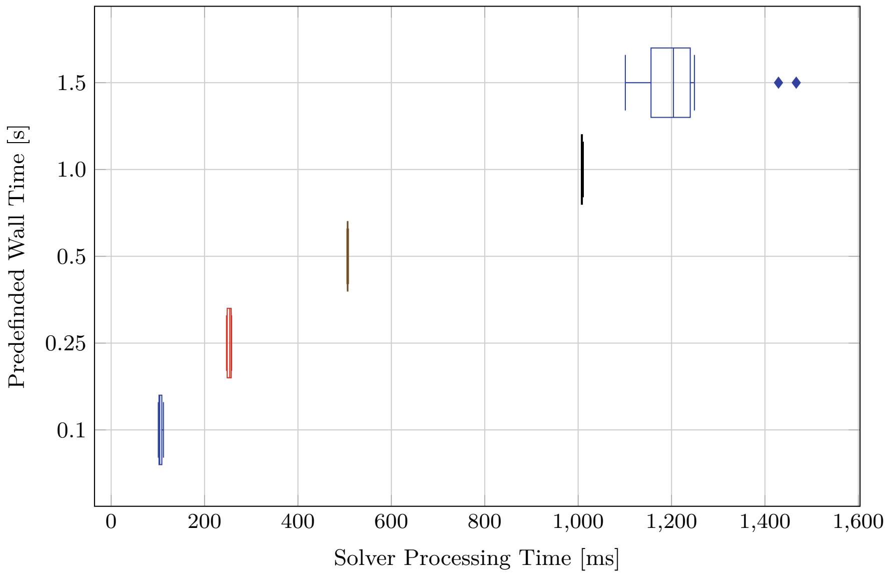


Fig. 9 Intersection scenario—solver time

stage with the algorithms available at this LTS level and increase the level. On the other hand, the solver is able to recognize optimal solutions as such and abort further optimization at an early stage. The term optimal here refers only to the given solution space based on the discretization used. For example, a finer discretization of the vehicle position would lead to an improved optimal solution.

A disadvantage of the used approach is the generally slower solution of complex constraint optimization problems with many variables. If in the given example the accuracy of the trajectory is increased from 1 m to 10 cm, the calculation time of the solution increases by a factor of 10-12 to around 1.2 s. The solver allows to set a wall time to reduce the calculation time. This represents the allowed calculation time. The best available solution at this time is used. Figure 9 shows the computation time of the algorithm for different given maximum computation times. For each calculation time, 10 runs were performed. It can be seen that the algorithm respects the specified maximal calculation time with a deviation of a few milliseconds. The lower value at a maximal calculation time of 1.5 s shows the automatic termination process, since on average an optimal solution is already found at 1.2 s.

The limitation of the calculation time has a significant influence on the reduction of the solution quality. Figure 10 shows that below 1.0 s in most cases no optimal solution can be found. Between 0.1 s and 1.0 s the algorithm finds sufficient solutions with constantly decreasing quality. The percentage increase in the duration of the cooperation maneuver relative to an optimal solution is shown in Fig. 11. With a limited calculation time of 0.1 s, the algorithm only finds a valid solution in 50 % of the cases. The measurements also show that the initial abandonment of an opti-

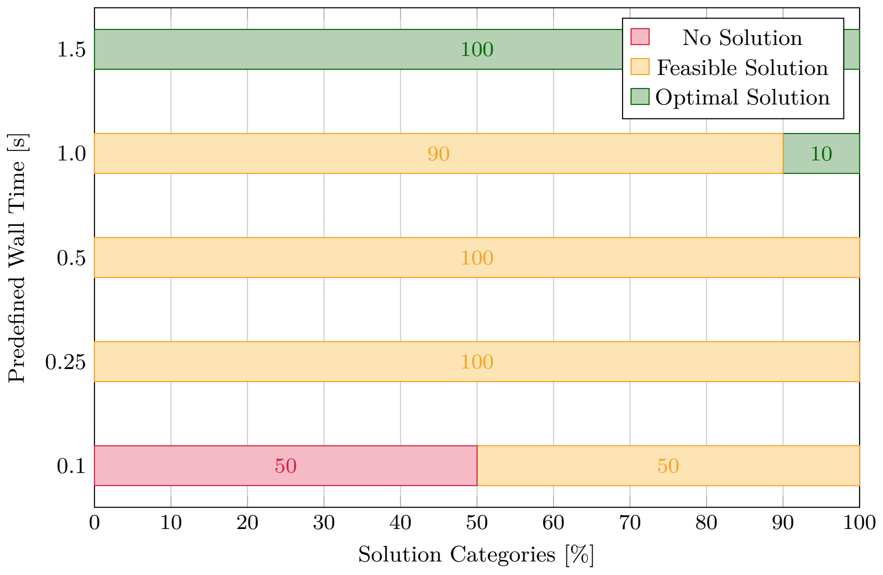


Fig. 10 Intersection scenario—solution categories

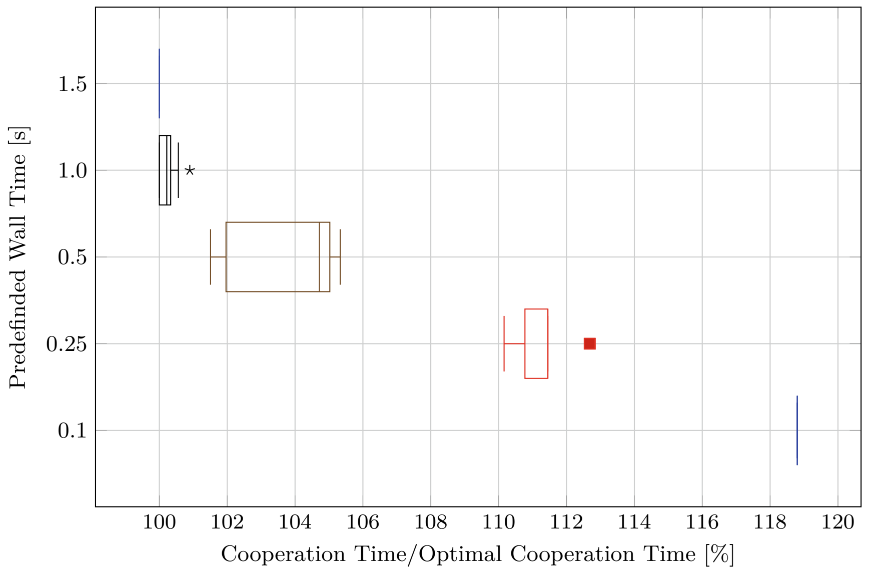


Fig. 11 Intersection scenario—cooperation time increase relative to optimal solution

mal solution brings significant speed advantages without a dramatic loss in solution quality and increase in maneuver duration.

2.4 Learning-Based Clustering

The architecture described in Fig. 6 successfully decouples the formation of Local Traffic Systems from the actual cooperation between the participating vehicles and the associated algorithm.

However, the limits for determining the LTS level are still statically defined by the developer. This static specification of the LTS parameters has several disadvantages. On the one hand, static optimization of the corresponding parameters is often suboptimal. The system is only adapted to a small set of possible conflict situations and traffic scenarios and is likely biased towards these scenarios used as test cases during the development. The administration and maintenance of a corresponding traffic scenario collection is time-consuming and often does not meet the requirement of completeness. On the other hand, there is no direct connection between the exchanged vehicle information shown in Fig. 5, the respective LTS Boundary Conditions and the capabilities of the underlying cooperation algorithm. However, changes to the underlying algorithms should logically also have an impact on the transmitted data as well as the LTS formation. Due to the disadvantages presented, a static parameter definition should be considered unsuitable for fully meeting the requirements of an LTS generation architecture described before.

A deep learning based approach offers a possible solution to the aforementioned problems. Here, the formation algorithm based on static parameters is replaced by a deep learning model. The model decides whether the LTS level should be increased or decreased, based on the exchanged vehicle data. The internal decision process is learned by the model during the training process based on a stimulative approach. The system can be trained in a simulation environment without managing a complex data set of conflict scenarios.

In this way, the model learns the link between the exchanged vehicle data and the underlying algorithms. The system learns not only the influence of a single parameter on the formation of the respective LTS level, but also the implicit relationships and similarities of individual traffic scenarios represented by the exchanged vehicle data. The detection of the traffic scenario takes place implicitly. If the underlying cooperation algorithms or the training scenarios are changed, the model can be trained again without additional changes.

However, a training process as described in Fig. 12 is unfortunately not applicable to the current problem. On the one hand, it is not possible to provide a static data set for training the deep learning model, since already after the first time step the LTS formation has an impact on the training environment surrounding the vehicle. Another central problem is caused by the time-delayed verifiability of the LTS formation for correctness. Common supervised/unsupervised deep learning approaches are based on presenting the model with input data based on a training data set. From this input data, the model generates the output data, which is then compared with a label. Labels are part of the training data set in the case of supervised learning and are generated from it in the case of unsupervised learning. The label is considered as the correct output of the model on the existing input data. The deviation from the

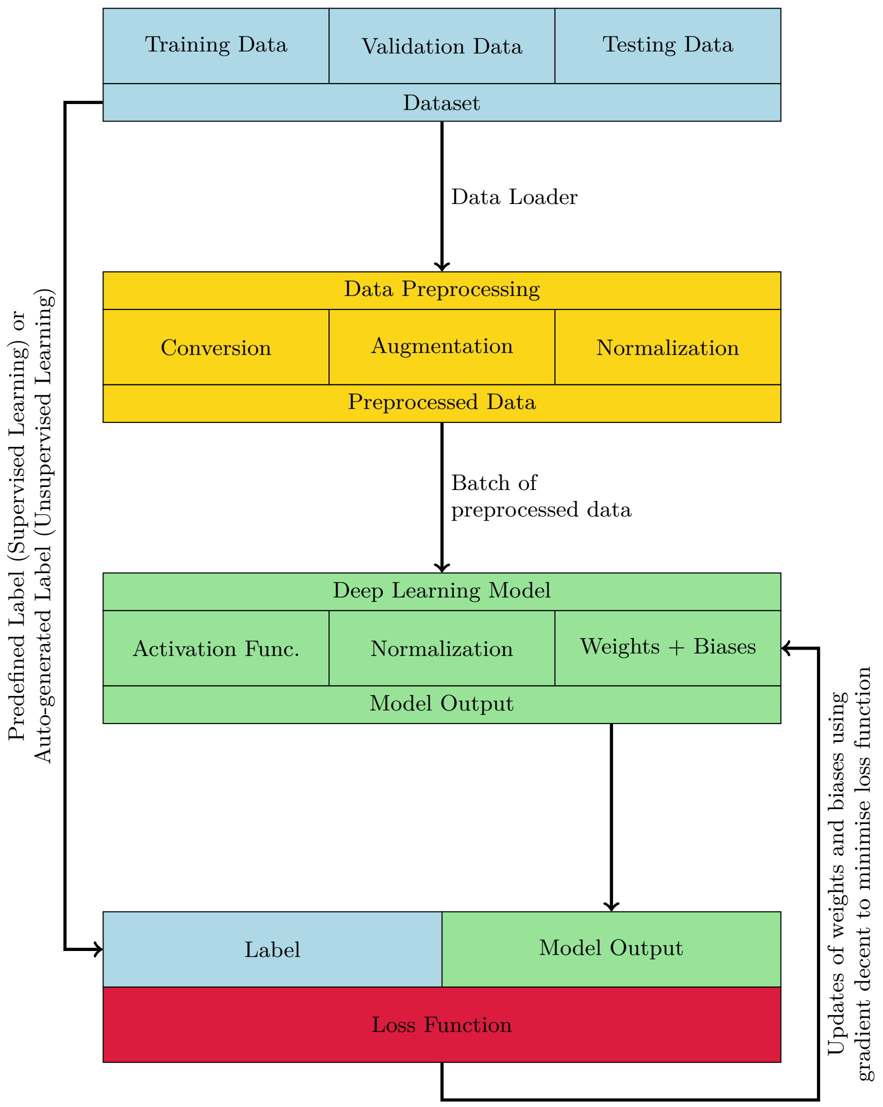


Fig. 12 Basic training architecture of supervised/unsupervised deep learning models

output of the model is represented by a loss function. The underlying parameters of the model are adjusted with the goal of minimizing the loss function. However, in the present case, such a label does not exist for a given set of input data. Whether an LTS formation was goal-directed becomes apparent only in the course of the executed cooperation maneuver several time steps after the actual LTS formation. Therefore, a reinforcement-based approach is used in the following.

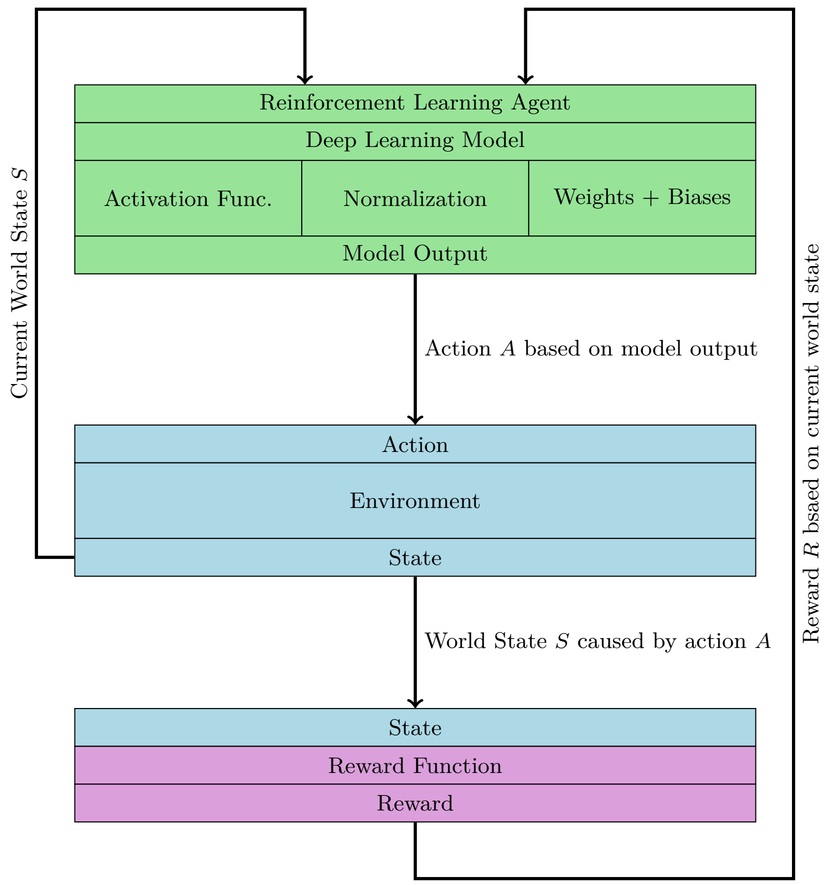


Fig. 13 Basic training architecture of reinforcement learning models

The diagram in Fig. 13 shows the general structure of a reinforcement learning algorithm. The algorithm consists of three main components. The reinforcement learning agent, the surrounding environment and a reward function. The reinforcement learning agent has the task to make optimal decisions based on the surrounding environment. The decision made by the agent at a time t is called action A_t. The action A_t is determined on the basis of the current environment. This is represented by the current state S_t. To evaluate the quality of a decision, the reward R_t is calculated by a reward function. Thus, the agent's goal is to maximize the total reward.

To transform the previous concept into a reinforcement-based approach, modifications to the architecture described in Fig. 6 are necessary. The changes are shown in Fig. 14.

A higher-level component RL-Agent is introduced. The previous algorithm based on static thresholds for determining whether a local traffic system is formed is removed from the LTS Manager. The LTS formation is made in the LTS Manager

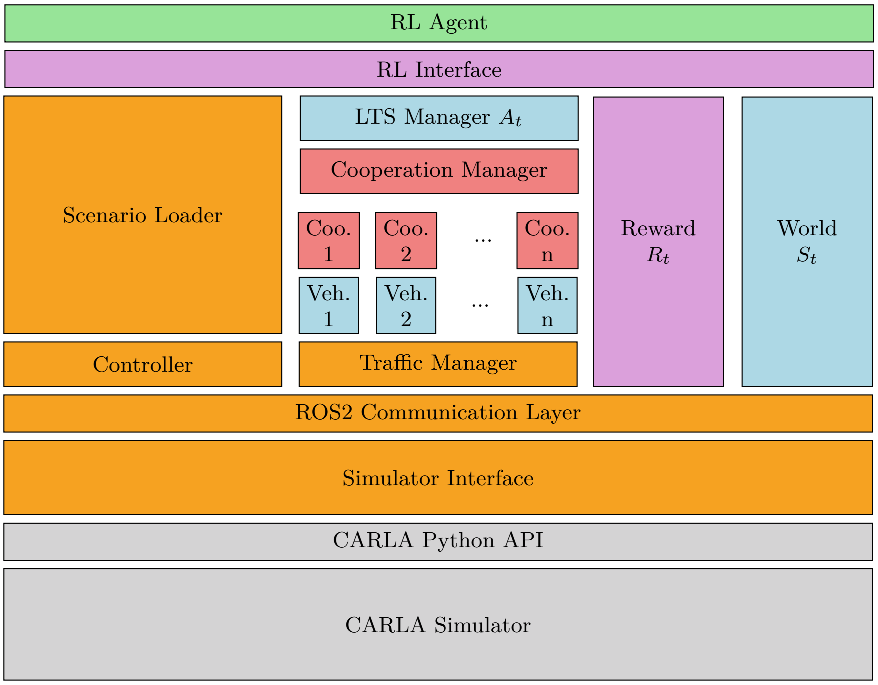


Fig. 14 Learning enabled Autoknigge architecture

on the basis of the action A_t by the RL agent. These actions are based on the current state S_t which is determined by the already existing World component. Furthermore, the previous pure data collection of the World component has been extended by a reward component to determine R_t. The required data S_t, R_t for the computation of A_t are provided by a reinforcement learning interface to the agent during training. The access to the Scenario Loader allows switching between different conflict scenarios during the training process.

2.5 *Example Cooperation Intersection and Highway Access*

The following section shows two example applications of the described concepts described and gives a visual impression about the cooperation result. The first scenario describes the conflict situation between two vehicles crossing an intersection. The second scenario describes the conflict situation at a highway access. The environment perception of the vehicles involved was completely deactivated. The only information exchanged is the data specified for LTS formation. Cooperative driving maneuvers are visualized within the simulator by a green line between the involved participants.

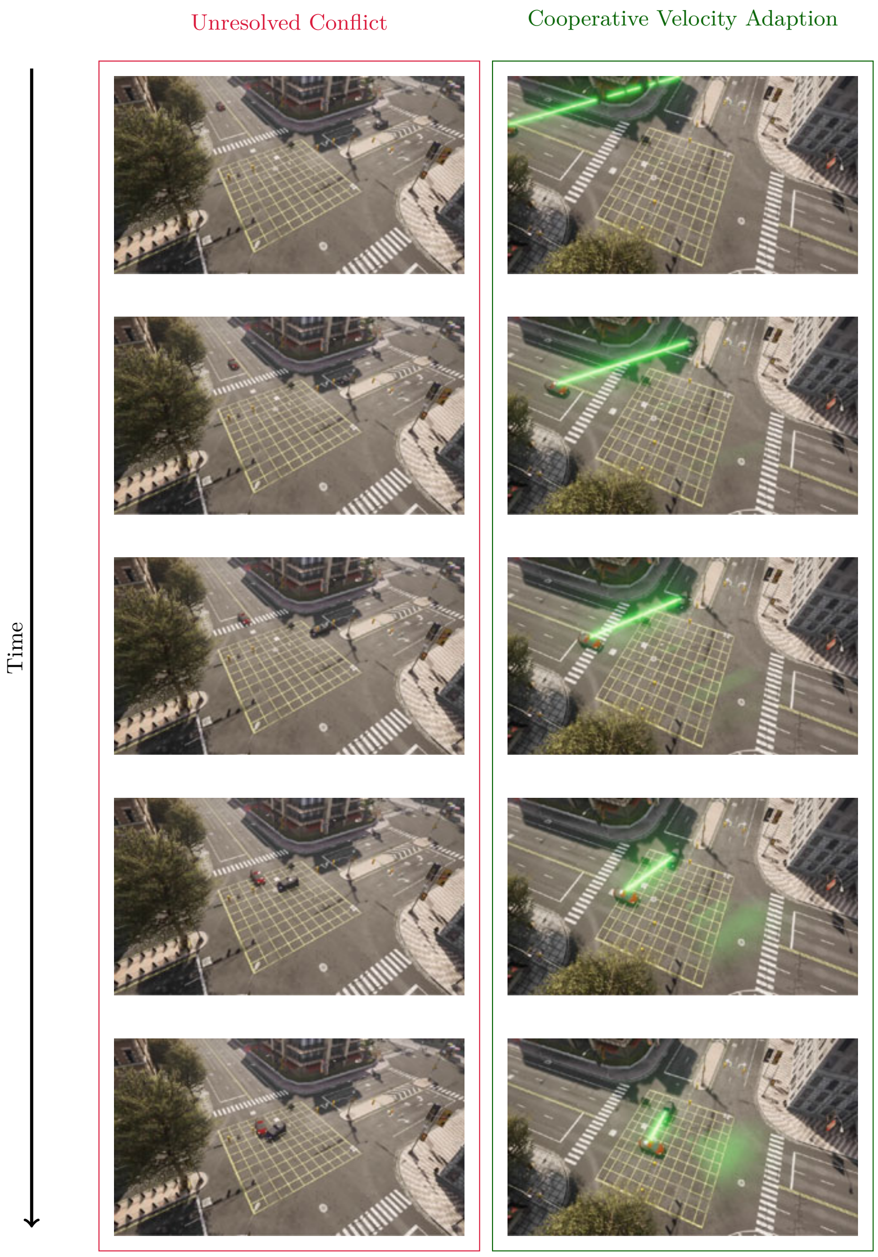


Fig. 15 Example cooperation intersection

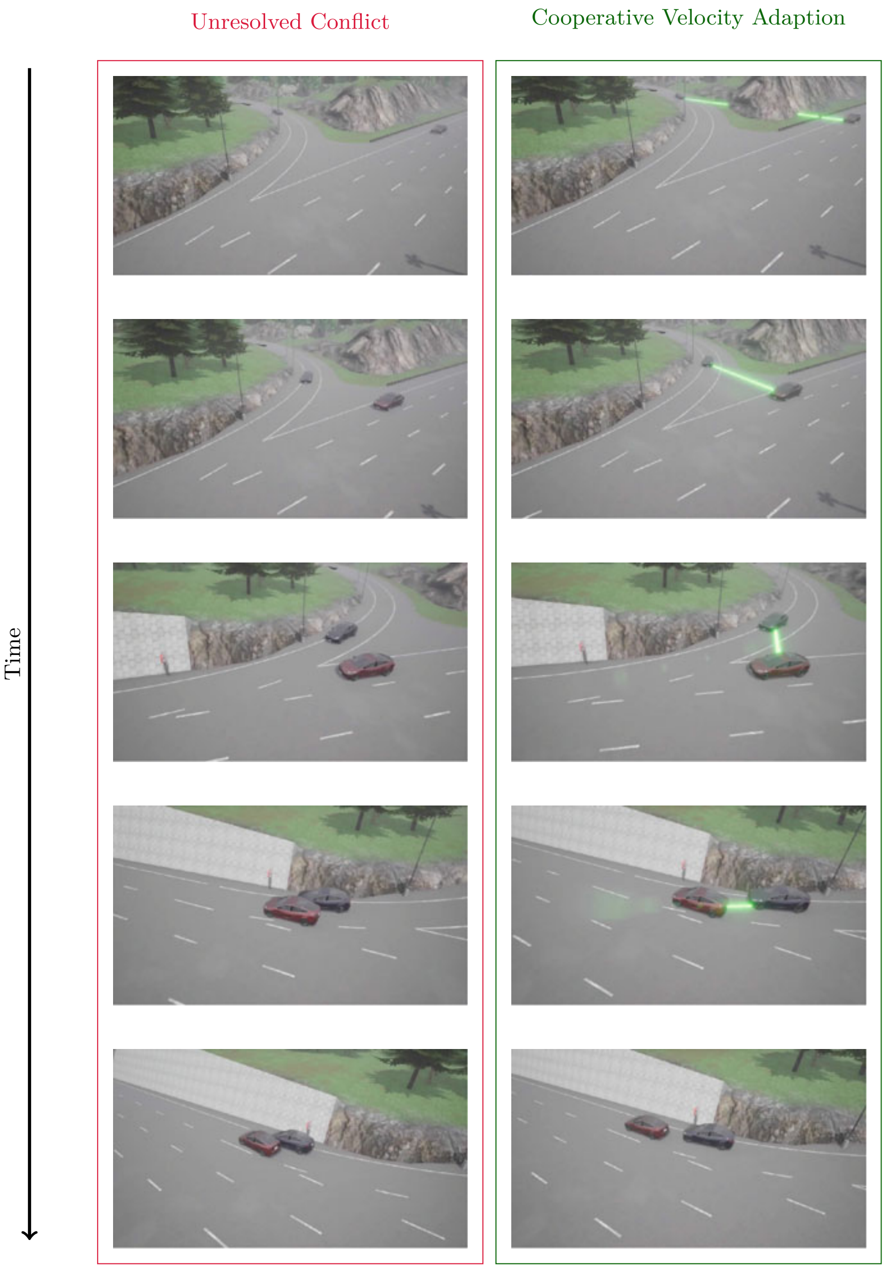


Fig. 16 Example cooperation highway access

As visible in Figs. 15 and 16, a short time gap was deliberately chosen in order to test the system at its limits. Both conflict situations are solved successfully on the LTS Level 1 by early cooperative adjustment of the vehicle speed. The fact that the present different conflict scenarios can be successfully solved with the same cooperation maneuver supports the chosen approach of using simple cooperation maneuvers, similar to human behavior, to solve different conflict situations.

2.6 Conclusion and Outlook

The presented architecture fulfills the requirements placed on the system. The transmitted vehicle data, the LTS formation as well as the underlying cooperation algorithms are successfully separated without neglecting the retroactive influences of the driving situation and cooperation algorithm on the LTS formation. Successful separation avoids the black box behavior of an end-to-end trained machine learning architecture. The cooperation algorithms are exchangeable. The introduction of LTS levels allows for easy prioritization in case of conflicting goals. The vehicle data assigned to the individual levels and the quantity of transmitted data, which increases proportionally to the urgency, as well as the constantly increasing interference in the longitudinal and lateral guidance of the vehicle, both reduce the necessary quantity of data for simple cooperation maneuvers and increase driving comfort.

Although the current approach is promising, there is still a need for research in the field of LTS education. This can be found in three main areas. First—The cost function. In addition to simplified basics such as a traffic flow optimization function, this should take into account other factors such as the CO_2 emissions of vehicles. Second—The underlying cooperation algorithms and the exchanged data. Since the focus of this work is on the optimization of approaches to LTS formation, there is still a high need for research in this area. In particular, as standardization continues, changes in V2X message definitions are to be expected. In the long run, V2X communication should be realized by frameworks like Veins, Artery [20] instead of ROS2 messages. Third—Further consideration of single-agent and multi-agent concepts of the reinforcement learning approach. The current system uses a single agent that learns the LTS formation. A multi-agent system where each vehicle uses an individual agent could offer significant advantages as there is no need to ensure that all vehicles have the same agent. This offers advantages in simplifying the learning process or realizing vehicle individual optimizations relevant to specific user preferences. Whether such a system contributes at all to the minimization of a global cost function if each agent follows an individual optimization remains to be researched.

3 Verification of Cooperative Interacting Automobiles

3.1 Introduction

This section proposes an approach to use formal methods for verifying trajectories in our LTS framework. Our algorithm generates behavior patterns that guarantee collision-free and deadlock-free trajectories. In order to generate the behavior patterns, we use the model checker nuXmv [6], which is specialised in synchronous finite-state systems.

This section is structured as follows. We introduce our verification architecture in Sect. 3.2. Section 3.3 presents the offline part of our approach, i.e., our modeling and verification of traffic scenarios and the generation of rule sets. Section 3.4 introduces the implementation of our rule checker and Sect. 3.5 evaluates the verification and rule checker. Finally, Sect. 3.6 concludes this section.

3.2 Verification Architecture

Figure 17 shows our verification architecture from [30]. The verification works in an offline and an online part. The offline part consists of modeling and verification of traffic scenarios. The verification classifies the traffic scenarios as collision-free and deadlock-free or provides a counter example in case of possible collisions or deadlocks. We generalize the counter examples to traffic rules for networked and autonomous vehicles. The traffic rules are stored in a rule set. The online part is a rule checker, which uses the map and planned trajectories of the current driving situation and the rule set generated by the offline part as input. The rule checker checks if the trajectories comply with the traffic rules of the rule set. If no rule is violated, the trajectories are considered safe.

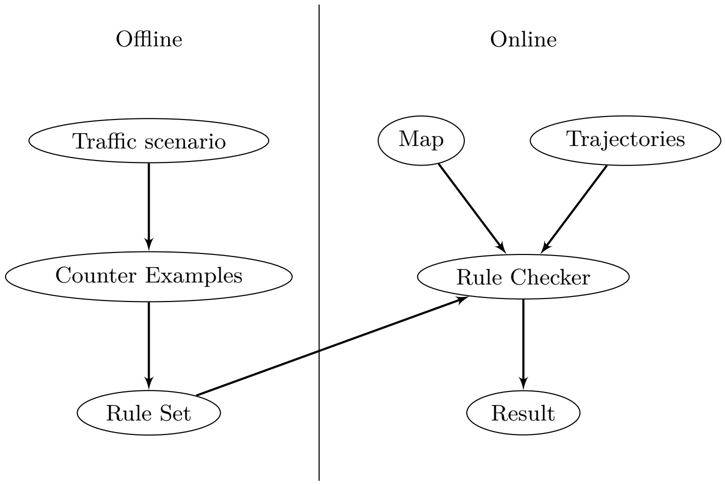


Fig. 17 Verification architecture of [30], consisting of an offline and an online part. Before deployment, counter examples of safety verification are generalized into rules that guarantee the absence of collisions and deadlocks. At run-time, a rule checker classifies trajectories of vehicles into safe and unsafe trajectories, depending if they follow the rules for the scenario

3.3 Rule Set Generation

We decompose the traffic scenario model into two parts: the map and vehicles' trajectories. Through this modular approach, it becomes easier to develop general purpose encodings for vehicles and maps independently of each other. We call the map the *static model* and we call the trajectory model *dynamic model*. We model both components time and spatial discrete. Section 3.3.1 and 3.3.2 summarize our modeling of [30, 46]. Section 3.3.3 introduces our extension to combined models of connected LTS. Section 3.3.4 presents our NuXmv encoding and Sect. 3.3.5 introduces the rule generation.

3.3.1 Roadway Model

The map consists of blocks and transitions. Each block represents a part of the physical road. Blocks are non-overlapping and identified by unique Identities (IDs). A discretization takes care of the vehicles' dynamics and safety distances. Each vehicle can occupy only one block at each time step. If a vehicle holds a block, the block is *occupied*, otherwise the block is *free*. To model valid transitions between blocks, each block has a list of successor-tuples.

Definition 1 (*Successor-Tuple* [30]) Successor-tuples are defined as

$$t_{suc} = (ID_{suc}, Cost, Watchlist, I), \tag{1}$$

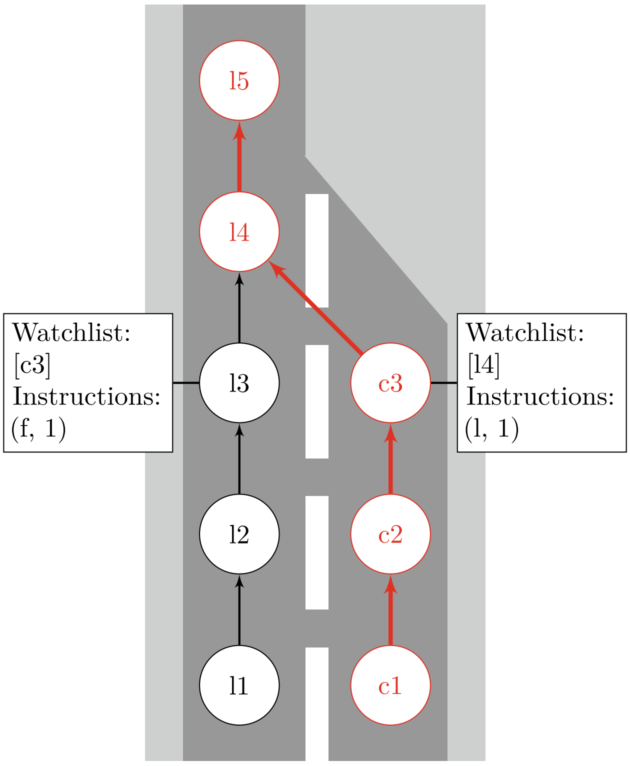


Fig. 18 Example model of a narrowing road, adapted from [30]

where:

- ID_{suc} denotes the ID of the successor block,
- $Cost$ stores the costs for the transition,
- $Watchlist$ is a list of block IDs. A vehicle can only use the transition if all blocks in its watchlist are free, and
- $I = (Type, Velocity) \in (String \times \mathbb{Z})$ is a scenario-dependent instruction. $Type$ describes which behavior is expected by the vehicle, e.g., “move forward”, “turn right”, and “switch to left lane”.

Figure 18 shows an example model of a narrowing scenario. Only physically possible transitions respective to road boundaries and vehicle dynamics are included.

3.3.2 Trajectory Model

Trajectories consist of a sequence of adjacent blocks. The first block of a trajectory is the vehicle’s current position. The last block represents the vehicle’s destination.

Each time step, the vehicles transit to the next block in their trajectories. The same block may be used multiple times in a trajectory. Trajectories can have different lengths. After leaving the LTS, the vehicle moves into a final block with the ID *n* with no further process. One of the following two statements hold for any consecutive blocks in each trajectory:

1. The blocks have the same ID, i.e., the vehicle does not move.
2. There is a valid transition between the blocks in the direction of movement.

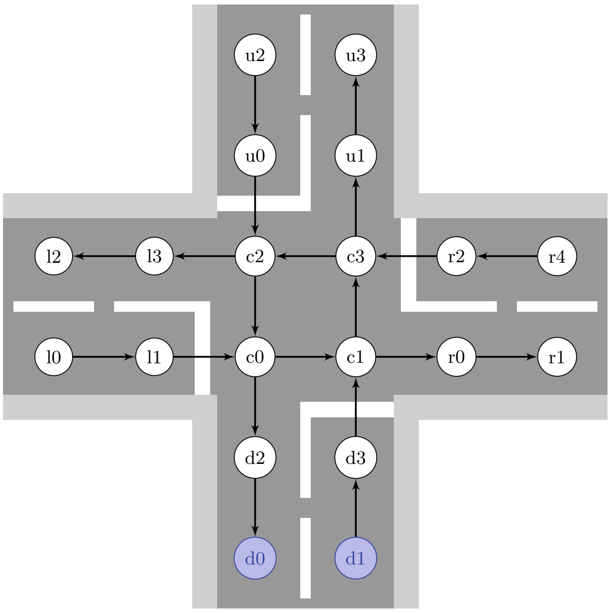
In traffic scenarios, some vehicles may be important for more than one LTS. We propose a method to create connected traffic scenarios. A connected traffic scenario combines two traffic scenarios with transitions from one traffic scenario to the other traffic scenario. Using these transitions, vehicles can travel between both traffic scenarios. In connected traffic scenarios, different rules may apply in comparison to separate traffic scenarios. Our approach extends the methods from our previous works done in [30, 46] by connecting traffic scenarios and generating rules for connected traffic scenarios. We classify pairs of traffic scenarios into overlapping traffic scenarios and non-overlapping traffic scenarios. Overlapping Traffic scenarios are scenarios where both single scenarios have entrance blocks, which have the same ID. In Fig. 19 two single traffic scenarios are sketched. Both have blocks with the same IDs.

3.3.3 Connected LTS

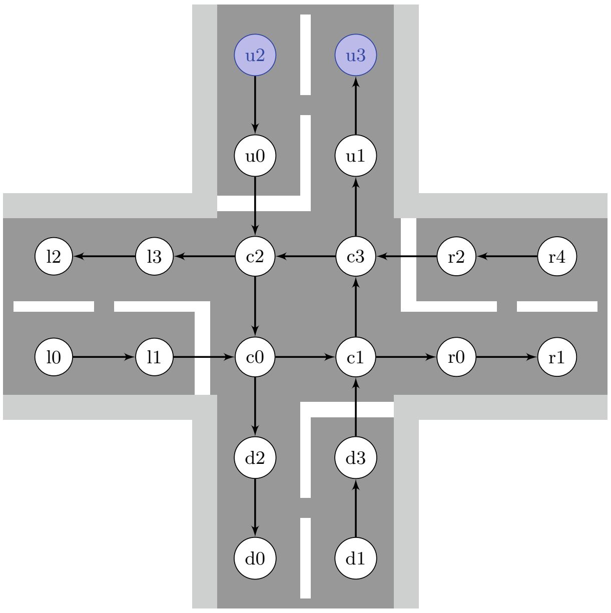
In order to verify connected LTS, this subsection extends the modeling of Sects. 3.3.1 and 3.3.2. We start with an example of collision-free and deadlock-free single LTS, while the combination of both LTS is collision-free but not deadlock-free.

Motivating Example

In the following, we give an example of rule sets generated for single traffic scenarios that do not provide deadlock-freeness in overlapping LTS. The rules were generated by our method in [30]. In this example, we use two overlapping intersections, both with 4 entrances. Each single intersection has only one rule. This rule does not allow vehicles in the center to drive in 4 different directions. In Fig. 20 we can see an initial configuration. Each center of the model is filled with vehicles and all cars try to reach the end of the opposite center. Using this configuration, we were able to show that this rule is not sufficient to avoid deadlocks. As seen in Fig. 21, this configuration leads to a deadlock in both centers of the intersections. In the upper part, both vehicles on position c0 and d3 try to move to c1. Since only one vehicle may occupy block c1, a deadlock is caused. The same holds for position c2 in the bottom part, which is blocked by vehicles at position c3 and u0. These two situations cause a deadlock, since every vehicle tries to take the entrance to get to the opposite center and block one another. The entrances are blocked by the vehicles on block d2 in the upper part and u1 in the bottom intersection. Both vehicles cannot make any progress. This example shows that rules that apply for a single scenario must



(a) Model 1, where the bottom entrance is overlapping with model 2



(b) Model 2, where the upper entrance is overlapping with model 1

Fig. 19 Two single traffic scenarios, here crossroads, with overlapping borders in blue

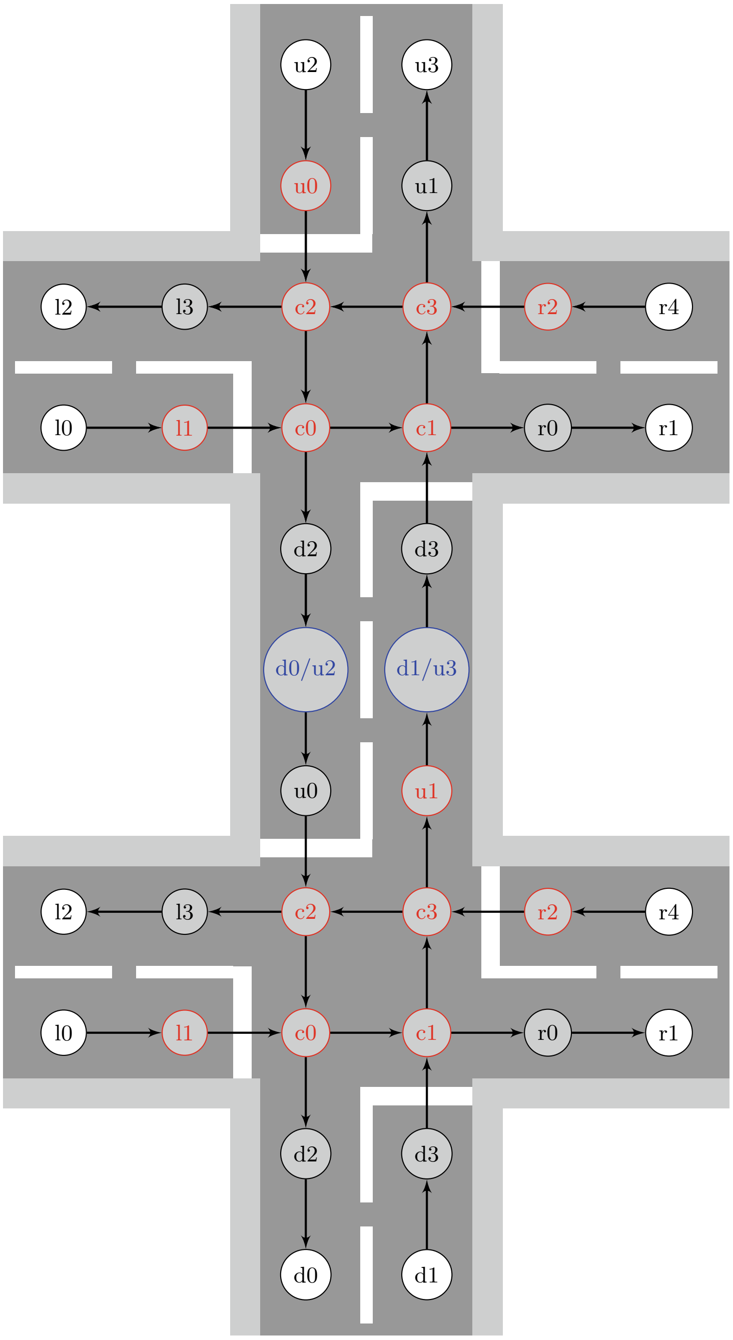


Fig. 20 Starting positions of vehicles for deadlock scenario in nuXmv

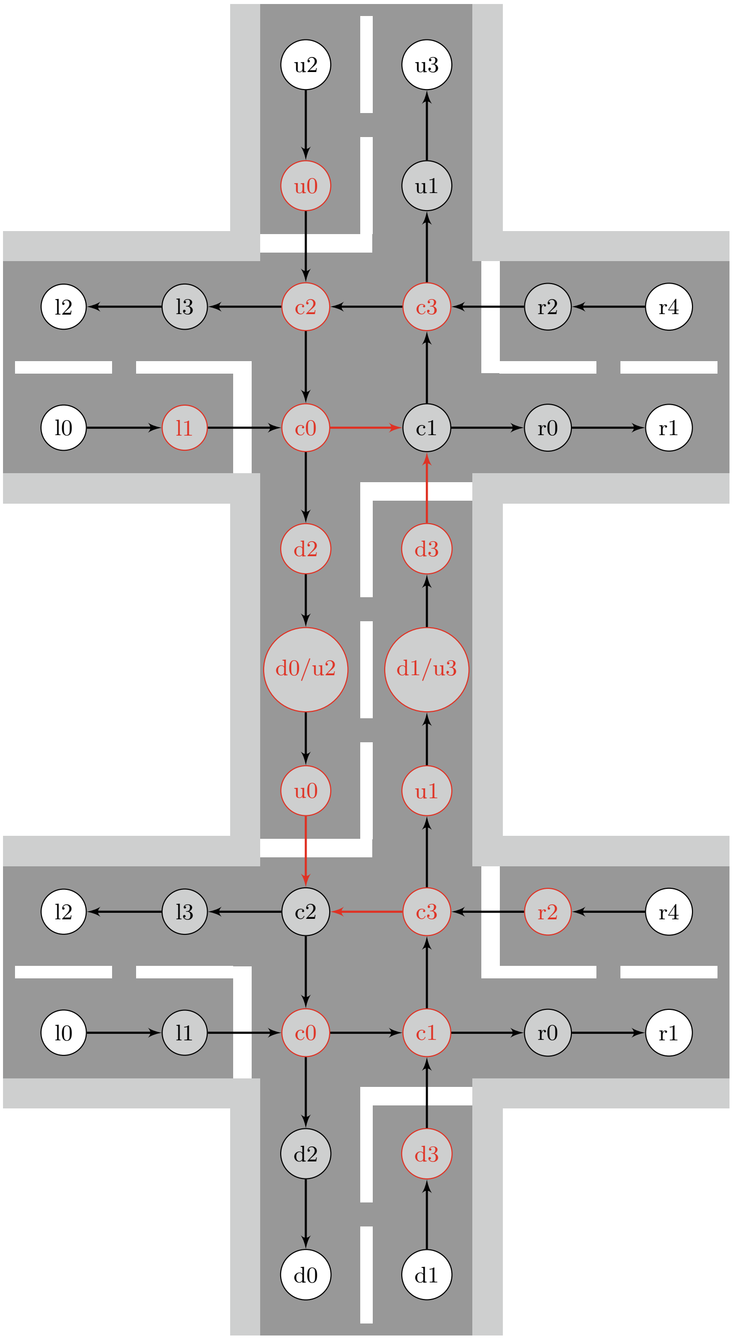


Fig. 21 Deadlock situation for a connected traffic scenario

not be enough to guarantee safety properties also in the connected traffic scenario. Therefore we generated rule sets for overlapping LTS.

The rest of this subsection introduces our model of connected LTS in order to guarantee collision-freeness and deadlock-freeness in connected LTS.

Border Blocks

Each entrance consists of multiple blocks. We divide these blocks in three categories:

1. blocks which border to a center block c1, c2, c3 or c4, e.g., the blocks l1 and l3 in Fig. 19a,
2. blocks which leave or enter the traffic scenarios, e.g., l2 and in Fig. 19a, and
3. all other blocks that are forming the middle of each lane.

In the following, we will call blocks of category 2 border blocks. Border blocks are critical states, because taking only single traffic scenarios into consideration when verifying for safety properties, e.g. collisions, everything that happens outside the current traffic scenario is not considered. For example, it could be that a vehicle's position is currently a border block which is the exit of the traffic scenario. Not considering the next traffic scenario after the border block could lead to deadlocks or collisions. Verifying only separate traffic scenarios could lead to false negative classification.

Transition States

There may be traffic scenarios that are not overlapping. In this case, no border blocks lead from one traffic scenario into another traffic scenario. We model connection points between traffic scenarios. Possible connection points are two border blocks, each belonging to the other traffic scenario. These two border blocks form a *connection pair*.

We use lane information to identify connection pairs of multiple traffic scenarios. In order to be connectable, the traffic scenarios require the same number of lanes. If two traffic scenarios have the same number of lanes, we use the lane positions to identify the border blocks of both traffic scenarios that form a connection pair. We introduce an argument *TransitionStates* to model connection pairs in nuXmv. The argument TransitionStates contains the following information:

- the number of border blocks contained in the input traffic scenario,
- the successor blocks of each border block,
- the corresponding lane to which a block belongs,
- the number of lanes existing in the input traffic scenario, and
- each lane's position compared to the other ones in the same entrance.

The argument TransitionStates is a list that contains all possible blocks for a connection pair. Each element of the list represents an entrance or exit of the traffic system.

Definition 2 (*Entrance*) An entrance consists of multiple blocks that form a group of pairing border blocks. In Fig. 19a, the pairs (l0, l2), (d0, d1), (r1, r4), (u2, u3) form four entrances.

Our method checks for opposing border blocks and create connection pairs. Then, a transition is created between the corresponding border blocks. The transition starts at the border block which is an exit block of its traffic scenario and is connected to the corresponding input block from the connection pair.

An example combined model is the model in Fig. 20. The combination of multiple LTS increases the size of the scenarios to be verified. Since large models cause performance issues during verification, we reduce the combined model, while maintaining the correctness of verification.

Model Reduction

We reduce the models of combined LTS to keep computational efficiency of the offline verification. To this end, we reduce the number of blocks in the resulting model. We include the center blocks of both single LTS models and all blocks connecting the center blocks. Each center block that lead to an exit state becomes an exit state, while each center block connected to an entrance becomes an entry block. The gray states in Fig. 20 are the states included in the reduced model of connected LTS.

3.3.4 NuXmv Encoding

Based on our work in [46], we translate our models into the nuXmv input language. NuXmv distinguishes four input types: variables, transitions, dictionaries, and specifications.

Variables

We model vehicles as variables in nuXmv. The possible states of each variable are the blocks the corresponding vehicle will occupy in the scenario. In the example shown in Fig. 18, Vehicle v_0 has the following path: c0 - c1 - c2 - c3 - l4 - l5.

In nuXmv, the path is represented as follows:

```
VAR
    v0 : {c0, c1, c2, c3, l4, l5, n};
```

The initial state of each vehicle is the first block of its trajectory, e.g., vehicle v_0 in Fig. 18 starts at block c0. The corresponding nuXmv code is

```
INIT
  v0 = c0.
```

Transitions To encode transitions, we use the *case* statement for every pair of consecutive blocks in a vehicle's trajectory (B_i, B_{i+1}). We use the following two statements:

$$(Pos = B_i)\&(\phi) : B_{i+1} \tag{2}$$

$$(Pos = B_i)\&(\neg\phi) : B_i, \tag{3}$$

where Pos denotes the current block and ϕ is a Boolean expression. ϕ evaluates to true, if the transition is safe to use, i.e., if all blocks in the watchlist of this transition are free. Equation (2) allows the vehicle to move to its next block B_{i+1}, if the transition is safe. Equation (3) forces the vehicle to remain in its current block, if the transition is not safe. The block c0 in Fig. 18 is given as

```
c0: [(c1, 1, [c1], (f,1))],
```

where $ID_{suc} = c1$, $cost = 1$, $Watchlist = [c1]$, and $I = (f, 1)$. The instruction I states that the vehicle moves forward one block.

Suppose an example vehicle v_1 that moves from c0 to c1 in the scenario in Fig. 18. If there is another vehicle in the LTS with ID v_0, the nuXmv statements for this transition are as follows:

```
((v1) = (c0)) & ((v0) != (c1))
                                        : c1;
((v1) = (c0)) & (!((v0) != (c1)))
                                        : c0;
```

The first statement states that if vehicle v_1 is on block c0 and vehicle v_0 is not on block c1, v_1 moves to c1. The second statement states that if vehicle v_0 is on block c1, vehicle v_1 remains on block c0.

This is done for every pair of consecutive blocks in the vehicle's trajectory. Once a vehicle reaches the last block of its trajectory, it moves to block n and stays there through the following equations:

$$Pos = B_{end} : n \tag{4}$$

$$Pos = n : n \tag{5}$$

Equation 4 causes all vehicles to move to block n after their trajectory ended. Equation 5 states that vehicles that reached block n will remain there.

Dictionaries

We use dictionaries to represent LTS entrances. Each entrance is represented by two dictionaries, because there are always at least two lanes per entrance, one exit and one entrance into the traffic scenario. Each dictionary can have multiple border blocks. The number of elements in this dictionary represents the number of lanes of the corresponding entrance's exit or entry and the position of a block in this dictionary represents its corresponding lane, to which the block belongs. An example TransitionStates argument for the intersection of Fig. 19a looks like the following:

```
[
# border blocks of upper entrance
  {'u2': [('u0', 1, ['u0'], ('f', 1))]}
, {'u3': [('u3', 1, ['u3'], ('s', 0))]}
# border blocks of right entrance
, {'r4': [('r2', 1, ['r2'], ('f', 1))]}
, {'r1': [('r1', 1, ['r1'], ('s', 0))]}
# border blocks of bottom entrace
, {'d1': [('d3', 1, ['d3'], ('f', 1))]}
, {'d0': [('d0', 1, ['d0'], ('s', 0))]}
# border blocks of left entrance
, {'l0': [('l1', 1, ['l1'], ('f', 1))]}
, {'l2': [('l2', 1, ['l2'], ('s', 0))]}
].
```

The first two elements represent the border blocks of the upper entrance, the next two the border blocks of the right entrance, the next two the border blocks of the bottom entrance, and the last two for the border blocks of the left entrance. Each block that is an element of an entrance lane stores the information about its successor block.

Specifications

We verify the safety of our traffic system. To this end, we formulate *specifications* by *invariants* and *temporal logic*. We give more details on the specifications in [30].

We use invariants to check for collision-freeness. A collision occurs, if multiple vehicles occupy the same block at the same time. The invariants to check collision-avoidance are

$$(pos_1 \neq n) \Rightarrow (pos_1 \neq pos_2), \tag{6}$$

where pos_i is the position of vehicle i. Equation (6) models that two vehicles 1 and 2 do not occupy the same block, unless vehicle 1 finished its trajectory and moved to the end block n. We check this invariant for each pair of vehicles at each time step.

We use temporal logic to check deadlock-freeness. We use Linear Temporal Logic (LTL) [43]. In LTL, we model deadlock-freeness as

$$F(pos_1 = n \wedge pos_2 = n \wedge \ldots), \tag{7}$$

where $F(\cdot)$ denotes the *eventually* operator of LTL. Equation (7) models that each vehicle eventually reaches block n, i.e., finished its trajectory.

3.3.5 Summarize Rules

We alter the static and dynamic models to create different verification scenarios. NuXmv provides counter examples if a verification scenario is not collision-free and

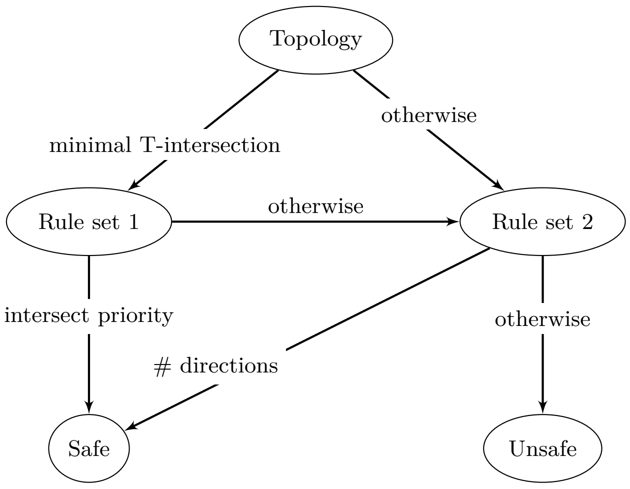


Fig. 22 Evaluations of rule sets in an intersection scenario. Different rule sets apply, depending on the intersection topology

deadlock-free. We generalize counter examples derived from the same group of static models to generate rules for this group. The rule sets formulated for each scenario group prevent any collisions or deadlocks found during verification. Depending on the static and dynamic model, different rule sets have to be applied.

We demonstrate the generalized rule sets for intersection models. In the following, “inner lanes” refer to the leftmost lane of each direction and “center” refers to the area of the intersection, where the lanes intersect. In the intersection model, the rule set depends on the intersection topology and the priority rule, i.e., the right of way, applied in the trajectories. Figure 22 gives an overview of the rule set selection process. If the map represents a T-intersection with only one lane in each direction, it is called a *minimal* T-intersection. For minimal T-intersections, we need to check the priority rule for vehicles, denoted as Rule set 1. If the vehicles in the intersection consistently have priority over vehicles outside, the rules of Rule set 1 are met and the trajectories are always safe to execute. In all other cases, Rule set 2 is applied. In Rule set 2, trajectories are considered safe if the center never has vehicles traveling in four different directions, denoted by the red arrows in Fig. 23.

3.4 Rule Checker

The rule checker takes the static and dynamic model, i.e., the map and trajectory data, as input. The output of the rule checker is the classification of the trajectories according to the rule sets generated in Sect. 3.3. The rule checker classifies the trajectories into *safe* and *unsafe* trajectories. The rule checker detects the vehicle’s

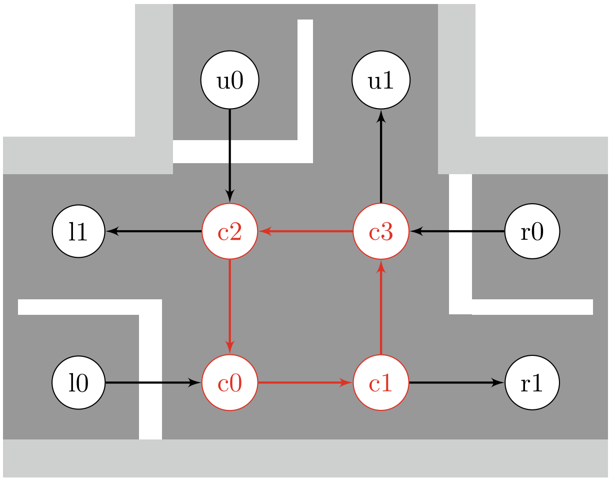


Fig. 23 Minimal T-intersection deadlock example

behavior, e.g., the applied priority rules. Different scenarios have different rule sets. We form groups of scenarios with similar rule sets. We demonstrate the idea using the intersection example in Fig. 19a.

One behavior that need to be identified is if the vehicles inside or outside the intersection have priority at the entrances. In the intersection model shown in Fig. 19a, the blocks c1 and r2 are a pair of interest, since vehicles on both blocks are able to move into c3. The rule checker checks for all pairs of interest if the following two conditions are satisfied at each time-step:

- both blocks in the pair of interest are occupied, and
- the vehicle in the intersection does not leave the intersection.

If both conditions are satisfied, the rule checker checks the next instructions of the vehicles. If the vehicle in the intersection is the only one instructed to move forward, then we have a case where vehicles in the intersection have priority. If the vehicle that tries to enter the intersection is the only one instructed to move forward, then we have a case where vehicles entering the intersection have priority. If none of the mentioned possibilities happened, then we cannot decide what has happened and conclude that there is no consistent priority rule between them. There are three possible cases after the rule checker iterated over each pair of interest:

- vehicles in the intersection consistently have priority over vehicles outside of the intersection,
- vehicles that enter the intersection consistently have priority over vehicles in the intersection, and
- there are no consistent priority rules.

We formulate rules for all three cases.

3.5 Evaluation

We evaluate the feasibility of our verification process for single LTS in Sect. 3.5.1 and for connected LTS in Sect. 3.5.2. Moreover, we evaluate the computation time of the offline verification and online rule checker in Sect. 3.5.3.

3.5.1 Feasibility in Single LTS

This section presents evaluation results of the verification process for single LTS. We evaluate the generation of rule sets and the performance of the rule checker. We divide combinations of roadways and trajectories into different classes. We evaluate scenarios of multiple classes.

We define classes of roadways according to their generated rule sets. Within the same model class, different rules need to be applied depending on the road topology and vehicles' trajectories. For example of an intersection model, the rules to apply depend on the number of entries of the intersection and the trajectories' priority rules. As such, a T-intersection has 3 potential classes:

1. The model is a T-intersection with one lane in each direction, vehicles in the intersection area always have priority over vehicles that are outside of the intersection.
2. The model is a T-intersection with one lane in each direction, all vehicles give priority to vehicles on the right.
3. The model is not a T-intersection with one lane in each direction.

We evaluate our rule checker on a four-way intersection. Table 1 presents the input trajectories for vehicles v_1, v_2, v_3, and v_4 and compares the expected and actual rule checker results. Figure 24 visualizes the first example of Table 1. The roadway is a minimal T-intersection and the vehicles' trajectories give priority to vehicles in the intersection. Please note that the lower entrance (the blocks d0 and d1) are not included in the T-intersection model. The rule checker gives the expected results in all cases. It classifies collision-free and deadlock-free scenarios as safe and unsafe otherwise. Nevertheless, the rule checker may classify collision-free and deadlock-free scenarios as unsafe. Figure 25 shows such a false positive result. The rule checker rejects these trajectories since vehicles in the center are traveling in all four directions on a non-minimal T-intersection. However, the rule checker will not classify unsafe scenarios as safe.

3.5.2 Feasibility in Overlapping LTS

We extend the rule sets for single LTS to guarantee collision-free and deadlock-free trajectories also in connected LTS. As an example we present two new rules for the intersection scenario:

Table 1 Evaluation scenarios for the intersection model. We show the trajectories of four vehicles in each scenario, the expected result and the actual result of the rule checker. The trajectories of the first evaluation are visualized in Fig. 24, represented by the corresponding color

Minimal T-intersection with priority of vehicles in the intersection		
Input trajectory	Expected	Result
v_1: c2 - c0 - d1 v_2: l0 - l0 - c0 - c1 - r1 v_3: d0 - c1 - c3 - c2 v_4: r0 - c3 - c2 - l1	Safe	Safe
v_1: c2 - c0 - d1 v_2: c0 - c1 - r1 v_3: c1 - c3 - c2 - l1 v_4: c3 - c2 - l2	Safe	Safe
Any other intersection model		
Input trajectory	Expected	Result
v_1: c2 - c2 - c0 - d1 v_2: l0 - c0 - c1 - r1 v_3: d0 - c1 - c3 - c2 v_4: r0 - c3 - c2 - l1	Unsafe	Unsafe
v_1: c2 - c2 - c0 - d1 v_2: l0 - c0 - d11 v_3: d0 - c1 - c3 - c2 v_4: r0 - c3 - c2 - l1	Safe	Safe
v_1: r0 - c3 - u1 v_2: u0 - c2 - c0 - d1 v_3: l0 - c0 - c1 - r1 v_4: d0 - c1 - c3 - u1	Safe	Safe
v_1: r0 - c3 - c2 - l1 v_2: u0 - c2 - c0 - d1 v_3: l0 - c0 - c1 - r1 v_4: d0 - c1 - c3 - u1	Unsafe	Unsafe

- Vehicles entering the center must be able to exit the center. We refer to this rule as *exit free*.
- Vehicles may not leave the center in the same entrance, which was used to enter the center. We will call this rule *entry and exit differ*.

All developed models have been checked for correctness. To verify the correctness of the scenario 2-intersection, we generate trajectories. Since the 2-intersection model is a connection of two single crossroad scenarios, we only generated trajectories that are valid for single intersection models. Tables 2 and 3 show the test cases for the newly generated rules *exit free* and *entry and exit differ*. For each rule, we show an expected positive classification and an expected negative classification. The results

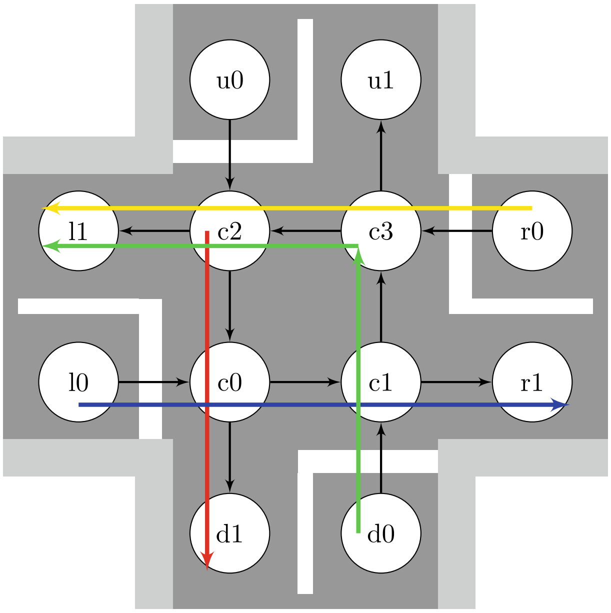


Fig. 24 Visualization of Table 1

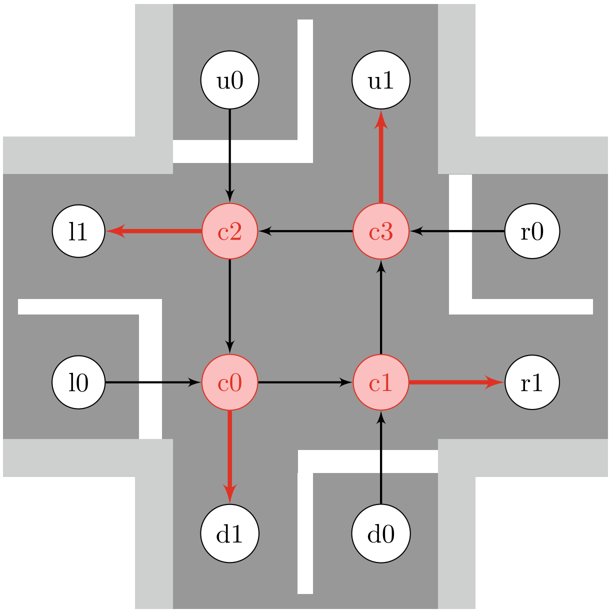


Fig. 25 False-positive result

Table 2 Rule *entry and exit differ* test cases for traffic scenario 2-intersection

Input trajectories						Expected	Results
u2	u0	c2	c0	d2	d0	Safe	Safe
l1	c0	c1	c3	u1	u3		
d3	c1	c3	c2	l3	l2		
r4	r4	r3	c3	u1	u3		
u0	c2	c0	c1	c3	u1	Unsafe	Unsafe
l1	c0	c1	c3	u1	u3		
d3	c1	c3	c2	l3	l2		
r4	r4	r3	c3	u1	u3		

Table 3 Rule *exit free* test cases for connected traffic scenario 2-intersection

Input trajectories								Expected	Results
l0	l1	c0	c1	c3	u1	u3	*n*	Safe	Safe
d1	d3	c1	c3	c2	l2	*l*4	*n*		
r4	r2	c3	u1	u3	*n*	*n*	*n*		
l0	l1	l1	c0	c1	c3	u1	u3	Unsafe	Unsafe
d1	d3	c1	c3	c2	l2	*l*4	*n*		
r2	c3	u1	u3	*n*	*n*	*n*	*n*		

of the rule checker were as expected. Both rules are also valid in single intersection scenarios.

3.5.3 Computation Time

As extension to our evaluation in [30], we evaluate the computation time of the rule generation for connected LTS. We measured the computation times on a laptop running nuXmv 2.0.0 on Windows 10 using a processor with 2x 3.20GHz and 8 GB RAM.

Offline Computations

We present the results for the 2-intersection scenario. Each intersection model consists of 4 entries, each consisting of one lane. We execute 100 runs per measurement. Figure 26 shows the results. The execution time increases exponentially for an increasing number of vehicles. For 6 vehicles, the execution time is less than 15 s. For more vehicles, the execution time increases to around 1.7 min for 12 vehicles.

Online Performance

Figure 27 shows the execution time of one run of the rule checker. The execution times are average values of 100 runs to reduce measurement inaccuracies. For up to

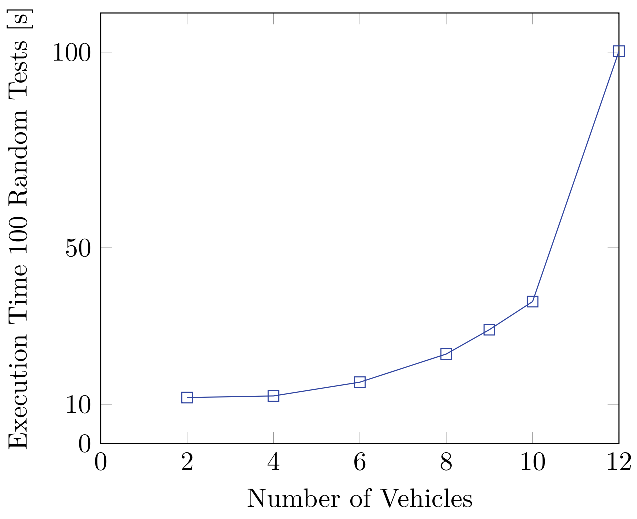


Fig. 26 Execution time of counter example generator for model of length 1

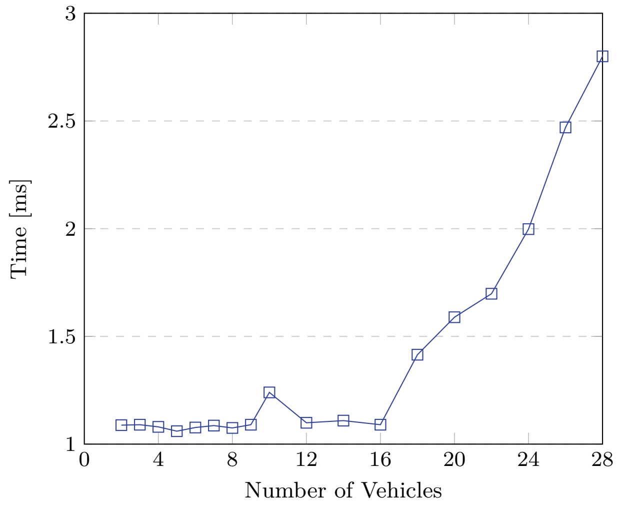


Fig. 27 Execution time of rule checker

16 vehicles, the execution time is almost constant and below 1.5 ms. For more than 17 vehicles, the execution time increases to around 2.8 ms for 28 vehicles.

3.6 Conclusion

This verification method is able to verify collision-freeness and deadlock-freeness of trajectories in one or more LTS. We summarized the space and time discrete model of [46] and the verification architecture of [30]. Our verification architecture generalizes counterexamples of the offline verification to generate rule sets. The rule sets restrict the solution space of valid trajectories so that all trajectories that fulfill the corresponding rule set are collision-free and deadlock-free. This work considered the verification of connected LTS. Evaluation results show that this method is real-time capable even for scenarios with a high number of vehicles. Further research may include evaluations on more complex LTS with more than one overlap.

4 Modeling Dynamic Systems

4.1 Why Modeling?

Vehicles of all SAE levels are safety critical systems and hence, their development needs to comply with legal regulations and safety standards. For instance, the functional safety standard ISO26262 highly recommends the usage of formal and semi-formal notations, hierarchical components of restricted size, the usage of strong type systems, range and plausibility checks, as well as the avoidance of hidden data-flows. In this section we are going to discuss the EmbeddedMontiArc (EMA) language family, a model-driven design approach for dynamic cyber-physical systems such as cooperative vehicles based on the component-and-connector (C&C) principle [37, 38]. The C&C paradigm views a software system as a composition of hierarchically organized components communicating with each other over connectors. The approach can help the development team to enforce the design principles required by ISO26262 by providing a domain-oriented syntax, a strong type system, verification mechanisms and a code generation toolchain.

4.2 The EMA Data Type System

Type systems are an important error avoidance mechanism of many programming languages. Strong typing is highly recommended by the ISO26262 for the development of automotive software. While most type systems are based on technical types such as integers, floats, and doubles, we are going to show how more abstract type systems can support modeling of cyber-physical systems on a more domain-oriented level. The type system of EMA is based on primitive types, which can be refined or grouped together, enabling the developer to create new types tailored to the application. The primitive types are abstract in the sense that they are not bound to a specific realization or standard such as IEEE754 [25]. Instead, EMA types resemble mathematical sets they aim to represent. EMA supports the following basic types: `N` represents the set of positive integers including 0, i.e. $\mathbb{N}$, `N1` represents the set of positive integers not including 0, i.e. $\mathbb{N} \setminus \{0\}$, `Z` represents the set of signed integers $\mathbb{Z}$, `Q` represents the set of signed rational numbers $\mathbb{Q}$, `C` represents the set of Gaussian rationals $\mathbb{Q}[j] = \{z \in \mathbb{C} : z = a + jb : a, b \in \mathbb{Q}\} \subset \mathbb{C}$, `B` represents the set of Booleans (`true` and `false`). For the sake of convenience the alias `Boolean` can be used interchangeably.

The types `N1`, `N`, `Z`, `Q`, and `C` form a directed compatibility relation, where a type is compatible with another type if the latter can represent all the elements of the former. For instance, `N` is compatible with `Z`, `Q`, and `C`, but not with `N1`, since the latter does not include zero. A variable of type `N` can hence be assigned to variables of types `Z`, `Q`, and `C`, but not to variables of type `N1`. Note that these types represent infinite sets of numbers. Since no technical system can represent arbitrarily large numbers, using primitive EMA types leads to a model that can only be implemented partially by definition. Obviously, this does not hold for Booleans (`B`). The decision how to implement such types is delegated to the compiler and can depend on the application.

Technical systems are generally bounded, e.g. a vehicle has a maximum velocity, a minimum turning radius, etc. To model such bounds explicitly, EMA types can be refined by ranges consisting of a lower and an upper bound. A bounded type is defined as `T(minValue : maxValue)`, where `T` can be any primitive type except `B`. The bounded type covers a subset of the primitive type `T` bounded by `minValue` and `maxValue`. `minValue` and `maxValue` must be of type `T` themselves and their values are included in the bounded type. For instance, the bounded type `N(5:7)` represents the set $\{5, 6, 7\}$. A type can be defined as half-open using the infinity operator `oo` as one of the bounds. For instance, `N(5:oo)` is a type covering all integers in $\{n \in \mathbb{N} | n \geq 5\}$.

Bounded types are not completely implementable if the base type is `Q` or `C`, as a technical system cannot handle arbitrarily high resolutions. To obtain a completely realizable type, a bounded type needs to be refined by a resolution or step size. This parameter is written between the minimum and maximum value of a bounded type, i.e. `T(minValue : resolution : maxValue)`. The refined type only contains values of the form `minValue`+$k\times$`resolution` satisfying `minValue` $\leq$ `minValue`+$k\times$`resolution` $\leq$ `maxValue`, where $k \in \mathbb{N}$. For instance, the type `Q(5:0.5:6.5)` represents the set $\{5.0, 5.5, 6.0, 6.5\}$ Similarly to the lower and the upper bounds, the step size needs to be of the basic type it is restricting.

Different levels of type refinements can be employed in different phases of a systems engineering process such as the specification method for requirements, design, and test (SMArDT) [11, 22] during the development of a cyber-physical system (CPS).

In complex technical systems, data is often multidimensional. For this reason, primitive types of EMA can be organized as one-, two- or multidimensional arrays. The syntax to do so is based on the LaTeX syntax for raising a base to a power. To specify the dimensionality of an array type, we need to append a circumflex followed by a list of comma-separated integer-valued dimension sizes in curly brackets to the primitive type's name: `T^{a,b,...}`. Each argument initializes the size of the respective array dimension. For instance, `Q^{5}` represents the set of all five-dimensional rational vectors $\mathbb{Q}^5$, `Z^{2,3}` represents the set of all integer-valued 2×3 matrices, and so on. We refer to one-dimensional arrays as vectors, to two-dimensional arrays as matrices, to three-dimensional arrays as cubes, and to multidimensional arrays as (n-dimensional) hypercubes. The base type of an array can also be a bounded type. For instance, the type `N(0:255)^{3,w,h}`, is often used to represent images with three channels, a size of `w×h`, and a color depth of 8 bit. In contrast to dynamic types systems as used by MATLAB or Python, dimensions are set at compile-time and cannot be changed at runtime. Variables of the aforementioned matrix type `Z^{2,3}` can only be assigned 2×3 matrices.

In EMA, a data type can be refined by the SI unit of the physical quantity it represents. For instance, `Q(0m:1dm:1km)` is a rational variable representing a length between 0 m and 1 km with a resolution of 1 dm. If the type has no range, only the unit is given in brackets. For instance, `Q(m)` denotes the rangeless rational number type to be interpreted as meters. Two EMA variables are only compatible if they represent the same physical quantity. Conversions are carried out automatically in assignments featuring compatible but different units. This way, the developer does not need to keep track of the physical quantities of the variables used in a program, nor does she have to carry out the conversions of units manually. EMA supports all SI units as well as common prefixes.

4.3 Components, Ports, and Connectors

In EMA components are first-level citizens. A component type is defined using the keyword `component` followed by a name which can later be used to create instances of this component type.[1] For instance, we declare the component type `Main` in L.1 of Fig. 28. Optionally, a component type declaration can include a list of generic parameters in angle brackets and another list of component parameters in round brackets. While generic parameters are allowed to change a component's interface, component parameters can only be used to parameterize a component's implementation. Depending on the use case, a generic parameter can be set to a component type, a data type, or a concrete value.

The syntax for declaring a generic component or data type in a component header definition is just the parameter name, cf. parameter `T` in L.1. If the generic parameter is a concrete value, its name needs to be preceded by its data type, cf. generic parameter `n`, which is of type `N(2:10)` in this example. Component parameters, in contrast to generic parameters, can only be of a data type. The syntax resembles the definition of

[1] The component type system is not to be confused with the data type system introduced in Sect. 4.2.

```
EMA
 1  component Main<T, N(0:10) n> (Q param1, N param2,…) {
 2    ports in T A,
 3          in T B,
 4          out T C;
 5
 6    instance Add<T, n> adder(0);
 7    instance Mult<T, n> multiplier(1);
 8
 9    connect A -> adder.A;
10    connect B -> adder.B;
11    connect adder.C -> multiplier.A;
12    connect B -> multiplier.B;
13    connect multiplier.C -> C;
14  }
```

Fig. 28 A basic example of an EMA architecture. The component `Main` contains two subcomponents `Add adder` and `Mult multiplier`

function parameters in many languages, where a type is followed by a unique name, cf. parameters `Q param1` and `N param2` in L.1.

The body of a component definition is enclosed in curly brackets and contains an interface and a structure definition. The interface definition is initiated with the keyword `ports` and is followed by a port list. A port definition consists of the port kind, which can be either `in` or `out` (EMA ports are strictly unidirectional), a data type, and a unique port name, cf. L.2-4 in Fig. 28. A component must have at least one input and one output port, since a major assumption of EMA is the absolute absence of side effects. Clean side effect-free models are crucial for testability, maintainability, and extensibility. An exception are components outputting a constant or a (possibly parameterizable) constant sequence. Such components obviously do not need an input port, but can require a component parameter, which alone defines the output behavior in every execution step.

Subcomponents are created using the keyword `instance` followed by the component type to instantiate and a component instance name, which is unique in the scope. If the component type to be instantiated has generic and/or component parameters, these have to be set by providing appropriate arguments in angle and/or round brackets, respectively. In L.6-7 of Fig. 28 two components are instantiated with their generic parameters being set to the type `T` and the value `n`. Furthermore, both subcomponents receive a component parameter in round brackets, which is 0 in L.6 and 1 in L.7.

To interconnect the subcomponents and to connect them to the parent component in the first place, we need to create connectors. The source of a connector must be either an output port of a sibling or subcomponent or an input port of the enclosing component. Similarly, the target of a connector must be either an input port of a sibling or subcomponent or an output port of the enclosing component. A connector is created using the `connect` keyword followed by the source port, the arrow operator `->`, and a target port. Ports of subcomponents can be referenced by using the subcomponent's name and the dot access operator. Connector examples are given

EMA

```
1  component Main<N(2:10) n> {
2    ports in Q A[n],
3          in Q B[n],
4          out Q C[n];
5          out Q D;
6
7    instance Add2 adder[n];
8    instance Mult2n<2*n> multiplier;
9
10   connect A[1:n] -> adder[1:n].firstSummand;
11   connect B[:] -> adder[:].secondSummand;
12   connect adder[:].sum -> C[:];
13
14   connect A[:] -> multiplier.factors[1:n];
15   connect B[:] -> multiplier.factors[n+1:2*n];
16   connect multiplier.product -> D;
17 }
```

Fig. 29 An EMA architecture example featuring port and component arrays. The component `Main` contains `n` `Add2` components, each operating on one of `n` operand pairs coming from the port arrays `A` and `B`. The `Mult2n` component computes the product of `2n` operands passed through the port arrays `A` and `B` of the `Main` component to the port array `factors` of `Mult2n`

in L.9-13. Connectors define explicit dataflows. At execution time, data is exchanged only between ports connected by connectors.

Once a component cannot be subdivided into smaller subcomponents, it can be linked to a concrete behavior as will be discussed later. In standard EMA, the structure, i.e. the subcomponents as well as the connectors between them, is fixed at design-time.

Modeling cooperative systems and agent networks often requires the replication of large numbers of similar components and the interconnection thereof. EMA enables the designer to create multiple similar components and/or ports by means of arrays. Based on the array syntax of many languages, an array is created by appending the array size to the port or component name in brackets. For instance, in Fig. 29 we define the input ports `A` and `B` as well as the output port `C` as port arrays of length `n`. Since parameter `n` affects the interface of `Main` by changing the length of the port arrays `A`, `B`, and `C`, it cannot be defined as a component parameter, but must be a generic parameter instead.

In this example we demonstrate two interconnection patterns which are commonly used when dealing with port and component arrays. In the first one, we instantiate an array of components to deal with an array of incoming streams. Therefore, we create `n` adders of the component type `Add2` in L.7, each instance to operate on two scalar inputs. Now, we need to connect the ports of the two arrays `A` and `B` of the parent component to the respective subcomponents, i.e. `A[1]` and `B[1]` should be connected to `adder[1]` and so on. This can be done in just one line, cf. L.10, by selecting the elements `1` to `n` from the port array `A` and, similarly, the components `1` to `n` from the `adder` component array. The connect operator connects each source element to the respective target element based on the index. Since this connection

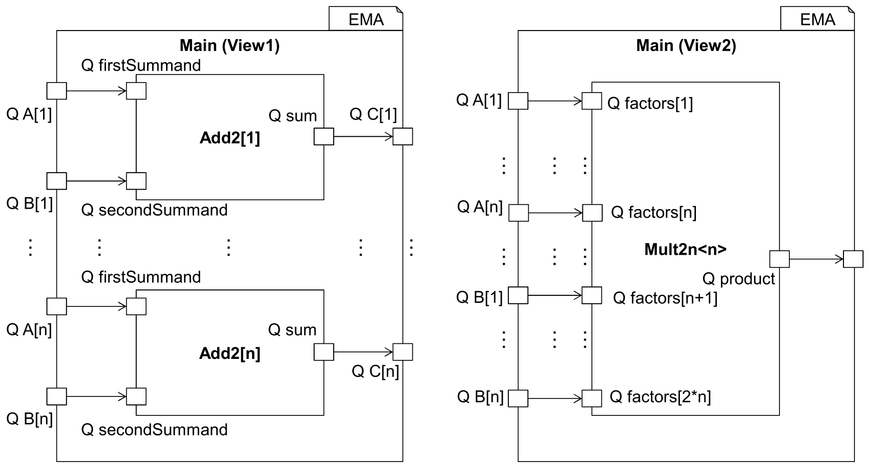


Fig. 30 Graphical views of the component defined in Fig. 29. On the lhs, the elements of two port arrays are connected to target ports of a component array. On the rhs, a port array is connected to another port array

pattern is often applied to *all* elements of an array, EMA offers syntactic sugar allowing the developer to leave out the indices of the first and last elements as is done in L.11. Similarly, in L.12 the output of each component in the `adder` array is connected to a corresponding port in the target port array `C`. This structural pattern is depicted graphically in the view on the left side of Fig. 30.

Furthermore, we can connect a port array to the port array of a target component, let this component aggregate the data and output a single result or a constant number of values. In our example, the port array `A` is connected element-wise to the first `n` elements of the input port array of the `multiplier` component of type `Mult2n` in L.14, while the port array `B` is connected to the remaining `n` input ports of `multiplier` in L.15. The output of the `multiplier` component is forwarded to the output port `D` of the enclosing component in L.16. This connection pattern is depicted graphically in the view on the rhs of Fig. 30.

4.4 Execution Semantics

Standard EMA has a synchronous and weakly causal execution semantics, which is based on the FOCUS theory [3] and inspired by Simulink [40]. In each cycle, every component is executed exactly once. Once a component has finished its execution, the computation results are immediately available at its output ports. We assume that data transmission over connectors is lossless and has no delay. Connectors transmit data instantly, i.e. when a source port of a connector is updated, the data is replicated immediately to the target port. A component is only allowed to be executed, once

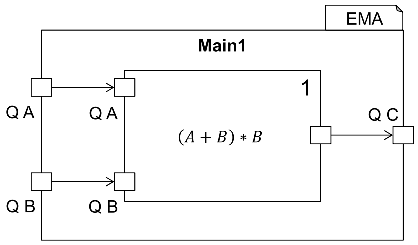


Fig. 31 This example shows two C&C architectures `Main1` and `Main2`, which are semantically equivalent in EMA due to its synchronized and weakly causal execution model, but which might have different interpretations in a language with strongly causal semantics

all of its predecessors, i.e. components connected to its input ports via a connector, have finished execution. Therefore, the identification of a dataflow-based execution order is crucial for a correct realization of the model semantics. A fixed execution order is established at compile-time and no re-scheduling needs to be performed at runtime. This is similar to Simulink's sorted execution order list.[2] In EMA, the C&C model is flattened at compile-time before the execution order is computed. Hence, only atomic components receive an execution order id. In EMA, multiple component instances can share a single execution order id if the execution order of these respective component instances can be exchanged without affecting the computation results. For instance, the adders of the `adder` component array of Fig. 28 can be executed independently.

At runtime all the components are executed sequentially based on the execution order list in each cycle. A cycle is finished when all components have been executed. The next cycle can be started, once the preceding cycle is finished. In EMA the input until time t completely determines the output until time t rendering the semantics weakly causal [3], which is convenient for modeling algorithms and physical processes. As an example consider the two architectures in Fig. 31. Both systems have the same semantics in EMA and can be described mathematically using the equation $C_k = (A_k + B_k)\,B_k$, where k is a sequential index. In contrast, if the system were strongly causal under the assumption that each subcomponent required n timesteps to compute and communicate the output, the equations describing `Main1` and `Main2` would become $C_k = (A_{k-n} + B_{k-n})\,B_{k-n}$ for the left and $C_k = (A_{k-2n} + B_{k-2n})\,B_{k-n}$ for the right model, respectively.

Finding an execution order for linear models, i.e. models without cyclic port dependencies, is straightforward: each component instance is put on the execution list after all component instances its input ports depend on. When structural loops are present in the model, i.e. when there is a path from a subcomponent's output to its own input without a delay, the compiler checks if the loop is algebraic. If yes, the compiler tries to transform the algebraic loop to a loop-free equivalent model.

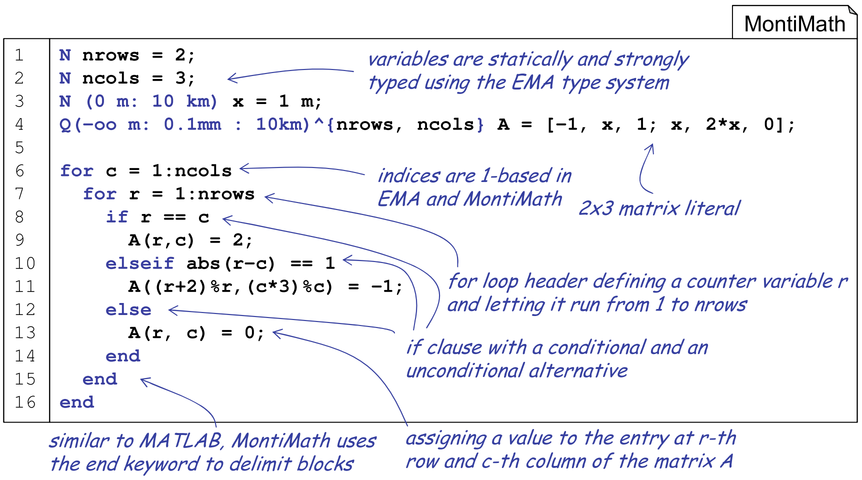
[2] https://de.mathworks.com/help/simulink/ug/controlling-and-displaying-the-sorted-order.html, accessed November 25, 2022.

If an explicit solution cannot be found, i.e. if the loop does not correspond to a known (solvable) pattern, it can be solved at runtime using an algebraic solver. Since this must be done in each timestep and there is no guarantee that a solution exists, a runtime solver would not only affect the runtime performance heavily, but might also lead to unpredictable behavior, which must be avoided in safety-critical systems. For this reason we only allow loops, which can be transformed into loop-free architectures at compile-time. If no such transformation can be found, the model is considered invalid.

To resolve algebraic loops, knowledge of the component behavior is required. A means to integrate behavior models into EMA components will be discussed in Sect. 4.5.

4.5 MontiMath

MontiMath is an imperative language developed for the design and implementation of math-heavy algorithms and to describe physical processes. It has been inspired by MATLAB's matrix-oriented paradigm. However, in contrast to MATLAB, MontiMath uses the EMA type system, which makes it a statically and strictly typed language similar to EMA itself. An example showing the basic language constructs is given in Fig. 32. The declaration of a MontiMath variable requires a type definition, which is expressed by preceding the newly declared variable by an EMA type, e.g. `Q(0 Ohm : 1 nOhm : 1 MOhm)^{2,2} impedance`. The syntax to define a matrix constant is the same as in MATLAB, but the literals inside the



```
N nrows = 2;
N ncols = 3;
N (0 m: 10 km) x = 1 m;
Q(-oo m: 0.1mm : 10km)^{nrows, ncols} A = [-1, x, 1; x, 2*x, 0];

for c = 1:ncols
  for r = 1:nrows
    if r == c
      A(r,c) = 2;
    elseif abs(r-c) == 1
      A((r+2)%r,(c*3)%c) = -1;
    else
      A(r, c) = 0;
    end
  end
end
```

MontiMath
variables are statically and strongly typed using the EMA type system
indices are 1-based in EMA and MontiMath
2x3 matrix literal
for loop header defining a counter variable r and letting it run from 1 to nrows
if clause with a conditional and an unconditional alternative
similar to MATLAB, MontiMath uses the end keyword to delimit blocks
assigning a value to the entry at r-th row and c-th column of the matrix A

Fig. 32 This listing shows a simple MontiMath example exhibiting the main language constructs including variable declarations, matrix literal definitions, loops and conditions

matrix can be enriched by système international d'unités (SI) units if needed. As in MATLAB, a matrix constant is defined in square brackets. Thereby, columns and rows are separated by commas and semicolons, respectively. The initialization of the impedance matrix `impedance` modeling a two-port network can hence be written as `impedance = [10 Ohm, 5 Ohm; 6 Ohm, 8 Ohm];`.

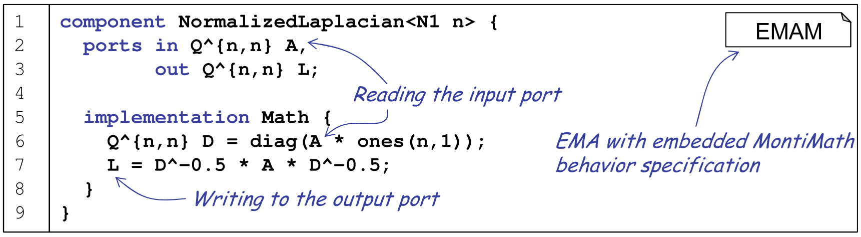
To maintain compatibility to MATLAB, MontiMath indices start with 1 as opposed to most general purpose programming languages (GPLs), where arrays are zero-based. Scalars are treated as 1×1 matrices, but the square brackets can be dropped when defining a scalar literal. Other than in MATLAB, statements, except conditional statements and loops, need to be terminated with a semicolon.

MontiMath supports the typical operators needed in many computations including addition (+), subtraction (-), multiplication (*), division (-), and power (^). If applied to matrices, these operators perform the corresponding algebraic matrix operation, e.g. a matrix multiplication. Division by a matrix, e.g. `A/X`, is semantically equivalent to multiplying the dividend with the inverse of `X`, i.e. `A/X` is equivalent to `A*X^-1` or `A*inv(X)`.

Furthermore, MontiMath supports the Hadamard product or element-wise multiplication (.*), inverse Hadamard product (./), and element-wise power (.^). The transpose operation for real and the Hermitian transpose operation for complex-valued matrices can be expressed by appending the apostrophe operator (`'`) to a matrix name, e.g. `A'`. Furthermore, the entries are conjugated in the complex case. Since matrix dimensions are statically typed, incompatibilities are detected at compile-time.

MontiMath supports the standard control flow constructs including `for` loops and `if` clauses, enabling us to write arbitrarily complex algorithms. Many tasks in CPS engineering can be expressed as optimization problems, e.g. model-predictive controllers. For this reason, we introduce optimization statements in MontiMath. The syntax provides dedicated keywords for optimization problems to come as close as possible to the original mathematical formulation enabling the developer to write down the objective function, to define the optimization variable, as well as a set of constraints.

A MontiMath program can be embedded into an EMA component by means of an implementation block as is shown in Fig. 33. This way the MontiMath script is executed in every execution cycle of the EMA component. It can read the input ports



```
component NormalizedLaplacian<N1 n> {
  ports in Q^{n,n} A,
        out Q^{n,n} L;

  implementation Math {
    Q^{n,n} D = diag(A * ones(n,1));
    L = D^-0.5 * A * D^-0.5;
  }
}
```

EMAM
Reading the input port
EMA with embedded MontiMath behavior specification
Writing to the output port

Fig. 33 An EMAM model embeds a MontiMath script into an EMA component

of the EMA component and write the computation results to the output ports. To let a MontiMath script pass variable values from one execution cycle to another, we introduce the `static` keyword. A variable declared with this modifier, e.g. `static Q cumulativeError`, is saved in a cycle-independent scope. Its value does not get lost when an execution cycle is finished and can be reused in the next cycle. Alternatively, variables can be passed between cycles by feeding the output of a component back to one of its input ports and putting a delay block in between.

The modular structure of the EMA language family enables an easy composition with other modeling languages to be used in the *implementation* block of an EMA component for the definition of the component behavior. The language used can be another domain-specific language (DSL) or a GPL such as C++ or Java. For the composition to work, the embedded language must have a MontiCore implementation [23]. A particularly important DSL for component behavior definition is the deep learning modeling language MontiAnna [27, 35, 36]. It enables a concise modeling of deep neural networks as directed acyclic graphs (DAGs) of neuron layers. The MontiAnna generator produces code for data loading, training, and execution of the neural network. Furthermore, it controls the machine learning lifecycle of the deep learning component, e.g. supporting data management [2] and deciding whether a training phase is needed or can be skipped if a trained model is already available, based on a machine learning artifact model [1]. MontiAnna has been applied to model deep neural networks for various domains, including image processing convolutional neural networks (CNNs) [35], language processing networks [35], reinforcement learning applications [19], generative adversarial networks (GANs), variational autoencoders (VAEs), etc. A CNN for the recognition of handwritten digits embedded into an EMA component is depicted in Fig. 34. The neural network is assembled from predefined layers and the custom layer `conv` in L.13-21. While the example is a linear graph of layers, arbitrary DAGs can be constructed using MontiAnna.

4.6 Cooperative Agents and EmbeddedMontiArc Dynamics

Until now the focus was on static architecture modeling of closed, isolated systems such as autonomous vehicles using EMA. The elements of a static architecture are fixed at design time and cannot be altered, removed, or added at runtime. With this approach we can cover the majority of closed systems such as embedded devices and control software. However, cooperative driving systems which are highly dynamic by nature require the ability to restructure or reconfigure parts of their architecture according to changing circumstances and requirements at runtime. For this reason, we are going to discuss an extension for EMA introducing dynamics to architectural elements such as ports, connectors, and components based on [26].

Different forms of dynamic architecture description languages (ADLs) are known in the literature tackling different concerns of architectural dynamics [5]. In particular, the choice of appropriate means of architectural runtime reconfiguration depends on the kind of system under development and the application domain. The concepts

EMADL

```
component Detector<Z(2:oo) classes = 10>{
  ports in Z(0:255)^{1, 28, 28} data,
  out Q(0:1)^{classes} softmax;

  implementation CNN
  {
    def conv(channels, kernel=1, stride=1){
      Convolution(kernel=(kernel,kernel),channels=channels) ->
      Relu() ->
      Pooling(pool_type="max", kernel=(2,2), stride=(stride,stride))
    }

    data ->
    conv(kernel=5, channels=20, stride=2) ->
    conv(kernel=5, channels=50, stride=2) ->
    FullyConnected(units=500) ->
    Relu() ->
    Dropout() ->
    FullyConnected(units=classes) ->
    Softmax() ->
    softmax;
} }
```

Fig. 34 A CNN for handwritten digit recognition embedded into an EMA component, also referred to as an EMADL component

discussed in this chapter are intended for the Local Traffic System (LTS) domain discussed in the previous sections. Our design decisions will hence be based on the following list of assumptions:

- The agents are instances of compatible types or share a common interface. In the automotive domain, for instance, agents are equal or similar vehicles or roadside units (RSUs). The agents are independent processes with proprietary goals. They are not part of and do not contribute to the functioning of a bigger system (in contrast to an aircraft architecture designed using a language like Architecture Analysis & Design Language (AADL), where architectural dynamics is used to model functional variations of a single but complex system).
- The agents do not know each other by default and there is no communication between them at the beginning. Furthermore, the total number of agents living in the system is not known to an agent. Each agent's knowledge about its peers is limited to what it perceives through its sensors and communication.
- The number of agents in the system can vary throughout time. Agents can be spawned without existing agents to be notified explicitly. In the cooperative vehicles domain, new vehicle instances can come into existence by being manufactured or by entering the area of interest from outside.
- There is a communication channel which can be used by the agents to send and receive messages to and from other agents, respectively. This channel can be used for both directed and broadcast communication. However, since we are dealing with the application layer, we will not care about lower network protocols in this

work, assuming an end-to-end channel connecting the logical interfaces, e.g. EMA ports, of two different agents directly.

To be able to model interactions between participants of a dynamically changing traffic system, the C&C language used needs to support changes in the component structure and variations of the dataflows at runtime. Such changes can be induced by specific events, such as the occurrence of a new traffic participant, which the developer should be able to model with the same language, as well.

The aim of this section is to introduce the main concepts of an EMA language extension for dynamic reconfiguration, which we are going to refer to as Embedded-MontiArc Dynamics (EMAD). The extension is conservative [24], meaning that standard, non-dynamic models can be parsed and generated by EMAD without changes.

4.7 EMAD Execution Semantics

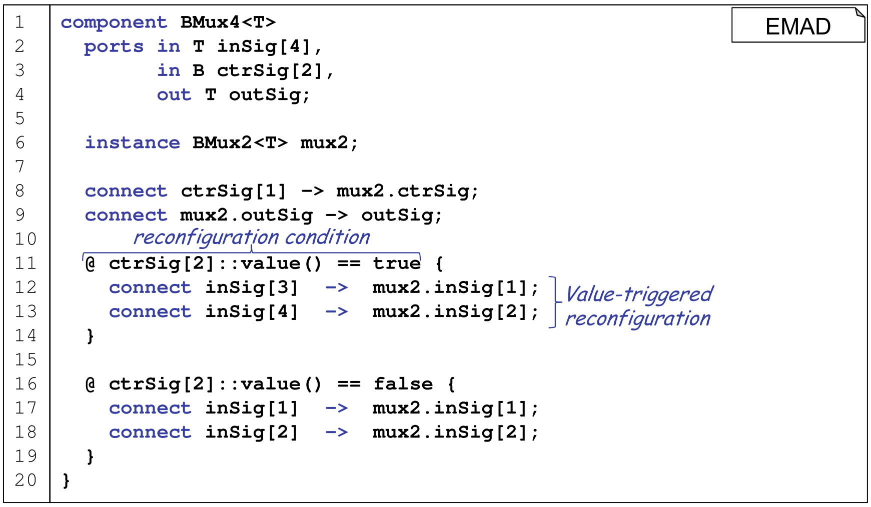
In Sect. 4.4 we have discussed the synchronous execution semantics of EMA. The system is executed stepwise. In each step all the subcomponents are executed according to an execution order determined at compile-time. To enable reconfiguration and to support dynamically evolving architectures, we extend the execution semantics of EMA by a reconfiguration phase which takes place in each execution cycle.

In the reconfiguration phase, reconfiguration triggers are checked and, if present, the corresponding reconfigurations are performed. This possibly activates further reconfiguration triggers which are then handled as well, until the reconfiguration queue is empty. We introduce two main concepts for runtime reconfiguration in EMAD: 1. Data-triggered and 2. Service-based reconfiguration.

4.7.1 Data-Triggered Internal Reconfiguration

The simplest way to trigger and model reconfiguration is the data-triggered approach. Thereby, a reconfiguration is initiated when a signal fulfills a given condition, e.g. a port value exceeds a predefined threshold. The reconfiguration is executed and maintained as long as the condition is satisfied. The approach can be easily motivated and illustrated by non-linear components used in electronics. For instance, a diode is conductive only if the applied voltage is higher than the threshold voltage; a multiplexer passes the data signal chosen by a control signal; when a battery electric vehicle (BEV) is connected to a charging station, the connection is signaled to the charging electronics which reacts by enabling the charging process as long as the connection signal is active.

To enable modeling data-triggered reconfiguration, we extend the body of an EMA component definition by a list of reconfiguration blocks. The header of such a reconfiguration block contains a condition formulated as a Boolean expression over port values and architectural properties, which needs to be fulfilled in order to trigger



```
1   component BMux4<T>
2     ports in T inSig[4],
3           in B ctrSig[2],
4           out T outSig;
5
6     instance BMux2<T> mux2;
7
8     connect ctrSig[1] -> mux2.ctrSig;
9     connect mux2.outSig -> outSig;
10
11    @ ctrSig[2]::value() == true {
12      connect inSig[3]  ->  mux2.inSig[1];
13      connect inSig[4]  ->  mux2.inSig[2];
14    }
15
16    @ ctrSig[2]::value() == false {
17      connect inSig[1]  ->  mux2.inSig[1];
18      connect inSig[2]  ->  mux2.inSig[2];
19    }
20  }
```

EMAD
reconfiguration condition
Value-triggered reconfiguration

Fig. 35 A multiplexer component choosing two of its inputs to be passed to the inner multiplexer dependent on a control signal

the reconfiguration. The body of the reconfiguration block follows for the most part the same syntax as the body of a standard non-dynamic component and contains a declarative definition of the architectural changes to be performed as a response to the triggering event. These changes are rolled back as soon as the reconfiguration condition in the reconfiguration block header ceases to hold.

To illustrate the syntax and the mechanics behind data-triggered reconfiguration, we introduce a simple multiplexer example in Fig. 35. The `BMux4` component has four data inputs of a generic type `T` and two Boolean control inputs. The purpose of the component is to choose one of the four input signals of the `inSig` port array based on the values of the control signals (`ctrSig` port array) and to forward it to the output port. The idea is to realize this behavior by altering the connectors corresponding to the control signal. Therefore, we first choose two of the four data signals (the first two *or* the second two ports of the `inSig` array) based on the value of `inSig[1]` and then forward them as well as a further control signal `inSig[2]` to a subcomponent of type `BMux2`, which in turn uses the received control signal `inSig[2]` to choose one of the remaining two data signals. Its choice is then output through the parent component's output port.

The static connectors of the component are defined in L.8-9 to connect the first control signal with the inner multiplexer and its output to the output of the parent `BMux4`. There are two reconfiguration definitions given in L.11-14 and L.16-19. In L.11 and L.16 the @ symbol denotes the beginning of a reconfiguration condition. The actual reconfiguration code is a block enclosed in curly brackets following the condition. As can be seen in L.12-13 and in L.17-18, the configuration code is composed of ordinary connect statements as we know them from the static EMA syntax. The connections defined in these two blocks are established and released in the reconfiguration phase at the beginning of an execution cycle as discussed earlier. In this example, this is used to choose two of the four incoming inputs to be forwarded to the child component `mux2`.

A reconfiguration is executed once the condition becomes true and remains active as long as the condition remains true, i.e. as long as the value at the port `ctrSig[1]` is true in L.11 and as long as it is false for L.16. When the condition of an active reconfiguration goes back to false, the reconfiguration is rolled back, i.e. all the architectural elements defined in the reconfiguration block are removed (irrespective of whether or not another reconfiguration becomes active instead). In our example, the two reconfiguration conditions are mutually exclusive, but their disjunction is always true. Consequently, exactly one of the two reconfigurations is active at any given point in time. In general, arbitrarily many reconfigurations (including zero) can be active in parallel. However, each combination must result in a valid architecture. That is, an input port must not be the target of more than one connector. Furthermore, under no circumstances an input port may be floating. This is verified by context conditions at compile-time. Consequently, none of the two reconfigurations can be removed from the component in the multiplexer example: when no dynamic reconfiguration is active, only the static part of the architecture is present. In this case, the `inSig` ports of `mux2` would be floating.

Note that in order to access the value of a port in an EMAD reconfiguration, we use the port function `value()` accessible for each port of the component using the `::` operator. The syntax highlights that we are not trying to use a model element in a conventional manner (which would require a dot), but want to perform a runtime query related to a model element instead. The function is available in reconfiguration conditions and bodies only. If the port we are referring to belongs to a subcomponent, we can access it by specifying the port's name preceded by the (subcomponents') instance name, e.g. `mux2.outSig::value()`. Note that a component can only query the values visible in its scope, i.e. values of its own or of its immediate subcomponents', but not of its subsubcomponents' or the parent component's ports.

A reconfiguration condition can be an arbitrary Boolean expression. Similarly to other languages the Boolean OR and the Boolean AND operators are denoted by `||` and `&&`, respectively. For equalities and inequalities we use the following operators: `==`, `<=`, `>=`, `<`, `>`.

Reconfiguration conditions can be formulated in terms of an expression sequence in order to identify sequence patterns. A value sequence can be notated similarly to an EMA row vector with the oldest value coming leftmost. To avoid confusions with vector-valued variables, the `tick` keyword is used as a separator instead of a comma. For instance, the condition `ctrSig[1]::value() == [true tick false tick false]` is evaluated to true at execution cycle n if the following sequence of values was observed: `true` at $n-2$, `false` at $n-1$, `false` at n. The type of each expression in the sequence must be compatible with the corresponding port type. The sequence notation implies that past values of the underlying port need to be stored at runtime. In this particular example, in addition to the current value at

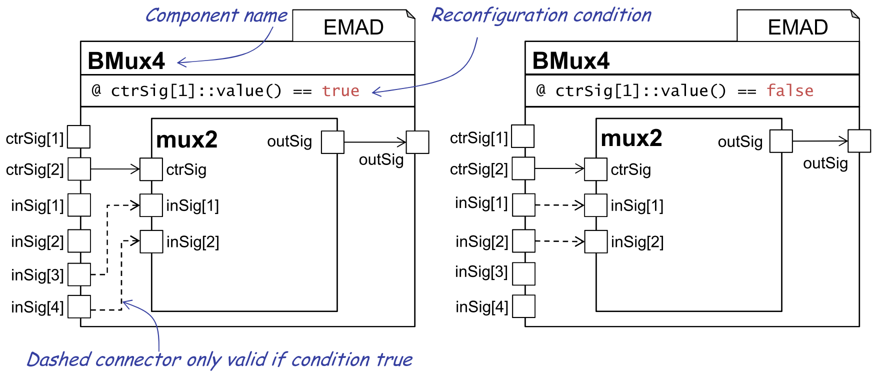


Fig. 36 The two architectural states of the BMux4 component

the `ctrSig[1]` port, the component needs to store two of this port's past values in order to be able to evaluate the reconfiguration condition in each execution step.

Until now, we have been using a graphical representation of EMA models to facilitate the understanding of the architecture. Given the fact that there is no single representation of an EMAD model, we need an appropriate extension of the graphical syntax. Diagrams representing the two reconfigurations of the `BMux4` model are depicted in Fig. 36. Thereby, we introduce two syntactic elements: first, the reconfiguration condition triggering the reconfiguration is specified in a box under the component's name. Second, model elements, which are added in this reconfiguration, are denoted by dashed figures instead of solid ones. In this example, only connectors are created dynamically at runtime. Components and ports can be added in a similar way by the means of dynamic arrays, which will be discussed in Sect. 4.7.2.

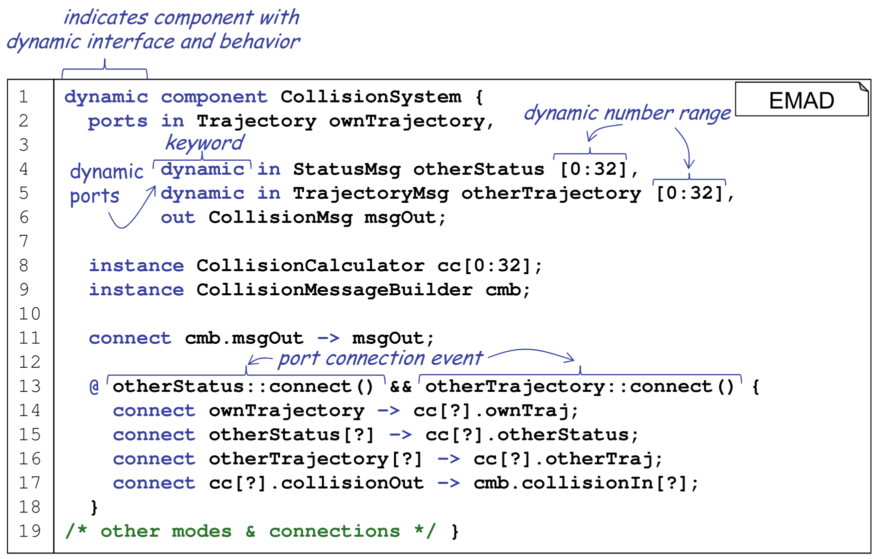
The aim of the example in Figs. 35 and 36 was to introduce the main ideas behind data-triggered reconfiguration. The exactly same behavior can be achieved with a mode model with two states [21]. Using a mode finite state machine (FSM) for a system with a small number of states and state transitions can be favorable as it facilitates a state-centric model analysis. In cases with many, possibly partially overlapping reconfiguration conditions and state transitions between all possible states, however, the data-triggered reconfiguration concept presented in this chapter can lead to much more concise models, since we don't need to define all possible states explicitly and no transitions need to be modeled at all. On the other hand, modes are more powerful since reconfigurations can depend on the current *architectural* state, which is not possible with our concept. We recommend using modes and data-triggered reconfiguration interchangeably depending on the requirements and the nature of the modeled system.

4.7.2 Service-Based External Reconfiguration

To enable the creation of more complex, propagating reconfigurations, we introduce a second way of triggering architectural changes at runtime, the service-based reconfiguration. The idea behind it is to trigger reconfigurations by external architectural change requests and to propagate such requests from component to component.

We are going to present the concepts of service-based reconfiguration by the example of a cooperative collision prediction component given in Fig. 37. The `CollisionSystem` component receives the planned trajectories from other vehicles of an LTS and checks each of these trajectories for a collision with its own one. Each trajectory is input into the component through a dedicated port. Furthermore, each pairwise collision check is executed by a dedicated subcomponent of type `CollisionCalculator`.

Before we proceed with the discussion of the service-based trigger mechanism, we need to introduce the concept of dynamic component and port arrays. In Sect. 4.3, static component and port arrays were introduced, allowing us to model an arbitrary but fixed number of similar components and ports in a single line of code. In the collision detection example described here we don't know at design time, how many traffic participants will be present in the LTS. Furthermore, the number of peers can change over time. The concept of dynamic arrays enables us to cope with this modeling challenge by allowing us to specify a range instead of a fixed number of elements in the array. At runtime the concrete number of elements in the array can change.



```
dynamic component CollisionSystem {
  ports in Trajectory ownTrajectory,
        dynamic in StatusMsg otherStatus [0:32],
        dynamic in TrajectoryMsg otherTrajectory [0:32],
        out CollisionMsg msgOut;

  instance CollisionCalculator cc[0:32];
  instance CollisionMessageBuilder cmb;

  connect cmb.msgOut -> msgOut;

  @ otherStatus::connect() && otherTrajectory::connect() {
    connect ownTrajectory -> cc[?].ownTraj;
    connect otherStatus[?] -> cc[?].otherStatus;
    connect otherTrajectory[?] -> cc[?].otherTraj;
    connect cc[?].collisionOut -> cmb.collisionIn[?];
  }
/* other modes & connections */ }
```

Fig. 37 Collision system of an autopilot calculating potential collisions with up to 32 other vehicles

The syntax is based on the range syntax of EMA types: the modeler needs to specify the minimum and the maximum number of elements inside the square brackets of an array declaration separated by a colon instead of a single length value. This is done in L.4 and L.5 of Fig. 37 to define a dynamic port array and in L.8 to define a dynamic component array. In the case of port arrays it is obligatory to use the `dynamic` keyword. If the component interface contains dynamic port arrays, it is also necessary to mark the component with the `dynamic` keyword in the header, cf. L.1 of Fig. 37.

In case the lower bound of the element count is greater than zero, the minimum number of elements will be created at instantiation of the component. Once the upper bound of the elements in an array has been reached, events leading to an instantiation of further elements cannot be handled. The availability of free port and/or component slots in an array can hence be regarded as a further implicit condition of a reconfiguration. Upper bounds on elements in an array have been introduced with embedded systems in mind often having very limited resources and strict performance requirements. The upper bound can be set to infinity by putting `oo`, similarly to EMA type bounds. However, since this can have a negative impact on the performance of an overloaded system, this is not an advisable modeling pattern and results in a warning. A system knowing its limits can react to an overly high demand in a controlled manner.

In our collision system example, the port arrays `otherStatus` and `otherTrajectory` are supposed to receive status and trajectory messages from other cooperative vehicles in the LTS. The maximum number of connections is limited to 32. On the other hand, if there are no other vehicles in the network, the port arrays can be empty.

For each connected vehicle, the `CollisionSystem` component provides an individual `CollisionCalculator` component instance. Accordingly, the number of these instances varies between 0 and 32, as well. At system start up, the minimum number of components and ports is instantiated, i.e. zero.

The question arises how the free slots in the component and port arrays can be used and released at runtime. We realize this by introducing a *reconfiguration service interface*. This interface allows external components or even external software to request reconfigurations. More precisely, it allows external clients to request a port from a dynamic array.

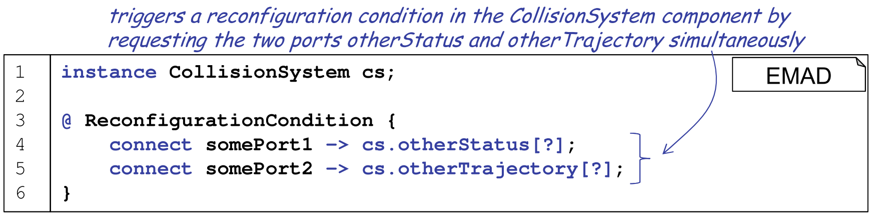
The reconfiguration interface is defined not just by declaring a dynamic port array, but by the reconfiguration conditions using it, cf. L.13 in Fig. 37. To query reconfiguration requests in a reconfiguration condition, we introduce the new port property `connect`, which is basically a Boolean flag indicating whether a connect request for this port has been issued, bundled with an id to avoid confusions with other requests sent to the same port. Similarly to the value at a port, the `connect` property can be queried using the `::` operator, i.e. as `portName::connect()`. A reconfiguration condition can be composed as a conjunction of arbitrarily many connect atoms, i.e. `portName1::connect() &&,...,&& portNameN::connect()`, where the port names used must be dynamic port arrays declared in the component's interface. Disjunctions and negations of connect atoms are forbidden by a context condition to prevent inconsistencies (in a disjunction we do not know at design-time which port(s) will be actually requested and hence, cannot define meaningful reconfigurations using these ports).

The resulting reconfiguration interface can be used by issuing connect request for all the ports required by the reconfiguration condition simultaneously. In our example this means that, due to the reconfiguration condition in L.13 of Fig. 37, connections to the `otherStatus` *and* the `otherTrajectory` port must be requested at once. Such a request is created in an EMAD model in the reconfiguration body of a parent component as connect statements targeting the corresponding dynamic port arrays. This is shown in Fig. 38, where a component holding an instance of `CollisionSystem` connects to the aforementioned port arrays `otherStatus` and `otherTrajectory` of the latter in L.4-5 of its own reconfiguration body.

Note that the reconfiguration bodies of Figs. 37 and 38 are chained: the reconfiguration of the latter triggers the one of the former. If `ReconfigurationCondition` in L.3 of Fig. 38 is a data-driven reconfiguration as discussed in Sect. 4.7.1, the chain starts in Fig. 38. If `ReconfigurationCondition` defines a reconfiguration interface similar to L.13 in Fig. 37, it must be triggered from another reconfiguration body itself. Hence, arbitrarily long service-based reconfiguration chains can be initiated by a data-driven reconfiguration.

Note that the reconfiguration request issued by the parent component of the `CollisionSystem` component in L.4-5 of Fig. 38 matches the reconfiguration interface defined in L.13 of Fig. 37 exactly. This is verified at compile-time by a context condition. An invalid usage of the reconfiguration interface of the `CollisionService` component is shown in Fig. 39. Here we are trying to connect to the `otherStatus` port only. However, this is not supported and results in a compile-time error as there is no such reconfiguration condition in the `CollisionSystem` component.

To be able to deal with dynamic port and component arrays in reconfiguration descriptions, we need a syntax allowing us to access the newly created elements. To do so, we introduce the `?`-operator. It is used instead of the element number in square brackets to request and access new elements in a dynamic port or component array, e.g. `myArray[?]`. Usage of the operator is restricted to reconfiguration bodies.



triggers a reconfiguration condition in the CollisionSystem component by requesting the two ports otherStatus and otherTrajectory simultaneously

```
1  instance CollisionSystem cs;
2
3  @ ReconfigurationCondition {
4      connect somePort1 -> cs.otherStatus[?];
5      connect somePort2 -> cs.otherTrajectory[?];
6  }
```

Fig. 38 The listing shows a valid usage of the reconfiguration service interface of the Collision System component of Fig. 37 by a parent component

invalid port request results in a compile-time error: the CollisionSystem component requires otherStatus and otherTrajectory to be requested together

```
instance CollisionSystem cs;                                   EMAD

@ ReconfigurationCondition {
    connect somePort1 -> cs.otherStatus[?];
}
```

Fig. 39 The listing leads to a compile-time error since CollisionSystem does not have a reconfiguration triggered by requesting only the otherStatus port

An example is given in L.14-17 of the `CollisionSystem` model in Fig. 37. In L.14 the `?`-operator is used to connect the `ownTrajectory` port to a new component `cc[?]`. Since this is the first access to `cc[?]` in this reconfiguration body, it implicitly triggers the creation of a new component instance. In contrast, further accesses to `cc[?]` in L.15-17 are pure access operations, no implicit instantiation is involved. If the component type of the component array requires component parameters, the parameter list can be passed in parenthesis right after the array brackets and before the dot operator, e.g. `cc[?](param1, param2,...).ownTraj.`

Since the `cc` array has a maximum capacity which cannot be exceeded, a further implicit reconfiguration condition is that the maximum capacity of this array has not yet been reached. If, however, the array is maxed out, the reconfiguration condition will evaluate to false and the reconfiguration will thus not be activated.

The reconfiguration service interface is available not only at modeling level allowing other components to use it, but also in the generated code. The latter can be used by any client. For instance, C++ code can be generated for the `CollisionSystem` component. Then it can be compiled to a library to be deployed as a building block of the vehicle run-time environment (RTE). The RTE receives a stream of vehicle to vehicle (V2V) messages and redirects them to the right ports of the `CollisionSystem` library (each sender is assigned to one port). If a new LTS participant starts sending, the RTE can request a new port from the `CollisionSystem` library by calling a generated request function. The library in turn checks whether the request is satisfiable. If yes, it provides a new port instance the RTE can forward messages of the new vehicle to. Otherwise no reconfiguration is carried out and the library call returns with an error. The client can then withdraw the request or wait until the dynamic component satisfies the request in a future reconfiguration cycle.

To facilitate the usage of the generated reconfiguration interface, we generate request methods allowing the client to require all necessary ports to activate a reconfiguration with a single function call, e.g. `requestOtherStatusAndOtherTrajectory(Port<T1> *otherStatus, Port<T2> *otherTrajectory)`, where `Port <T>` is a generic class representing an EMA port of type `T` at C++ level. This way, it is not possible to create invalid request, e.g. requiring only an `otherStatus`, but no `otherTrajectory` port, when using the generated code as a library.

EMAD

```
1  dynamic component DynamicSum {
2    port dynamic in Q summands[0:32],
3                 out Q sum;
4
5  implementation Math {
6    Q tmp = 0;
7    for i = 1:size(summands)
8        tmp = tmp + summands(i);
9    end
10   sum = tmp;
11 } }
```

iterates over all ports in the input port array summands

Fig. 40 Adder with 0 to 32 inputs

Figure 40 shows an example combining a dynamic interface with a MontiMath implementation. The purpose of the component is to compute a sum of all inputs and to output the result. This is a typical data aggregation example working on a varying number of inputs. The dynamic input port array `summands` can contain 0 to 32 elements, i.e. at instantiation the component has no inputs and outputs zero due to the initial assignment `tmp = 0` in L.6. The loop in L.7-9 iterates over all ports in the `summands` array and adds each port's value to the overall sum, which is accumulated in the `tmp` variable. In this example, we treat the dynamic port array in a stateless anonymous way. We iterate over the port array and are only interested in the value present at each available port without caring about its history. This is the natural way to deal with dynamic port arrays in MontiMath. Tracking states related to dynamic ports using MontiMath is possible but should be avoided. Instead, to track a concrete dynamic port's history, we need to replicate a dynamic subcomponent for each dynamic port instance, as was done in Fig. 37. This way, each communication partner requiring a port in a dynamic port array is assigned a dedicated processing subcomponent maintaining the corresponding state. Each of these dedicated processing subcomponents only sees a single input port of the dynamic port array it is assigned to instead of the whole port array. This pattern enforces the separation of concerns and high cohesion principle as the processing related to each communication partner is clearly encapsulated and limited to the actual logic (no explicit iterating over the port array is needed in the behavior implementation).

Based on the reconfiguration mechanism described in this section, we can model whole *reconfiguration chains* to realize deep or flat reconfigurations. A deep reconfiguration means that reconfiguration of a parent component triggers reconfigurations in child components. A `connect` to a subcomponent's port activates this port's connect flag which can in turn be used to trigger a reconfiguration in the subcomponent. In the same way, the subcomponent can trigger reconfigurations in its subcomponents and so on. When a parent component instantiates a static subcomponent in an EMAD model, it can connect its output ports immediately, e.g. as is done in L.11 of Fig. 37. However, the subcomponent might be dynamic and new output ports might be added throughout the subcomponent's reconfiguration procedures. In this case, the parent component can react to newly created ports of the subcomponent by observing the dynamic ports' `connect` flags in the same way as it would observe connect request to its own input ports. This enables us to create reconfiguration chains propagating downwards into the hierarchy as well as those coming from the bottom and propagating upwards.

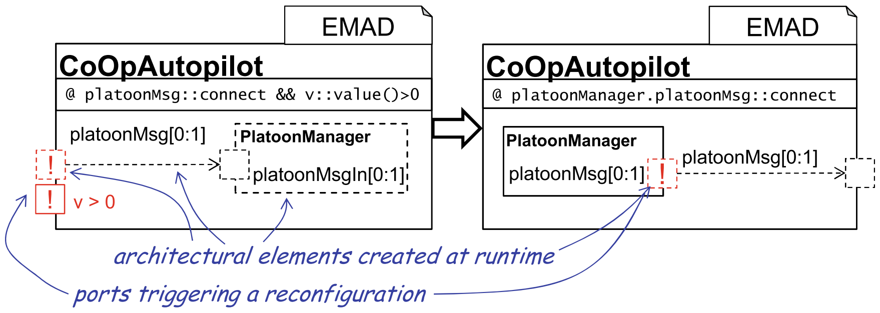
A reconfiguration chain is always performed in one single reconfiguration phase as an atomic transaction, i.e. if the chain breaks at some point, the whole reconfiguration is considered infeasible. If a failure occurs after some reconfiguration steps of the chain have already been carried out, these steps will be rolled back.

As in data-triggered reconfiguration, a reconfiguration remains active as long as the respective condition is fulfilled. Whenever a new port request is issued, the port is created and a connector connected to it, the `port::connect()` property is activated for this port. This flag and hence, the configuration remain active until the requesting client removes its connector to the dynamic port. If the client created the connector as part of an EMAD reconfiguration, it would remove it, when the condition of this original reconfiguration ceased to hold. If the client is an external software, it can use the reconfiguration service interface to roll back a reconfiguration available in the generated code. Such a rollback would remove all architectural elements created in the reconfiguration and trigger the rollback of reconfigurations of subcomponents. This way, a reconfiguration chain is rolled back completely. The rollback interface is not usable explicitly in an EMAD model to prevent arbitrary removals of ports leading to inconsistencies in an architecture.

The service-based reconfiguration procedure of EMAD models boils down to the following steps:

1. Request: an external component sends a set of connect requests.
2. Reservation: the receiving component checks if the requested ports are available, i.e. if the corresponding dynamic port arrays do not violate their respective upper limit constraint. If yes, the component returns references for the new ports, i.e. the newly allocated array indices, to the requester so that explicit access is possible in the future. Otherwise, the requester is informed that its request has been rejected.
3. Reconfiguration: in the reconfiguration phase of the component, the reconfiguration bodies of all valid reconfiguration requests, i.e. those fulfilling a reconfiguration condition, are realized (L.14-17 in the `CollisionSystem` example). Consequently, the component reacts to the external reconfiguration request by internal self-modifications.
4. Follow-up requests: possibly, the reconfiguration instructions of the previous step contain the creation of new ports and/or subcomponents, as well. In this case, the component becomes a requester itself initiating a follow-up reconfiguration in its subcomponents or external components.

In our target domain of interconnected vehicles we mostly need the combination of both data-driven and service-based reconfiguration, which, when used together, can result in a powerful symbiosis. Reconfigurations which emerge as reactions to environmental changes measured by sensors or to incoming messages can be modeled using the following pattern: a data-driven event stands at the beginning of an



EMAD
CoOpAutopilot
@ platoonMsg::connect && v::value()>0
platoonMsg[0:1]
PlatoonManager
platoonMsgIn[0:1]
!
!
v > 0
EMAD
CoOpAutopilot
@ platoonManager.platoonMsg::connect
PlatoonManager
platoonMsg[0:1]
!
platoonMsg[0:1]
architectural elements created at runtime
ports triggering a reconfiguration

Fig. 41 A reconfiguration chain involving input and output ports of the `PlatoonManager` component. An arriving platoon message causes the creation of new input ports in the diagram on the left. Follow-up reconfigurations inside the `PlatoonManager` result in a new output port and a new outgoing connector as depicted in the diagram on the right

event chain. The reconfiguration caused by this event requests new components and ports triggering service-based reconfigurations, which in turn trigger further service-based reconfigurations. As soon as the original trigger vanishes, the reconfiguration chain is rolled back completely and the architecture returns to its initial state. A data-driven source event can be based on a sensor measurement (including the vehicle's antenna receiving messages from other cooperating traffic participants). A particular measurement value or the reception of a specific message would trigger a reconfiguration of the controller architecture, the internal reconfigurations of which are mostly service-based.

An important aspect of EMAD is that there is no explicit way to *remove* architectural elements. Instead, elements are removed implicitly, whenever the triggering reconfiguration condition switches back to false. This guarantees that an architecture can always be put back into its original state.

A further important property is that all possible reconfigurations are fixed by the design time model. Component and port replication is limited by an upper dynamic array size. Consequently, there is only a finite number of possible architectural states at runtime. This is an important design decision preventing a system to reach unexpected states and behaviors and facilitating verification.

Often reconfigurations trigger each other resulting in reconfiguration chains. We can visualize such chains using reconfiguration views, each view only showing the part of the model which is being changed in the current reconfiguration step. One such reconfiguration chain is depicted using views in Fig. 41. In the first reconfiguration view, depicted on the left, the `CoOpAutopilot` component, a controller of a cooperative vehicle, instantiates a platoon manager when a platoon port is requested and the velocity is greater than 0. In a second reconfiguration step, an inner component of the platoon manager requests a new output port and the `CoOpAutopilot` component reacts by creating a new connector. The ports triggering the reconfigurations are emphasized with an exclamation mark. Additionally, the data condition (`v>0`) is set next to the corresponding `v` port. Note that the `PlatoonManager` component is depicted using a dashed line in the left view, while it is solid in the view on the rhs. This is because the component is already there, when the second reconfiguration event is triggered. A big arrow between the two views stresses the order of the reconfigurations. Obviously, a reconfiguration must have taken place inside the `PlatoonManager` component to request the creation of its new output port `PlatoonManager.platoonMsg`. This reconfiguration (chain) is not part of the depicted sequence as it is not in the scope of the `CoOpAutopilot` component and should be visualized in a separate view chain.

4.8 Conclusion

In this chapter we discussed EMA, an architecture description language based on the component-and-connector paradigm. The language facilitates the component-based design of technical systems such as cooperative vehicles thereby enforcing a compliance with functional safety standards. While core EMA only allows the description of static architectures, its conservative extension EMAD enables the developer to model architectural changes such as the creation, removal, and (re)connection of components which are performed at runtime. Due to the conservative extension property, each valid EMA model is also a valid EMAD model [24].

EMAD introduces an event-based reconfiguration system which can react to data-driven as well as architectural events. An EMAD component can instantiate ports, subcomponents, and connectors at runtime as a reaction to a trigger event. Thereby, it can trigger further events of its subcomponents, enabling the modeler to define complex reconfiguration chains.

In EMAD, all possible configuration states are implicitly defined at design time, maintaining the possibility to analyze, predict, and verify the behavior of dynamic components at design and compile-time. A set of context conditions ensures that reconfigurations never clash, making the language applicable to safety-critical systems.

In particular, EMAD can be used to model cooperative systems and their dynamically changing communication channels and processing chains, e.g. in the context of local traffic systems.

To embed behavior into EMA and EMAD components, two behavior description languages are presented: first, MontiMath is a strongly typed matrix-based scripting language offering common constructs such as loops and conditions; second, the MontiAnna language can be used to describe deep neural networks as DAGs of neuron layers, enabling the integration of AI components into larger software architectures.

5 Conclusion

This work demonstrates basic concepts for cooperation and interaction of autonomous vehicles. Basic approaches to the architecture and formation of local traffic systems are shown which are subsequently verified in a real-time verification method for cooperative vehicles. The verified trajectories are collision free and deadlock free. The presented modeling language allows a formal description of vehicle software architectures as well as cooperation and interaction of distributed systems.

Although the presented concepts represent and implement the feasibility and basic approaches, there is a need for further research. The focus of future work should be on further investigation of the reciprocal influence of local traffic systems as well as the cooperation algorithms used and the resulting requirements for necessary modeling languages for distributed systems. Further research is needed in the standardization of necessary V2X messages as well as algorithms used. In the area of V2X messages, it is still not clear whether WLAN-based standards such as ITS-G5 or cellular network-based standards such as C-V2X will prevail. While WLAN-based standards are already used by some manufacturers, C-V2X offers significantly greater potential. With regard to the algorithms used for grouping and actual cooperation, there is a need for more research when considering possible failure cases such as a spontaneous communication interruption with regard to functional safety and achieving required safety standards like ISO 26262 ASIL D [9].

Acknowledgements This research is supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) within the Priority Program SPP 1835 “Cooperative Interacting Automobiles”.

References

1. Atouani, A., Kirchhof, J.C., Kusmenko, E., Rumpe, B.: Artifact and reference models for generative machine learning frameworks and build systems. In: Tilevich, E., De Roover, C., (eds.) Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 21), pp. 55–68. ACM SIGPLAN (2021). http://www.se-rwth.de/publications/Artifact-and-Reference-Models-for-Generative-Machine-Learning-Frameworks-and-Build-Systems.pdf
2. Baumann, N., Kusmenko, E., Ritz, J., Rumpe, B., Weber, M.B.: Dynamic data management for continuous retraining. In: Burgueño, L., Bork, D., Nguyen, P., Zschaler, S. (eds.) Proceedings of MODELS 2022. Workshop MDE Intelligence (2022)
3. Broy, M., Stølen, K.: Specification and Development of Interactive Systems: Focus On Streams, Interfaces, and Refinement. Springer Science & Business Media (2012)
4. Burger, C., Lauer, M.: Cooperative multiple vehicle trajectory planning using miqp. In: 2018 21st International Conference on Intelligent Transportation Systems (ITSC), pp. 602–607. IEEE (2018)
5. Butting, A., Heim, R., Kautz, O., Ringert, J.O., Rumpe, B., Wortmann, A.: A classification of dynamic reconfiguration in component and connector architecture description languages. In: Proceedings of MODELS 2017. Workshop ModComp, CEUR 2019

(2017). http://www.se-rwth.de/publications/A-Classification-of-Dynamic-Reconfiguration-in-Component-and-Connector-Architecture-Description-Languages.pdf
6. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: International Conference on Computer Aided Verification, pp. 334–342. Springer (2014)
7. Dankert, J., Dernehl, C., Eckstein, L., Kowalewski, S., Kusmenko, E., Rumpe, B.: Rapidcoop-robuste architektur durch geeignete paradigmen für kooperativ interagierende automobile. Automatisiertes und Vernetztes Fahren (AAET'17) **7**, 1–6 (2017)
8. Dankert, J., Kowalewski, S., Eckstein, L.: Architekturen und algorithmen für kooperative automobile. Technical report, Lehrstuhl und Institut für Kraftfahrzeuge (ika) (2021)
9. Debouk, R., et al.: Overview of the 2nd Edition of iso 26262: Functional Safety-Road Vehicles. General Motors Company, Warren, MI, USA (2018)
10. Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., Koltun, V.: Carla: An open urban driving simulator. In: Conference on Robot Learning, pp. 1–16. PMLR (2017)
11. Drave, I., Greifenberg, T., Hillemacher, S., Kriebel, S., Kusmenko, E., Markthaler, M., Orth, P., Salman, K.S., Richenhagen, J., Rumpe, B., Schulze, C., Wenckstern, M., Wortmann, A.: SMArDT modeling for automotive software testing. Softw.: Pract. Exper. **49**(2), 301–328 (2019)
12. ETSI, T.: 102 637-1 v1.1.1 intelligent transport systems (its) vehicle communications basic set of applications part 1: Functional requirements. Intelligent transport systems (ITS) (2010)
13. ETSI, T.: 102 637-2 v1.3.1 intelligent transport systems (its) vehicle communications basic set of applications part 2: Awareness basic service. Intelligent transport systems (ITS) (2014)
14. ETSI, T.: 102 637-3 v1.2.1 intelligent transport systems (its) vehicle communications basic set of applications part 3: Specifications of decentralized environmental notification basic service. Intelligent transport systems (ITS) (2014)
15. ETSI, T.: Etsi tr 103 578 "intelligent transport systems (its); vehicular communication; informative report for the maneuver coordination service. Intelligent transport systems (ITS) (2018)
16. ETSI, T.: Etsi ts 103 561 vehicular communications basic set of applications maneuver coordination service (2018). Draft
17. ETSI, T.: 103 562 v2.1.1 intelligent transport systems (its) vehicle communications basic set of applications analysis of the collective perception service (cps). Intelligent transport systems (ITS) (2019)
18. ETSI, T.: Etsi tr 103 324 intelligent transport systems (its) cooperative perception services (2022). Draft
19. Gatto, N., Kusmenko, E., Rumpe, B.: Modeling deep reinforcement learning based architectures for cyber-physical systems. In: Proceedings of MODELS 2019. Workshop MDE Intelligence, pp. 196–202 (2019). http://www.se-rwth.de/publications/Modeling-Deep-Reinforcement-Learning-based-Architectures-for-Cyber-Physical-Systems.pdf
20. Hegde, A., Festag, A.: Artery-c: An omnet++ based discrete event simulation framework for cellular v2x. In: Proceedings of the 23rd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems, pp. 47–51 (2020)
21. Heim, R., Kautz, O., Ringert, J.O., Rumpe, B., Wortmann, A.: Retrofitting controlled dynamic reconfiguration into the architecture description language MontiArcAutomaton. In: Software Architecture - 10th European Conference (ECSA'16). LNCS, vol. 9839, pp. 175–182. Springer (2016). http://www.se-rwth.de/publications/Retrofitting-Controlled-Dynamic-Reconfiguration-into-the-Architecture-Description-Language-MontiArcAutomaton.pdf
22. Hillemacher, S., Kriebel, S., Kusmenko, E., Lorang, M., Rumpe, B., Sema, A., Strobl, G., von Wenckstern, M.: Model-based development of self-adaptive autonomous vehicles using the SMARDT methodology. In: Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'18), pp. 163 – 178. SciTePress (2018)
23. Hölldobler, K., Kautz, O., Rumpe, B.: MontiCore Language Workbench and Library Handbook: Edition 2021. Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag (2021). http://www.monticore.de/handbook.pdf

24. Hölldobler, K., Rumpe, B.: MontiCore 5 Language Workbench Edition 2017. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag (2017). http://www.se-rwth.de/phdtheses/MontiCore-5-Language-Workbench-Edition-2017.pdf
25. IEEE: IEEE-754, Standard for Floating-Point Arithmetic. IEEE Std 754-2008 pp. 1–58 (2008)
26. Kaminski, N., Kusmenko, E., Rumpe, B.: Modeling dynamic architectures of self-adaptive cooperative systems. J. Object Technol. **18**(2), 1–20 (2019). https://doi.org/10.5381/jot.2019.18.2.a2. http://www.se-rwth.de/publications/Modeling-Dynamic-Architectures-of-Self-Adaptive-Cooperative-Systems.pdf. The 15th European Conference on Modelling Foundations and Applications
27. Kirchhof, J.C., Kusmenko, E., Ritz, J., Rumpe, B., Moin, A., Badii, A., Günnemann, S., Challenger, M.: MDE for machine learning-enabled software systems: a case study and comparison of MontiAnna & ML-Quadrat. In: Burgueño, L., Bork, D., Nguyen, P., Zschaler, S. (eds.) Proceedings of MODELS 2022. Workshop MDE Intelligence (2022)
28. Kloock, M., Alrifaee, B.: Coordinated cooperative distributed decision-making using synchronization of local plans (2021). https://doi.org/10.36227/techrxiv.16622017.v2. Submitted to IEEE Transactions on Intelligent Vehicles (T-IV)
29. Kloock, M., Dirksen, M., Kowalewski, S., Alrifaee, B.: Generation of coupling topologies for multi-agent systems using non-cooperative games. In: 2022 IEEE Intelligent Vehicles Symposium (IV). IEEE (2022)
30. Kloock, M., He, Q., Kowalewski, S., Alrifaee, B.: Trajectory verification for networked and autonomous vehicles using temporal logic and model checking. In: 2021 IEEE International Intelligent Transportation Systems Conference (ITSC), pp. 244–250. IEEE (2021)
31. Kloock, M., Kragl, L., Maczijewski, J., Alrifaee, B., Kowalewski, S.: Distributed model predictive pose control of multiple nonholonomic vehicles. In: 2019 IEEE Intelligent Vehicles Symposium (IV), pp. 1620–1625. IEEE (2019)
32. Kloock, M., Muehleisen, M., Calvo, J.A.L., Mathar, R.: Adaptive modulation and coding for reliable vehicular real-time communication. In: Mobile Communication-Technologies and Applications; 25th ITG-Symposium, pp. 1–9. VDE (2021)
33. Kloock, M., Scheffe, P., Botz, L., Maczijewski, J., Alrifaee, B., Kowalewski, S.: Networked model predictive vehicle race control. In: 2019 IEEE Intelligent Transportation Systems Conference (ITSC), pp. 1552–1557. IEEE (2019)
34. Kloock, M., Scheffe, P., Marquardt, S., Maczijewski, J., Alrifaee, B., Kowalewski, S.: Distributed model predictive intersection control of multiple vehicles. In: 2019 IEEE Intelligent Transportation Systems Conference (ITSC), pp. 1735–1740. IEEE (2019)
35. Kusmenko, E., Nickels, S., Pavlitskaya, S., Rumpe, B., Timmermanns, T.: Modeling and training of neural processing systems. In: Conference on Model Driven Engineering Languages and Systems (MODELS'19), pp. 283–293. IEEE (2019). http://www.se-rwth.de/publications/Modeling-and-Training-of-Neural-Processing-Systems.pdf
36. Kusmenko, E., Pavlitskaya, S., Rumpe, B., Stüber, S.: On the engineering of AI-driven systems. In: ASE'19. Software Engineering Intelligence Workshop (SEI'19), pp. 126–133. IEEE (2019). http://www.se-rwth.de/publications/On-the-Engineering-of-AI-Powered-Systems.pdf
37. Kusmenko, E., Roth, A., Rumpe, B., von Wenckstern, M.: Modeling architectures of cyber-physical systems. In: European Conference on Modelling Foundations and Applications (ECMFA'17), LNCS 10376, pp. 34–50. Springer (2017). http://www.se-rwth.de/publications/Modeling-Architectures-of-Cyber-Physical-Systems.pdf
38. Kusmenko, E., Rumpe, B., Schneiders, S., von Wenckstern, M.: Highly-optimizing and multi-target compiler for embedded system models: C++ compiler toolchain for the component and connector language EmbeddedMontiArc. In: Conference on Model Driven Engineering Languages and Systems (MODELS'18), pp. 447 – 457. ACM (2018). http://www.se-rwth.de/publications/Highly-Optimizing-and-Multi-Target-Compiler-for-Embedded-System-Models.pdf
39. Malikopoulos, A.A., Beaver, L., Chremos, I.V.: Optimal time trajectory and coordination for connected and automated vehicles. Automatica **125**, 109469 (2021)

40. Mathworks Inc.: Simulink User's Guide. Technical Report. R2016b, MATLAB & SIMULINK (2016)
41. de Paula Veronese, L., Auat-Cheein, F., Mutz, F., Oliveira-Santos, T., Guivant, J.E., de Aguiar, E., Badue, C., De Souza, A.F.: Evaluating the limits of a lidar for an autonomous driving localization. IEEE Trans. Intell. Transp. Syst. **22**(3), 1449–1458 (2020)
42. Perron, L., Furnon, V.: Or-tools. https://developers.google.com/optimization/
43. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), pp. 46–57. IEEE (1977)
44. Sakaguchi, K., Fukatsu, R., Yu, T., Fukuda, E., Mahler, K., Heath, R., Fujii, T., Takahashi, K., Khoryaev, A., Nagata, S., et al.: Towards mmwave v2x in 5g and beyond to support automated driving. IEICE Trans. Commun. **104**(6), 587–603 (2021)
45. Shuttleworth, J.: Levels of Driving Automation are Defined in New Sae International Standard j3016. SAE International, Warrendale, PA, USA (2014)
46. Völker, M., Kloock, M., Rabanus, L., Alrifaee, B., Kowalewski, S.: Verification of cooperative vehicle behavior using temporal logic. IFAC-PapersOnLine **52**(8), 99–104 (2019)
47. Wong, K., Gu, Y., Kamijo, S.: Mapping for autonomous driving: opportunities and challenges. IEEE Intell. Transp. Syst. Mag. **13**(1), 91–106 (2020)