Distributed Simulation of Cooperatively Interacting Vehicles*

Christian Frohn[1], Petyo Ilov[1], Stefan Kriebel[2], Evgeny Kusmenko[1], Bernhard Rumpe[1], and Alexander Ryndin[1]

***Abstract*— The field of cooperatively interacting vehicles requires complex simulation infrastructures dealing with various aspects such as vehicle, traffic, and communication models. In this work we present a modular and extensible simulator architecture, design patterns, and best practices for this domain. We show how extension points for co-simulators can be employed allowing the engineer to tailor a simulation environment to his needs. Moreover, we introduce the sectoring approach distributing the computational burden of a simulation over a series of workers thereby allowing us to cope with a large number of participants residing in large urban areas.**

I. INTRODUCTION

The development of highly-interconnected transportation systems is a challenging process requiring a large amount of testing, validation, and verification in order to guarantee safety and standard-compliance, e.g. with ISO26262. Simulators have always played a key role in automotive engineering processes revealing properties of a system under development or parts thereof without having to deploy it on expensive hardware or endangering human lives. In [1] we presented a simulator framework for the evaluation of model-based software engineering methods in the field of autonomous vehicles. The question arises which capabilities a simulation platform needs to exhibit in order to support an elaborate *cooperative* vehicle development process. From our experience in previous academic and industrial projects we derived the following set of requirements a cooperative driving simulator for off-line application in automated development processes needs to fulfill and which we missed in available solutions:

(R1) World model: depending on the simulation task, different abstraction levels of the physical laws and the environment may be required. The detail level should be adaptable in order to adequately capture reality while ensuring an optimal utilization of computational resources. **(R2)** Communication: efficient cooperation is only possible if an adequate communication channel is available. Similarly to **(R1)**, the detail level of the employed communication model depends highly on the simulation task. While an abstract channel model only simulating the quantities latency, bandwidth, and outage may be sufficient for the majority of simulation scenarios, more details including multi-path propagation and the Doppler effect particularly inherent for mobile communication but also protocol stacks might become indispensable. To allow a seamless integration of co-simulators, e.g., for communication, a simulator coupling approach allowing for the creation of simulator *product lines* is required. **(R3)** Distribution: due to a potentially large number of participants and vast simulation areas, simulation tasks may become computationally expensive. Therefore, a scalable concept to distribute the workload over multiple workers is required. **(R4)** Driving system integration: seamless integration of vehicle control software into a simulator is mostly accomplished using ROS [2] or similar middleware ensuring a low coupling between simulator and controller. However, such middleware mostly relies on asynchronous execution models potentially leading to synchronization problems. Therefore, synchronous integration alternatives still ensuring a high level of decoupling are required. **(R5)** Vehicle extensibility: it must be possible to extend a vehicle by new elements like sensors or physical parts. **(R6)** User interface: both headless and visualized simulation delivering reproducible results are needed. The former enables efficient *continuous simulation* and testing inside an automated work-flow while the second helps to identify and understand erroneous vehicle behavior.

The aim of this work is a modular object-oriented simulation platform for **cooperative vehicles** living up to the aforementioned requirements. The project web site containing practical information can be found at https://se-rwth.de/materials/montisim. We encourage the reader to inspect our simulator and to use it for further research. The main principles as well as our first main contribution, namely design patterns for event-driven **simulator coupling** are presented in section II. In section III, as the second contribution we present a **distributed simulation** approach enabling realistic simulations of large urban areas by load distribution. In section IV we elaborate on our design patterns for controller integration. In section V, based on a cooperative crossing example we demonstrate how the reference implementation of our simulator can be used for cooperative driving simulation.

II. COOPERATIVE VEHICLE SIMULATION

Research on cooperatively interacting vehicle simulation has been conducted in works such such as TraNS [3], iTETRIS [4], and Veins [5]. These solutions use SUMO [6] for the microscopic traffic simulation, whereas the discrete event simulation of the networking part is delegated to ns-2, ns-3, and OMNeT++, respectively [7]. The bidirectional simulator coupling in Veins relies on the TraCI approach described in [8]. Since these simulators don't live up to all of the requirements introduced in section I, e.g., due

*This work was supported by the Grant SPP1835 from DFG, the German Research Foundation.

[1]Department of Software Engineering, RWTH Aachen University, Germany, {kusmenko,rumpe}@se-rwth.de

[2]BMW Group, Munich, Germany, stefan.kriebel@bmw.de

[FIK+18] C. Frohn, P. Ilov, S. Kriebel, E. Kusmenko, B. Rumpe, A. Ryndin:
Distributed Simulation of Cooperatively Interacting Vehicles.
n: International Conference on Intelligent Transportation Systems (ITSC'18). IEEE, 2018.
www.se-rwth.de/publications/

to the lack of realistic physical behavior, no means for simulation distribution and vehicle extensibility, we present a new software architecture for the simulation of cooperatively interacting vehicles based on the simulation framework for model-driven autonomous vehicle development introduced in [1] and recapped in the following paragraph.

In [1], the main vehicle and environment simulator is a **discrete time simulator**. After the initial setup of the simulation model and the driving scenario, the simulation is started and the ongoing physical processes of the virtual environment are simulated in a *simulation loop*. This loop acts as a time emitter and advances the simulation time according to a predefined, possibly dynamically adapted resolution. Thereby it governs the execution and synchronization of all simulation modules while decoupling simulation time from real time. In this way we ensure that no race conditions are possible and the simulation results are by no means affected by the executing hardware or the operating system. As demanded by **(R5)**, simulated vehicles are assembled of modular and exchangeable components such as a communication system, virtual sensors, and most important, a controller unit computing actuator commands for acceleration, braking, and steering of the vehicle based on the incoming sensor signals. The virtual environment is created from OpenStreetMap data enabling the simulation of real areas. The simulation of object movement and collision detection in the environment is performed using the rigid body model [9] by solving its differential equations in an Euler loop. Alternative physics and vehicle models can be integrated in order to achieve the desired detail level as required by **(R1)**. Particularly, the Euler loop can be replaced by Modelica models implementing the functional mockup interface and solving the system of differential equations internally [10].

So, how can we reuse this existing solution as a blackbox while extending it by all the functionality needed for vehicle cooperation? In contrast to physics, communication networks are best simulated using the **discrete event simulation** paradigm. Instead of advancing the simulation time in small steps possibly not containing any state changes, the idea of discrete event simulation is to schedule and process a sequence of events such as *sending* and *receiving* of messages while no simulation is needed for the time intervals in between. Information attached to an event defines how the event needs to be processed, which usually creates new events scheduled for a future cycle. Thereby, channel properties such as bandwidth but also the message size and protocol overhead are taken into account in order to schedule the upcoming events realistically. We developed a lightweight general purpose discrete event simulator, which can later be reused for other discrete event co-simulators, e.g. varying tire pressure values or weather effects. To create a specialized variant of the general purpose discrete event simulator exhibiting domain specific behavior such as communication we rely on extensibility concepts provided by object orientation, namely inheritance and delegation. As described in requirement **(R2)**, a high degree of flexibility

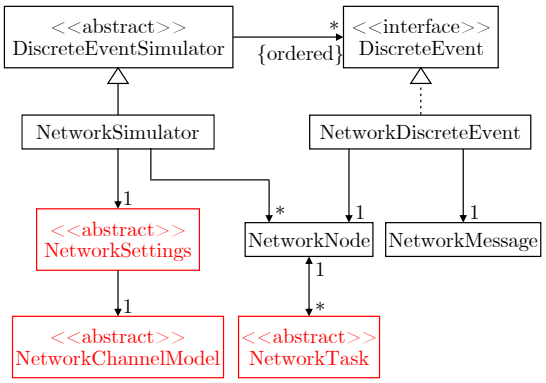

Fig. 1. Class diagram for the network simulator

and support for different network models are needed for the simulation of cooperatively interacting vehicles. Different entities in the network simulator are implemented to simulate various types of communication. Abstract classes and interfaces are used to define the required basic functionality of these entities. New network models can be introduced by subclassing the provided interfaces. The principal class hierarchy is illustrated as a UML class diagram in Fig. 1.

`NetworkNodes` represent devices equipped with the ability to communicate. Each network node is capable of handling a variety of networking tasks. The different tasks of the simulated network stack are implemented as subclasses of the abstract `NetworkTask` class. Each network node forwards its `NetworkDiscreteEvents` to all its network tasks which in turn decide individually if and how to handle these events. This mechanism allows network tasks to exchange messages without knowing each other, which is inspired by the **mediator** design pattern [11]. Algorithms for collision avoidance, priority handling at intersections, and traffic optimization can be implemented in network tasks with the help of data from vehicular communication.

The `NetworkChannelModel` represents the physical channel between communicating network nodes. For each message sent over the network it decides if and when it arrives at other nodes. The concrete implementation can be exchanged by sub-classing to enable the application of different channel models ranging from fixed delays and outages to multi-path propagation and Doppler effect simulation. Model specific network settings such as the modulation scheme, code rate, and data rate are defined in subclasses of the abstract `NetworkSettings` class. Tasks of the network stack should not be forced to operate on the technical level of raw bit sequences using up a large amount of computational resources but seldom improving the accuracy of simulation results. For many scenarios it is desirable to pass application level data such as driving decisions through the communication system directly. On the other hand, a well designed simulator architecture should allow low level processing, as well. We provide this flexibility by the `NetworkMessage` class. Network messages serve as basic entities for information exchange containing all relevant data

to be transmitted. Common header information for protocols in the network stack, such as sender and receiver addresses, port numbers, etc. can be specified directly in a network message object. If the implementation of the network stack is required to operate on bit sequences, the corresponding bit level representation can be provided by this class.

The original vehicle simulator is completely ignorant of the discussed networking framework and we want to attach the new functionality as a black-box through an event-driven co-simulator interface. We achieve this by introducing the **co-simulator observer** design pattern. The general idea of the **observer** pattern [11] is that an observed process can notify a series of observing system components about occurring events thereby granting access to relevant information without an explicit relation between the observable and the observers. An observable component needs to provide a mechanism for the registration and management of observers, whereas observers have to implement an interface to receive the desired information. This design pattern enables unlimited extensibility in a software system and plays a central role for event driven communication architectures.

We enable a high degree of extensibility in our vehicle simulator by a variant of this design pattern by introducing a series of simulator related events. For example, the registered observers are notified at the start and at the end of the simulation as well as before and after every simulation cycle. New co-simulators need to implement the co-simulator event interface in order to receive information from the main simulator. Therefore, every co-simulator should be registered as an observer in the main simulator before a simulation run begins. As each co-simulator needs to implement the co-simulator interface, we provide it through our general purpose discrete event simulator. Hence, our network simulator simply inherits this functionality and can be integrated as a simulator observer without the need for any glue code. Once it is registered within the main simulator, the network simulator receives access to all the objects residing in the virtual world and creates network nodes for those which are able to communicate. This information is acquired through the role interface by requesting the communication role based on the role object pattern [12].

Fig. 2 illustrates the interaction of the coupled simulators for an exemplary simulation run. In this example, the discrete time step for each update is set to 10 ms and a notification is sent to the network simulator after each update step of the main simulator. Based on the time information included in this notification the network simulator advances its own time and processes all scheduled events chronologically until it reaches the current time of the main simulator. This simulator coupling mechanism is repeated for all co-simulators as long as the simulation is running. Note that a co-simulator might need to simulate events with a higher time resolution than its parent simulator. For instance, the event in the network simulator at 9 ms might have to access sensor data of a vehicle, but the main simulator has only simulated this information at t = 0 and t = 10 ms and hence needs to interpolate. We assume that interpolation inaccuracies are negligible for appropriately chosen time steps.

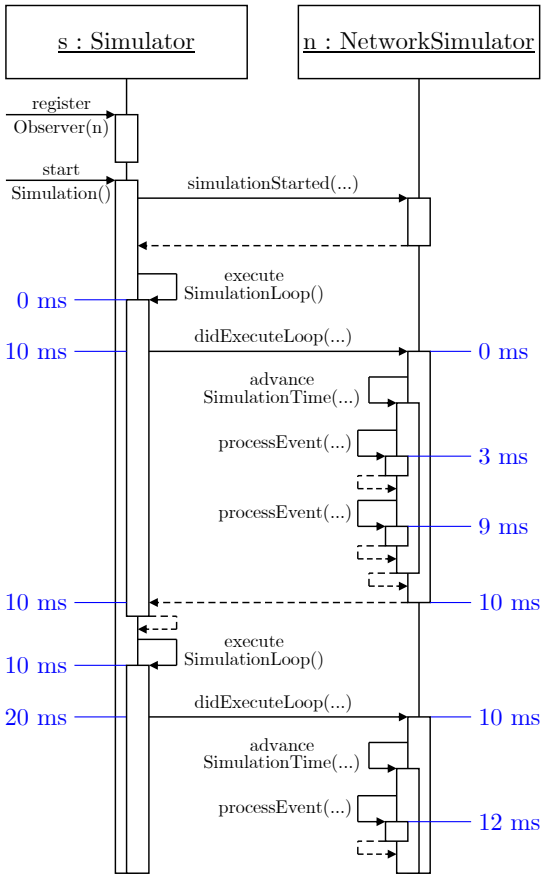

Fig. 2. Exemplary sequence diagram for co-simulator communication

III. DISTRIBUTED SIMULATION

Most software simulators are based on monolithic architectures and require all necessary resources, such as environment data and software modules including sensors, controllers, etc. to be bundled all together, similar to the simulator described in [13]. A downside of this approach is its constrained extensibility and scalability due to tight coupling of modules and their inability to be distributed over separate machines. The top system design of the simulator proposed in this work exhibits a distributed *client/server architecture* [14] in combination with the *Three-tier architecture* [15] and the *Model-View-Controller* pattern [11] as shown in Fig. 3. These architectural styles leverage the decoupling of simulator components and allow us to distribute them as bundles on different machines, thus increasing the overall scalability of the whole platform. Both, the three-tier architecture and the MVC pattern introduce the separation of data (model), logic (controller), and representation (view) into different layers allowing us to keep the client application thin and decoupled from the simulation logic making the simulator and its results accessible remotely, even from mobile devices.

However, this functional distribution is not enough if the simulator has to deal with massive amounts of data, large environments, and thousands of participants. To live up to

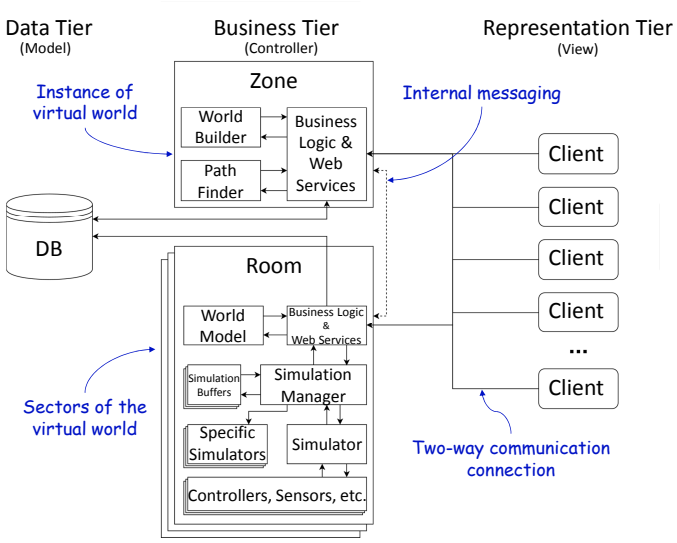

Fig. 3. Simulator overview

(R3), we employ concepts from the *SmartFoxServer* development kit for distributed multi-user online games splitting the simulator logic into *zone* and *room* modules [16]. This allows us to distribute the simulator over a series of workers, each responsible for a small spatial part of the simulation as suggested in the *SmartFoxServer architecture paper* [17]. Thereby, a zone module represents an instance of the whole virtual world. It is responsible for splitting the world into spatial sectors of predefined size, which are later managed by independent room modules. It is the first layer of the web server providing an interface for clients. Therefore, zone modules manage global data and events such as hand-overs, process simulation requests, and scenario definitions. They create rooms and keep track of load balancing and user management.

A *room* module on the other hand manages a single spatial sector as well as its dedicated *sub-simulator*. Each room can serve multiple simulations requested by different clients and provide simulation frame data. The latter captures *state changes* of all objects in a simulation including vehicles, pedestrians, etc. necessary for the presentation on client machines and result analysis. Since each room handles only its own sub-simulation, it requires just a small fraction of resources a single simulator processing the whole virtual world would allocate. We observed that this divide and conquer approach increases the overall map size a single machine can handle. Distributing the simulation over several machines in a cluster as well as a large scale performance study are subjects of ongoing work.

Both zone and room modules are parts of the *business tier* in the overall platform architecture. Thus each of these modules resides on a simulation server. This comes with the advantage that a user does not have to provide computational resources for distribution management. Whenever a vehicle leaves a sector and hence needs to be transferred to another room or the result of an action of one sector is required in another one, *internal messages* are sent between user defined *room extensions*. *Internal messaging* is a SmartFoxServer feature allowing extensions loaded in separate class loaders and performing semantically different tasks such as room management and in-room simulation management to exchange messages. This mechanism allows developers to define custom data flows and to handle simulator specific events.

In order to split the virtual world into sectors, we make use of a map splitting approach similar to the one used in [18] where the world is subdivided into rectangular sectors with common borders. Additionally, we let sectors overlap to avoid unnecessary sector changes whenever a vehicle is driving along a sector border. Furthermore, the overlapping areas are of great importance for the synchronization process: only the objects residing in those areas need to be synchronized between adjacent sectors, thereby dramatically reducing the synchronization overhead. In order to react to changing traffic load, dynamic sector sizes need to be investigated in the future. The simulation complexity can be either estimated in advance to create appropriate static sector sizes or measured on-line in order to adapt the sectors according to changing requirements.

The building process of the distributed environment takes place as follows: An OSM map file provided by the scenario model is loaded and parsed from its original XML format into a Java based graph representation. Then the world is split into rectangular regions with regard to the desired sector size and overlapping border area width. Thereby, the sector size should be chosen carefully to ensure that computational resources are exploited at the best. The overlapping border areas have a direct influence on the simulation behavior: too narrow overlapping areas result in numerous unnecessary vehicle hand-overs introducing co-simulation inaccuracies. Too wide overlapping areas however require a lot of information to be stored and processed redundantly thereby wasting valuable simulation resources. For each overlapping area in a sector, references to the sectors that share the same overlapping area are assigned to achieve a two way binding between the adjacent sectors. Note that an overlapping area can be shared between no more than two sectors. Thus adjacencies on the diagonals are not directly possible, but rather achieved by lookups in overlapping areas of directly adjacent sectors.

Although the map splitting approach allows for server load distribution over multiple workers serving different sectors, it does not come without cost: distributed path finding is intricate and not always leads to optimal results, since conventional path finding algorithms only work for complete graphs [19]. As each sector can only see the part of the world it is responsible for (which also implies a limited view on the world *map*), a distributed path finder is required, introducing algorithmic overhead. Our solution to this problem is to release the navigation from the vehicle and to provide it as a *simulator service* available for all objects such as vehicles residing in a simulation. After resolving the sectors containing the start and the destination

node, a heuristic is used to estimate the best sequence of sectors to traverse - in most cases and provided that the sector size is sufficiently large, it does not make sense to search sectors to the east when the destination is situated to the west of the vehicle. The first sector starts a standard Dijkstra route search. Whenever it finds a border to the heuristically determined sector to go through next, it can ask it via external messaging to find a path to the subsequent, third sector. As a result the first sector obtains a list of shortest distances between the given border node and the border nodes leading to the third sector found by the second sector. Thereby border nodes not having a direct access to the subsequent sector are pruned in order to reduce the search tree. Using this information, the first sector extends its map graph without the need to care about intermediate nodes inside the other sectors. The extended graph is now used to continue the Dijkstra search by requesting routes from further sectors until the destination node is found in the target sector. Of course, this algorithm does not guarantee optimality of the solution found. But provided the sectors are reasonably large, it mostly provides excellent results while keeping the complexity low. To facilitate the work with the proposed simulation framework as required by **(R6)** we designed a simple interface accessible through a web browser to maintain platform independence and high availability. It allows a user to define new scenario models using a simulation definition language, request simulations, and visualize or access simulation results. Upon selection of a scenario all involved vehicles are registered in the sub-simulator serving the sector in which these vehicles are initially spawned. A *simulation frame buffer* is created to which simulation frames of this scenario are written. Once the simulation is finished, the frames can be fetched by the client, thus enabling a smooth visualization independent of the actual simulation time. This is another application of the mediator pattern decoupling the delivery of simulator results from the presentation and enabling simulation replays without having to re-simulate the scenario.

IV. Vehicle Model Integration

The purpose of a simulator for automated cooperative driving is of course the evaluation and testing of cooperative driving systems and parameters. Often control systems are integrated into a simulator using a middleware such as ROS using asynchronous non-blocking calls, e.g. in Gazebo [2]. Although this approach ensures low coupling and is easy to integrate, it suffers from synchronization issues. If the simulator needs less time for one simulation cycle than the controller application, the latter will not be able to serve actuator values on time. Consequently, simulation results highly depend on the executing hardware and software setup and may not be reproducible as required by **(R6)**. In Gazebo the problem can be worked around using plug-ins inserting time delays. With the concept of model proxies we introduce a more controllable model integration strategy as required by **(R4)**. A model proxy exhibits a standard controller interface. For the simulator it appears to be a controller integrated directly into the vehicle which can be used by standard function calls. However, instead of actually computing actuator commands, the proxy forwards the sensor inputs to a remote controller application which in turn has to implement the corresponding adapter to be able to talk to the model proxy. What looks similar to the Gazebo/ROS approach at first sight has the advantage that the model proxy ensures synchronization by halting the simulator loop while waiting for the controller application to deliver a reaction for the actual simulation cycle. As proof of concept we provide an RMI based model proxy to allow for remotely executed controllers. This solution involves an RMI server, which contains all the implemented model adapters and a manager, which is responsible for the creation and deletion of adapter instances on demand as well as providing access to these instances to the RMI client. Whenever an external controller model is required by a model proxy, the RMI client requests an adapter for this model to be created on the RMI server. In contrast to ROS there is no need to run a separate controller process for each vehicle manually. Once the model adapter is accessible, the RMI client is allowed to use it via the general model adapter interface, allowing adapters, respectively the models they are wrapping, to be executed as functional blocks. The calls are blocking and synchronous ensuring full control over the execution cycle. As the simulator itself does not need to care about the protocol behind the communication, any middleware can be supported by adding further model proxies. The desired integration scheme can then be set in the simulator configuration. An example integrating our simulator into a model driven development methodology for vehicle controllers is EmbeddedMontiArc Studio [20]

V. Experiments

The presented simulator platform can be employed for various simulation scenarios in the domain of cooperative driving. To illustrate potential applications we evaluated a situation with four vehicles approaching an intersection with the priority to the right rule. The complete example is captured in https://youtu.be/kN-hS6lgqOk. All vehicles have the same velocity and start with the same distance to the intersection. Although sensors are able to detect this situation, a deadlock is likely to occur without any kind of cooperative interaction by vehicular communication. Since successful cooperation should avoid collisions by handling the priority rules at the intersection correctly, vehicle collisions are measured to evaluate the properties of the simulated vehicular communication and algorithms. This is the reason why we disabled collision avoidance features based on sensor data for this scenario. Additionally, we measure average latencies and average message counts on different layers of the simulated network stack. Thereby, we compare three channel models in this evaluation. The simple channel model represents direct vehicular communication with latencies and different data rates, but without any kind of message loss and outages. The direct channel model and the cellular channel model include computations for an outage probability and

failed message transmissions. The cellular model simulates a cellular network topology with base stations controlling the communication. In this example, the simulation loop frequency is set to 30 Hz and the three available channel models are used with various settings for data rate, modulation, and code rate indicated by the channel model indices 0, 4, and 7. Thereby, index 0 operates with a low data rate but a robust modulation scheme (e.g. BPSK@3 MBit/s), index 4 is intermediate (e.g. 16QAM@12 MBit/s) and 7 is fast but more error-prone (e.g. 64QAM@27 MBit/s). Due to channel randomness we employ a MonteCarlo simulation for each setting. Fig. 4 depicts the average message latencies in the left plot, i.e. the expected delay it takes a message to arrive at the intended receiver. Obviously, message latencies decrease with an increased data rate due to a reduced transmission time. The plot on the right shows how reliable the communication schemes in our scenario are. Thereby *collision simulations* denotes the total number of simulations featuring at least one collision. *Collision objects* is the number of vehicles involved in a crash due to failed communication. Fig. 5 depicts the number of messages sent through the network on average using direct and cellular communication, respectively. Note that messages are counted hop-wise. A message forwarded by the base station in the cellular communication case counts as a new message. This leads to approximately twice the number of messages when using cellular communication compared to direct communication.

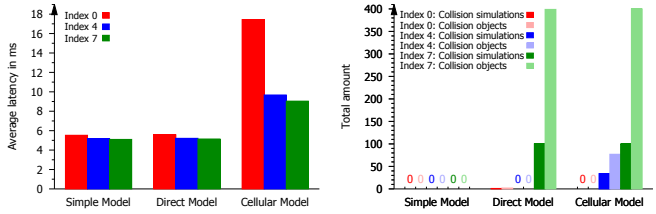

Fig. 4. Scenario evaluation for latencies and collisions

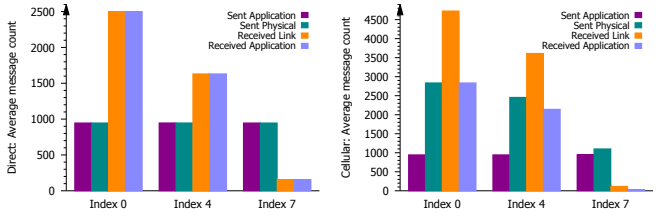

Fig. 5. Scenario evaluation for message amounts on network stack layers

VI. CONCLUSIONS

In this work we have presented a versatile simulator architecture fulfilling a given core set of requirements for cooperative vehicle simulation. The employed design patterns including co-simulation observers, mediators, zoning, and others can be re-used to build generic and customizable simulator frameworks not restricted to the cooperative driving domain. We discussed how complex simulation tasks can be tackled successfully by the zoning approach and how extensions can be added using our co-simulation infrastructure, e.g. for network simulation. Investigations on dynamic sector sizes depending on the load distribution in the simulated areas and a scalability study are subject of future work.

REFERENCES

[1] Filippo Grazioli, Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Simulation Framework for Executing Component and Connector Models of Self-Driving Vehicles. In *EXE at MODELS*, 2017.

[2] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an Open-Source Robot Operating System. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[3] M. Piórkowski, M. Raya, A. Lezama Lugo, P. Papadimitratos, M. Grossglauser, and J.-P. Hubaux. TraNS: Realistic Joint Traffic and Network Simulator for VANETs. *SIGMOBILE Mob. Comput. Commun. Rev.*, 12(1):31–33, January 2008.

[4] Michele Rondinone, Julen Maneros, Daniel Krajzewicz, Ramon Bauza, Pasquale Cataldi, Fatma Hrizi, Javier Gozalvez, Vineet Kumar, Matthias Röckl, Lan Lin, et al. iTETRIS: a modular simulation platform for the large scale evaluation of cooperative ITS applications. *Simulation Modelling Practice and Theory*, 34:99–125, 2013.

[5] Christoph Sommer, Reinhard German, and Falko Dressler. Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis. *IEEE Transactions on Mobile Computing*, 10(1):3–15, January 2011.

[6] Daniel Krajzewicz, Jakob Erdmann, Michael Behrisch, and Laura Bieker. Recent development and applications of SUMO - Simulation of Urban MObility. *International Journal On Advances in Systems and Measurements*, 5(3&4):128–138, December 2012.

[7] Saba Siraj, A Gupta, and Rinku Badgujar. Network simulation tools survey. *International Journal of Advanced Research in Computer and Communication Engineering*, 1(4):199–206, 2012.

[8] Axel Wegener, Michał Piórkowski, Maxim Raya, Horst Hellbrück, Stefan Fischer, and Jean-Pierre Hubaux. TraCI: An Interface for Coupling Road Traffic and Network Simulators. In *Proceedings of the 11th Communications and Networking Simulation Symposium*, CNS '08, pages 155–163, New York, NY, USA, 2008. ACM.

[9] Roy Featherstone. *Rigid body dynamics algorithms*. Springer, 2014.

[10] Hilding Elmqvist, Sven Erik Mattsson, and Martin Otter. Modelica-a language for physical system modeling, visualization and interaction. In *CAV*, 1999.

[11] Erich Gamma. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass, 1995.

[12] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. The role object pattern. In *Washington University Dept. of Computer Science*. Citeseer, 1998.

[13] Miha Ambroz, S. Krasna, and Ivan Prebil. 3d road traffic situation simulation system. *Advances in Engineering Software*, 36(2):77–86, 2005.

[14] Andrew S. Tanenbaum and David Wetherall. *Computer networks, 5th Edition*. Pearson, 2011.

[15] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory and Practice*. Addison-Wesley, 2007.

[16] gotoAndPlay(). *SmartFoxServer 2X*. Available at `http://docs2x.smartfoxserver.com/`.

[17] Marco Lapi. Smartfoxserver 2x server architecture white paper, 2012. Available at `http://www.smartfoxserver.com/downloads/sfs2x/documents/SFS2X_WP_ServerArchitecture.pdf`.

[18] Marios Assiotis and Velin Tzanov. A distributed architecture for MMORPG. In Adrian David Cheok and Yutaka Ishibashi, editors, *Proceedings of the 5th Workshop on Network and System Support for Games, NETGAMES 2006, Singapore, October 30-31, 2006*, page 4. ACM, 2006.

[19] George T. Heineman, Gary Pollice, and Stanley M. Selkow. *Algorithms in a nutshell - a desktop quick reference*. O'Reilly, 2009.

[20] Evgeny Kusmenko, Jean-Marc Ronck, Bernhard Rumpe, and Michael von Wenckstern. EmbeddedMontiArc: Textual modeling alternative to Simulink. In *EXE at MODELS*, 2018.