[HKKR19] A. Hellwig, S. Kriebel, E. Kusmenko, B. Rumpe:
Component-based Integration of Interconnected Vehicle Architectures.
In: 30th Intelligent Vehicles Symposium (IV'19). Workshop on Cooperative Interactive Vehicles, 2019.
www.se-rwth.de/publications/

Component-based Integration of Interconnected Vehicle Architectures*

Alexander David Hellwig[1], Stefan Kriebel[2], Evgeny Kusmenko[1], Bernhard Rumpe[1]

Abstract— **Mapping the logical software architecture of a vehicle to a technical solution is not a straightforward task. A particular challenge is communication: software components developed by different teams and deployed across the E/E architecture need to be able to exchange data. Middleware solutions have been developed to enable low coupling of distributed logical software components. Building a distributed architecture on a middleware solution is mostly accomplished by encapsulating logical components into middleware wrappers. This is not only time-consuming, but also requires platform-specific understanding, and results in a multitude of architectural variants tailored for particular set-ups. For instance, lengthy validation processes ensuring functional correctness and safety require simulations of intelligent vehicle systems in different simulators, environments, and on different abstraction levels. This leads to the necessity of individual integration schemes for both simulation and deployment. We propose a component-based modeling approach separating platform-agnostic logical models from middleware aspects. Therefore, the model compiler is instrumented with middleware tags related to the elements of the logical model. Generating the required middleware code automatically, we aim at better component re-usability minimizing the need for hand-crafted glue-code for interprocess and simulator integration.**

I. Introduction

The integration of automotive E/E architectures is a challenging process. Software components are often developed by different teams or suppliers and need to be integrated afterwards. Thereby, they are likely to be distributed across different Electronic Control Units (ECUs) of the vehicle and need to communicate over a network. In V2X communication this challenge is further intensified as traffic participants need to be able to talk to peers they have never seen before.

To maintain low coupling between logically independent components, middleware solutions such as Robot Operating System (ROS) provide common protocols and implementations for different platforms [1]. Two parties communicating via a middleware neither need to know each other, nor do they have to run on the same platform. Technical aspects of the communication as well as data conversion are handled by the middleware.

Usually, logical components are enriched with middleware code by encapsulating them into middleware wrappers. The architect then assembles the wrapped components into a system architecture. Although, the logical components remain untouched, the composed architecture is polluted by middleware-specific details. Changing the middleware requires re-writing the middleware code and leads to new architectural variants although the logical model remained the same.

Since interconnected vehicles are not isolated software entities, but rather complex Cyber-Physical Systems (CPSs), tests are usually supported by simulators. Integrating a software component under test into a simulator is often realized using middleware solutions, as well. Simulators such as Gazebo [2], MontiSim [3], [4], and CoinCar [5] provide middleware interfaces to exchange sensor and actuator commands with external processes. However, through the course of development tests might require different simulator environments. Adapting the architecture under test to another simulator interface at every development stage leads to a multitude of test variants requiring a lot of maintenance, but without providing additional functional value.

We propose a component-based modeling approach separating platform-agnostic logical models from middleware aspects. Therefore, the model compiler is instrumented with **middleware tags** related to logical model elements such as ports. The tags contain a minimum set of information required to generate the middleware code automatically. If another middleware scheme is required, e.g. if ROS needs to be replaced with ROS2, all we need to do is to exchange the middleware model. Tag-based compiler instrumentation enhances component re-usability while minimizing the need for variant management and hand-crafted glue-code for interprocess and simulator integration.

In section II we give a brief overview of the component-based engineering concepts this work is based on. Section III presents a running example to illustrate the problems we want to tackle. The core of this work covering code generation is given in section IV. In section V we present an evaluation project to demonstrate the proposed approach in action. Section VI provides a comparison of related approaches to distributed systems modeling. We conclude our work in section VII. More examples are provided on the paper website `http://www.se-rwth.de/materials/middleware/`.

II. Background

While in mechanical engineering it is common to decompose a system in its physical parts, the software engineering approach is to find appropriate levels of abstraction and to refine them until a state is reached where the final system can be assembled. The Component & Connector (C&C) modeling paradigm known from tools such as Simulink has been widely used in engineering domains to tackle system

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

[1]Department of Software Engineering, Faculty of Computer Science, RWTH Aachen University, Ahornstraße 55, 52074 Aachen, Germany
alexander.hellwig@rwth-aachen.de,
{kusmenko,rumpe}@se-rwth.de
[2]BMW Group, Munich, Germany, stefan.kriebel@bmw.de

architecture design in a divide and conquer manner. The system under development is broken down into *components* which in turn can be hierarchically subdivided into further *subcomponents*. A component is a functional block fulfilling a self-contained task. A component's interface is given by a set of typed data entry and exit points called *ports*. Components only exchange data if their ports are connected by a *connector*. **The question to be answered in this work is how communication schemes for C&C architectures can be modeled without adapting the functional models.** The presented concepts are not bound to a specific language, but can be transferred to any textual or graphical C&C language.

We will build our middleware modeling framework on top of EmbeddedMontiArc (EMA), a family of textual Domain Specific Modeling Languages (DSMLs) developed for CPS and embedded systems design [6], [7]. As is inherent to the C&C paradigm, components are first-level citizens in EMA and can be defined using the `component` keyword followed by a name and optionally a set of parameters as depicted in our example model in fig. 1. The body of a component contains a type-safe interface defined by a set of named input and output ports, c.f. lines 2-5. To define the component's behavior, it is decomposed into smaller subcomponents, which are instantiated using the `instance` keyword followed by the component type to instantiate, c.f. lines 6-8. Finally the ports of the subcomponents are reconnected using the `connect` operator, thereby defining the data flow of the system, c.f. lines 9-12.

EMA provides an abstract math oriented type system with the primitive types `Z`, `Q`, and `C` denoting integers, rational, and complex numbers, respectively. A primitive type can be enriched by a range, a resolution, and a corresponding SI unit which facilitates modeling physical processes and imperfect devices' properties: `Q(-2.7V : 100mV : +2.7V)` denotes a voltage variable taking values from -2.7 V to 2.7 V in 0.1 V steps. Since many CPS tasks rely on matrix calculus, a primitive type can be extended to a matrix by specifying its dimensions, e.g. `Q^{2,3}`. The dimensions are fixed at compile-time and cannot be changed at runtime, facilitating verification, contributing to system robustness, and enabling the EMA code generator to produce high performance code.

To demonstrate the generation of *distributed* architectures, we will use ROS [8]. ROS is a platform independent middleware designed for the development of robotics applications. In particular, it enables asynchronous communication of distributed components using the publisher/subscriber pattern: ROS **nodes** can share data by publishing it to named and typed **topics**. Other nodes can receive this data by **subscribing** to the topics of interest.

III. Running Example and Problem Statement

Consider the collision-preventing cooperative intersection controller modeled as a C&C architecture depicted in fig. 5. It analyzes trajectories of vehicles approaching an intersection and warns those heading towards a collision. Such a system might be deployed in a Roadside Unit (RSU), a master vehicle of a local traffic system (LTS) [9], or in a testbed receiving data from a simulator.

The corresponding textual EMA syntax is given in fig. 1. For simplicity of notation we keep the types abstract and

```
1  component IntersectionController{
2      ports in Time timeCutoff,
3            in Trajectory trajectory[n],
4            in Boolean isActive,
5            out Boolean stop[n];
       instances of subcomponents defined in other artifacts
6      instance SingleSetCompare compare;
7      instance TrajectoryCollision trajCollision[x];
8      instance CollisionToStop collisionToStop;
9      connect trajectory[:] -> compare.trajectoryIn[:];
10     connect timeCutoff    -> trajCollision[:].timeCutoff;
11     connect collisionToStop.stop[:] -> stop[:];
12     /* other connections */}
```

Fig. 1: IntersectionController in written in EMA

leave out details irrelevant for the problem statement. The system receives the planned trajectories of nearby vehicles (`trajectory[n]`), a minimum safety time interval between two vehicles passing the intersection (`timeCutoff`), as well as the `isActive` signal to activate or deactivate the controller (lines 2-4). Furthermore it is expected to output `stop` signals to vehicles that currently drive on a dangerous trajectory (line 5) forcing these vehicles to decelerate. Following the divide-and-conquer principle we decompose our system into multiple subcomponents (lines 6-8): a `SingleSetCompare` component will create all possible pairs of trajectories which are then supplied to a proportional number of `TrajectoryCollision` components. The latter will check each trajectory pair for dangerous overlaps. Based on these overlaps, the `CollisionToStop` component determines a subset of vehicles that need to be stopped and informs them by sending a stop signal. The data flow is defined by reconnecting the ports of the system in lines 9-12.

An E/E modeling methodology for such a system should fulfill the following requirements: **R1 Middleware agnostic (models):** Domain models must remain middleware-agnostic to ensure re-usability in different environments as well as independence of the technical realizations. **R2 Middleware agnostic (generator):** The C&C and behavior code generator must remain middleware-agnostic and must not generate any middleware-related code. **R3 Minimization:** As middleware communication is expensive, it must not be used unless required (explicitly or implicitly) by the modeler. For all other data flows in the model the standard tight-coupling communication pattern as offered by the core generator must be used. **R4 Semantics invariance:** The semantics of the code generated by the core generator must remain untouched, i.e. preserving the scheduling and synchronization concepts. **R5 Build infrastructure:** The generated target code must include a build configuration providing the required depen-

dencies and linking the artifacts. **R6 Middleware coupling:** The combination of different middleware in one single model must be possible.

In the following sections we are going to present a model-based solution for middleware component communication living up to the aforementioned requirements.

IV. Tag-Based Multi-Platform Code Generation

EMA was developed for modeling *embedded* systems, i.e. systems often having a limited amount of resources but expected to deliver fast response rates or even real-time behavior. Based on the supplied Abstract Syntax Tree (AST) and the Symbol Management Infrastructure (SMI) of the model, the EMA code generator produces plain C++ code for the architecture and component communication. In contrast to some other component-based languages, EMA refrains from the use of a runtime scheduler. Instead, the scheduling is based on the *sorted order* algorithm known from Simulink. The data flow is analyzed at compile time and each component is tagged with an execution order ID. The execution order list is then used by the generator to create a static *synchronous* execution model.

To tackle the requirements introduced in section III we are going to present a tagging-based approach for generator instrumentation. Tagging provides a non-invasive way to enrich models with additional information such as extra-functional properties. In this paper we will use tags to declare middleware-specific information, thereby preserving the agnostic requirement **(R1)**: the C&C model itself does not contain any middleware-specific elements, the tagging model is defined as a separate artifact. A tagging model is a list of tags, each declaring the name of the model element in the referenced C&C model (which can be either a component or a port in a C&C language), the tag type carrying the semantic information, as well as some data refining the tag type. For our middleware tagging scheme we only allow attaching tags to ports. By adding a tag of a middleware type to a port (e.g. a ROS tag), we specify that the message exchange with its counterpart should be realized using the communication pattern of the indicated middleware, e.g. the ROS publisher/subscriber pattern. Furthermore, the tag can carry ROS specific configuration data including the data type and the topic name.

To ensure that the existing EMA-to-C++ code generator remains free of middleware-specific code, as required in **R2**, we maintain a loose coupling between the middleware generation process and the core C++ generator. Therefore, we decide to use the core generator as a black box and to develop a separate generator for each supported middleware. This requires coordination of the generation workflow and extension points for middleware generators. A well-suited design pattern for this problem is the star-bridge which extends the abstraction of the standard bridge [10] by arbitrarily many variation points for the implementation. The resulting architecture for our generator coupling mechanism is depicted in fig. 2.

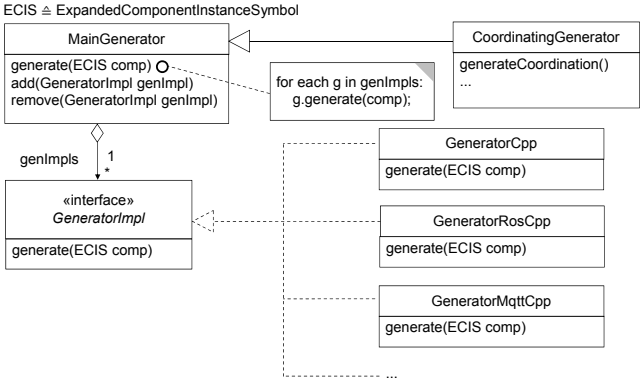

Fig. 2: Overview of the generator coupling architecture

The `MainGenerator` is an abstraction of the generation workflow. The concrete coupling mechanisms of the generated code are provided in the `generateCoordination()` method of its subclass `CoordinatingGenerator`. The coordinating generator produces an infrastructure to initialize, couple and call all the other generated system parts. Furthermore, the `MainGenerator` manages the set of needed code generators, the implementation parts of the star-bridge. Note that the list of concrete generators implementing `GeneratorImpl` includes both the EMA generator to generate the actual application logic as well as all the middleware generators.

All of these generators are unaware of each other's existence producing completely independent artifacts controlled only by the coordinating generator. The presented generator coupling approach is tailored to our specific problem and therefore extremely efficient: under the assumption that the core EMA generator produces code exhibiting a well-defined interface for accessing the ports of the C&C model, all middleware generators can create compliant middleware adapters without knowing more about other generators involved.

Note that the C&C model and the middleware model reside in two separate files, c.f. fig. 1 and fig. 4. However, the parser composes an intermediate representation of the two models, thereby creating a common AST and symbol table making it possible for the generators to navigate from the ports of the C&C model directly to the corresponding tags of the middleware model. Based on the composed intermediate model, all necessary artifacts including architecture, behavior, and middleware code are produced by the respective generators.

The architecture of the generated C++ code for the running example can be seen in fig. 3. Each middleware generator outputs a middleware adapter for the ports specified in the tag model of fig. 4 implementing the `IAdapter` interface. This interface contains an `init(...)` and a `publish()` method both to be used by the coordinator to control the communication. The former receives the component instance to be adapted as argument thereby creating the link between adapter and adaptee. Furthermore, it registers the adapter in the middleware system, e.g. by subscribing for a certain ROS

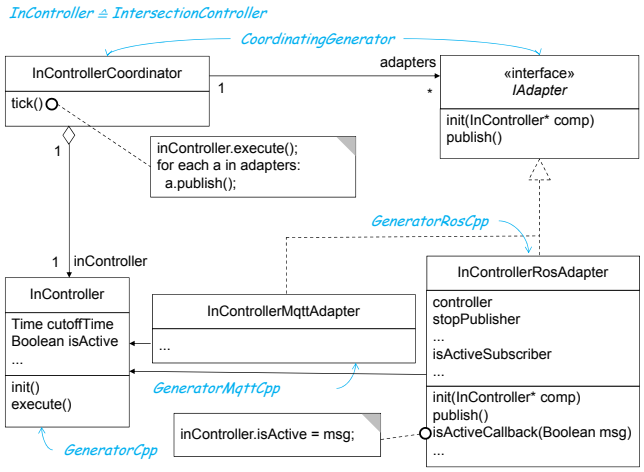

Fig. 3: Overview of the generated C++ code

topic. The latter is invoked periodically at the end of each execution cycle to publish the data at the adaptee's output ports. Moreover, an adapter obtains a middleware-specific callback function which is triggered by the middleware whenever new data becomes available and forwards this data to the corresponding input port of the EMA component. Since the coordinator can manage multiple adapters for one component, combinations of different middleware inside one model and even for the same port are possible satisfying **R6**.

To keep the semantics of the original C++ code, as required by **R4**, there is no synchronization between the system execution and middleware communication. A component is executed by the coordinator periodically *without* waiting for new inputs to arrive. If the middleware has not provided new data until the beginning of an execution cycle, the component will be executed using old data at the respective input ports.

Hence, leaving the original EMA-to-C++ generator untouched by the middleware generation process ensures that the previously discussed efficient synchronous execution model of EMA is not influenced by middleware. Generation of the technical communication infrastructure is also separated from the generation of the logic.

Now let's go back to our running example presented in fig. 1. To ensure that the tag model for the intersection

```
1 tags RosTags{
2   tag intersectionController.timeCutoff with RosConnection =
      {topic= (name=/timeCutoff, type=struct_msgs/Time)};
3   tag intersectionController.isActive with RosConnection =
      {topic= (name=/isActive, type=struct_msgs/Boolean)};
4   tag intersectionController.collisionToStop.activeIn with
    RosConnection;
5   tag intersectionController.compare.outA[1] with RosConnection;
6   [...]}
```

Fig. 4: Tag model for the running example

controller given in fig. 4 is valid, it is first checked against the corresponding *tag scheme*. The scheme we use here allows one to tag *ports* of component instances with all the necessary information needed to generate the desired middleware code. The middleware to use is specified using the **`tag`** `port` **`with`** `type` syntax. In the given model, ROS is the only middleware used, and hence all the ports are tagged with a `RosConnection` tag type.

The middleware-specific message description such as a ROS message type is generated automatically based on the port type information of the EMA model, e.g. for the `outA[1]` port of the `SingleSetCompare` component in line 5 of fig. 4. This automation however is not applicable if the generated code needs to be compliant with third-party systems, e.g. a simulator having predefined message types and topic names for its sensors and actuators. For ports at system boundaries the user can instruct the generator to use a predefined type and topic name, c.f. lines 2 and 3 in fig. 4.

Note that the underspecified `activeIn` port receives its data directly from the `isActive` port of the parent component which has explicit ROS type and topic information. In this case, the middleware tag of the parent component overrides the underspecified tag of the child component as depicted in fig. 5.

Now that the composed intermediate model has been enriched with all the necessary middleware meta-data, particularly by appending full middleware information to the tagged `PortSymbols` in the common symbol table, we use so called context conditions to verify that the user has defined a meaningful communication scheme. First, two ports can only be connected if they have either no or corresponding middleware tag types. Second, ports with middleware tags need to have compatible middleware meta-data, e.g. compatible message types and equal topic names for ROS. An overview of the files generated by the toolchain and the source code of the generated adapters can be found on the paper website.

The generated project hierarchy comes with all build files required to compile and link the projects, fulfilling requirement **R5**. The resulting architecture exhibiting the coupling between the coordinator, the adapters, and the actual application logic is depicted in fig. 3. Note once again, how the system is governed by the `IntersectionControllerCoordinator`. In the initialization phase the coordinator creates an instance of the `IntersectionController` and hands over its pointer to the adapter objects. In the operation phase, the coordinator triggers the `execute()` method of the `IntersectionController` and the `publish` methods of the adapters within a predefined frequency.

Now consider the `CollisionToStop` and the multiple `TrajectoryCollision` components in our `IntersectionController` example. The question arises why a developer would want to let subcomponents inside a self-contained system communicate via an expensive middleware connection instead of simple C++ calls as produced by the EMA core generator for systems

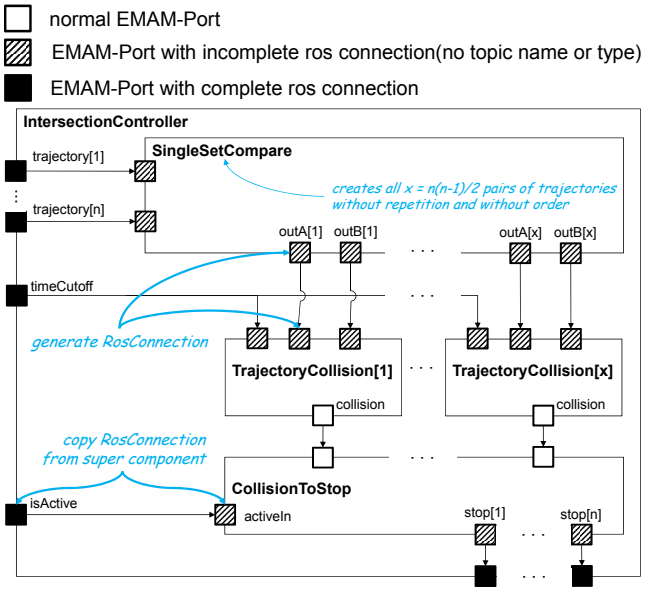

Fig. 5: Propagation of ros connections

run in a single process. The intention here is to model a *distributed* system architecture and to deploy it on different ECUs of a CPS.

This is achieved by assuming that our C&C model is an *undirected* graph where each component is represented by a node. Two nodes are connected with each other if and only if there is at least one *non-middleware* connector between the corresponding components in the C&C model. The resulting graph can now be clustered into disjoint partitions such that two nodes belong to the same partition if and only if there is a path between them. The obtained partitions are equivalent to clusters containing the components which need to be generated as plain C++ code. By performing the graph partitioning on our running example we can conclude that the specified ROS connections split the system into two clusters. Since the `SingleSetCompare` component only has ROS ports, it needs to reside in its own cluster. The `CollisionToStop` and `TrajectoryCollision` components end up in the second cluster, as they are connected by non-middleware connectors.

For each cluster we run the generator workflow separately to obtain an independently executable component for each ECU. The intermediate model is split up accordingly and each generation process receives only the part corresponding to the components of its cluster.

The described procedure enables a compact description of a distributed yet loosely coupled system architecture in a single model while the generator ensures that there is no middleware communication inside a partition, fulfilling the minimization requirement **R3**.

V. Evaluation

To test our system, we employ the ROS-based CoInCar simulator [5]. **TC1**: in a series of simulations, we want to verify that our intersection controller can prevent vehicle collisions on a 4-way intersection under non-optimal communication conditions between the controller and the vehicles.

In the test scenario two vehicles drive towards an intersection with randomized starting positions and velocities. To test the robustness of the controller a network delay between 0 and 1.5 s and a message drop chance of 0 to 100% is added. The vehicles have no intelligence and only stop upon a signal from the intersection controller. The crash rate is recorded to evaluate the system performance.

To integrate our cooperative intersection system with CoInCar's sensor and actuator system we enrich our model with ROS tags. As CoInCar prescribes the topics and data types, we adopt this information in the port tags instead of generating new message types for the respective ports. The only glue code we have to write addresses a fundamental difference between the used technologies: the simulator relies on dynamically sized arrays for the desired trajectories and EMA only supports fixed size arrays. Therefore, we have to write a converter cutting or padding these messages to a fixed size. The whole system, including the ROS code and build files, is then generated automatically while the driving model itself does not contain any middleware related information and could be re-used in other contexts.

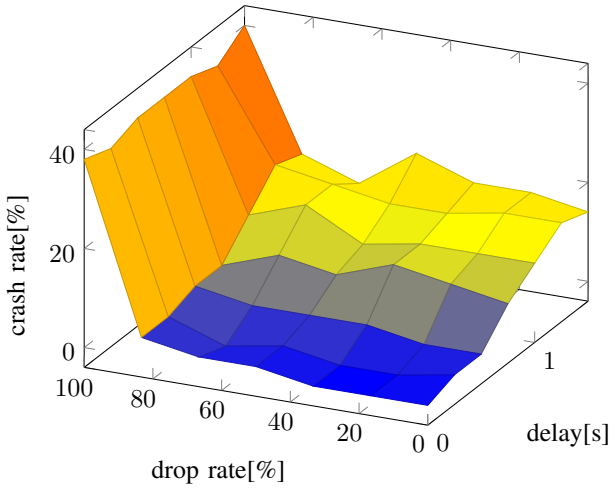

Fig. 6: Crash rate in relation to network delay and drop rate

From the simulation results, depicted in fig. 6, we can confirm that **TC1** is successful. For network delays of up to 250 ms and package drop rates of up to 80% the controller prevents collisions. Using a middleware model reduces the time to integrate the controller into the simulator significantly. The developer neither has to understand the implementation of the functional EMA model, nor to write ROS code. Whenever the functional model needs to be integrated into another simulator or a real system, all the developer needs to do is adapting the middleware tags. We observed that using EMA coupled with middleware tags to describe ROS-based communication eliminates several error sources and design pitfalls. Since the generated architecture is based on EMA semantics it forbids the following communication anti-patterns which are possible when writing ROS code

in C++ or Python (as long as all ROS components are situated inside a single model): first, in contrast to hand-written ROS code the type compatibility of publisher and subscriber is checked at compile time; second, it is not possible to publish to a topic nobody listens to and vice versa as an EMA model explicitly specifies a communication scheme by its connectors. Hence, misspelled topic names are identified at compile-time. Third, since EMA allows only one incoming connector per input port, it is not possible to let two publishers publish to the same topic. We observed that this eliminates overhead, e.g. adding sender information to each message. Hence, our generative modeling approach is an example, where the abstraction brought by model-based software engineering controls the usage of the underlying platform in a restrictive way reducing potential error sources.

VI. Related Work

In Simulink [11] middleware communication is modeled using predefined components. The Robotics System Toolbox provides the *Blank Message* to create empty messages which can be modified using bus assignments, the *Publish* component to publish data to a specific topic, as well as a *Subscribe* component to receive data from another ROS publisher. The logical architecture is polluted with middleware-related components; the user needs to use Simulink's variability modeling functionality in order to keep track of different model variants emerging throughout the development cycle. Aspect-oriented programming (AOP) can be used to add additional functionality as *Advices* to *Pointcuts* in a model [12]. Our tagging-based approach is specialized to extend the original behavior of a port with a middleware adapter. Thereby, the compiler combines the original logical model and the tags into a final consistent model guaranteeing type-safety. This is typically not possible with AOP, but indispensable for safety-critical systems.

Interface Description Languages (IDLs) can be used to describe communication between the components of a system in a platform independent way. Since EMA already describes the interface between components with ports and connectors, no additional IDL is needed. The tagging-based approach also allows us to create a single consistent model that contains all middleware-specific information, which can therefore be included in the static analysis of the model. Support for specific IDLs, such as CORBA IDL [13], can be added to our framework in the same way as ROS.

Platform-based design (PBD) is a model-based approach focused on orthogonalization of concerns [14]. The modeler splits the system into different levels of abstraction, called *platforms*, and describes their relationships using *mappings*. Each platform contains a number of *components*, which either contain a behavior or a communication model (including middleware). In contrast, our tag-based approach stores all middleware-specific information outside of components and ensures that maintainability and testablilty of the core model are not affected by the communication paradigm.

VII. Conclusion and Future Work

In this paper we presented a self-contained component-based engineering methodology for the development and integration of distributed cooperative systems. It enables the system designer to concentrate on the system's functionality while abstracting away from the technical realization. Middleware information can be specified in a dedicated tagging model, thereby separating the logical components from the integration scheme in a clean way while maintaining type-safety between the models. The toolchain enables the generation of loosely coupled distributed architectures as well as a seamless integration into existing simulators straight out of the C&C models.

References

[1] 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.

[2] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2149–2154. IEEE, 2004.

[3] 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 *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017.

[4] Christian Frohn, Petyo Ilov, Stefan Kriebel, Evgeny Kusmenko, Bernhard Rumpe, and Alexander Ryndin. Distributed Simulation of Cooperatively Interacting Vehicles. In *International Conference on Intelligent Transportation Systems (ITSC'18)*, pages 596–601. IEEE, 2018.

[5] Maximilian Naumann, Fabian Poggenhans, Martin Lauer, and Christoph Stiller. Coincar-sim: An open-source simulation framework for cooperatively interacting automobiles. In *IEEE Intl. Conf. Intelligent Vehicles*, 2018.

[6] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *ECMFA*, 2017.

[7] Evgeny Kusmenko, , Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Highly-Optimizing and Multi-Target Compiler for Embedded System Models. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*. IEEE, October 2018.

[8] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, May 2009.

[9] Jens Dankert, Christian Dernehl, Lutz Eckstein, Stefan Kowalewski, Evgeny Kusmenko, and Bernhard Rumpe. RapidCoop - Robuste Architektur durch geeignete Paradigmen für Kooperativ Interagierende Automobile. In *Automatisiertes und Vernetztes Fahren (AAET'17)*, February 2017.

[10] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[11] Mathworks Inc. Simulink User's Guide. Technical Report R2016b, MATLAB & SIMULINK, 2016.

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

[13] Jon Siegel and Dan Frantz. *CORBA 3 fundamentals and programming*, volume 2. John Wiley & Sons New York, NY, USA:, 2000.

[14] Kurt Keutzer, A Richard Newton, Jan M Rabaey, and Alberto Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE transactions on computer-aided design of integrated circuits and systems*, 19(12):1523–1543, 2000.