RWTH Aachen University
Software Engineering Group

Multi-Target Code Generation for Component and Connector Models in Intelligent Vehicles

Seminar Paper

presented by
Hellwig, Alexander

1st Examiner: Prof. Dr. B. Rumpe

Advisor: Evgeny Kusmenko

The present work was submitted to the Chair of Software Engineering

Aachen, June 1, 2018

Abstract

Cyber-physical systems combine complex mechanical, electrical and software systems. The integration of these worlds is a challenging process and results in highly distributed systems featuring a series of physical components including but not limited to sensors, actuators, electronic control units, as well as signal processing, planning, and decision making software. Middleware solutions are applied to decouple the software architecture from the underlying mechanical architecture. However, lengthy validation processes ensuring functional correctness and safety require simulations of cyber-physical systems in different simulators, environments, and on different abstraction levels. This leads to the necessity of individual integration scheme variants for the various simulation and deployment platforms used as these may exhibit different interfaces in conjunction with different middleware solutions. However, all the required variants should be generated from implementation agnostic logical models. In this work we present a code generator instrumentation and coupling approach increasing model re-usability during the different validation and integration steps thereby minimizing the need for hand-crafted glue-code for interprocess and simulator integration.

Contents

1 Introduction **1**

2 Preliminaries **3**

3 Running Example and Problem Statement **5**

4 Tag-Based Multi-Platform Code Generation **7**

5 Evaluation **15**

6 Related Work **19**

7 Conclusion and Future Work **21**

Bibliography **21**

Chapter 1

Introduction

here uppercase in abstract lowercase

Software development for Cyber-Physical Systems (CPSs) differs from other domains in the first place by their steady **interaction with their environment**, difficult and expensive validation processes. Apart from physical laws which are generally well understood, the environment contains a lot of stochastic behavior introduced by thermal noise, sensor imperfections, parasitic effects, model abstractions, as well as the behavior of third parties which cannot be influenced. Since such systems are often safety-critical and may endanger human lives, particularly if they malfunction, the manufacturer is obliged to provide evidence for the correctness of the system implementation as well as compliance with standards such as the ISO26262 and law in force. This becomes more and more difficult using general purpose languages and conventional programming methods. Instead, automation and digitalization of software engineering processes is required at least in the same extent as in other industries. Therefore, model based software development technologies have been prospering in recent years providing direct support for domain concepts as well as delivering big amounts of generated, quality assured code, verification routines and the like. In [HKK+18] it has been shown how BMW's **SMARDT** development process which is steadily developing towards a quasi standard in the automotive industry supports the development of such systems by model based test code generation from Component & Connector (C&C) models and artifact consistency on and between different abstraction levels. Furthermore, it is capable of finding design inconsistencies in early development stages based on the employed formal syntax of SyML diagrams [KKRvW18]. Given the potential power of model driven engineering, the question arises, how model and modeling tool design should look in order to ensure model re-usability in different phases of a product. There should be no need to change a logical or technical CPS model to deploy it in different test environments often exhibiting incompatible interfaces. Therefore a set of configurable code generators is necessary to deliver the right code variants for all development phases and execution environments while maintaining the separation of concerns principle.

The contribution of this paper comprises **a tag-based generative approach to multi-target modeling for C&C models** enabled by a so called *product line* of generators as well as a component clustering approach enabling the automated distribution and deployment of C&C models on different target platforms such as simulators and prototypes. A **case study** dealing with the development and testing of cooperative driving controllers in a simulated environment shows the application of the presented concepts in action .

Chapter 2

Preliminaries

The **SMARDT** methodology is an automotive development process aiming to combine engineering disciplines with computer science principles and to tackle the complexity of CPS by introducing the two development axis abstraction and decomposition. 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 reach which can be compiled into executable code. The former axis was presented in [HKK+18] and consists of the four layers **(1) object of reflection**, **(2) logical layer**, **(3) technical concept**, and the **(4) hardware/software realization layer**. The process relies heavily on formalized SysML diagrams combined with OCL constraints allowing for automated test case generation from specifications at each development stage, structural refinement checks between the levels as well as test case refinement transformations. In [HKK+18] it was discussed, how the SMARDT process supports automated test case derivation from models such as Architecture Description Languages (ADs). Of particular interest for this work is the technical concept layer and its generative transformation into the fourth layer. We will dive into the model to code transformations, particularly focusing on multi-platform code generation and generator coupling targeting CPS applications and the integration of middleware adapters. As introduced in [HKK+18], models of the third and before last layer of SMARDT, i.e. the ones the SMARDT process uses before the final software generation can be developed using **MontiCAR**, a family of textual Domain Specific Modeling Languages (DSMLs) designed with regard to CPS and embedded systems domain [KRRvW17] using MontiCore, a language workbench facilitating the creation of language families by supporting multiple language composition techniques on grammar, Abstract Syntax Tree (AST) and Symbol Management Infrastructure (SMI) level[KRV10]. The core language of MontiCAR is *EmbeddedMontiArc* which is a MontiArc based C&C Architecture Description Language (ADL) providing an abstract math oriented type system with the primitive types `Z`, `Q`, and `C` denoting integers, rational and complex numbers, respectively. Furthermore, 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, e.g `Q(-2.7V : 100mV : +2.7V)` denotes a voltage variable able to take values from -2.7V to 2.7V in 0.1V steps. Since many engineering applications rely heavily on matrix calculus, a primitive type can easily be extended to a matrix by specifying its dimensions, e.g. `Q^{2,3}` denotes a 2×3 matrix of rational numbers. Further features of EmbeddedMontiArc comprise component and array port definitions allowing to instantiate and reconnect large amounts of similar components such as redundant sensors but also cooperative vehicle networks making the

Domain Specific Language (DSL) suitable for modeling of cooperative driving. The behavior of an EmbeddedMontiArc component can be described either by breaking it down into subcomponents or directly by using **MontiMath**, a tensor oriented DSL of the MontiCAR family inspired by MATLAB but enhanced by the previously described strong type systems, SI units, and a matrix property system. MontiCAR will serve as the main DSML family throughout this work. **Robot Operating System (ROS)** is a platform independent middleware designed with regard to robot software development. In particular, it enables asynchronous communication of distributed components acting as ROS nodes by publishing and subscribing so called ROS topics. Thereby, the format of the messages is defined at compile time and is basically an aggregation of C++-like primitive types as well as other ROS messages. OpenDaVinci is a similar but more light-weight middleware targeting the automotive domain[Ber15].

Chapter 3

Running Example and Problem Statement

The SMARDT process relies heavily on model based engineering using the models not just for consistency checking and verification but for model-to-model transformations and ~~also for~~ code generation. However, there is no such thing as a linear generation workflow. Different development tasks such as testing and validation require variants of models which are not needed at other process steps. This is similar to different compiler options in classical programming, where a *debug* target produces code which is instrumented for the debugger being much slower and therefore inappropriate for release. Although debugging is not an issue in Model Based Software Engineering (MBSE) since models are validated mainly using testing, there are parallels which we will explain in an intuitive running example. Consider a simple trajectory planning system modeled as a C&C architecture in EmbeddedMontiArc syntax in fig. 3.1. For simplicity of notation we keep the types abstract and leave out details irrelevant for the problem statement. The system receives the street map, its position, as well as some sensor measurements to detect obstacles as its inputs (lines 2-4) and is expected to output a trajectory plan (line 5). Following the divide and conquer principle we decompose our system into three subcomponents (lines 6-8): A `Filter` component will denoise sensor measurements to supply the `ObstacleDetection` with a smooth sensor signal allowing for a robust picture of the situation. The `Planner` component will then output a tentative trajectory based on the provided map and the detected obstacles. The data flow is defined by reconnecting the ports of the system in lines 9-12. Usually, CPS applications are distributed by nature. Different processes running on a distributed hardware architecture need to gather the required information from a heterogeneous sensor landscape, communicate computation results among each other, and send commands to the actuator system. In practice, such systems are implemented by means of middleware solutions like ROS [QGC+09] (robotics domain), AUTOSAR [AUT16a, AUT16b], and OpenDaVinci [Ber15] (automotive domain) in order to deal with the asynchronous data flows and to abstract away from the underlying hardware and network realization. However, before CPS software gets deployed in a real system, it will need to go through a lengthy testing and validation process. Since CPS highly depend on the the interaction with their environment as well as physical laws, acceptance testing will mostly involve simulations. Modern simulators like Gazebo [KH04, ÖRB+08] offer ROS interfaces through which sensor data and actuator commands can be exchanged. This not only enables a seamless controller integration into the simu-

```
component TrajectorySystem{
        direction   type      name
  ports in      Map     map,
        in Position pos,
        in SensorSignal senSig,
        out Trajectory desiredTraj;

    instances of subcomponents defined in other artifacts
  instance ObstacleDetection obsDetection;
  instance Filter filter;
  instance Planner planner;
         source port      target port
  connect map   ->  planner.mapIn;
  connect pos   ->  planner.posIn;
  connect planner.trajOut -> desiredTraj;
  connect obsDetection.obstacles -> planner.obsIn;
  /* other connections */}
```

Figure 3.1: TrajectorySystem in written form

lator but also allows facilitates testing and validating our CPS software as if it was indeed distributed. The aforementioned leads to the following problems. First, a lot of glue-code needs to be written in order to integrate all the components in the desired way. Second, this glue-code needs to be changed throughout the development process as different simulators will likely have different interfaces. Third, middleware software introduces a series of dependencies making leading to a tedious build process which needs to be adapted for each integration scheme variant.

The question arises how a multi-platform modeling approach preserving the structure of the core models as well as the strength and efficiency of the original generator while featuring a high degree of flexibility concerning the underlying middleware should look like. The modeling methodology should thereby fulfill the following requirements: **(R1) Middleware agnosticy (models):** Domain models must remain middleware-agnostic to ensure re-usability in different environments as well as independence of the technical realizations. **(R2) Middleware agnosticy (generator):** The C&C and behavior code generator must remain middleware-agnostic. **(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 must be used. **(R4) Semantics invariance:** The semantics of the generated code must remain untouched, i.e. preserving the scheduling and synchronization concepts. **(R5) Build infrastructure:** The generated target code must include a build configuration identifying and providing the required dependencies as well as linking the resulting artifacts of the project. **(R6) Middleware coupling:** The combination of different middlewares in one single model must be possible.

Chapter 4

Tag-Based Multi-Platform Code Generation

MontiCAR 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 SMI of the model, the EmbeddedMontiArc code generator produces plain C++ code for the architecture and component communication, i.e. no third-party libraries are involved. For the generation of atomic component behavior defined in MontiMath the Armadillo Basic Linear Algebra Subprograms (BLAS) library is used. Armadillo delivers a wide range of BLAS functions, high performance, and automated parallelization of expensive operations [SC16].

In contrast to other ADLs, MontiCAR refrains from the use of a runtime scheduler. Instead, the scheduling is based on the *sorted order* algorithm employed by 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. This approach is generally much faster than asynchronous code requiring synchronization and runtime scheduling. A detailed performance study is subject of ongoing work.

To tackle the requirements introduced in chapter 3 we are going to present a tagging-based approach for generator instrumentation. Tagging provides a non-invasive way to extend models with additional information such as extra-functional properties [MRRW16]. In this paper we will use tags to declare middleware specific information for the code generator, thereby preserving the agnosticy requirement **(R1)**, i.e. the C&C model itself does not contain any middleware-specific elements. By adding a middleware (e.g ROS) tag to a port, we specify that the message exchange with its counterpart should be realized using this middleware's communication pattern, e.g. the ROS publisher/subscriber pattern. Most of our requirements are easier to satisfy if 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 a design including coordination of the workflow and variation points for the generators used. A well-suited design pattern for this problem is the star-bridge which extends the abstraction of the standard bridge by arbitrary many variation points for the implementation. The resulting class diagram for our generator coupling mechanism is depicted in fig. 4.1. The `MainGenerator` is an abstraction of the generation workflow ignorant of how to generate the actual code. 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 `MainCoordinator` manages the set of needed code generators, the implementation parts of the bridge. Note that the list of concrete generators implementing `GeneratorImpl` includes both the EmbeddedMontiArc generator to generate the actual CPS logic as well as all the middleware generators. All these generators are unaware of each other's existence producing completely independent artifacts controlled only by the coordinating generator. The generator interface expects an `Expanded Component Instance Symbol` object as input, a data structure provided by the MontiCore SMI representing the instance of a component to be generated including references to its subcomponent instances. Based on the information in the symbol, architecture, behavior and middleware code is produced by the respective generators. Each middleware generator outputs a middleware adapter for the ports specified in the tag model implementing the `IAdapter` interface. This interface contains an `init(...)` and a `publish()` method both to be used by the coordinator. 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 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 EmbeddedMontiArc component. By design, there is no synchronization between the system execution and middleware communication, i.e. a component is executed by the coordinator periodically *without* waiting for new inputs to arrive. If the middleware has not provided any new data until the beginning of the execution cycle, the component would see the old data at the respective input ports. Note, that even if no single port has been updated, the component needs to be re-computed as it might not only depend on the actual input but on the whole input history. This is for example the case for a PID controller due to the integration. Hence, leaving the original MontiCAR to C++ generator untouched by the middleware generation process ensures that the previously discussed efficient synchronous execution model of EmbeddedMontiArc is not influenced by the middleware used, while separating the logic generation concern from the generation of the technical communication infrastructure.

Now let's go back to our running example presented in fig. 3.1. The tag model for the trajectory planner is given in fig. 4.4. To ensure that the tag model is a valid model itself, it is first checked against the corresponding *tag scheme*. The scheme we use here allows 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. However, combinations of different middlewares inside one model and even for the same port are possible, as well. The middleware-specific message description such as a ROS message type is generated automatically based on the port type information, e.g. for the ports `mapIn` and `mapOut` in lines 4 and 5. This automation however is not applicable if the generated code needs to be compliant with third-party systems, e.g. a simulator such as Gazebo having predefined message types and topic names for its sensors and actuators. In such a case the user can instruct the generator to use a predefined type and topic name which mostly makes sense for system boundaries, e.g. lines 2 and 3 in our example.

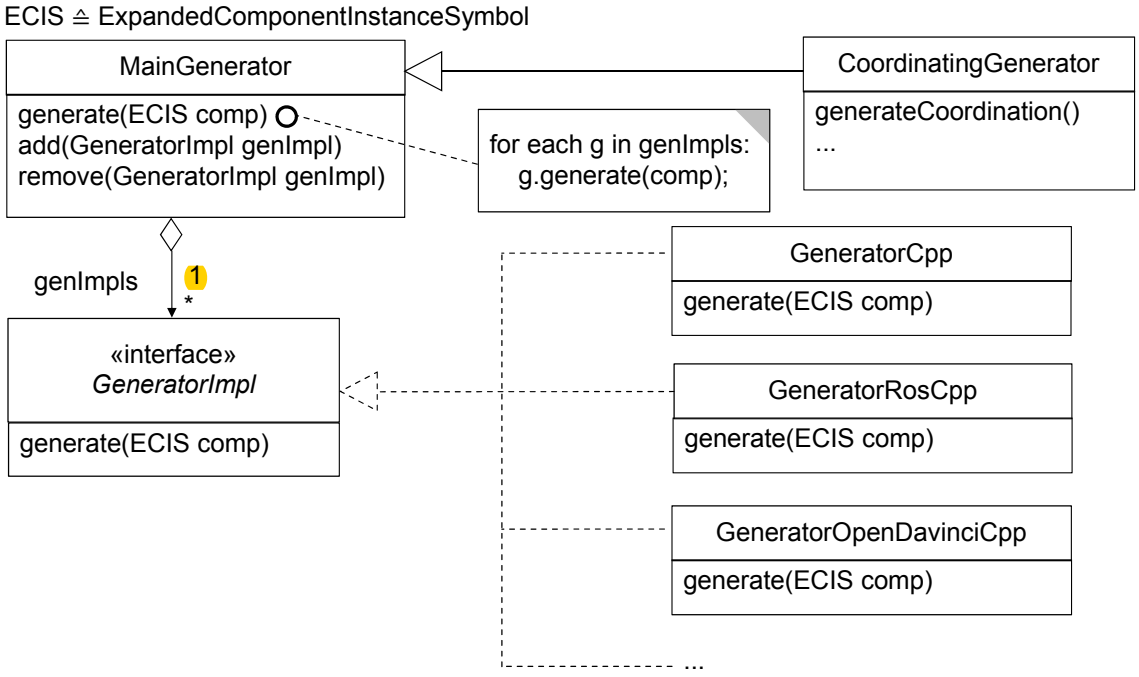


Figure 4.1: Overview of the generator architecture

Note that the underspecified `mapIn` port receives its data directly from the `map` 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. 4.5. Finally, the needed ROS information gets loaded via tagging into the respective middleware symbol, e.g. a `RosConnectionSymbol` of the corresponding port which is either empty or consists of a `topicName`, a `topicType`, and an optional `msgField`. Now that the MontiCore SMI has been enriched by all the necessary middleware meta-data as depicted in fig. 4.7, we use so called context conditions on it 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 generated projects and their respective artifacts is provided in fig. 4.3 (according to the tag file, no OpenDaVinci adapter needs to be generated here; the correpsponding project is listed anyway to underline the multi-target capability of the approach). The generated project hierarchy comes with all the build files required to compile and link the projects. The resulting class diagram exhibiting the coupling between the coordinator, the adapters, and the actual application logic is depicted in fig. 4.3. Here, it is illustrated once again, how the system is governed by the `TrajectorySystemCoordinator`. In the initialization phase the coordinator creates an instance of the `TrajectorySystem` and hands over its pointer to the adapter objects. In the operation phase, the coordinator triggers the `execute()` method of the the `TrajectorySystem` and the `publish` methods of the adapters within a predefined frequency. The concrete generated adapter code for this running example is depicted in fig. 4.6.

Now consider the components `ObstacleDetection` and `Planner` in our `TrajectorySystem` 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

Generator	Generated files
CoordinatingGenerator	├── CMakeLists.txt ├── coordinator │ ├── CMakeLists.txt │ ├── TrajectorySystemCoordinator.cpp │ └── IAdapter.h
GeneratorCpp	├── cpp │ ├── CMakeLists.txt │ └── TrajectorySystem.h
GeneratorOpenDavinciCpp	├── opendavinci │ ├── CMakeLists.txt │ └── TrajectorySystem │ OpenDavinciAdapter.h
GeneratorRosCpp	└── roscpp ├── CMakeLists.txt └── TrajectorySystemAdapter.h

Figure 4.2: Overview of the generated Project

simple C++ calls as produced by the EmbeddedMontiArc generator. Obviously, the intention here is to model a *distributed* system architecture and to deploy it on different control units of the CPS. To do so we imagine 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. Technically, this can be accomplished by constructing a binary and undirected, i.e. symmetric adjacency matrix of the graph and solving the eigenvalue problem. The resulting eigenvectors can then be used as cluster indicator vectors [VL07]. Thereby, two nodes belong to the same cluster if there exists at least one eigenvector such that the two vector dimensions corresponding to the two nodes are non-zero. The obtained partitions contain components which need to be generated as plain C++ code. The generator ensures that there is no middleware communication inside a partition, i.e. middleware tags of ports connecting two components within the same partition are ignored and the user receives a corresponding warning. Next, for each partition an imaginary parent component is created unless it has only one top-level component and the generation process is run independently for each partition. Based on the tagging it can be identified for the running example that `ObstacleDetection` and `Filter` communicate directly and, hence, belong to the same partition while the `Planner` can be decoupled. The new system boundaries are illustrated in fig. 4.7 by the dashed boxes.

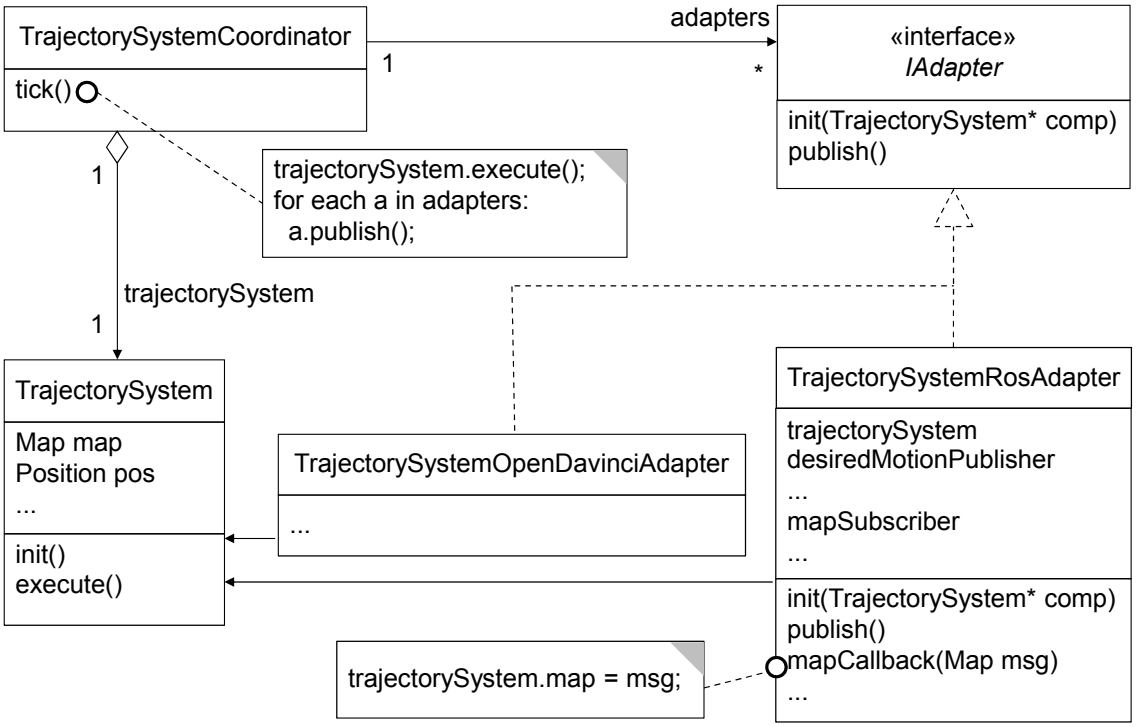


Figure 4.3: Overview of the generated c++ code

```
tags RosTags{
tag trajectorySystem.map with RosConnection =
  {topic= (name=/map, type=struct_msgs/Map)};

tag trajectorySystem.desiredTraj with RosConnection =
  {topic= (name=/desiredMotion, type=struct_msgs/Trajectory)};

tag trajectorySystem.planner.mapIn with RosConnection;
tag trajectorySystem.planner.obsIn with RosConnection;
[...]}
```

tagged symbol; tag type; tag value; RosConnection with complete information(■); RosConnection with incomplete information(▨)

```
struct Position{
  Q      ( -90° : 0.000001° : 90° ) longitude;
  Q      ( -180° : 0.000001° : 180°) latitude;
  Q      ( -10km : 10cm : 10km) altitude;}
```

type; min value; step size; max value; name

Figure 4.4: Tag model for the running example

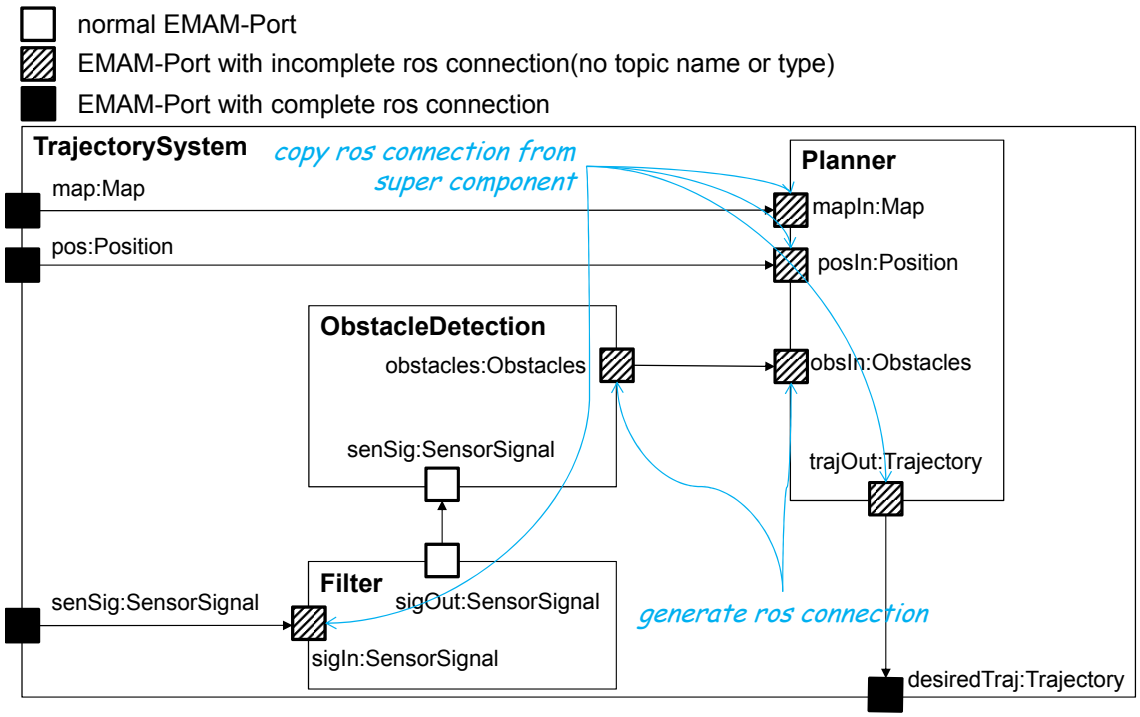


Figure 4.5: Propagation of ros connections

```
1  #include "TrajectorSystem.h"
2  [...] /* other includes*/
3  class TrajectorySystemRosAdapter : public IAdapter{
4    TrajectorySystem* trajectorySystem;
5    ros::Subscriber mapSubscriber;
6    ros::Publisher desiredMotionPublisher;
7
8    public:
9    void init(TrajectorySystem* comp){
10     trajectorySystem = comp;
11     /*init publishers, subscribers and start ROS thread*/
12     [...]
13   }
14   void publish(){
15     struct_msgs::Map tmpMsg =
               msgFromStructMap(trajectorySystem->desiredTraj);
16     desiredMotionPublisher.publish(tmpMsg);
17   }
18
19   void mapCallback(struct_msgs::Map& msg){
20     trajectorySystem->map = structFromMsgMap(msg);
21   } [...] };
```

instance shared between the coordinator and other adapters

called once by TrajectorySystemCoordinator

called in a defined interval by TrajectorySystemCoordinator

called by ROS every time a message is published on the topic /map

Figure 4.6: Generated RosAdapter for the TrajectorySystem

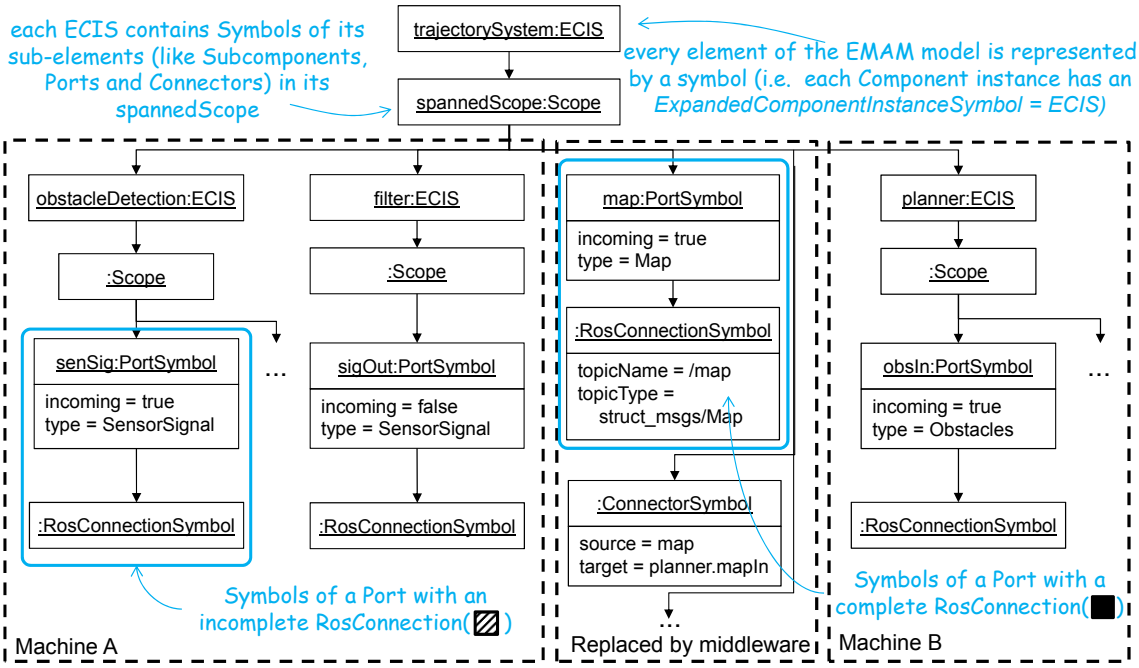


Figure 4.7: Clustering of the TrajectorySystem shown in the Symbol table

Chapter 5

Evaluation

We applied the presented generation approach in several case studies related to autonomous driving using a series of simulators such as Gazebo, SUMO in combination with Veins for cooperative driving [KEBB12, SGD11], TORCS [WEG+14], and MontiSim [GKR+17]. Consider the lane following C&C model depicted in fig. 5.1 which serves as the core component of a self-driving system. It is decomposed into the three sub-components `TrajectoryPlanner`, an `ObstacleDetector`, and a `LaneController` executing the planned trajectory. The latter is a classical PID based design controlling the velocity, distance from desired trajectory, as well as the orientation of the vehicle. To test and validate our system, we decide to use the widely used Gazebo simulator featuring realistic environment conditions, a physics engine, sensor models, and a 3D visualization [ÖRB+08]. In a series of simulations we want run a series of test cases, e.g. **TC1**: verifying that our vehicle controller does not leave the street. As our experimental vehicle we use an RBCar with Ackerman kinematics and ROS interfaces for control. Furthermore, we equip the vehicle with three laser sensors of the type Hokuyo UTM-30L for obstacle detection. An experimental scenario using an elliptic track is depicted in fig. 5.2 (a). The first part of the track contains a simple obstacle-free turn to test the basic lane following functionality. In particular, this part can be used for parameter tuning, e.g. using a genetic algorithm as described in [HKK+18]. Thereafter, we have parking cars forcing our test vehicle to slightly change its trajectory towards the middle of the road. The following construction site with an SUV requires a more difficult maneuver. Finally, the car needs to detect the wall blocking the whole street and to use its emergency brake. To integrate our self-driving system with Gazebo's sensor and actuator system we enrich our model with ROS tags. As Gazebo prescribes the topics and data types we adopt this information in the port tags instead of generating new message types for the respective ports. We do not have to write any glue code, the whole system including the ROS code and build files is generated automatically while the driving model itself does not contain any middleware related information and could be re-used in other contexts. The scenario is captured in https://vimeo.com/246709003 and the driven trajectory is depicted in fig. 5.2 (b). **TC1** fails as our vehicle leaves the road in order to pass the parked cars. This cannot be found out without a simulator, e.g. using unit tests.

We observed that using MontiCAR to describe ROS based communication we eliminated several error sources and design pitfalls which we used to encounter in the past. Since the generated architecture is based on MontiCAR semantics it forbids the following communication anti-patterns which are possible when writing ROS publishers and subscribers in

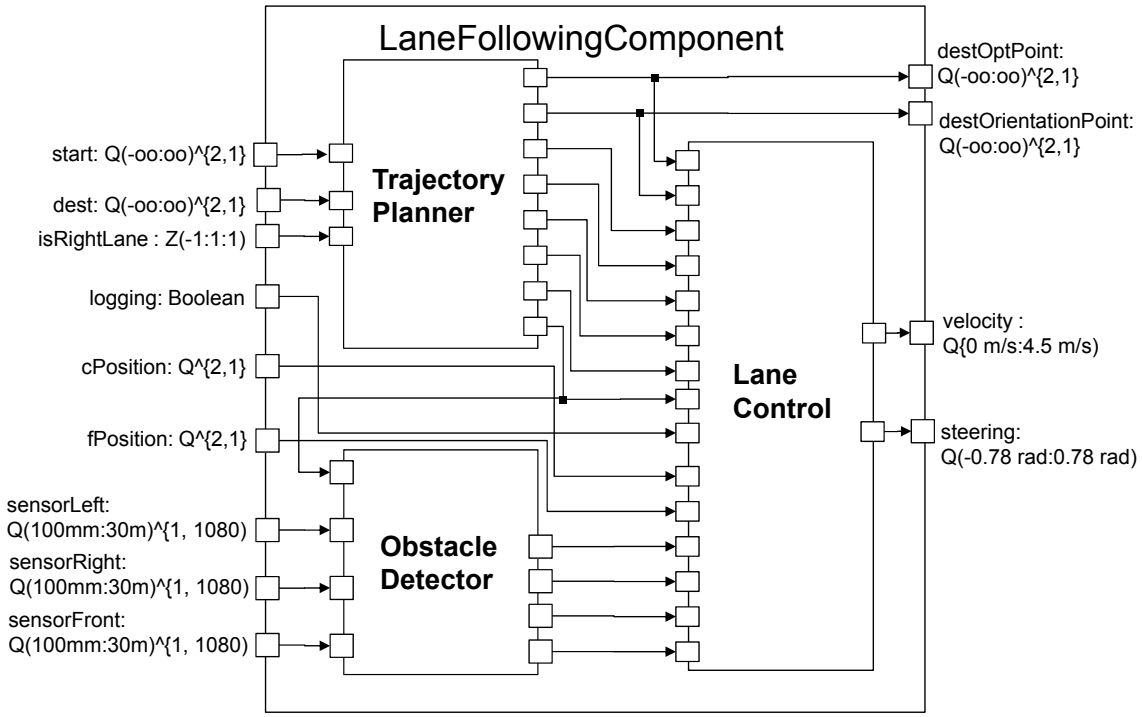
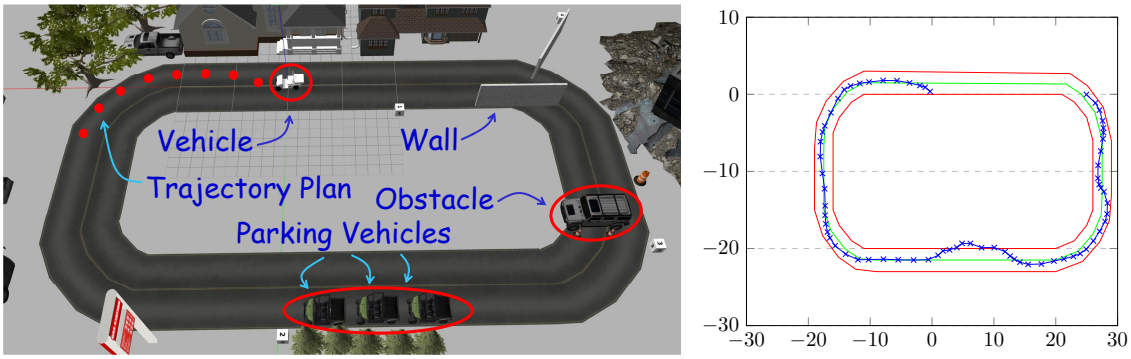


Figure 5.1: Lane Component



(a) Gazebo Test Track

(b) Driven trajectory (blue)

Figure 5.2: Track and driven trajectory

C++ or Python: **Type compatibility**: in contrast to hand-written ROS code the type compatibility of a publisher and a subscriber is checked at compile time thereby massively reducing the probability of uncontrolled system crashes and undesired behavior. **Naming errors**: it is not possible to publish to a topic nobody listens to and vice versa as a MontiCAR model explicitly specifies a communication scheme by its connectors. Hence, misspelled topic names are identified at compile time. **Sender ambiguities**: since MontiCAR allows only one incoming connector per input port, it is not possible to let two publishers publish to the same topic. Thus, all data flows are guaranteed to be bilateral. We observed that this eliminates the temptation to develop workarounds enabling the receiver to identify the publisher, e.g. by adding sender information to a message definition. 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 while reducing potential error sources.

Chapter 6

Related Work

In Simulink [Mat16] distributed communication is modeled using predefined components ~~and~~ ROS is no exception. The Robotics System Toolbox provides the designer with the following three components: The *Blank Message* to create empty messages which can be changed using bus assignments, the *Publish* component to publish data to a specific topic, as well as a *Subscribe* component used to receive data from another ROS publisher. The user needs to use Simulink's variability modeling functionality in order to keep track of different model variants needed throughout the development cycles. However, the SMARDT methodology strictly separates the technical concept layer from the implementation layer. Thus, a technical concept model should not be pervaded by middleware-specific aspects in a SMARDT context. A generative approach using *MontiArcAutomaton* keeping the model free of middleware aspects was presented in [AHRW17]. MontiArcAutomaton is a textual ADL with Focus based semantics [BS12]. As in this work the whole architecture is considered to be distributed no generator instrumentation approach was considered and, hence, the developer has no control over the way how two components are coupled. Furthermore, the integration of different middlewares was not analyzed. RobotML is an Eclipse Modeling Framework (EMF) based graphical modeling framework for the robotics domain providing code generation for a variety of robotics platforms [DKS+12]. Code generator composition however is not treated.

The integration of modeling languages and simulators not always has the goal to provide means of describing the robot behavior but also its physics. In [BBCM17] the authors propose a framework to extend Gazebo by multi-body dynamics in Modelica, a language for physical system modeling [A+05, EMO99]. However, the communication between Gazebo and Modelica is managed using IPC sockets.

Chapter 7

Conclusion and Future Work

We presented a code generation approach in the context of the SMARDT process fulfilling a set of previously declared requirements. Thereby, wecover three important kinds of artifacts: the actual application code, middleware adapters, as well as build scripts needed to assemble an executable solution. The tool-set enables the generation of loosely coupled distributed architectures as well as a seamless integration into existing simulators straight out of the C&C models of SMARDT layer 3 without touching the application logic artifacts.

The main DSL for behavior specification in MontiCAR models is MontiMath. Although MontiMath provides meaningful abstractions for the domain of mathematical modeling, it is still too low level to describe complex work-flows or human computer interactions. Therefore, it is likely that with increasing demand for robot applications and new emerging technologies like artificial intelligence and deep learning the integration of alternative behavior description languages will become indispensable. Therefore, a framework for code generator composition automatically choosing the right behavior generator and also capable to produce glue code for the heterogeneously generated artifacts will be necessary.

Bibliography

[A+05] Modelica Association et al. Modelica language specification. *Linköping, Sweden*, 2005.

[AHRW17] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Modeling Robotics Software Architectures with Modular Model Transformations. *Journal of Software Engineering for Robotics (JOSER)*, 8(1):3–16, 2017.

[AUT16a] AUTOSAR. Layered Software Architecture. Technical Report 053 (4.3.0), AUTOSAR, 2016.

[AUT16b] AUTOSAR. Modeling Guidelines of Basic Software EA UML Model. Technical Report 117 (4.3.0), 2016.

[BBCM17] Gianluca Bardaro, Luca Bascetta, Francesco Casella, and Matteo Matteucci. Using modelica for advanced multi-body modelling in 3d graphical robotic simulators. In *Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017*, number 132, pages 887–894. Linköping University Electronic Press, 2017.

[Ber15] Christian Berger. OpenDaVinci Middleware, 2015.

[BS12] Manfred Broy and Ketil Stølen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement.* Springer Science & Business Media, 2012.

[DKS+12] Saadia Dhouib, Selma Kchir, Serge Stinckwich, Tewfik Ziadi, and Mikal Ziane. Robotml, a domain-specific language to design, simulate and deploy robotic applications. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 149–160. Springer, 2012.

[EMO99] Hilding Elmqvist, Sven Erik Mattsson, and Martin Otter. Modelica-a language for physical system modeling, visualization and interaction. In *Computer Aided Control System Design, 1999. Proceedings of the 1999 IEEE International Symposium on*, pages 630–639. IEEE, 1999.

[GKR+17] 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.

[HKK+18] Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Mike Lorang, Bernhard Rumpe, Albi Sema, Georg Strobl, and Michael von Wenckstern. 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)*, pages 163 – 178. SciTePress, January 2018.

[KEBB12] 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), 2012.

[KH04] 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.

[KKRvW18] Stefan Kriebel, Evgeny Kusmenko, Bernhard Rumpe, and Michael von Wenckstern. Finding Inconsistencies in Design Models and Requirements by Applying the SMARDT Process. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme XIV (MBEES'18)*, Univ. Hamburg, April 2018.

[KRRvW17] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *ECMFA*, 2017.

[KRV10] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: a framework for compositional development of domain specific languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5), 2010.

[Mat16] Mathworks Inc. Simulink User's Guide. Technical Report R2016b, MATLAB & SIMULINK, 2016.

[MRRW16] Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. Consistent Extra-Functional Properties Tagging for Component and Connector Models. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'16)*, volume 1723 of *CEUR Workshop Proceedings*, pages 19–24, October 2016.

[ÖRB+08] Ümit Özgüner, Keith Redmill, Scott Biddlestone, Ming Feng Hsieh, Ahmet Yazici, and Toth Charles. Simulation and testing environments for the darpa urban challenge. *IEEE International Conference on Vehicular Electronics and Safety*, 2008.

[QGC+09] 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.

[SC16] Conrad Sanderson and Ryan Curtin. Armadillo: a template-based c++ library for linear algebra. *Journal of Open Source Software*, 2016.

[SGD11] 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, 2011.

[VL07] Ulrike Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.

[WEG+14] Bernhard Wymann, Eric Espié, Christophe Guionneau, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. TORCS, The Open Racing Car Simulator. http://www.torcs.org, 2014.