Teaching Playground for C&C Language EmbeddedMontiArc

Evgeny Kusmenko, Bernhard Rumpe, Ievgen Strepkov, Michael von Wenckstern
Software Engineering RWTH Aachen University
http://www.se-rwth.de

ABSTRACT

The development of self-driving vehicles is one of the most innovative research domains these days. To inspire students to get involved in this technology as well as model-based design, particularly using the Component and Connector (C&C) paradigm, we created a web-playground allowing us to model controllers for a simulator and almost instantly see the results in a 3D environment. We believe that visualization and gamification motivate students and make the studying process more attractive. This paper uses EmbeddedMontiArc to implement C&C architectures; however, the presented approach can be easily adopted to any other C&C based language and/or tooling.

CCS CONCEPTS

• **Computer systems organization** → **Embedded software**; • **Software and its engineering** → *Data flow architectures*; • **Computing methodologies** → Online learning settings;

KEYWORDS

C&C, Component and Connector, Tutorial, Playground

Reference Format:
Evgeny Kusmenko, Bernhard Rumpe, Ievgen Strepkov, Michael von Wenckstern. Teaching Playground for C&C Language EmbeddedMontiArc. In *Proceedings of ACM/IEEE Models Conference Workshops (ModComp'18).* 6 pages.

1 INTRODUCTION

Self-driving vehicles are a very important part of our future and thus an important research area. To inspire students to be involved in this technology we created a web-playground which allows us to model controllers for a simulator and almost instantly see the results in a 3D environment. We believe that visualization will motivate students and make the studying process more attractive using gamification [12].

To create an outstanding tutorial platform for C&C models, we analyzed existing playgrounds and tutorials (also including programming languages) in order to understand which tools can be reused and how we should present the knowledge to students so that it is easy to understand and well-structured.

C&C models are often used to describe software functionalities in embedded domains such as avionics [8], robotics [19], and automotive [5]. One of the main advantages of the C&C paradigm is that it leads to a hierarchical decomposition of the system. The obtained C&C models consist of components at different levels, directed and typed ports, and connectors between them. On the other hand, hidden communication and side-effects are strictly forbidden. Encapsulation and logical decomposition allow efficient development, modular reuse, and evolution.

EmbeddedMontiArc[14, 15] is a textual domain-specific language for the modeling of logical functions based on the C&C paradigm. Thus, its main elements are components and connectors. Each component has ports that can be either incoming or outgoing. Through these ports, components are interconnected. Each port has a unit and a range. Units are an inherent part of signal types and preventing re-connections of physically incompatible signals like *km/h* and *kg*. The given range of the port type specifies that an incoming signal must be within specified boundaries enabling one to model limits of non-ideal systems, e.g. to account for the maximal velocity of a vehicle. Furthermore, the EmbeddedMontiArc language supports generics. It allows creating generic components for multiple purposes.

A powerful standalone simulator for EmbeddedMontiArc models called MontiSim already exists [9, 11]. It has a built-in physics engine, creates its road network from OpenStreetMap data and even provides weather effects. However, due to its multitude of features it rather aims at expert users than newcomers. The set up procedure is too lengthy to be performed during a short tutorial and requires some configuration.

Code sharing between a standalone application and a separate tutorial is often not convenient. Furthermore, at the moment, there is no tutorial for the existing simulator which can teach the C&C paradigm by using EmbeddedMontiArc language step-by-step. Because of the explained reasons, we decided to develop a lightweight solution taking fewer details into account during the simulation. While it can manage all basic actions and use a full version of the existing EmbeddedMontiArc language family, it does not provide an elaborate physics engine, co-simulation, and communication protocol stacks.

We believe, that the community can benefit from this paper. We provide two tutorials for students and solutions for C&C models. Due to our modular architecture, the tutorials and the domain solutions are nearly language independent. Therefore the models and solutions provided in our paper can be used for other C&C languages such as Modelica [2], Simulink [16], LabView [13], or AutoFocus3 [1]. The light-weight 3D simulator which works entirely in a browser can be used for other projects to visualize the various scenarios as in the paper *Specifying Intra-Component Dependencies for Synthesizing Component Behaviors* [7] where an example of a car overtaking is shown.

The outline of this paper is the following: section 2 presents the parking and elk test tutorials as running examples for this paper; section 3 summarizes current related work on tutorials and playgrounds for other languages; section 4 shows the technical aspects of the online teaching playground for EmbeddedMontiArc; section 5 illustrates how students or other persons being interested

ModComp'18, October 14th – 16th 2018, Copenhagen, Denmark

in C&C modeling can use this playground - it shows the users' viewpoint; and finally section 6 concludes this paper.

2 RUNNING EXAMPLE

The aim of our tutorial is to create a controller for a self-driving car. But the task is too complex to be tackled in one shot. We have borrowed the idea from the agile development manifest [3] to divide the large task into small ones.

The small tutorials are based on user-stories which allow to get faster feedback for students. Implementing simple tutorials and learning the basics of the language, students gain experience in controller design for autonomous vehicles. It is extremely hard to solve the big and complex task in one step. Therefore, sub-dividing the task into smaller pieces is important. The latter can be solved in shorter amount of time and, hence, lead to prompt feelings of success.

The running example presents the process of creating controllers that solve given small but common tasks. To achieve the goal we have to understand the key aspects of the tasks. It helps to find out which components are needed. The first task is to carry out parallel parking between two cars. The second task is to successfully pass the elk test.

Parking Tutorial. Parallel parking is a well-known every days scenario for students having a car. But it is not so easy to perform this maneuver for an inexperienced driver. Therefore, a standard solution is proposed by German driving schools to accomplish this task (see Figure 1).

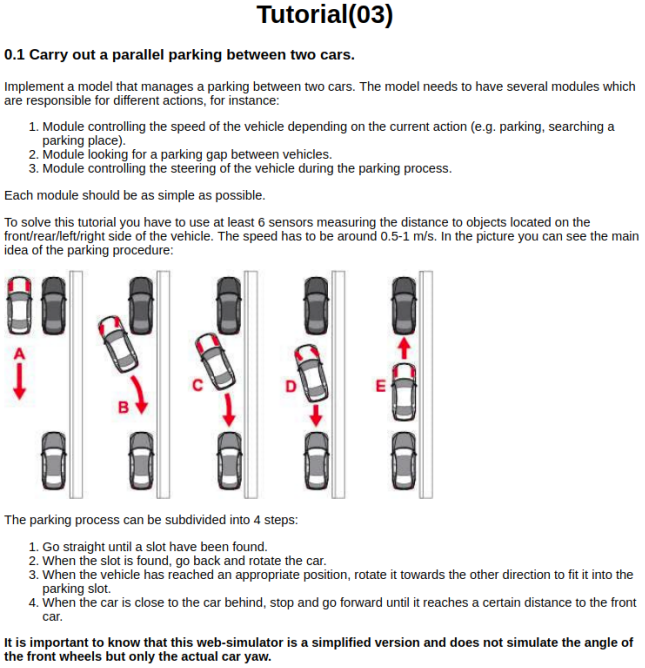

Figure 1: Screenshot of the parking tutorial

The most interesting point in this example for C&C developers is that this maneuver consists of several independent steps. This enables us to delegate different tasks to different components such that each component only solves one simple task. To manage the maneuver we have to perform the following steps: find a parking spot; go back pulling the car over in the parking space; and drive forward aligning the car with the curb.

During the parking process the given requirements must be met: **(P1)** the car must not get into an accident during parking; **(P2)** at the end of the parking process, the car is parallel to the curb; **(P3)** distances to the front and to the rear car should be about the same; and **(P4)** the car should only perform three steps to park (as shown in Figure 1).

Elk Test Tutorial. Another test that we want to introduce is the elk test. It is performed to determine how well a certain vehicle evades a suddenly appearing obstacle. These days, the test is performed by major automotive OEMs, because it proves the ability of a car to maneuver on average speed (60 km/h) without losing control. In our context, this test gives an idea how the car responds to objects on the track and how maneuverable it is depending on its current speed.

To pass the elk test the following requirements must be met: **(E1)** the car does not drive into cones during the test; **(E2)** it drives on the shortest path, being as close as possible to the cones; **(E3)** the car does not leave the testing area by violating track boundaries; **(E4)** the start and end positions are specified; and **(E5)** the car has to drive between each pair of cones (as shown in Figure 3).

Tutorial Set-up. Each tutorial is organized as a module, it has all required elements inside its package. It comprises several files including a **description of the tutorial**: this is the text which students are given to understand the task and to find some useful hints and recommendations for the implementation; a **solution description** providing a detailed specification on how to solve the current task and displaying a working code solution which can be used directly; a **sample controller**: a controller which has been implemented in advance and is able to pass the current test; an **environment configuration**: a file, which contains all important objects and their positions; a **reference trajectory** used to check the students' solutions.

To prepare the simulator environment for tutorials, a configuration file is used. The configuration defines an initial position of the car. For each tutorial the position specified depends on the task and an area on the track where an action is going to happen. The configuration also defines the positions of further objects relevant for the scenario. It is very convenient to have configurations for different tutorials, because it provides flexibility during the tutorial preparation process and simplifies the creation of new tutorials.

3 EXISTING SOLUTIONS WITH TUTORIAL CHARACTER

This section presents how tutorials are created by other tool vendors for their languages. We have taken in consideration different tutorials from different areas.

Simulink. Simulink [16] is a C&C modeling environment for simulation and model-based design of various domains. There are

plenty of tutorials from many different areas with detailed descriptions and videos explaining the solution step-by-step. But the problem here is, that they don't have methods for validating the correctness of the user's solution and do not encourage the users to try it out by themselves instead of just copying the sample solution. It is however obvious that the learning effect rises dramatically when students have to find the solution on their own.

Rust. Rust [17] is a very popular programming language the prevalence of which is growing every day. It has a consistent tutorial which describes language constructs with gradually increasing complexity. It has an informative and structured index, where users can easily jump from one topic to another almost instantly and then just go back to the place they were reading before. They use highlighted areas to show code examples, which facilitate understanding of the presented material.

Microsoft Z3 Solver. Z3 [6] is a state-of-the art theorem prover from Microsoft. They provide an experience similar to the Rust tutorial. Additionally, they provide the possibility to execute the tutorial code directly in the browser accelerating the learning process. It is helpful to see the direct connection between written commands and the real result and improves the understanding of given material.

Octave Online. Octave Online is a web-playground for the high-level language Octave [18] primarily intended for numerical computations. It has a simple and intuitive interface despite the complexity of the internal implementation. It provides fast execution and error handling directly in the browser. Even if you do complex computations it is not needed to install any software on the PC. Everything works on-line out of the box.

Wolfram Alpha. Wolfram Alpha [20] is a very powerful tool, which works by using expert-level knowledge and algorithms to automatically answer questions, do analysis and generate reports. It supports matrix operations and calculations to describe atomic components, similarly to the EmbeddedMontiArcMath language. Beyond solving linear equations, it allows to specify the behavior of controllers. Then by solving the equations a controller can be synthesized. Furthermore, it has one very interesting and useful feature providing an interactive visualization of given data. The idea behind is that you can "feel" how certain parameters influence the final result. It promises a better understanding of the dependencies between the components or elements of the system.

TypeScript Playground. TypeScript [4] is a typed superset of JavaScript. It has a clean and simple playground which shows the difference and benefits of TypeScript over JavaScript. It has preloaded examples showing the actual difference and thus provides a direct comparison giving a better understanding of the language and facilitating further analysis.

Swift Playgrounds. Swift [10] Playgrounds has been created for teaching the Swift language in a hands-on experience. You can create small programs that instantly show the results of your code. In the right part of the screen a 3D world is shown. The tutorials are pretty simple but the concept is very interesting. They have automatic verification of the correctness of an implemented solution in the 3D environment. To produce many diverse game oriented tutorials, it would be convenient to have a simple 3D model import for various 3D formats designed in different tools.

Summary. Having thoroughly analyzed the solutions listed above, we have derived the following list of requirements for our tutorial: **(R1)** an appropriate visualization for demonstration purposes; **(R2)** Simple, clean and intuitive interfaces; **(R3)** Support for all common operating systems and no need for an installation; **(R4)** Automatic verification of obtained results; **(R5)** Import and reuse of existing 3D models for the visualization; **(R6)** Displaying the object's trajectory; **(R7)** Integrated unit testing support.

Table 1: The table summarizes the comparison between all considered tutorials. (+ supported, P partially supported, - not supported)

	Z3	Octave	Wolfram	TypeScript	Swift	Rust	Simulink	EMAM
R1	-	+	+	-	+	-	+	+
R2	+	+	+	+	+	+	+	+
R3	+	+	+	+	-	+	-	+
R4	-	-	-	-	+	-	+	+
R5	-	-	-	-	-	-	+	+
R6	-	-	P	-	P	-	P	+
R7	-	-	-	-	-	-	+	+

Table 1 summarizes the differences between the tutorials regarding the derived requirements.

(R1) Appropriate visualization for demonstration purposes: Four considered tutorials have a 3D visualization. Octave online has a possibility to generate plots and graphs for given data. Wolfram Alpha has a powerful tool for the generation of 3D models. The student can interact with these models and see the changes in real time. The Swift Playground has the most advanced 3D world which is part of the tutorial and result presentation. Simulink provides means to build appealing 3D models which can be involved in the simulation process. However, the users have to build everything themselves.

(R2) Simple, clean and intuitive interface: This is the only requirement which all tutorials were able to satisfy. We believe that it is very important to have an understandable and clear interface which does not distract from the educational process.

(R3) Work on any operating system and without installation: Almost all examined tutorials have a web-implementation and work without installation, except the Swift tutorial and Simulink. The Swift tutorial has only an iOS realization. Simulink does not provide an on-line interface, however Matlab has a partial web-implementation.

(R4) Automatic verification of obtained results: Only one among the examined tutorials, the Swift tutorial, has an automatic verification of the solution provided by the student. It generates additional interest during the studying process, and can be a motivation to keep solving the tasks by analogy with computer games. In Simulink such a feature can at least be implemented by the user but is not provided out-of-the-box.

(R5) Import and reuse of existing 3D models for the simulation: Only Simulink provides the ability to import 3D models.

It helps to create tutorials quickly and efficiently by using the previously created models and configurations. An example of reusing a 3D object can be a cone that is used in many exercises. This feature, in our opinion, simplifies the process of creating new tutorials and decreases the time which has to be invested in the creation process.

(R6) Displaying the object's trajectory: Wolfram Alpha, Swift, and Simulink have partial support for this feature in case that you can see the whole process of movement of the object from the very beginning to the end. But we decided to improve the concept and to add a separate window which permanently displays the traversed route of the object for better visual perception and visual comparison of results. In our case the object is a car.

(R7) Integrated testing support: Only Simulink has integrated testing capabilities. In the domain of our tutorial, however, testing plays an important role. By writing blackbox unit tests for a component (called stream tests in EmbeddedMontiArc) we can ensure correct handling of incoming data. Tests make the components more reliable and robust. What is more, we think that today's tutorials should teach students to develop tests before starting to work on the implementation. Taking into account all these derived requirements we are going to present our solution.

We want to emphasize that our platform is not the best although it outperforms the presented tools regarding the proposed requirements according to Table 1. The other tutorial platforms, especially Simulink and Swift, support a much broader domain for tutorials; Simulink for example provides excellent tutorials including high-quality videos and *Matlab on Ramp* for image processing, deep learning, aerospace, and an electrical powertrain.

The main purpose of Simulink is to provide templates showing how a problem can be solved using this tool; our main purpose is to teach the students the C&C modeling paradigm including testing. We chose the domain of self-driving cars for our tutorials and therefore developed a simple TypeScript based simulator. Note that we only covered a small selection of available tutorials.

4 ARCHITECTURE

This section presents the software architecture referring to the high-level structure of our online web tutorial; it also discusses some decisions we needed to revert due to technical problems.

At the very beginning, we came up with the idea of a serverless application which can be hosted on GitHub pages. First, the solution looked very promising, as there already is a Clang In Browser (CIB) project to compile the generated C++ code to WebAssembly and to run it directly in a browser. But then we figured out some issues with this approach: CIB does not support dynamic linking yet. Therefore, we cannot link against the large Armadillo mathematics library, which is used by EmbeddedMontiArc. Static linking on the other hand would extend the linking process to several minutes.

Due to this reason, we decided to use a stateless server application: a user sends the C&C files to the server which translates them to WebAssembly [15]. Then the client receives the WebAssembly controller which is used by the web simulator. The huge advantage here is that the server is not involved in the simulation process. Thus, it is much easier to handle multiple users and it does not run into critical performance issues, due to multiple requests. The browser is responsible for each user and, in our opinion, has a massive impact on the user experience. Also, the maintenance of a stateless server is much simpler than the maintenance of a stateful one.

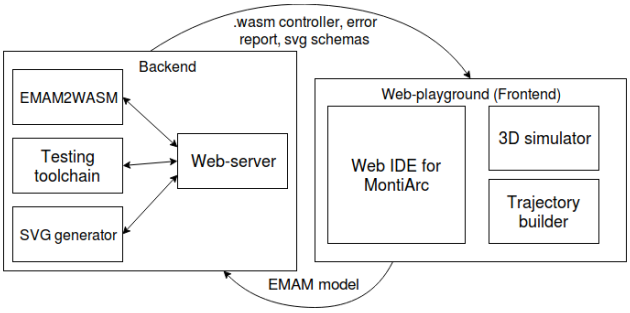

Figure 2: Architecture of C&C online tutorial platform

The disadvantage is that users have to wait for the compilation on the server side (compilation for a single tutorial for one user needs about 15s) and it can take longer due to multiple requests. Furthermore, compilation is not possible without a connection to the server. Nevertheless, the user can observe a fluent visualization showing the car driving in the simulator.

In our case, EmbeddedMontiArc is already developed as a self-contained service based on jar archives. Due to the derived requirement *(R3)*, we have to develop a web-based application. Therefore, on the server-side some services from EmbeddedMontiArcStudio can be selected and integrated.

But still, one server-side component is missing. The server must handle multiple users at the same time as well as the messaging from the back-end to the front-end. Students should get a response from the server to receive compiled controllers or just fix errors which can appear during the compilation process. EmbeddedMontiArcStudio, the offline development IDE of EmbeddedMontiArc, has its own 3D simulator having a lot of powerful features [9, 11]. But it requires a powerful computer and a more rigorous configuration thereby contradicting some of our requirements. So we decided to develop a new simple simulator satisfying our requirements. For implementation, we have picked TypeScript.

Working on the front-end, the simulator delivers a smooth and fluent experience for the user. Therefore we are using fully independent front-end for the simulator and back-end for the preparation phase of the controller. We went even further and decided to use WebAssembly for the controller. WebAssembly is a new type of code that runs in modern web browsers; it is a low-level assembly-like language with a compact binary format that runs with near-native performance. WebAssembly compilers are available for different source languages such as C/C++ facilitating the usage of such languages in web applications. WebAssembly is well suited for our task due to the fact, that EmbeddedMontiArc comes with a C++ code generator, i.e. a generator taking EmbeddedMontiArc models as input and producing equivalent C++ code as output. For this reason, the EmbeddedMontiArc to WebAssembly converter was developed.

Figure 2 shows the architecture overview.

To clarify the goal of each component which is shown in the picture, we will consider the seven most important components that are linked together:

IDE for the EmbeddedMontiArc language: it helps to write components, reveals the errors and shows incoming and outgoing ports of the components.

Web-server: it receives the requests for compiling the EmbeddedMontiArc models and sends back a finished controller, packs and extracts models, controls the compilation process providing error handling for users. The server has a queue which provides the handling of multiple users simultaneously.

EMAM2WASM generator: it gets the model from the web-server and compiles it, generating the web-assembly file, which is the "brain" of the simulator.

Testing toolchain: it provides stream testing for incoming models. The toolchain consists of EMAM2CPP (EmbeddedMontiArc to C++) generator, which generates tests, then the tests are compiled and executed. The output from the stream testing phase can be used to be shown to a user or for generating the WebAssembly file.

SVG generator: it generates a picture of the components and connections for better readability. Users can find errors easier using the visual component schema.

Browser-based simulator: it receives a compiled model from the server and instantiates it directly in the browser. Then the controller is used to process data from sensors, which are located in the car to produce commands for the actuator.

Trajectory builder and comparator: It builds a trajectory of the car movements in real time and performs a comparison with the reference trajectory. The comparator allows some deviations from the reference trajectory.

In this architecture we reuse some previously developed components and introduce new ones allowing us to accomplish our goal in the most efficient and optimal way.

5 USERS' VIEWPOINT

This section shows the users' viewpoint of the C&C tutorial; e.g. how students are going to use the web-playground to understand C&C modeling languages like EmbeddedMontiArc.

The main idea of the playground is to increase interest in the learning process using gamified tutorials. There are several simple steps in the learning process. The first tutorial is a task which already has a solution but the idea behind it is to show the main constructions and principles of the language and the playground.

Following tutorials have tasks with increasing complexity and provide some hints motivating students to use particular patterns. The visualization of the process provides a feeling for the language and an understanding of the connection between written code and real actions caused by this code.

The process of writing tests shows benefits of test-driven development and helps to understand the importance of independent testing of the components. The process of using the web-playground is very simple. Students don't have to install any applications on their computers and it is possible to use it from any platform, e.g. Mac, Windows, or Linux; only a modern browser with WebAssembly and HTML 5 support is needed. IDE, tutorials, and the visualization are located in one window and have a very intuitive interface as shown in Figure 3.

Solution for Example 1 (Parking Scenario). To start working on the solution we have to know the way how to communicate with the car to achieve the desired behavior. For this purposes, there is an interface to the simulator which is given for every tutorial. The interface is shown in Listing 1. The interface has 8 sensors to measure distances to objects, velocity, steering angle, acceleration, a position of the car and execution time.

As shown in Listing 1, ports have units and ranges in EmbeddedMontiArc. The ranges give an advantage during the testing phase and provide the possibility to use a variety of units depending on the particular case, e.g. km/h or m/s.

Listing 1: Interface of MainController to communicate with TypeScript simulator

```
component MainController{
  ports
    in Q(0m:200m) fl, //front left sensor from 0m to 200m
    in Q(0m:200m) fr, //front right sensor
    in Q(0m:200m) slf,//side left front sensor
    in Q(0m:200m) slb,//side left back sensor
    in Q(0m:200m) srf,//side right front sensor
    in Q(0m:200m) srb,//side right back sensor
    in Q(0m:200m) bl, //back left sensor
    in Q(0m:200m) br, //back right sensor
    in Q(0s:oos) time, //simulation time from 0s to infinity
    in Q(0m/s:25m/s) velocity, //car velocity
    in Q(-200m:200m) x,//car position X
    in Q(-200m:200m) y,//car position Y
    out Q(-2m/s\^2:2m/s\^2) acceleration, //car acceleration
    out Q(-180Âř:180Âř) steering; //car steering
... }
```

Based on the predefined interface, students can now create new subcomponents and connect their ports to solve this task. In this particular example (see Listing 2), we use three components which are responsible for different actions during the parking process: **VelocityController**: this component controls the velocity of the car depending on the current action (e.g. parking, searching a parking place, etc.). **ParkingController**: this component controls the steering angle of the car during the parking process. **SearchParkingPlace**: this component looks for a gap between cars for the parking.

When we have decided which component is responsible for what, it is necessary to understand which ports are needed and how they have to be interconnected.

Listing 2: Add subcomponents to MainController and connect the ports of the subcomponents

```
instance VelocityController velocityController;
instance ParkingController parkingController;
instance SearchParkingPlace searchParkingPlace;

connect velocity->velocityController.velocity;
connect velocityController.acceleration->acceleration;
connect slf->searchParkingPlace.frs;
...
```

To improve the visual perception of the interconnections, the demonstrator has an SVG generator producing a graphical C&C model. Figure 4 shows the graphical model of the solution controller.

Solution for Example 2 (Elk Test). Due to paper length limitation, the solution of this example is only available at our website: http://www.se-rwth.de/materials/ema_tutorial/.

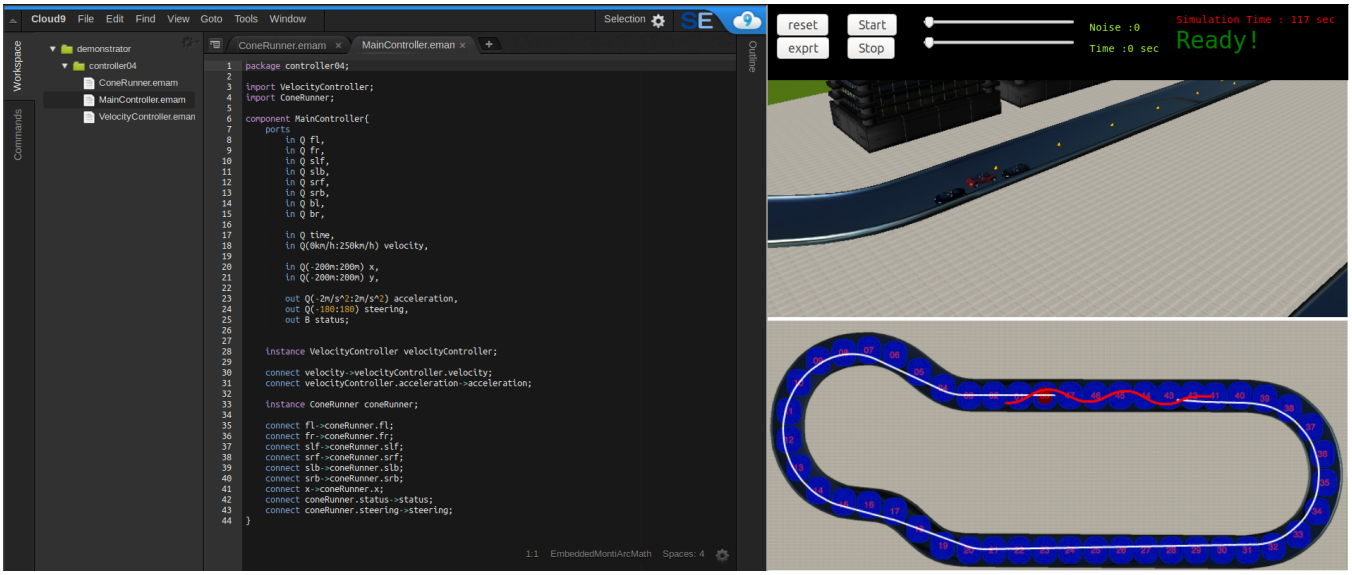

Figure 3: Screenshot of Tutorial Environment: left IDE, right top: 3D visualization, right bottom: trajectory

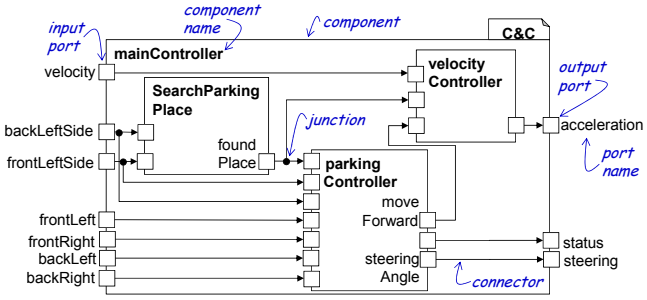

Figure 4: Graphical C&C model of Listing 1 and Listing 2

6 CONCLUSION

In this paper, we have analyzed existing tutorials from various domains. Our aim was to find the most important features having an influence on the studying process of modeling languages, to discover weaknesses of existing approaches and to overcome them. We came up with a suitable architecture using already implemented building blocks thereby decreasing the amount of work without having a negative influence on the user experience. Furthermore, we showed how to use the tutorials and demonstrated two interesting real-world examples presenting the main concepts of our component and connector modeling language and revealing important integrated features like stream testing and result verification.

REFERENCES

[1] Vincent Aravantinos, Sebastian Voss, Sabine Teufl, Florian Hölzl, and Bernhard Schätz. 2015. AutoFOCUS 3: Tooling Concepts for Seamless, Model-based Development of Embedded Systems.. In *ACES-MB*.

[2] Modelica Association et al. 2005. Modelica language specification. *Linköping, Sweden* (2005).

[3] Kent Beck, Mike Beedle, Arie van Bennekum, et al. [n. d.]. Agile Manifesto `http://agilemanifesto.org/`.

[4] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding typescript. In *Conference on Object-Oriented Programming*. Springer, 257–281.

[5] Eckard Bringmann et al. 2008. Model-based testing of automotive systems. In *2008 International Conference on Software Testing, Verification, and Validation*. IEEE, 485–493.

[6] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[7] Stefan Dziwok, Sebastian Goschin, and Steffen Becker. 2014. Specifying Intra-Component Dependencies for Synthesizing Component Behaviors.. In *ModComp@ MoDELS*. Citeseer, 16–25.

[8] Peter H Feiler and David P Gluch. 2012. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley.

[9] Christian Frohn, Petyo Ilov, Stefan Kriebel, Evgeny Kusmenko, Bernhard Rumpe, and Alexander Ryndin. 2018. Distributed Simulation of Cooperatively Interacting Vehicles. In *21st IEEE International Conference on Intelligent Transportation Systems (ITSC'18)*.

[10] Cristian González García, Jordán Pascual Espada, Begoña Cristina Pelayo García Bustelo, and Juan Manuel Cueva Lovelle. 2015. Swift vs. objective-c: A new programming language. *IJIMAI* 3, 3 (2015), 74–81.

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

[12] Wendy Hsin-Yuan Huang and Dilip Soman. 2013. Gamification of education. *Research Report Series: Behavioural Economics in Action, Rotman School of Management, University of Toronto* (2013).

[13] National Instruments. 1998. *BridgeView and LabView: G Programming Reference Manual*. Technical Report 321296B-01. National Instruments. 667 pages.

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

[15] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. 2018. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *MODELS*.

[16] Mathworks. 2016. *Simulink User's Guide*. Technical Report R2016b. MATLAB & SIMULINK. 4022 pages.

[17] Nicholas D Matsakis and Felix S Klock II. 2014. The rust language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.

[18] Sandeep Nagar. 2018. *Introduction to Octave: For Engineers and Scientists, volume 1 of 1*. Apress.

[19] Christian Schlegel, Thomas Haßler, Alex Lotz, and Andreas Steck. 2009. Robotic software systems: From code-driven to model-driven designs *(International Conference on Advanced Robotics)*. IEEE, 1–8.

[20] Stephen Wolfram. 2013. Wolfram research. *Inc., Mathematica, Version* 8 (2013), 23.