TUM

TECHNISCHE UNIVERSITÄT MÜNCHEN
INSTITUT FÜR INFORMATIK

# Towards Single-System Illusion in Software-Defined Vehicles - Automated, AI-Powered Workflow

Krzysztof Lebioda, Viktor Vorobev, Nenad Petrovic, Fengjunjie Pan, Vahid Zolfaghari, Alois Knoll

TUM-I24108

Technischer Bericht
Technische Universität München
Institut für Informatik

# Towards Single-System Illusion in Software-Defined Vehicles – Automated, AI-Powered Workflow

March 14, 2024

Krzysztof Lebioda*
email: krzysztof.lebioda@tum.de
orcid: 0000-0002-7905-8103

Viktor Vorobev*
email: vorobev@in.tum.de
orcid: 0000-0002-0473-148X

Nenad Petrovic*
email: nenad.petrovic@tum.de
orcid: 0000-0003-2264-7369

Fengjunjie Pan*
email: f.pan@tum.de
orcid: 0009-0005-8303-1156

Vahid Zolfaghari*
email: v.zolfaghari@tum.de
orcid: 0009-0004-0039-6014

Alois Knoll*
orcid: 0000-0003-4840-076X

*Technical University of Munich (TUM)
School of Computation, Information and Technology (CIT)
Chair of Robotics, Artificial Intelligence and Embedded Systems

***Abstract*—We propose a novel model- and feature-based approach to development of vehicle software systems, where the end architecture is not explicitly defined. Instead, it emerges from an iterative process of search and optimization given certain constraints, requirements and hardware architecture, while retaining the property of single-system illusion, where applications run in a logically uniform environment. One of the key points of the presented approach is the inclusion of modern generative AI, specifically Large Language Models (LLMs), in the loop. With the recent advances in the field, we expect that the LLMs will be able to assist in processing of requirements, generation of formal system models, as well as generation of software deployment specification and test code. The resulting pipeline is automated to a large extent, with feedback being generated at each step.**

## I. Introduction

The costs of vehicle software development are rising at a high rate. It is estimated that the development efforts for various software components, including OS, middleware and infotainment, but also implementation of functions and their integration, will double in 2030 as compared to 2020 [1].

Classical software development paradigms are very rigid and slow to adapt to the rising system complexity. V-model, being the de facto industry standard, is very inflexible, lacking early feedback mechanisms, and with costly scope adjustments [2]. Combined with highly standardized architectures like AUTOSAR [3], this leads to very long development cycles. On the one hand, standardization and code reuse are certainly advantageous. On the other hand, over-engineering, complexity and steep learning curve are pointed out as the main problems with these well-established frameworks [4].

Software-defined vehicles are becoming the new trend in the automotive industry, where the functionality of the car is defined, updated and modified mainly by changes in the software. This trend affects both intra-vehicular networks [5]–[7], as well as the internal car systems [8]. The increasing demand for various features and software components means that the OEMs are not able to provide the full software stack anymore. Instead, third-party software is starting to play an increasingly significant role [9].

One of the trending paradigms of software development, hardware modeling and resource allocation is model-based system engineering (MBSE) [10]–[13]. Coupled with the principles of design by contract, where the obligations of all interacting components are written down in a formal way, MBSE becomes a powerful tool that enables software-defined vehicles.

With the advent of Large Language Models (LLMs), new automation possibilities are opening. We would like to leverage the generative power of modern AI to define a new software development paradigm, which goes beyond the current standards, and which will be easily extensible in the future.

The presented approach to software development draws from agile principles, like feature-driven development (FDD), test-driven development (TDD), and low-code, model-driven development. Combining different techniques allows addressing the complexity of the problem, developing rapidly, and providing necessary feedback that is used to improve the overall system design. This fits well with the software-defined vehicle paradigm, where the system keeps evolving by updating and modifying mainly the software components.

When using the proposed workflow, software designers and developers should perceive the system as a single logical entity – single-system illusion. Different software modules are not bound to the underlying OS, middleware, or hardware topology. The RACE project is an example of early designs that show how such a system may look like [14]. This is in contrast

with the current systems, where the functions are spread to many different ECUs, and are oftentimes implemented in a way that accounts for the inter-component connections, which breaks modularity.

In order to achieve the goal of providing a single-system illusion, the various components of the software stack must be clearly separated. Applications, middleware, OS and hardware must be sufficiently modular to allow modifications of one layer without the need to totally rewrite the others. Again, RACE has shown how to clearly delineate applications from the runtime environment [14], [15] and even how to hide specific safety mechanisms from the service-level developers [15]. However, the resulting system was strictly bound to the runtime environment with a single message exchange paradigm (publish-subscribe). We would like to go beyond and allow the designers to choose their middleware without such constraints, possibly even allow for multiple middlewares within the same system. We also strive for a system where certain general safety mechanisms, like process redundancy, watchdogs and monitoring, or data integrity checks, are separated from the applications and the runtime environment and where they can be applied to other components as required.

The innovative aspect of the workflow which we propose, is the use of modern AI, specifically LLMs, in the development process. The goal is to use AI in synergy with model-driven development approaches, where formal system information, including the system model (so-called instance model) and constraints in formal languages, is generated from the available components' (software and hardware) description, and from the set of requirements, both functional (which functions to include) and non-functional (safety, performance). Based on the safety requirements, functions have different safety measures (redundancy, failure handling mechanisms) applied to them. Formal verification is performed on the instance model relying on a rule-based approach, such as Object Constraint Language (OCL) [16]. This way, it is possible to prove that logical connections between elements are correct while the properties of distinct elements are within the desired bounds.

The generated model is used as a base for an automatic resource allocation method, which takes into consideration the hardware and software specifications as well as allocation constraints. Flexibility of the optimization algorithm must be emphasised, as it ought to support both test environments and the target vehicle architectures. It must also be able to generate an optimal configuration in the Pareto sense, i.e., based on the chosen optimization criteria, like maximal performance or minimal power consumption.

The resulting allocation matrix together with the instance model become inputs to a code-generating system based on generative AI. The functional model is combined with information about the desired runtime environment, acceptance criteria, and the list of available software functions from the software catalog. The system outputs working code that can be deployed to the desired architecture. The process of code generation includes parametrization of deployment code, e.g., with connection details like addresses and ports, generation of adapter code that translates data from function-specific formats into middleware primitives, but also tests that will be used for functional and non-functional verification. The synthesized architecture can be verified in simulation against the non-functional requirements, e.g., by using fault injection into the stream, similar to RACE [15].

## II. Methodology

The workflow was designed with the following principles in mind: short development cycles, automated feedback at each step, focus on modularity and flexibility, and automated generation of boilerplate code. A number of artifacts is produced as a result of each step. All of these artifacts should be available to the users in a human-readable form. However, they should also have a machine-readable format, which can be used for automatic verification. Verification at each step provides early feedback, and should keep the development cycles short. The software components at all levels of abstraction should be modular to allow the AI tools to easily (re)generate the boilerplate code as needed.

The proposed workflow is presented on Fig. 1. It consists of two phases – the design phase, and the run-time phase. During the design phase, formal descriptions of the desired system in the form of models and constraints are generated, based on the provided requirements, standards, available software and hardware components etc. During the run-time phase, glue code, tests and deployment descriptors are generated from the model. The software is deployed and tested either on the test bench, or on the target architecture.

### A. Feature-driven and model-driven development

The developers should first and foremost focus on the features that are supposed to be deployed to the target vehicle hardware. At a very abstract level it can be achieved by writing a set of requirements in natural language. A generative AI in the form of LLM is used to process these abstract ideas and put them against the available software functions, hardware specifications, safety standards and vehicle abstraction specifications. The LLM outputs an instance model, which is based on the given metamodel and which abstracts the target system, and formal constraints that are derived from requirements. The generated instance model represents how the function graph is connected, what the requirements are for all nodes, as well as what properties the target hardware has. At this point, the designers should not be concerned about the details of virtualization, operating systems, or the precise mapping between software and hardware.

**Inputs:**

- *Metamodel* – an abstract language of system description. Metamodel does not describe any particular system. Instead, it is a template that must be populated, creating a so-called instance model. One way the meta- and instance models can be written is the Object Management Group (OMG) standard. When it comes to implementation, we rely on Ecore [17] within Eclipse Modelling Framework

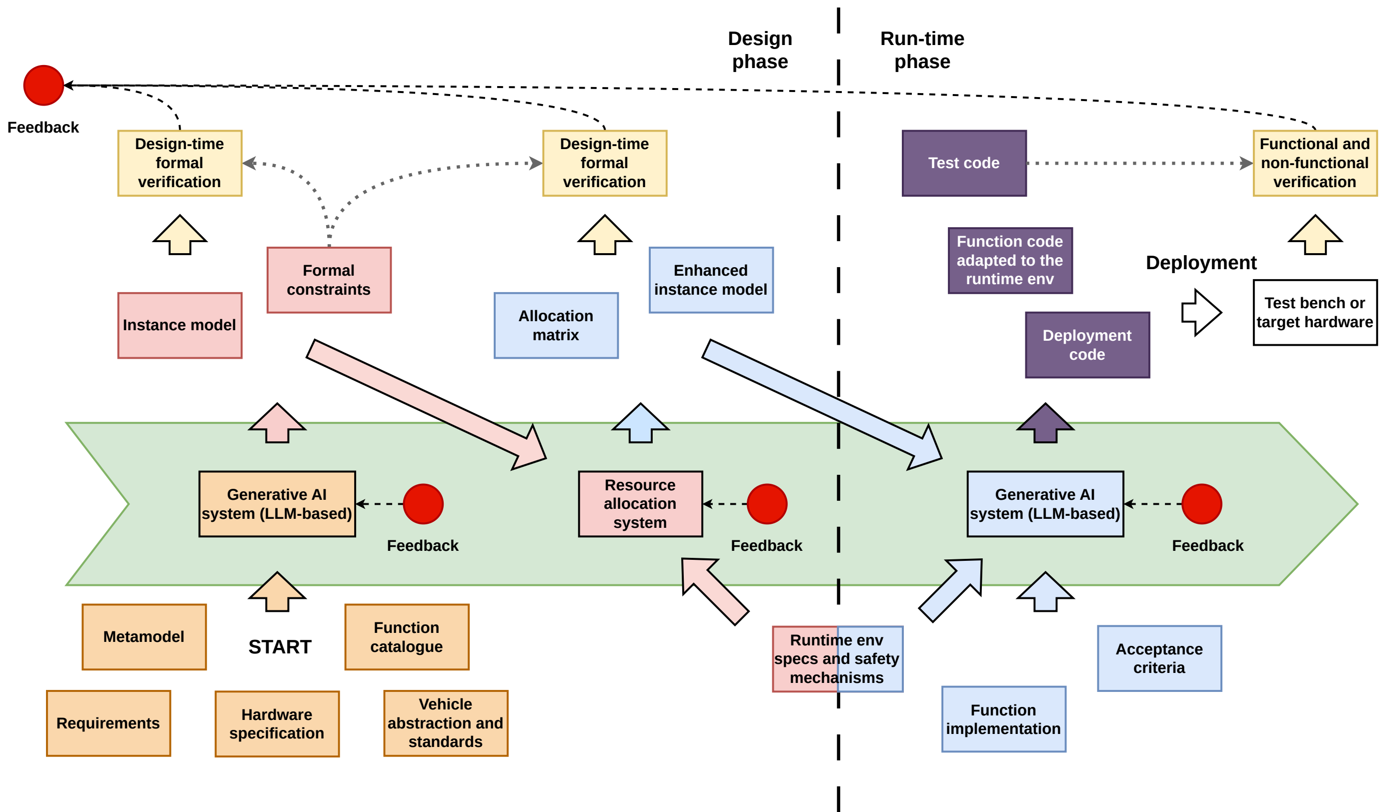


Fig. 1. Proposed workflow with generative AI in the loop. Steps are color-coded: 1) Inputs in orange, outputs in red - generation of the instance model and formal constraints. The formal constraints are used for automatic verification of the instance models; 2) Inputs in red, outputs in blue - resource allocation, which produces an allocation matrix and enhances the instance model with details about software-to-hardware mapping; 3) Inputs in blue, outputs in purple - code generation. The tests generated during this step are used for automatic testing of the deployed system; 4) Yellow - validation and verification, which is performed as part of the other steps, and is not a separate step as such.

(EMF), which provides the tools for metamodel design, model instance creation and API for model instance manipulation. Other modeling language, like SpesML, may be used, depending on the desired toolchain.

- *Requirements* – a list of requirements for the system in free-form text. Requirements contain basic system description, constraints, etc. Although the process of gathering, extraction and improving of the requirements set is an interesting research problem on its own, we are not going to address it in this document. However, we do intend to assess the quality of models generated by the LLMs using existing rule-based systems, and test cases where requirements are contradictory or incomplete.
- *Vehicle abstraction and standards* – essential for defining vehicular systems, for example, ISO 26262 [18], ISO 23150 [19], Vehicle Signal Specification [20].
- *Function catalogue* – a list of available software components (functions), interface specifications, and requirements regarding computational power, power consumption, etc.
- *Hardware specification* – specifications of the available hardware. The specification could be in the form of a graph, data sheet, or potentially another instance model.

**Outputs:**

- *Instance model* – an instance of the metamodel, which describes the connections between all involved functions. The function nodes are annotated with their requirements in terms of safety, required power, etc. At this point in the design phase, the hardware graph is included in the model, but the details of mapping of software to the hardware are missing. The model will be enhanced with full mapping in the following step – resource allocation. As in the case of the metamodel, we make use of Ecore for the implementation, but other modeling languages may also be used.
- *Formal constraints* – a set of logical constraints that must be satisfied within a user-provided model instance. These rules are typically derived from a reference architecture, requirements or relevant ISO standards. In our approach, we employ Object Constraint Language (OCL) to express and enforce these verification rules.

**Verification and feedback:**

- Feedback includes testing that interfaces of different components are compatible and meet the criteria specified in the formal contract. Failure to adhere to this contract may indicate that the chosen components are incompatible, or

that the interface specification is incorrect.
- Violations of formal constraints within the scope of user-created specification (including component quantity, property values, or inter-component relationships) will result in model verification failure. The system will provide the user with natural language suggestions outlining necessary modifications. To generate these suggestions, the LLM will translate the verification results into actionable guidance.

### B. Resource allocation

During this step, the generated instance model and formal constraints are translated into a format recognizable by the chosen mapping and optimization algorithm. The algorithm should be flexible to support various optimization criteria to develop a Pareto-optimal solution based on the selected goals (cost, performance, power use, etc.). Existing approaches to mapping and optimization offer many possibilities: from classical optimization techniques like integer linear programming [21], [22] through genetic algorithms [23], [24], to the use of graph neural networks [25]–[27].

**Inputs:**

- *Instance model* – model generated in the previous step.
- *Formal constraints* – formal constraints generated in the previous step. They are derived from requirements and contain restrictions and rules for resource allocation.
- *Runtime environment specification* – specification of the chosen communication middleware (ROS, ZeroMQ), available message passing paradigms, and virtualization mechanisms (Docker, VM).
- *Available safety mechanisms* – abstract description of safety mechanisms, like redundancy, failure masking by voting, etc. These will later be instantiated according to the chosen runtime environment and available software components (functions).

**Output:**

- *Allocation matrix* – mapping between software and hardware components.
- *Enhanced instance model* – instance model generated in the previous steps, but enhanced with detailed software-hardware mapping.

**Validation and feedback:**

- Similar to the previous step, validation includes testing of the enhanced model against a set of OCL rules covering redundancy aspects and safety-critical constraints.

### C. Code generation and deployment

In this step, another instance of generative AI is used to put all available pieces together into working code. Techniques like discriminative reranking [28], [29], application of verifier models [30], [31] or self-collaboration [32] may be applied to improve the code quality. The generated code may include (but is not limited to) glue code for various functions, like adapters from function interfaces and middleware interfaces, deployment files parametrized with proper addresses, ports and including process redundancy, injection of test code into the pipelines, and actual test cases [33], [34] based on acceptance requirements.

**Input:**

- *Allocation matrix* – mapping of software to hardware generated in the previous step
- *Runtime environment specification* – specification of the chosen communication middleware (ROS, ZeroMQ), available message passing paradigms, and virtualization mechanisms (Docker, VM).
- *Acceptance criteria* – functional and non-functional criteria used for the generation of test code and test cases.

**Output:**

- *Deployment files* – adapted to the particular architecture and runtime environment
- *Test cases* – automatically generated test cases that can be used to validate both functional and non-functional requirements.
- *Function code* with proper wrapping (Docker) and adapter code for data translation.

**Verification and feedback:**

- Execution of the acceptance tests, either in a simulated environment, or on the target architecture.
- Feedback is divided into functional and non-functional. Failures detected during execution of the functional test could indicate bugs in the software modules or in the integration code. Failures of the non-functional tests may indicate that certain requirements were incorrect or missing, or that the optimization criteria during the allocation phase must be modified.

## III. Scope, complementary work, limitations

In this document we are focusing on automatic generation of software systems, integration and deployment code, and mapping to hardware. However, there are many places where complementary research can be conducted. The first issue is completeness and consistency of the user requirements. Although the instance model and formal constraints produces by the LLM in the first step are verifiable, we do not know how well the LLM will be able to handle conflicting and incomplete requirements, and how it will impact the quality of the generated artifacts. Another possible venue of research is the hardware specification and representation. In the current workflow it is assumed that this description is provided to the LLM in the form of a graph (or a semi-formal textual description), but it ought to be possible to generate the hardware model automatically from the requirements, similar to how we generate the software model. An example of Ecore model instances creation relying on ChatGPT in presented in [35].

Code generated by LLMs is far from perfect [36]. We are planning on focusing on certain simple use cases and building a proof-of-concept pipeline. Human supervision will

definitely be needed, especially to examine correctness of the automatically generated verification code, like test cases. We expect that in the near future, the quality of code generated by LLMs will improve massively, which will allow us to generate more complex systems, and to progressively remove the need for human supervision.

## IV. Conclusion

We propose a software development process that extends beyond the current industrial standards. In this process, the generative AI should become an integral part and progressively take care of many menial tasks. The advantage of using AI over classical tools for automatic translation is the generative power of the models, as well as the ability to understand natural language. Although the currently available LLMs have many shortcomings, the progress in the field is blazingly fast, and their rapid evolution suggests the potential to revolutionize code generation in the coming years. Our proposed workflow is designed for extensibility and adaptability, ensuring it remains relevant amidst ongoing advancements in AI capabilities, allowing an increasing number of stages of the process to become automated.

## V. Glossary

contract
: Design by contract is a software development methodology that emphasizes the explicit definition of formal contracts between software components [37]. These contracts specify preconditions (what must be true before a component is used), postconditions (what must be true after execution), and invariants (conditions that must always hold true). Design by contract can be enforced through runtime assertions, unit tests, or even integrated into a programming language's syntax. This approach enhances software reliability, eases debugging, and facilitates code comprehension. 1, 3

Ecore
: Language of the metamodel used in Eclipse Modeling Framework. 2, 3

ECU
: Electronic Control Unit. It is an electronic device in a vehicle that is responsible for a single function. 2

FDD
: Feature-Driven Development. It is a paradigm where the software system is iteratively developed in a series of steps, starting with an abstract model of the system, followed by extraction of a set of desired features, and ending with feature implementation and integration [38]. 1

feature
: Composed of one or more functions connected together using a certain runtime environment, usually corresponds to a certain use-case. 1

function
: A single, self-contained piece of software, that performs a certain function. 1, 4

LLM
: Large Language Model. 2–5

MBSE
: Model-Based Systems Engineering is a formalized methodology within systems engineering that emphasizes using models as the primary means of information exchange and system representation [39]. This contrasts with traditional document-centric approaches. MBSE centers on creating and leveraging domain-specific models or metamodels, which capture system requirements, design, analysis, and verification elements throughout the development lifecycle. 1

OCL
: Object Constraint Language. 4

OEM
: Original Equipment Manufacturer. 1

OMG
: Object Management Group. 2

RACE
: Centralized Platform Computer Based Architecture for Automotive Applications. 1, 2

runtime environment
: Communication middleware and virtualization mechanisms. 2

TDD
: Test-Driven Development is a software development methodology that centers on the iterative creation of unit tests prior to the implementation of functional code [40] This approach mandates that a test case specifying the desired behavior of a code unit be written before the production code itself. As development progresses, the test suite continuously executes. New code is only written if it fulfills the requirements outlined in a failing test. 1

## References


[1] O. Burkacky, J. Deichmann, and J. P. Stein, *Automotive software and electronics 2030: Mapping the sector's future landscape*, 2019. [Online]. Available: https://www.mckinsey.com/~/media/mckinsey/industries/automotive%5C%20and%5C%20assembly/our%5C%20insights/mapping%5C%20the%5C%20automotive%5C%20software%5C%20and%5C%20electronics%5C%20landscape%5C%20through%5C%202030/outlook%5C%20on%5C%20the%5C%20automotive%5C%20software%5C%20and%5C%20electronics%5C%20market%5C%20through%5C%202030/automotive-software-and-electronics-2030-full-report.pdf.

[2] G. Kumar and P. K. Bhatia, "Comparative Analysis of Software Engineering Models from Traditional to Modern Methodologies," in *2014 Fourth International Conference on Advanced Computing & Communication Technologies*, 2014, pp. 189–196. DOI: 10.1109/ACCT.2014.73.

[3] M. Staron and D. Durisic, "AUTOSAR Standard," in *Automotive Software Architectures: An Introduction*. Cham: Springer International Publishing, 2017, pp. 81–116, ISBN: 978-3-319-58610-6. DOI: 10.1007/978-3-319-58610-6_4.

[4] S. Martínez-Fernández, C. P. Ayala, X. Franch, and E. Y. Nakagawa, "A Survey on the Benefits and Drawbacks of AUTOSAR," in *Proceedings of the First International Workshop on Automotive Software Architecture*, ser. WASA '15, Montréal, QC, Canada: Association for Computing Machinery, 2015, pp. 19–26, ISBN: 9781450334440. DOI: 10.1145/2752489.2752493.

[5] M. M. Islam, M. T. R. Khan, M. M. Saad, and D. Kim, "Software-defined vehicular network (SDVN): A survey on architecture and routing," *Journal of Systems Architecture*, vol. 114, p. 101 961, 2021, ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2020.101961.

[6] S. Correia, A. Boukerche, and R. I. Meneguette, "An Architecture for Hierarchical Software-Defined Vehicular Networks," *IEEE Communications Magazine*, vol. 55, no. 7, pp. 80–86, 2017. DOI: 10.1109/MCOM.2017.1601105.

[7] I. Ku, Y. Lu, M. Gerla, R. L. Gomes, F. Ongaro, and E. Cerqueira, "Towards software-defined vanet: Architecture and services," in *2014 13th Annual Mediterranean Ad Hoc Networking Workshop (MED-HOC-NET)*, 2014, pp. 103–110. DOI: 10.1109/MedHocNet.2014.6849111.

[8] M. Haeberle, F. Heimgaertner, H. Loehr, *et al.*, "Softwarization of automotive e/e architectures: A software-defined networking approach," in *2020 IEEE Vehicular Networking Conference (VNC)*, 2020, pp. 1–8. DOI: 10.1109/VNC51378.2020.9318389.

[9] *Multi-source automotive software stacks white papers*, 2023. [Online]. Available: https://insight.sbdautomotive.com/rs/164-IYW-366/images/Multi-source%5C%20Automotive%5C%20Software%5C%20Stacks%5C%20White%5C%20Papers.pdf.

[10] J. D'Ambrosio and G. Soremekun, "Systems engineering challenges and MBSE opportunities for automotive system design," in *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2017, pp. 2075–2080. DOI: 10.1109/SMC.2017.8122925.

[11] U. Pohlmann and M. Hüwe, "Model-driven allocation engineering: specifying and solving constraints based on the example of automotive systems," *Automated Software Engg.*, vol. 26, no. 2, pp. 315–378, Jun. 2019, ISSN: 0928-8910. DOI: 10.1007/s10515-018-0248-3.

[12] I. Al-Azzoni, J. Blank, and N. Petrović, "A Model-Driven Approach for Solving the Software Component Allocation Problem," *Algorithms*, vol. 14, no. 12, 2021, ISSN: 1999-4893. DOI: 10.3390/a14120354.

[13] F. Pan, J. Lin, M. Rickert, and A. Knoll, "Automated Design Space Exploration for Resource Allocation in Software-Defined Vehicles," in *2023 IEEE Intelligent Vehicles Symposium (IV)*, 2023, pp. 1–8. DOI: 10.1109/IV55152.2023.10186605.

[14] S. Sommer, A. Camek, K. Becker, *et al.*, "RACE: A Centralized Platform Computer Based Architecture for Automotive Applications," in *2013 IEEE International Electric Vehicle Conference (IEVC)*, 2013, pp. 1–6. DOI: 10.1109/IEVC.2013.6681152.

[15] K. Becker, J. Frtunikj, M. Felser, *et al.*, "RACE RTE: A runtime environment for robust fault-tolerant vehicle functions," in *3rd Workshop on Critical Automotive applications - Robustness & Safety (CARS)*, Paris, France, Sep. 2015. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01192987.

[16] "Object Constraint Language." (2024), [Online]. Available: https://www.omg.org/spec/OCL/2.4/About-OCL (visited on 02/19/2024).

[17] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, 2nd ed. Boston, MA: Addison-Wesley, 2009, ISBN: 978-0-321-33188-5.

[18] ISO 26262:2018, "Road vehicles – functional safety," International Organization for Standardization, Geneva, Switzerland, Standard, Dec. 2018.

[19] ISO 23150:2023, "Road vehicles – Data communication between sensors and data fusion unit for automated driving functions," International Organization for Standardization, Geneva, Switzerland, Standard, May 2023.

[20] "Vehicle Signal Specification (VSS) v.4.1," COVESA, Standard, 2023. [Online]. Available: https://covesa.github.io/vehicle_signal_specification/.

[21] H. Askaripoor, M. H. Farzaneh, and A. Knoll, "A Model-Based Approach to Facilitate Design of Homogeneous Redundant E/E Architectures," in *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*, 2021, pp. 3426–3431. DOI: 10.1109/ITSC48978.2021.9565115.

[22] F. Pan, J. Lin, M. Rickert, and A. Knoll, "Resource Allocation in Software-Defined Vehicles: ILP Model Formulation and Solver Evaluation," in *2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC)*, 2022, pp. 2577–2584. DOI: 10.1109/ITSC55140.2022.9922526.

[23] S.-C. Kao and T. Krishna, "GAMMA: automating the HW mapping of DNN models on accelerators via genetic algorithm," in *Proceedings of the 39th International Conference on Computer-Aided Design*, ser. ICCAD '20, Virtual Event, USA: Association for Computing Machinery, 2020, ISBN: 9781450380263. DOI: 10.1145/3400302.3415639.

[24] N. Weng, N. Kumar, S. Dechu, and B. Soewito, "Mapping task graphs onto Network Processors using genetic algorithm," in *2008 IEEE/ACS International Conference on Computer Systems and Applications*, 2008, pp. 481–488. DOI: 10.1109/AICCSA.2008.4493576.

[25] M. J. A. Schuetz, J. K. Brubaker, and H. G. Katzgraber, "Combinatorial optimization with physics-inspired graph neural networks," *Nature Machine Intelligence*, vol. 4, pp. 367–377, 2022. DOI: https://doi.org/10.1038/s42256-022-00468-6.

[26] Q. Cappart, D. Chételat, E. Khalil, A. Lodi, C. Morris, and P. Veličković, *Combinatorial optimization and rea-*

*soning with graph neural networks*, 2022. arXiv: 2102.09544 [cs.LG].

[27] K. Rusek, J. Suárez-Varela, P. Almasan, P. Barlet-Ros, and A. Cabellos-Aparicio, "RouteNet: Leveraging Graph Neural Networks for Network Modeling and Optimization in SDN," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 10, pp. 2260–2270, 2020. DOI: 10.1109/JSAC.2020.3000405.

[28] A. Lee, M. Auli, and M. Ranzato, "Discriminative Reranking for Neural Machine Translation," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, C. Zong, F. Xia, W. Li, and R. Navigli, Eds., Online: Association for Computational Linguistics, Aug. 2021, pp. 7250–7264. DOI: 10.18653/v1/2021.acl-long.563.

[29] Z. Li and T. Xie, *Using LLM to select the right SQL Query from candidates*, 2024. arXiv: 2401.02115 [cs.CL].

[30] A. Ni, S. Iyer, D. Radev, *et al.*, *LEVER: Learning to Verify Language-to-Code Generation with Execution*, 2023. arXiv: 2302.08468 [cs.LG].

[31] Y. Li, Z. Lin, S. Zhang, *et al.*, *Making Large Language Models Better Reasoners with Step-Aware Verifier*, 2023. arXiv: 2206.02336 [cs.CL].

[32] Y. Dong, X. Jiang, Z. Jin, and G. Li, *Self-collaboration Code Generation via ChatGPT*, 2023. arXiv: 2304.07590 [cs.SE].

[33] B. Chen, F. Zhang, A. Nguyen, *et al.*, *CodeT: Code Generation with Generated Tests*, 2022. arXiv: 2207.10397 [cs.CL].

[34] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, *Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation*, 2023. arXiv: 2305.01210 [cs.SE].

[35] N. Petrovic and I. Al-Azzoni, "Automated Approach to Model-Driven Engineering Leveraging ChatGPT and Ecore," in *6th International Conference on Applied Electromagnetics – PES 2023*, 2023, pp. 166–168.

[36] C. Spiess, D. Gros, K. S. Pai, *et al.*, *Quality and Trust in LLM-generated Code*, 2024. arXiv: 2402.02047 [cs.SE].

[37] R. Mitchell and J. McKim, *Design by contract, by example*, eng. Boston, MA: Addison Wesley Boston, MA, 2002, ISBN: 9780201634600.

[38] S. R. Palmer and M. Felsing, *A practical guide to feature-driven development*. Pearson Education, 2001.

[39] D. D. Walden, T. M. Shortell, G. J. Roedler, *et al.*, *INCOSE Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities*, en, 5th ed., INCOSE, Ed. New York: John Wiley & Sons Inc., Jun. 2023.

[40] Beck, *Test Driven Development: By Example*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002, ISBN: 0321146530.