Toward Intelligent Generation of System Architectures

Oliver von Heißen[1], Isaac Mpidi Bita[1], Fabian Hanke[1], Aschot Hovemann[1], Roman Dumitrescu[2]

[1]Fraunhofer IEM, Department Digital Engineering, Paderborn, Germany
[2]University of Paderborn, Heinz-Nixdorf-Institute, Paderborn, Germany

Abstract: Designing new systems involves collaboration across mechanics, electronics, and computer science, often through multiple iterations. We introduce a method using large language models to streamline architecture creation, incorporating the requirements, functional, logical, and physical (RFLP) concept, thus speeding up the process and reducing errors. A proof of concept is demonstrated within Cameo Systems Modeler, applied to a remote-controlled autonomous car project. Feedback from model-based system engineering experts confirm the approach's relevance and provides direction for enhancements.

Keywords: Engineering Design, Systems Engineering (SE), Artificial Intelligence (AI), Product Development

1 Introduction

In our modern world systems become more and more complex. The different areas of mechanics, electronics and computer science must work together to design and implement a new system, e.g., a car in the automotive industry (Joseph D'Ambrosio, 2017). Model-Based Systems Engineering (MBSE) introduces a modern approach to handle the complexity and eases collaboration between experts from different fields. By using diagrams to represent various system aspects, MBSE streamlines the design of system architectures (Madni and Sievers, 2018).

The development of a system architecture is a process involving multiple experts and iterations. At the beginning, requirements are derived from the Stakeholders, that must be understood and interpreted by the system architects. A design process like RFLP (Requirements, Functional, Logical, and Physical design) (Baughey, 2011) systematically guides experts through the steps of system development. This design process is iterated several times by going from a simple and shallow structure to a detailed and complex architecture following the twin peaks model (Chung and do Prado Leite, 2009). Such iterative improvements are very time-consuming. Furthermore, the creation and change of architectures, requires the creation and maintenance of tracelinks. They ensure the correct connections between system views and can be especially tedious work and error prone (Gotel et al., 2012). Parallel to advancements in systems engineering, there has been a significant evolution in artificial intelligence, highlighted by the emergence of generative AI models such as ChatGPT (Bahrini et al., 2023) and LLaMA (Touvron et al., 2023). These models, present new opportunities for automating system architecture generation.

This paper explores the potential of leveraging large language models to automate the initial drafting of system architectures. We propose the development of a Cameo Systems Modeler plugin that integrates these AI capabilities to modify diagrams and model elements directly, thus supporting experts in refining these drafts into detailed, final designs. This is a novel approach to create system designs in contrast to established approaches like function-based elicitation and design space exploration (Müller et al., 2019; Vanommeslaeghe et al., 2019).

The structure of this paper is organized as follows: Section 2 introduces essential concepts in MBSE and AI to provide a foundation for understanding the plugin's functionality. Section 3 offers a comprehensive examination of the plugin and its role in the architectural design process. Section 4 showcases a practical application using a RaceCar model project and includes an evaluation from MBSE experts. Finally, Section 5 summarizes the main findings and discusses directions for future research.

2 Fundamentals and Related Work

2.1 Fundamentals of Model-Based Systems Engineering

MBSE uses models as the primary medium for information exchange throughout the system development lifecycle to improve productivity, quality, and traceability. As defined by INCOSE, MBSE supports system requirements, design, analysis, verification and validation from conceptual design through development and into later lifecycle phases (Walden et al., 2003). Key benefits include complexity management, unified cross-domain models for better communication between disciplines, and integration of disparate data sets to maintain consistency throughout the development process. The RFLP process, which details requirements, functional and logical architectures, and the creation of physical structures, emphasizes the method's systematic approach to design and traceability (Baughey, 2011). Guidelines, recommendations, or so-called architecture drivers can improve the design process (Kaiser et al., 2016; Kharatyan et al., 2022).

2.2 Advances in Large Language Models

The convergence of AI and Large Language Models (LLMs) represents a significant leap forward in computational technologies, particularly in natural language processing (NLP). AI, machine learning, NLP, robotics and computer vision, aims to automate complex tasks and improve decision making (McCarthy et al., 1955). LLMs, especially with the advent of the Transformer architecture (Vaswani et al., 2017), have revolutionized language understanding and generation, offering improved training speed and efficiency. Innovations such as the GPT series and more efficient models such as LLaMA and Mistral 7B (Jiang et al., 2023; Touvron et al., 2023) demonstrate the evolving capabilities of LLMs and their adaptability to different linguistic tasks, highlighting the rapid progress of the field and the potential for further advances.

2.3 Opportunities and Challenges of LLMs in Engineering Applications

The integration of LLMs into engineering, particularly within Model-Based Systems Engineering (MBSE), is an evolving frontier with great potential and notable challenges. LLMs, with their advanced capabilities in natural language processing, machine learning, and data mining, offer innovative solutions for requirements structuring, design validation, quality assurance, and data model generation. This section explores the opportunities offered by LLMs in engineering applications, supported by emerging trends and case studies, as well as the challenges that need to be addressed for their effective integration.

Case Studies: Demonstrating the real-world application of LLMs to engineering use cases:

- Software Engineering: FAN ET AL. (2023) demonstrate the utility of LLMs in code generation, error detection and optimization, marking a leap in software engineering practice (Fan et al., 2023).
- Engineering design: GÖPFERT ET AL. (2023) discuss the role of LLMs in enhancing creativity and decision making in design, indicating a shift towards more intelligent methods (Göpfert et al., 2023).

Challenges: Despite these opportunities, the application of LLMs within engineering faces several challenges:

- Model hallucination and value misalignment: Issues such as irrelevant outputs and misalignment with engineering values hinder the effectiveness of LLMs, with CHEN ET AL. (2024) proposing the LLM-SE framework to mitigate this (Chen et al., 2024).
- Integration complexity: The difficulty of integrating LLMs with existing tools and workflows, as well as data biases and the need for hybrid models, remains a significant barrier.
- Informed Machine Learning: RUEDEN ET AL. (2021) emphasize the integration of domain-specific knowledge to improve the performance and reliability of LLMs, addressing data scarcity and model interpretability (Rueden et al., 2021).

The integration of LLMs into MBSE represents a methodological advance that promises to improve engineering practice through increased efficiency and innovation. However, realizing this potential requires overcoming challenges related to model reliability, integration complexity and adherence to engineering standards. The forthcoming discussion on the "AI4Cameo Plugin" illustrates a practical application of AI in MBSE, demonstrating the theoretical benefits of LLMs through an intelligent assistant within the Cameo modelling tool, highlighting a move towards seamless integration and methodological innovation in engineering workflows.

3 AI4Cameo Plugin

The plugin simplifies using LLMs with Cameo Systems Modeler by utilizing several endpoints from the Cameo Java Application Programming Interface (API) to generate system elements and connections. First, we introduce the technologies used, then outline the plugin's software architecture, detailing its module functions and interfaces. Finally, we demonstrate the workflow for system architecture creation via the plugin.

3.1 Technologies

The application, functioning as a plugin for Cameo Systems Modeler, employs the Java Open API for direct interaction with diagram elements, alongside additional libraries for model processing. Cameo Systems Modeler, by Dassault Systèmes, enables system engineering with a focus on model-based systems engineering (MBSE), facilitated by its Java Open API for enhanced software customization and integration. Our work utilizes Version 2021x of this software. In the realm of artificial intelligence, LLMs like OpenAI's ChatGPT and Meta's LLaMA, utilize deep learning and the transformer architecture for a range of natural language processing tasks, from text generation to question-answering. For our project, we leverage Java libraries including OkHttp for efficient HTTP requests and network management, and the org.json library for JSON data manipulation, enhancing our application's functionality.

3.2 Software Architecture

The main components for the AI4Cameo plugin are illustrated in Figure 1, where each individual module will be presented in the following.

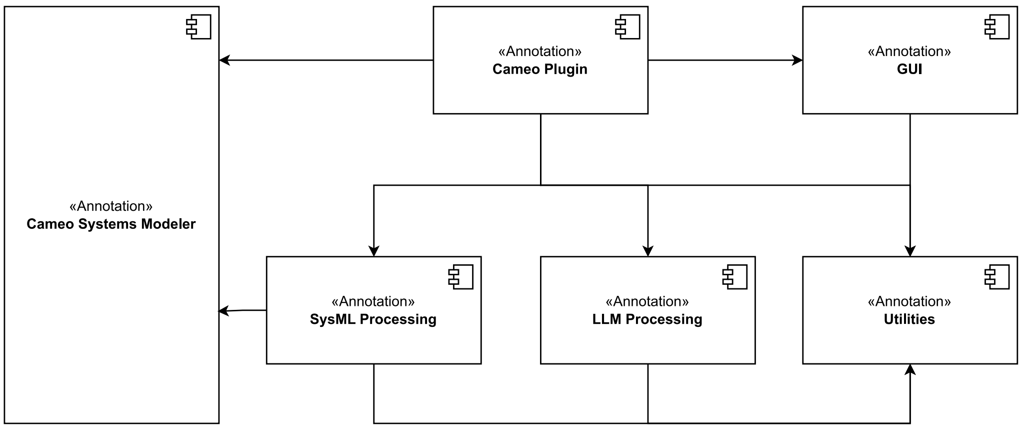


Figure 1. AI4Cameo Software Architecture

The Cameo Plugin component realizes the basic structure for the plugin to be working with Cameo Systems Modeler. This is handled by the class *AI4CameoPlugin*. Furthermore, *MainMenuConfigurator* handles the menu item in the Cameo Systems Modeler interface, such that the function can be easily called. The main routine of the plugin is implemented in a class called *ArchitectureCreationAction*, which will be called when starting the plugin in the interface. The connection to large language models was realized in the LLM Processing module. The class *LlmController* was implemented for sending REST requests to the LLM servers. It is compatible to LLM servers offering the ChatGPT REST interface. For simplification we introduced a class *Prompt*, which bundles all informations, which is necessary for sending the request. This includes the prompt text itself, as well as temperature for further configuration. The interaction with the plugin was implemented using graphical user interfaces (GUIs) in Java, utilizing Swing. The main window, *CreateArchitectureWindow*, handles the configuration for the architecture and LLM. We implemented *RequirementDialog* for selecting requirements and *ProcessingWindow* to indicate that the architecture is being created, as this can take a longer processing time. The SysML Processing module performs the actual creation of SysML objects. Based on LLM results, *FunctionalArchitectureCreator* and *LogicalArchitectureCreator* generate functional and logical architectures, including SysML blocks, connections, stereotypes, and visual elements. The *Requirement* class simplifies handling requirement data, which the *RequirementHandler* processes. General functionalities are managed by the Utilities module. Configurations, including prompts and LLM specifications, are stored in config.json and processed by the *Config* class. Logging is managed by the *Logger* class. We also created an “AI4Cameo-Profile” in Cameo Systems Modeler, defining “Functional Block” and “Logical Block” stereotypes for architecture creation. This profile must be imported when generating a new project with an architecture.

3.3 Plugin Workflow

The novel workflow for the automatic generation of a system architecture involves three steps. First, we prepare a system specification that includes a general system description and optional requirements. Next, we start the plugin using the integrated user interface, which facilitates the use of the prepared system specification. Then, the plugin generates the system architecture: a functional architecture based on the system description and requirements, followed by a logical architecture using the system specification and functional architecture. Both architectures are created as block definition diagrams utilizing the stereotypes from the AI4Cameo-Profile.

Preparation: For the architecture creation process, we consider two information sources for the system specification. A general system description shall guide the LLM to a correct overall system and set the correct context for the requirements. Furthermore, we consider an optional set of requirements to further specify the system and allow for a usage of tracelinks to the created functional elements. The requirements must be specified in Cameo Systems Modeler to be properly processed. This is achieved by storing the requirements in the containment tree of Cameo as a requirement element. For the architecture generation, we process the name and textual description of each requirement. A detailed description can thereby improve the quality of the architecture.

Starting the plugin: The plugin is integrated into the user interface of Cameo Systems Modeler. The most important user configuration is accessible through these interfaces. An expert can do further configuration using the internal files, which will be presented in the architecture creation process. The plugin can be started from the main menu bar in Cameo using the custom menu item “AI4Cameo” and “Generate architecture from specification” as shown in Figure 2.

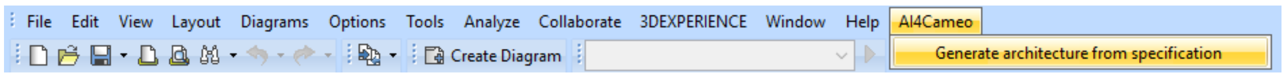
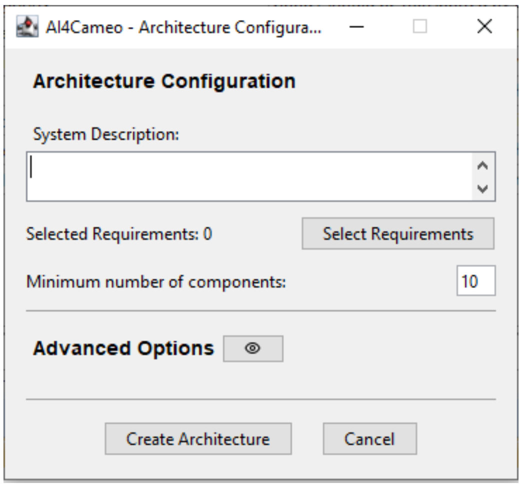


Figure 2. Interface integration of AI4Cameo in Cameo Systems Modeler

In the following step, we configure the system architecture in the specification window (Figure 3). This allows us to insert the general system description. The "Select Requirements" button leads us to another dialog window to specify the requirements for the architecture generation. This reads all requirements from the current Cameo project and allows the user to select only a subset of requirements for a more fine-grained control of the specification. As models tend to respond with a short answer, we want to adjust the complexity of the system with our request. This is handled by specifying a minimum number of components for the functional and logical architecture in the interface. Lastly, the advanced options provide further configuration for the LLM. We can hereby, specify the number of max tokens for the answer and regulate the temperature of the model.


AI4Cameo - Architecture Configura...
Architecture Configuration
Select Requirements
Advanced Options
Create Architecture
Cancel

System Description:

Selected Requirements: 0

Minimum number of components: 10

Figure 3. AI4Cameo interface for the architecture configuration

Generating a functional architecture: Based on the configuration in the user interface, the plugin first creates a functional architecture. We achieve this by prompting a LLM including chat history to create both architectures. For the functional architecture we use the following prompt as a template:

You are a systems architect. Create a hierarchical functional architecture of the following system: {system}. Each function in the tree should contain partial functions as children. Take the following rules into account: {rules}. Try to create at least {nrOfComponents} functions. The parent (source) to child (target) relationship is modeled by connectors. Consider the following list of requirements for references (ref): {requirements}. Use only the following JSON structure for the functional architecture tree: {structure}.

Each term in curly braces is replaced dynamically by the configuration from the user interface. Thereby we replace *{system}* with the system description and *{nrOfComponents}* with the given number from the interface. The *{requirements}* are given as a list in the format *[id] name: text*. Additionally, we provide a set of rules to follow for the creation of the functional architecture. This set of rules can be use-case specific and will replace the term {*rules*} during generation. They are stored in a separate "rules" file. The guidelines, recommendations, and architecture drivers from MBSE can be used to derive these rules. The response of the LLM shall be given in a certain form to simplify processing in Cameo Systems Modeler. For the functional architecture, we replace the term {*structure*} with the following Json template:

{"meta":{"name":"DIAGRAM_NAME"},"content":{"blocks":[{"id":"F-01","name":"FUNCTION_NAME","ref": ["REQUIREMENT_ID"]},{"id":"F-02","name": "FUNCTION_NAME", "ref": ["REQUIREMENT_ID"]}],"connectors": [{"id": "C-01","source": "F-01","target": "F-02"}]}}.

We use the structure to create a diagram named by the value of DIAGRAM_NAME. This diagram is filled with the elements, which were returned in the Json array blocks. For each element, we generate a block with the stereotype "Functional Block" and create connections according to the "connectors" Json array. Finally, we generate the tracelinks by creating relations from the block elements to the requirements given in the "ref" field for each block respectively.

Generating a logical architecture: We use the chat history to create the context for the generation of the logical architecture. Similarly, to the functional architecture, we use a prompt template and dynamically augment the missing information:

Create a hierarchical logical architecture of the following system: {system}. The logical architecture is a block definition diagram in form of a tree and contains exactly one root component, which models the system itself. Further partial components of the system are modeled as child components on multiple levels of the tree. Try to create at least {nrOfComponents} components. Take the following rules into account: {rules}. Consider the elements of the functional architecture for references (ref). The parent (source) to child (target) relationship is modeled by connectors. Use only the following JSON structure for the logical architecture tree: {structure}.

The terms *{system}* and *{nrOfComponents}* are replace by the same values from the user interface. The process to create a logical architecture differs from a functional architecture. Hence, we provide a new set of rules for the logical architecture generation. The Json structure in *{structure}* is slight adjusted for the logical architecture:

{"meta": {"name": "DIAGRAM_NAME"},"content":{"blocks": [{"id": "L-01","name": "COMPONENT_NAME", "ref": ["FUNCTION_ID"]},{"id": "L-02","name": "COMPONENT_NAME", "ref": ["FUNCTION_ID"]}],"connectors": [{"id": "C-01","source": "L-01","target": "L-02"}]}}.

Finally, we generate the blocks and connectors with the same procedure as for the functional architecture. Here, we use the stereotype "Logical Block" for the elements in the diagram.

4 Generation of Different Architectural Views of a RC Car

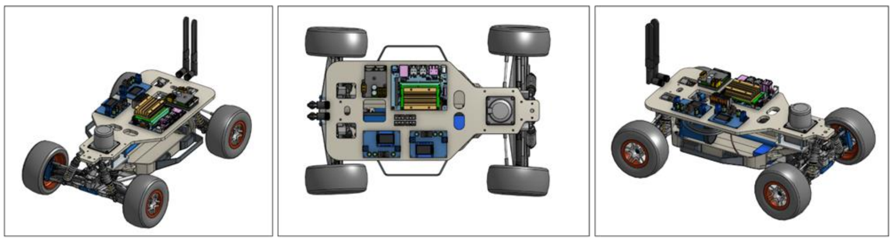


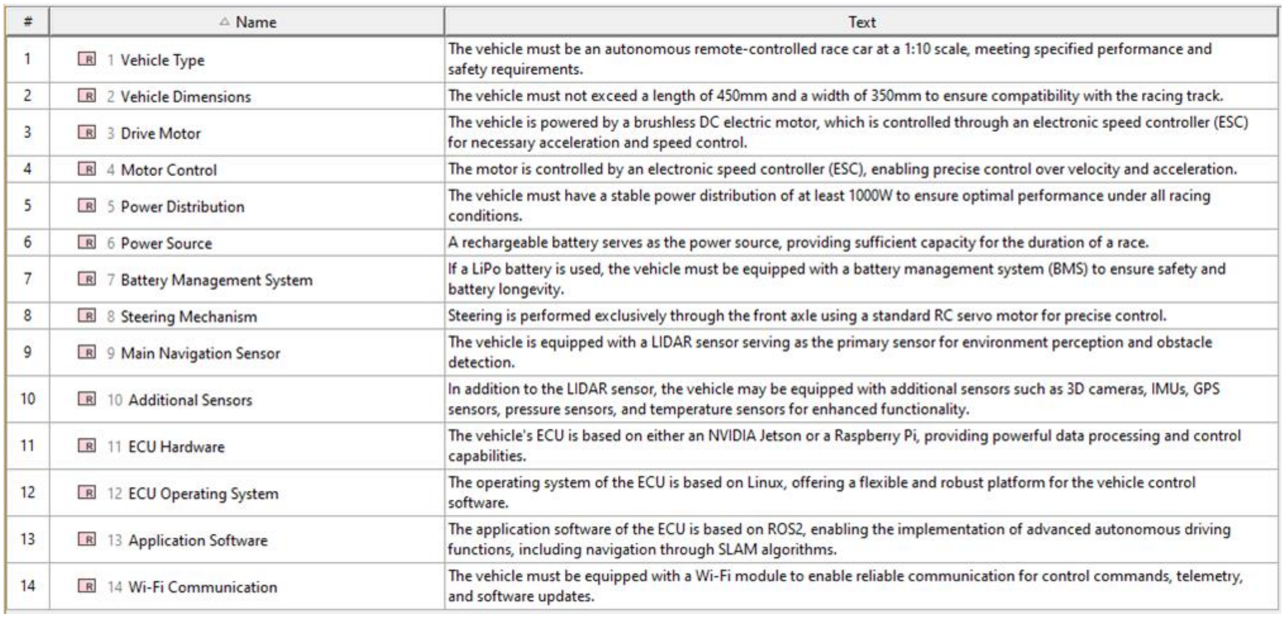
Figure 4. Fraunhofer IEM autonomous RC Car

In this updated study, we introduce a bespoke plugin for the Cameo System Modeler, demonstrated through a detailed case study of an autonomous remote-controlled car (RC Car) modeled on the MIT standard at a 1:10 scale. This case study integrates advanced components essential for autonomous driving, including a central computing unit (NVIDIA Jetson), a stereo camera system, a LIDAR system (Hokuyo UST-10LX), an accelerometer (Razor 9 DOF IMU), and an Electronic Speed Control. The MIT Race Car model is chosen for its practical application and evaluation of complex, real-world driving functionalities. The manageable number of components allows for an in-depth analysis of system interactions, bridging theoretical concepts with practical insights (Karaman et al., 2017).

We generate various architectural views of the RC Car using the plugin workflow described in Section 3.3. These views undergo rigorous evaluation based on established guidelines and predefined criteria. A pivotal aspect of our study involves inviting experts to assess the AI-generated architectures through a structured survey, aiming to evaluate their practical applicability against traditional, expert-developed architectures. This methodological approach not only benchmarks the AI-generated system architectures but also sheds light on the potential benefits and challenges of embedding AI in systems modeling. Figure 4 shows our RC Car. It clearly shows the arrangement of the electronic components such as the central computing unit, camera system, LIDAR system, the accelerometer, and the electronic speed control. The Right view shows a side perspective that providing a view of lower layer and to see the batterie system of the Car.

4.1 Importing Requirements to Cameo Systems Modeler & Tool Configuration

Importing system requirements into the Cameo System Modeler represents the initial steps in the process of modeling various architectural views. System requirements are typically captured in requirement management tools such as codebeamer, DOORS, or in Excel and form the foundation for generating architectural views. These requirements must be imported using existing solutions like the ReqIF importer, to be able to use the plugin. It is essential to recognize that importing into Cameo could potentially create two sources of the same artifact. To ensure consistency, it is advisable to implement regular data synchronization between the system requirements imported into Cameo and those maintained in the original requirement management tool. For the gathering of the requirements, we organized internal interviews with the RaceCar users. Figure 5 illustrates the system requirements imported into Cameo.



#	Name	Text
1	1 Vehicle Type	The vehicle must be an autonomous remote-controlled race car at a 1:10 scale, meeting specified performance and safety requirements.
2	2 Vehicle Dimensions	The vehicle must not exceed a length of 450mm and a width of 350mm to ensure compatibility with the racing track.
3	3 Drive Motor	The vehicle is powered by a brushless DC electric motor, which is controlled through an electronic speed controller (ESC) for necessary acceleration and speed control.
4	4 Motor Control	The motor is controlled by an electronic speed controller (ESC), enabling precise control over velocity and acceleration.
5	5 Power Distribution	The vehicle must have a stable power distribution of at least 1000W to ensure optimal performance under all racing conditions.
6	6 Power Source	A rechargeable battery serves as the power source, providing sufficient capacity for the duration of a race.
7	7 Battery Management System	If a LiPo battery is used, the vehicle must be equipped with a battery management system (BMS) to ensure safety and battery longevity.
8	8 Steering Mechanism	Steering is performed exclusively through the front axle using a standard RC servo motor for precise control.
9	9 Main Navigation Sensor	The vehicle is equipped with a LIDAR sensor serving as the primary sensor for environment perception and obstacle detection.
10	10 Additional Sensors	In addition to the LIDAR sensor, the vehicle may be equipped with additional sensors such as 3D cameras, IMUs, GPS sensors, pressure sensors, and temperature sensors for enhanced functionality.
11	11 ECU Hardware	The vehicle's ECU is based on either an NVIDIA Jetson or a Raspberry Pi, providing powerful data processing and control capabilities.
12	12 ECU Operating System	The operating system of the ECU is based on Linux, offering a flexible and robust platform for the vehicle control software.
13	13 Application Software	The application software of the ECU is based on ROS2, enabling the implementation of advanced autonomous driving functions, including navigation through SLAM algorithms.
14	14 Wi-Fi Communication	The vehicle must be equipped with a Wi-Fi module to enable reliable communication for control commands, telemetry, and software updates.

Figure 5. System Requirements of the RC Car

Proper tool configuration is necessary to create the architecture view. For our example, we use the standard ChatGPT-4 without finetuning. The configuration starts with the "System description," where we entered "An autonomous Remote-Control Car." All requirements from Figure 5 were selected, highlighting the importance of previous import steps. We adjusted the system complexity by selecting at least 10 components. A critical step is setting the temperature to 1, controlling the creativity and variability of AI modeling, directly influencing the generated architectural views. This setup is expected to produce comprehensive views that provide an initial idea and element of the system model.

4.2 Generation of a Functional Architecture

In this section, we generate a functional architecture from the imported system requirements. The architecture generation was initiated using the functional architecture prompt from section 3.3. To guide the accurate generation of the architectural view, additional information was provided to the language model. This is done using a separate JSON “rules” file.

In our approach, we used guidelines to generate the functional architecture, interpreting six key points. First, the Race Car's primary purpose should be the main node of the hierarchy, aligning all underlying functions towards a common goal. Second, functions should be systematically decomposed into manageable sub-functions. Third, the number of sub-functions should be kept manageable to avoid unnecessary complexity. Fourth, the nomenclature of elements should descriptively reflect the activity each function will undertake. Fifth, completeness is crucial, ensuring all necessary aspects of superior functions are covered by subordinate ones. Lastly, system interactions and interfaces should be considered, as a higher number increases complexity. Figure 6 illustrates the generated functional architecture.

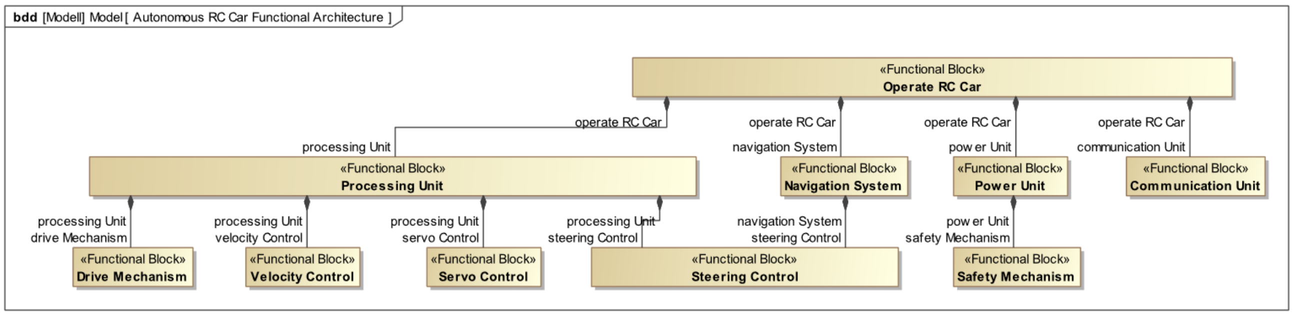
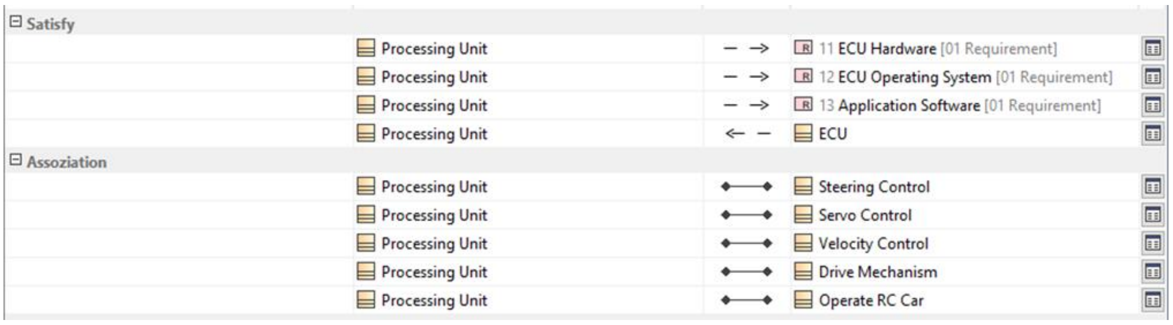


Figure 6. AI-Generated functional architecture

The generated architecture contains a total of 10 elements. Following the 1st guideline, the main node corresponds to the primary purpose of the system. In our case, it is “Operate RC Car”. On the 2nd level, there are 4 sub-functions to be found: the Processing Unit, the Navigation System (fulfill requirement 9 and 10), the Power Unit (fulfill requirement 5, 6 and 7), and the Communication Unit (fulfill requirement 14).

Figure 7 shows the traceability between the requirements (see Figure 2) and the functional architecture (see Figure 3). The Processing Unit, for example, is responsible for calculating, decision-making, and executing driving commands. This unit meets requirements 11, 12, and 13, which are associated with the ECU. Under the Processing Unit, there are 4 sub-functions: the Drive Mechanism (fulfill Requirement 3 and 4), Velocity Control (fulfill requirement 4), Servo Control (fulfill requirement 8). The Steering Control is assigned to both the Processing Unit and the Navigation System. The Power Unit has the Safety Mechanism on the 3rd level to meet requirement 7. Under the Communication Unit, no further function is subdivided.



⊟ Satisfy			
	Processing Unit	— →	11 ECU Hardware [01 Requirement]
	Processing Unit	— →	12 ECU Operating System [01 Requirement]
	Processing Unit	— →	13 Application Software [01 Requirement]
	Processing Unit	← —	ECU
⊟ Assoziation			
	Processing Unit	◆—◆	Steering Control
	Processing Unit	◆—◆	Servo Control
	Processing Unit	◆—◆	Velocity Control
	Processing Unit	◆—◆	Drive Mechanism
	Processing Unit	◆—◆	Operate RC Car

Figure 7. Traceability between requirement and functional architecture using processing unit as an example.

The transition from this functional architecture to the logical architecture will be addressed in the following chapter, where we will see how these functional blocks translate into specific components to fulfill the operational needs of the autonomous RC Car.

4.3 Generation of a Logical Architecture

Following the functional architecture laid out in Chapter 4.2, we now continue with the generation of the logical architecture. This architecture is built upon the functional structure and includes logical elements that fulfill one or more functions. The architecture generation was initiated using the logical architecture prompt from section 3.3.

Guidelines were also provided for the creation of the logical view in the “rules” file. We have adopted a total of 6 guidelines. Firstly, we have the direct derivation of logical elements from the functions, where each logical element should realize at least one function. The second guideline is modularity, suggesting that the architecture should be partitioned so that the logical elements are clearly and definitively defined. This enhances the testability and reusability of the logical elements and allows for the development of clearly defined logical elements to be taken over externally. Next, we address the hierarchy and decomposition of the individual elements, which should follow a systematic breakdown, beginning with the overarching elements that capture the important system functions and accordingly subdividing these into subordinate elements. The fourth guideline relates to the depth of the hierarchy. The logical architecture should have 3 to 4 levels for our application to maintain clarity and manageability of the system. The manageability is additionally influenced by the interfaces in the system; hence this is adopted as the fifth guideline. Lastly, scalability and flexibility are established, ensuring that the architecture is adaptable to new or changed requirements or technology without extensive rework. Figure 8 illustrates the generated logical architecture.

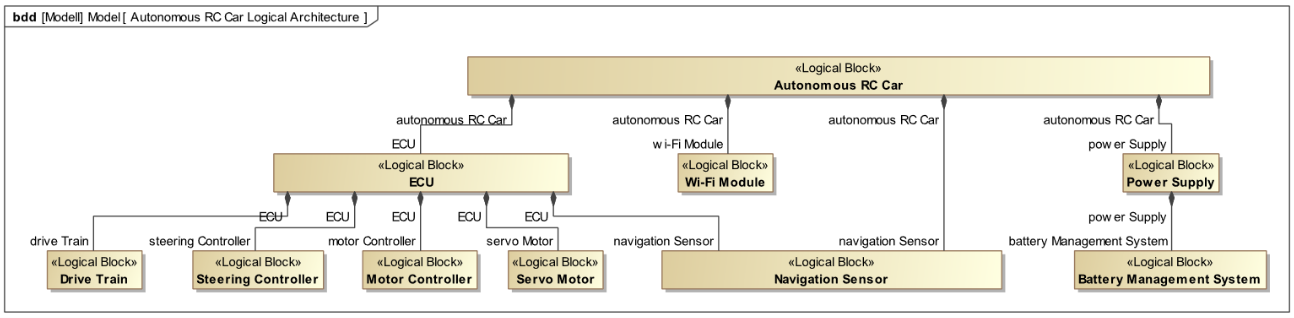
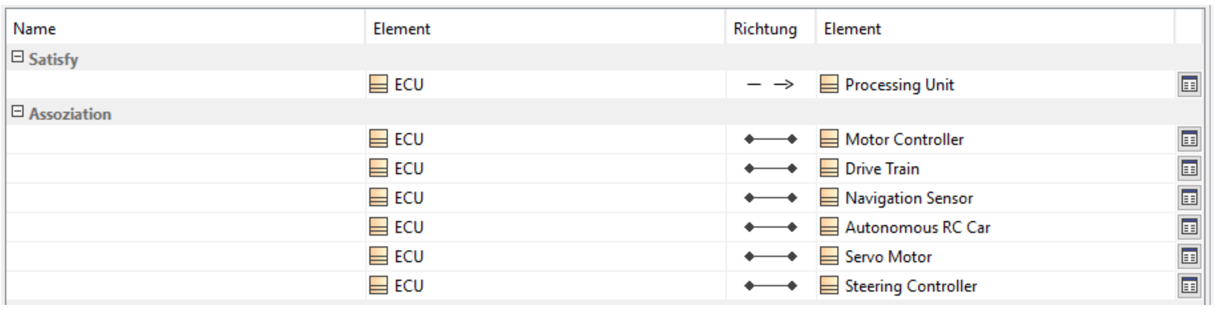

bdd [Modell] Model [Autonomous RC Car Logical Architecture]
«Logical Block» Autonomous RC Car
autonomous RC Car
ECU
«Logical Block» ECU
autonomous RC Car
wi-Fi Module
«Logical Block» Wi-Fi Module
autonomous RC Car
autonomous RC Car
power Supply
«Logical Block» Power Supply
ECU
ECU
ECU
ECU
ECU
power Supply
drive Train
steering Controller
motor Controller
servo Motor
navigation Sensor
navigation Sensor
battery Management System
«Logical Block» Drive Train
«Logical Block» Steering Controller
«Logical Block» Motor Controller
«Logical Block» Servo Motor
«Logical Block» Navigation Sensor
«Logical Block» Battery Management System

Figure 8. AI-Generated Logical Architecture

The generated architecture consists of 3 levels and contains 10 logical elements. The main node is the "autonomous RC Car" system. The 2nd level consists of 3 logical elements: an ECU, Wi-Fi Module, and Power Supply. In this case, the ECU serves as the processing unit of RC Car. As seen in Figure 6, the logical element is linked with the functional element Processing Unit. The ECU itself is comprised of 5 elements: Motor Controller, Steering Controller, Drive Train, Servo Motor, and Navigation Sensor.



Name	Element	Richtung	Element
⊟ Satisfy			
	ECU	⟶	Processing Unit
⊟ Assoziation			
	ECU	◆—◆	Motor Controller
	ECU	◆—◆	Drive Train
	ECU	◆—◆	Navigation Sensor
	ECU	◆—◆	Autonomous RC Car
	ECU	◆—◆	Servo Motor
	ECU	◆—◆	Steering Controller

Figure 9. Traceability between logical Elements and functional element using ECU as an example.

The Motor Controller assumes control of the motor through velocity control and is hence linked with the function Velocity Control. The "Drive Train" is responsible for energy transmission and is associated with the function "Drive Mechanism." The Steering Controller, Servo Motor, and Navigation Sensor are each connected to the functions Steering Control, Servo Control, and Navigation System, respectively. Alongside the ECU on the 2nd level is the Power Supply, which represents the Power Unit. This includes the Battery Management System, where the Safety Mechanism is realized. Lastly, on the 2nd level is the Wi-Fi Module, corresponding to the Communication Unit.

4.4 Evaluation of the Generated Architectures

The evaluation is an initial test to gather expert feedback. The paper's title shows that the solution and its implementation are not finished. The following subchapter explains how feedback is collected, which helps assess the designs. The evaluation uses a questionnaire completed by potential users. It should include a mix of quantitative and qualitative questions to gather diverse data. It should adhere to ethical standards such as informed consent and participant anonymity. The questionnaire should be distributed to a representative sample of users via accessible platforms to maximize response rates. Data analysis should employ statistical tools for quantitative responses and thematic analysis for qualitative feedback. Once the findings are in, they should be used to evaluate the software and identify improvements. (John W. Creswell & J. David Creswell, 2018)

Our expert panel includes individuals with extensive experience in System Engineering, particularly in developing the IEM RC Car, a versatile system used as a demonstrative model. Our evaluation method uses a structured questionnaire with three response options per query, comprising ten inquiries grouped into four sections. The first two sections gather opinions on the functional and logical architectures, each with three questions. The third section includes two queries about the innovation of the generated components. Finally, we conduct a holistic assessment of the architectural perspectives and their added value, using the architectural drivers from the "rules". Criteria include assessing component suitability, coherence, complexity, traceability, added value, and innovation potential. Table 1 summarizes the expert opinions, listing evaluation criteria in the first column and the percentage of experts who believe each criterion is fully, partially, or not met in subsequent columns.

Table 1. Expert evaluation result

No.	Evaluation Criterion	Full Met	Partially Met	Not Met
1	Components of the functional architecture	20,0%	40,0%	40,0%
2	Complexity of the functional architecture	0,0%	60,0%	40,0%
3	Traceability (requirement to functions)	20,0%	40,0%	40,0%
4	Components of the logical architecture	0,0%	100,0%	0,0%
5	Traceability (functions to logical elements)	20,0%	80,0%	0,0%
6	Added value of the logical architecture	40,0%	20,0%	40,0%
7	Innovation	20,0%	0,0%	80,0%
8	Futureproofing	20,0 %	80,0 %	0 %
9	Overall Architecture Assessment	0,0 %	60,0 %	40,0 %

The ensuing discussion aims to elaborate on various aspects of the evaluation outcomes. Compared to the components of the logical architecture (Criterion No. 4), where 100% of the experts believe that most logical elements in the architecture have been identified and included, opinions on the functional architecture are divided (Criterion No. 1). 20% of experts believe that the architecture has fully generated suitable functions and sub-functions. Meanwhile, 40% each believe that the architecture lacks essential aspects (partially met) or even has significant deficiencies (not met). Moreover, there are duplicate affiliations in the architectural perspectives, such as with Navigation Sensors in the logical and steering control in the functional architecture. This could pose integration or accountability difficulties during development.

Regarding whether the logical architecture adds value (No. 6), expert opinions are also divided. Among those who voted "not met," some believe that the logical architecture largely reflects the functional structure or that some elements have been incorrectly incorporated, such as the Drive Train example, or are missing, such as sensors like LiDAR. In terms of Traceability (Criterion Nos. 3 and 5), 80% of experts believe this is not fully met. The degree of innovation (Criterion No. 7) was rated as not met by 80%. This means that the generated elements were mainly known or expected. Regarding futureproofing with respect to emerging technologies, the majority rated it as partially met. This means that the architecture considers future technological developments to some extent but is still improvable. The final evaluation point in the table (Criterion No. 9) is the overall assessment of the architecture. 60% of experts consider it "good." This means there is good design quality, but some areas could be improved. The usefulness of the architecture was asked as the last question. It aims to understand how helpful such a generated architecture would be for the system architect (Table *2*). 80% of experts agree that the architecture proposal is moderately helpful. This means that the architecture can be used as inspiration or a first draft. The proposed architectures are thus not directly implementable, yet they contain valuable ideas and concepts that can serve as a starting point for further consideration.

Table 2. Utility Assessment of the architecture proposal

Nr.	Utility	Very Helpful	Moderate Helpful	Little/No Helpful
10	Architecture Proposal	20,0%	80,0%	0,0%

5 Conclusion and Future Work

5.1 Conclusion

In this paper, we introduced a concept and implementation of system architecture creation tool. This tool was integrated as a plugin in the popular engineering tool Cameo Systems Modeler, which allows easy usage. This plugin allows hence a process to ease the construction of new system architectures by providing a first draft. A validation on the Fraunhofer RaceCar provides a proof of concept for the plugin workflow. The following expert evaluation could provide valuable insights for further improvement and underscore the relevance of the process. The proposed solution is limited and represents a rapidly developed initial solution, which is without a doubt not final nor flawless, but it highlights the potential that is attached to leveraging uprising technology of generative AI in engineering applications.

5.2 Future Work

The AI4Cameo plugin already covers some basic architecture generation but opens the possibility for many more extensions.

Our objective is to refine the solution methodology by creating additional system views, such as sequence or activity diagrams, and improving the review process through tracelink reports that identify unlinked requirements. We aim to further improve the methodical enhancement of the solution by introducing a step-by-step approach with the large language model, which should improve the design reasoning. To ensure a higher trustworthiness we aim to improve the traceability, which is a crucial property. Especially in the engineering industry traceability is essential to ensure the safety and security of products. In technical terms, an intermediate step between the functional and logical architecture would provide more usability. Especially, a rerun button would allow for multiple specific architecture designs to choose from. Retrieval-augmented generation (RAG) or finetuning can contain additional information and examples to improve the understanding of MBSE for large language models.

References

Baughey, K., 2011. Functional and Logical Structures: A Systems Engineering Approach, in: SAE Technical Paper Series. SAE 2011 World Congress & Exhibition. APR. 12, 2011. SAE International400 Commonwealth Drive, Warrendale, PA, United States.

Chen, W., Yan-yi, L., Tie-zheng, G., Da-peng, L., Tao, H., Zhi, L., Qing-wen, Y., Hui-han, W., Ying-you, W., 2024. Systems engineering issues for industry applications of large language model. Applied Soft Computing 151, 111165. https://doi.org/10.1016/j.asoc.2023.111165.

Chung, L., do Prado Leite, J.C.S., 2009. On Non-Functional Requirements in Software Engineering, in: Borgida, A.T., Chaudhri, V.K., Giorgini, P., Yu, E.S. (Eds.), Conceptual Modeling: Foundations and Applications, vol. 5600. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 363–379.

Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., Zhang, J.M., 2023. Large Language Models for Software Engineering: Survey and Open Problems, 23 pp. http://arxiv.org/pdf/2310.03533v4.

Göpfert, J., Weinand, J.M., Kuckertz, P., Stolten, D., 2023. Opportunities for Large Language Models and Discourse in Engineering Design, 15 pp. https://arxiv.org/pdf/2306.09169.pdf (accessed 2 February 2024).

Gotel, O., Cleland-Huang, J., Hayes, J.H., Zisman, A., Egyed, A., Grünbacher, P., Dekhtyar, A., Antoniol, G., Maletic, J., Mäder, P., 2012. Traceability Fundamentals, in: Cleland-Huang, J., Gotel, O., Zisman, A. (Eds.), Software and Systems Traceability. Springer London, London, pp. 3–22.

Jiang, A.Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D.S., Casas, D.d.l., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Lavaud, L.R., Lachaux, M.-A., Stock, P., Le Scao, T., Lavril, T., Wang, T., Lacroix, T., Sayed, W.E., 2023. Mistral 7B, 9 pp. https://arxiv.org/pdf/2310.06825.pdf (accessed 2 February 2024).

John W. Creswell & J. David Creswell, 2018. Research Design: Qualitative, Quantitative, and Mixed Methods Approaches.

Joseph D'Ambrosio, G.S., 2017. 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC): Banff Center, Banff, Canada, October 5-8, 2017. IEEE, Piscataway, NJ, 1 p.

Kaiser, L., Bremer, C., Dumitrescu, R., 2016. Exhaustiveness of Systems Structures in Model-based Systems Engineering for Mechatronic Systems. Procedia Technology 26, 428–435. https://doi.org/10.1016/j.protcy.2016.08.055.

Karaman, S., Anders, A., Boulet, M., Connor, J., Gregson, K., Guerra, W., Guldner, O., Mohamoud, M., Plancher, B., Shin, R., Vivilecchia, J., 2017. Project-based, collaborative, algorithmic robotics for high school students: Programming self-driving race cars at MIT, in: 2017 IEEE Integrated STEM Education Conference (ISEC). 2017 IEEE Integrated STEM Education Conference (ISEC), Princeton, NJ, USA. 11.03.2017 - 11.03.2017. IEEE, pp. 195–203.

Kharatyan, A., Günther, M., Anacker, H., Japs, S., Dumitrescu, R., 2022. Security- and Safety-Driven Functional Architecture Development Exemplified by Automotive Systems Engineering. Procedia CIRP 109, 586–591. https://doi.org/10.1016/j.procir.2022.05.299.

Madni, A., Sievers, M., 2018. Model-based systems engineering: Motivation, current status, and research opportunities. Systems Engineering. https://doi.org/10.1002/sys.21438.

McCarthy, J., Minsky, M.L., Rochester, N., Shannon, C.E., 1955. A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence.

Müller, J.R., Isaksson, O., Landahl, J., Raja, V., Panarotto, M., Levandowski, C., Raudberget, D., 2019. Enhanced function-means modeling supporting design space exploration. AIEDAM 33, 502–516. https://doi.org/10.1017/S0890060419000271.

Rueden, L.v., Mayer, S., Beckh, K., Georgiev, B., Giesselbach, S., Heese, R., Kirsch, B., Walczak, M., Pfrommer, J., Pick, A., Ramamurthy, R., Garcke, J., Bauckhage, C., Schuecker, J., 2021. Informed Machine Learning - A Taxonomy and Survey of Integrating Prior Knowledge into Learning Systems. IEEE Trans. Knowl. Data Eng. 17, 1. https://doi.org/10.1109/TKDE.2021.3079836.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., Lample, G., 2023. LLaMA: Open and Efficient Foundation Language Models, 27 pp. http://arxiv.org/pdf/2302.13971v1.

Vanommeslaeghe, Y., Denil, J., Viaene, J. de, Ceulemans, D., Derammelaere, S., Meulenaere, P. de, 2019. Leveraging Domain Knowledge for the Efficient Design-Space Exploration of Advanced Cyber-Physical Systems, in: 2019 22nd Euromicro Conference on Digital System Design (DSD). 2019 22nd Euromicro Conference on Digital System Design (DSD), Kallithea, Greece. 28.08.2019 - 30.08.2019. IEEE, pp. 351–358.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I., 2017. Attention Is All You Need, 15 pp. https://arxiv.org/pdf/1706.03762.pdf (accessed 2 February 2024).

Walden, D.D., Roedler, G.J., Forsberg, K.J., Hamelin, R.D., Shortell, T.M., 2003. INCOSE Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities.

Contact: Oliver von Heißen, Fraunhofer IEM, Digital Engineering, Zukunftsmeile 1, Paderborn, Germany, +49 5251 5465172, oliver.von.heissen@iem.fraunhofer.de