TUM

Synergy of Large Language Model and Model Driven Engineering for Automated Development of Centralized Vehicular Systems

Nenad Petrovic, Fengjunjie Pan, Krzysztof Lebioda, Vahid Zolfaghari, Sven Kirchner, Nils Purschke, Muhammad Aqib Khan, Viktor Vorobev, Alois Knoll

Technischer Bericht

Technische Universität München
Institut für Informatik

Synergy of Large Language Model and Model Driven Engineering for Automated Development of Centralized Vehicular Systems

March 25, 2024

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

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

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

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

Sven Kirchner*
email: sven.kirchner@tum.de
orcid: 0009-0004-3845-6772

Nils Purschke*
email: nils.purschke@tum.de
orcid: 0009-0008-9470-5795

Muhammad Aqib Khan*
email: aqib.khan@tum.de
orcid: 0000-0002-1082-253X

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

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 present a prototype of a tool leveraging the synergy of model driven engineering (MDE) and Large Language Models (LLM) for the purpose of software development process automation in the automotive industry. In this approach, the user-provided input is free form textual requirements, which are first translated to Ecore model instance representation using an LLM, which is afterwards checked for consistency using Object Constraint Language (OCL) rules. After successful consistency check, the model instance is fed as input to another LLM for the purpose of code generation. The generated code is evaluated in a simulated environment using CARLA simulator connected to an example centralized vehicle architecture, in an emergency brake scenario.

I. INTRODUCTION

The rise of large language models (LLMs), initiated by popularity of the now well-known ChatGPT towards the end of 2022, has affected almost every aspect of our everyday life – from simple customer services and question answering to entertainment and science. Apart from its innovation, the main drive behind LLM adoption in various areas working field and scientific areas is to either automatize repetitive tasks which are traditionally done by humans, or help them achieve improved outcomes by providing suggestions or hints. In area of computer science and software engineering the usage of LLMs is identified for several distinct purposes in literature [1], [2]:

1) **code generation** – automatically generating of either boilerplate code or full applications, algorithms and other useful code scripts (tests, configuration files);

2) **code augmentation** – adding the missing parts to the code or additional functions/classes to achieve the given goal;

3) **code analysis** – providing suggestions how the provided source code can be improved in order to avoid well-known bugs, security vulnerabilities and other code-related issues which are identifiable in design-time;

4) **code explanation** – LLM is able to provide quite verbose comments which correspond to specific parts of the code, explaining its purpose and usage;

5) **data analysis** – it was also shown that LLMs trained on enormous amount of textual data are also able to mimic the traditional machine learning algorithms and perform supervised learning tasks, such as classification.

On the other side, when it comes to automotive industry, software is the crucial aspect which shapes the innovation in this area. However, traditional automotive architectures rely on hundreds of heterogeneous devices (electronic control units – ECUs), provided by different OEMs, using distinct architectures, OS and software platforms. Therefore, software development and its maintenance in automotive area becomes increasingly difficult, taking into account the demand for additional functionalities (such as infotainment) and advanced automated control scenarios (such as autonomous driving), which usually involve the usage of even more devices, such as sensors, actuators and specific co-processors. The latest concept of centralized automotive architecture enables a flexible software-defined vehicle.

Therefore, in this paper, we propose the means for LLM-

aided design and developments of centralized car systems, starting from high-level feature specification. Apart from LLMs, our proposed framework relies on model-driven software engineering principles. Utilization of formal specifications – such as metamodels and constraint rules – enables verification and validation of LLM outcomes, which are, due to their nature, prone to the so-called hallucinations, where the generated output is incorrect. Such unreliability can be fatal in domains such as automotive. Therefore, we propose design-time, formal verification of generated results. In synergy with user supervision, and manual intervention and correction where needed, it should largely mitigate the cons of LLM usage.

In this work, LLMs will be adopted to tackle the following aspects:

1) requirements extraction and summarization – generating formal list of verifiable requirements based on free-form ISO standard or reference architecture text;
2) automated design – creating verifiable car system model instances based on user requirements;
3) code generation – automated parametrization of deployment configuration templates.

II. MOTIVATION BEHIND CENTRALIZED ARCHITECTURE

The main goal of a centralized car architecture is to provide the so-called "single-system illusion", which has many advantages compared to other state-of-art solutions [3], [4].

Lower Hardware Costs: Centralized systems consolidate multiple control functions into powerful processors, significantly reducing the need for numerous individual control units with separate processors, memory, and electronics. This consolidation can lead to economies of scale in purchasing, resulting in lower overall hardware costs. Additionally, the amount of computing power can be minimized through efficient scheduling of the different functionalities on a centralized system. In contrast, in distributed systems redundant computing hardware exists since there is a need to process data on different hardware components that are often not shared across the system, especially if components from multiple manufacturers are involved. This leads to parts of the hardware being idle instead of working on other processing tasks.

Improved Energy Efficiency: By minimizing the number of control units and optimizing computing resources, centralized architectures can decrease vehicle's energy consumption. This efficiency is crucial for electric vehicles, where energy usage directly affects range and performance. Since there is only the central unit that requires power for data processing, it is much easier to monitor and influence it.

High Application-Level Communication Speed: Centralization allows for faster communication speeds at the application level, as data does not have to traverse multiple nodes (control units) before reaching its destination. Instead, the applications run on the same computational hardware and can directly access any data without the need to communicate with external components.

Simplicity in Application Development: Writing applications becomes simpler in a centralized system, because developers only need to target a single computing environment, streamlining the development process. All applications can be unified in terms of inter-application-communication, sensor and actuator abstraction through the use of an underlying hardware abstraction layer, isolation mechanisms like containers, scheduling systems and so on. This vastly simplifies the software development and makes it largely hardware independent.

Simplified Software Failure detection: Since all software runs on a single system, it is much easier to identify occurring errors. All systems can be monitored on a software level instead of the need to debug the communication on the wire. What is more, the amount of proprietary, black box ECUs from various manufacturers is minimized. Uniform runtime environment simplifies error handling since common low level error detection routines can be used, which monitor all applications and handle errors on the same platform using unified error handling mechanisms, without the external communication overhead.

Full System Control: A centralized architecture ensures that all vehicle data and processing capabilities are centralized, providing comprehensive control over the vehicle's functionality through software. The amount of private intellectual property, including custom hardware and software, in the vehicle is reduced. Therefore, many software components and functionality can be adapted by the car manufacturer itself, without the need of help from external suppliers.

Simplified Wiring and Connectivity: Direct connections to the central computational unit or straightforward data tunneling through network switches eliminate the complex wiring and communication layouts, characteristic to distributed systems.

Facilitates Software-Defined Vehicle Concepts: Centralizing data and processing simplifies the transition towards software-defined vehicles, where software, rather than hardware, defines vehicle functionality. This centralization supports easy software updates and upgrades, feature improvements, and addressing safety issues.

Ease of Hardware Upgrades: If designed with hardware extensibility in mind, centralized systems simplify the addition of new hardware, contrasting with the challenges of integrating new components in hardware-heavy distributed systems, where the hardware is often hardwired with the rest of the system and permanently installed on the chassis, meaning that any hardware update or upgrade is a complex task. Additionally, when exchanging components of a distributed system, the compatibility with the rest of the system must be ensured, which may be difficult if the components were heavily interconnected. A centralized hardware built on top of proper hardware abstraction mechanisms is largely unchallenged by those problems.

III. LLM-ENABLED WORKFLOW

The proposed workflow leveraging synergy of LLMs and model-driven engineering is depicted in Fig. 1.

In the first step, a domain expert enters the requirements in free-form natural language into the given form with pre-defined fields. The arguments from this form are forwarded, together with the predefined Ecore metamodel, to a prompt constructor (described later). The prompt for the LLM is constructed using the following template:

Prompt1 – Model instance creation: Create XMI model instance for [Requirements input] and [Ecore metamodel]

As outcome of the process, an XMI model instance depicting the requirements with respect to the Ecore metamodel is generated.

For the purpose of requirements consistency verification, OCL rules [5] based on a reference architecture and the metamodel are extracted as well:

Prompt2 – Constraints extraction: Based on reference requirements [Reference requirements or part of ISO standard document] and metamodel [Ecore metamodel] extract OCL constraints

Once the instance model is verified, resource allocation step is preformed, which assigns car features to corresponding chips available to the central car server. The outcome of this process is an allocation matrix giving the information on how features and their corresponding containers will be mapped to the hardware.

The created model instance is leveraged for code generation, either using conventional rule-based solutions or an LLM. There are several possible steps where model instance can be used for code generation: parametrizing pre-defined components (CARLA excerpts [6]), containerizing components, generating deployment descriptors. The corresponding parameters are extracted from model instance and inserted into prompt templates, or the whole metamodel/model instance is used.

Prompt3 – Parametrizing components: Based on Ecore instance [Ecore instance model] fill in the template [CARLA excerpt]

The goal of this prompt is to leverage parameters from Ecore model instance, describing car component characteristics and add insert them CARLA code excerpt.

Prompt4 – Containerizing components: Write Docker-file to containerize script [script name] written in [Language/Technology used] using dependencies [Libraries and so on]

Prompt5 – Automated generation of adapters between components: Generate Python code to convert [source format] to [target format]

Prompt6–Deployment descriptor generation: Generate [Docker Compose or Kubernetes descriptor] to run containers: [For each container in ZoneController.containers:] [Container.image] on ZoneController open port [container.communicationPort] with environment variables [Container->Sensor/Acutator properties] based on model instance [Ecore model instance]

Finally, the generated Docker Compose file or Kubernetes descriptor is deployed and run on Central Car Server. The corresponding Docker images (Python scripts relying on CARLA client library in our case) are assumed to be available within local or custom (remote) Docker image repository. Customization of these docker images is performed by parametrizing the corresponding Docker Compose environment variables, based on sensor/actuator characteristics.

The whole code generation workflow from user requirements to CARLA code is described in form of pseudocode in 1.

Data: $requirement, metamodel, standard$
Result: $CARLAcode$
$xmi \leftarrow$
$\quad executePrompt(prompt1, requirement, metamodel);$

$ocl \leftarrow$
$\quad executePrompt(prompt2, standard, metamodel);$
$pass \leftarrow verify(ocl, xmi, metamodel);$
if $pass$ is true **then**
$\quad | \quad CARLAcode \leftarrow executePrompt(prompt6, xmi);$
else
$\quad | \quad CARLAcode \leftarrow null;$
$\quad | \quad print(Requirement not compliant with standard)$
end

Algorithm 1: Code generation

Additional details and whole vision of full LLM integration, including user feedback steps are described in [7]. However, they are not included in the presented prototype implementation, but planned to be included in the final version of the tool.

IV. CENTRALIZED CAR SERVER METAMODEL

Centralized Car Server metamodel covers both the hardware and software components, together with aspects of functional (FR) and non-functional (NF) requirements. Hierarchical overview of aspects covered by metamodel is given in Fig. 2.

As it can be seen, on the top level, metamodel covers the use cases/scenarios in automotive as a concept, such as highway pilot. These scenarios are further broken down into features, which are mapped to the actual software components, represented as Docker containers in our case. On the other side, when it comes to resource allocation, there are two different perspectives involved: 1) Software component allocation - how features are mapped to containers 2) Hardware component allocation - how software components are mapped to set of available hardware resources. For the first aspect, the crucial elements are functional requirements which determine the set of features that will be taken into account. On the other side, when it comes to aspects of hardware allocation, non-functional requirements, such as costs, power consumption, processing power and others are considered.

This metamodel is based on the idea that Central Car Server-controlled car contains several *ZoneControllers*, where each
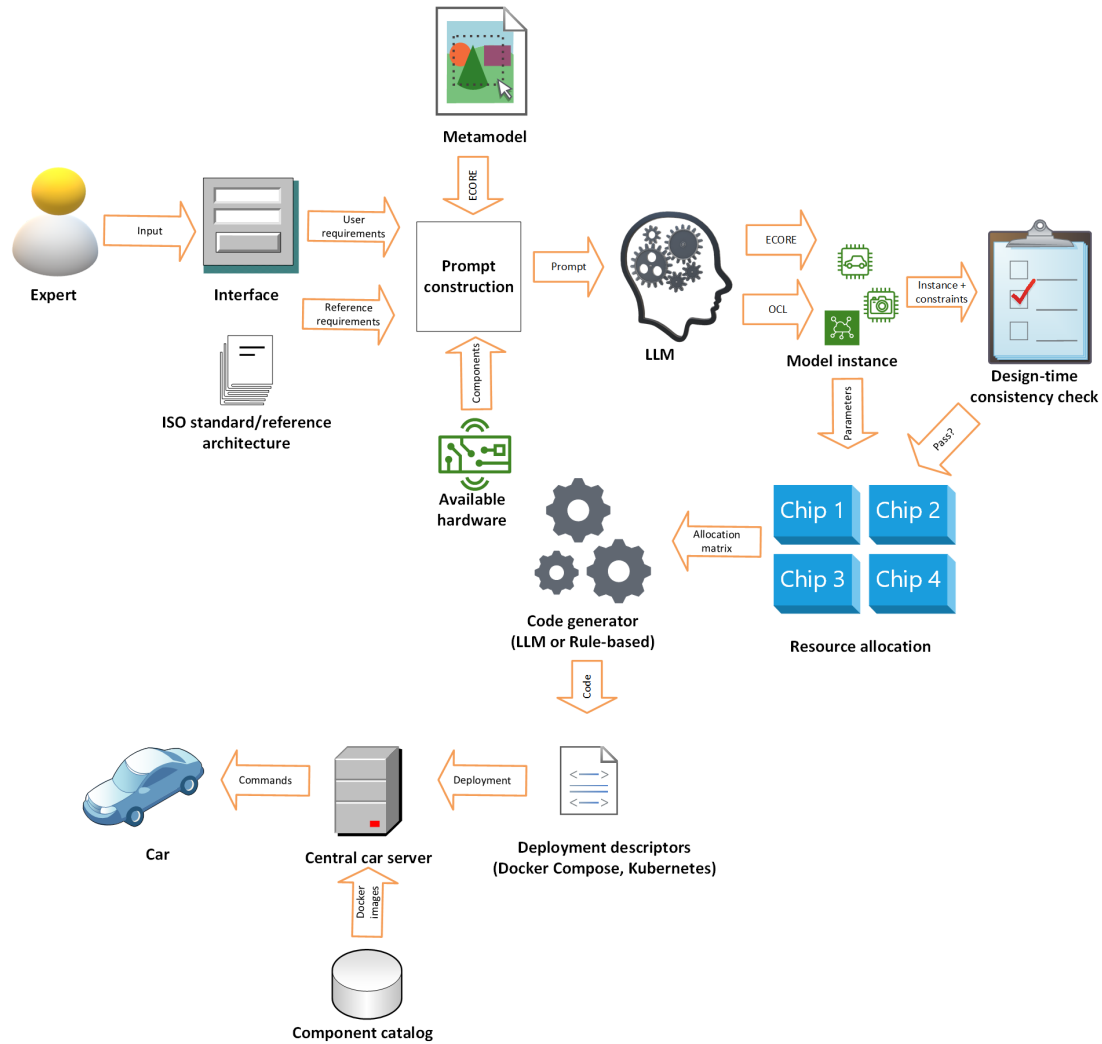
Fig. 1. LLM-enabled workflow for model-driven automotive development.

of them is responsible for distinct part of car (like front, rear, side etc). However, from the developer's perspective, the architecture is designed to provide illusion of unified, centralized system. ZoneController could be either an isolated virtual machine running within Central Car Server (therefore, it is important to take into account which VM platform and image are used) or physically distinct application-specific chip - Co-Processor (such as Infineon's solution).

ProcessingNode represents element of metamodel that generalizes components which are able to execute processing tasks. For such elements, the following aspects are relevant and taken into account:

- memory capacity
- processing power
- number of cores
- maximum bandwidth/data rate
- architecture - ARM/x86 etc.
- realtimeCapability - whether the processing node has support for realtime execution

Within the system's scope, we identify two types of processing node components: Master and Slave. While slave components are usually single, dependent chips (required to be connected to a master) - such as GPU, TPU and FPGA, master components represent more general boards with additional components which can act independently, attach/detach slave devices and schedule tasks for execution. In that sense, ZoneControllers are subclass of master processing nodes.

To each ZoneController, a set of Components (either Hardware or Software) is assigned in order to implement particular usage Scenario. Moreover, scenario consists of one or many Features. Each of the Features is derived from free-form textual specification coming from Requirements Specification. Moreover, a feature can be safety-critical, which is reflected in container redundancy and other strategies adopted to ensure safety compliance by the components. When it comes to hardware components there are three identified types: Co-Processors, Sensors and Actuators. Co-Processor represents slave ProcessingNodes, such as GPU, TPU, FPGA which
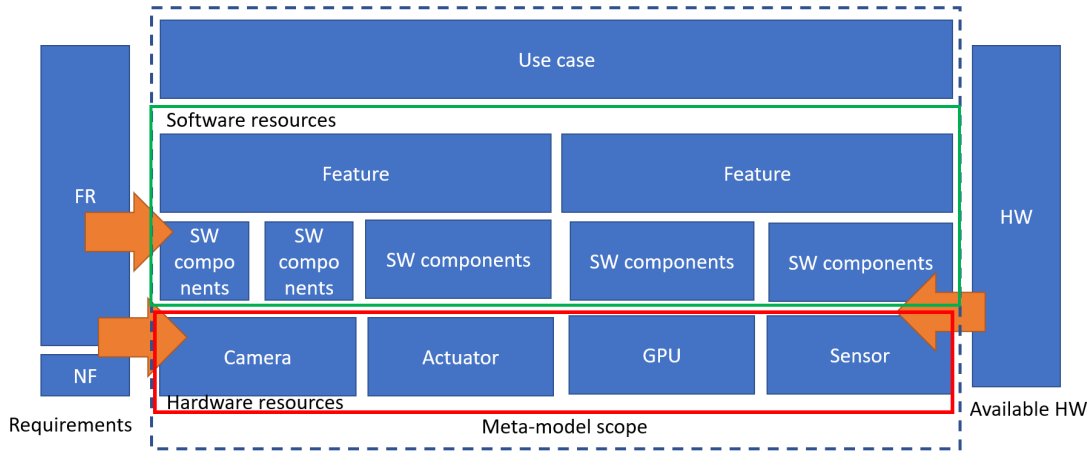
Fig. 2. Hierarhical representation of Centralized Car Server Metamodel.

are designed to execute some specific tasks with high performance, such as object detection and sensor data analysis. Sensors cover commonly used sensing devices in vehicular systems, such as Camera, Lidar, Ultra-violet and others. Role of these devices is to record quantitative measurements representing some real-phenomenon.

For *Sensors*, the following aspects are relevant:

- measurementUnit - such as pixels, intensity
- measurementFormat - RGB bitmap, CSV data and other
- measurementsPerSecond- how frequently the data is measured
- parameterList - list of sensor-specific parameters, such as Camera's fov angle, image Height and Width, LIDAR's number of channels etc.

On the other side, for *Actuator*, relevant aspects are:

- commandFormat - such as MQTT JSON message or some other specific protocol
- parameters - list of command parameters, such as acceleration amount, steering angle

Additionally, to each of the Sensors/Actuators, corresponding software container Controller is assigned in runtime, which is run within the isolated environment on the ZoneController. This container is responsible for enabling the function of sensor/actuator device. For sensors, it can start sensing, performing reading sensor measurements and stop sensing. For actuators, it can turn the device on or off and issue commands to them.

When it comes to *Software* resources, they have the following common properties:

- memoryDemand - estimation how much memory is required on the assigned ProcessingNode to run it
- processingDemand - estimated processing power required
- bandwidthDemand - how much data is exchanged or consumed by the software component

- realtimeRequired - denotes whether the software components require realtime capabilities

The main unit of software abstraction is *ApplicationContainer*. It represents an isolated environment where some specific processing task or sensor/actuator controller is executed. Container has the following properties:

- image - name of the image which is used as base for creation of container
- targetTechnology - which kind of underlying containerization/orchestration engine is used, such as Docker, Docker Compose or Kubernetes engine
- repository - URL of the corresponding online repository, private container registry or automotive vendor software catalogs where resources required for running the container can be accessed
- script [optional] - exact name of the script which will be run inside the container, useful only for additional capabilities of automated script containerization
- dependencies [optional] - list of libraries and runtime environments (such as Python) used by container, also useful only in case of automated containerization
- architecture - for which processor architecture was the container built for, such as ARM/x86
- communicationPorts - list of ports which are used for communication with this container, so they have to be exposed in order to access them from other components, such as port 2000 in case of CARLA simulator-based components

Additionally, one more type of Software components are *ProcessingTasks*. They represent either data analysis method implementaton (image or LIDAR-measurement based object detection) or control algorithms. ProcessingTasks usually take data as input from either Sensor element or another ProcessingTask. For each of them, the following aspects need to be taken into account:

- inputs - list of aspects which are taken as inputs of the performed computation/processing/transforamtion. For example, it is "image" for object detection.
- outputs - what is generated as outcome of the performed transformation, such as steering or brake command in case of car control algorithm
- inputFormat - list of formats corresponding to each of the inputs specified, such as "RGB" or "CMYK" for image
- outputFormat - list of formats for the products generated
- compatibleFormat - list of compatible formats, recognized by task as proper input. In case that provided inputs are not in corresponding format, adoption of adapter methods between the components would be needed

Regarding the abstract interface of these components, they perform processing which takes into account inputs, inputs formats and output formats, while the result of the transformation outcome is output. There are various possibilities for input and output combinations:

- Image based object detection - inputs are images, outputs is a number of objects detected
- Control algorithm - input is a number of obstacles, outputs are acceleration, steering angle or brake activation
- ConnectionLink elements are used for representation of relationships - either virtual or physical connections between two components, considering the protocol, type of connection and latency.

The aspects of power consumption, together with costs by each of the hardware components (co-processors, sensors and actuators) can be taken into account as well, especially in the design phase when the decision regarding the selection of corresponding parts have to be taken into account by non-functional requirements-aware optimization-based resource allocation mechanisms.

V. COMPONENT INTERFACES

This section gives an overview of high-level, abstract interfaces characteristic for various types of system component interfaces.

`ProcessingNode`
- `scheduleTask(containerId)` – processing nodes should be able to execute the assigned computational tasks
- `freeResources (containerId)` – processing nodes should be able to free up the resources for app containers which are no longer required

`ZoneController` (inherited from `ProcessingNode` → `Master`): Apart from task scheduling, ZoneCntrollers should be able to attach/detach slave devices, such as sensors, actuators and co-processors

- `attachDevice(deviceId)`
- `detachDevice(deviceId)`

`Actuator Controller`: application container responsible for managing of actuator devices (turn on or off) and execution of commands for given protocol, format and specific parameters.

- `turnOn()`
- `turnOff()`
- `executeCommand(protocol, parameters, template, allowedLatency)` – parametrizes command template with given set of actuator-specific parameters, while the deadline for command completion should not exceed the value of allowed latency

`Sensor Controller`: application container enabling the acquisition of measured sensor data and managing of sensing devices

- `startMeasuring(measurementsPerSecond, parameters)` – sets up sensing device to start measuring the observed real world phenomenon, generating the given number of measurements per second, for given sensor setup configuration parameters
- `stopMeasuring()` – device should stop measuring values
- `getMeasurements(allowedLatency)` – component should return the current measurements in no longer time than the one given by `allowedLatency` parameter

`Processing Task`: application container which takes sensor data or other processing task's output values as its input, performs some kind of data transformation and generates outputs in given format

- `process(inputs, inputFormat, compatibleFormat, outputFormat, allowedLatency): outputs`

`Co-Processor`: additional chips used for processing should be able to go to power saving mode if not used for computation

- `powerSavingMode()`

Interfaces are not included as part of Ecore metamodel specification. Therefore, another formalism is used for interface specification, such as Interface Definition Language (IDL). Example of adopting such notation for ZoneController is given:

```
interface ZoneController{
    void scheduleTask(String containerId);
    void freeResource(String containerId);
    void attachDevice(String deviceId);
    void detachDevice(String deviceId);
}
```

VI. PROTOTYPE IMPLEMENTATION

The initial implementation of prompting tool is done in Python programming language. When it comes to construction of prompts that will be forwarded to LLM, we can distinguish four crucial processing steps:

- model parsing – depending on particular scenario involving LLM, different information, such as property values could be required from the model instances. For that purpose, considering that we rely on Ecore framework, the corresponding parsing library for Python is used –
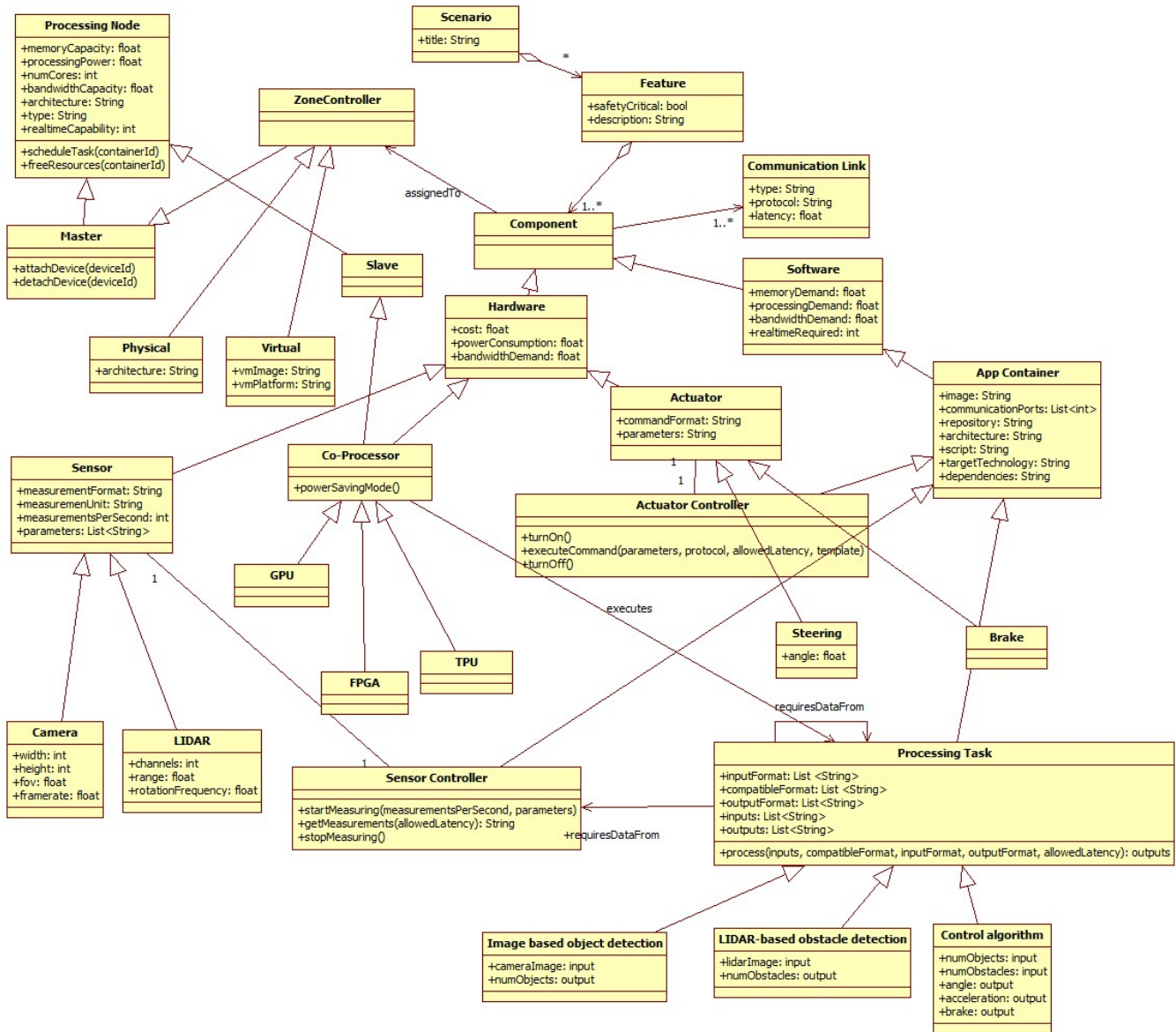
Fig. 3. Centralized Car Server Metamodel.

PyEcore. This library enables convenient model instance parsing, element traversal and parameter retrieval.

- prompt construction – in this step, we take the parsed values from the model instance or user-provided free-form text and parametrize pre-defined prompt templates.
- prompt execution – invoking the prompting service API of the underlying LLM – either locally deployed or in cloud (assuming that the corresponding LLM is properly deployed, accessible via current network and exposes such interface). As we make use of ChatGPT in our current experiments, OpenAI's Python API for this LLM-based service is used.
- interpreting LLM outputs – post-processing of the results which are returned from LLM, including their parsing,

extraction of relevant results and even further transforming them to suitable form for the next block within the LLM-enabled pipeline. In this step, it is of utmost importance to clear the LLM-generated result from all the additional text which resulted due to "hallucination" effect. Therefore, after this step, we should have clean, at least syntactically correct output of LLM which can be further used.

UML class diagram illustration the prototype tool implementation is given in Fig 4.

In the first step, Ecore model instance file is loaded into the tool and parsed using `loadModel` method of `ModelParser` which has `currentModelPath` as argument. As outcome of this process object representation is
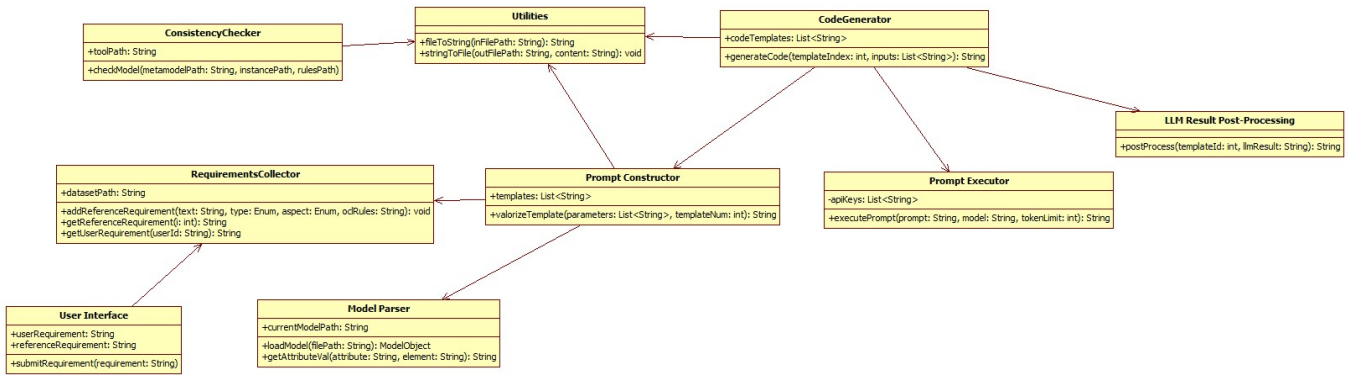
Fig. 4. LLM-enabled tool prototype.

formed in memory, so its structure and properties can be traversed. After that, it is possible to retrieve the values of distinct element attributes, using `getAttribute` method, which looks for the given model instance element and returns the value of target attribute/property. Moreover, we have `PromptConstructor` which contains a list of prompt templates. In our current version of the prototype, we include three types of templates, which are relevant to the scope of our proposed workflow: model instance creation, OCL rule extraction and code generation template.

The implementation of the prompt construction mechanism is in `valorizeTemplate`, which parametrizes the prompt template from the list with given identifier value. The second argument of this method is the list of parameter values which will be used to parametrize the prompt selected by id. These parameters are extracted from the model instance itself, relying on `ModelParser`.

Once the prompts are constructed, the next step is `PromptExecutor`, which is responsible for executing the prompts parametrized with appropriate values from the instance model. `PromptExecutor` can optionally have one or more `apiKeys` stored, which are used for authentication in case of external LLM-based services, such as ChatGPT. These values are user-specific and should not be exposed publicly. The crucial method of prompt executor is `executePrompt` which sends the constructed prompt to selected LLM model, taking into account the maximal answer length, limited by `tokenLimit` (which prevents from unexpected charges in case of external, pay per use services). The result of this method is raw output coming directly from LLM in response to the forwarded prompt.

`CodeGenerator` coordinates the LLM-based flow – from user-defined and reference requirements in textual form to generated executable code, such as Docker container. It contains a list of code templates for various types of assets that are expected to be generated (such as Docker Compose YAML file, Kubernetes YAML descriptors or CARLA Python scripts) which are inserted as input to prompt constructor for parametrization of specific prompts.

However, the generated response could contain redundant or additional text which is not useful for the further workflow, so it needs to be post-processed. For this purpose, we perform post-processing of the generated results and the outcome depends on the template identifier itself, as the same template is likely to lead to the generation of similar result by LLM. The underlying method is `postProcess`, and it has two arguments: `templateId` – identifier of prompt template; `llmResult` – the raw output of LLM. The output of this method is the string which contains only the useful information, without additions.

For intermediary steps involving model instance usage, before the process of code generation, the XMI Ecore model instance is forwarded to `ConsistencyChecker`. Its main method performs XMI model instance verification for given Ecore metamodel definition and set of OCL rules. For each of the rules, the output would be pass or fail. In case that all checks pass, the flow goes further to code generation phase. Otherwise, a message to the end-users is sent, so they know that manual intervention or correction is needed. In our prototype, this class relies on an external executable file implemented in Java programming language and packed as JAR file.

Additionally, our current prototype implementation also includes several auxiliary classes:

- `UserInterface` – Handles user input from web form - either reference or custom requirements, and forwards them to `RequirementsCollector`.
- `RequirementsCollector` – Used for construction of requirements dataset in automated manner. It is assumed that web-based GUI is used by experts in order to enter the requirements in textual form with respect to the structure, which is later described in the section about the used datasets. There are three methods in this class. The first one adds the requirement to the dataset in CSV format, based on reference requirements. The second retrieves the i-th row from the requirements. The third one accepts user-defined requirement, coming from the web input form.
- `Utilities` – Provides functionalities enabling to read textual file as string from disk and vice versa – to write

string variable into given textual file.

VII. EXPERIMENT DEMONSTRATION ENVIRONMENT

The Central Car Server runs isolated environments for ZoneControllers, where each of the ZoneControllers runs a set of assigned Docker containers. ZoneControllers can be either physically (distinct chips or co-processors) or logically isolated (VMs or containers).

Practically, to each of the ZoneControllers, a distinct Docker Compose file defining the running containers and ports they use for communication can be assigned. Additionally, adoption of Kubernetes is considered as well, especially considering the cloud-over-the-air update scenario and redundancy mechanism incorporation in order to increase the overall robustness of the resulting system. When it comes to code generation, the idea is to extract model instance parameters and target to produce Docker Compose files or Kubernetes descriptor YAML files, where each of these files will be executed by one of the zone controllers.

In the given example (shown in Fig. 5), we have two ZoneControllers – back and front. The first one is responsible for the frontal part of the car and runs camera sensing container which produces input for object detection-based emergency brake. In case that object is detected at short distance, automatic emergency brake will be activated and send signal to central car control container, as shown. A basic demonstration of object detection implemented in CARLA simulator using one virtual camera in the 3D environment is shown in Fig. 6. Similarly, there is a distinct Zone Controller for the back part of the vehicle. In our demonstration, it is assumed that two-way integration with CARLA simulation is established, so the images are recorded from CARLA simulation environment, while commands issued by control component affect the car's position in simulation. The communication port in our case is 2000 when it comes to communication with CARLA simulator server in both directions.
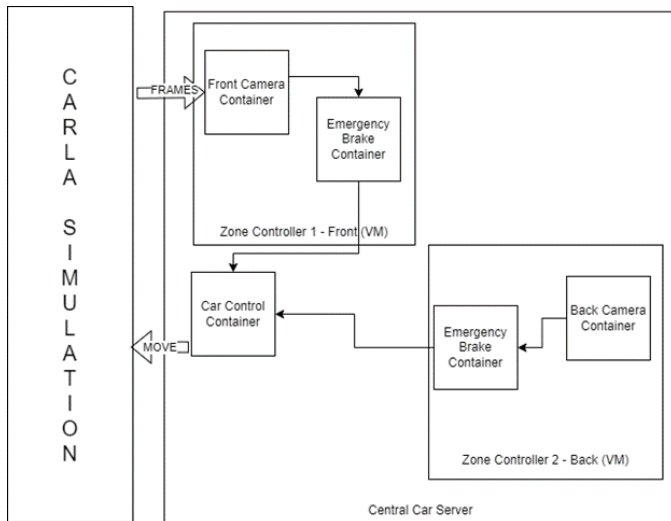
Fig. 5. Integration of central car server with CARLA simulation.

Fig. 6. 3D Object Detection in CARLA simulation environment for central car server evaluation.

A screenshot of CARLA simulator, showing the perspective of multiple virtual cameras in 3D environment, is given in Fig. 7.

Fig. 7. CARLA simulation environment for central car server evaluation.

VIII. RESOURCE ALLOCATION

The resource allocation problem in this case can be treated as a mapping of software resources: m application containers $(c_1, c_2, ..., c_m)$ to n processing nodes $(p_1, p_2, ..., p_n)$ with respect to some pre-defined criterion policy, such as minimal price, minimal power consumption, minimal latency or maximum execution speed.

In what follows, the developed model for optimal deployment is described, built upon the works from [8] and [9]. For each pair of processing node and container, we assign a decision variable $allocation[i, j]$ indicating whether the container c_j is deployed on the node p_i or no:

$$allocation[i, j] = \begin{cases} 1, & c_j \text{ is deployed on } p_i, \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

An example objective function would be to minimize the cost of components required, by deploying the containers to the cheapest nodes that fulfill all the criteria:

$$minimize \sum_{i=1}^{n} \sum_{j=1}^{m} cost[i] \cdot allocation[i, j], \quad (2)$$

where $cost[i]$ is the price of the processing node p_i.

Furthermore, the following constraints have to be considered. First, the sum of the application container memory demands – $memoryDemand[j]$ – of the allocated containers should not exceed the memory capacity of the processing node – $memoryCapacity[i]$, where this container is deployed, given as:

$$\sum_{j=1}^{m} allocation[i,j] \cdot memoryDemand[j] \leq$$
$$\leq memoryCapacity[i], \quad \forall i \in [1,n] \quad (3)$$

Similar conditions hold for bandwidth constraints and processing power:

$$\sum_{j=1}^{m} allocation[i,j] \cdot bandwidthDemand[j] \leq$$
$$\leq bandwidthCapacity[i], \quad \forall i \in [1,n] \quad (4)$$

$$\sum_{j=1}^{m} allocation[i,j] \cdot processingDemand[j] \leq$$
$$\leq processingPower[i], \quad \forall i \in [1,n] \quad (5)$$

Each container, which requires real-time capabilities, must be deployed on the processing node with such support:

$$\sum_{j=1}^{m} allocation[i,j]\cdot$$
$$\cdot |realtimeCapability[i] - realtimeRequired[j]| = 0,$$
$$\forall i \in [1,n] \quad (6)$$

Similarly, for each container, its architecture type (x86, ARM, GPU, TPU) must match with the processing node's computing architecture, which is relevant in our case as various co-processors can be adopted as well

$$\sum_{j=1}^{m} allocation[i,j]\cdot$$
$$\cdot |nodeArchitecture[i] - appArchitecture[j]| = 0,$$
$$\forall i \in [1,n] \quad (7)$$

IX. Experiments

This section gives overview of the examples relying on LLM usage, including distinction between the inputs, output and the underlying prompt for each of them.

A. Model instance creation

Requirements + Metamodel → Model instance (Fig. 8)

B. Constraint extraction

Requirements + Metamodel → OCL constraints (Fig. 9)

C. CARLA code generation

Model instance + Template → Component Script (Fig. 10)

D. Constraint extraction

Requirements + Metamodel → OCL constraints (Fig. 12)

E. Deployment configuration generation

Model instance → Docker Compose (Fig. 11)

F. Device interoperability

Model instance → Python adapter code (Fig. 12)

G. Requirements verification

For purpose of formal design-time model instance verification, we make use of the so-called Object Constraint Language (OCL) declarative language. It represents an extension of UML notation, giving the ability to specify some system design aspects with more details, which are verification rules in our case. Using OCL notation, a set of logical rules based on reference architecture/ISO standard is defined which are checked against the user-provided model instance. An example of ISO standard which can be used for this purpose is ISO/TR 4804 [10] (going to be replaced soon by ISO/CD TS 5083 [11]). In our case, OCL rules are generated relying on LLM, starting from free-form text. Let us assume that we start from reference hardware requirements for Conditional Automated Highway Drive: "This system represents an example sensor configuration with six cameras (at least 2.1MP), two LIDARs and five radars...". In OCL notation, we would have the following rules for verification:

```
context Feature
inv CamerasCount:
self.cameras->size() = 6
context Feature
inv LIDARCount:
self.lidars->size() = 2
context Feature
inv RADARCount:
self.radars->size() = 5
context LIDAR
inv LIDARChannels:
channels > 0
inv LIDARRange:
range > 0
context RADAR
inv RADARRange:
range > 0
context Camera
inv CameraResolution:
width > 0 and height > 0
inv MinimumResolution:
(width * height) >= 2.1 * 1000000
-- Assuming resolution is measured
-- in megapixels (MP)
```

X. Dataset

For purpose of experimentation, we created a custom dataset of mock automotive industry requirements based on AVC Consortium document [12], which is publicly available. The reason for this decision comes from fact that most of the automotive vendors, OEMs and other industry partners might

Fig. 8. Model instance creation experiment illustration.

be concerned about their data becoming public, which could lead to potential legal issues. As for now, two custom datasets are created: 1) reference requirements + Ecore metamodel ->OCL constraints 2) user requirements + Ecore metamodel ->Ecore model instance.

Table I depicts the layout of the dataset. We have the following inputs: 1) Requirement text – Natural language, unstructured text covering some requirement defined within either the ISO standard or reference architecture used for vehicle system development (first dataset) or user-defined requirement (second dataset). Regarding the first dataset, the idea is to collect this kind of requirements from publicly available examples, reference documents or ISO standards for automotive industry. This way, we aim to enforce the adoption of good practices within the user-created system instances. On the other side, the requirements of the second type are collected from application form filled by user; 2) Feature – to which feature the particular requirement is related to; 3) Scenario – to which higher level functionality the particular feature belongs to (such as emergency break within highway pilot); 4) metamodel – reference, human-created metamodel.

On the other side, we have two possible outputs: 1) OCL rules – free-form text about the architectural and standardisation-related constraints expressed in form of OCL rules with respect to the input metamodel; 2) model instance – the resulting annotation of free-form text with respect to the input metamodel.

TABLE I
LAYOUT (HEADER) OF THE DATASET: INPUTS (REQUIREMENTS TEXT, FEATURE, SCENARIO, METAMODEL) ARE MARKED WHITE, OUTPUTS (OCL RULE, MODEL INSTANCE) ARE MARKED GRAY.

Requirement text	Feature	Scenario	Metamodel	OCL rule	Model instance

XI. CONCLUSION

In this paper, we presented a prototype of model-driven framework leveraging LLMs, as an attempt to introduce automated software development to the automotive industry. Several use cases and examples adopting such development approach are demonstrated, with the main goal of generating an extensible and future-proof, centralized software vehicle system.

According to the initial findings, such approach seems promising, especially when it comes to reduction of time and effort needed for development of automotive software because of automated code generation. Additionally, auto-mated verification of system models relying on Ecore model instance representation and OCL rules in design-time could potentially reduce the later efforts needed for run-time testing of the created systems. However, the presented workflow is a proof-of-concept, and it must be properly evaluated with large datasets of requirements used in the industry. Our plan is to incorporate more advanced techniques for processing of the ISO standards and reference architectures using LLMs, such as Retrieval Augmented Generation (RAG).

Fig. 9. OCL constraint rule extraction experiment illustration.

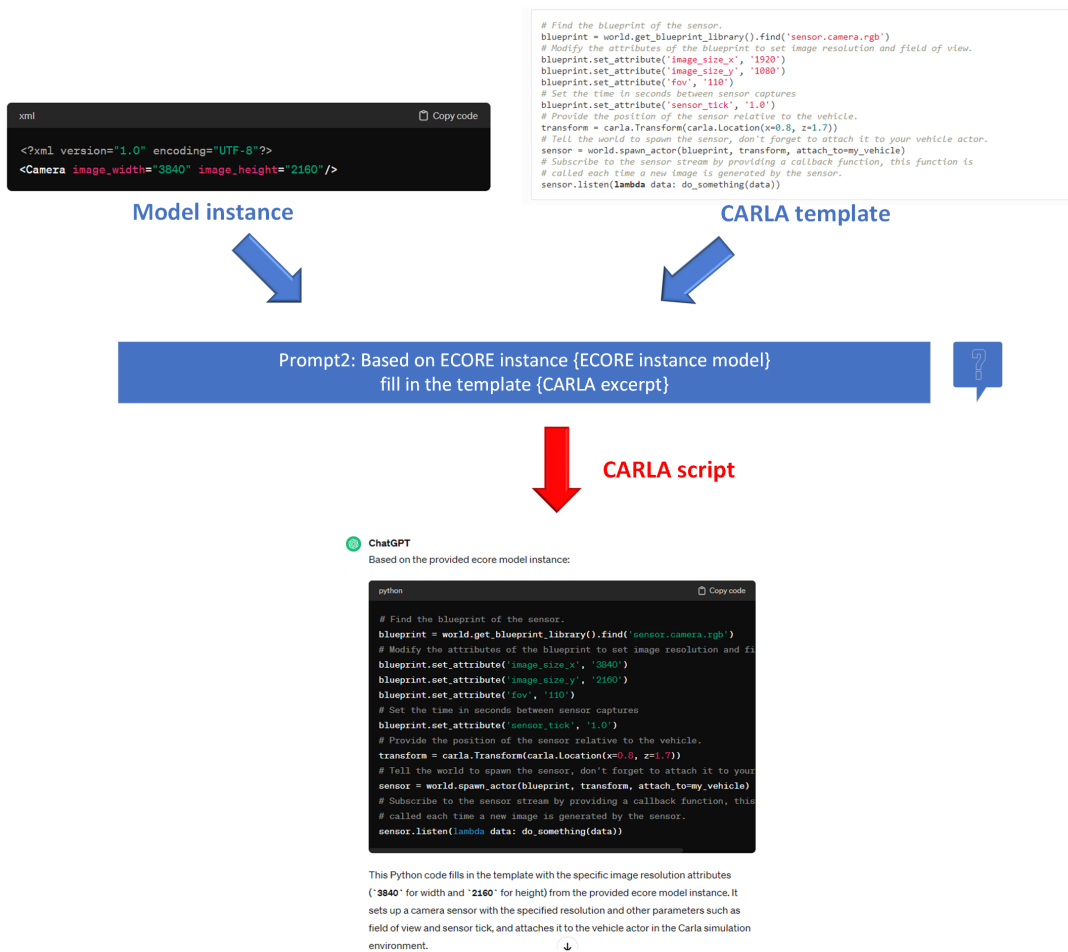Fig. 10. CARLA code generation experiment illustration.

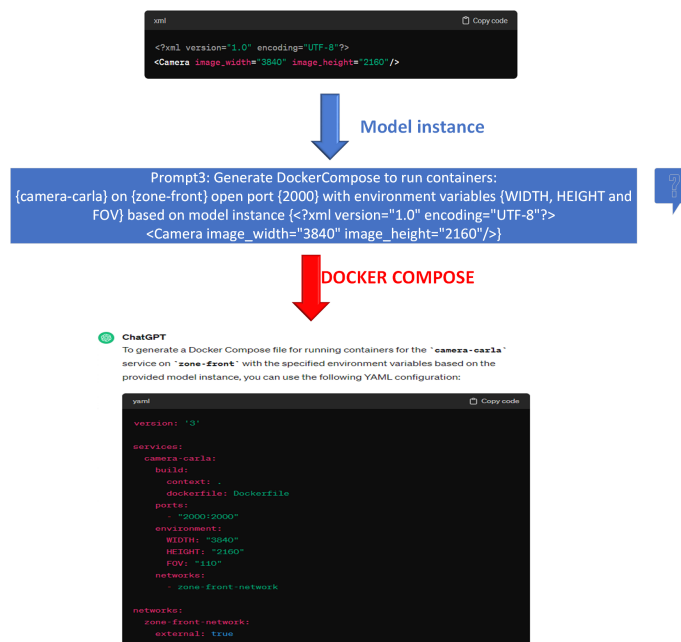Fig. 11. Deployment configuration generation experiment.

```
<Feature xsi:type="ObjectDetection" compatibleFormat="RGB">
  <camera xsi:type="Camera" format="CMYK"/>
</Feature>
```

Model instance

Prompt: Write script in {Language -Python} which converts {ProcessingTask.requiresDataFrom.Sensor.format - CMYK} to {ProcessingContainer.compatibleFormat[0] - RGB image}

PYTHON CODE

ChatGPT

You can use the Python Imaging Library (PIL), which is now maintained as Pillow, to convert CMYK images to RGB. Here's a script that demonstrates how to achieve this conversion:

```python
from PIL import Image

def cmyk_to_rgb(cmyk_image_path, rgb_image_path):
    # Open the CMYK image
    cmyk_image = Image.open(cmyk_image_path)

    # Convert CMYK to RGB
    rgb_image = cmyk_image.convert('RGB')

    # Save the RGB image
    rgb_image.save(rgb_image_path)

if __name__ == "__main__":
    # Input CMYK image path
    cmyk_image_path = "input_cmyk_image.jpg"

    # Output RGB image path
    rgb_image_path = "output_rgb_image.jpg"

    # Convert CMYK to RGB
    cmyk_to_rgb(cmyk_image_path, rgb_image_path)

    print("Conversion completed!")
```

Fig. 12. Data format interoperability experiment.

REFERENCES

[1] 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.

[2] N. Petrovic and I. Al-Azzoni, "Model-Driven Smart Contract Generation Leveraging ChatGPT," in *ICSEng 2023*, 2023, pp. 387–396. DOI: 10.1007/978-3-031-40579-2_37.

[3] 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.

[4] Ö. Ş. Taş, S. Hörmann, B. Schäufele, and F. Kuhnt, "Automated vehicle system architecture with performance assessment," in *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, 2017, pp. 1–8. DOI: 10.1109/ITSC.2017.8317862.

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

[6] "CARLA Simulator Sensors Reference." (2024), [Online]. Available: https://carla.readthedocs.io/en/latest/ref_sensors/ (visited on 03/17/2024).

[7] K. Lebioda, V. Vorobev, N. Petrovic, F. Pan, V. Zolfaghari, and A. Knoll, "Towards single-system illusion in software-defined vehicles - automated, ai-powered workflow," Technical University of Munich, Tech. Rep., 2024. [Online]. Available: https://mediatum.ub.tum.de/doc/1737800/1737800.pdf.

[8] 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.

[9] 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.

[10] "ISO/TR 4804:2020." (2024), [Online]. Available: https://www.iso.org/standard/80363.html (visited on 03/17/2024).

[11] "ISO/CD TS 5083." (2024), [Online]. Available: https://www.iso.org/standard/81920.html (visited on 03/17/2024).

[12] "AVC Consortium, Technical Reports & More from AVCC." (2024), [Online]. Available: https://avcc.org/documents/ (visited on 03/17/2024).