IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 15 November 2024, accepted 1 December 2024, date of publication 5 December 2024,
date of current version 26 December 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3512002

RESEARCH ARTICLE

Toward Software-Defined Vehicles: From Model-Based Engineering to Virtualization-Based Deployment

FENGJUNJIE PAN[1], MARKUS RICKERT[2], (Member, IEEE), TOBIAS BETZ[3], (Graduate Student Member, IEEE), LONG WEN[1], (Graduate Student Member, IEEE), JIANJIE LIN[1], NENAD PETROVIC[1], MARKUS LIENKAMP[3], (Member, IEEE), AND ALOIS KNOLL[1], (Fellow, IEEE)

[1]Chair of Robotics, Artificial Intelligence and Real-Time Systems, School of Computation, Information and Technology, Technical University of Munich, 85748 Garching, Germany
[2]Chair of Multimodal Intelligent Interaction, Faculty of Information Systems and Applied Computer Sciences, University of Bamberg, 96047 Bamberg, Germany
[3]Institute of Automotive Technology, School of Engineering and Design, Technical University of Munich, 85748 Garching, Germany

Corresponding author: Fengjunjie Pan (panf@in.tum.de)

This work was supported in part by the MANNHEIM-Central Car Server-Supercomputing für Automotive (CeCaS) Project through Bundesministerium für Bildung und Forschung (BMBF).

ABSTRACT The automotive industry is moving towards software-defined vehicles, where software shapes the car's features and user experience. Unlike traditional vehicles that rely on multiple separate computing units with tightly coupled software, the new centralized high-performance computers offer a more flexible, software-defined platform. Mapping software to hardware is the process of resource allocation. As the number of software components in software-defined vehicles increases, managing this process with the existing approach becomes increasingly challenging. In this paper, we propose utilizing an automated and model-based approach for analyzing/planning resource allocation during the design phase and implementing a virtualization-based environment using virtual machines and containers for deployment in software-defined vehicles. Our approach enables automated resource allocation, thereby ensuring flexible software deployment and updates in the vehicles. The proposed workflow begins with the formal definition of system information through models and constraint languages. Based on this, an optimization problem is generated and solved automatically. Afterwards, deployment-related code is generated based on the optimization results from the optimization solver and is then transmitted and deployed to the vehicle. We demonstrate our method by setting up a simulated software-defined vehicle platform using an ARM-based automotive reference central computer with Autoware applications.

INDEX TERMS Automotive, E/E architecture, model-based engineering, resource allocation, software-defined vehicles, systems, software engineering.

I. INTRODUCTION

In recent years, the automotive industry has experienced a remarkable evolution due to the growth of software development, driven primarily by intelligent vehicle functions. The expansion of vehicular software is transforming automobiles from conventional electromechanical terminals into adaptable electronic platforms [1]. This trend has given rise to the concept of Software-Defined Vehicles (SDVs), which are gaining significant attention in the industry. With the evolving demands of SDVs, the automotive industry is shifting from traditional distributed Electrical and Electronic (E/E) architectures towards centralized architectures equipped with High-Performance Computers (HPCs) (Fig. 1). The central car architecture enhances the manageability of vehicle functions while reducing the complexity of maintenance. At the same time, applying hardware and Operating System (OS)-level virtualization

The associate editor coordinating the review of this manuscript and approving it for publication was Ivan Wang-Hei Ho.

in future SDVs has drawn increasing attentions [2], [3], [4]. By utilizing virtualization technology in a centralized architecture (Fig. 2), the central high-performance computer can be partitioned into isolated Virtual Machines (VMs), allowing the co-existence of applications with different requirements. This brings improved flexibility, scalability, and upgradability to the vehicle, enabling seamless software updates and over-the-air upgrades.

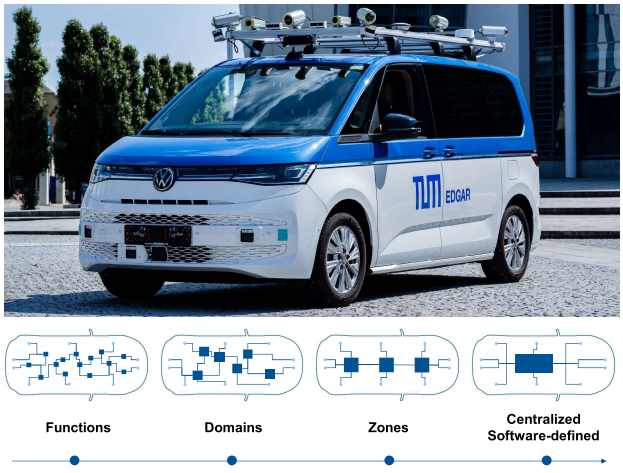

FIGURE 1. Trend of automotive E/E architecture. The EDGAR research vehicle serves as example demonstrator for the centralized compute architecture [5].

Design Space Exploration (DSE) is one of the crucial processes when it comes to decisions regarding automotive architecture. It encompasses a wide range of considerations, from the layout of the vehicle's components to the integration of diverse bus systems, hardware and software elements. The resource allocation process, involving the precise mapping of software components onto hardware resources, poses a crucial and complex challenge in vehicular design and deployment. Traditionally, this design process involves a large number of human engineers. Various tools for system representation and requirement management are employed to ensure a comprehensive understanding among all involved parties. The design decisions are primarily based on human experience. However, with the exponential increase in applications and frequent software updates in SDVs, manual and experience-based resource allocation has become increasingly complex and can hardly be managed by human engineer. To address these challenges in SDVs, automated approaches need to be investigated. In the current automotive domain, Model-Based System Engineering (MBSE) approaches, such as those presented in [6] and [7], have been proposed to accelerate resource allocation design. However, existing methods mainly target distributed E/E architectures, limiting their flexibility for central computer-based SDVs. Our previous work [8] focuses on an automated approach for resource allocation in SDVs. In this work, we extend our DSE algorithm with assumption-based solving for locating infeasible requirements during problem-solving. In addition, we utilize the power of code generation to connect our MBSE approach (Fig. 2) as shortly discussed in our previous work [9] and showcase the feasibility of the proposed approach with realistic SDV scenarios.

The proposed approach follows the concept of correctness by construction. It leverages well-known definitions from the software and systems engineering domains to formally model system information. This includes meta-models, instance models, constraints, and objectives. The formal information is then be automatically converted into optimization problems, which are addressed by optimization solvers. Subsequently, solutions are transformed back into the instance model, on the basis of which deployment code for VM and container configuration will be generated. Through a proper communication channel, the deployment files will be transmitted and executed on the target SDV platform. We illustrate the effectiveness of this approach through a case study involving an SDV scenario with an ARM-based vehicle HPC [10]. The experiments are conducted in the development environment of the EDGAR research vehicle (Fig. 1) [5] including the Autoware applications [11] in combination with the AWSIM environment [12].

The main contributions of this study are as follows:

- We introduce a model-based approach to address the resource allocation problem in SDVs. This method generates resource allocation decisions automatically based on system components and design constraints, thereby accelerating the SDV design process. Additionally, it provides feedback in cases of constraint conflicts, allowing for more efficient resolution.
- We propose integrating this model-based approach with virtualization technologies, such as containers and hypervisors, to facilitate SDV deployment. This enables flexible deployment while maintaining the independence of subsystems and minimizing interference among them.
- We present a reference software stack concept for the proposed end-to-end workflow, which integrates model-based design and virtualization-based deployment for SDVs. This concept is demonstrated on an SDV reference hardware across three different scenarios: software deployment, updates, and platform reconfigurations.

The rest of this paper is structured as follows: Sections II and III show the challenges of designing SDVs as well as related works in the field of automotive model-based engineering and software deployment. Section IV introduces the proposed end-to-end workflow for the design and deployment of SDV systems. Sections V and VI delve into the technical details of models, transformation algorithms, and the SDV software stack using virtualization technologies. To illustrate the applicability of our method, different SDV related scenarios are demonstrated in Section VII.

II. CHALLENGES IN SDVS

The automotive industry is transitioning from hardware-based automotive architecture to a software-based,

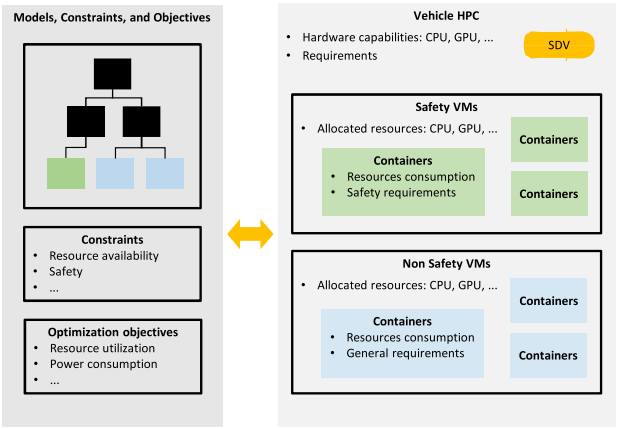

FIGURE 2. Model-based system engineering and virtualization-based deployment of SDVs.

software-defined design. An SDV describes a vehicle whose features are primarily enabled by software applications [13], [14]. This concept involves more than software upgrades over the air. It represents the transformation towards a more adaptable automotive system. However, the development of SDVs encounters various challenges.

A. SHORT TIME TO MARKET

The software in SDVs, which facilitates various vehicle functions, is anticipated to grow in both quantity and complexity. Integrating such software presents significant difficulties when designing SDVs. The software integration needs to consider several aspects, including the vehicle's performance, safety, power consumption, and more. Traditionally, human engineers had years to design such vehicle systems. However, the development process for customized SDVs should be automated and accelerated to reduce the time to market.

B. SCALABILITY

Scalability of vehicle features is a primary characteristic of SDVs. The software in traditional vehicles is typically embedded in individual Electronic Control Units (ECUs). This tight integration of software and hardware, while ensuring defined performance of vehicle functions, also limits the scalability. Introducing new functions often requires the introduction of new ECUs, which increases the overall cost and requires significant integration work. In contrast, SDVs promote a centralized architecture, where functions previously controlled by multiple ECUs are consolidated. New functions can flexibly be deployed on the central computer of SDVs. This approach demands the decoupling of software and hardware to enhance adaptability of the vehicle system.

C. SAFETY AND SECURITY

Ensuring the safe and secure execution of automotive applications is crucial in any automotive system,. Traditional vehicles typically dedicate specific software and hardware configurations to each vehicle function to ensure safety and security. This includes allocating fixed resources such as processors and Random-Access Memory (RAM) when an application is coupled with a specific ECU. For example, infotainment systems are physically isolated from Advance Driver-Assistance System (ADAS) to prevent resource conflicts. However, in SDVs, where different functions coexist on a centralized platform, maintaining freedom from interference among applications is crucial and presents a significant challenge that needs to be addressed.

Many of the challenges discussed above have already been addressed or are currently being developed both within and outside the automotive industry. The model-based development approach, widely used in the aviation industry, is now being investigated for automotive design. In terms of software management, a package manager automates the process of installing, configuring, and removing software on a typical computer. On Linux systems, Flatpak offers similar functionalities, including resource isolation and managing software versions. Recent advancements in virtualization technologies have proven beneficial in cloud computing. Hardware virtualization tools, such as hypervisors, can create independent and isolated environments on the same physical hardware. OS-level virtualization tools, like containers, provide resource isolation for safety and security, and lifecycle management for applications.

In this work, we intend to address challenges of automated SDV design and deployment by combining MBSE methods and virtualization technologies on SDVs. We focus on three representative SDV use cases (Fig. 3):

- Software deployment (Fig. 3a): During the development phase of SDVs, a significant amount of applications are integrated. Given an unconfigured hardware platform, along with a set of applications and their respective requirements, the deployment process should be thoroughly analyzed and executed automatically.
- On-the-fly software update (Fig. 3b): Once the vehicle is operational at the customer's side, software updates need to be implemented seamlessly. These updates should occur automatically, requiring no manual intervention and ensuring that there is no disruption to the vehicle's ongoing functions.
- Platform reconfiguration for new applications (Fig. 3c): When the vehicle owner intends to install new applications on the vehicle, the SDV platform should be capable of reconfigure itself, if necessary, to host the new application without violating existing requirements and interfering with existing applications.

III. LITERATURE REVIEW

Our research integrates diverse perspectives, ranging from MBSE for resource allocation problems to automotive software deployment concepts.

Pohlmann et al. [6] developed the MechatronicUML methodology for solving resource allocation problem in distributed systems, leveraging a specific description language for modeling system components like cores and memories.

(a) Software deployment (b) Software update (c) Platform reconfiguration for new application

FIGURE 3. Scenarios in SDVs. The illustration of SDV scenarios is based on the demonstration setup from Section VII. (a) Eight Autoware applications should be deployed on an in-configured SDV HPC. Due to varying requirements of the applications, two different VMs should be created. (b) The application Planning is planned to be updated on the fly. (c) The platform requires a new VM to host a non-safety application Stress.

In addition, a custom Domain-Specific Language (DSL) was created to address five types of allocation constraints. These pre-defined constraints, coupled with transformation templates, enabled the automatic generation of Integer Linear Programming (ILP) problems from the DSL. The formulated ILP problems were processed using ILP solvers to find feasible resource allocation configurations. Eder et al. [7] introduced AutoFocus3 for automotive software and hardware mapping. Their approach incorporated custom meta-models and a self-defined DSL for the describing system components, allocation constraints, and optimization goals. This DSL facilitates the definition of constraints and objectives following pre-defined patterns, which can be translated to Satisfiability Modulo Theories (SMT) problems using pre-defined rules. Al-Azzoni et al. [15] presented a flexible framework targeting resource allocation, enabling the creation of customized meta- and instance models for system description. This framework differs from the above-mentioned works by provides a meta-model for software component allocation problem descriptions containing classes for target components and constraints. By creating model transformation from user models to the problem description models, Mixed Integer Linear Programming (MILP) problems can be further generated in this framework. However, the pre-defined constraint class in the meta-model limits the expressiveness of constraint specification. Our previous work [8] introduced a model-independent transformation, EMF2SMT, that transforms user-defined models, constraints and optimization goals into mathematical problems, which brings the flexibility of designing SDV systems with future requirements. The generated expressions are presented as SMT formulations because both SMT and Object Constraint Language (OCL) exhibit the characteristics of first-order logic. This work further enriches our method with mechanism of detecting design conflicts and combines MBSE with further deployment execution in SDVs.

In term of automotive software deployment methods, the traditional approach is the flashing of embedded ECUs [16]. Using diagnostic tools and a boot loader within the ECU, dedicated software is downloaded via physical access and written into the flash memory of ECUs. Flashing all necessary ECUs for a single vehicle can take hours [17]. Any software changes in a vehicle must typically be performed at a dealership. To address this limitation, Over-the-Air (OTA) updates have been introduced in the automotive domain, enabling software updates to be performed remotely at the customer's location. OTA updates allow code on embedded ECU to be updated remotely, supporting remote improvements and fixes. However, despite OTA capabilities, the software flashed or updated on ECUs is often highly customized for each individual ECU [18]. Software deployment and feature configurations remain manually predefined during the vehicle design phase, which limits flexibility at runtime [19]. Therefore, the traditional software deployment methods of ECU flashing and OTA updates face limitations in SDVs, where software changes happen frequently. To enable a more flexible automotive platform, Yoo et al. [20] introduced an Android-based software architecture that supports plug-and-play applications in distributed automotive systems. Their approach uses a primary ECU running Android with network access for downloading applications and distributes tasks to secondary ECUs operating with real-time OS. While this approach introduces flexible deployment in distributed automotive systems, it does not support centralized automotive architectures using a central HPC and relies heavily on the Android platform for software management. Recently, virtualization technologies have gained popularity in cloud computing due to their flexibility and scalability in resource allocation, and they are now being explored for automotive applications. Rajan et al. [21] investigated the integration of virtualization into automotive multicore controllers, evaluating virtualized system performance in terms of core loading, interrupt timing, and task timing parameters. Wen et al. [2] conducted benchmark tests for hypervisors and containers, showing that virtualized or containerized environments exhibit performance comparable to that of bare-metal setups. Additionally, they discussed the startup times for Autoware-based automotive applications, emphasizing the reduced startup times made possible through container orchestration. Betz et al. [22] analyzed application latency within containerized environments, showing that containers improve latency performance for complex applications such as Autoware. In this study, we further explore and apply virtualization technologies for software deployment in SDVs.

In summary, existing concepts in MBSE for resource allocation lack the flexibility needed to adapt to diverse scenarios with varying requirements, limiting their support for future SDV architectures. Furthermore, traditional automotive software deployment methods, such as ECU flashing,

lack the scalability and adaptability required for SDVs, which demand shorter development cycles and frequent software installations and updates. Recent studies on automotive software deployment focus primarily on benchmarking the performance of different technologies. However, there is currently no literature that addresses end-to-end SDV software deployment, encompassing the entire process from the design phase to vehicle runtime.

In this work, we propose utilizing a model-based approach to address SDV resource allocation problem during the design phase and implementing a virtualization-based environment using VMs and containers for the software deployment on the SDV central HPC. The proposed MBSE method not limited to a specific SDV architecture, thus it can support customized SDV concepts and requirements. By apply virtualization for deployment, the SDV system gains flexibility while maintaining security and safety considerations. Additionally, we employ code generation techniques to bridge the gap between SDV design and deployment, enabling an automated end-to-end procedure.

IV. METHODOLOGY

Fig. 4 presents the proposed end-to-end workflow. By combining the MBSE method with virtualization-based deployment strategy, we aim to ease the design and deployment of SDVs and enhance the scalability and flexibility of the target SDV system.

The proposed workflow starts with model creation from the system information using the Eclipse Modeling Framework (EMF). The meta-model serves as an abstraction of the target SDV system, defining classes that encapsulate meta-information for hardware and software components including computational units, applications, and hypervisors. The instance model represents the practical instantiation of a specific SDV system based on the meta-model. It contains detailed information about system components, as well as design decisions for resource allocation and application mapping. In the proposed workflow, the system developers are required to provide fundamental properties of system components within the instance model and can leave design decisions undefined or partially defined. For example, in the instance model for automotive resource allocation (Fig. 6), the developer can specify that one application requires six cores. However, which cores should be assigned to this application may remain undefined if this information is undecided. We refer to this instance model as the partial instance model of the SDV system. The complete system configuration will then be automatically generated in subsequent steps. Beyond the properties of system components, both safety and non-safety-related requirements, along with the design constraints raised by system developers, play an important role in developing an SDV system. Building upon the EMF meta-model, we employ the OCL [23] to express these constraints and optimization objectives formally. Through the model validation framework, e.g., in Eclipse, the instance model can be verified against formal OCL constraints. In the proposed workflow, we do not specify the exact step at which validation should be introduced, as the subsequent solving/optimization steps will implicitly perform model validation. However, in practice, users have the flexibility to introduce stand-alone model validation procedures at any point in the workflow according to their demands.

Once the model specification is presented, we leverage the EMF2SMT transformation, presented in [8], to automatically generate mathematical formulations as SMT problems. Additionally, we extend it for the implication of conflicts in constraints (Section V-C). State-of-the-art SMT solvers, such as Z3 [24], are employed to address the system design problem and perform optimization. Should the expected design configuration be identified, the solver's results are converted into the solution instance model through the Text2Model process. However, if the system's design cannot be resolved, the EMF2SMT algorithm will identify constraints in conflict. These conflicts are then reported to the system developer for manual inspection.

Utilizing the Model2Text technique, different configuration files are generated for deployment seamlessly. In this paper, we propose a centralized and virtualization-/containerization-based SDV architecture (Section VI). These configuration files facilitate the deployment of VMs and application containers.

The designing process from model creation to the generation of deployment files through Model2Text transformation can be performed on a PC used by system developers. The resulting configuration files are then transmitted to the SDV central computer via communication interfaces, such as REST API [25], to initiate deployment. Should there be any modifications to the target SDV system, e.g., incorporating a new application into the existing SDV configuration, the system developer can iteratively execute the proposed process by modifying or updating the existing instance model. The automated procedures can also be executed on the cloud platform or directly within the vehicles themselves to achieve flexible system configuration, reconfiguration and updates.

On the SDV, automation tools like Ansible [26], along with virtualization management tools such as Libvirt [27] and Kubernetes [28], can be utilized for automated SDV configuration. The advantage of applying such technologies is that changes in part of the system do not affect the rest. This means that by creating new VMs and containers, the existing systems remain unaffected. Details of virtualization-based SDV development will be introduced in Section VI.

V. MODEL-BASED SYSTEM ENGINEERING

The proposed end-to-end workflow (Fig. 4) for design and deployment of SDVs starts from the MBSE approaches. MBSE is a method for developing and managing complex systems. It forms the basis of our automated deployment approach and brings reusability, comprehensibility, maintainability, traceability and efficiency [29]. MBSE provides structured methods for documenting the system in various aspects. Within MBSE, the entire system is described

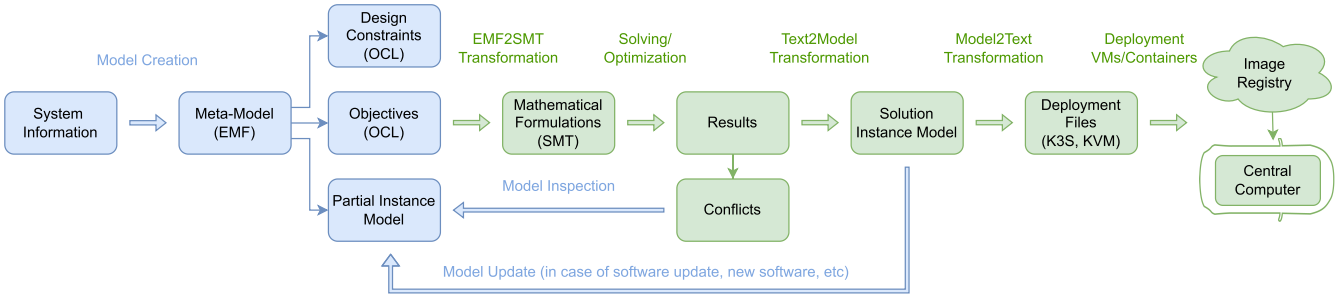

FIGURE 4. End-to-End workflow for the design and deployment of SDVs. Steps highlighted in green represent automated procedures, and those in blue should be done manually.

through diverse models, facilitating system analysis such as consistency checking and validation.

In this work, we employ model-based methods to create formal system models, requirements and optimization objectives. Leveraging design space exploration and code generation techniques, we aim to generate deployment plans and relevant configuration files automatically. The proposed approach aligns the existing modeling standards from Object Management Group (OMG). Therefore, it can be seamlessly integrated to other tools adhering to the same standard.

A. META-MODEL AND INSTANCE MODEL

As the abstraction of a target system, meta-models include rules, meta-types, and properties. As a system can incorporate various abstraction levels and concepts, its meta-models can differ. A meta-model can be used to construct concrete semantic models, known as instance model. An instance model represents a particular system instances. In this work, we ultilze the OMG standards for both specification meta-model and instance model for ensure broad acceptance across industries. The OMG Meta Object Facility (MOF) proposes a metadata architecture, defining the correspondence between model elements on each layer (instances in instance models) and those on the layer above (classes in meta-models) [30]. This concept is reflected in many modeling languages such as Unified Modeling Language (UML) and Systems Modeling Language (SysML). Our work utilizes EMF for modeling purposes. It is designed to adhere the MOF standard [31] and provides diagrammatic notations that facilitate the specification of meta-models and instance models. Through model transformations, models created in specific modeling languages can be converted into the EMF format.

In an EMF meta-model diagram, classes (represented as blocks) and attributes (keys within class blocks) are used to specify hardware/software components and their properties. Although the color of class blocks can be customized, blocks are generally colored to differentiate between class types: normal classes are shown in yellow, while abstract classes appear in grey. Relationships among components are defined using references (lines with arrows and diamond markers). Relationships among components are defined using references (lines with arrows and diamond markers). Labels on the references specify reference names and boundary conditions (integer intervals), which constrain the number of instances of target classes that can be associated with instances of source classes. Figure 5 presents an example meta-model for extended from [8] with deployment-related elements. The hardware of *SDV* consists individual *Devices* and *Boards*. *Devices* can have multiple *DeviceFunctions* and are connected via *Ports* in *Buses* to the *Board*. Boundary constraints of connected components are specified as well. These boundary constraints are enforced during instance creation. For example, multiple *Port* instance objects can be instantiated as child objects under a single *Bus*, but each *Port* can only be associated with one parent *Bus* object. *Board* is the central processing platform which provides fundamental resources, including *Cpu*, *Ram* and *Buses*. Each *Cpu* contains multiple *Cores*. The properties of hardware elements are modeled as attributes in each class. For instance, we define a boolean attribute *freqGovernor* in the *Core* class to represent if the core is in performance mode (true) or in power saving mode (false). Other heterogenous hardware, e.g., *Gpu*, is defined as a child class inherited from the abstract class *Device* that can be attached via *Port* of Bus. In this work, we propose a virtualization-based deployment strategy (Section VI). Therefore, the *Vm* class is created to represent isolated partitions to host applications. *Vm* configurations can be described via attributes (e.g., *os*, *storageSize*) and references (e.g., *core*, *port*). As a resource allocation constraint, we define a boundary condition in the model where each *Core* instance can be allocated to a maximum of one *Vm* instance. This condition is implemented through the reference between the *Core* and the *Vm* classes. In each *Vm*, applications can be deployed. Application requirements (e.g., *ramSize*) are modeled as attributes in the class *App*. We further utilize the class *Volume* and the class *AppEnv* to describe container-related settings including the volume configurations and environment variables which will further be used for the generation of deployment files.

Fig. 6 presents an instance model instantiated from the meta-model in Fig. 5. It describes our experimental SDV system with an SDV reference hardware platform, AVA AP1 by ADLINK [10], and open-source autonomous driving software, Autoware [11]. In a seperate property view, the properties of each element are specified. The modeling of instances are based on existing hardware and software

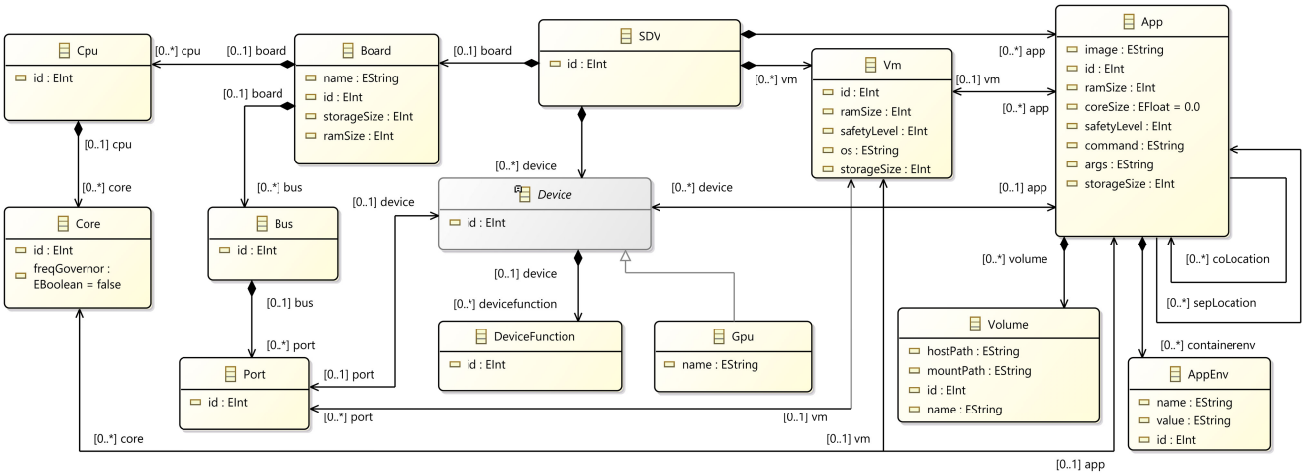

FIGURE 5. An example meta-model for SDVs with classes for hardware and software components and their mapping.

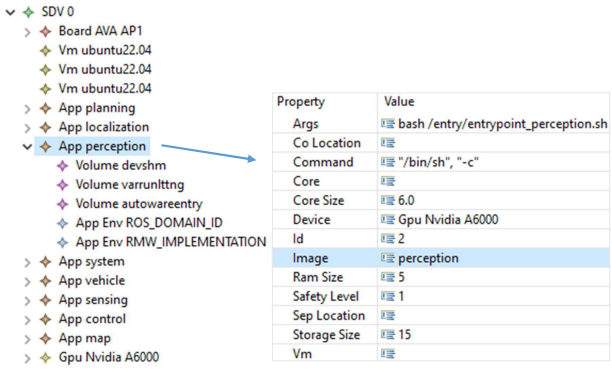

FIGURE 6. Instance model example (partial) for a SDV system.

requirements. For example, the RAM requirements of each applications are defined in the model. At the same time, decisions regarding resource allocation, such as assigning specific applications to particular VMs, can be defined as preliminary knowledge or treated as parameters within the exploration space to be resolved.

B. DESIGN CONSTRAINTS AND OPTIMIZATION OBJECTIVES

Design constraints and optimization goals play a crucial role in the system development, ensuring that the system provides desired functions while adhering to system requirements. In this work we employ the OCL, a declarative formal language introduced by OMG, to describe design constraints and optimization objectives that cannot be directly expressed in models. It enables automated model verification and design space exploration. An OCL constraint is comprised of a context (denoted as *context*) and an invariant (*inv*). Because of its similarity to first-order logic, OCL is capable of expressing various automotive requirements.

Automotive system is a complex system involving requirements from many different aspects including resource availability, application relationships, safety requirements, safety-related platform constraints, and etc [8]. Listing 1 presents some formally described OCL requirements related to the resource allocation in SDVs.

In automotive resource allocation scenarios, resource availability is always one of the most fundamental requirement. The constraint *appVm* specifies each application must have a hosting VM. The constraints *appCore, vmCore, appDevice, vmRam, vmStorage* indicate that VM should at least provide necessary resources for applications running on it. Such resources include cores, devices (such as GPU) and RAM. At the same time, all resource allocation should not exceed the capability of the physical hardware, as presented in the constraint *sdvRam, sdvStorage*. The constraint *unuseCore* specifies a minimal number of assigned cores, indicating the extendability for installing new applications and creating new VMs. Besides the above mentioned basic requirements, we have further leveraged requirements related to safety and platform configuration. As mentioned in ISO26262 [32], Freedom from Interference (FFI) guarantees the absence of cascading failures between two or more elements that could break system's safety state. Therefore, in the constraint *safety*, we specify that applications of different safety critical level should be separated into different VMs, ensuring that any failure in non-safety critical applications and VMs will not affect safety critical applications and VMs. Furthermore, in the constraint *freqGovernor* we define that cores hosting safety critical applications should be set to power mode instead of power save mode to ensure the stable performance in applications.

Apart from constraints, we also employ the extended OCL syntax [8] for specifying optimization objectives. We define the minimal number of utilized VMs (using VM's RAM size as an indicator) as the optimization goal for our example SDV:

```
minimize:
Vm.allInstances() -> select(ramSize > 0)
    -> size()
```

```
context App
 -- Each application should be assigned to
     one VM.
 inv appVm:
  vm -> size() = 1
 -- Each application should be assigned with
      enough core resources.
 inv appCore:
  core -> size() = coreSize
 -- Each application should be provided with
      required cores in VM.
 inv vmCore:
  core -> forAll(vm = self.vm)
 -- For any application that requires access
      to a specific device, the hosting VM
     must be provisioned with the port
     connecting to the required device.
 inv appDevice:
  device -> forAll(port.vm = self.vm)
 -- For each critical application, freqence
     governor of core should be set to power
     mode (true) to ensure stable performance
     .
 inv freqGov:
  safetyLevel > 0 implies core -> forAll(
      freqGovernor = true)
 -- Applications of different safety
     critical level should be seperated in
     different VMs.
 inv safety:
  safetyLevel = vm.safetyLevel

context Core
 -- Unused cores should not be allocated.
 inv unuseCore:
  app -> size() = 0 implies vm -> size() = 0

context Vm
 -- Each VM shouid provide enough RAM and
     storage for hosted Applications.
  inv vmRam:
   app->collect(ramSize)->sum() = ramSize
  inv vmStorage:
   app->collect(storageSize)->sum() <=
       storageSize
context SDV
 -- RAM and storage assignment of VM should
     not exceed the limitation of the
     hardware platform
  inv sdvRam:
   board.ramSize -> sum() >= vm -> collect(
       ramSize) -> sum()
  inv sdvStorage:
   board.storageSize -> sum() >= vm ->
       collect(storageSize) -> sum()
```

LISTING 1. OCL constraints related to resource allocation

C. DESIGN SPACE EXPLORATION VIA EMF2SMT TRANSFORMATION

DSE describes the process of searching for system configurations that meet various requirements and constraints. With the system information defined in the model for hardware and software, as well as formally defined requirements and constraints, validation and optimization of system configurations can be automatically performed. We propose to transform formal system information into mathematical optimization problems in a solver-independent format. This paper utilizes EMF models and OCL constraints to describe system information formally. In our previous work [8], we developed the EMF2SMT transformation to convert these models into SMT problems. This is an exogenous and vertical model transformation that performs transformations on models expressed in different languages and residing at different abstraction levels [33]. We employ the SMT-Lib format to represent mathematical optimization problems. This format is a first-order logic-based, solver-independent standard for describing SMT problems, enabling verification of the satisfiability of mathematical formulas. In this work, we extend our EMF2SMT algorithm with the ability to label model constraints in SMT formulations, allowing conflicts to be tracked when no solution can be found under the given constraints. State-of-the-art optimization solvers can be used to explore the design space, identifying either optimal solutions or potential conflicts.

Algorithm 1 EMF2SMT Algorithm Extended From [8] for Checking Unsatisfiable Constraints

Require: OCL expressions, EMF models
for *class* **in** classes of the meta-model **do**
  for *ins* **in** instances of *class* in instance model **do**
    for *property* **in** properties of *class* **do**
      if *property* refers to a reference **then**
        Create binary decision variables for referred instances of *ins*
      else
        Create decision variable for *property* of *ins* according to the attribute type
      end if
      if *property* of *ins* is set **then**
        Assign values to decision variable
      end if
    end for
  end for
end for
for *expression* **in** OCL expressions **do**
  Create binary variables (assumptions) for tracking
  SMT expressions generated from *experssion*
  for *ins* **in** instances of context class **do**
    Set ins as context instance for *expression*
    res ← visitExpression(*expression*)
    Associate assumption variables to *res*
    Add tracked *res* to the list of SMT expressions
  end for
end for
return list of SMT expressions

The extended EMF2SMT algorithm is presented in Alg. 1. In the first step, we parse the EMF model instance. The information about classes and their properties is derived from the meta-model. In EMF, a property can be either an attribute representing values of various types, or a reference establishing connections to other elements. Within the instance model, every instance and its properties are iterated. Decision

variables are created for each property. For properties in form of attributes, the decision variables are formulated to match the attribute's type. For example, to represent the memory size of the first application, an integer variable *app*1_*ram* will be introduced. Given our focus on automotive resource allocation problem, our main consideration lies in attributes that can be quantitatively described. For the reference of each instance, we propose the creation of binary decision variables for every potential reference target instance. For example, to represent the assignment of cores to application 1, binary variables like *app*1_*core*1 could be used to establish a link to each potential core. Should a property be pre-defined with values in the instance model, these values will be directly incorporated as constants into the corresponding decision variables.

Once the model data is processed and decision variables are established, the OCL expressions and constrained instance will be analyzed. SMT expressions in SMT-Lib will be generated for each OCL expression and each constrained instance. We follow the same visitor-based algorithm proposed in our previous work [8] for the generation of SMT expressions. OCL expressions have a recursive structure, where a single expression can contain multiple nested expressions as sub-components. These expressions may vary in type. Some may refer to specific attributes or references in the model, while others contain operators to connect sub-components. The visitor pattern enables efficient handling of OCL expressions by iteratively and recursively processing each sub-component. During the transformation process, expressions that reference model attributes or elements are mapped directly to their values in the instance model, or decision variables are inserted when values need to be solved. For expressions containing operators, we convert OCL operators to equivalent SMT-Lib operators, allowing us to systematically combine values and decision variables from sub-components into valid SMT expressions. Since automobile is a complex system incorporating with various constraints, any conflict in constraints will lead to a unsolvable resource allocation problem. In order to tracking the satisfiability of each constraints, we further create a binary variable for each OCL constraints and associate it with all derived SMT expressions. By checking the state of these variables during problem solving and optimization, the conflict in constraints can be detected.

In the proposed workflow, the optimization solver verifies the syntactic correctness of the generated formulations during the subsequent solving and optimization steps. We do not discuss the semantic correctness of the generated SMT formulations in this paper, as that would involve verifying the correct implementation of the EMF2SMT transformation itself. However, to prevent any cascading effects from semantically incorrect SMT formulations, an additional model validation step can be introduced in practical use cases where accurate resource allocation is critical (as discussed in Section IV). This validation would take place before the deployment execution. In this step, tools such as Eclipse OCL [34] can be used to validate the instance model containing resource allocation solutions from the generated SMT expressions against the original OCL constraints. This ensures that any failure during SMT problem generation and optimization does not jeopardize the correctness of final deployment decisions.

The proposed transformation provides a flexible approach for converting various formal system specifications into optimization problems in SMT-Lib format. This flexibility means that the generated problem can vary in nature, such as continuous or discrete problem, depending on the system model, constraints, or optimization objectives. Therefore, we utilize the general-purpose optimization solver, Z3 [24], which can utilize different strategies to achieve the optimal solution for the target problem, especially for linear problems [35]. We reply on the capability of this solver to find optimal solution for generated problem. However, since the generated problem may include nonlinear constraints, and while Z3 provides certain methods to linearize these nonlinear terms for optimization, global optimality in such cases is not guaranteed [36]. Thus, if global optimum is required, further methods may need to be investigated. Since SMT-Lib is a solver-independent format, multiple optimization solvers can also be employed to solve and optimize the same problem, allowing users to compare results and analyze optimality. Alternatively, if the target problem has a fixed type and limited constraints, heuristic optimization strategies can be integrated into the workflow to enable faster solving and increase the likelihood of a finding global optimization. In the scope of this paper, we focus on the general DSE workflow, and thus do not discuss optimality assurance in detail. We consider that the optimization solver will either generate an appropriate resource allocation plan for subsequent automated steps based on the current configurations and constraints or provide users with feedback regarding any conflicts. In case of conflicts, users, typically system or integration engineers, will conduct manual analysis and adjustments to adapt models according to system requirements, resolving any conflicts in the system design.

D. TEXT2MODEL AND MODEL2TEXT TRANSFORMATION

Apart from the EMF2SMT transformation, we have developed additional model transformations for EMF model completion based on solver solutions and code generation for creating executable deployment files, as described in Section VI. In the proposed workflow (Fig. 4), we employ a state-of-the-art solver to determine an optimal system design, which is described through texts that include names of decision variables and their values. We implement the Text2Model Transformation to convert these text-based solutions into instance models.

Subsequently, we use rule-based Model2Text (M2T) transformations to convert instance models into text-like artifacts, such as source code and configuration descriptions. The

OMG MOFM2T specification [37] defines a language for transforming MOF models into text representations. In our implementation, we utilize Eclipse Acceleo 3 [38], an open-source implementation of OMG MOFM2T specifically designed for EMF-based models, to perform the transformation of the solution instance model into deployment files, which include configuration files for both VMs and containers. During the M2T process, input models and the code template are processed by the Acceleo core, resulting in the generation of target code. These input models encompass both the meta-model and the instance model. The code template defines transformation rules via the Acceleo Query Language (AQL) [39], which is a language used for navigating and querying EMF models. AQL shares a similar syntax with OCL. The Acceleo core is implemented as Java code.

Listing 2 presents the code template for generating Kubernetes container specification. The M2T program will go through the instance model and extract all required information for container deployment including image name, command, environments, etc. Furthermore, the allocation decision contained in the optimized instance model will be exposed, ensuring that application is scheduled on the target VM.

Listing 3 shows the generation rule design for creating VMs via KVM. The VM image and volume are created based on the information contained in the instance model, including VM name, OS and storage size. We utilize virt-install, a tool for based on the "libvirt" hypervisor management library, to provision new VMs. Given the information of core affinity, RAM size, devices including bus/port number in the instance model, the VMs will be generated accordingly.

```
containers:
  - name: [app.image/]
    image: >-
       gitlab.lrz.de:5005/autoware/[app.
           image/]:humble-arm64
    command:  ['['/][app.command/]']'/]
    args: ['['/]"[app.args/]"']'/]
    env:
    [for (env : AppEnv | app.containerenv)]
      - name: [env.name/]
        value: '[env.value/]'
    [/for]
    volumeMounts:
    [for (vol : Volume | app.volume)]
      - name: [vol.name/]
        mountPath: [vol.mountPath/]
    [/for]
nodeSelector:
    kubernetes.io/hostname: [app.vm/]
```

LISTING 2. M2T Template for container deployment (partial)

VI. VIRTUALIZATION-BASED DEPLOYMENT

After the system analysis and design, we proposed a virtualization-based SDV deployment. Virtualization brings flexibility and efficiency to cloud computing, which has been widely adopted in the field. Recently, it has garnered attention from both academia and industry for automotive systems. By leveraging both hardware and OS-level virtualization, the influences among different applications can be minimized. No single fault in containers or VMs can lead to the failure of the whole system.

```
vmname=tumvm[vm.id/]
ram=[vm.ramSize/]
cpus=[vm.core -> size()/]
os=[vm.os/]
...
# create a disk image for VM
qemu-img create -b /home/[vm.os/]-arm64.img
    -f qcow2 -F qcow2 $vmname.img [vm.
    storageSize/]G
# create VM volume
genisoimage -output cidata_$vmname.iso -V
    cidata -r -J user-data meta-data
# create VM
virt-install --name=$vmname --ram=$ram \
  --vcpus=$cpus --import \
  --disk path=$vmname.img, format=qcow2 \
  --disk path=cidata_$vmname.iso,device=
      cdrom \
  --os-variant=$os --network network=host-
      bridge  \
  [for (port: Port | vm.port)]
  [for (df: DeviceFunction | port.device.
      devicefunction)]
  --device=[port.bus.id/]_[port.id/]_[df.id/
      ]\
  [/for][/for]
  --noautoconsole
```

LISTING 3. M2T Template for VM deployment (partial)

In this paper, we introduce a centralized automotive E/E architecture with virtualization for SDVs (Fig. 7). The proposed architecture is based on a powerful computing platform equipped with proper communication channel for exchanging data with the development platform and the cloud infrastructure. Hypervisor is used to create isolated VM environments, equipping them with necessary computational resources partitioned from the physical system. Such resources include processors, memory, storage, network cards, and etc. Within each VM, applications are downloaded from the cloud registry and run within containers, allowing mixed-criticality applications to coexist on shared physical hardware without interfering with each other. Container Orchestrator can be utilized for the management of containers, enhancing the flexibility of software deployment and updates.

Fig. 8 illustrates the example system architecture, which will be used in Section VII for automatic platform configuration and software integration for SDV demonstration. The resource allocation decision is determined by the proposed automated MBSE procedure, which can be flexibly performed in an IDE on the development PC, or as an executable file on the cloud or directly on the SDV's HPC. Deployment is managed by virtualization and automation tools. We use a development PC to perform manual modeling tasks. The REST API facilitates the transmitting

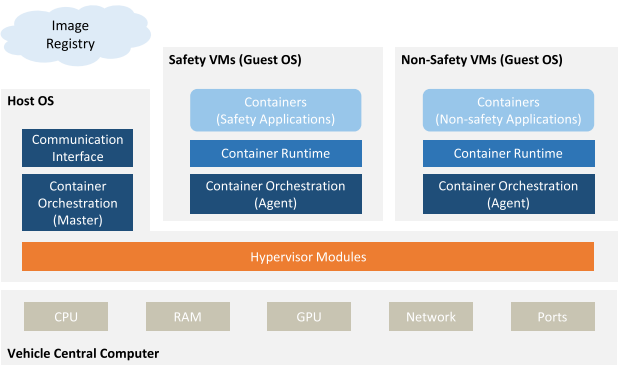

FIGURE 7. Virtualization-based SDV deployment. The SDV HPC is partitioned into multiple VMs. Applications are executed as containers within these VMs.

of software installation requests from the development PC to the target SDV platform. On the development PC, the Integrated Development Environment (IDE) functions as a REST API client, compiling deployment-related requests and forwarding them to the REST API server to carry out software installations or updates. During the SDV development phase, developers create system and constraint modeling within the IDE. As discussed in Section V, we propose a model-based approach to automate SDV system design. In the MBSE process, the EMF2SMT module transforms models and constraints into optimization models in SMT format. Afterwords, we employ the Z3 optimization engine [24] to calculate the solution. Ultimately, the resolved configurations for SDV resource allocation are converted into various configuration files for VMs and containers via Acceleo [38]. In our demonstration, the MBSE process is executed within the IDE, and the resulting deployment files are transferred to the SDV HPC for execution. However, the location of the MBSE process, whether on the development PC, SDV HPC, or cloud infrastructure, is adaptable to specific user scenarios. Alternate workflows are depicted with dashed boxes and lines in the figure. For instance, consider an on-the-fly software installation initiated by the customer, after selecting a new application in the app store, the vehicle system models can be automatically updated onboard by integrating information from the new application into the existing models. The vehicle can then analyze and plan the installation through the onboard MBSE process, reducing the need to formally request a vehicle upgrade from a dealership.

The SDV HPC is managed by the KVM hypervisor [40], an open-source hypervisor integrated within the Linux kernel. The host OS of KVM supports multiple isolated virtual environments and provides each VM with functionalities of the physical system. The REST API server, located in the host OS of the SDV HPC, receives requests from the client. Once the deployment files are ready, Libvirt [27], an open-source tool for managing platform virtualization, is employed to create and configure guest VMs. Within these VMs, we utilize K3S [41], a lightweight variant of Kubernetes [28], for container orchestration. The K3S server, running on the host OS, manages the containers across the system, serving as the control plane for container deployment. K3S agents within the VMs handle tasks such as creating, starting, stopping, and monitoring containers. Pods, which are the smallest deployable units in Kubernetes, host applications in the form of containers provided by the runtime. In our configuration, each pod is used to host a single container to simplify application management. An example of such pods would be those running the planning container for Autoware. The complete list of application containers can be obtained in Table 1. Prior to the container deployment activities, Ansible [26], an automation tool for system configurations, is used to establish the K3S environment on the guest VMs. This includes setting up Kubernetes nodes for deployment, configuring communication between Kubernetes nodes and the host OS, and installing K3S agents and container runtimes like Docker.

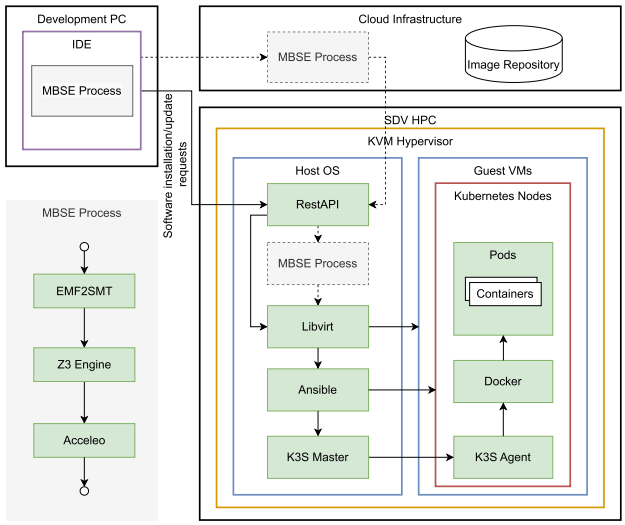

FIGURE 8. Example system architecture for implementing automated SDV design and deployment.

A detailed sequence diagram for deployment execution is presented in Fig. 9, once the host OS receives deployment files, it downloads the required VM image from the cloud registry and triggers Libvirt to create, configure, and start KVM VMs. After the VMs are initialized and running, the host OS scans their IPs, uses them to establish connections to the VMs, and prepares the execution environment inside VMs via Ansible. This setup includes the installation and configuration of K3S agents and container runtimes. When the VMs are ready, the K3S server within the host OS takes charge of orchestrating the container deployment within each VM. Individual container images are then downloaded to each VM and instantiated as applications.

VII. EXPERIMENT

In this section, we present a case study of SDV deployment using the proposed end-to-end workflow including the automated design space exploration, code generation and virtualization-based deployment.

As the development platform for conducting MBSE-based analysis and design, we employed a Window 10 PC equipped

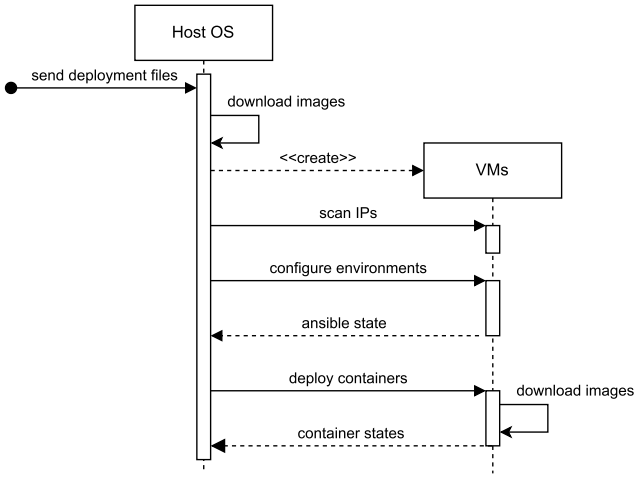

FIGURE 9. Sequence diagram of VM creation and container deployment.

with Intel i7-8568U CPU (four cores) and 40GB RAM as the development platform. The MBSE demonstration was conducted in Eclipse IDE 2022-06. The meta-model and instance model (presented in Section V) were developed using Eclipse Modeling Tools 4.24.0.20220609-1200. To define OCL constraints and objectives, we used text-based files with the assistance of the OCL All-In-One SDK 6.17.1.v20220309-0840. Additionally, the OCL visitor API and Java SE Development Kit version 16.0.1 were leveraged for implementing and executing EMF2OCL transformation algorithms. We utilized the optimization solver Z3 (version 4.8.17) to perform solving and optimization tasks. An open-source code generator, Acceleo (3.7.11.202102190929), was used for M2T transformation.

As the SDV platform, we introduced ADLINK AVA developer platform, an ARM-based HPC [10] as the SDV platform. This platform is equipped with 32 cores, 32 GB of RAM, 2 TB of storage and a NVIDIA RTX A6000 GPU. Out of the 32 CPU cores, 2 are reserved for system processes, while the remaining 30 cores are dedicated to running SDV software applications. We utilized KVM (6.2.0) to create VMs and k3s (v1.27.3+k3s1) to deploy and manage the containers. Ubuntu 22.04 was chosen as the operating system for both the host system, VMs and containers. In the demonstration, we employed container-based Autoware software from [22] within an AWSIM simulation environment as the target software. We chose closed-loop simulation to demonstrate the methodology in a realistic saftey-realted autonomous driving use case scenario. The software simulates a vehicle driving on a defined test route in Nishi-Shinjuku, Tokyo, Japan. It has been divided into eight container images, which we consider as SDV applications to be deployed. Additionally, we used the stress tool als a dummy container to represent non-safety applications. The requirements for applications are presented in Table 1. The specific numbers for each requirement, e.g., required RAM and storage size (in GB), are approximations based on the number of nodes in each container image. The containers were ranked by criticality and distributed to the appropriate VMs. The criticality of the controller was assumed to be the highest (level 2) because it sends control commands directly to the vehicle. As the last safety instance, the controller should be able to follow an emergency trajectory in the event of a system failure. In addition, the system container was considered critical because it is responsible for error monitoring of the software and must pass emergency commands to the controller. The remaining containers of the Autoware software were defined as level 1. It is important to be aware of the changing environment and to react to objects. The stress service, which is intended to emulate an infotainment system or similar, was given a safety level of 0. An error in this container has no safety-critical effect on the driving function.

Our demonstration contains three SDV scenarios, illustrated in Fig. 3. The first scenario (Fig. 3a) describes the end-to-end deployment of Autoware applications. Here, the Autoware applications and the SDV hardware were modeled using Eclipse Modeling Tools (Fig. 5, Fig. 6). Through our proposed approach, the deployment configurations of SDV were generated and executed. The second scenario (Fig. 3b) demonstrates the on-the-fly software update process in SDVs. Here, the planner container within the vehicle, which is part of an existing Autoware deployment, is updated to include a new speed limitation parameter that modifies the vehicle's driving behavior. The third scenario (Fig. 3c) also builds on the deployed SDV with Autoware software. Our goal is to deploy a new application on the operational SDV platform where a new platform configuration should applied. This process starts by integrating the new application into the solution instance model from the first scenario, without altering the current Autoware deployment. The DSE algorithm shoulld determine a new VM to host this application, since its requirement cannot be filled in the existing VMs containing Autoware.

In our experiments, we plotted vehicle velocity, steering angle, and the number of utilized cores as indicators of successful deployment.

A. MBSE PROCESS

Our end-to-end SDV design and deployment workflow begins with the model-based system analysis. The input information for the proposed approach, which includes the meta-model, instance model, OCL constraints, and optimization objectives, has partly been presented in Section V. This section will mainly introduce the steps and artifacts produced through the automated MBSE process for the software deployment scenario of eight Autoware applications (Fig. 3). Given the similarity of the artifacts for other demonstrated scenarios, those will not be discussed in detail here.

The enabler of the proposed automated DSE approach is the EMF2SMT, with which a mathematical optimization problem will be generated. Our experiments focus on the SDV software deployment decision, namely resource allocation. As discussed in Section VI, we propose a virtualization-based SDV deployment. Therefore, the resource allocation

TABLE 1. Application requirements. Requirements are derived and approximated from number and usage of contained nodes.

name (node number)	ram-Size	core-Size	storage-Size	safety-Level	device
Sensing (48)	5	6	15	1	-
Perception (49)	5	6	15	1	GPU
Localization (33)	5	4	12	1	-
Map (6)	1	2	3	1	-
Planning (25)	2	3	6	1	-
Control (8)	1	2	3	2	-
Vehicle (1)	1	1	3	1	-
System (21)	2	3	6	2	-
Stress (1)	1	2	3	0	-

involves allocating resources to VMs and mapping them to applications. Table 2 defines all related notations of presenting the generated optimization problem. In our experiments, following resource allocation problem has been generated based on the OCL constraints from Listing 1:

$$\min \sum_{v\in\mathcal{V}}(r_v > 0) \tag{1}$$

$$\text{s.t.} \sum_{v\in\mathcal{V}} x_{a\to v} = 1, \quad \forall a \in \mathcal{A} \tag{2}$$

$$\sum_{c\in\mathcal{C}} x_{a\to c} = r_a, \quad \forall a \in \mathcal{A} \tag{3}$$

$$x_{a\to c} \Longrightarrow \bigwedge_{v\in\mathcal{V}}((x_{a\to c} \wedge x_{c\to v}) = x_{a\to v}), \quad \forall a \in \mathcal{A}, c \in \mathcal{C} \tag{4}$$

$$(\sum_{a\in\mathcal{A}} x_{a\to c} = 0) \Longrightarrow (\sum_{v\in\mathcal{V}} x_{c\to v} = 0), \quad \forall c \in \mathcal{C} \tag{5}$$

$$x_{a\to d} \Longrightarrow \bigwedge_{v\in\mathcal{V}}((x_{a\to d} \wedge x_{p_d\to v}) = x_{a\to v}), \quad \forall a \in \mathcal{A}, d \in \mathcal{D} \tag{6}$$

$$\bigwedge_{c\in\mathcal{C}}(((s_a \geq 0) \wedge x_{a\to c}) \Longrightarrow (f_c = 1)), \forall a \in \mathcal{A} \tag{7}$$

$$\sum_{v\in\mathcal{V}} x_{a\to v} s_v = s_a, \quad \forall a \in \mathcal{A} \tag{8}$$

$$r_v \geq \sum_{a\in\mathcal{A}} x_{a\to v} r_a, \quad \forall v \in \mathcal{V} \tag{9}$$

$$R_v \geq \sum_{v\in\mathcal{V}} r_v \tag{10}$$

In the mathematical model, (1) ensures the minimum number of used VMs. (2) is generated from inv *appVM* guaranteeing the mapping of applications to VMs. (3) derives from inv *appCore* which restricts the core assignment to applications. (4) reflects inv *vmCore*. It indicates that required cores should be provided on the applications' host VM. (5) represents inv *unuseCore*, ensuring a minimal number of used cores. The allocation of devices (inv *appDevice*) is defined via (6). (7) refers to the constraint related to the configuration of cores (inv *freqGovernor*). It specifies that cores running critical application should be in performance model to guarantee the performance stability. (8) grantees FFI defined in inv *safety*. Applications of different safety levels will be separated into different VMs. (9) and (10) ensures the resource availability both in the VM level (inv *vmRam, vmStorage*) or the entire vehicle level (inv *sdvRam, sdvStorage*).

TABLE 2. Summary of notations.

Symbol	Meaning
$\mathcal{A}$	Set of applications
$\mathcal{C}$	Set of cores
$\mathcal{V}$	Set of VMs
$\mathcal{D}_a$	Set of external devices required by application a
R	Integer value expressing the total amount of available platform resources as quantitative numbers. In the example SDV, it includes size of RAM and storage.
a	Application $a \in \mathcal{A}$
c	Core $c \in \mathcal{C}$
v	Vm $v \in \mathcal{V}$
d	Device $d \in \mathcal{D}$
p_d	Port p that connects d
r_a	Integer variable representing amount of allocated resources in a. In the example SDV, it includes number of cores, size of RAM and storage.
r_v	Integer variable representing amount of allocated resources in v. In the example SDV, it includes size of RAM and storage.
s_a	Integer variable representing safety criticality of a
s_v	Binary variable representing safety criticality of v
f_c	Binary variable representing if core is in power mode (1) or energy save mode (0)
$x_{a\to v}$	Binary variables depicting if a is mapped on v
$x_{a\to c}$	Binary variables depicting if a is mapped on c
$x_{c\to v}$	Binary variables depicting if c is mapped on v
$x_{a\to d}$	Binary variables depicting if a requires d
$x_{p_d\to v}$	Binary variables depicting if p_d is assigned to v

The mathematical model is presented as SMT expressions in SMT-Lib format, which can be processed by state-of-the-art SMT solvers such as Z3. In the first deployment scenario involving eight Autoware applications, 491 variable declarations (declare-fun) and 426 assertions (assert) were generated. For the scenario of adding an additional application, 443 variable declarations and 462 assertions were created. The reduction in variable declarations is due to known values from the solution of the first scenario. The assertions include those extended from the previous scenario as well as new ones created for the additional application. Thus, previous configurations can be re-verified against the constraints, and additional configurations can be generated for the new application. In SMT, each assertion (assert) instantiates an OCL constraint against a specific model instance. It is a function that returns a Boolean value, formulated as a logical statement involving variables. For the readability of the generated SMT formulations, we assigned prefixes to each variable that reflect the related class, reference, or attribute name. The prefix "as" is assigned for the assumption variables (proposed in Alg. 1) for tracking unsatisfiable constraints. The prefix "ref" denotes boolean variables that represent reference relationships within the models. For example, "ref" is used as a boolean variable to denote the association between "appVII" and "vm1." Most

OCL operators have direct counterparts in SMT operators, allowing for straightforward transformations. Furthermore, we utilize the if-then-else (ite) operator in SMT to translate boolean variables into the numeric values 0 or 1. Listing 4 presents part of the generated SMT expressions for constraint *vmStorage*, *sdvRam*, *sdvStorage* and optimization goal.

```
(assert (=> as_vmStorage (<= (+  (* (ite
    ref_app4_vm1_e 1 0) 11) (* (ite
    ref_app2_vm1_e 1 0) 15) (* (ite
    ref_app1_vm1_e 1 0) 15) (* (ite
    ref_app6_vm1_e 1 0) 11) (* (ite
    ref_app5_vm1_e 1 0) 15) (* (ite
    ref_app0_vm1_e 1 0) 15) (* (ite
    ref_app3_vm1_e 1 0) 12) (* (ite
    ref_app7_vm1_e 1 0) 11)) (* (ite (and
    true ) 1 0) storageSize_vm1_e))))
(assert (=> as_sdvRam (>= (+  (* (ite
    ref_board0_sdv0_e 1 0) 32)) (+  (* (ite
    true 1 0) ramSize_vm1_e) (* (ite true 1
    0) ramSize_vm3_e) (* (ite true 1 0)
    ramSize_vm2_e)))))
(assert (=> as_sdvStorage (>= (+  (* (ite
    ref_board0_sdv0_e 1 0) 1024)) (+  (* (ite
     true 1 0) storageSize_vm3_e) (* (ite
    true 1 0) storageSize_vm2_e) (* (ite true
     1 0) storageSize_vm1_e)))))
(minimize (+  (ite (> (* (ite (and true ) 1
    0) ramSize_vm1_e) 0) 1 0) (ite (> (* (ite
     (and true ) 1 0) ramSize_vm3_e) 0) 1 0)
    (ite (> (* (ite (and true ) 1 0)
    ramSize_vm2_e) 0) 1 0)))
```

LISTING 4. Generated SMT expressions (partial)

Utilizing the Z3 optimization solver, the results of the design space exploration were calculated. In the scenario for deploying Autoware software, we firstly introduced a conflict in the instance model by specifying an insufficient storage size on the board. The solver detected the unsatisfiability in 26 ms and identified the conflict within the *sdvStorage* constraint. Since our work does not discuss heuristic methods for conflict resolution, conflict analysis remains a manual process as presented in Fig. 4. Typically, constraints in conflict are interdependent, and multiple approaches can be taken to resolve conflicts depending on project-specific needs or the priority of requirements. For example, the conflict in *sdvStorage* constraint can be resolved by reducing the application number or increasing the storage capacity. Such adjustments should be determined by engineers. The corresponding field in the instance model (an example of instance model is shown in Fig. 6) should be manually modified. After adjusting the storage size of the board to an adequate level in the instance model, the optimal solution was calculated in 452 ms and subsequently reintegrated into the model. For the software update scenario, since there were no changes to resource allocation-related parameters, the solving procedure merely verified the existing solution with the new parameters in the planer application. The verification was accomplished in 82 ms. In the scenario involving an additional application, extending the existing solution from the first scenario was accomplished in 125 ms. The deployment solutions of VMs and applications are illustrated in Fig. 3.

In the subsequent process, configuration files for VMs and containers were generated according to the solving solution and following the template described in Section V-D. These files were then transmitted and used in the SDV HPC for the creation and configuration of VMs and containers. The deployment behavior will be discuss in Section VII-B.

B. SDV DEPLOYMENT SCENARIOS

After the deployment files have been generated, these were executed on the SDV hardware to deploy SDVs and containers. The deployment follows the procedure described in Fig.9. Three different SDV scenarios will be demonstrated: firstly, the deployment of Autoware applications (Fig. 3a); secondly, the on-the-fly update of the planner container in the existing deployment (Fig. 3b); and thirdly, the addition of the stress container in the existing deployment, which led to the creation of a new VM to host this new application (Fig. 3c). The MBSE process was conducted on the development PC (as illustrated in Fig. 9). Each change in the existing SDV deployment was triggered by newly generated deployment files. The simulated vehicle in all scenarios was given the same start point and destination. There was no obstacles planned on the street. The driving vehicle in the experiment was visualized through Rivz (Fig. 10). In the picture, the start point and destination are located at the top right and bottom left. We measured the core utilization of the HPCs (via Ubuntu htop), as well as the speed and steering angle of the simulated vehicle (via ROS), to illustrate the deployment status. Given the unpredictability of the downloading process due to large container images, we created the VMs in advance and pre-downloaded all necessary images into them respectively. The measurement in the experiment begins from the start of VMs.

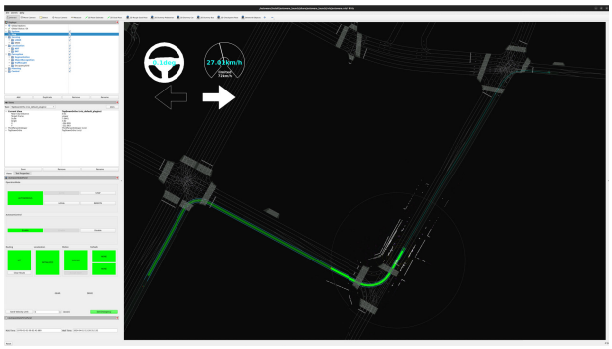

FIGURE 10. Visualization of the driving vehicle in the simulation. The vehicle is approaching an intersection at a speed of 27.01 km/h.

1) SCENARIO 1: AUTOWARE DEPLOYMENT (FIG. 3a)

This experiment describes the initial deployment of eight Autoware applications. Two VMs were created on the ARM-based HPCs platform, with applications deployed and

run according to their individual requirements. Fig. 11 presents the state of the simulated vehicle and the core utilization of the HPCs during software deployment and execution. The first 40 s marked the startup of two VMs and eight Autoware containers. This period describes the initialization of simulated vehicle, thus signals for velocity and steering angle were not available (presented as dashed lines). At same time, the planner application was performing calculations for a global route. During this period, the number of utilized cores of the HPCs reached nearly 28, which is the maximal capability of the allocated platform resources, as initialization and global route calculation are computationally intensive. After successful initialization, information on velocity and steering angle could be retrieved. Here, we simulated a driving function that needs manual trigger. At time t_1, we activated the driving function through Rviz, and the vehicle began to move.Changes in vehicle velocity and steering angle indicate that the vehicle was passing through crossroads. Three crossroads were planned in the simulator (Fig. 10). The entire driving period lasted approximately 80 seconds. Since there were no obstacles on the driveway in the simulation and no intensive calculations were performed during driving, the average number of utilized cores during driving was about 14. After the vehicle reached its destination, we stopped the Autoware, resulting in a corresponding decrease in the core utilization of the HPCs. This scenario provides a reference measurement for the deployment and execution of Autoware, which we use as a benchmark for other experimental scenarios.

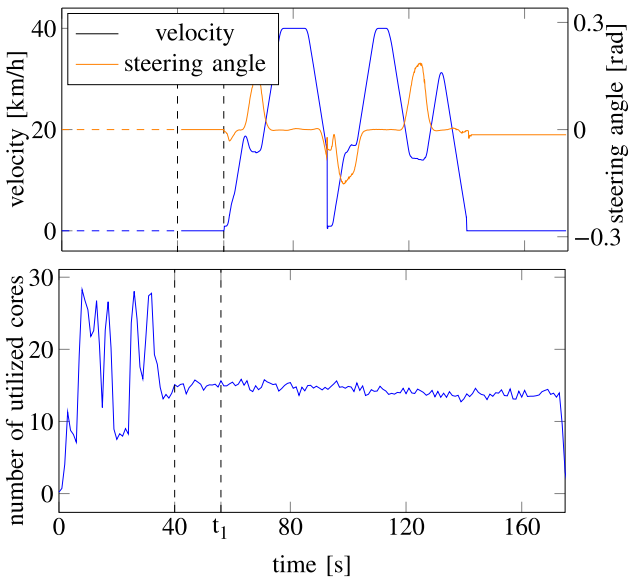

FIGURE 11. Vehicle states and core utilization during the Autoware deployment. Time t_1 presents the moment when the vehicle started to move.

2) SCENARIO 2: UPDATE OF PLANNER APPLICATION (FIG. 3b)

Fig.12 illustrates the vehicle status during the update of the planner container. In this scenario, the maximal vehicle speed was initially set to 20 km/h in the planner container. After the software update, the new planner defined a speed limit of 40 km/h. During the initial deployment phase (from time 0 to t_2), it can be observed that both core utilization and steering angle exhibit the same trends as those shown in Fig.11. However, the maximal vehicle speed remained within the predefined limit of 20 km/h. Once the speed stabilized at this limit, we manually stopped the vehicle at time t_2 and initiated the software update by applying a new configuration file (generated via the proposed workflow), which included information about the new planner container where the speed limit was increased to 40 km/h. Between t_2 and t_3, the planner was updated and recalculated the vehicle's route with updated parameters. As previously discussed, performing path planning is a calculation-intensive task. Thus, the HPCs core utilization once again reached maximum, indicating that the allocated 28 cores were briefly fully loaded. After time t_3, the vehicle resumed driving and reached its maximal speed of 40 km/h. The resulting measurements are similar to those shown in Fig. 11.

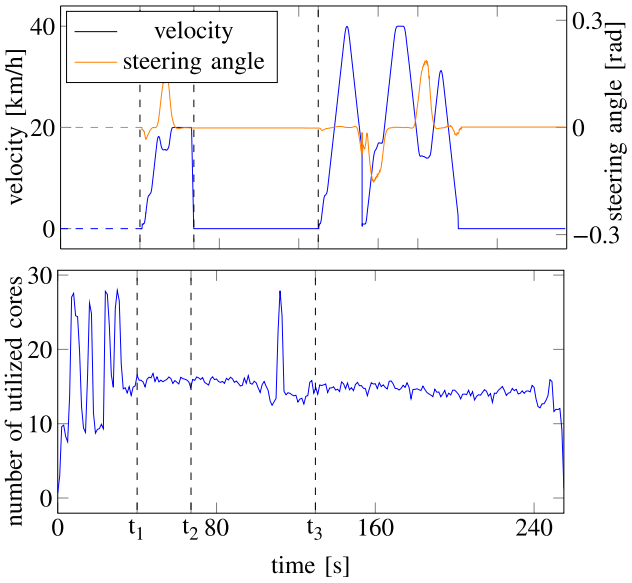

FIGURE 12. Vehicle states and core utilization during the update of planner container. At t_1, the vehicle started to move; at t_2, the vehicle was stopped to update its planner; at t_3, the vehicle continued driving after the software update.

3) SCENARIO 3: PLATFORM RECONFIGURATION FOR THE ADDITIONAL APPLICATION (FIG. 3c)

Fig. 13 depicts a scenario where the HPCs platform needed reconfiguration. A third VM was initialized to host an additional container while the vehicle was running in the simulation environment. We utilized the stress tool from Ubuntu to demonstrate a non-safety critical application, which occupied two cores of the HPCs. The deployment of Autoware applications followed the experiment from scenario 1. Consequently, similar trends in velocity, steering angle, and core utilization can be observed from time 0 to t_2. At t_2, the additional VM was started, and the stress tool was

deployed on it, leading to an increase in number of cores by approximately two. These cores were utilized by the new VM and occupied by the stress container. Despite these changes in overall core utilization, the vehicle's driving states were not influenced and exhibited the same trends as in Fig. 11.

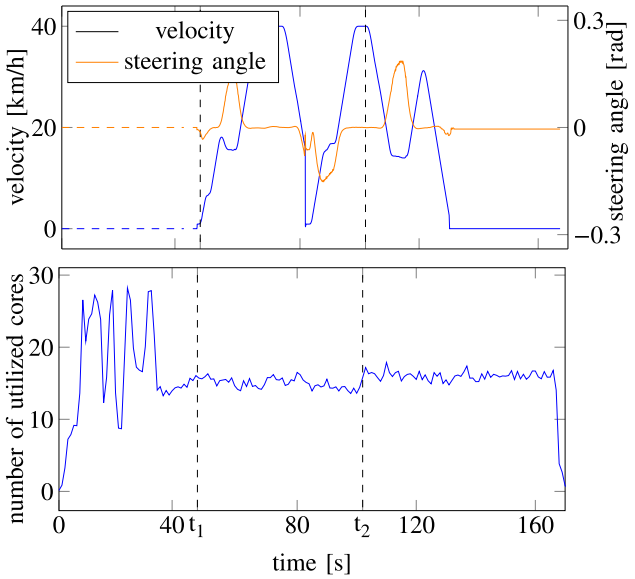

FIGURE 13. Vehicle states and core utilization during the deployment of an additional application. At t_1, the vehicle started to move; at t_2, the third VM was initialized and the additional application was deployed on it.

C. FURTHER DISCUSSIONS

The demonstration in previous subsections illustrates the feasibility and applicability of our approach for the deployment design of future SDVs. In this part, we further analyze the scalability of the proposed approach and compare bare-metal deployment with virtualization-based deployment.

1) SCALABILITY ANALYSIS OF THE PROPOSED DSE APPROACH

The proposed automated DSE procedure is designed to be compatible with both design-time and runtime development. While the scalability of virtualization technologies has been proven in cloud computing [42], our focus here is on evaluating the scalability of the DSE approach itself, including transforming model information into mathematical formulation, e.g., as SMT problem, and utilizing general sense optimization solvers, e.g., Z3 for problem solving and optimization. In this experiment, we increased the complexity of the scenario and analyzed the time required for problem generation and optimization.

Starting with our Autoware deployment demonstration, we scaled the problem by increasing the number of Autoware applications while ensuring solvability by adjusting hardware resources accordingly. To do this, we modified the instance model by creating additional application and CPU core instances, which increased the size of the generated SMT expressions. We modified other hardware properties, such as RAM and storage, by adjusting attributes in related instances.

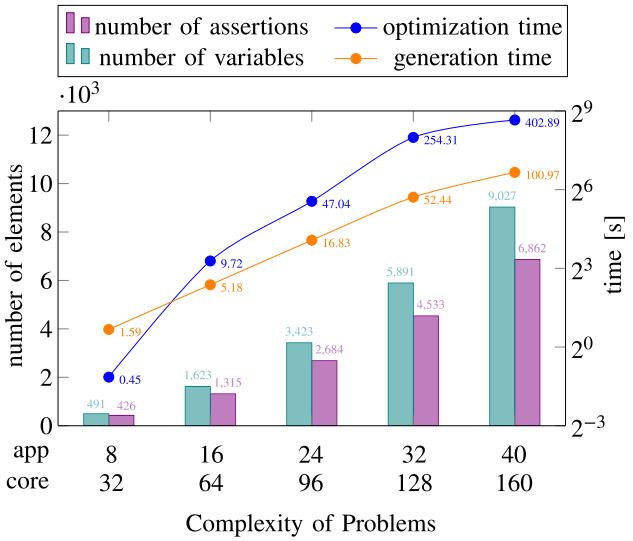

FIGURE 14. Scalability analysis of the proposed DSE approach. As the number of model instances for applications and cores increases, the generated formulations rise, leading to longer generation and optimization time.

However, these adjustments did not impact the number of SMT formulations generated. To minimize uncertainties in our scalability analysis, we maintained a consistent set of constraints and optimization goals.

In the experiment, we analyzed the size of the generated SMT problem, including variable declarations and assertions, the generation time for SMT formulations, and the optimization time for increased problem sizes. The results, shown in Fig. 14, indicate that as we multiplied the number of applications and required cores, the number of variable declarations and assertions increased linearly. However, the generation and optimization times increased approximately exponentially. For instance, while the original scenario with 8 applications and 32 cores was solved in 452 ms, scaling to 40 applications and 160 cores raised the optimization time to 400 s.

This benchmark experiment shows that as the number of model components (e.g., applications and cores) increases, the optimization solver's processing time grows exponentially. While this is manageable for design phase development, which allows for a relatively long processing time in complex systems, runtime solving demands faster decisions and integration. To meet runtime requirements, the flexibility of processing may need to be reduced by setting fixed constraints and pre-configurations, or introducing heuristic methods for faster problem solving.

2) BARE-METAL VS. VIRTUALIZATION-BASED AUTOWARE DEPLOYMENT

The previous SDV scenarios demonstrate the effectiveness and flexibility of the proposed virtualization-based deployment strategy. To further examine this, we compared Autoware deployment directly on bare-metal hardware and within a virtualization environment.

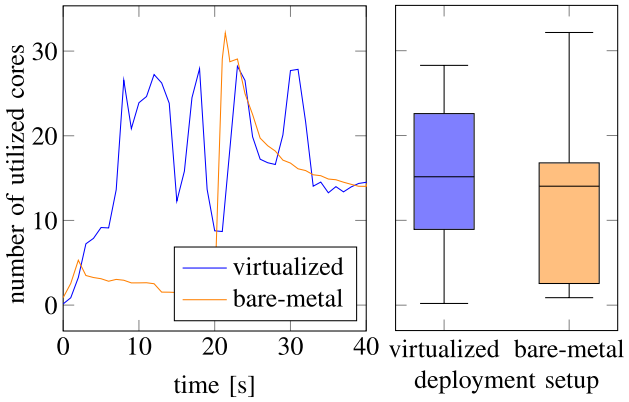

FIGURE 15. CPU utilization of Automare deployment on different system setups. The line chart compares the utilization over time on the bare-metal and in the virtualized environment. The box plot shows the average and distribution.

In Fig. 15, we compare CPU core utilization for both setups over the first 40 s of deployment, a period during which Autoware completes initialization. The virtualization-based deployment into containers hosted by VMs resulted in multiple CPU utilization peaks due to the initialization of virtualization components, including VM startup and container creation. The complete initialization of Autoware in the virtualization environment took approximately 40 s. In comparison, the CPU utilization curve for bare-metal deployment showed only a single peak, with CPU consumption remaining relatively low throughout the rest of the period. Without the overhead of additional components such as VMs, bare-metal deployment naturally achieves faster initialization of Autoware compared to virtualization-based deployment. The box plot indicates that average core usage in the virtualized environment is slightly higher than in bare-metal. However, the maximum number of utilized cores in virtualization (around 28) is lower than in bare-metal (around 32). This highlights an advantage of virtualization-based deployment in resource allocation and limitation. In the virtualized setup, each software component is allocated a fixed amount of resources, preventing interference among components and with the rest of system. In bare-metal deployment, each software component can access as many resources as needed, potentially leading to resource competition.

Additional benchmark experiments comparing bare-metal and virtualization-based deployments are presented in our other works [22], [43]. These studies include general performance evaluations of CPU, memory, and network metrics, demonstrating that virtualized environments achieve performance metrics similar to bare-metal systems [43]. We also analyzed startup time (deployment and initialization time), showing that Autoware's startup time in a virtualized environment is slightly slower than on bare-metal [43]. Further analysis of end-to-end latency indicates that Autoware deployed in containers exhibits better latency compared to its performance on bare-metal [22]. These benchmarks demonstrate that virtualization is a suitable concept for software deployment in SDVs. Given that automotive platforms are safety-critical, developing safety-certified virtualization software for automotive deployment is worthwhile.

VIII. CONCLUSION

In this paper, we propose an end-to-end workflow that addresses resource allocation and software deployment for future SDVs. This workflow starts with the system modeling and formal constraint specification by users. Through a novel MBSE procedure, we proceed to analyze and design the resource allocation on the target platform, which includes configuration for VMs and containers. Subsequently, a template-based mechanism is utilized to generate deployment files, which are transmitted and executed on the SDV platform. The SDV HPC incorporates two levels of virtualization to host applications of varying criticality levels and requirements. This is accomplished by creating and configuring VMs and deploying applications within containers, thereby reducing the potential for interference between applications. Container orchestration tools are employed to monitor and manage all containers, ensuring efficient operation. The feasibility and flexibility of our workflow is demonstrated through a case study involving a reference SDV HPC hardware setup, along with the use of open-source autonomous driving software and simulation tools. Three scenarios, including software deployment, software update, and platform reconfiguration, have been demonstrated.

In terms of future research directions, as generative AI shows its ability to undertake diverse tasks, we plan to investigate the analysis of automotive system information and requirements through large language models. This will enable the automatic generation of system models and formal constraints, allowing engineers without specialized modeling skills to the design and deploy an SDV system.

ABBREVIATIONS

IDE	Integrated Development Environment
M2T	Model2Text
AQL	Acceleo Query Language
FFI	Freedom from Interference
SysML	Systems Modeling Language
UML	Unified Modeling Language
MOF	Meta Object Facility
OMG	Object Management Group
EMF	Eclipse Modeling Framework
OTA	Over-the-Air
OCL	Object Constraint Language
MILP	Mixed Integer Linear Programming
SMT	Satisfiability Modulo Theories
ILP	Integer Linear Programming
DSL	Domain-Specific Language
ADAS	Advance Driver-Assistance System
ECU	Electronic Control Unit
MBSE	Model-Based System Engineering
DSE	Design Space Exploration
VM	Virtual Machine
HPC	High-Performance Computer

E/E Electrical and Electronic
SDV Software-Defined Vehicle
OS Operating System
RAM Random-Access Memory

REFERENCES

[1] *Software-defined Vehicles—A Forthcoming Industrial Evolution*. Accessed: Mar. 5, 2024. [Online]. Available: https://www2.deloitte.com/cn/en/pages/consumer-business/articles/software-defined-cars-industrial-revolution-on-the-arrow.html
[2] L. Wen, M. Rickert, F. Pan, J. Lin, and A. Knoll, "Bare-metal vs. hypervisors and containers: Performance evaluation of virtualization technologies for software-defined vehicles," in *Proc. IEEE Intell. Vehicles Symp. (IV)*, Jun. 2023, pp. 1–8.
[3] G. Shirasat and S. Marzani, "Accelerating software-defined vehicles through cloud-to-vehicle edge environmental parity," ARM, Cambridge, U.K., AWS Whitepaper, 2022. [Online]. Available: https://www.researchgate.net/publication/358275623_Accelerating_Software-Defined_Vehicles_through_Cloud-To-Vehicle_Edge_Environmental_Parity and https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/arm-aws-edge-environmental-parity-wp.pdf
[4] N. Nayak, D. Grewe, and S. Schildt, "Automotive container orchestration: Requirements, challenges and open directions," in *Proc. IEEE Veh. Netw. Conf. (VNC)*, Apr. 2023, pp. 61–64.
[5] P. Karle et al., "EDGAR: An autonomous driving research platform—From feature development to real-world application," 2023, *arXiv:2309.15492*.
[6] U. Pohlmann and M. Hüwe, "Model-driven allocation engineering: Specifying and solving constraints based on the example of automotive systems," *Automated Softw. Eng.*, vol. 26, no. 2, pp. 315–378, Jun. 2019.
[7] J. Eder, S. Voss, A. Bayha, A. Ipatiov, and M. Khalil, "Hardware architecture exploration: Automatic exploration of distributed automotive hardware architectures," *Softw. Syst. Model.*, vol. 19, no. 4, pp. 911–934, Jul. 2020.
[8] F. Pan, J. Lin, M. Rickert, and A. Knoll, "Automated design space exploration for resource allocation in software-defined vehicles," in *Proc. IEEE Intell. Vehicles Symp. (IV)*, Jun. 2023, pp. 1–8.
[9] F. Pan, J. Lin, and M. Rickert, "Automatic platform configuration and software integration for software-defined vehicles," 2024, *arXiv:2408.02127*.
[10] *Soafee for Software Defined Vehicles*. Accessed: Feb. 19, 2024. [Online]. Available: https://www.adlinktech.com/en/soafee
[11] *Autoware—The World's Leading Open-source Software Project for Autonomous Driving*. Accessed: Mar. 5, 2024. [Online]. Available: https://github.com/autowarefoundation/autoware
[12] *Open Source Simulator for Self-Driving Vehicles*. Accessed: Mar. 5, 2024. [Online]. Available: https://github.com/tier4/AWSIM
[13] Z. Liu, W. Zhang, and F. Zhao, "Impact, challenges and prospect of software-defined vehicles," *Automot. Innov.*, vol. 5, no. 2, pp. 180–194, Apr. 2022.
[14] G. Keßler, D. Sieben, A. Bhange, and E. Börner, "The software defined vehicle–technical and organizational challenges and opportunities," in *Proc. Int. Stuttgart Symp.* Wiesbaden, Germany: Springer-Vieweg, 2023, pp. 414–426.
[15] I. Al-Azzoni, J. Blank, and N. Petrović, "A model-driven approach for solving the software component allocation problem," *Algorithms*, vol. 14, no. 12, p. 354, Dec. 2021.
[16] T. Yamauchi, Y. Yamaguchi, T. Kono, and H. Hidaka, "Embedded flash technology for automotive applications," in *IEDM Tech. Dig.*, Dec. 2016, pp. 28.6.1–28.6.4.
[17] R. Prashanth, "Improving the flashing speed of electronic control unit in automotive industries," in *Proc. Innov. Power Adv. Comput. Technol. (i-PACT)*, vol. 1, Mar. 2019, pp. 1–6.
[18] M. Kathiresh, R. Neelaveni, M. A. Benny, and B. J. S. Moses, "Vehicle diagnostics over internet protocol and over-the-air updates," in *Automotive Embedded Systems*. Cham, Switzerland: Springer, 2021, pp. 89–100, doi: 10.1007/978-3-030-59897-6_5.
[19] D. Gangadharan, J. H. Kim, O. Sokolsky, B. Kim, C.-W. Lin, S. Shiraishi, and I. Lee, "Platform-based plug and play of automotive safety features: Challenges and directions (invited paper)," in *Proc. IEEE 22nd Int. Conf. Embedded Real-Time Comput. Syst. Appl. (RTCSA)*, Aug. 2016, pp. 76–84.
[20] J.-W. Yoo, Y. Lee, D. Kim, and K. Park, "An Android-based automotive middleware architecture for plug-and-play of applications," in *Proc. IEEE Conf. Open Syst.*, Oct. 2012, pp. 1–6.
[21] A. K. S. Rajan, A. Feucht, L. Gamer, I. Smaili, and M. N. Devi, "Hypervisor for consolidating real-time automotive control units: Its procedure, implications and hidden pitfalls," *J. Syst. Archit.*, vol. 82, pp. 37–48, Jan. 2018.
[22] T. Betz, L. Wen, F. Pan, G. Kaljavesi, A. Zuepke, A. Bastoni, M. Caccamo, A. Knoll, and J. Betz, "A containerized microservice architecture for a ROS 2 autonomous driving software: An end-to-end latency evaluation," in *Proc. IEEE 30th Int. Conf. Embedded Real-Time Comput. Syst. Appl. (RTCSA)*, Aug. 2024, pp. 57–66.
[23] *Object Constraint Language Version 2.4*, OMG, Needham, MA, USA, Feb. 2014.
[24] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Berlin, Germany: Springer, 2008, pp. 337–340.
[25] M. Masse, *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. Sebastopol, CA, USA: O'Reilly Media, 2011.
[26] L. Hochstein and R. Moser, *Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way*. Sebastopol, CA, USA: O'Reilly Media, 2017.
[27] *Libvirt Virtualization API*. Accessed: Feb. 9, 2024. [Online]. Available: https://libvirt.org/
[28] *Kubernetes*. Accessed: Feb. 19, 2024. [Online]. Available: https://kubernetes.io/
[29] J.-P. Steghöfer, B. Koopmann, J. S. Becker, M. Törnlund, Y. Ibrahim, and M. Mohamad, "Design decisions in the construction of traceability information models for safe automotive systems," in *Proc. IEEE 29th Int. Requirements Eng. Conf. (RE)*, Sep. 2021, pp. 185–196.
[30] *OMG Meta Object Facility (MOF) Core Specification Version 2.5.1*, OMG, Needham, MA, USA, Oct. 2019.
[31] R. F. Paige, D. S. Kolovos, and F. A. C. Polack, "A tutorial on metamodelling for grammar researchers," *Sci. Comput. Program.*, vol. 96, pp. 396–416, Dec. 2014.
[32] *Road Vehicles—Functional Safety*, Standard ISO 26262:2018, 2018.
[33] M. Brambilla, J. Cabot, M. Wimmer, and L. Baresi, *Model-Driven Software Engineering in Practice*, 2nd ed., Cham, Switzerland: Springer, 2017.
[34] *Eclipse OCL (Object Constraint Language)*. Accessed: Nov. 14, 2024. [Online]. Available: https://projects.eclipse.org/projects/modeling.mdt.ocl
[35] N. Bjørner, A.-D. Phan, and L. Fleckenstein, "*νZ*—An optimizing SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Germany: Springer, 2015, pp. 194–199.
[36] *Soundness of Non-linear Optimization*. Accessed: Nov. 14, 2024. [Online]. Available: https://github.com/Z3Prover/z3/issues/2247
[37] *MOF Model To Text Transformation Language, V1.0*, OMG, Needham, MA, USA, Jan. 2008.
[38] *Generate Anything From Any EMF Model*. Accessed: Mar. 25, 2024. [Online]. Available: https://eclipse.dev/acceleo/
[39] *Acceleo Query Language*. Accessed: Jul. 19, 2024. [Online]. Available: https://eclipse.dev/acceleo/documentation/aql.html
[40] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The Linux virtual machine monitor," in *Proc. Linux Symp.*, Ottawa, ON, Canada, 2007, vol. 1, no. 8, pp. 225–230.
[41] *Lightweight Kubernetes*. Accessed: Feb. 19, 2024. [Online]. Available: https://k3s.io/
[42] L. Qian, Z. Luo, Y. Du, and L. Guo, "Cloud computing: An overview," in *Cloud Computing*, M. G. Jaatun, G. Zhao, and C. Rong, Eds., Berlin, Germany: Springer, 2009, pp. 626–631.
[43] L. Wen, M. Rickert, F. Pan, J. Lin, Y. Zhang, T. Betz, and A. Knoll, "Virtualization & microservice architecture for software-defined vehicles: An evaluation and exploration," Tech. Rep., Dec. 2024.

FENGJUNJIE PAN received the B.Eng. degree in electrical engineering from Hamburg University of Applied Sciences, in 2016, and the M.Sc. degree in electrical engineering from the Technical University of Berlin, in 2019. He is currently pursuing the Ph.D. degree with the Chair of Robotics, Artificial Intelligence, and Embedded Systems, Technical University of Munich (TUM). Afterwards, he worked on continuous integration and testing at Magna Telemotive Munich, until 2021. He is currently a Research Assistant at the Chair of Robotics, Artificial Intelligence, and Embedded Systems, TUM. His research interests include automotive systems and software engineering, and applying generative AI to this area.

MARKUS RICKERT (Member, IEEE) received the academic degree (Dipl.-Inf. Univ., M.Sc. equiv.) in computer science and the Ph.D. degree (Dr.rer. nat.) (summa cum laude) from the Technical University of Munich (TUM), in 2004 and 2011, respectively. From 2010 to 2021, he was with fortiss, an affiliated institute of TUM, where he founded the Virtual Engineering and Robotics Research Group and the Autonomous Driving Research Group. At fortiss, he was the Deputy Head of Cyber-Physical Systems, from 2013 to 2016, and the Head of Robotics and Machine Learning, from 2016 to 2021. From 2021 to 2023, he was at TUM and the Head of the TUM/Huawei Intelligent Automotive Solutions Innovation Laboratory. Since 2023, he has been a Full Professor and the Chair of Multimodal Intelligent Interaction at the University of Bamberg. His research interests include robotics, human–robot interaction, cognitive systems, artificial intelligence, multimodal interaction, motion planning, advanced systems engineering, and software engineering.

TOBIAS BETZ (Graduate Student Member, IEEE) received the B.Eng. degree from the Regensburg University of Applied Sciences, in 2018, and the M.Sc. degree from the Technical University of Munich (TUM), in 2020, where he is currently pursuing the Ph.D. degree with the Institute of Automotive Technology. His research interests include ROS 2, latency analysis, system optimization, and software deployment on real-world applications in autonomous driving.

LONG WEN (Graduate Student Member, IEEE) received the B.S. degree from the School of Automotive Engineering, Wuhan University of Technology, Wuhan, China, in 2017, and the M.S. degree from the School of Automotive Engineering, Jilin University, Changchun, China, in 2020. He is currently pursuing the Ph.D. degree with the Chair of Robotics, Artificial Intelligence and Real-Time Systems, School of Computation, Information and Technology, Technical University of Munich, Germany. His research interests include autonomous driving, cloud robotics, software engineering, and deep learning.

JIANJIE LIN received the Ph.D. degree in computer science from the Technical University of Munich (TUM), in 2022. He was a Research Associate with the Chair of Robotics, Artificial Intelligence and Real-Time Systems, TUM, where he has been a Research Assistant, since June 2017. During his studies, he engaged in collaborative research projects with Fortiss GmbH and German Aerospace Center (DLR), focusing on autonomous systems, motion and trajectory planning, and computer vision. His current research interests include robotics, machine learning, computer vision, software-defined vehicles, autonomous driving, grasp planning, and embodied AI.

NENAD PETROVIC was born in Pirot, Serbia, in1992. He received the bachelor's degree in computer science and informatics from the Faculty of Electronic Engineering, in 2015, and the Laurea Magistrale degree in computer engineering from Politecnico di Milano, Milan, Italy. During his Ph.D. studies, he was a Teaching Assistant at the Faculty of Electronic Engineering. He is currently a Postdoctoral Researcher and a Scientific Coordinator/Supervisor at the Chair of Robotics, Artificial Intelligence and Real-Time Systems, Technical University of Munich (TUM), focused on automotive-related projects in the area of LLM-driven software development. His research interests include model-driven software engineering, semantic technology, the IoT, and large language models (LLM).

MARKUS LIENKAMP (Member, IEEE) received the degree in mechanical engineering from TU Darmstadt and Cornell University and the Ph.D. degree from TU Darmstadt, in 1995. He is currently a Professor with the Institute of Automotive Technology, Technical University of Munich (TUM). He has been the Head the Institute of Automotive Technology, TUM, since November 2009. He carries out research in the area of autonomous vehicles with the objective of creating an open-source software platform. He worked at Volkswagen as part of an international trainee program and took part in a joint venture between Ford and Volkswagen in Portugal. Returning to Germany, he led the Brake Testing Department, VW Commercial Vehicle Development Section, Wolfsburg. He was later appointed as the Head of the Electronics and Vehicle Research Department, Volkswagen Group's Research Division.

ALOIS KNOLL (Fellow, IEEE) received the M.Sc. degree in electrical/communications engineering from the University of Stuttgart, Stuttgart, Germany, in 1985, and the Ph.D. degree (summa cum laude) in computer science from the Technical University of Berlin (TU Berlin), Berlin, Germany, in 1988. He was on the Faculty of the Computer Science Department, TU Berlin, until 1993. He joined the University of Bielefeld, Bielefeld, Germany, as a Full Professor, where he was the Director of the Technical Informatics Research Group, until 2001. Since 2001, he has been a Professor with the Department of Informatics, Technical University of Munich (TUM), Munich, Germany. His research interests include cognitive, medical, and sensor-based robotics; multi-agent systems; data fusion; adaptive systems; multimedia information retrieval; model-driven development of embedded systems, with applications to automotive software and electric transportation; and simulation systems for robotics and traffic. He was on the board of directors of the Central Institute of Medical Technology at TUM (IMETUM). From 2004 to 2006, he was the Executive Director of the Institute of Computer Science, TUM. From 2007 to 2009, he was a member of the EU's highest advisory board on information technology, ISTAG, the Information Society Technology Advisory Group, and a member of its subgroup on future and emerging technologies (FET). In this capacity, he was actively involved in developing the concept of the EU's FET Flagship projects.