Systems Engineering

WILEY

[GKM+25] C. Granrath, C. Kugler, J. Michael, B. Rumpe, L. Wachtmeister:
Generating Logical Architectures from SysML Behavior Models.
In: Systems Engineering, Volume 28(6), pp. 762-778, DOI 10.1002/sys.70005, Wiley, Nov. 2025.

REGULAR ARTICLE OPEN ACCESS

Generating Logical Architectures from SysML Behavior Models

Christian Granrath[1] | Christopher Kugler[2] | Judith Michael[2] | Bernhard Rumpe[2] | Louis Wachtmeister[1,2]

[1]Systems Engineering, FEV.io GmbH, Aachen, Germany | [2]Software Engineering, RWTH Aachen University, Aachen, Germany

Correspondence: Louis Wachtmeister (wachtmeister@se-rwth.de)

Received: 12 November 2023 | **Revised:** 29 March 2025 | **Accepted:** 11 July 2025

Keywords: architecture framework | model-based systems engineering | system architecture | systems design

ABSTRACT

Modeling solution-neutral operating principles of system features as SysML behavior models enable systems engineers to specify reusable product features for a variety of similar products and to specify and document a product's functionality in an easy-to-understand manner. In practice, multi-perspective modeling methods from systems engineering require a high manual effort from experts who do not have the needed time. Current systems engineering methodologies provide multi-perspective modeling approaches. However, they do not consider real-world sizes of the problem domain. Thus, if the systems engineers change their perspective, it requires excessive manual effort, and they must perform redundant tasks. This paper presents a CUBE-based, feature-driven methodology that provides generative methods to systems engineers to address this challenge. In detail, we are using SysML activity diagrams to model operating principles of system features and generate the logical architecture as SysML Internal Block Diagrams.

1 | Introduction

Motivation. Modeling solution-neutral features at a functional level [1] enables the reuse of basic and comfort features, shortens time-to-market, and sustainably documents a product's functionality. This is particularly relevant in the automotive domain, where many manufacturers, such as Toyota Motor, Volkswagen Group, and General Motors, operate as strategic groups sharing common modules for vehicle development. Consequently, it is reasonable for these groups to reuse not only final products but also product specifications [2], enhancing product quality and development speed while reducing development time.

Model-based systems engineering (MBSE) [3, 4] provides a means to describe automotive specifications from different perspectives [5]. Systems Modeling Language (SysML) Activity Diagrams are one option to specify operating principles [6], offering a solution-neutral view of the system by specifying its functional behavior without considering technical implementation alternatives or structural/architectural choices [7]. Functional elements, as part of the operating principle, are logically grouped by structural elements and are thus subject to a structural realization decision in the form of the logical product architecture [7]. Although no technical decisions are made at this stage, it becomes the focus when designing the system's technical architecture in the subsequent specification phase.

In the automotive domain, real-world models encompass several hundred features and tens of thousands of signals, making manual creation and maintenance of these models for various perspectives time-consuming.

Research Gap. Existing approaches often overlook the effort required to create models from various perspectives, leading to unrealistic time demands or impractical automation. Thus, some companies fear that MBSE may negatively impact budgets,

schedules, efficiency, safety, and morale [8]. Moreover, high-stakes engineering projects are particularly challenging for experimentation but can potentially benefit the most from the application of MBSE [9]. Various methodologies suggest modeling systems from different perspectives [10], including MBSE [11–14] and feature-driven systems engineering [15] and offer advanced specification methods [16–18] but still require significant time for model creation. This issue is exacerbated in industry projects where experts have limited time [8].

Model-driven approaches [19], on the other hand, focus on specific tasks such as transformations for verification purposes [20, 21], generating design models [19], or generating software code [22–24]. However, they lack integration into a holistic approach or methodology. Based on these observations, we identified the following gaps that our research aims to address:

- **High Effort for Model Creation**: Many MBSE approaches aim to create an "all-inclusive" model, neglecting the effort required to develop them.
- **Redundant Tasks for Multi-View Modeling**: Although multiple views are beneficial for readers, modelers face the challenge of performing redundant tasks during their creation.
- **Applicability in Industry Projects**: Many approaches focus on general applicability without addressing the specific needs of concrete projects and industries.

Research Question. To address these challenges, our aim is to answer the following research questions for the concrete application scenario of generating logical architectures from SysML behavior models:

- **RQ1**: How can the efficiency of MBSE be increased?
- **RQ2**: Which models suit project-independent reuse?
- **RQ3**: Are models from previous development steps or views reusable in later development steps?

To mitigate the high efforts for model creation, we aim at identifying optimization potentials in the considered projects from the automotive industry in **RQ1**. To apply them in the context of multiple industry projects, we analyze potentials for a project-independent reuse in **RQ2**. Finally, we ask **RQ3** to combine the efforts from **RQ1** and **RQ2** to not only increase the efficiency and applicability in multiple projects, but to also consider the additional efforts for multi-view modeling in multiple approaches.

Contribution. To increase modeling efficiency, We propose a model-to-model transformation approach to generate logical architectures from operating principles, based on the Compositional Unified System-based Engineering (CUBE) [7] methodology (see Section 2.3). This feature-driven methodology:

1. Helps systems engineers formulate operating principles with object flow and logical elements.
2. Automatically generates logical architectures with minimal effort.

We reuse functional descriptions from SysML Activity Diagram (act) to generate SysML Internal Block Diagram (IBD) based on a Logical Reference Architecture (LRA). We demonstrate this approach with an automotive example and discuss its practical applicability. Therefore, this paper is structured as follows: Section 2 presents relevant foundations and related work. Section 3 introduces an example from the automotive domain and shows its feature specification. In Section 4, we introduce our method for generating logical architectures in the form of a SysML IBD from an operating principle modeled as a SysML Activity Diagram (AD). Section 5 discusses the method and results, and the final section concludes.

2 | Foundations and Related Works

In recent years, systems engineering has become increasingly important in industry and especially in the automotive industry [25–27]. In this process, systems engineers consider systems holistically during development and take other life-cycle phases into account at all times [11]. In particular, this should achieve an improved system specification that adequately addresses customer requirements, ensures high-quality product, and reduces the need for rework. Furthermore, the holistic system approach should avoid incompatibilities of system components and thus reduce the effort required for system integration [28].

2.1 | Systems Engineering

The systems engineering approaches developed in recent years often combine established principles, such as structured partitioning, system decomposition, or system description using different points of view [15]. Approaches such as OOSEM [11], SPES [12] and its extension SPES XT [13] or SMArDT [6, 29] also use the model-based development method, which extends the purely textual and document-centric specification of the system with a graphical specification, enabling improved interdisciplinary collaboration. The additional built database can enable continuous enhancement, improved reusability, or even automated evaluations of development artifacts during development.

2.2 | Feature-Driven Development

As a second trend in the automotive industry, agile development approaches have become more observable in recent years. In particular, feature-driven development originated in software development and has become especially relevant. Thus, it not only enables the definition of agile development processes and feature-oriented system design [7], but also improves complexity management, better variant management, and feature-specific reuse of development artifacts [7].

2.3 | Compositional Unified System-based Engineering

To apply the feature-driven development paradigm in systems engineering, the CUBE methodology, which has its roots in the SPES [12] and SMArDT [6, 29] methodologies, combines the established systems engineering fundamentals with the feature-driven development paradigm [7]. CUBE achieves this by extending the general concepts of decomposition and abstraction that SMArDT [30] and SPES [12]/SPES XT [13] utilize and embedding a feature-driven development paradigm into these methodologies. This means that CUBE uses similar abstraction layers as SMArDT to define the use cases for the application of the system, operating principles to specify the system behavior executing these use cases, and logical and technical architecture views to describe the implementation of a system. Moreover, as suggested in SPES, CUBE uses techniques for decomposing the system into sub-systems. To enable the systems engineers to apply the feature-driven development paradigm during the system design, CUBE begins with a use case elicitation phase, in which the systems engineers define the application scenarios for the system under development. To implement the use cases identified by the system stakeholders, the next CUBE abstraction layer requires the systems engineer to cluster the use cases into features, for which an operating principle is developed that describes the implementation of the use case in a solution-neutral way. With this operating principle, the systems engineers need to assign the steps from the operating principle to executing elements from the system. By this, not only the decomposition but also the required interfaces are encompassed. As this is one of the main drivers of the system design, this paper aims to support the systems engineers in this task.

2.4 | Feature Modeling Using Activity Diagrams

Specifically, so-called function nets can serve to model automotive features [31]. As an implementation of this concept, our approach specifies function nets using SysML internal block diagrams to represent a logical architecture of the entire system. Views on these function nets limit the model scope to specific features or feature variants [32]. Scenario diagrams model these function nets, similar to UML communication diagrams. However, the authors acknowledge that the focus of function nets lies on structural system properties and that functional behavior is only exemplary, illustrated in the defined scenario diagrams [33]. For proper full-fledged feature specification, the approach has to be enhanced by dynamic behavioral models.

2.5 | Related Works

Apart from the methodical aspects presented in this paper, other approaches exist for similar model transformation. For instance, several transformations for ADs exist; however, approaches transforming SysML diagram automatically into other diagrams are usually only a corner case. More common are approaches that translate SysML diagrams to code or use transformation for system validation. For system validation, SysML ADs are transformable to Petri nets [34]. Based on this transformation, related projects transformed SysML ADs to Petri nets for automatic system verification [35, 36]. As an extension of this concept, Petri nets are transformable to VHDL[1]-AMS[2] [37] using the Atlas Transformation Language (ATL), which is a hybrid model transformation language defining transformation rules and helpers [38]. As for ADs, also Internal Block Diagrams (IBDs) are often transformed for validation e.g., for formal validation by mapping the SysML IBDs to the input language PRISM [20].

The authors are unaware of other methods for transforming from ADs to IBDs—however, a semi-automatic approach for transforming in the opposite direction exists [39].

The translation presented in ref. [39] begins with the transformation from the main block of the SysML IBD to an activity of the Unified Modeling Language (UML) AD. Afterward, the element transformation is in focus. First, the approach transforms each part of the IBD and each reference into an action of the AD. Then input and output ports are transformed into parameters and set as initial and final states. After that, each port, linked to a part or a reference, is converted to an input or output pin and linked to a corresponding action in the AD. Each connector that connects parts or references transforms into an object flow connecting the actions in the AD. Finally, if the main block has ports, these are transformed into activity parameter nodes following the connections between the ports of the main block and parts or references [39]. One of the limitations of the reverse mapping [39] is the activity flow, which is not automatically derivable because the IBD does not contain this information. It has to be set manually by the system engineers [39]. In our approach, this is not an issue due to the consideration of the reverse direction where the engineer creating the AD can directly specify this information. For the transformation with ATL, the source and target models and the transformation definition must correspond to their respective metamodels, which limits the structure of the AD. However, ATL does not provide a direct graphical representation of the diagrams, which is a clear advantage of our concept.

Another related approach is the functional architecture for systems method [40], for which another generation process for functional architectures exists [41]. To this end, activity diagrams serve as the basis for functional architecture generation by transforming these activity diagrams into a SysML IBD. In contrast to our approach, this functional architecture functions as an additional architecture description, in which an assignment to executing elements is limited. Moreover, only a one-to-one mapping of the use cases to activity diagrams is allowed, and the authors only sketch the automated transformation without formalizing the applied transformation rules.

Regarding the specification of logical architectures in literature, several systematic approaches starting from use cases are outlined [40, 42]. The focus here is on the creative process as part of the transformation from a functional analysis model to a logical architecture. Usually, this is achievable by allocation function partitions to logical architecture elements, and as properly-outlined, this process step cannot be reasonably automated [40]. Instead, these papers propose the use of heuristics in order to enhance this allocation step systematically. However, the automated generation of a logical architecture representation after allocation must be considered and introduced as part of this contribution.

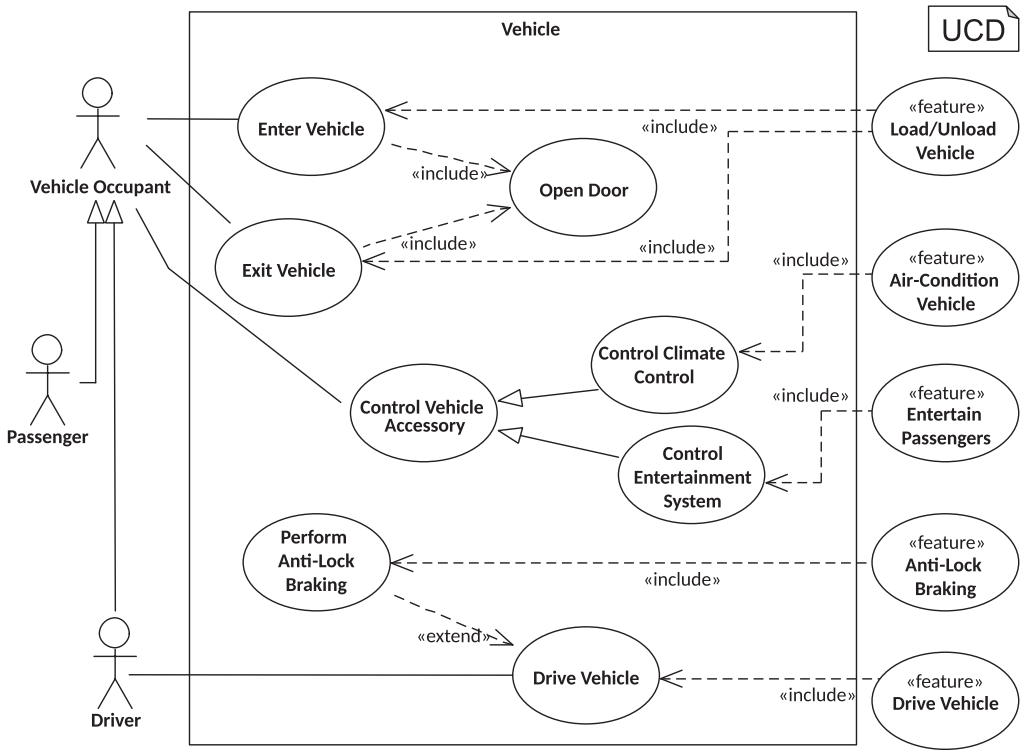

FIGURE 1 | Use cases and feature derivation of automotive use cases presented in the literature [43].

3 | Example From the Automotive Domain

As we present the method and the lessons learned in the context of an automotive development projects, we chose a running example from the automotive domain, as this domain can greatly benefit from MBSE [9]. To protect the project details and intellectual property of the stakeholders involved in the projects evaluated in Section 5, this paper uses a use case diagram from an automotive example presented in the literature [43]. Unlike the methodology in the literature [43], this running example develops an alternative functional specification for the basic functionalities of a vehicle. Although the use cases in Figure 1 remain largely unchanged from the literature [43], the rest of the functional specification and this section intentionally deviate to follow the feature-oriented functional specification approach according to CUBE. To specify the execution semantics of the use case diagram, we use the syntax and semantics from another source [44]. A use case diagram consists of use cases. A system executes a context as indicated by the system boundary (a rectangle in the diagram). The actor is connected to this use case using the communication relationship to indicate that an actor can execute a use case.

The use case diagram in Figure 1 specifies various vehicle use cases such as "Enter Vehicle", "Exit Vehicle", "Control Vehicle Accessory", and "Drive Vehicle.". According to CUBE, multiple use cases may form a group that features implement following a feature-oriented development method [45], similar to a use case study in the context of autonomous vertical take-off and landing use cases [46]. In this paper, features are independent end-to-end functionalities of the system that benefit at least one stakeholder. A use case diagram represents a stakeholder as an actor and a feature as a non-empty set of satisfied use cases.

In this example (*cf.* Figure 1), five features address all use cases from the use case diagram:

1. "Load/Unload Vehicle" combines use cases to enter and exit the vehicle (including 'Open Door' due to include relations).
2. "Air-Condition Vehicle" addresses the "Control Climate Control" aspect of the 'Control Accessory' use case.
3. "Entertain Passengers" addresses the "Control Entertainment System" of the same use case.
4. "Drive Vehicle" combines all actions required to drive the vehicle to its desired destination.
5. "Anti-Lock Braking" addresses "Perform Anti-Lock Braking" as an extension of the "Drive Vehicle" aspect.

In the next step of CUBE, a systems engineer must specify the operating principle for each feature, considering all related use cases. Since "Drive Vehicle" is a main feature of almost all vehicles, we will use it as the primary feature in this paper. To specify the operating principle, the systems engineer must define the actions the system must perform solution-neutrally. For this purpose, a AD, as presented in Figure 2, is the primary specification model.

As specified in the diagram, to implement the "Drive Vehicle" feature, the driver (human or machine) must be informed about the environment and the vehicle's driving state initially. Since the driver, as an external entity, is not part of the "Drive Vehicle" feature, the model does not represent the steps required to make a driving input decision. Hence, only the resulting steering inputs in the form of "Steering Command", "Acceleration Command", "Brake Command", and "Gear Selection" are considered in the

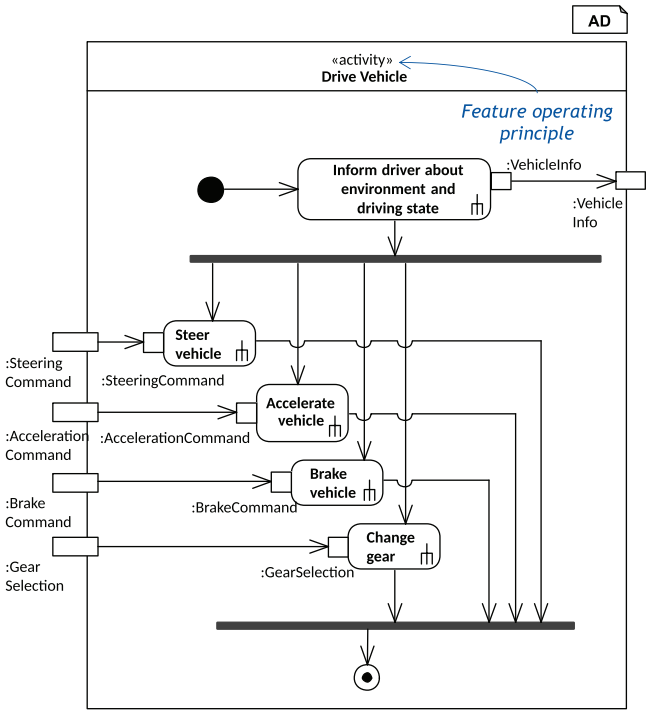

FIGURE 2 | A feature specific operation principle for the "Drive Vehicle" feature.

operating principle of this feature. To effectively operate a vehicle using these inputs, the vehicle must respond by adjusting its steering, acceleration, braking, and gear shifting. Since this specification is only at a functional level, the physical cross-relationships between steering, accelerating, and braking are deliberately not modeled at this step, as they depend on the vehicle's technical realization. For example, a motorcycle would react differently to simultaneous steering and braking than a car or truck [47].

4 | Generating Logical Architectures from SysML Behavior Models

To implement use cases identified by stakeholders, the CUBE methodology develops functional behavior in a solution-neutral way. We focus on the "Drive Vehicle" feature to explore our approach for generating logical architectures from SysML Activity Diagrams. Our approach builds on CUBE's modeling of feature operating principles. To be applicable in the industry, the approach must fulfill several requirements and fulfill several process requirements as, e.g., specified in ref. [48]:

- **MR1**: Specification models must be reusable to realize reduced time-to-market constraints
- **MR2**: Specifications must be adaptable to specific markets or vehicle projects
- **MR3**: Specification models must be consistent
- **MR4**: Specification models must be bidirectionally traceable (e.g., from a requirement to a component and vice versa)
- **MR5**: Creating specification models must be possible without redundant manual modeling tasks

To describe how our method implements these requirements, first, Section 4.1 addresses the reuse of our requirements stated in MR1 and motivates how to model ADs as a specification of reusable feature operating principles. Then, Section 4.2 introduces the concept of a LRA, which enables system engineers to implement and adapt the functionality in the context of a predefined LRA to implement the challenges that arise from MR2. Because the actions from the operating principles require an executing element from the Logical Reference Architecture (LRA), Section 4.3 deals with the allocations between functional and logical execution views. Finally, Section 4.4 addresses MR3 and MR4 by providing transformation rules from ADs to IBDs for a give allocation to the LRA, which enables the systems engineer to generate a logical architecture without the need for a redundant modeling task as required by MR5.

4.1 | Modeling Operating Principles Based on CUBE

In CUBE, eliciting relevant stakeholders and use cases is part of the customer value specification [7]. Systems engineers identifies use cases and functional features to specify the "Operating Principle." In the running example, the vehicle has multiple use cases for the driver, such as "Enter Vehicle", "Exit Vehicle", and "Drive Vehicle." Since these features only provide value to an individual customer and do not necessarily specify an entire sequence of actions required to operate the vehicle, features group use cases into client-valued functionalities that benefit the stakeholder in achieving their aims with the system. The AD in Figure 2 specifies the steps to drive a vehicle, ensuring solution-neutral principles that are reusable for other systems like bicycles or trucks, supporting feature reuse (MR1) and adaptability (MR2). In our running example, the driver is informed about the vehicle's environment and driving state in the first action. After that, they can steer, accelerate, brake, and change gears in parallel. Apart from the control flow specified by these actions, the AD in Figure 2 also specifies the object flow, i.e., the objects or information required to perform these actions.

In the next step of CUBE (and similar methodologies), a systems engineer must identify the logical elements that perform these actions and allocate the actions to these elements. Based on this evaluation, they would update and evaluate the resulting logical architecture to optimize their decision, e.g., by using a reasoning framework [49]. In all concurrent industry projects known to the authors, these steps are performed manually, as none of the commonly used SysML modeling tools (e.g., IBM Engineering Systems Design Rhapsody [50], Enterprise Architect [51], MagicDraw/Cameo Systems Modeler [52]) support this generation natively without further tailoring.

As a result, systems engineers often view the creation of logical architectures (or operating principles) as desirable but unnecessarily difficult, leading them to avoid this effort, which negatively impacts project documentation. This observation is also reflected in the study by ref. [18], where a common need for developing description methods that logically reflect the functional and

architectural structures and rules of the system is identified. However, investments in systems engineering tools are not made, and the benefits of traceability analysis are not estimated as highly as other benefits of MBSE in the questionnaire.

To address this, we propose an automated transformation that reduces effort by generating logical architectures from ADs. Systems engineers can allocate actions to elements in the LRA, using the object flow to generate logical architecture diagrams based on transformation rules presented in Section 4.4.

4.2 | Logical Reference Architecture for Element Allocation

Most established manufacturers and suppliers of complex or safety-relevant systems, such as rail vehicles, aircraft, and autonomous vehicles, leverage their extensive experience for future development projects. These systems are often not developed from scratch; instead, results from previous projects serve as inputs for subsequent generations. Therefore, it is crucial that the reference architectures used are accurate. We assume that either this architecture is known beforehand or quickly converges to an improved version when applying our approach, or that a family of LRAs developed for each family of problems may is already established. In domains where the architecture is not initially known, there is still an opportunity to develop a preliminary sketch during development and refine it by reallocating the executing elements multiple times. Since the development of system reference architecture is highly domain-specific, this paper does not delve into its development and refers detailed approaches in the literature [33, 53–55].

Using a LRA as presented in this paper embraces the idea of reuse and adaptability as required by MR1 and MR2 by providing systems engineers with elements to implement functions modeled according to an operating principle. Because this logical architecture is independent of the technical realization, the same logical architecture can be used as a reference for multiple technical architectures, allowing the required adaptability by MR2. Moreover, since the resulting logical architecture can be generated automatically by the transformation rules presented in Section 4.4, re-adoption to different LRAs is also possible, increasing the reusability of the operating principle specification as required in MR2.

Like other system architectures [56–60], the LRA has a structural and a behavioral part. The structural part is based on the logical decomposition of the system and specifies subsystems for the next level of decomposition. The behavioral part of a system is modeled here by a state machine that describes the states the system can take in its life cycle. To put this idea into practice, the structural part of the logical system reference architecture is modeled in a SysML BDD, and the behavioral part of the logical system reference architecture is modeled in a SysML state machine diagram.

For the sake of our running example, we refer to the elements presented in the Block Definition Diagram (BDD) from Figure 3 as LRA. In this architecture, the "Vehicle" is decomposed into the "Powertrain," providing the drive energy and transmitting it

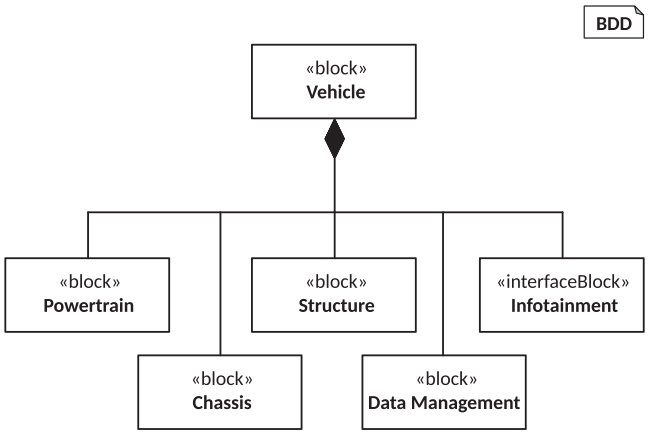

FIGURE 3 | A (simplified) LRA as an example for automotive system development.

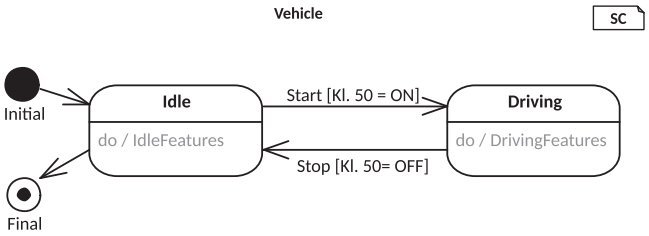

FIGURE 4 | A simplified state machine for a vehicle that is either driving or idle waiting for its next drive

to the road to guide the vehicle longitudinally; the "Chassis,", responsible for steering, lateral control, and roll stabilization; the structure, consisting of structural elements for crash security as well as exterior and interior components; the "Data Management", responsible for collecting and providing information from or to the environment; and the "Infotainment" system, responsible for all vehicle-machine interactions and passenger entertainment.

To better locate individual system functions or system features in the context of the overall system and to model their interaction in addition to their pure assignment, the features should be located even further in the behavioral part of the LRA. For this purpose, we recommend refining the SysML state's behavior further and modeling the behavior of the modeled features as an activity diagram for the entry/do/action operations of the state machine.

For the "Drive Vehicle" feature developed as a running example in this paper, we only consider two states in the state machine presented in Figure 4. According to this state machine, our vehicle is either "Idle" or "Driving" and changes between those states by a "Start/Stop" trigger. These are intentionally not further specified to allow different implementations in the product's technical architecture. An electric vehicle could use a drive mode selection, whereas a vehicle with an internal combustion engine could implement this state change by referring to the ignition. Since these states from the LRA only restrict the number of possible states but do not relate to the developed features, the AD from Figure 5 exemplarily locates the "Drive Vehicle" feature in the context of the driving state of the vehicle. To visualize

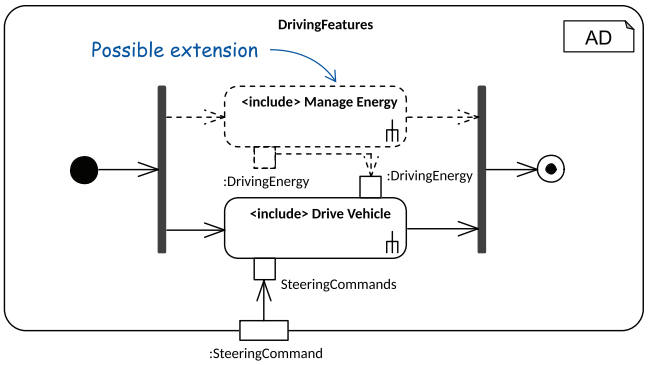

FIGURE 5 | The activity diagram showing the features that are active/executed during the driving state.

feature inter-dependencies, the diagram additionally presents the potential support feature "Manage Energy" that provides the required driving energy to the "Drive Vehicle" feature, as indicated by the object flow in the diagram.

4.3 | Allocating Elements to Their Executing Architectural Parts

Identifying the executing entity for each action in the operating principle is a core task of functional/structural decomposition, which determines the work-split and subsystems of the system development process.

Consequently, all actions must be mapped unambiguously to their executing entities in the LRA to automate the generation of the logical architecture. In our example, most elements of the feature operating principle (cf. Figure 2) can be mapped straightforwardly to their executing entities in the LRA (cf. Section 4.2). For instance, "Steer vehicle" is performed by the "Chassis", while "Accelerate vehicle", "Brake vehicle", and "Change gear" are mapped to the "Powertrain.".Mapping the vehicle's braking action can be a philosophical question, as the chassis can also perform this task if decomposed structurally rather than functionally.

However, the "Inform driver about environment and driving state" action cannot be mapped unambiguously to the Infotainment system alone, as the structure also needs to enable the driver to see the vehicle environment. Thus, in Figure 6, the action is decomposed into its elementary steps, enabling the driver to see the environment and receive vehicle information.

4.4 | Transformation Rules

The transformation of operating principles expressed in a AD into a logical architecture expressed in a IBD, as proposed in this paper, follows formal transformation rules and implies some preconditions described in this section.

To make a formal transformation possible, all activity diagrams for the operating principle must ensure that all elements are mapped to an executing element from the LRA. Moreover, this allocation must be unique, such that an unambiguous mapping from action to its executing element in the LRA is possible.

The transition is then performed by applying the following rules. Note that all other elements from the AD are still allowed to be part of the diagrams for the generation, but are not regarded in the transformation.

The first rule, also called the execution rule (Figure 7), states that an action allocated to an 'Executing Element' from the LRA is executed by this element. Thus, a system composed of the executing element must have this element to perform this action. Consequently, the logical architecture diagram of the system must contain this executing element to perform this action.

Since the necessity of an element to perform an action is not sufficient for a functional specification, the second rule, also called the action interface rule (Figure 8), states that the object flow an action requires to be executed must also be regarded in the resulting IBD. For this, the rule states that if two actions are allocated to different elements 'A' and 'B' from the LRA and have an object flow between two action pins 'a1' and 'a2' with the same or a compatible type 'T,' the information flow must be part of the logical architecture. Hence, the rule implies that the IBD created during the transformation must contain matching ports 'a1' and 'a2' with compatible type 'T.' Since the SysML specification [61] allows some ambiguities here, we require that the ports are typed with an interface block 'T' having 'a1' and 'a2' as flow properties and the same direction (ingoing/outgoing) as indicated by the object flow.

Because the feature operating principles presented in this paper are not stand-alone but connected in the behavioral part of the LRA, these interfaces must be especially regarded during the transformation. To this aim, Figure 9 states a rule to resolve these inter-feature dependencies during the logical architecture generation. According to this rule, if an 'Action1' has an action pin connected to an outgoing parameter of its 'Operating Principle 1' Activity and an 'Action2' has an action pin connected to an ingoing parameter of its 'Operating Principle 2' Activity, and both operating principle activities are connected in invoked actions allocated to different elements 'A' and 'B' from the LRA, then 'A' and 'B' must be connected in the generated logical architecture and present in the feature operating principle diagram.

Finally, all actions connected to activity parameter nodes that are not connected to elements from different elements must be provided from the environment of the system and thus must result in interfaces from the system environment as stated in the fourth rule, also called the input/output rule (Figure 10).

The following implications arise from the presented transformation rules:

- From the first rule (*cf.* Figure 7), actions are only allowed to be allocated to subsystems of the currently regarded system.
- From the second rule (*cf.* Figure 8), object flows between actions allocated to the same element of the LRA are not visible/regarded in the logical architecture (as no interface between the sub-elements will be generated).

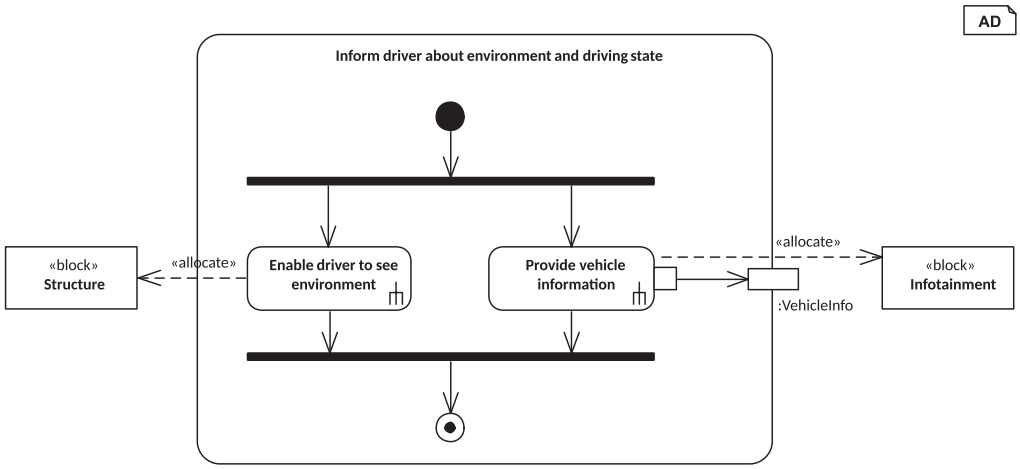

FIGURE 6 | The activity diagram showing the decomposition of the "Inform driver about environment and driving state" action.

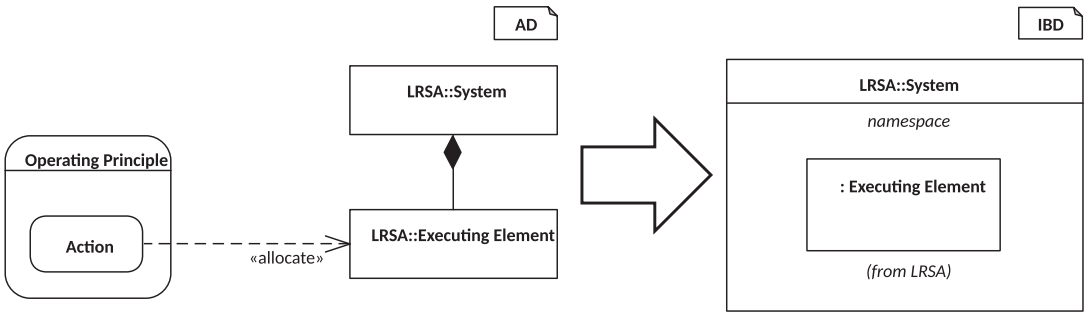

FIGURE 7 | **Rule 1 (Execution Rule)**: Transformation rule for action execution in the logical architecture

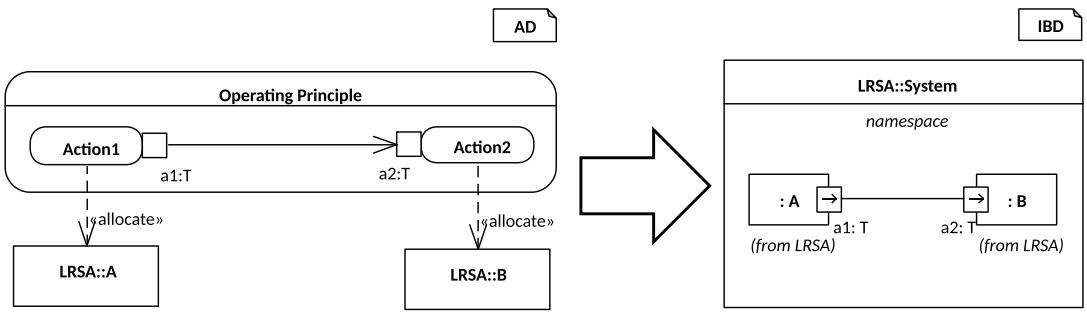

FIGURE 8 | **Rule 2 (Action Interface Rule)**: Action interface transformation rule creating interfaces for actions connected with an object flow

- The third rule (*cf.* Figure 9) does not filter out illegal connections (e.g., a port becomes multiple inputs from different systems). Thus, the resulting logical architecture might consist of illegal port connections that must be later found and resolved in a validation phase of the logical architecture.
- Following the fourth rule (*cf.* Figure 10), all unconnected activity parameters are interpreted as system interfaces. Thus, the resulting logical architecture might contain external interfaces that were not intended by the modelers of the operating principle.
- Since an automated application of these rules is possible, the requirements from MR3-5 can be verified by design.

4.5 | Example Logical Architecture Generation

Applying the aforementioned rules to the diagrams provided in Figure 2 and Figure 6 together with the allocation provided in Table 1 leads to the diagram presented in Figure 11.

As an example application Figure 11 connects the operating principles activity diagrams from Figure 2 and 5, and visualize the application of the transformation ruels from Section 4.4 with blue arrows. Following the first rule, all LRA elements shown in the 'Executing Element' column of Table 1 are included in the IBD from Figure 11. The fourth rule implies that all elements must be interfaces and are routed to the system boundary of the vehicle. Following from the feature connection in Figure 5 and the

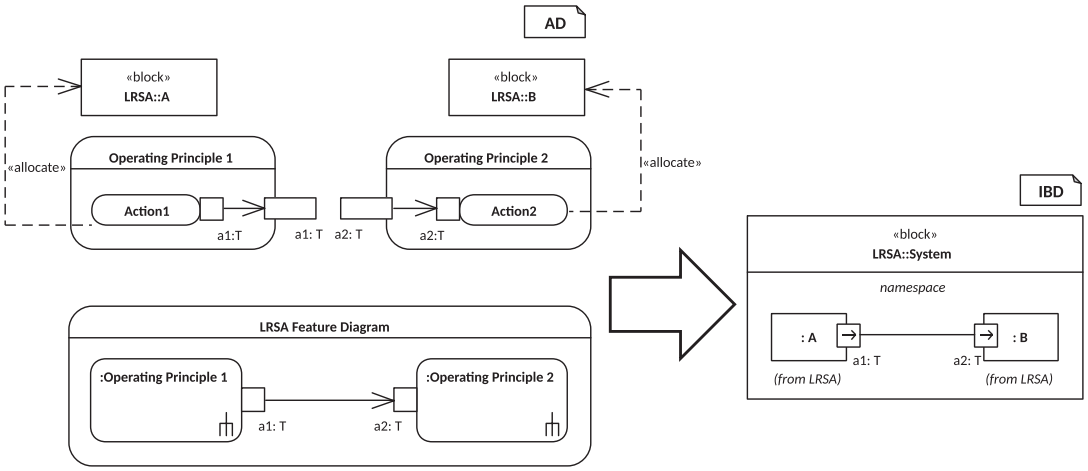

FIGURE 9 | **Rule 3 (Feature Interface Rule)**: Feature interface transformation rule to connect features using parameters provided in other features in the logical architecture

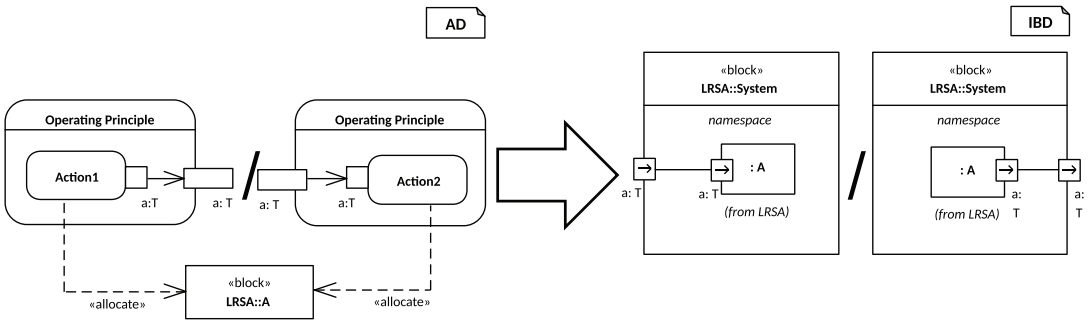

FIGURE 10 | **Rule 4 (Input/Output Rule)**: Input/Output rule to create system interfaces for objects that are not provided by other features

TABLE 1 | Allocation of the actions from Figure 2 and 6 to executing elements in the LRA shown in Figure 3

Action	Executing Element
Enable driver to see environment	Structure
Provide vehicle information	Infotainment
Steer vehicle	Chassis
Accelerate vehicle	Powertrain
Brake vehicle	Powertrain
Change gear	Powertrain

third transformation rule, the structure system provides driving energy to the powertrain. Since no feature internal connections apart from this connection are made in the diagram, and no other features are considered in our running examples, the second transformation rule is not applied in this example. As this rule would lead to the same result as allocating the drive vehicle action to the powertrain, we did not include an additional example here and leave this application as an example task for the reader.

In the example shown in Figure 11, two familiar situations from examples of this kind are presented. First, one might note that the 'Structure' component is unconnected to the other features. This is because the ability of the driver to look at the vehicle's exterior is deliberately not further specified as a logical architecture interface here. From similar applications to other automotive examples, we noticed that similar situations also appear in the logical architecture that is realized implicitly in the product architecture, such as screw points, contact areas, or energy provision.

As an attentive reader might also have noticed, most of the performed actions require different kinds of energy. For example, most concurrent infotainment systems require (low-volt) electrical energy, powertrains are operated with fuel or (high-volt) electrical energy, and the chassis usually transmits rotational energy from the powertrain to the wheels. Since we aimed at modeling a solution-neutral drive operation, we deliberately decided not to model these aspects in the drive feature and already indicated in Figure 5 that these tasks are modeled in a 'Manage Energy' feature that is required to be executed in parallel to the 'Drive Vehicle' feature. Finally, there is the allocation of the 'Brake vehicle' action. Depending on the decomposition, this is either part of the powertrain (following the functional decomposition philosophy, the powertrain is responsible for all tasks of longitudinal guidance and thus also for longitudinal deceleration) or the chassis (following the mechanical decomposition, the brakes are mounted to the chassis and decelerate the wheels, thus part of this system). An advantage of our approach is that these different philosophies reduce this conflict since the allocation and logical architecture generation can easily be changed.

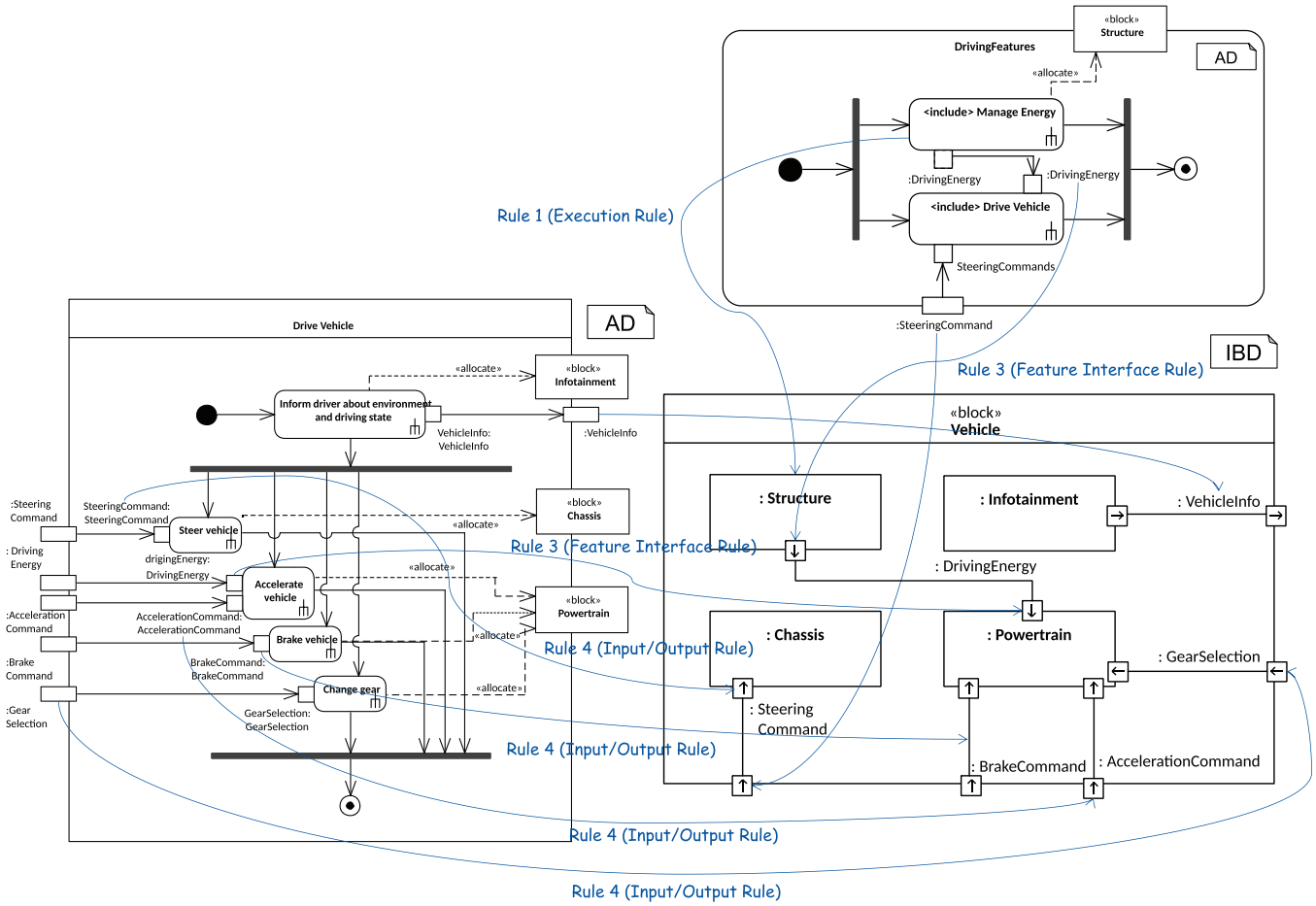

FIGURE 11 | Feature-specific logical architecture diagram for the "Drive Vehicle" feature.

5 | Lessons Learned from Application in Practice

We applied our approach in three automotive systems engineering projects. This section summarizes and compares the results, describes our application, and derives lessons learned. We analyze how our approach increased efficiency and reflect on the fulfillment of methodical requirements, considering limitations. Finally, we present future improvements and possible extensions.

5.1 | Lessons Learned from Industrial Application

To evaluate our approach and quantify the advantages, we applied it in three automotive development projects. In the first project, Project A (Project A), the project team created a high-level vehicle architecture for software and hardware development. A comparably small team of 7 systems engineers developed a model-based specification for 16 critical high-level vehicle features. Due to the small team size and a tight schedule, it was agreed with the project management to use the approach presented in this paper to accelerate development. They used Enterprise Architect as a modeling tool and implemented the transformation rules from Section 4.4. The team reported their times to analyze the performance and determine potential time savings.

In the second project, Project B (Project B), the systems engineers used the same tool setup but developed a demonstration example for internal training and showcases of autonomous driving functions. Due to this different focus and setup, the involved systems engineers did not track their exact modeling times and did not aim to develop an industrial product. Thus, the project results might not reflect the best architectural decisions but showcased the possibilities of the underlying CUBE framework and tooling.

Finally, in the third project, Project C (Project C), the project team developed a functional specification of 122 vehicle functions and used the approach presented in this paper to derive a logical architecture specification. As the approach was already established, no additional time was planned for the derivation of the logical architecture, and more time was scheduled for the development of the operating principle. Therefore, no exact speed-up of the process using the generation approach is possible.

Table 2 provides more details about the number of actions, the required parameters/object flows between those actions, and their effect on the logical architecture in the three considered projects. As all three projects were concerned with the top-most decomposition layer, it is important to note that the figures only represent the top-level interfaces on a small excerpt of the overall vehicle and system models. Thus, the number of interfaces almost grows exponentially for the following decomposition levels as soon as these interfaces are refined in the next development steps. From these numbers, we see that a small number of high-level features already result in a high number of interfaces

TABLE 2 | Number of elements generated in three considered projects.

Project	Project A	Project B	Project C
Operating Principle			
Features	16	7	122
Actions	380	170	2456
Parameters	749	477	2297
Logical Architecture			
Logical components	6	5	14
Ports	549	57	1250

on the subsystem level in the logical architecture. Moreover, comparing the three project results reveals that the modelers used 20.13 actions per feature in Project C, 23.75 actions per feature in Project A, and 24.29 actions per feature in Project B. Thus, approximately 20–24 actions per feature is the average number of actions a feature performs in the considered projects. Furthermore, each of the actions has between 0.94 in Project C, 1.91 in Project A, and 2.81 in Project B parameters per action. Therefore, each action processes one to three objects during this operation.

As the example in Figure 11 showed, the allocation of two adjacent actions to the same system of the LRA results in the object flow from the activity diagram not appearing as an interface in the logical architecture. As the results in all three projects show, this is not only a theoretical possibility due to the design of the transformation rules but occurred often in the three considered projects. In fact, only 0.73% of the defined object flows on the operating principle level of Project A resulted in interfaces on the logical architecture level, whereas in Project C only 0.54% and in Project B with 0.11% even fewer activity parameters result in ports on the logical architecture level. Although the Project B project aimed to minimize system interfaces, leading to an average of 11.4 ports per component with their action allocation in an idealized project environment, Project A and Project C gained comparable numbers with 91.5 respectively 89.29 ports per component.

For that reason, interface handling is one of the following lessons we have learned during the application:

- **L1**: The execution times of the transformation are sufficient. The implemented automation did not cause any considerable waiting times for the modelers. However, the size of the overall model does not scale well with a large number of features in the prototype implementation of the modeling tool.
- **L2**: Collaborative modeling and variant handling are still an issue for the native implementation of the modeling tool used in the project.
- **L3**: Additionally, created links between the parameters and ports achieve easier navigation and support traceability.
- **L4**: The allocations from actions to executing elements, as well as a naming convention, enable further analysis and specification of a component.

Although the implementation of the approach worked well in the modeling tool, the underlying database quickly reached its limitations when multiple features were considered and multiple modelers modeled different features in parallel, as expressed in L1. With this observation, we also reproduced the assumption made in ref. [33], which may indicate a general lesson for systems engineering projects.

5.2 | Increasing Efficiency of the Project Team

Since the method we present aims to reduce the effort to create and model logical architectures through automatic generation, it is reasonable to assume that our approach increases the efficiency of the development team. We analyzed this in the context of the Project A project. Project management tracked the team's performance during the creation of the logical architecture by examining the hours spent in each project phase. The goal was to compare the team's performance with benchmarks from previous projects and account for the increased efficiency due to automation in future effort estimations.

The project team consisted of 3 MBSE experts experienced with the CUBE methodology and 4 domain experts with extensive experience in function development. The methodology experts reported that automation improved their workflow compared to previous projects. Project management noted that the remaining 4 team members, who had less experience with the methodology, integrated faster and performed better after a shorter familiarization phase.

A rough schedule for logical architecture creation was required to coordinate the project effort. Previous projects served as a basis for initial estimation, and the expected efficiency increase (50%) was factored into the project plan. After the project phase, we verified whether the planning was accurate by comparing the hours booked by project members with the initial plan. Modelers booked only 46.87% of the hours compared to other projects without automation, exceeding our initial expectations under real-life conditions.

One threat to validity is that the adaptation beforehand (less effort planned and communicated) might have influenced the modelers' performance. The tighter schedule increased pressure, potentially improving efficiency at the cost of quality or productivity, as implied by ref. [62] in construction performance. However, based on previous projects, we assume a similar increase is achievable with automation. Customer and project management concerns required us to consider this risk.

The project team reported that the schedule was tough but realistic, suggesting minimal impact on our observations. Feedback indicated that the bottleneck shifted from modeling systems engineers to functional domain experts, whose availability and response time limited the schedule improvement.

Automation significantly reduced the effort required to create the logical architecture, covering all modeling tasks with transformation rules. However, previous projects showed that effort is also needed for diagram layout. The used tooling (Enterprise Architect) does not fully address this issue, impacting diagram

understanding [63]. Modelers had to manually adapt suggested layouts. Errors in operating principles and interactions were identified during logical architecture reviews, requiring rework and regeneration of the logical architecture, which was factored into the effort estimation.

5.3 | Reflection on the Fulfillment of the Methodical Requirements

In defining the approach from Section 4, we formulated several methodical requirements and reflect on their fulfillment.

MR1: Reusability The split between a solution-neutral operating principle and the use of an LRA enables reuse across different development projects. Specifications created using this method are reusable, reducing time-to-market constraints. Operating principles can be adapted to different environments by changing the allocation to the LRA, such as for electric vehicles, bicycles, or other driving vehicles. The logical architecture is reusable without implementing the operating principle, as interfaces are derived from actions. However, evaluating the time-to-market constraint is not possible without a complete industrial application.

MR2: Adaptability Deriving a logical architecture based on an operating principle ensures adaptability to specific markets or projects. Solution-neutral operating principle models enable reuse in different logical architectures. Different allocations from action to LRA elements allow variability on a logical level, and different realizations on a product level. More ways to express variability might be required, such as modeling variability directly on control flow or object flow level [64, 65]. Future work could analyze how this variability influences logical architectures.

MR2: Adaptability Deriving a logical architecture based on an operating principle ensures adaptability to specific markets or projects. Solution-neutral operating principle models enable reuse in different logical architectures. Different allocations from action to LRA elements allow variability on a logical level, and different realizations on a product level. More ways to express variability might be required, such as modeling variability directly on control flow or object flow level [64, 65]. For example, a positioning feature might only be required for autonomous aircraft with high-precision requirements, but not for standard road vehicles [45]. Instead of modeling two features for positioning, a systems engineer might want to model only one positioning feature for better adaptability and reusability. Future work could analyze how this variability influences logical architectures.

MR3: Consistency Transformation rules ensure consistency between multiple model views. All relevant elements from the operating principle are regarded in the logical architecture. The risk is that operating principles may not contain all required interfaces as object flow, which must be minimized through model reviews, validation, or expert interviews.

MR4: Traceability Our transformation supports traceability in system specifications, as required by standards like ISO 26262 [66] and A-SPICE [48]. Requirements must be bidirectionally traceable, meaning all elements from the operating principle must be traceable to their executing element in the logical architecture and vice versa. This is achieved through the design of the transformation rules. Actions are allocated to their executing element following the action execution rule (Figure 7), creating a bidirectional mapping. Interfaces derived from actions and their object flows ensure traceability of each object flow to the resulting parameters in the logical architecture. Since the reverse direction and application, as presented in ref. [39], is more complicated (but generally possible), we suggest creating links between parameter/activity pin and port in the modeling tool to simplify navigation for systems engineers, as stated in Section 5.1. The architecture modeling language might impact traceability [67], so future work could explore whether similar approaches in other modeling languages support traceability better or worse. Apart from traceability, ISO 26262 also states other requirements for developing an architecture for safety-critical systems, which are not discussed here. Other works focus on developing a SOTIF-compliant modeling process [68] or the technical electric/electronic architecture [69], which could be good research aspects for future work.

MR5: Reduced Effort Automated transformation between different views in system specification reduces the effort required by systems engineers. If the systems engineer changes perspective, there should be no manual modeling effort required. Although additional tasks like diagram layout or view selection may still be needed, the SysML v2 specification, which provides a textual modeling interface, offers a solution [70]. From a critical perspective, however, one could argue that additional tasks such as the diagram layout or view selection of single or multiple requirements and update tasks are still required. Regarding the layout, we are confident that the SysML v2 specification, which also provides a textual modeling interface, provides a solution for this issue.

5.4 | Applicability of the Approach

Although the proposed methodology offers significant benefits in terms of automation and efficiency, it is essential to understand the types of problems for which this approach is most valid.

The CUBE-based feature-driven methodology is particularly effective for systems that require:

- *High Reusability*: Systems where features can be reused across different products or projects, such as automotive systems, aerospace, and industrial automation.
- *Complex Multi-Perspective Modeling*: Projects that benefit from multi-perspective modeling to capture different aspects of the system, including functional, structural, and behavioral views.
- *Consistency and Traceability*: Systems that require consistent and traceable specifications across different development stages and views, ensuring high-quality documentation and reduced errors.

In addition to the automotive systems considered, the CUBE-based feature-driven methodology could also be effective in various domains and applications, including aerospace systems,

critical safety medical systems, consumer electronics, industrial automation, cyber-physical systems, transportation networks, customizable manufacturing systems, research and development projects, Enterprise software solutions, and other highly regulated industries. This methodology enhances the development of complex features, ensures consistency and traceability, supports high reusability, and facilitates the integration of physical and digital components, making it ideal for projects requiring complex multi-perspective modeling and compliance with regulatory standards. As an implementation guideline, potential users of our approach could ask if the following steps are required in their application scenario:

- *Action Identification*: Clearly identify all actions that need to be executed by the element.
- *Action Allocation*: Use a LRA to allocate these actions to the appropriate elements.
- *Dependency Modeling*: Ensure that dependencies between actions are well-documented and modeled using SysML diagrams.
- *Transformation Rule Application*: Apply transformation rules to generate the logical architecture from the behavior models, ensuring that these rules account for multiple actions and their interactions.

For instance, in an automotive system as specified in our running example, actions such as steering, braking, and accelerating can be allocated to different elements of the vehicle architecture. Transformation rules can be used to generate a consistent logical architecture that accounts for these actions and their dependencies. Moreover, our approach can be also extended to generate and analyze multiple allocation scenarios resulting in different architecture alternatives. Consequently, our approach provides several benefits in the aforementioned scenarios and application areas with similar problems could benefit from its application.

5.5 | Assumptions, Limitations, and Guidelines

In other scenarios, however, there are some assumptions and limitations that need to be discussed and addressed with an additional guideline. As presented in Section 4, our approach has some inherent implications and limitations from its design. First, only subsystems of the currently regarded systems are able to execute actions of an operating principle. This is not only a limitation of the approach but also a process requirement for the underlying method since a system can only perform actions through its own elements and not through the elements of other systems. Moreover, internal information flows within the same system element are not regarded. While this implication from the second rule (*cf.* Figure 8) is crucial for the system design to prevent the introduction of unused interfaces, it bears a risk for system optimization, as new functionalities that require the same information, material, or energy might be ideally performed by the same system elements. To consider this information in the extension of the logical architecture, further works could focus on a capability model to take this information for architecture optimization into account. As the third rule (*cf.* Figure 9) does not filter out illegal connections (e.g., a port becomes multiple inputs from different systems), additional steps must be performed to filter out these takes. In our example implementation for the evaluation project, we implemented additional checks for these tasks analogous to context conditions. Finally, the fourth rule (*cf.* Figure 10) interprets all unresolved interfaces as system interfaces. Even though this allows high reuse of the features concerning MR1 and is technically correct (since not provided objects must inherently be provided from the system environment), in a later validation phase of the generated logical architecture, it must be assured that this information is actually provided to the system. In our application project, we solved this by introducing additional reviews with domain experts and changing the operating principles according to their feedback when errors were found.

From the application point of view, we did not perform a complete tool evaluation. Since we have focused only on the methodological feasibility of the transformation and have deliberately excluded the aspect of tool evaluation. As a result, Section 5.1 is primarily a qualitative statement about the functionality demonstration, which is not to be understood as a complete tool evaluation.

The execution and timing behavior of the operating principle and resulting logical architectures is not part of the approach presented in this paper. As mentioned in Section 2, other works on ADs mostly focus on their semantics and transformations to Petri nets. Since we decided not to further focus on this aspect and highlight their characteristics as a specification in this paper, the aspect of simulating or executing the operating principles is not further addressed in this paper. However, we want to highlight that the activity diagrams that model the operating principles alone are not always sufficient to describe the timing behavior of the system. To address this problem, previous works provided methods to derive test cases from activity diagrams [29, 71], or scenario-based specification [72]. Apart from these aspects, the modeling approach for modeling functionalities is based on modeling and decomposing activity diagrams, which itself is an interesting research and application topic for which several approaches exist [73, 74]. Since we did not focus on the several possible decomposition strategies, the approach we present relies on the modeler correctly modeling and decomposing the activity diagrams in the operating principles of a feature.

Moreover, our approach inherently only allows for a single allocation of an action to an element from the LRA, and thus, cardinalities cannot be modeled using the approach. Since we use this approach only for the generation of logical architectures, in which we assume that only one instance shall be responsible for the execution of an action, this might become a problem if a similar approach for the generation of a product architecture is developed, where multiple executing elements might also perform a task. For example, in most vehicles, transmitting the driving force to the road is realized by four tires in the technical architecture, which cannot be expressed or transformed in the currently presented approach, since we assume that logical architectures, in contrast to the technical realization in the technical architecture, are free of redundancies. Consequently, similar automation for the next abstraction layer, the technical architecture, would require an extended set of transformation rules capable of handling redundancy and cardinalities.

Even though we focused on the automotive industry in our example, we believe that the approach presented in this paper is also applicable to other industries, e.g., for creating energy systems and wind turbines [75], in avionics [76], transportation systems, industrial automation [77], or maritime engineering [78]. We cannot make any definitive statements about this aspect, however, we are convinced that our presented approach might have a beneficial impact during the development of systems in other industries where a shared reference structure is identifiable, as the transformation we proposed in Section 4.4 is generally applicable in SysML.

5.6 | Future Improvements and Possible Extensions

To automate our approach and further improve model-driven systems engineering, transformation rules could be extended to generate ADs from use case specifications, as sketched in early works on Use Case Diagrams (UCDs) [79, 80]. Additionally, transformation rules do not currently consider event-based communication, as the considered projects did not use these model elements. Future work may introduce rules to cover this form of actions.

As mentioned in L2 and Section 5.3, the variability of ADs used for operating principle specification and their impact on the generated logical architecture could be interesting for future research. This approach focuses on state machine diagrams in the LRA and ADs in operating principle modeling. Future work could extend our approach by using other SysML behavior models, such as sequence diagrams or models from our Domain Specific Languages (DSLs). The current approach does not account for the timing behavior of the system, which could be a future extension.

The evaluation of our approach focused on an automotive example. Follow-up research might apply the same concepts to other domains, such as railway engineering, aerospace engineering, marine engineering, manufacturing, civil engineering, or power plant engineering.

Practically, the approach is extendable to a SysML toolchain, which applies the proposed transformation rules from Section 4.4 to create new models and uses these rules for compliance checks for model validation. Models not initially created with our methodology could benefit from these insights by checking the conformance of functional behavioral interfaces.

Our approach relies on using a correct LRA for optimal action allocation. Although we assumed in Section 4.2 that this architecture is known beforehand, this is not always the case in other domains or automotive problems where required knowledge is not known. Iteratively optimizing the reference architecture for action allocation could be beneficial. Starting with a draft of the LRA and continually improving it before, during, and after using our approach to generate the full architecture could identify gaps and improve outcomes. Metrics for architecture quality aspects, such as modularity, coexistence, or interoperability, could evaluate designs and aid systems engineers in improving the logical architecture. Developing a family of architectures for a family of problems might also be beneficial.

Finally, with the upcoming SysML v2, the challenge arises to transfer our approach to the different viewpoints of this new standard.

6 | Conclusion

We have demonstrated how to apply model-to-model transformation methods in an industrial context to create SysML IBD models from SysML AD models, allowing domain experts to derive logical architectures from operating principles based on a LRA. We applied this approach to a reduced example from the automotive domain, providing insight into the high number of actions, parameters, and object flows resulting from a relatively small number of features. Clearly, manual modeling of these concepts while keeping the models consistent is not realistic in practice. Thus, generative approaches significantly improve the process.

Applying this method improves the consistency between models from different perspectives and reduces the manual effort for developers. This enables the use of model-driven methods in the industry to enhance the systems engineering process and increase the quality of the resulting products.

With respect to our research questions, we can draw the following conclusions:

- **RQ1**: Automation, as suggested in Section 4, enhances efficiency. Logical architectures and solution-neutral operating principles are often independent of the development project and the concrete system, making them reusable on multiple levels.
- **RQ2**: We suggest these modeling methods and artifacts to other practitioners of MBSE due to their adaptability and reusability.
- **RQ3**: The operating principles in CUBE are usable for the generation of logical architectures. However, the suggested methods and applications cannot answer all aspects of our research question, such as whether more ways of automation of CUBE and other systems engineering exist. Therefore, we see potential for further research in this area.

In summary, our findings indicate that automating model creation and reusing systems engineering artifacts whenever possible and beneficial can significantly improve the efficiency and quality of the systems engineering process. For other practitioners of MBSE, we recommend adopting these practices to enhance their development workflows.

Acknowledgments

Open access funding enabled and organized by Projekt DEAL.

Data Availability Statement

Research data are not shared.

Endnotes

[1] Very High Speed Integrated Circuit Hardware Description Language (VHDL)

[2] Analog and Mixed-Signal extensions (AMS)

References

1. I. Drave, B. Rumpe, A. Wortmann, et al., "Modeling Mechanical Functional Architectures in SysML," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (ACM, 2020), 79–89.

2. C. Seidl, D. Wille, and I. Schaefer, "Software Reuse: From Cloned Variants to Managed Software Product Lines," in *Automotive Systems and Software Engineering: State of the Art and Future Trends*, ed. Y. Dajsuren, V. dM. Brand (Springer, 2019), 77–108.

3. A. W. Wymore, *Model-Based Systems Engineering* (CRC press, 2018).

4. K. Hölldobler, J. Michael, J. O. Ringert, B. Rumpe, and A. Wortmann, "Innovations in Model-based Software and Systems Engineering," *Journal of Object Technology (JOT)* 18, no. 1 (2019): 1–60, https://doi.org/10.5381/jot.2019.18.1.r1.

5. J. D'Ambrosio and G. Soremekun, "Systems engineering challenges and MBSE opportunities for automotive system design," in *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)* (2017), 2075–2080.

6. S. Kriebel, J. Richenhagen, C. Granrath, and C. Kugler, "Systems Engineering with SysML: The Path to the Future?," *MTZ worldwide* 78 (2018): 44–47, https://doi.org/10.1007/s38313-018-0030-8.

7. C. Granrath, C. Kugler, S. Silberg, et al., "Feature-Driven Systems Engineering Procedure for Standardized Product- Line Development," *Systems Engineering* 24, no. 6 (2021): 456–479, https://doi.org/10.1002/sys.21596.

8. K. X. Campo, T. Teper, C. E. Eaton, A. M. Shipman, G. Bhatia, and B. Mesmer, "Model-Based Systems Engineering: Evaluating Perceived Value, Metrics, and Evidence Through Literature," *Systems Engineering* 26, no. 1 (2022): 104–129.

9. A. M. Madni and S. Purohit, "Economic Analysis of Model-Based Systems Engineering," *Systems* 7, no. 1 (2019): 12, https://doi.org/10.3390/systems7010012.

10. J. Michael, D. Bork, M. Wimmer, and H. C. Mayr, "Quo Vadis Modeling? Findings of a Community Survey, an Ad-Hoc Bibliometric Analysis, and Expert Interviews on Data, Process, and Software Modeling," *Journal Software and Systems Modeling (SoSyM)* 23, no. 1 (2024): 7–28, https://doi.org/10.1007/s10270-023-01128-y.

11. D. D. Walden, G. Roedler, K. Forsberg, D. Hamelin, and T. Shortell, *Systems Engineering Handbook - A Guide for System Life Cycle Processes and Activities*, 4th ed. (John Wiley & Sons, 2015).

12. K. Pohl, H. Hönninger, R. Achatz, and M. Broy, *Model-Based Engineering of Embedded Systems* (Springer Berlin, Heidelberg, 2012).

13. K. Pohl, M. Broy, H. Daembkes, and H. Hönninger, *Advanced Model-Based Engineering of Embedded Systems, Extensions of the SPES 2020 Methodology* (Springer International Publishing, 2016).

14. G. Shea, *NASA Systems Engineering Handbook Revision*, 2nd ed. (NASA, 2019), https://www.nasa.gov/connect/ebooks/nasa-systems-engineering-handbook (2017).

15. C. Granrath, "Feature-Getriebene Systementwicklung Von Produktlinien Mittels Referenzarchitektur für Simulationsmodelle Elektrischer Automobilantriebe," (PhD Thesis, RWTH Aachen University, 2022).

16. J. A. Estefan, "Survey of Model-Based Systems Engineering (MBSE) Methodologies," *INCOSE MBSE Focus Group* 25, no. 8 (2007): 1–70.

17. C. B. Nielsen, P. G. Larsen, J. Fitzgerald, J. Woodcock, and J. Peleska, "Systems of Systems Engineering: Basic Concepts, Model-Based Techniques, and Research Directions," *ACM Comput. Surv.* 48, no. 2 (2015), https://doi.org/10.1145/2794381.

18. T. Huldt and I. Stenius, "State-of-Practice Survey of Model-Based Systems Engineering," *Systems Engineering* 22, no. 2 (2019): 134–145.

19. M. Dalibor, N. Jansen, B. Rumpe, L. Wachtmeister, and A. Wortmann, "Model-Driven Systems Engineering for Virtual Product Design," in *Proceedings of MODELS 2019. Workshop MPM4CPS*, eds. L. Burgueño, A. Pretschner, S. Voss, et al., (IEEE, 2019), 430–435.

20. S. Ali, M. A. Basit-Ur-Rahim, and F. Arif, "Formal Verification of Time Constrains SysML Internal Block Diagram Using PRISM," in *2015 15th International Conference on Computational Science and Its Applications* (Springer Cham, 2015), 62–66.

21. H. Kausch, J. Michael, M. Pfeiffer, D. Raco, B. Rumpe, and A. Schweiger, "Model-Based Development and Logical AI for Secure and Safe Avionics Systems: A Verification Framework for SysML Behavior Specifications," in *Aerospace Europe Conference 2021 (AEC 2021)* Council of European Aerospace Societies (CEAS, 2021).

22. R. Eikermann, K. Hölldobler, A. Roth, and B. Rumpe, "Reuse and Customization for Code Generators: Synergy by Transformations and Templates," in *Model-Driven Engineering and Software Development*, ed. S. Hammoudi, L. F. Pires, B. Selić (Springer, 2019), 34–55.

23. J. C. Kirchhof, J. Michael, B. Rumpe, S. Varga, and A. Wortmann, "Model-Driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems," in *23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (ACM, 2020), 90–101.

24. C. Buschhaus, A. Gerasimov, J. C. Kirchhof, et al., "Lessons Learned From Applying Model-Driven Engineering in 5 Domains: The Success Story of the MontiGem Generator Framework," *Science of Computer Programming* 232 (2024): 103033, https://doi.org/10.1016/j.scico.2023.103033.

25. A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner, "Software Engineering for Automotive Systems: A Roadmap," in *Future of Software Engineering (FOSE '07)* (2007), 55–71.

26. M. Broy, "Automotive Software and Systems Engineering," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE '05* IEEE Computer Society 1730 (2005): 143–149.

27. A. Rasmus, "Systems Engineering Roadmap for Dependable Autonomous Cyber-Physical Systems" in *2021 Design, Automation Test in Europe Conference Exhibition* (DATE 2021), 1622–1625.

28. J. Ewald, P. Orth, C. Granrath, and J. Andert, "Model Based Systems Engineering for Standardized Simulation Frameworks: Case Study Development of Electrical Vehicles," in *41st International Vienna Motor Symposium* (VDI Verlag, 2020), 426–441.

29. I. Drave, T. Greifenberg, S. Hillemacher, et al., "SMArDT Modeling for Automotive Software Testing," *Journal on Software: Practice and Experience* 49, no. 2 (2019): 301–328.

30. S. Kriebel, V. Moyses, G. Strobl, et al., "The Next Generation of BMW's Electrified Powertrains: Providing Software Features Quickly by Model-Based System Design," in *26th Aachen Colloquium Automobile and Engine Technology*, (RWTH 2017).

31. H. Grönniger, J. Hartmann, H. Krahn, S. Kriebel, L. Rothhardt, and B. Rumpe, "Modelling Automotive Function Nets With Views for Features, Variants, and Modes," in *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, (RWTH 2008).

32. H. Grönniger, J. Hartmann, H. Krahn, S. Kriebel, and B. Rumpe, "View-Based Modeling of Function Nets," in *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)* (2007).

33. H. Grönniger, J. Hartmann, H. Krahn, S. Kriebel, L. Rothhardt, and B. Rumpe, "View-Centric Modeling of Automotive Logical Architectures," in *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte*

Entwicklung eingebetteter Systeme IV Informatik Bericht 2008-02. TU Braunschweig (arXiv, 2008).

34. E. Andrade, P. Maciel, G. Callou, and B. Nogueira, "A Methodology for Mapping SysML Activity Diagram to Time Petri Net for Requirement Validation of Embedded Real-Time Systems with Energy Constraints," in *2009 Third International Conference on Digital Society* IEEE Computer Society , (IEEE, 2009), 266–271.

35. E. Huang, L. Mcginnis, and S. Mitchell, "Verifying SysML Activity Diagrams Using Formal Transformation to Petri Nets," *Systems Engineering* 23, no. 1, (2020): 23.

36. M. Rahim, A. Hammad, and M. Boukala-Ioualalen, "Towards the Formal Verification of SysML Specifications: Translation of Activity Diagrams into Modular Petri Nets," in *3rd Int. Conf. on Applied Computing and Information Technology/2nd Int. Conf. on Computational Science and Intelligence* (IEEE, 2015), 509–516.

37. D. Foures, V. Albert, J. C. Pascal, and A. Nketsa, "Automation of SysML Activity Diagram Simulation with Model-Driven Engineering Approach," in *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium* TMS/DEVS '12. (Society for Computer Simulation International, 2012).

38. F. Jouault and I. Kurtev, "Transforming Models with ATL," in *Satellite Events at the MoDELS 2005 Conference*, ed. J. M. Bruel (Springer, 2006).

39. M. Melo, J. Franca, E. Oliveira Jr, and M. Soares, "A Model-Driven Approach to Transform SysML Internal Block Diagrams to UML Activity Diagrams," in *ICEIS 2015-17th International Conference on Enterprise Information Systems, Proceedings*, (ICEIS-2015), 92–101.

40. J. Lamm and T. Weilkiens, "Functional Architectures in SysML," *Proceedings of the TdSE* (GfSE Verlag, 2010).

41. J. G. Lamm and T. Weilkiens, "Method for Deriving Functional Architectures From Use Cases," *Systems Engineering* 17, no. 2 (2014): 225–236.

42. M. Fockel, J. Holtmann, and J. Meyer, "Semi-Automatic Establishment and Maintenance of Valid Traceability in Automotive Development Processes," in *2012 Second International Workshop on Software Engineering for Embedded Systems (SEES)* (IEEE Press, 2012), 37–43.

43. S. Friedenthal, A. Moore, and R. Steiner, *A practical guide to SysML: the systems modeling language* (Morgan Kaufmann, 2014).

44. O. Kautz, B. Rumpe, and L. Wachtmeister, "Semantic Differencing of Use Case Diagrams," *Journal of Object Technology (JOT)* 21, no. 3 (2022): 1–14.

45. N. Jäckel, C. Granrath, L. Wachtmeister, A. Karaduman, B. Rumpe, and J. L. Andert, "Feature-Driven Specification of VTOL Air-Taxis with the Use of the Model-Based Systems Engineering (MBSE) Methodology CUBE," in *77th Annual Vertical Flight Society Forum and Technology Display (FORUM 77)* (Curran Associates, Inc. , 2021), 2776–2784.

46. N. Jäckel, C. Granrath, R. Schaller, et al., "Integration of VTOL Air-Taxis Into an Existing Infrastructure With the Use of the Model-Based System Engineering (MBSE) Concept CUBE," in *Vertical Flight Society Annual Forum & Technology Display*, (Curran Associates, Inc. , 2020).

47. M. Guiggiani, *The Science of Vehicle Dynamics*, 3th ed. (Springer Cham, 2023).

48. [The SPICE User Group], "Automotive SPICE Process Assessment/Reference Model V3.0,"*Automotive SPICE Process Assessment/Reference Model V3.0*. (VDA QMC, 2015).

49. C. Stoermer, A. Rowe, L. O'Brien, and C. Verhoef, "Model-Centric Software Architecture Reconstruction," *Software: Practice and Experience* 36, no. 4 (2006): 333–363, https://doi.org/10.1002/spe.699.

50. IBM, *IBM Engineering Systems Design Rhapsody* (IBM, 2020).

51. Sparx Systems, *Enterprise Architect* (spark system, 2021).

52. Dasslault Systems, *Cameo Systems Modeler* (Dasslault Systems, 2020).

53. M. Broy, "Challenges in Automotive Software Engineering," in *Proceedings of the 28th international conference on Software engineering* (Association for Computing Machinery, 2006), 33–42.

54. M. Broy, "Model-Driven Architecture-Centric Engineering of (Embedded) Software Intensive Systems: Modeling Theories and Architectural Milestones," *Innovations in Systems and Software Engineering* 3 (Springer, 2007): 75–102.

55. M. Broy, "A Logical Approach to Systems Engineering Artifacts: Semantic Relationships and Dependencies beyond Traceability–from Requirements to Functional and Architectural Views," *Softw. Syst. Model.* 17, no. 2 (2018), 365–393, https://doi.org/10.1007/s10270-017-0619-4.

56. C. Riva and J. V. Rodriguez, "Combining Static and Dynamic Views for Architecture Reconstruction," in *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering* (IEEE, 2002), 47–55.

57. J. Grundy and J. Hosking, "High-Level Static and Dynamic Visualisation of Software Architectures," in *Proceeding 2000 IEEE International Symposium on Visual Languages* (IEEE, 2000), 5–12.

58. F. Oquendo, "π-ADL: An Architecture Description Language Based on the Higher-Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures," *ACM SIGSOFT Software Engineering Notes* 29, no. 3 (2004): 1–14.

59. J. O. Ringert, B. Rumpe, and A. Wortmann, "A Case Study on Model-Based Development of Robotic Systems Using Montiarc With Embedded Automata," in *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme*, ed. H. Giese, M. Huhn, J. Philipps, B. Schätz (2013), 30–43.

60. A. Butting, A. Haber, L. Hermerschmidt, O. Kautz, B. Rumpe, and A. Wortmann, "Systematic Language Extension Mechanisms for the Montiarc Architecture Description Language" in *European Conference on Modelling Foundations and Applications (ECMFA'17)* LNCS 10376 (Springer, 2017), 53–70.

61. Object Management Group, *OMG Systems Modeling Language* version 1.6th (OMG, 2019).

62. M. P. Nepal, M. Park, and B. Son, "Effects of Schedule Pressure on Construction Performance," *Journal of Construction Engineering and Management* 132, no. 2 (2006): 182–188.

63. H. Störrle, "On the Impact of Layout Quality to Understanding UML Diagrams," in *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (IEEE, 2011), 135–142.

64. C. Natschläger, V. Geist, F. Kossak, and B. Freudenthaler, "Optional Activities in Process Flows," in *EMISA 2012 – Der Mensch im Zentrum der Modellierung* LNI. GI (Gesellschaft für Informatik eV, 2012), 67–80.

65. A. Heuer, V. Stricker, C. Budnik, S. Konrad, K. Lauenroth, and K. Pohl, "Defining Variability in Activity Diagrams and Petri nets," *Science of Computer Programming* 78 (2013): 2414–2432, https://doi.org/10.1016/j.scico.2012.06.003.

66. ISO 26262:2011, ISO 26262: Road Vehicles: Functional Safety Standard, (ISO, 2011).

67. M. Ballarín, L. Arcega, V. Pelechano, and C. Cetina, "On the Influence of Architectural Languages on Requirements Traceability," *Software: Practice and Experience* 53, no. 3 (2022): 704–728.

68. M. Meyer, C. Granrath, L. Wachtmeister, and N. Jäckel, "Methods for the Development of Collaborative Embedded Systems in Automated Vehicles," *ATZelectronics worldwide* 15, no. 12 (2020): 58–63.

69. M. Hillenbrand, M. Heinz, J. Matheis, and K. D. Müller-Glaser, "Development of Electric/Electronic Architectures for Safety-Related Vehicle Functions," *Software: Practice and Experience* 42, no. 7 (2012): 817–851, https://doi.org/10.1002/spe.1154.

70. N. Jansen, J. Pfeiffer, B. Rumpe, D. Schmalzing, and A. Wortmann, "The Language of SysML v2 under the Magnifying Glass," *Journal of Object Technology (JOT)* 21 (2022): 1–15.

71. C. Mingsong, Q. Xiaokang, and L. Xuandong, "Automatic Test Case Generation for UML Activity Diagrams," in *Proceedings of the 2006 International Workshop on Automation of Software Test*AST '06, (Association for Computing Machinery, 2006): 2–8.

72. M. A. Meyer, S. Silberg, C. Granrath, et al., "Scenario- and Model-Based Systems Engineering Procedure for the SOTIF-Compliant Design of Automated Driving Functions" in *2022 IEEE Intelligent Vehicles Symposium (IV'22)* (IEEE, 2022), 1599–1604.

73. H. Chen, J. m. Jiang, Z. Hong, and L. Lin, "Decomposition of UML Activity Diagrams," *Software: Practice and Experience* 48, no. 1 (2018): 105–122, https://doi.org/10.1002/spe.2519.

74. H. Grönniger, D. Reiß, and B. Rumpe, "Towards a Semantics of Activity Diagrams With Semantic Variation Points," in *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*LNCS 6394, (Springer, 2010), 331–345.

75. J. Michael, I. Nachmann, L. Netz, B. Rumpe, and S. Stüber, "Generating Digital Twin Cockpits for Parameter Management in the Engineering of Wind Turbines," in *Modellierung 2022*, (Gesellschaft für Informatik, 2022), 33–48.

76. H. Kausch, M. Pfeiffer, D. Raco, B. Rumpe, and A. Schweiger, "Enhancing System Model Quality: Evaluation of the Systems Modeling Language (SysML)-Driven Approach in Avionics," *Journal of Aerospace Information Systems* 22, no. 2 (2025): 1–12, https://doi.org/10.2514/1.I011476.

77. P. Brauner, M. Dalibor, M. Jarke, et al., "A Computer Science Perspective on Digital Transformation in Production," *Journal ACM Transactions on Internet of Things* 3 (2022): 1–32, https://doi.org/10.1145/3502265.

78. B. Sullivan and M. Rossi, "SE Based Development Framework for Changeable Maritime Systems," in *Product Lifecycle Management. Leveraging Digital Twins, Circular Economy, and Knowledge Management for Sustainable Innovation*, ed. C. Danjou, R. Harik, F. Nyffenegger, L. Rivest, A. Bouras, (Springer Nature Switzerland cham, 2024), 203–214.

79. A. Cockburn, *Writing Effective Use Cases* (Addison-Wesley Longman Publishing Co., Inc. , 2001).

80. M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (Addison-Wesley Professional, 2004).