CASE STUDY => addresses the research question about how SysML v2 changes the landscape of modeling guidelines.

Forschung im Ingenieurwesen (2025)89:53
https://doi.org/10.1007/s10010-025-00806-1

ORIGINALARBEITEN/ORIGINALS

Enhancing model-based development with formalized requirements: integrating temporal logic and SysML v2 for comprehensive state and transition modeling

Simon Dehn[1] · Georg Jacobs[1] · Philipp Höck[1] · Gregor Höpfner[1]

Received: 16 September 2024 / Accepted: 12 February 2025

Abstract

Modern vehicles are complex cyber-physical systems with numerous customer functions and operating states, requiring careful management during development. Ensuring that all potential states and transitions are rigorously defined at the requirement stage is critical for vehicle safety and reliability. Due to stringent regulatory requirements and competitive market pressures, it is essential to articulate these requirements unambiguously, free of redundancy, and correctly.

However, even a formally correct set of requirements can be underspecified, leading to unintended operating states that compromise vehicle functionality. To ensure requirement completeness, various approaches have been proposed to generate operating states and transitions allowed by a given set of requirements, ensuring that only intended behaviors are permitted. Existing methodologies often rely on natural language requirements, prone to ambiguities and errors. Furthermore, several methods formalize requirements using temporal logic languages, which can be checked algorithmically.

Because of the complexity of modern vehicles, Model Based Systems Engineering (MBSE) has become state of the art in vehicle requirements engineering, as MBSE enables rigorous traceability of requirements and early model-based validation of requirement fulfillment.

Currently a gap remains in integrating the comprehensive generation of operating states and transitions with the formalization in temporal logic languages, specifically for the use in MBSE development environments.

This paper introduces a multi-step approach that formalizes requirements using Linear Temporal Logic (LTL) and generates operating states and transitions, which are then modeled in a SysML v2 state machine diagram, enhancing both the development process and the system's reliability.

1 Introduction

Modern vehicles are becoming increasingly sophisticated cyber-physical systems, characterized by a vast array of customer functions and a multitude of operating states and transitions that must be carefully managed during the development process. The complexity of these systems demands that all potential operating states and their permissible transitions are rigorously defined to ensure safe and reliable vehicle behavior. Given the strict regulatory requirements and intense market competition, it is essential that these requirements are articulated in a manner that is unambiguous, free of redundancy, and correct in content. Despite this, even a formally correct set of requirements can be underspecified, leading to the possibility of unintended operating states that could compromise the vehicle's functionality or safety.

To prevent such outcomes, an approach based on [1, 2] is proposed, in which all operating states and state transitions that are allowed by a given set of requirements are generated, thereby enabling to check that only intended behaviors are permitted. However, existing methodologies predominantly rely on requirements written in natural language, which are inherently prone to ambiguities, redundancies, and other errors [3]. While the literature offers several approaches to address these issues through the formalization of requirements—e.g. translating them into temporal logic languages such as Linear Time Logic (LTL) that can be checked algorithmically [4, 5]—there remains a gap in in-

✉ Simon Dehn
post@imse.rwth-aachen.de

[1] Institute for Machine Elements and Systems Engineering at RWTH Aachen University, Schinkelstraße 10, 52062 Aachen, Germany

Published online: 14 March 2025

tegrating this formalization with the comprehensive generation of operating states and state transitions given by formalized requirements.

In response to this challenge, this paper presents a multi-step approach that utilizes requirement specifications formalized by LTL expressions to generate every operating state and transition permitted, according to the specified requirements. The outcome is presented as a SysML v2 state machine diagram, facilitating further model-based development activities, i.e. as described in [6, 7]. This method not only ensures a formally consistent requirements base but also provides a robust framework for the accurate and unambiguous generation of vehicle operating states, thereby enhancing both the development process and the final product's reliability.

The work is structured as follows: The following chapter introduces the current state of the art in formalization and modeling of state-based system behavior, Chap. 3 explains the need for an integrated approach to requirement formalization and modeling. The individual steps of the proposed approach are described in Chap. 4. A simplified industry example is introduced to illustrate the process. Chapter 5 discusses current limitations of the approach regarding applicability as well as scalability and suggests further research need to enable full automation of the process.

2 State of the art

The formalization of system requirements is a critical step in ensuring the development of reliable and error-free systems, particularly in the context of complex, safety-critical applications. This chapter presents an overview of current methodologies and tools used in the formalization of requirements (2.1), the detection of off-nominal behavior (2.2), and the modeling of operational states and transitions (2.3) in cyber-physical systems (CPS).

2.1 Requirements formalization

The formalization of requirements in temporal logic expressions has emerged as an approach for ensuring the correct and reliable behavior of systems that exhibit operating states and transitions. Temporal logic provides a robust framework for specifying requirements in a mathematically precise manner [8].

Temporal logic languages use mathematical operators to define rules that govern the transitions between states. These operators, whose meanings are unambiguously defined, eliminate the vagueness often associated with natural language specifications. This precision ensures that requirements formulated in temporal logic carry a single meaning and can be consistently interpreted across different stages of the system development process.

Specifically, the Linear Temporal Logic language (LTL) is widely used to specify temporal properties of systems [9]. It allows for the expression of requirements such as safety ("something bad never happens") and liveness ("something good eventually happens"). LTL is particularly useful in model checking, where it helps to verify that a system model adheres to its temporal specifications [10]. LTL also incorporates the basic Propositional Logic operators that are commonly used to express logical relationships between propositions. The Propositional Logic and LTL operators used in this paper are shown in Table 1.

One of the most significant advantages of using temporal logic for requirements formalization is the ability to perform model checking [4]. Model checking is an automated process that verifies whether the behavior of a system complies with the specified requirements. By applying temporal logic rules to the system model, it is possible to rigorously check if all possible state transitions meet the defined requirements, thereby identifying potential design flaws early in the development cycle.

By using temporal logic languages to formalize requirements, the risk of ambiguities and errors in the system design can be significantly reduced, ultimately leading to safer and more dependable products.

2.2 Detecting off-nominal behavior in cyber-physical-systems

Underspecification remains a pervasive challenge in the requirements engineering process, often leading to unintended or off-nominal system behavior [11]. This issue arises when requirements are not detailed enough to cover all possible scenarios, leaving gaps that can result in system states not anticipated during the design phase. Such oversights can lead to critical failures or unexpected behavior in

Table 1 Propositional Logic and LTL operators used in this paper

Operator	Connection	Meaning
XOR	*A* XOR *B*	*A* XOR *B* is true if either *A* or *B* is true. It is false if *A* and *B* is true
!	*!A*	*!A* is true if *A* is false
→	$A \rightarrow B$	If *A* is true, then *B* must also be true
X	X *B*	In the next state, *B* must be true
G	G *B*	*B* must be true in all states

complex systems, particularly in domains such as automotive engineering, where reliability and safety are paramount [12].

One notable approach to identifying and mitigating off-nominal behavior caused by under specification is presented in [1]. Their methodology focuses on the concept of global system states, which are defined as the aggregate of the states of all system components. By considering all possible combinations of component states, this approach ensures a comprehensive examination of the system's operating space.

In the framework introduced in [1], transitions between global system states are triggered by specific events, with each transition corresponding to a change in exactly one sub-state. This granular approach to state transitions allows for precise tracking of how the system evolves in response to different stimuli. The rules governing these sub-state changes are derived directly from natural language (NL) requirements, which are systematically extracted and applied to the explicit global system states. By mapping these rules onto the full range of possible system states, the approach aims to identify transitions that lead to undesirable or off-nominal states. This method is particularly effective in uncovering potential system vulnerabilities that may not be immediately apparent from the requirements alone. The identification of these undesirable state transitions is crucial for refining the requirements and ensuring that the system behaves as intended under all possible conditions.

By ensuring that all potential global system states are considered and that transitions are rigorously analyzed, their approach provides a robust mechanism for enhancing system reliability and reducing the likelihood of unexpected behavior.

2.3 Formal modeling of operational states and state transitions

Modeling the operating states and transitions of cyber-physical systems (CPS) supports evaluating their correct and reliable functioning, especially given the complex interplay between physical processes and computational control. The state of the art in this domain encompasses a variety of methodologies, each with its strengths and application areas, aimed at accurately representing and managing the behavior of these systems.

Finite State Machines (FSM) are one of the most widely used approaches for modeling the operating states and transitions of CPS [13, 14]. FSMs represent the system as a finite number of states and transitions between those states, triggered by events or conditions. Each state represents a particular system configuration at a specific point in time. The discrete changes in system configuration over time are modeled as the state transitions. This method is particularly useful for systems with a well-defined number of states and predictable behavior. FSMs are straightforward to implement and understand, making them suitable for many industrial applications.

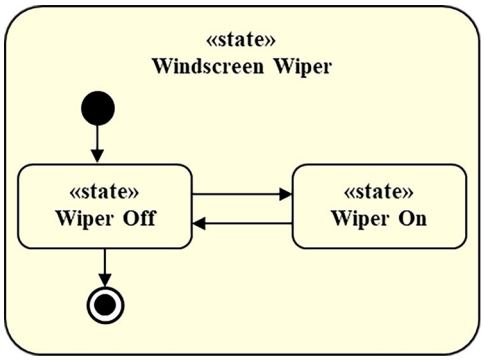

Fig. 1 Structure of a SysML v2 state machine diagram

SysML is a modeling language designed specifically for systems [15]. It provides a comprehensive suite of diagrams and tools for modeling the structure, behavior, requirements, and parameters of CPS. SysML state machine diagrams are based on the concept of FSM and are particularly useful for modeling the operating states as well as the transitions of complex systems. By integrating state-based behavior with other aspects of the system, SysML enables a holistic approach to system design and analysis. Its versatility and support for model-based systems engineering (MBSE) make it a preferred choice for large-scale and interdisciplinary projects. In its most recent version SysML v2, models can be described with code and saved to a database via an API interface, allowing for an easier way to automatically generate models using algorithms.

In this paper we choose to model the operating states and state transitions in SysML v2 state machines, as it can be continuously used in an MBSE design process. The structure of such a state machine is shown in Fig. 1. State machines consist of a composite state, capturing the full system behavior, and nested states that define the system modes. Transitions between states are depicted as open arrows. Full circles mark the starting state of a system. The end state is represented by a banded circle. In the given example, the behavior of a windscreen wiper is modeled. When the system is initialized, the wiper is turned off. The system can transition from the 'Wiper on' state to the 'Wiper off' state any number times. While in the 'Wiper off' state, the system can also transition to its end state.

3 Problem statement and research need

Despite the established methods in the individual areas in requirements modeling [1, 14, 16], there currently exists

no comprehensive method that effectively integrates the formalization of requirements using logic languages with the automated generation and analysis of permissible system states within an MBSE framework. This gap represents a limitation in the MBSE development process, where a unified approach could enhance the reliability, safety, and correctness of complex systems.

The lack of such an integrated method prevents engineers from fully leveraging the strengths of formal logic in the context of MBSE, particularly in ensuring that all possible system states are accounted for and that unintended behaviors are systematically identified and mitigated. As a result, the development process remains fragmented, with potential vulnerabilities that could lead to system failures being overlooked until later stages of development, where rectification is costlier in both, time and money [12, 17].

Therefore, there is a need for a method that combines the formalization aspect of logic languages with the capability to generate and evaluate permissible system states and transitions, seamlessly integrating this process into the MBSE workflow. Such a method would not only enhance the accuracy and thoroughness of system design but also provide a robust framework for early detection and correction of potential system failures, thereby improving the overall safety and reliability of complex cyber-physical systems. This leads to the following problem statement:

How can LTL expressions be transformed into a SysML v2 state-machine model, that represents all permissible system states and state transitions as they are specified by the LTL expressions?

In the following chapter, a multi-step approach is introduced to address this challenge. This approach is founded on several key concepts:

Capturing all system states and state transitions allowed to occur according to the requirements, is an effective strategy for formalizing these requirements in a system model. This approach facilitates further analysis, particularly in assessing the completeness of the requirements.

To enhance integration with MBSE workflows, the model is developed using SysML v2 state machines. As SysML v2 is a contemporary modeling language tailored for mechatronic systems and state machines are well suited to capture state-based behavior [14, 15]. Creating such a model requires extracting detailed information about system behavior from natural language requirements. To structure this information and ensure unambiguousness, the data is formalized using LTL. This formalization supports automatic processing through algorithms for generating the desired model.

4 Approach for generating and modeling operational states and state transitions

In this chapter, an approach to generate a SysML v2 state machine from natural language requirements is introduced. The state machine contains all system operating states and state transitions that are conform to the requirements set. These elements are referred to as permissible states and permissible transitions. Each permissible state consists of a unique set of system requirements, that must be fulfilled in the given state.

To facilitate the generation and subsequent modeling of permissible states and transitions, information must be extracted and transformed from the requirement specifications, which are usually stated in natural language. To model the permissible states, the state dependent requirements of the system, as well as rules regarding possible combinations of these requirements must be identified. These rules are structured using Propositional Logic, as this allows for automatic identification of permissible states using algorithms. Additionally, rules regarding the temporal order of states are identified and formalized using LTL. These rules are used to identify the permissible transitions.

Figure 2 shows the steps and artifacts of the presented methodology as well as the corresponding sections, in which the steps of the methodology are presented in detail. Section 4.1 describes how the state-dependent system parameters and requirements as well as rules for states and

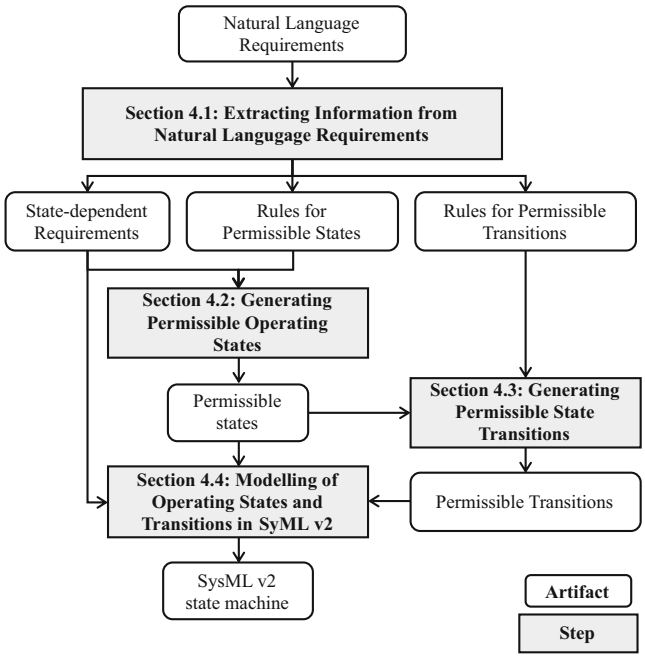

Fig. 2 Overview of the steps and artifacts of the presented methodology

Table 2 Requirements for a simplified lighting system of a car

#	Requirement Expression Natural Language
1	The system shall be operating at brightness levels of either 0 cd, 17,500 cd or 50,000 cd
2	The traffic sensor shall always be off while the system operates at a brightness of 0 cd
3	Before the system is set to a brightness level of 50,000 cd, it has to be set to 17,500 cd first
4	While the system is operating at a brightness level of 17,500 cd or 50,000 cd, the traffic sensor is also activated
5	When traffic is detected, the brightness level shall be either 17,500 cd or 0 cd

state transitions are extracted and modeled. Section 4.2 describes how the information regarding state-dependent system requirements and Propositional Logic expressions is used to generate the system states that are permitted by the requirements. In Sect. 4.3 the generation of permissible state transitions based on LTL expressions is shown. Section 4.4 closes Chap. 4 by describing the compilation of a SysML v2 state machine using the artifacts created in the prior sections.

The steps of the methodology will be explained using a simplified example of a cars' lighting system and corresponding requirements, shown in Table 2.

The lighting system is composed of a low beam light, a high beam light as well as a traffic sensor that can be turned on and off. The traffic sensor is used to detect oncoming vehicles in order to prevent blinding other road users by turning off the high beam light. Over the course of the next chapters, these requirements will be systematically transformed into a state machine diagram, which enables engineers to validate the correct specification as well as the intended behavior of the system. The resulting SysML v2 model can further be used in model-based development processes.

4.1 Extracting information from natural language requirements

In order to create the necessary artifacts for generating the permissible states and state transitions, it is necessary to extract information about the systems' state-dependent parameters as well as the validity of states and state transitions. This chapter describes how this information can be derived from natural language requirements.

Identification of state-dependent system parameters and requirements The identification of state-dependent system parameters and requirements represents the first step in our multi-step approach. We define a state-dependent system parameter as a variable of the system, which has to take on different discrete values, depending on the requirement specification. To create a machine-readable model, each natural language requirement expression is converted into a parameter-based requirement expression. This is done by manually analyzing the system requirements. In the requirements, first parameters that are assigned multiple values are identified. For each value assigned to the parameter, a parameter-based requirement expression is created. These expressions consist of the identified state-dependent parameter, i.e. *brightness_level* (cf. Table 3), and one of the values assigned to it. One expression is created for each value the parameter can take according to the requirements. For reasons of readability, this parameter-based requirement expression is then defined as a Boolean parameter, i.e. *B_1*, and assigned to a specific slot in a list. The corresponding Boolean parameters' value will be used to indicate whether the requirement needs to be fulfilled in a given state or not. Specifically, if a requirement is true in a state, the corresponding Boolean parameter value is set to one; if the requirement does not need to be fulfilled, the value is zero. This approach captures system states as vectors of Boolean values.

The process of creating this representation begins with the identification of requirements that contain state-dependent parameters within the natural language specifications. This identification is done manually by analyzing the re-

Table 3 Requirement specification containing state-dependent system parameters

#	Requirement Expression Natural Language	Slot	Requirement Expression Parameter-based	Boolean Parameter
1	The system shall be operating at brightness levels of either 0 cd, 17,500 cd or 50,000 cd	1	Brightness_level = 0 cd	B_1
		2	Brightness_level = 17,500 cd	B_2
		3	Brightness_level = 50,000 cd	B_3
2	The traffic sensor shall always be off while the system operates at a brightness of 0 cd	4	Traffic_sensor = on	T_1
		5	Traffic_sensor = off	T_2
5	When traffic is detected, the brightness level shall be either 17,500 cd or 0 cd	6	Traffic_detected = true	D_1
		7	Traffic_detected = false	D_2

Table 4 Deriving Propositional Logic expressions from natural language requirements

#	Requirement Expression Natural Language	Derived Propositional Logic Expression
1	The system shall be operating at brightness levels of either 0 cd, 17,500 cd or 50,000 cd	(*B_1* XOR *B_2* XOR *B_3*)
2	The traffic sensor shall always be off while the system operates at a brightness of 0 cd	(*T_2*) → (*B_1*)

quirement specification. Table 3 shows the requirement expressions *#1*, *#2 and #3,* stated in natural language, and their derived state-dependent parameters *brightness_level, traffic_sensor,* and *traffic_detected.* Their respective values are assigned to the list slots 1–7, as stated in the requirement. The corresponding Boolean parameter names are defined as *B_1, B_2, B_3, T_1, T_2, D_1* and *D_2.* Subsequently, the requirements that need to hold in a system state can be unambiguously defined by assigning the values 0 or 1 to the Boolean parameters. For example, if in a state the traffic sensor shall be turned on and the brightness level of the system shall be 50,000 cd, the corresponding Boolean vector (B_1, B_2, B_3, T_1, T_2, D_1, D_2) could be expressed as $(0, 0, 1, 1, 0, 0, 1)^T$.

Definition of rules for operating states In the next step, we define rules for single operating states in Propositional Logic expressions from the requirements stated in Table 2. These rules constrain the set of valid state-dependent requirements combinations. Some of these rules are explicitly stated within the requirements, while others are inferred from implicitly defined relations.

Table 4 shows the derived Propositional Logic expressions based on the explicitly stated requirements #1 and #2. Note, that this is not the full list of derived expressions. Requirement #1 states, that only one of the systems' brightness levels shall apply at the same time. This can be expressed using the "exclusive-or" (XOR) operator and the corresponding Boolean parameters B_1, B_2 and B_3. Requirement #2 states, that the traffic sensor shall be in off state (T_2), when the system is operating at a brightness of 0 cd (B_1). The derived logical expression is stated as T_2 "implies" (logical operator: →) B_1.

An example for an implicit requirement would be, that the traffic sensor cannot be "on" and "off" at the same time (T_1 XOR T_2). This relation will most likely not be expressed in a typical requirement document, but is necessary to clearly define the system's behavior.

Definition of rules for state transitions The third information artifact that needs to be extracted from the requirement are rules for state transitions, stated as Linear Temporal Logic (LTL) expressions (cf. Fig. 2). LTL expressions can be used to explicitly define which state transitions are allowed within the system. In our approach, this formalization process is done by manually analyzing and translating the natural language requirements set. By translating the requirements into LTL, potential ambiguities and inconsistencies are eliminated, resulting in a precise and unambiguous representation of the system's permitted state transitions.

Table 5 Translation of natural language requirements into LTL expression

#	Requirement Expression Natural Language	Derived LTL Expression
3	Before the system is set to a brightness level of 50,000 cd, it has to be set to 17,500 cd first	G((*B_1*) → X!(*B_3*))

In Table 5, an example for this translation step is shown. The natural language expression of requirement #3 is translated into the LTL expression "G((B_1) → X!(B_3))". This means, that at all times—globally ("G")—B_1 implies, that in the next state B_3 cannot be true.

4.2 Generating permissible operating states

For the generation of all permissible operating states, every possible combination of Boolean values for the vectors representing the system's states is considered, creating a comprehensive set of potential states. This exhaustive list of combinations is then subjected to a verification process, where each state vector is checked for conformity against the predefined logical rules regarding requirement combinations as outlined in Sect. 4.2 (Fig. 3).

To facilitate this verification, the rules are implemented into an algorithm designed to evaluate each possible state. For each identified type of relationship, a specific function is created to ensure that all rules of that type are satisfied. The algorithm systematically checks each state vector, and those that comply with all the established rules are compiled into a list of allowed states, ensuring that only permissible states are considered in the model of the system's operating states.

In our example, any of the states that begin with (1, 1, 1, ...), (1, 1, 0, ...), (1, 0, 1, ...) or (0, 1, 1, ...) will be considered as not permissible, as these combinations violate the derived Propositional Logic expression "(B_1 XOR B_2 XOR B_3)". The process is repeated until every parameter combination has been checked against every Propositional Logic expression.

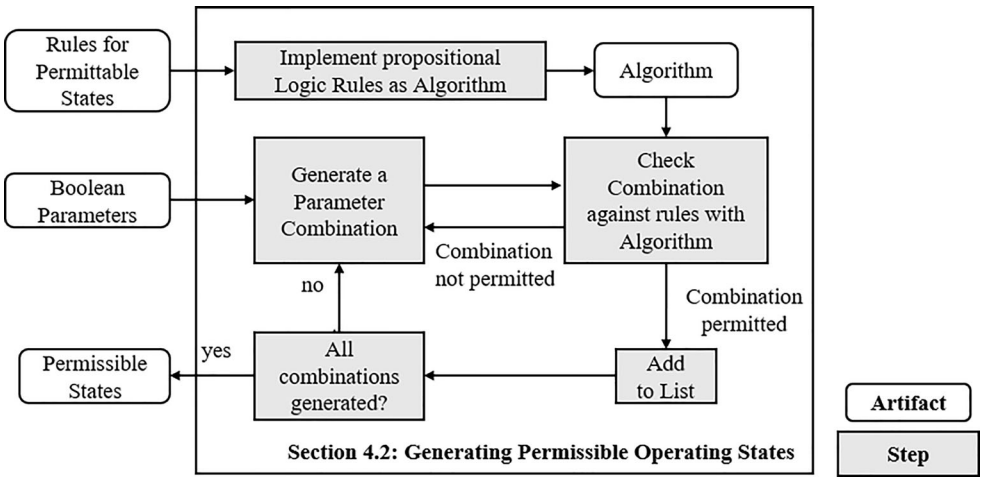

Fig. 3 Steps for generating the permissible operating states

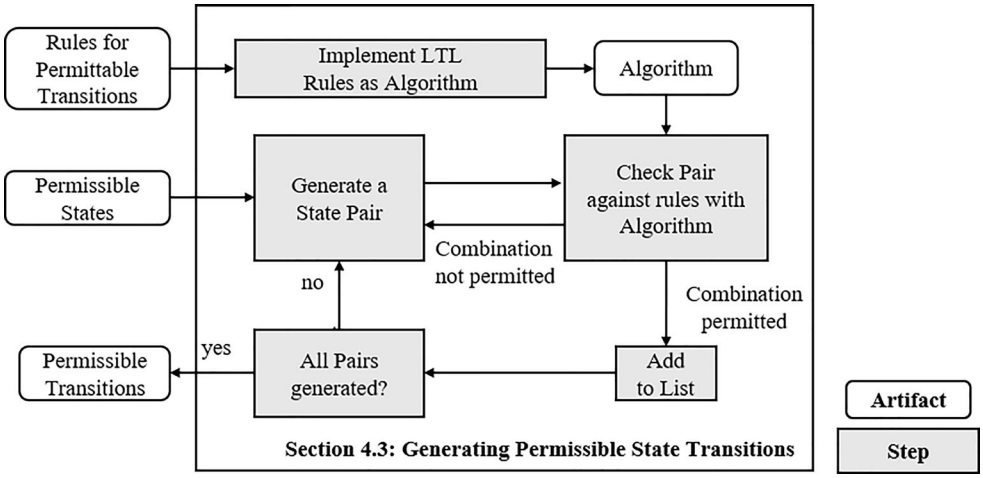

Fig. 4 Steps for generating the permissible state transitions

4.3 Generating permissible state transitions

The necessary steps for generating the permissible state transitions are shown in Fig. 4. In the generation process, state transitions are represented as pairs of allowed states, where the first state in the pair is the starting state and the second state is the target state. To describe all possible state transitions, the Cartesian product of the allowed states is taken, which systematically combines each starting state with every potential target state. These generated state pairs are then subjected to a verification process, where an algorithm checks each pair against the Linear Temporal Logic (LTL) rules that govern permissible transitions. These LTL rules are implemented within the algorithm to ensure that only those state pairs that fully comply with the specified rules are retained. The algorithm returns a list of state pairs that meet all the requirements, representing the valid transitions within the system.

This verified information can then be used to construct a state machine diagram that accurately reflects the allowed behavior of the system, providing a clear and precise model for further development and analysis. For smaller examples, this approach can be demonstrated to illustrate how the system's behavior is systematically captured and verified. The following Fig. 5 shows the implementation of the LTL expression introduced in Table 5 as a 'LTL rule' in the algorithm.

The LTL expression "G((B_1) → X!(B_3))" is implemented by evaluating two vectors 'V_1' and 'V_2' in terms of meeting a 'condition' and a 'consequence'. 'V_1' and 'V_2' consist of seven slots to store information about the

```
temporal_rules = [{
        # G((B_1) -> X!(B_3))
        #return true, if condition is met:
        "condition": lambda V_1, V_2: V_1[0] == 1,
        #return true, if consequence is met:
        "consequence": lambda V_1, V_2: V_2[2] == 0
        }]
```

Fig. 5 Implementation for the evaluation of the LTL expression "G((B_1) → X!(B_3))"

Boolean parameters as stated in Table 3. The Boolean parameter 'B_1' is stored in slot no. 1 and the parameter 'B_3' is stored in slot no. 3. In code, this translates to the array positions 'V_1[0]' and 'V_2[2]'. To meet the 'condition', slot 'V_1[0]' must be equal to '1'. In 'consequence', 'V_2[2]' must be equal to '0'. For every generated pair of vectors—representing a transition between two system states—the algorithm checks, if the implemented rules are met. If a generated pair is compliant to the rules, it is added to a list of valid state transitions. The algorithm loops through all possible combinations and finally returns a list of all valid state transitions.

4.4 Modeling of operating states and transitions in SysML v2

The formal graphical modeling of the generated operating states and transitions is achieved by utilizing the two lists of permissible states and state transitions to create a SysML v2 state machine. The resulting state machine based on the requirements presented in Table 2 is shown in Fig. 6.

This state machine incorporates the state-dependent system parameters and requirements, ensuring a comprehensive representation of the system's behavior. The entire system model is hosted on a server using the SysML v2 API services implementation provided by the Object Management Group (OMG) [15].

To facilitate this, the information is transformed into a JSON file structure, where each system element is assigned a unique identifier (ID) during the initial publishing to the server. These IDs are crucial for correctly interrelating the JSON objects that represent the various elements of the system. After the IDs are assigned, the detailed information necessary to fully specify the state machine is added to the elements. This structured information is then submitted to the server via API POST requests, ensuring that the model is accurately captured and accessible for further development and analysis within the SysML v2 framework.

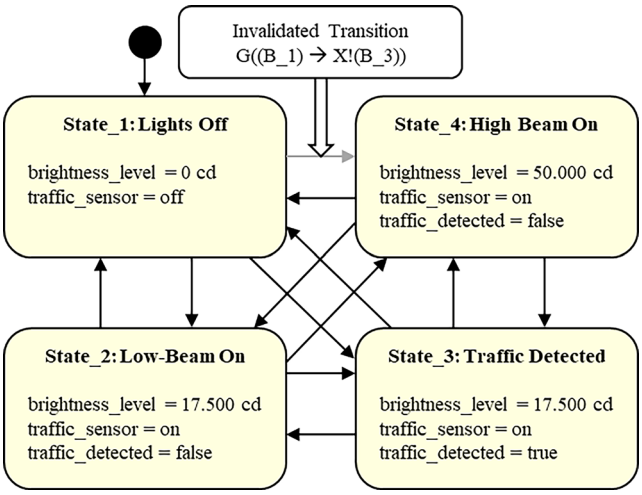

Fig. 6 State-machine based on the requirements of the exemplary system

5 Discussion, conclusion and future work

In this paper we presented a multi-step approach that enables the formal representation and verification of the operating states and state-transitions of a CPS in an MBSE compatible framework, i.e. a SysML v2 state machine diagram.

For a given set of rules specifying the operating states and their permitted transitions, a procedure was presented, in that all possible combinations were formed and checked for validity using algorithms. Currently, integrating the rules into the algorithms is not automated and thus requires manual effort. Moreover, as the number of state-dependent requirements increases, the effort to determine valid states and transitions grows. The potential number of states to be verified rises exponentially with the number of requirements, causing a corresponding increase in computation time. As a result, the current implementation only supports a limited number of state-dependent requirements. More efficient verification algorithms that do not rely on generating all states could address this issue.

Furthermore, as of now the formalizing of the requirements from natural-language to Propositional Logic and LTL has also been carried out manually. However, there have been efforts to automate such a formalization step in a similar context [18], so that a further automatization of this approach could be investigated in future work.

Automating the entire method could significantly decrease the modeling workload, thereby streamlining the adoption of model-based development methodologies.

Acknowledgements This research has been carried out as part of the research project "KIZAM" which is funded by the Federal Ministry for Economic Affairs and Climate Action (BMWK). On behalf of all authors, the corresponding author states that there is no conflict of interest.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Aceituna D, Do H Exposing the susceptibility of off-nominal behaviors in reactive system requirements. In: 2015 IEEE 23rd international requirements engineering conference (RE). IEEE, pp 136–145
2. Wymore AW, Wymore AW (1993) Model-based systems engineering: an introduction to the mathematical theory of discrete systems and to the tricotyledon theory of system design. Systems engineering series. CRC Press, Boca Raton, Fla
3. Mokos K, Katsaros P (2020) A survey on the formalisation of system requirements and their validation. Array 7:100030. https://doi.org/10.1016/j.array.2020.100030
4. Clarke EM, Henzinger TA, Veith H (2018) Introduction to model checking. In: Clarke EM, Henzinger TA, Veith H et al (eds) Handbook of model checking. Springer, Cham, pp 1–26
5. Pnueli A The temporal logic of programs. In: 18th annual symposium on foundations of computer science (sfcs 1977). IEEE, pp 46–57
6. Höpfner G, Jacobs G, Zerwas T et al (2021) Model-based design workflows for cyber-physical systems applied to an electric-mechanical coolant pump. IOP Conf Ser Mater Sci Eng 1097:12004. https://doi.org/10.1088/1757-899X/1097/1/012004
7. Zerwas T, Jacobs G, Spütz K et al (2021) Mechanical concept development using principle solution models. IOP Conf Ser Mater Sci Eng 1097:12001. https://doi.org/10.1088/1757-899X/1097/1/012001
8. Cherukuri H, Ferrari A, Spoletini P (2022) Towards explainable formal methods: from LTL to natural language with neural machine translation. In: Gervasi V, Vogelsang A (eds) Requirements engineering: foundation for software quality, vol 13216. Springer, Cham, pp 79–86
9. Brunello A, Montanari A, Reynolds M (2019) Synthesis of LTL formulas from natural language texts: state of the art and research directions. Schloss Dagstuhl – Leibniz-zentrum Inform Lipics. https://doi.org/10.4230/LIPIcs.TIME.2019.17
10. Baier C, Katoen J-P (2008) Principles of model checking. MIT Press, Cambridge, Mass
11. VDA (2024) Automotive VDA-Standardstruktur komponentenlastenheft: empfehlung zur spezifikation von systemen, software, modulen, komponenten und einzelteilen
12. Leveson NG (2004) Role of software in spacecraft accidents. J Spacecr Rockets 41:564–575. https://doi.org/10.2514/1.11950
13. Delligatti L (2014) SysML distilled: a brief guide to the systems modeling language. Addison-Wesley, Upper Saddle River, NJ, Munich
14. Zdanis L, Cloutier R The use of behavioral diagrams in SysML. In: 2007 IEEE long island systems, applications and technology conference. IEEE, p 1
15. OMG (2024) OMG systems modeling language
16. Salado A, Wach P (2019) Constructing true model-based requirements in SysML. Systems 7:19. https://doi.org/10.3390/systems7020019
17. Bender B (2022) Anforderungsengineering im kontext des IDE. In: Vajna S (ed) Integrated design engineering. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 659–688
18. Bertram V, Kausch H, Kusmenko E et al Leveraging natural language processing for a consistency checking toolchain of automotive requirements. In: 2023 IEEE 31st international requirements engineering conference (RE). IEEE, pp 212–222