Model-Based Systems Engineering of the Event-Driven Industrial Internet of Things

Amirali Amiri
Institute of Computer Engineering
TU Wien, Vienna, Austria
amirali.amiri@tuwien.ac.at

Max Thoma
Institute of Computer Engineering
TU Wien, Vienna, Austria
max.thoma@tuwien.ac.at

Gernot Steindl
Institute of Computer Engineering
TU Wien, Vienna, Austria
gernot.steindl@tuwien.ac.at

Christoph Klaassen
Institute of Energy Systems and Thermodynamics
TU Wien, Vienna, Austria
christoph.klaassen@tuwien.ac.at

Wolfgang Kastner
Institute of Computer Engineering
TU Wien, Vienna, Austria
wolfgang.kastner@tuwien.ac.at

***Abstract*—The Industrial Internet of Things (IIoT) is characterized by a multitude of standards, protocols, and tools. Moreover, the convergence of Information Technology (IT) and Operational Technology (OT) brings new architectural styles, e.g., event-driven communication, that are easier to scale and integrate IIoT devices. Architects need extensive expertise to manage the complex task of designing systems that encompass hardware, software, information, communication, and users. Proven approaches such as Model-Based Systems Engineering (MBSE) can simplify the design of IIoT systems by offering abstractions through system models and views. In this paper, we introduce an MBSE approach grounded in Systems Modeling Language (SysML) 2.0. Our method involves defining metadata tailored to event-driven IIoT systems. System designers can generate tagged model instances that undergo automatic validation against system requirements, such as asynchronous messaging, authentication, and health checks. Following validation, these models are utilized by an artifact generator to produce code, test cases, or documentation. Our approach is designed for reusability, and we provide tool support to streamline the implementation of requirements checking for emerging standards and guidelines. This enhances the flexibility and efficiency of IIoT system design, ensuring compliance with diverse and evolving industry requirements.**

***Index Terms*—IIoT, MBSE, Event-Driven Communication**

I. Introduction

Nowadays, the Industrial Internet of Things (IIoT) encompasses a vast array of standards, protocols, and tools. Consequently, architects must possess profound expertise to design such comprehensive systems. The traditional automation pyramid [8] can hinder scalability as components are tightly-coupled and linearly-integrated. The real-time requirements play a vital role in the domain of Operational Technology (OT). However, when IIoT devices are integrated using the Information Technology (IT), the even-driven communication [2] can be beneficial and provide loose coupling, e.g., using MQTT brokers [9]. The IT/OT convergence can result in contradicting requirements. This contradiction requires a framework that detects possible violations for specific requirements, e.g., communication schema or security with authenticity.

Nevertheless, there still exists a barrier to entry as IIoT system designers must learn different technologies, as well as how to implement and integrate them in their system. Established methodologies, such as Model-Based Systems Engineering (MBSE) [11], facilitate the design process of IIoT systems by offering abstractions through system models and views. Moreover, supporting tools can automate different steps of MBSE and make the approach easier to use. Thus, we set out to answer the following research questions:

RQ1: *How can MBSE assist the design of IIoT systems according to the event-driven communication standards and patterns, e.g., asynchronous messaging [12]?*

RQ2: *What is the architecture of a prototypical tool that facilitates the automatic requirements validation and generates artifacts, e.g., code, test cases, or documentation?*

In this paper, we introduce an MBSE approach [11] to facilitate the requirements validation of the event-driven IIoT systems. We introduce metadata describing these systems that are used as tags in Systems Modeling Language (SysML) 2.0[1] model instances. Our framework automatically converts the instances to graphs and saves them in a Neo4j database[2]. We perform graph-based validation of the requirements, and inform of any violations. Moreover, we present a prototypical tool support that facilitates using our concepts. We provide architectural details into our tool to demonstrate how system designers can extend and tailor our framework to their needs.

The structure of the papers is as follows. Section II presents the related work of our study. We give an overview of our proposed approach in Section III, and explain the details of our study in Section IV. Section V presents our accompanying

[1]https://www.omg.org/spec/SysML/2.0/Beta1
[2]https://neo4j.com

prototypical tool. Section VI presents a sample case and discusses our findings. Section VII concludes the paper.

II. Related Work

Vogel-Heuser et al. [14] argue that properties and environmental dependencies of industrial systems impact the overall performance of automated production systems. These properties, often in the form of characteristic curves, are only accessible from suppliers' documents, like operating instructions or online catalogs. They propose SysML profiles and create disciplinary views. A metamodel is used for a more precise presentation of the proposed reference mechanism. In an earlier work, Vogel-Heuser et al. [13] introduce a SysML-based approach for MBSE in manufacturing automation software projects. They adapt SysML to create SysML for Automation (SysML-AT), a specialized language profile that encompasses functional and non-functional requirements, software applications, and hardware properties. They developed a prototype tool that integrates adapted SysML parametric diagrams into an industrial automation software development environment, allowing debugging within the model.

De Saqui-Sannes et al. [4] examine the progression of Systems Engineering towards MBSE, highlighting the shift from document-centric methods to model-centric approaches for system development. While previous research has detailed the benefits and limitations of MBSE, this paper specifically aims to equip industry professionals with criteria for selecting MBSE languages, tools, and methods. It categorizes these elements beyond the commonly associated techniques like SysML and provides selection criteria. Fadhlillah et al. [5] argue that companies building Cyber-Physical Production Systems (CPPS) often create many variants, increasing engineering and maintenance efforts. So the systematic management of software variability would reduce these costs, but they offer limited support. They propose using a textual delta modeling approach from the Software Product Line (SPL) domain to manage IEC 61499-based control software. This approach can express software variability and provide a semi-automatic generator. Case studies and a user study indicate their potential benefits for implementing and maintaining control software.

Cederbladh et al. [3] indicate that there's been a shift from document-based to model-based development in systems engineering due to increased complexity and the need for digital workflows. They propose that MBSE is essential, providing early models for analysis and automated tasks. However, there's no common approach for early Verification and Validation (V&V) of system behavior in MBSE. They performed a systematic literature review with 149 of 701 relevant publications on early V&V in MBSE. Their findings show early V&V aims to ensure design quality before implementation, with SysML as the standard language. The authors conclude that V&V solutions vary, often targeting functional properties and being context-specific, with common issues in readiness, simplifications, and tool integration. Kharatyan et al. [7] highlight the growing integration of information and communication technologies in technical systems due to digitization, which offers benefits like autonomous driving but also presents development challenges. To tackle these challenges, MBSE approaches are used to manage the increasing complexity and interconnectivity of products. However, ensuring the reliability of future systems necessitates the early consideration of security aspects.

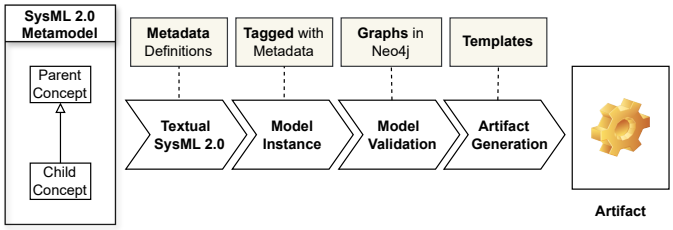

Fig. 1: The Workflow of Model-Based Systems Engineering

III. Approach Overview

This section introduces our MBSE approach. The MBSE workflow [11] is outlined in Fig. 1. Utilizing SysML 2.0's textual representation[1], we define metadata for the IIoT domain, which are then used as tags when creating a *Model Instance*. These instances are transformed into graphs and stored in a Neo4j database[2]. A *Model Validator* verifies that system requirements are met. Once validated, the models are sent to an *Artifact Generator* to produce artifacts such as code, test cases, or documentation. Fig. 2 outlines the high-level activities involved in our approach, with further details

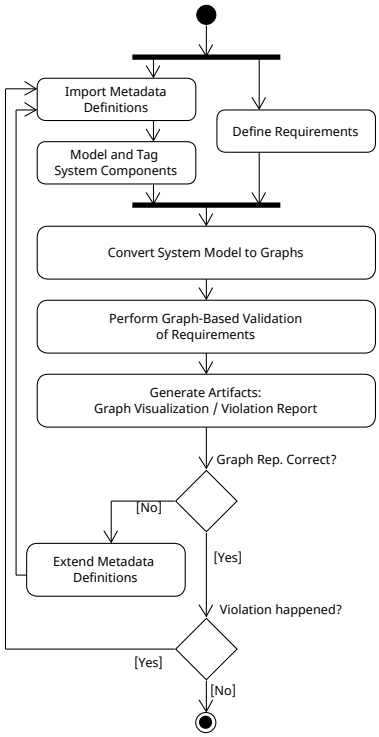

Fig. 2: High-Level Activities of Our Approach

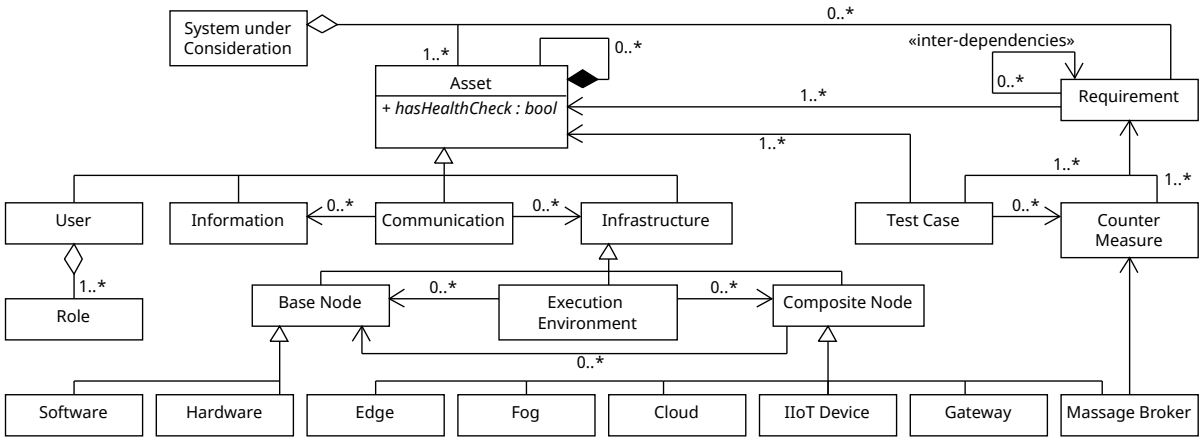

Fig. 3: Metadata of Event-Driven IIoT Systems

provided in the subsequent section. The only additional task for architects or system designers is to tag system components with our predefined metadata definitions in SysML 2.0. Our methodology automatically converts the system model to graph representations, validates requirements using graph-based methods, creates a graph visualization, and highlights any system components that violate requirements.

IV. Approach Details

Our metadata definitions for event-driven IIoT systems are presented in Fig. 3. A *System under Consideration* includes at least one *Asset*, which can be categorized as *Information*, *Communication*, *User* with various *Roles*, or *Infrastructure*. A *Base Node* represents *Software* and *Hardware*. A *Composite Node*, composed of base nodes, can be an *Edge*, *Fog*, *Cloud*, *Message Broker*, *Device Gateway*, or *IIoT Device*. An *Execution Environment* models either virtual or physical environments, such as virtual machines, containers, or bare-metal servers. A *Requirement* encapsulates system requirements, while *Counter Measures* are actions taken to meet these requirements. For example, message brokers allow the asynchronous messaging as a counter measure to the requirements of event-driven communication and loose coupling of system components. *Test Cases* evaluate the effectiveness of these counter measures. Note that architects can easily extend our metadata, e.g., by defining user roles as detailed in this section. Listing 1 provides a snippet of our SysML 2.0 metadata.

Graph Representation of Model Instances System designers simply need to import our metadata definitions into a SysML 2.0 textual representation and tag their system components accordingly. Once the system components are tagged, we convert the system model into a graph. This graph representation reflects the components-and-connectors view of the system [1]. Section VI offers an illustrative sample case.

Graph-Based Validation of Requirements We traverse the graph representation of model instances for requirements validation. To validate the requirement regarding asynchronous messaging, we check all paths of the converted system graph to make sure that all communications go through message brokers. Our approach indicates a violation of system requirements, if there exists a path that the requests can be passed between system parts without going through a message broker. Algorithm 1 presents our validation of requirements.

```
package Meta_EventDrivenIIoT {
    enum def UserRole { enum Admin; }

    metadata def Asset {
        import ScalarValues::*;
        attribute hasHealthCheck : boolean;}

    metadata def User :> Asset {
        import Meta_EventDrivenIIoT::UserRole::*;
        attribute Role : UserRole;}

    metadata def Infrastructure :> Asset;
    metadata def BaseNode :> Infrastructure;
    metadata def CompositeNode :> Infrastructure;
    metadata def Edge :> CompositeNode;
    metadata def Fog :> CompositeNode;
    metadata def Cloud :> CompositeNode;
    metadata def IIoTDevice :> CompositeNode;
    metadata def Gateway :> CompositeNode;
    metadata def MessageBroker :> CompositeNode;
        ...
}
```

Listing 1: Excerpt of Metadata Definitions in SysML 2.0

Algorithm 1: Graph-Based Requirement Validation: *Communication must be asynchronous.*

Input: $G \leftarrow Graph(SystemModel)$
$violation \leftarrow$ false
foreach *path in G* **do**
  if *path does not include a MessageBroker* **then**
    $violation \leftarrow$ true
    break
  end
end
return $violation$

We also study system health checks using, e.g., the Heartbeat pattern [6] or the Health Check API pattern [10]. In these patterns, the system ensures the availability of a system

Algorithm 2: Graph-Based Requirement Validation: *Health checks must be available.*

```
Input: G ← Graph(SystemModel)
function stop_function_exists(G)
begin
    HealthNodes ← find_health_nodes(G)
    MessageBrokers ← find_message_brokers(G)
    violation_array ← []
    foreach node in HealthNodes do
        foreach broker in MessageBrokers do
            if direct path exists from broker to node then
                violation_array[component] ← false
                break
            else
                violation_array[component] ← true
            end
        end
    end
    return violation_array
end
violation_array ← stop_function_exists(G)
violation ← false
foreach element in violation_array do
    if element == true then
        violation ← true
        break
    end
end
return violation
```

component, either by periodically checking in case of the Heartbeat pattern, or by communicating with an endpoint in case of the Health Check API. We check that the health check is available for system components and that there is a direct path, i.e., path of length one, from message brokers to system parts. Algorithm 2 presents our validation of health checks.

Moreover, the Application Programming Interface (API) Gateway pattern[3] provides a single entry point to the system. The gateway usually authenticates user requests to ensure security as also mandated by the IEC 62443-3-3 standard. Similarly to Algorithm 2, we check that all requests coming from users go through a directed path to gateways. To avoid repetition, this algorithm is not shown. However, Section V gives an implementation of this requirements validation.

Artifact Generation After validating the models, our approach generates a visualization of the system graph along with information about any requirement violations. This step allows system architects to verify that the converted graph accurately represents the system model, ensuring the information provided is correct. Our approach identifies the specific system parts that violate safety and security requirements. If the graph is found to be inaccurate, architects or system designers can re-tag their system components and rerun the process. Additionally, the metadata definitions can be extended to better meet the needs of different systems.

[3]https://microservices.io/patterns/apigateway.html

V. Tool Support

Tool Architecture We provide a prototypical tool to demonstrate our approach, available in the online artifact of our study[4]. Fig. 4 shows the tool architecture. The frontend is implemented in React[5] and the backend is developed as a RESTful API[6]. We provide a Dockerized Jupyter notebook[7] to model the system under consideration in SysML 2.0. Note that any editor can be used instead. The system model is converted into multiple JSON files and loaded into a Neo4j graph database. We query the graph database using Cypher[2]. Finally, our algorithms in Section IV check the results of the queries for requirements validations. The *Artifact Generator* provides a visualization of the converted graph and gives information on the exact parts that violate the requirements.

Implementation Details Listing 2 shows the partial implementation of Algorithm 1 in the Cypher language. We check that all paths go through message brokers, excluding the paths with gateways. The *Model Validator* and *Artifact Generator* of our tool perform additional processing on the returned values of the Cypher query.

```
MATCH paths=(root) -[*0..]-> (node) -[*0..]-> (leaf)
  WHERE length(paths) > 0 AND
  ALL (node IN nodes(paths)
    WHERE node.tag <> 'MessageBroker' ) AND

  ALL (node IN nodes(paths)
    WHERE node.tag <> 'Gateway' )
return distinct paths
```

Listing 2: Requirement Checking: Asynchronous Messaging

Algorithm 2 is partially implemented in the Cypher language as shown by Listing 3. We check that heath checks exist from brokers to system assets with a path length of one.

```
MATCH (HealthNodes:Asset)
  WHERE HealthNodes.attribute = 'true'
MATCH (Brokers) WHERE Brokers.tag = 'MessageBroker'

return distinct HealthNodes, exists((Brokers)
    -[*1]->(HealthNodes)) AS HealthCheckExists
```

Listing 3: Requirement Checking: Health Check

The implementation of the requirements checking regarding gateways is shown by Listing 4. As mentioned, API gateways usually provide authentication as well to ensure security. We check that all user requests go through gateways as the entry point of the system with a direct path of length one.

```
MATCH (Users) WHERE Users.tag = 'User'
MATCH (Gateways) WHERE Gateways.tag = 'Gateway'

return distinct Users, exists((Users)-[*1]->(
    Gateways)) AS GatewayExist
```

Listing 4: Requirement Checking: Gateway

Extension Capabilities We designed our approach to be easily extensible. System architects can tailor our tool to automatically check for requirements, standards and guidelines.

[4]https://zenodo.org/records/11415428 DOI:10.5281/zenodo.11415428
[5]https://reactjs.org
[6]https://restfulapi.net
[7]https://jupyter.org

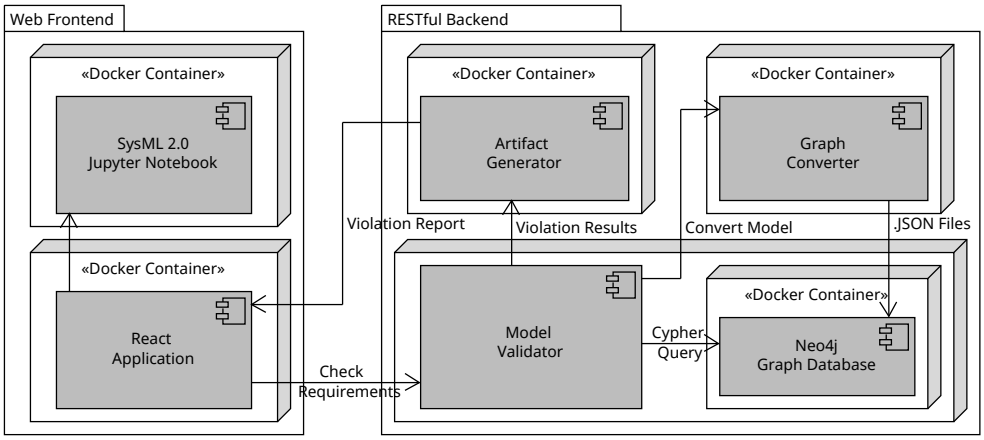

Fig. 4: Tool Architecture Diagram

For example, it is possible to extend our metadata definitions to include the role *OTExpert* and *ITExpert* as shown by Listing 5. It is also possible to define more counter measures, and adding the corresponding metadata definitions. Architects can write new algorithms for requirements validation and implement them using the Cypher queries. For example, queries can check for more specific security-authentication requirements, or security levels for different system zones.

```
package Meta_EventDrivenIIoT {
    enum def UserRole {
      enum Admin;
        enum OTExpert;
        enum ITExpert;
    ...
}}
```

Listing 5: Extended Metadata with User Roles

VI. Discussion

We provide a sample case, and discuss our results.

Illustrative sample case We model a system with a central broker in SysML 2.0, where edge, fog, and cloud services are modeled. Listing 6 shows the tagged model instance, and Fig. 5 presents the converted graph of the sample case.

```
package EventDrivenIIoT {
    import Meta_EventDrivenIIoT::*;

    part MQTT  {@MessageBroker;}
    part CloudService {@Cloud;}
    part FogAnalytics {@Fog {hasHealthCheck=true;}}
    part SCADA {@Edge;}
    part DeviceGateway {@Gateway;}
    part Device {@IIoTDevice;}
    part Admin {@User {Role=Admin;}}

    connect MQTT to SCADA;
    connect MQTT to CloudService;
    connect Device to DeviceGateway;
    connect DeviceGateway to MQTT;
    connect SCADA to FogAnalytics;
    connect FogAnalytics to MQTT;
    connect CloudService to MQTT;
    connect Admin to SCADA;
}
```

Listing 6: SysML 2.0 Model of the Sample Case

Fig. 5: Graph Representation of the Sample Case

Results Fig. 6 shows the results of our framework regarding the sample case. Our approach informs about the paths that violate the asynchronous messaging. Also, system parts violating the health-check and gateway patterns are highlighted.

Failed the requirements of the asynchronous messaging:
Admin -> SCADA , SCADA -> Device
Admin -> SCADA -> Device
Failed the requirements of the health-check pattern:
FogAnalytics
Failed the requirements of the gateway pattern:
Admin

Fig. 6: Results regarding Failed Requirements

Following the information regarding asynchronous messaging, we update the system model event-driven communications. Fig. 7 shows our updated design, and Fig. 8 shows the results of our framework regarding the updated graph.

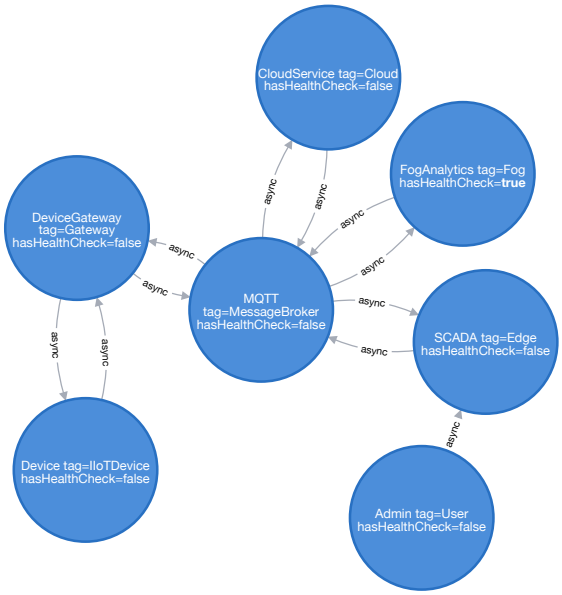

Fig. 7: Updated Graph of the Sample Case

Failed the requirements of the asynchronous messaging:
Admin -> SCADA
Passed the requirements of the health-check pattern.
Failed the requirements of the gateway pattern:
Admin

Fig. 8: Results regarding the Updated Graph

Finally, we add an API gateway as the system entry point to pass all requirements as shown by Fig. 9. The results of our framework is shown in Fig. 10 passing all requirements. The recommendations of our approach helps the architects to update the system design early in the process.

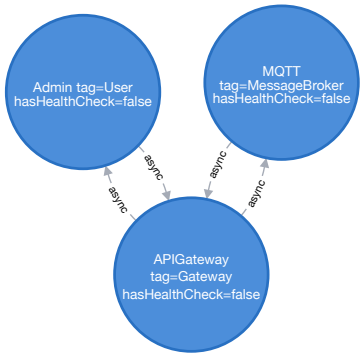

Fig. 9: Adding an API Gateway to the Sample Case

Passed the requirements of the asynchronous messaging.
Passed the requirements of the health-check pattern.
Passed the requirements of the gateway pattern.

Fig. 10: Results regarding the Passed Requirements

VII. Conclusions

In this paper, we set out to answer the research questions how MBSE can assist the design of IIoT systems according to the event-driven communication standards and patterns, e.g., asynchronous messaging [12] (**RQ1**), and what the architecture of a prototypical is tool that facilitates the automatic requirements validation and generates artifacts, e.g., code, test cases, or documentation (**RQ2**). We proposed a MBSE approach focusing on the event-driven IIoT systems to answer **RQ1**. We defined metadata in SysML 2.0 and provided support to automatically validate requirements based on the converted system graphs. Regarding **RQ2**, we demonstrated support for the basic functionality of standards and patterns as a proof-of-concept. As system designers utilize our framework, we expand our approach and develop a knowledge base of guidelines specific to the domain of event-driven IIoT. Our approach finds violation of requirements early at the design-time, and gives architects recommendation to update the system design.

We plan to evaluate our framework in real-world scenarios by generating code based on the standard IEC 61131-3:2013. Moreover, we apply our concepts in an experimental setting, where we measure empirical data regarding the runtime of our generated artifacts. Having done so, we plan to study automatic adaptation of system design. The self-adaptation provides feedback from the runtime to the design-time of industrial systems to improve the quality of service.

References

[1] V. Bertram, S. Maoz, J. O. Ringert, B. Rumpe, and M. von Wenckstern. Component and connector views in practice: An experience report. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 167–177, 2017.

[2] H. Cabane and K. Farias. On the impact of event-driven architecture on performance: An exploratory study. *Future Generation Computer Systems*, 153:52–69, 2024.

[3] J. Cederbladh, A. Cicchetti, and J. Suryadevara. Early validation and verification of system behaviour in model-based systems engineering: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.*, 33(3), mar 2024.

[4] P. De Saqui-Sannes, R. A. Vingerhoeds, C. Garion, and X. Thirioux. A Taxonomy of MBSE Approaches by Languages, Tools and Methods. *IEEE Access*, 10:120936–120950, 2022.

[5] H. S. Fadhlillah, S. Sharma, A. M. Gutierrez Fernandez, R. Rabiser, and A. Zoitl. Delta modeling in iec 61499: Expressing control software variability in cyber-physical production systems. In *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, Sep. 2023.

[6] U. Joshi. *Patterns of distributed systems*. Addison-Wesley Professional, 2023.

[7] A. Kharatyan, J. Tekaat, S. Japs, H. Anacker, and R. Dumitrescu. Metamodel for safety and security integrated system architecture modeling. *Proceedings of the Design Society*, 1:2027–2036, 2021.

[8] C. Lucizano, A. A. de Andrade, J. F. Blumetti Facó, and A. G. de Freitas. Revisiting the automation pyramid for the industry 4.0. In *2023 15th IEEE International Conference on Industry Applications (INDUSCON)*, pages 1195–1198, 2023.

[9] S. Mirampalli, R. Wankar, and S. N. Srirama. Evaluating nifi and mqtt based serverless data pipelines in fog computing environments. *Future Generation Computer Systems*, 150:341–353, 2024.

[10] P. Raj, A. Raman, and H. Subramanian. *Architectural Patterns: Uncover essential patterns in the most indispensable realm*. Packt Publishing, December 2017.

[11] A. L. Ramos, J. V. Ferreira, and J. Barceló. Model-based systems engineering: An emerging approach for modern systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 42(1):101–111, 2012.

[12] C. Richardson. Microservice architecture patterns and best practices. *http://microservices.io/index.html*, 2019.

[13] B. Vogel-Heuser, D. Schütz, T. Frank, and C. Legat. Model-driven engineering of Manufacturing Automation Software Projects – A SysML-based approach. *Mechatronics*, 24(7):883–897, 2014.

[14] B. Vogel-Heuser, M. Zhang, B. Lahrsen, S. Landler, M. Otto, K. Stahl, and M. Zimmermann. Sysml' – incorporating component properties in early design phases of automated production systems. *at - Automatisierungstechnik*, 72(1):59–72, 2024.