Mechanisms for Model Consistency: A Comparative Analysis of Guideline Implementation in SysML v1 and SysML v2

1st Nassim Awabdy
Fachbereich 5
Aachen University of Applied Sciences
Aachen, Germany
nassim.awabdy@alumni.fh-aachen.de

***Abstract*—The increasing complexity of Cyber-Physical Systems (CPS) is managed by rigorous Model-Based Systems Engineering (MBSE) practices that ensure architectural and technical consistency. This study compares guideline implementation mechanisms in SysML v1 and SysML v2, exploring their specific impact on automated verification. Our comparative analysis reveals a fundamental shift in paradigm between the two versions. SysML v1, relies on extrinsic mechanisms, specifically Profiles and the Object Constraint Language (OCL), which necessitate transforming models into external formalisms, often introducing semantic gaps and inconsistencies. In contrast, SysML v2's foundation in Kernel Modeling Language (KerML) allows for intrinsic implementation and enforcement of modeling guidelines. The utilization of Metadata Definitions and the separation of Definition and Usage, allows guidelines to be inherited as native logical assertions. Consequently, the verification process evolves from model transformation to mathematical reformulation, enabling models to be solved directly as Constraint Satisfaction Problems (CSPs). This formalization resolves the semantic ambiguities found in v1, but shifts the burden of consistency from the accuracy of manual mapping to the computational management of NP-hard solution spaces.**

***Index Terms*—cyber-physical systems, model-based systems engineering, sysml, modeling guidelines, model validation, model verification, systems engineering**

I. Introduction

Modern Cyber-Physical Systems (CPS) are technical systems that combine mechanical, electronic, and software subsystems with physical elements embedded in the real world. The development of CPSs is becoming increasingly complex and challenging, due to their interdisciplinary nature and the need to ensure seamless integration between their physical and computational components [2], [3].

Model-Based Systems Engineering (MBSE) is a methodology for the development and management of such complex systems, that addresses issues arising from the complexity and interdisciplinary nature of CPS, and provides the agility required to adapt to changing requirements and technologies. MBSE incorporates a centralized system model as the primary source of information, throughout the system lifecycle [2]–[4], [6].

SysML v1 has been widely adopted as the standard for modelling CPS and served as a key enabler for MBSE. SysML v1 is a graphical, general purpose modelling language that is defined as an extension of the Unified Modeling Language (UML). Because it was built on top of UML, SysML v1 inherited several limitations from UML that limited its expressiveness and usability for CPS modelling. However, it still provided a solid foundation for specifying and analyzing a systems's behavior, structure and requirements [3], [4], [6].

The release of SysML v2 represents the next generation of the Systems Modeling Language, designed as an overhaul of SysML v1 that addresses its limitations and enhances the efficacy of MBSE practices. Unlike its predecessor, SysML v2 is built upon the Kernel Modeling Language (KerML), this approach ensures that SysML v2 inherits a formal semantic foundation that is crucial for enhanced precision and automation in MBSE workflows. [3], [4], [6], [11].

While these advancements introduced by SysML v2 are promising, the abstract nature of the language still presents challenges for ensuring consistent modeling practices across diverse engineering teams. Model inconsistencies create a high risk of redundant effort, potential modeling errors, and lack of reuse of system elements, preventing them from being aggregated into a coherent overall system model. Therefore, mechanisms for implementing and enforcing modeling guidelines must evolve to leverage the native formal capabilities of SysML v2, that enable effective model verification processes throughout the development lifecycle [2]–[4], [11], [13].

Within the context of model analysis, a distinct differentiation between model validation and model verification is necessary. Following the ISO 15288 standard, this work adopts the following definitions:

- *Verification* is the process of proving that a design solution conforms to defined architectural and technical standards. This includes requirements for *traceability* and *syntactic verification* [5].
- *Validation*, conversely, aims to prove that the system fulfills its business objectives and stakeholder requirements in its intended operational environment. This is achieved through *behavioral simulation* and the execution of *operational scenarios* [5].

This work addresses the following research question: **How do the mechanisms for implementing modeling guidelines differ between SysML v1 and SysML v2, and how do these differences impact the capabilities of automated verification?**

To answer this, we conduct a comparative analysis of the underlying implementation mechanisms, specifically contrasting the profile-based constraints of SysML v1 with the metamodel-driven and formal semantic capabilities of SysML v2. We analyze the implications of this shift by examining specific scenarios where these mechanisms facilitate structural verification, rather than providing a holistic overview of all available commercial tools.

II. Theoretical Background

A. SysML v1 Foundations

SysML v1 is a graphical, general purpose modeling language that is widely recognized as the standard language for MBSE, serving as a foundational tool for specifying, analyzing, designing and verifying complex multidisciplinary systems [1], [4], [8].

SysML v1, adopted by the Object Management Group (OMG) in 2007, was essential in advancing MBSE practice by providing capabilities for formally capturing system requirements, structure, behavior, and parametric [6]. Since the language is defined as an extension of the Unified Modeling Language (UML), it allows it to adopt established modeling concepts [4].

SysML v1's modeling constructs are organized in four main categories:

- **Structure**: This structure is modeled using Block Definition Diagrams (BDDs) and Internal Block Diagrams (IBDs). BBDs are used to define components, interfaces, and relationships at black-box level and IBDs offer a white-box perspective by outlining the internal structure of a single block [8].
- **Behavior**: SysML v1 provides several behavioral diagrams, including State Machine Diagrams, Sequence Diagrams, and Activity Diagrams [8]. These diagrams capture distinct uses for modeling system behavior.
- **Requirements**: System requirements are specified in Requirement Diagrams, that largely rely on natural language representation, although they can be linked logically to other model elements [8], [11].
- **Parametric**: Parametric Diagrams define mathematical constraints and equations between system elements that support fundamental calculations and analysis within the model. For complex evaluations they often require external tools [8].

The restriction of modeling practices in SysML v1 is achieved through UML profiling mechanism that enable the construction of specialized extensions. The main element for customization is the *stereotype*, which functions as a distinct metaclass within UML [1]. Stereotypes enable the customization of existing metaclasses by associating them with specific properties and constraints, thereby tailoring the modeling language to meet domain-specific requirements [1].

Object Constraint Language (OCL) is an expression language that enables formally defining rules, invariants and constraints on model elements that must be satisfied for a model to be considered valid [13]. It identifies what a valid state is but does not provide the mechanism to calculate variables to reach that state.

B. SysML v2 Foundations

SysML v2 represents a major evolution over its predecessor, having been engineered independently from UML to overcome the limitations inherited from it [4], [6]. It aims to enhance MBSE adoption and effectiveness by focusing on improving precision, expressiveness, consistency, usability, interoperability, and extensibility. [6] The foundation of SysML v2 is built upon a new general-purpose modeling language called the *Kernel Modeling Language (KerML)* [11].

The mechanism of KerML is built upon a hierarchical, three-layered architecture, successively progressing from general to specific constructs [12].

- The **Root Layer** establishes the essential syntactic scaffolding for constructing models. The main focus of this layer is to define organizational constructs, such as *Elements*, *Namespaces*, and *Relationships*, leaving out model-level semantic interpretation relative to the modeled system [11], [12].
- The **Core Layer** introduces the language's semantic foundation based on classification, defined in first-order logic axioms. The central primitive is *Type*, which is divided into *Classifiers*, *Features*. It also defines key relationships necessary for organizing classification hierarchies [11], [12].
- The **Kernel Layer** finalizes the language specification by adding specialized constructs used in common modeling applications, such as *Data Types*, *Classes*, *Structures*, and *Behaviors* [11], [12].

KerML achieves its consistent semantics through formal mathematical logic and library-based ontological modeling that maintain a precise interpretation of complex models. Since the semantics in the *Core Layer* are defined using first-order logic, a consistent basis for mathematical reasoning about models is established [11], [12].

For comprehensive concepts introduced in the *Kernel Layer*, KerML extends its semantics through the reuse of elements found in the *Kernel Semantic Library*. This library is itself expressed in KerML, meaning that all concepts in the language are ultimately grounded in the same formal semantic framework [11], [12].

SysML v2 introduces several key features for further enhancing modeling capabilities, accessibility, and tool integration.

- **Textual Notations**: In addition to graphical notation, SysML v2 features a standardized textual syntax that provides advantages for interoperability with external tools and exchange of models [11], [12].

- **Standardized API**: The language includes the new Systems Modeling API (SysML API), which enables full access to the model and general Model as Code workflows [11], [12].
- **Metadata Definitions**: In SysML v2, *Metadata Definitions* (`metadata def`) serve as the primary mechanism for annotating model elements with domain-specific semantics, analogous to Stereotypes in SysML v1. They define a specific schema (properties and constraints) that extends the language's metaclasses, enabling the creation of domain-specific modeling languages (DSMLs) [3], [11].
- **Constraints**: KerML provides the foundation for defining constraints through constructs such as *assert*, *invariant*, *requirement*. These KerML constructs enable the model to be treated as a mathematical system that can be analyzed and solved directly [13].
- **Cases**: SysML v2 introduced a generic *Case* construct which is essentially a calculation that can declare a subject and an objective to provide a formal and executable way to check model correctness and evaluate system properties. Two specialized cases are provided: *Analysis Case* (Quantitative Analysis) and *Verification Case* (Qualitative Analysis), further enhancing model analysis capabilities [11].

C. Computational Solvers

Computational Solvers in MBSE are engines that automate the search for solutions within a "design space". By processing a set of declarative constraints (mathematical and logical) they identify valid configurations, verify that requirements are met, and optimize system performance. The utility and implementation of solvers differ fundamentally between SysML v1 and SysML v2, due to the fundamental shift in language architecture.

III. Mechanisms for Guideline Implementation

This section presents structural mechanisms used to define and enforce domain-specific modeling guidelines for both SysML v1 and SysML v2. For each version, we outline known methods of implementation and exemplify their application.

A. SysML v1: Profile-Based Implementation

In SysML v1, the implementation of domain-specific guidelines is achieved through a multiple layer approach. First, system engineers utilize the *Stereotypes* to build a custom profile that overlays the native SysML v1 metamodel, thereby introducing semantic labeling and domain specific terminology [1]. The application of SysML v1 elements is then restricted to a specific palette of stereotyped elements, ensuring that the model structure reflects domain-specific semantics rather than generic block definitions [1].

This mechanism is illustrated by Beers et al. [1] in the development of a Domain-specific Modeling Language (DSML) for formal process description. To enforce the guideline that system functions must be standardized, the author extends the metaclass *CallBehaviorAction* to create a specific *Process Operator* stereotype. This ensures that an element cannot be designated as a *Process Operator* unless it inherits the specific meta-attributes defined by the stereotype.

While stereotypes provide semantic labeling, UML profiles are not expressive enough to represent constraints on the models [10]. Therefore, *Object Constraint Language (OCL)* is layered onto the profile to specify invariants, constraints and complex relationships between elements [1], [10]. Beers et al. [1] demonstrates this when enforcing the VDI/VDE 3682 standard that provides the rules to implement the "Product-Process-Resource" (PPR) concept, which mandates that production systems are modeled through the strict interconnection of the *Process*, the *Product*, and the *Resource* [1]. In their implementation, a "State" (e.g., a Product or Energy) cannot legally connect directly to another "State" without an intermediary process. Since standard SysML v1 syntax permits the connection of any two compatible nodes, the guideline is enforced through an OCL invariant attached to the *Flow* stereotype, that prevents two state-describing elements from being connected together [1].

B. SysML v2: Metamodel Definitions

SysML v2 implements guidelines intrinsically by the extension of the language's ontology directly using KerML (Kernel Modeling Language). Thereby, removing the separation between the model and the rules found in v1.

Metadata Definitions is used to construct domain-specific metamodel hierarchies directly within the language architecture by specializing standard KerML constructs [3].

1) Metamodel-Driven Guidelines: Boelsen et al. [3] demonstrates this by implementing modeling guidelines for reusable mechanical system elements based on the *motego* methodology. By defining an abstract metadata definition for a general *solution* the structural foundation is achieved that is set up as a specialization of the standard *SysML::PartDefinition* (see Fig. 1). From this abstract base, concrete domain-specific types *SolutionElement* and *SystemSolution* are derived [3].

Specific structural and behavioral components are required by the guideline. The *ActiveSurface* and *ActiveSurfaceSet* are defined as specializations of *SysML::PartDefinition* to represent geometrical structure, while the *Material* is integrated as a reusable structural part within these surfaces.

The behavioral logic is enforced by defining the *PhysicalEffect* not as a block, but as a specialization of the *SysML::ConstraintDefinition* meta type. Corresponding metadata definitions are also derived for *PartUsage* to enable the instantiation of these elements within the system model [3].

The enforcement of these guidelines is demonstrated in the modeling of a specific component, such as a "Lubricated Mechanical Line Rolling Contact". The component is instantiated using the *#SolutionElementDef* metadata definition (see Fig. 2). The use of this definition guides the inclusion of the required sub-elements defined in the meta model, specifically the *ActiveSurface* and *PhysicalEffect* [3]. Within this structure, the *ActiveSurface* integrates essential parameters via attribute

```
metadata def SolutionElementDef :> PartDefinition;
metadata def ActiveSurfaceSetDef :> PartDefinition;
metadata def ActiveSurfaceDef :> PartDefinition;

metadata def PhyEffectDef :> ConstraintDefinition;

metadata def SolutionElement :> PartUsage;
metadata def ActiveSurfaceSet :> PartUsage;
metadata def ActiveSurface :> PartUsage;
metadata def PhyEffect :> ConstraintUsage;

metadata def Material :> PartUsage;
```

Listing 1. Definition of the domain-specific metamodel for mechanical elements in SysML v2 (adapted from Boelsen et al. [3])

definition and references the material using the *#Material* command [3]. Simultaneously, the *PhysicalEffect* is completed by adding input and output attributes and defining the calculation specification as expression, thereby ensuring that the relationship between the physical effect and the functional flows are strictly quantified accordingly [3].

```
#SolutionElementDef def LubMechLineRollingContact {
    #PhyEffect pe1 : SurfacePressure;
    #PhyEffect pe2 : CurvedCurvedKinematics {
        in omega : Real;
        out v_out : Real = r * omega;
    }

    #ActiveSurfaceSet ass : CylindricalLatSurfaces {
        #ActiveSurface as1 : CylindricLatSurface {
                attribute radius : Real;
        }
        #ActiveSurface as2 : CylindricLatSurface {
            #Material mat : Steel;
        }
    }

    bind pe2.radius = ass.as1.radius;
}
```

Listing 2. Implementation of a "Lubricated Mechanical Line Rolling Contact" using the enforced metadata definitions (adapted from Boelsen et al. [3])

2) Usage Definition Separation: In SysML v2, the language distinguishes between the *definition* of an element (Its "Type") and the *usage* of an element (Its "Instance") [3], [6].

- **Definition** defines the reusable template, including features, attributes, and constraints. This acts as the "Library" element [6].
- **Usage** represents the occurrence of that definition within the systems. It inherits features for the definition but can redefine them to adapt to the specific context [3], [6].

Boelsen et al. leverages this to enforce a standardized structure for mechanical system elements, to ensure that engineers cannot deviate from the required structure. Using this approach Boelsen et al. was able to enforce guidelines in two ways.

- **Structural Inheritance**: When an engineer uses a library element, they create a *Usage* defined by the *Definition*. Because the usage inherits from the *Definition*, it automatically includes all required internal structures (e.g., *Physical Effect*) mandated by the guideline [3].
- **Restricted Customization**: the guideline can dictate that the internal structure is defined once in the library (*Definition*). In the system model (*Usage*), the engineer interacts with the exposed parameters (e.g., input/output flows) but relies on the validated internal logic of the *Definition*.

Overall, this separation allows the guidelines to treat the *Definition* as the "Single Point of Truth" (SPOT) stored in libraries. The *Usage* are merely pointers to these definitions. Thereby, preventing redundant modeling efforts and diverse formalization.

IV. Mechanisms for Model Verification

A. SysML v1: Constraint Checking via Profiles and Model Transformation

The native verification capabilities of SysML v1 are inherently limited due to its UML-based architecture. While stereotypes provide a mechanism for structural consistency by restricting permissible element types, they lack the foundations required for formal verification. [1].

As already stated, OCL is layered onto profiles to define formal constraints and requirements that can be evaluated against the model. Native UML/SysML v1 tools are limited to verifying syntactic correctness, rather than the semantic validity of the model [10].

To bridge this gap, external solvers are required to perform *semantic verification*. However, since these solvers cannot directly interpret SysML v1 models, the diagrams must be transformed into ontology-based representations or compatible formats (e.g., OWL, SMT-LIB) [9], [10].

This verification workflow is exemplified by Lu et al., where they present a Cloud Agility Baseline (CAB) model, representing a logistics system comprised of *Shipment*, *Dispatcher*, and *Transporter* blocks, governed by specific state invariants expressed in OCL [10]. They initially show how the native SysML v1 tool successfully verifies a model containing significant semantic contradictions. Their approach involved mapping the SysML blocks and OCL constraints into Web Ontology Language (OWL) DL axioms. Only through the external application of an OWL inference engine (Pellet) could the logical inconsistency be identified, revealing that the *Shipment* class was unsatisfiable (equivalent to Nothing).

B. SysML v2: Formal Verification via Constraint Satisfaction

In contrast to SysML v1, where verification requires model transformations to external formalisms (e.g., OWL) to check logical consistency, SysML v2 native constraints are built upon KerML, which is founded on first-order logic [11]. This intrinsic formalism allows SysML v2 models to be directly interpreted as mathematical systems that can be analyzed and solved using specialized solvers [11], [13].

By encapsulating verification objectives and constraints directly within the model, the language ensures that the criteria for model correctness are traceable and independent of specific external tools. [11]. This capability supports the Verification process by allowing models to be reformulated into Constraint Satisfaction Problems (CSPs) that can be solved using specialized solvers [11], [13].

As described by Ratzke et al. this process begins with the compilation of the model into a KerML abstract representation. To render the model solvable, first the abstract syntax must represent specific instances rather than generic types. This involves cloning features from general definitions to specialized usages and utilizing KerML binding relationships to identify set-intersections between variables and their constraints. The result is a mathematically rigorous CSP consisting of a finite set of variables and their respective domains (Boolean, Integer, or Real), which can then be processed by specialized solvers to enforce logical consistency [13].

Ratzke et al. [13] demonstrate the application of this workflow by introducing range-based semantics to verify the system's variability and precision. In their implementation the semantic library is extended to include three range-based constraint operators:

- **oneOf**: This semantic assigns exactly one value from a specified range that satisfies the constraint, which can be used to represent variation (e.g., choosing a specific diameter for a part) [13].
- **anyOf**: This semantic allows any value within a range to satisfy the constraint, useful for approximations (e.g., tolerances or acceptable performance ranges) [13].
- **allOf**: This mandates that all contained values in a range must fulfill the constraint. This is applicable for operational envelopes (e.g., ensuring a system operates across an entire temperature range) [13].

To demonstrate this, a *Tank* part definition was modeled with attributes for width, height, and length (see Listing 3). These attributes leverage the *oneOf* operator to define permissible dimensions. The requirement *tankBigEnough* then enforces the technical standard that the derived volume attribute falls within a specific range [13].

```
part def Tank {
    attribute width: ISO::Length = oneOf(10.0 .. 100.0) [cm];
    attribute height: ISO::Length = oneOf(1.0 .. 3.0) [m];
    attribute length: ISO::Length = oneOf(1.0 .. 1.2) [m];

    attribute volume: ISO::Volume = width * height * length;
}

requirement tankBigEnough {
    subject t: Tank;
    require t.volume == oneOf(1000.0 .. 2000.0) [L];
}
```

Listing 3. Implementation of constructive model analysis using Range-Based Semantics (adapted from Ratzke et al. [13])

A formal verification of the design solution against its requirements ist achieved by applying constraint propagation, where the solver actively reduces the design space. This process restricts the dimensional attributes (width, height, length) to their valid domain, thereby ensuring that only values capable of satisfying the tankBigEnough requirement remain [11], [13].

V. Discussion

A. The Shift from Extrinsic to Native Constraints

The migration from SysML v1 to SysML v2 marks a transition from visual modeling augmented by extrinsic constraints to a framework where constraints are native mathematical primitives.

SysML v1 requires a disjointed verification workflow, where engineers must define the model structure, annotate it with extrinsic OCL constraints, and subsequently transform the entire artifact into formal languages (e.g., OWL, Alloy) to enable execution [1], [10]. As Lu et al. [10] establish, this reliance on external transformation creates a "semantic gap", where discrepancies between the semi-formal UML metamodel and the target logic often corrupt design intent [10].

SysML v2 fundamentally alters this flow by grounding the language in KerML (First-Order Logic), merging model definition and constraint specification into a single, natively formal step. As Molnar et al. [11] demonstrate, this formal foundation allows a single SysML v2 model to act as a "formal base ontology" that can be verified across diverse mathematical domains—ranging from theorem proving (Imandra) to model checking (Gamma) without the ambiguity of profile-based transformation. However, as Ratzke et al. [13] note, native declarative constraints (e.g., anyOf ranges) must still be reformulated into solver-specific inputs (e.g., Linear Programming) for analysis [13]. Thus, the burden shifts from transforming the model to generate meaning to reformulating the mathematical definition for computational efficiency.

B. Scalability and Complexity

The methodological shift from transformation to reformulation, discussed above, fundamentally alters the scalability and complexity of model verification. In SysML v1, the primary bottleneck was the *fidelity* of the mapping between the modeling tool and external solvers. In SysML v2, where system concepts are defined directly as native logical assertions, the bottleneck shifts to the *solvability* of the resulting mathematical structures.

Ratzke et al. [13] demonstrate that by compiling SysML v2 models into an underlying system of linear constraints, model analysis can be structured as a Constraint Satisfaction Problem (CSP). While this approach leverages the language's native semantics for rigorous verification, it exposes the analysis to the computational limits of CSP solvers, which are classified as NP-hard in best-case scenarios and NP-complete in others [11], [13].

Consequently, the "cost" of consistency is no longer paid in the manual effort of accurate transformation, but in the algorithmic runtime required to solve the reformulated logic. This trade-off is empirically evidenced by Cibrian et al. [4], who analyzed metamodel-driven verification in SysML v2. Their findings indicate that while reformulation ensures higher semantic coherence, the verification time can grow non-linearly with model size due to the depth of the resulting logical dependency graph. This contrasts sharply with SysML

v1, where OCL-based profile checks were computationally lightweight, albeit semantically shallow [4], [13].

To manage this computational complexity, Kausch et al. [9] argue that reformulated systems must adhere to strict compositional refinement. By decomposing high-level requirements into independent components, verification can be performed on subsystems rather than aggregated system elements. This approach ensures that the complexity of proofs grows linearly with the system's structural decomposition rather than exponentially with its state space, allowing theorem provers to verify system models that would otherwise be computationally infeasible [9].

VI. Conclusion and Outlook

This work addressed the research question of how the mechanisms for implementing modeling guidelines differ between SysML v1 and SysML v2, and how these differences impact the capabilities for automated verification. The comparative analysis reveals a fundamental paradigm shift from extrinsic constraint application in SysML v1 to intrinsic semantic definition in SysML v2.

SysML v1 relies on extrinsic constraints via stereotypes and OCL. Verification requires transforming models into external formalisms (e.g., OWL), a disjointed process susceptible to semantic gaps where design intent is often lost. Conversely, SysML v2 utilizes a metamodel-driven approach grounded in KerML. Guidelines become intrinsic properties inherited through the separation of Definition and Usage. This foundation in first-order logic shifts verification from transformation to reformulation, allowing models to be solved directly as Constraint Satisfaction Problems (CSPs) within the native environment.

While the semantic formalization of SysML v2 enhances consistency, it introduces scalability challenges as verification becomes bound by the computational limits of solving NP-hard Constraint Satisfaction Problems. Future research should look into the development of standardized semantic libraries to replace ad-hoc constraint definitions. This corresponds with emerging metamodel-driven tools that automate consistency checking against the formal ontology. Furthermore, "Model as Code" workflows enable SysML v2 to serve as a central artifact for formal analysis, allowing models to be transpiled for theorem provers and model checkers like Imandra and Gamma. Continued effort is required to bridge the gap between static structural verification and dynamic behavioral validation, ensuring rigorous consistency extends to operational logic.

Acknowledgment

The author would like to express sincere gratitude to Professor Sebastian Voss for his supervision and guidance throughout the preparation of this work. His support in defining the research scope regarding SysML modeling guidelines was fundamental to the direction of this comparative analysis. The author also acknowledges the use of Google Gemini as an AI thought partner during the drafting process, specifically for generating code snippets and reviewing the grammatical correctness of the manuscript.

References

[1] L. Beers, H. Nabizada, M. Weigand, F. Gehlhoff, and A. Fay, "A sysml profile for the standardized description of processes during system development," in 2024 IEEE International Systems Conference (SysCon). IEEE, 2024, pp. 1–8.

[2] S. Bergemann, "Challenges in multi-view model consistency management for systems engineering," in Modellierung 2022 Satellite Events. Bonn: Gesellschaft f¨ur Informatik e.V., 2022, pp. 77–89.

[3] K. Boelsen, M. May, G. Jacobs, and et al., "Sysml v2 based modelling guidelines for mechanical system elements," Forsch Ingenieurwes, vol. 89, no. 60, 2025.

[4] E. Cibrián, J. Olivert-Iserte, C. Díez-Fenoy, R. Mendieta, J. Llorens, and J. M. Álvarez Rodríguez, "Ensuring semantic consistency in sysml v2 models through metamodel-driven validation," IEEE Access, vol. 13, pp. 121 444–121 457, 2025.

[5] Systems and software engineering — System life cycle processes, ISO/IEC/IEEE Standard 15288:2023, May 2023.

[6] S. Friedenthal, "Future directions for mbse with sysml v2." in MODELSWARD, 2023, pp. 5–9.

[7] D. D. Walden, G. J. Roedler, K. J. Forsberg, R. D. Hamelin, and T. M. Shortell, Eds., *INCOSE Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities*, 4th ed. Hoboken, NJ, USA: John Wiley & Sons, 2015.

[8] N. Jansen, J. Pfeiffer, B. Rumpe, D. Schmalzing, and A. Wortmann, "The language of sysml v2 under the magnifying glass." J. Object Technol., vol. 21, no. 3, pp. 3–1, 2022.

[9] H. Kausch, M. Pfeiffer, D. Raco, B. Rumpe, and A. Schweiger, "Model-driven development for functional correctness of avionics systems: a verification framework for sysml specifications," CEAS Aeronautical Journal, vol. 16, no. 1, pp. 33–48, 2025.

[10] S. Lu, A. Tazin, Y. Chen, M. M. Kokar, and J. Smith, "Detection of inconsistencies in SysML/OCL models using OWL reasoning," SN Computer Science, vol. 4, no. 175, 2023.

[11] V. Molnár, B. Graics, A. Vörös, S. Tonetta, L. Cristoforetti, G. Kimberly, P. Dyer, K. Giammarco, M. Koethe, J. Hester et al., "Towards the formal verification of sysml v2 models," in Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, 2024, pp. 1086–1095.

[12] Object Management Group, "Kernel Modeling Language (KerML), Version 1.0," Object Management Group, Specification formal/25-09-01, Sep. 2025. [Online]. Available: https://www.omg.org/spec/KerML/1.0/PDF

[13] A. Ratzke, J. Koch, and C. Grimm, "Modeling and analysis of system models with constraints in sysml v2," in 2025 20th Annual System of Systems Engineering Conference (SoSE), 2025, pp. 1–6.

[14] Zavada, G. Kulcsár, V. Molnár, and Horváth, "Towards a configurable verification and validation framework for critical cyber-physical systems," in 2025 IEEE 8th International Conference on Industrial Cyber-Physical Systems (ICPS), 2025, pp. 1–6.