$N°$ d'ordre : 370/2025-C/IN

People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University of Science and Technology HOUARI BOUMEDIENE

Faculty of Computer Science

Thesis of Doctorate

Presented to obtain a **DOCTORATE degree**

In : COMPUTER SCIENCE

SPECIALITY : Formal Methods and Application

By : KHELIFATI Adel

Subject

Modeling and Verification of Reliable Cyber-Physical Systems Using SysML and the Component-Oriented Approach

Publicly defended, on 30/10/2025, before a jury composed of :

Mrs.	GHARBI Nawel	Professor	at USTHB	President of the Jury
Mrs.	BOUKALA Malika	Professor	at USTHB	Thesis supervisor
Mr.	HAMMAD Ahmed	Associate Professor	at UMLP	Thesis co-supervisor
Mr.	SEKHRI Larbi	Professor	at Oran University	Examiner
Mr.	FEREDJ Mohamed	Associate Professor	at USTHB	Examiner

To my beloved parents,
for their unconditional love, sacrifices, and unwavering support.

Acknowledgements

I would like to express my sincere gratitude to Professor Malika Boukala-Ioualalen, my thesis supervisor, for her continuous support, valuable guidance, and insightful feedback throughout this research. Her scientific rigor and attentiveness have been instrumental in the development and completion of this work.

I also extend my heartfelt appreciation to Dr. Ahmed Hammad for his steadfast guidance and availability during my doctoral journey. His technical expertise, critical thinking, and constructive advice have significantly shaped the direction of this research. I am sincerely grateful for his confidence in me and his constant encouragement.

I wish to pay tribute to the late Dr. Samir Chouali, who initially co-supervised this work. His early involvement, generous spirit, and encouragement left a lasting impression and were essential in setting the foundations of this research. His memory remains a source of inspiration.

I sincerely thank the members of the jury, Professor Nawel Gharbi, Professor Larbi Sekhri, and Dr. Mohamed Feredj, for dedicating their time and expertise to evaluate this thesis.

I would also like to thank Dr. Naila Houacine for her support and encouragement, as well as for her helpful advice throughout my doctoral studies.

I am profoundly grateful to my parents for their unwavering support, sacrifices, and constant encouragement throughout this academic journey. I also wish to express my deep gratitude to my brothers, Riad and Merouane, for their continuous support, patience, and belief in me during the most challenging moments of this work.

I would like to express my special thanks to my sister Nihed, her husband Mohamed, and their son Firas for their constant support, kindness, and encouragement. Their presence, understanding, and care have been a great source of strength and motivation during the most demanding periods of this thesis.

I am also deeply thankful to my friends Nour Nekhoul, Walid Bencheikh, Youcef Bourahla, Rachid Boudour,Tayeb Benzenati and Ahmed Zebouchi for their friendship, support, and encouragement. Their help, insightful discussions, and constant moral support have greatly contributed to making this journey more enriching and fulfilling.

Abstract

Cyber-Physical Systems (CPS) tightly couple software, communication networks, and physical processes in safety-critical domains such as autonomous transportation, smart buildings, and industrial automation. Ensuring their reliability requires modeling and verification techniques that can handle heterogeneous architectures, concurrent interactions, and requirements that may be vague or context dependent. Model-Based Systems Engineering (MBSE) with the Systems Modeling Language (SysML) offers rich multi-view models, but mainstream tools still focus mainly on syntactic checks and provide limited support for early structural analysis, formal interaction verification, and reasoning about soft or fuzzy requirements.

This dissertation proposes an integrated, model-driven workflow that strengthens SysML-based CPS development along three complementary axes. First, we formalize the static semantics of core SysML structural diagrams and derive consistency rules for early detection of architectural inconsistencies and missing traceability links between blocks and requirements. Second, we define a systematic mapping from SysML v2 interaction constructs to the Behavior-Interaction-Priority (BIP) framework, producing executable models for the simulation and formal analysis of deadlocks and interaction-level design flaws. Third, we introduce a method for modeling and verifying fuzzy, soft requirements in SysML v2 by combining native expression evaluation with batch, scenario-based simulations, enabling quantitative assessment of requirement satisfaction under uncertainty. The approach is evaluated on three CPS case studies: an autonomous transportation platform, a swarm of coordinated drones, and a Heating, Ventilation, and Air Conditioning (HVAC) system. The results demonstrate improved support for structural verification, interaction analysis, and handling of fuzzy requirements while preserving traceability at the SysML model level.

Keywords

Cyber-Physical Systems; Model-Based Systems Engineering; SysML; SysML v2; BIP framework; structural consistency; interaction verification; fuzzy requirements.

Résumé

Les systèmes cyber-physiques (CPS) couplent étroitement logiciels, réseaux de communication et processus physiques dans des domaines critiques (transport autonome, bâtiments intelligents, automatisation industrielle). Leur fiabilité exige des techniques de modélisation et de vérification capables de traiter des architectures hétérogènes, des interactions concurrentes et des exigences parfois floues ou dépendantes du contexte. L'ingénierie des systèmes basée sur les modèles (MBSE) avec SysML offre des modèles multi-vues riches, mais les outils courants restent centrés sur des vérifications syntaxiques et apportent peu de support à l'analyse structurelle précoce, à la vérification formelle des interactions et au raisonnement sur les exigences floues.

Cette thèse propose un flux de développement intégré, piloté par les modèles, qui renforce l'usage de SysML pour la conception de CPS selon trois axes complémentaires. (1) Nous formalisons la sémantique statique de diagrammes structurels SysML et dérivons des règles de consistance pour détecter tôt les incompatibilités architecturales et les ruptures de traçabilité exigences–blocs. (2) Nous définissons un mappage systématique des constructions d'interaction de SysML v2 vers le cadre Behavior–Interaction–Priority (BIP), produisant des modèles exécutables pour la simulation et l'analyse formelle des interblocages et des défauts de conception au niveau des interactions. (3) Nous introduisons une méthode de modélisation et de vérification des exigences floues dans SysML v2, combinant l'évaluation native d'expressions avec des simulations par scénarios en mode batch, afin de quantifier la satisfaction des exigences sous incertitude. L'approche est évaluée sur trois études de cas de CPS : une plateforme de transport autonome, un essaim de drones coordonnés et un système de chauffage–ventilation–climatisation (HVAC), et elle améliore la vérification structurelle, l'analyse des interactions et la prise en compte des exigences floues, tout en préservant la traçabilité au niveau des modèles SysML.

Mots-clés

Systèmes cyber-physiques ; Ingénierie des systèmes basée sur les modèles ; SysML ; SysML v2 ; cadre BIP ; consistance structurelle ; vérification des interactions ; exigences floues.

ملخص

تجمع الأنظمة السيبرانية-الفيزيائية (CPS) بين البرمجيات، وشبكات الاتصال، والعمليات الفيزيائية في مجالات حرجة مثل النقل الذكي والمباني والأتمتة الصناعية. ويتطلّب ضمان موثوقيتها تقنيات نمذجة وتحقق قادرة على التعامل مع معماريات غير متجانسة، وتفاعلات متزامنة، ومتطلبات قد تكون ضبابية أو معتمدة على السياق. توفّر هندسة النظم القائمة على النماذج (MBSE) باستخدام SysML نماذج غنيّة متعددة المناظير، غير أنّ الأدوات الحالية تركّز أساسًا على الفحوصات التركيبية ولا تدعم بما يكفي التحليل البنيوي المبكر، والتحقق الشكلي من التفاعلات، والتعامل المنهجي مع المتطلبات «المرنة» أو الضبابية .

تقترح هذه الأطروحة سلسلة تطوير متكاملة موجهة بالنماذج تعزّز استخدام SysML في تصميم الـ CPS وفق ثلاثة محاور متكاملة: (1) ترسيم الدلالة الساكنة للمخططات البنيوية في SysML واستخلاص قواعد اتساق للكشف المبكر عن التعارضات المعمارية وانقطاع التتبع بين الكتل والمتطلبات؛ (2) تعريف مطابقة منهجية لبنى التفاعل في SysML v2نحو إطار Behavior–Interaction–Priority (BIP) من أجل توليد نماذج تنفيذية للمحاكاة والتحليل الشكلي للانسدادات (deadlocks) وأخطاء التصميم على مستوى التفاعلات؛ (3) اقتراح منهجية لنمذجة والتحقق من المتطلبات الضبابية في SysML v2 من خلال الجمع بين التقييم الأصلي للتعبيرات ومحاكاة جماعية قائمة على السيناريوهات لقياس درجة تحقق المتطلبات تحت عدم اليقين.

تُطبَّق المقاربة على ثلاث دراسات حالة لأنظمة سيبرانية-فيزيائية: منصة نقل ذاتي، وسرب طائرات مسيّرة منسّقة، ونظام للتدفئة والتهوية والتكييف (HVAC)، وتُظهر النتائج تحسّن دعم التحقق البنيوي، وتحليل التفاعلات المعقدة، والتعامل المتكامل مع المتطلبات الضبابية مع الحفاظ على التتبع على مستوى نماذج SysML.

الكلمات الدالة

الأنظمة السيبرانية-الفيزيائية؛ هندسة النظم القائمة على النماذج؛SysML ؛SysML v2 ؛ BIP؛ الاتساق البنيوي؛ التحقق من التفاعلات؛ المتطلبات الضبابية.

Table of contents

List of figures

List of tables

List of Algorithms

List of abbreviations

AADL	**A**rchitecture **A**nalysis and **D**esign **L**anguage
AGREE	**A**ssume **G**uarantee **R**easoning **E**nvironment
API	**A**pplication **P**rogramming **I**nterface
AVATAR	A SysML profile for real-time systems with formal semantics
BDD	**B**lock **D**efinition **D**iagram
BIP	**B**ehavior, **I**nteraction, **P**riority
CPS	**C**yber-**P**hysical **S**ystem
FAME	**F**uzzy **A**daptive **M**odeling **E**nvironment
fUML	**F**oundational **U**ML (Executable UML subset with formal semantics)
HVAC	**H**eating, **V**entilation, and **A**ir **C**onditioning
IBD	**I**nternal **B**lock **D**iagram
MARTE	**M**odeling and **A**nalysis of **R**eal-**T**ime and **E**mbedded systems (UML profile)
MBSE	**M**odel-**B**ased **S**ystems **E**ngineering
OCaml	A general-purpose functional programming language (used in tooling)
OCL	**O**bject **C**onstraint **L**anguage
OCRA	**O**perational **C**ontracts for **R**eal-time **A**pplications

OMG	**O**bject **M**anagement **G**roup
PACTI	**P**assivity **A**nd **C**ompatibility for **T**imed **I**nterfaces
Reo	A channel-based coordination language for compositional interaction modeling
SysML	**S**ystems **M**odeling **L**anguage
UML	**U**nified **M**odeling **L**anguage
UPPAAL	**U**ppsala **P**rotocol **A**nalysis **A**utomata **L**ibrary (tool for verifying timed automata)
XMI	**X**ML **M**etadata **I**nterchange (format for model exchange between tools)

Part I

Introduction

Chapter 1

Introduction

1.1 Context and Motivation

The increasing integration of computational elements into physical environments has led to the emergence of Cyber-Physical Systems (CPS), which combine computing, communication, and physical processes through continuous feedback loops [31]. These systems are now widely deployed in safety-critical domains such as autonomous transportation, smart energy grids, industrial automation, and healthcare, where they must satisfy strict requirements related to reliability, safety, and adaptability [22].

The design and verification of CPS present important challenges, mainly due to their intrinsic complexity and heterogeneous nature. Multiple viewpoints, such as structural, behavioral, and functional aspects, must be consistently modeled and maintained throughout the system lifecycle. To manage this complexity, Model-Based Systems Engineering (MBSE) has become an essential approach. It relies on high-level abstractions and modeling standards that promote consistency, traceability, and reuse. The Systems Modeling Language [44] (SysML) is one of the most widely used languages in this context, offering support for representing system requirements, architecture, and behavior.

Despite its popularity, SysML remains limited when used in the development of dependable CPS [47]. Most modeling tools focus on syntactic checks, leaving many semantic inconsistencies undetected. For example, mismatches between ports, improper delegation paths, or missing requirement links may persist in a model without being flagged. In addition, SysML lacks a formal execution semantics to represent and analyze component interactions, making it difficult to reason about properties such as synchronization correctness, deadlock avoidance, or execution priorities. Furthermore, SysML does not natively support the representation and evaluation of requirements

that are qualitative or context-dependent, such as those involving comfort, energy efficiency, or usability.

These limitations highlight the need for complementary approaches that can enhance SysML-based modeling with rigorous verification capabilities. In particular, addressing structural inconsistencies, validating dynamic interactions, and reasoning about vague or context-dependent requirements remain open challenges in the development of dependable CPS. These challenges form the backdrop against which this thesis is situated.

1.2 Research Problem and Objectives

The central research problem addressed in this thesis concerns the enhancement of verification capabilities in the modeling of Cyber-Physical Systems (CPS) using the Systems Modeling Language (SysML). As CPS increasingly integrates complex architectures, concurrent behaviors, and context-dependent requirements, ensuring their correctness from the early stages of design becomes essential.

To address these challenges, this research pursues four key objectives. The first is to enable the early verification of structural correctness in SysML models by defining and enforcing semantic constraints between model elements and requirements. The second objective is to support the formal verification of component interactions by establishing a transformation from SysML models to a suitable formal framework, thereby ensuring coordination correctness and behavioral soundness. Third, the thesis aims to propose a verification approach tailored to imprecise, qualitative, or context-sensitive requirements by leveraging modeling constructs coupled with mechanisms for approximate evaluation. Finally, the proposed methods are validated through representative case studies drawn from various CPS domains, demonstrating their applicability and effectiveness in real-world scenarios.

1.3 Contributions

This thesis presents three key contributions, each addressing a critical challenge in the model-based engineering and verification of Cyber-Physical Systems (CPS). These contributions are summarized in Figure 1.1 and outlined as follows:

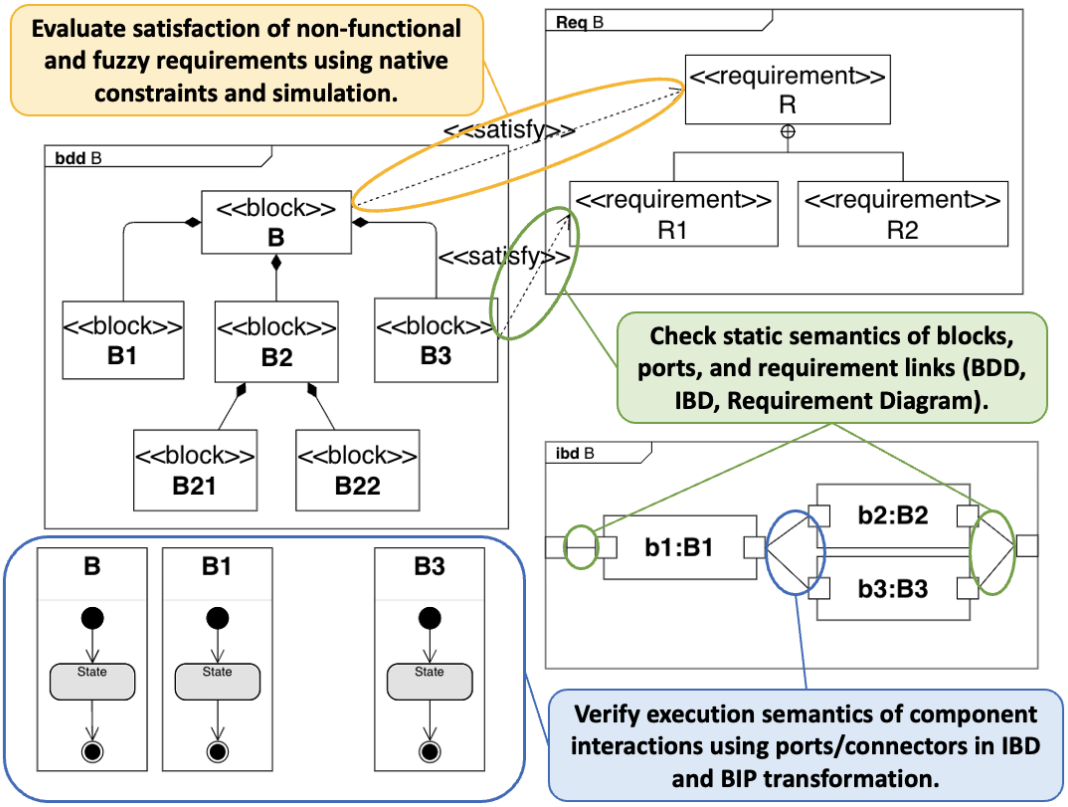

Fig. 1.1: Overview of Thesis Contributions and Corresponding SysML Modeling Views

- **Early Verification of Structural Consistency in SysML Models**: This contribution introduces a formal methodology for verifying the structural consistency of SysML models using well-defined semantic constraints. It enables early detection of modeling inconsistencies without relying on behavioral specifications and supports traceability throughout the design process. It corresponds to the green-highlighted area in Figure 1.1, covering the static semantics of blocks, ports, and requirements across block definition diagrams (BDD), internal block diagrams (IBD), and requirement diagrams.

- **Formal Interaction Verification through SysML-to-BIP Transformation**: This contribution defines a transformation from SysML v2 models to the Behavior, Interaction, Priority (BIP) framework. It enables the formal analysis of component coordination using BIP's execution semantics, ensuring correctness and mitigating unintended behaviors. The blue-highlighted area in Figure 1.1 illustrates this contribution, focusing on port-based interactions and their verification.

- **Modeling and Verification of Fuzzy Requirements in SysML v2**: This contribution proposes an approach for specifying and evaluating soft or imprecise system requirements using fuzzy logic. It combines native expression evaluation in SysML v2 with external scenario-based simulation to support quantitative assessment under uncertainty. This is illustrated in the yellow-highlighted section of Figure 1.1, which shows how non-functional requirements are linked to structural elements via `«satisfy»` relations and evaluated using constraint expressions and simulation results.

1.4 Publications

In the following, we list the references for the published articles:

1. Adel Khelifati, Ahmed Hammad, Malika Boukala-Ioualalen. *Combining SysML v2 and BIP to Model and Verify CPS Interactions*. Proceedings of the 20th International Conference on Software Technologies (ICSOFT 2025). DOI: 10.5220/0013645700003964

2. Adel Khelifati, Malika Boukala-Ioualalen, Ahmed Hammad. *Fuzzy Requirements Verification in SysML v2: Direct Modeling and Scenario-Based Analysis for CPS*. Proceedings of the 20th International Conference on Software Technologies (ICSOFT 2025). DOI: 10.5220/0013648400003964

3. Adel Khelifati, Malika Boukala-Ioualalen, Ahmed Hammad. *Construction of consistent SysML models applied to the CPS*. ACM Journal on Emerging Technologies in Computing Systems, 2024. DOI: 10.1145/3702326

1.5 Dissertation Outline

This dissertation is organized into four main parts, each reflecting a major phase of the research process, from problem framing to methodological contributions and final reflections.

Part I – Introduction provides a global overview of the research. It introduces the context and motivation for the work, formulates the research problem and objectives, outlines the main contributions, and lists the associated scientific publications. This part ends with a description of the dissertation structure.

Part II – Context and State-of-the-art establishes the theoretical and methodological background. Chapter 2 (*State-of-the-art*) reviews foundational concepts of Cyber-Physical Systems (CPS), explores key modeling languages such as SysML and BIP, and identifies major verification challenges including structural consistency, interaction correctness, and requirement imprecision. Chapter 3 (*Modeling Foundations*) introduces the formal languages and tools used in the thesis, including SysML v1, SysML v2, the BIP framework, fuzzy logic principles, and the OCaml language used for rule-based verification.

Part III – Research Contributions presents the core technical contributions of the thesis. Chapter 4 details a formal method for verifying structural consistency in SysML models without requiring behavioral specifications. Chapter 5 defines a model transformation from SysML v2 to BIP, enabling the verification of structured interactions. Chapter 6 addresses the modeling and verification of fuzzy requirements by combining SysML v2 constructs with fuzzy logic and simulation-based evaluation. Each contribution is supported by a dedicated case study drawn from real-world CPS domains.

Part IV – Conclusion and Future Work concludes the thesis by summarizing the main contributions and discussing their impact. It also outlines possible directions for future research, including toolchain automation, integration with runtime monitoring, and application to large-scale industrial systems.

Part II

Context & State-of-the-art

Chapter 2

State-of-the-art

2.1 Introduction

Cyber-physical systems (CPS) constitute a foundational paradigm in the design of modern engineering systems where software components interact continuously with the physical environment [24]. By unifying computation, communication, and control, CPS reshapes the way systems are designed and operated. As they become increasingly pervasive and complex, ensuring their correctness, safety, and performance is both a scientific and engineering imperative.

This chapter lays the foundational groundwork for the rest of the thesis. It begins by introducing the core concepts and definitions of CPS, then provides a structured overview of the modeling languages used to describe CPS architectures and behaviors, distinguishing between semi-formal, formal, and hybrid approaches. Finally, although CPS development involves a wide range of verification challenges, this thesis focuses on three fundamental aspects: structural consistency, dynamic interaction correctness, and requirement satisfaction. Throughout, particular emphasis is placed on the limitations of existing approaches and the rationale for selecting SysML, particularly SysML v2, as a pivotal modeling language. These foundational insights establish the context and motivation for the novel contributions proposed in the subsequent chapters.

2.2 Cyber-Physical Systems: Definitions and Application Domains

2.2.1 Definitions from the Literature

Cyber-Physical Systems (CPS) are an emerging class of engineering systems that tightly integrate computational algorithms and physical components. Over the past two decades, CPS have gained prominence as a paradigm for critical infrastructures and advanced technologies, spanning domains such as Transportation systems, smart cities, healthcare, and Emergency rescue systems [22]. The term "cyber-physical system" was originally coined around 2006 by the U.S. National Science Foundation, reflecting the vision of next-generation systems, where computing elements monitor and control physical processes in a networked feedback loop [31]. A well-known definition by [50] characterizes CPS as "physical and engineered systems whose operations are monitored, coordinated, controlled and integrated by a computing and communication core". In essence, a CPS links the cyber world of software and networks with the physical world of sensors, actuators, and processes, achieving a synergy of computation, communication, and control.

Beyond simply combining physical and computational parts, CPS are fundamentally about their continuous interaction. As emphasized in [32], understanding CPS requires more than analyzing the cyber and physical aspects independently; it demands a unified view of their mutual influence through real-time feedback loops. The behavior of a CPS is thus co-defined by its physical dynamics and its computational logic, making their integration a core intellectual and engineering challenge.

2.2.2 Application Domains

Cyber-Physical Systems (CPS) significantly impact various sectors of modern society, with applications spanning automotive, aerospace, energy, healthcare, manufacturing, infrastructure, and other domains [22]. These systems integrate computational elements with physical processes, resulting in complex, responsive, and interactive technologies embedded across different industries.

In the automotive and transportation sector, CPS technologies support advanced driver-assistance systems (ADAS), autonomous vehicles, and communication infrastructures like vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2X) [41]. Modern vehicles embody CPS principles, incorporating numerous electronic control units that interact with physical subsystems such as engines, braking mechanisms, and environmen-

tal sensors. Similarly, unmanned aerial vehicles (UAVs) and intelligent transportation systems (ITS) rely on CPS to manage navigation, optimize traffic flow, and enhance operational safety through coordinated real-time communication and control.

In aerospace and avionics, CPS play a central role in the operation of fly-by-wire systems, autonomous spacecraft, and integrated air traffic management. These systems require precise coordination between sensor inputs, flight control software, and actuators, with communication loops extending between airborne and ground systems. Given the safety-critical nature of aerospace applications, CPS in this domain are subject to rigorous verification aligned with standards like DO-178C [17], emphasizing reliability and robustness.

The energy sector has witnessed a transformation through the development of smart grids, where CPS technologies integrate distributed sensors, smart meters, and control algorithms to monitor and adjust electricity generation, distribution, and consumption [2]. The inclusion of renewable energy sources, such as solar and wind, further increases system complexity, necessitating sophisticated CPS architectures capable of maintaining grid stability, efficiency, and resilience.

In healthcare, CPS are embedded in intelligent medical devices, robotic surgical tools, and networked patient monitoring systems [27]. These systems combine real-time physiological data acquisition with software-driven control to support interventions, such as automated insulin pumps that regulate dosage based on continuous glucose monitoring. The critical nature of medical CPS underscores the importance of regulatory compliance, safety assurance, and thorough validation to prevent harm to patients.

Modern manufacturing, under the Industry 4.0 initiative, leverages CPS to enhance automation, adaptability, and efficiency [36]. Cyber-physical production systems comprise interconnected machines, sensors, and actuators capable of autonomous operation and reconfiguration. Real-time feedback enables systems to adjust production processes dynamically in response to changing requirements. Interoperability among heterogeneous systems and the integration of CPS into planning and control hierarchies remain central challenges in this domain.

Urban infrastructure and smart city applications exemplify the integration of CPS at scale. Building automation systems, smart water and energy management networks, and environmental monitoring platforms all utilize CPS to optimize performance and sustainability. Smart homes, featuring adaptive control over lighting, heating, and security systems, as well as city-wide initiatives like automated traffic and emergency response management, illustrate the breadth of CPS deployment [22]. These systems,

while promising enhanced quality of life, raise important issues related to privacy, reliability, and resilience.

In this thesis, we focus particularly on three representative domains that exemplify distinct verification challenges in CPS engineering. First, autonomous vehicular systems, such as driverless transport platforms, illustrate the need for early structural consistency checking to prevent architectural mismatches. Second, aerial swarm coordination scenarios emphasize the verification of complex dynamic interactions and priorities among multiple autonomous agents. Third, building automation systems, such as HVAC (Heating, Ventilation, and Air Conditioning system) control, highlight the challenge of reasoning about uncertain or imprecise requirements using fuzzy logic. These application domains serve as case studies throughout the thesis to evaluate and demonstrate the proposed methodologies.

2.3 Modeling Languages for CPS

Cyber-Physical Systems (CPS) are complex integrations of software, hardware, and physical processes, and their design involves multiple levels of abstraction. In practice, engineers use modeling languages at different levels: high-level modeling languages for capturing the system architecture and requirements, and low-level formalisms for rigorous analysis and verification. This section discusses the conceptual and methodological differences between these two categories and the necessity of combining them in a model-based CPS development process. High-level languages (e.g. SysML [44], AADL [21], MARTE [23]) enable early design exploration and communication, while low-level formalisms (e.g., BIP [5], timed automata [1], Petri nets [49], hybrid automata [28]) provide mathematically precise semantics for formal verification. We explain why high-level languages are indispensable for managing design complexity and why low-level formalisms are necessary for ensuring correctness. Finally, we examine bridging approaches, model transformations, and hybrid methods that aim to leverage the strengths of both, leading to the verification-focused methodology adopted later in this thesis.

2.3.1 High-Level Modeling Languages

High-level modeling languages are essential in the early design phases of Cyber-Physical Systems (CPS), where they help describe system architecture, behavior, requirements, and interfaces at an abstract level. These languages are central to Model-Based

Systems Engineering (MBSE), emphasizing clarity and modularity over implementation specifics. They often use graphical or structured notations to enable cross-disciplinary communication and early design validation.

A key example is the Systems Modeling Language [44] (SysML), a general-purpose language standardized by the Object Management Group (OMG), tailored for the development of complex systems. SysML supports a variety of diagram types, including requirements, block definitions, state machines, and parametric diagrams, enabling a multi-view representation of the system. Its abstraction mechanisms and extensive tool support (e.g., Cameo [14], Eclipse Papyrus [19]) help engineers manage complexity, maintain consistency, and align with industrial standards. SysML also serves as a pivot modeling language due to its extensibility and compatibility with domain-specific profiles.

Other examples include the Architecture Analysis and Design Language [21] (AADL) and the MARTE [23] profile for UML. AADL is oriented toward embedded systems, focusing on platform-specific aspects like scheduling and fault modeling. MARTE [23] introduces constructs for real-time and embedded systems, particularly targeting timing and performance analysis. These languages facilitate multi-domain modeling and benefit from standardization and extensibility.

The strengths of high-level modeling languages include their intuitive notations, support for abstraction and multi-view modeling, and the ability to facilitate collaboration among stakeholders. They enable designers to conduct early analysis, simulate high-level behaviors, and trace requirements through the design. Their integration into MBSE toolchains offers features like consistency checking and model traceability, making them highly usable in large-scale projects.

However, a key limitation is their lack of precise execution semantics. While high-level languages have well-defined syntax, they often lack mathematically rigorous semantics needed for direct formal verification. For instance, SysML's behavioral models cannot be directly subjected to model checking without transformation, due to their narrative and often ambiguous execution semantics. Although extensions like OCL [57] or fUML [43] attempt to add formality, most high-level languages remain semi-formal, limiting their utility for rigorous correctness guarantees.

In summary, high-level modeling languages offer expressive and accessible means for CPS design and communication but fall short in providing the precision needed for formal verification. This motivates the integration of formal methods to bridge the gap between intuitive modeling and mathematical assurance, a theme further developed in the following sections.

2.3.2 Low-Level Modeling Formalisms

Low-level Modeling formalisms are mathematical modeling languages designed to rigorously describe and verify system behavior, especially in concurrent, real-time, or hybrid settings. These languages include timed automata, Petri nets, hybrid automata, and component-based frameworks like BIP (Behavior-Interaction-Priority). They operate at a lower level of abstraction, capturing exact behavior through states, transitions, timing, and interaction rules.

Timed automata [1], for instance, use clock variables and guards to model timing constraints in systems. Tools like UPPAAL [6] allow verification of such models using temporal logic, enabling checks on deadlines, safety properties, and reachability. Petri nets [49], another formalism, model concurrency and resource sharing using token-based transitions. They are effective for analyzing synchronization, deadlocks, and throughput in distributed systems. Hybrid automata [28] extend these principles to systems with continuous dynamics, modeling interactions between software logic and physical processes using differential equations.

The BIP [5] framework offers a component-based formalism with precise semantics for behavior, interactions, and execution priorities. It allows the construction of correct-by-design systems by enforcing properties like deadlock-freedom at the model level. BIP also supports simulation and code generation while preserving formal properties.

These formalisms excel in expressiveness, precision, and suitability for exhaustive verification techniques like model checking or theorem proving. Their mathematical rigor enables strong correctness guarantees, particularly vital in safety-critical CPS domains such as automotive or aerospace.

However, they present practical challenges. Formal models are often challenging to scale and analyze effectively when applied to large and complex systems. They require expert knowledge and can be less accessible to multidisciplinary teams. Additionally, the lack of intuitive notations and tool fragmentation hampers their direct use in early design phases.

In summary, low-level formalisms are indispensable for rigorous analysis and verification but are challenging to apply directly at the architectural design level. This necessitates hybrid approaches that connect them with high-level modeling languages, enabling both usability and analytical rigor.

2.3.3 Bridging the Gap: Transformation and Hybrid Approaches

Bridging the gap between high-level modeling and formal verification is essential in CPS development. Hybrid approaches aim to combine the user-friendly abstractions of high-level languages like SysML with the rigorous analysis capabilities of formal methods. This integration is achieved through model transformations, formal extensions, or co-simulation frameworks.

Model transformation is a common strategy where high-level models are translated into formal representations such as timed automata, BIP, or Petri nets. These transformed models can then be analyzed using verification tools, enabling property checks while retaining links to the original design. Tools supporting SysML-to-formal transformations have been explored for automata-based and logic-based verifications.

Another approach involves enriching high-level languages with formal semantics through dedicated profiles, such as AVATAR [48] for SysML, allowing annotated models to be verified directly. With SysML v2's focus on formal semantics, such integration is expected to improve, narrowing the semantic gap.

Co-modeling and co-simulation frameworks like INTO-CPS [30] provide a third strategy. They enable the simulation of heterogeneous components described in different modeling languages while preserving architectural consistency. Though not fully formal, such frameworks often incorporate formal checks, ensuring model integrity across domains.

Despite progress, challenges persist: ensuring semantic alignment during transformation, maintaining traceability of verification results, and achieving tool interoperability. Managing consistency across evolving models is also complex. These issues highlight the importance of methodical integration techniques.

In summary, hybrid approaches offer promising solutions by connecting high-level design and formal verification, facilitating rigorous analysis without compromising on usability and abstraction.

2.4 Verification of Cyber-Physical Systems

The verification of Cyber-Physical Systems (CPS) encompasses a broad spectrum of challenges, including timing analysis, fault tolerance, probabilistic behaviors, and hybrid dynamics. This thesis focuses on three fundamental and complementary aspects: structural consistency, dynamic interaction correctness, and soft requirement

satisfaction. Each of these verification aspects addresses a distinct but interrelated dimension of system correctness. Structural consistency verification ensures architectural coherence and consistent interconnections between components. Dynamic interaction verification focuses on the correctness of interactions at runtime, such as synchronization, communication, and concurrency. Soft requirement verification, on the other hand, focuses on determining whether the system satisfies vague, imprecise, or context-dependent expectations.

This section reviews the state of the art related to these three verification axes, specifically aligned with the scope of this research. Other critical verification challenges, including real-time scheduling, probabilistic reasoning, and fault resilience, are recognized but lie outside the scope of this thesis.

In the subsections that follow, we analyze current methods and tools for structural, dynamic, and requirement verification, highlighting their limitations and identifying opportunities that motivate the contributions developed in the upcoming chapters.

2.4.1 Structural Consistency Verification

Structural verification consists of analyzing the architecture and static structure of Cyber-Physical Systems (CPS) to ensure consistency and correctness from the earliest stages of the design process. This early validation is critical, as errors detected in later development phases, particularly during integration, are significantly more expensive and difficult to correct.

However, verifying the structural correctness of CPS models presents inherent challenges, mainly due to the semi-formal nature of widely used modeling languages like SysML. These languages, while effective in supporting high-level architectural representation and interdisciplinary communication, lack the formal execution semantics required for thorough automated verification. Traditional SysML-based tools primarily enforce syntactic and typing constraints. However, they often fail to detect deeper structural inconsistencies, such as mismatched component assumptions or architectural cycles.

To address these limitations, several formal approaches have been proposed. Bhave et al. [7] use typed graphs to verify consistency between architectural views and base models, relying on graph morphisms. Magureanu et al. [38] employ the Z formal language to validate static properties within UML deployment diagrams, though their approach does not support SysML integration.

In the context of UML, Li et al. [34] introduce semantics for sequence diagrams to check inter-diagram consistency. Chanda et al. [10] provide a traceability framework for

verifying syntactic and semantic correctness across multiple UML diagrams, while Torre et al. [54] consolidate consistency rules. Nonetheless, most of these efforts overlook SysML-specific diagrams, particularly internal block and requirement diagrams, which are crucial in CPS design.

Some authors have addressed structural consistency between MBSE and security modeling paradigms, as in the work of Demachy et al. [15] and Vidalie et al. [56]. These contributions, however, are security-focused and do not offer general solutions for system-wide structural consistency.

Contract-based approaches offer an alternative paradigm, enabling structural and behavioral reasoning via assumptions and guarantees. AGREE [12], PACTI [29], and OCRA [11] exemplify such tools. Nevertheless, their reliance on detailed behavioral specifications limits their use in early architectural validation.

Translation-based methods, like that of Carillo[9], generate formal models such as Petri nets or automata from SysML diagrams. These methods are potent but again depend on comprehensive behavioral input.

In summary, although several techniques support structural verification, few operate directly on SysML structural models without requiring behavioral enrichment. There remains a need for dedicated, behavior-independent methods for early structural consistency verification in CPS design.

2.4.2 Dynamic Interaction Verification

Cyber-Physical Systems (CPS) exhibit intricate run-time behaviors due to the concurrent execution and real-time coordination of heterogeneous components, including sensors, controllers, and actuators. This concurrency raises critical challenges such as synchronization mismatches, race conditions, and deadlocks. Ensuring correct dynamic behavior is especially vital in safety-critical domains, where coordination errors can lead to catastrophic outcomes.

Although system-level modeling languages like SysML are widely adopted for their expressive design capabilities, they fall short in supporting the formal specification of dynamic behavior. In particular, SysML lacks precise semantics for concurrent interactions, making it unsuitable for verifying properties such as liveness, mutual exclusion, or correct synchronization patterns. Recent efforts, including the work of Molnár et al. [40], have explored extensions to SysML v2 to support property validation, but fine-grained interaction semantics and priority modeling remain largely unaddressed.

Formal frameworks such as BIP (Behavior-Interaction-Priority) have been proposed to overcome these limitations. Previous attempts to bridge SysML with BIP, such as the bidirectional transformation approach by Nussbaumer and Kieliger [42], primarily targeted SysML v1 using custom UML profiles. However, these mappings often emphasize structural elements and neglect dynamic interaction constructs like rendezvous synchronization, broadcast coordination, and execution priorities. Moreover, the richer semantics and modularity introduced in SysML v2 have not yet been fully exploited in these approaches.

To address dynamic behaviors thoroughly, alternative research has investigated constraint-based verification approaches, such as combining SysML with Reo and Object Constraint Language (OCL). Tannoury et al. [51] introduced SysReo, a method integrating SysML models with constraint automata to specify and verify interactions rigorously. This was further extended by incorporating real-time constraints using MARTE, making it particularly relevant for automotive systems [52]. Despite its utility, the Reo-based method relies heavily on external automata translations, potentially complicating the design and analysis process. In contrast, frameworks like BIP offer more comprehensive support for compositional modeling and inherent execution semantics, eliminating additional translation overhead.

Further studies, such as Dokter et al. [18], have demonstrated semantic mappings between BIP and Reo. Yet, these remain disconnected from mainstream system modeling environments and do not support seamless model continuity throughout the engineering workflow.

To enable rigorous and practical dynamic verification, there is a growing need for explicit interaction modeling constructs within high-level languages like SysML, supported by formal backends like BIP. Such integration would empower engineers to model, simulate, and verify critical interaction behaviors early in the development process without abandoning the expressiveness and usability of system-level design environments.

2.4.3 Verification of Requirements

Requirements verification plays a vital role in ensuring that Cyber-Physical Systems (CPS) meet both their intended functionalities and quality expectations. These requirements span a broad spectrum, from well-defined functional specifications, such as timely actuator responses, to more qualitative and subjective concerns, including usability, robustness, and maintainability. Accordingly, requirements verification necessitates

a multifaceted approach that combines formal analysis, simulation techniques, and reasoning under uncertainty.

For functional requirements, formal verification techniques such as model checking and theorem proving are widely adopted. Model checking involves encoding requirements as temporal logic properties and exhaustively exploring the system's state space to verify compliance. This method has been effectively used to uncover subtle logical errors in abstract models of CPS components that are difficult to detect using standard testing methodologies. Theorem proving, relying on deductive reasoning, offers another rigorous approach for establishing the logical validity of design properties. Despite their strong guarantees, both methods face scalability and complexity issues when applied to full CPS architectures, often requiring simplified models or the verification of selected subsystems.

As a practical complement, scenario-based testing and simulations remain indispensable. Approaches like hardware-in-the-loop and software-in-the-loop simulations enable validation in realistic operational contexts, helping to reveal integration issues and increase confidence in system behavior. However, as these techniques rely on sampled inputs and predefined scenarios, they cannot guarantee system correctness across all operating conditions. This limitation underscores the inherent trade-off between the completeness offered by formal verification and the empirical scope of simulation-based validation.

Verifying non-functional requirements, including performance, reliability, and safety attributes, often requires quantitative or probabilistic modeling. This typically involves extending system models with constructs such as clocks, stochastic parameters, or continuous dynamics, followed by analysis using specialized tools or simulation frameworks. Unlike functional requirements, these evaluations rarely yield binary outcomes and instead involve performance thresholds, probabilistic guarantees, or trade-off analyses.

A particularly challenging class of requirements are those that are imprecise, context-dependent, or linguistically vague, often referred to as fuzzy requirements. These include stakeholder expectations expressed in terms such as "the system should respond quickly" or "maintain acceptable temperature," which cannot be reduced to crisp true/false evaluations. To handle this uncertainty, fuzzy logic offers a mathematical framework for modeling such requirements using degrees of truth, enabling partial satisfaction evaluation based on membership functions.

Early contributions, such as those by Bubenko et al. [8] and Liu and Yen [35], introduced fuzzy sets as a means to represent vague or incomplete stakeholder demands. These works focused on elicitation and high-level requirement refinement but lacked

executable or verifiable models. Later, Baresi et al. [4] incorporated fuzzy reasoning into goal-oriented frameworks to support runtime adaptation and satisfaction analysis, although these methods were not integrated into system-level modeling practices.

Fuzzy logic has also been embedded into modeling languages, particularly UML. For example, Ma et al. [37] proposed fuzzy classes and relationships in UML class diagrams to capture uncertainty at the conceptual level. Han et al. [25, 26] further developed the FAME UML profile to support fuzzy modeling of adaptive systems. Despite their relevance, these approaches primarily focus on design-time representation and adaptation mechanisms, offering limited support for integrated and automated verification.

Recent frameworks like PERSA [16] and the approach of Egesoy and Güzel [20] further emphasize the applicability of fuzzy techniques in handling soft requirements. However, these proposals often remain abstract and external to mainstream system modeling languages such as SysML or UML, making them less practical for integration in model-driven engineering workflows.

Domain-specific applications provide additional insights. Taratoukhine and Yadgarova [53] employed fuzzy logic in product lifecycle management to handle requirement variability. In contrast, Dagli et al. [13] combined fuzzy reasoning with evolutionary algorithms for architectural evaluation under uncertainty. These studies demonstrate the potential of fuzzy logic for exploring design spaces, though they are less concerned with systematic verification.

Some efforts, such as Yoo and Lee [58], attempted partial integration of fuzzy verification with standard modeling environments by combining SysML parametric diagrams with MATLAB-based maintainability analysis. However, such hybrid approaches require extensive external toolchains and lack native, traceable verification support within the modeling environment itself.

In summary, current methods for verifying fuzzy requirements generally fall into three categories: conceptual models lacking execution semantics, UML extensions with limited verification capabilities, and hybrid approaches relying on external tools. A significant research gap persists in enabling direct, coherent, and reusable verification of fuzzy requirements within standard CPS modeling environments. Bridging this gap would contribute to unified frameworks capable of handling both crisp functional and soft qualitative requirements in an integrated manner.

2.5 Conclusion

Cyber-Physical Systems (CPS) represent a foundational paradigm for the design of modern systems characterized by a tight integration between computation, communication, and physical processes. This chapter has established the conceptual and methodological underpinnings necessary for the rest of the thesis. We first introduced the core definitions and diverse application domains of CPS, emphasizing their critical role in sectors such as transportation, healthcare, energy, and manufacturing. These domains not only exemplify the technological impact of CPS but also reflect the increasing complexity and diversity of verification needs.

To manage this complexity, CPS engineering relies on a range of modeling languages, from high-level semi-formal notations such as SysML to low-level formalisms like BIP and timed automata. While high-level languages support early design and communication among stakeholders, they often lack the formal precision required for rigorous verification. Conversely, formal methods offer strong correctness guarantees but are less accessible in early development phases. We reviewed hybrid strategies that attempt to bridge this gap through model transformations, semantic extensions, and co-modeling frameworks.

Focusing on the verification perspective, this chapter identified three fundamental challenges that structure the scope of this thesis: verifying the structural consistency of system architectures, analyzing dynamic interaction correctness, and reasoning about requirement satisfaction, including fuzzy or imprecise requirements. Our review of the state of the art in each of these areas reveals significant gaps in current methodologies, especially regarding early and integrated verification within standard modeling environments.

These limitations motivate the core contributions of this thesis, which aim to (i) enable behavior-independent structural consistency checking in SysML-based designs, (ii) support formal interaction modeling and verification through a dedicated SysML-to-BIP transformation, and (iii) facilitate the evaluation of fuzzy requirements through native SysML v2 constructs and simulation-based reasoning. The next chapter introduces the modeling foundations that underpin these contributions.

Chapter 3

Modeling Foundations

3.1 Introduction

This chapter presents the modeling formalisms that underpin the methodologies proposed in this dissertation. Cyber-Physical Systems (CPS) require rigorous and expressive modeling languages to capture their structural, behavioral, and non-functional aspects. Model-Based Systems Engineering (MBSE) plays a central role in this context, with the Systems Modeling Language (SysML) [44] serving as a standard foundation.

However, SysML v1 exhibits limitations in terms of formal semantics. This has motivated the adoption of complementary or enhanced formalisms. The five main modeling and formalization foundations introduced in this chapter are:

- **SysML v1** [44], the first standardized system modeling language, widely used in MBSE but lacking native execution semantics.
- **SysML v2** [45], a recent evolution of the language that offers more coherent syntax, formal textual notation, and improved extensibility.
- **BIP (Behavior, Interaction, Priority)** [5], a formal framework that supports modular specification and rigorous verification of component-based systems.
- **Fuzzy logic** [59], a reasoning paradigm used to express and verify non-functional and soft requirements that are inherently imprecise or context-dependent.
- **OCaml** [33], a statically typed functional programming language known for its strong type system, pattern matching, and support for symbolic computation, commonly used in formal methods and the development of verification tool.

Each of these formalisms is presented in the following sections, focusing on their modeling capabilities, formal properties, and relevance to CPS verification. This chapter serves as a technical foundation for the contributions developed in the remainder of this dissertation.

3.2 SysML v1: Modeling Principles and Key Diagrams

The Systems Modeling Language [44] (SysML) is a general-purpose modeling language developed by the Object Management Group (OMG) as a UML [46] profile to support system-level engineering. SysML v1 has been widely adopted in industrial Model-Based Systems Engineering (MBSE) practices due to its flexibility and its graphical notations to represent structure, behavior, and requirements.

SysML v1 defines nine types of diagram, organized into structural, behavioral, requirement, and parametric categories. This section presents the key modeling constructs relevant to the structural consistency verification method introduced in Chapter 4, focusing primarily on block definition diagrams (BDD), internal block diagrams (IBD), and requirement diagrams.

3.2.1 Blocks and Interfaces

Blocks are the core modeling construct in SysML. They represent system entities that can be physical, logical, or software components. A block may include properties (attributes), operations, ports, and internal parts. SysML distinguishes between:

- **Elementary blocks**, which are atomic units that are not further decomposed.
- **Composite blocks**, composed of internal parts and used to model hierarchical architectures.

In SysML, interface blocks are used to formally specify the set of operations, properties, and signals that a block provides or requires, thereby enabling precise modeling of interaction points. These are typically used to type proxy ports, which serve as externally visible connection points. Proxy ports support abstraction and encapsulation by exposing only selected features of internal elements.

3.2.2 Block Definition Diagram (BDD)

The Block Definition Diagram (BDD) specifies the static architecture of the system by defining blocks and their relationships, such as composition, inheritance, and associations. It provides a high-level view of the system's structure and supports reuse and modularity.

Figure 3.1 illustrates a simple BDD example, where a composite block B aggregates sub-blocks B1, B2, and B3.

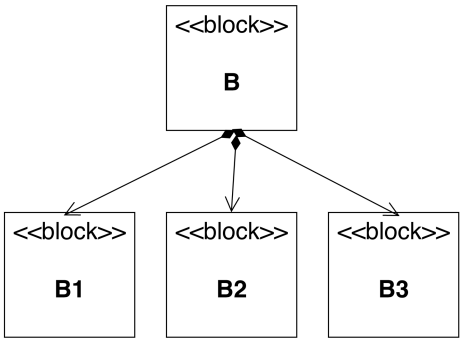

Fig. 3.1: Example of a Block Definition Diagram (BDD) for a composite system.

3.2.3 Internal Block Diagram (IBD)

The Internal Block Diagram (IBD) describes the internal composition of a block, showing its subparts and how they are interconnected. It distinguishes between: Assembly connectors, which link internal parts together; Delegation connectors, which route interactions from a composite block's interface to internal parts, or vice versa. These connectors help formalize the architecture of the system and enable the traceability of interactions through different layers of decomposition.

Figure 3.2 illustrates the use of delegation and assembly connectors in a composite block structure.

3.2.4 Requirement Diagram

Requirement diagrams capture the system requirements in textual form and describe their relationships with other model elements, such as blocks or test cases. SysML supports:

- **Atomic requirements**, which represent indivisible specifications;
- **Composite requirements**, which aggregate or refine multiple sub-requirements.

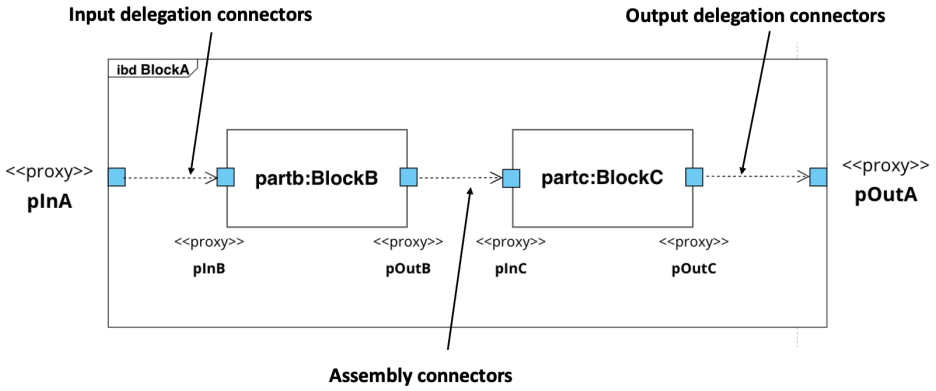

Fig. 3.2: Example of an Internal Block Diagram (IBD) showing connectors and port delegation.

These diagrams are essential for maintaining traceability and ensuring that the system design meets all specified requirements.

Figure 3.3 shows the hierarchical organization of the composite and atomic requirements.

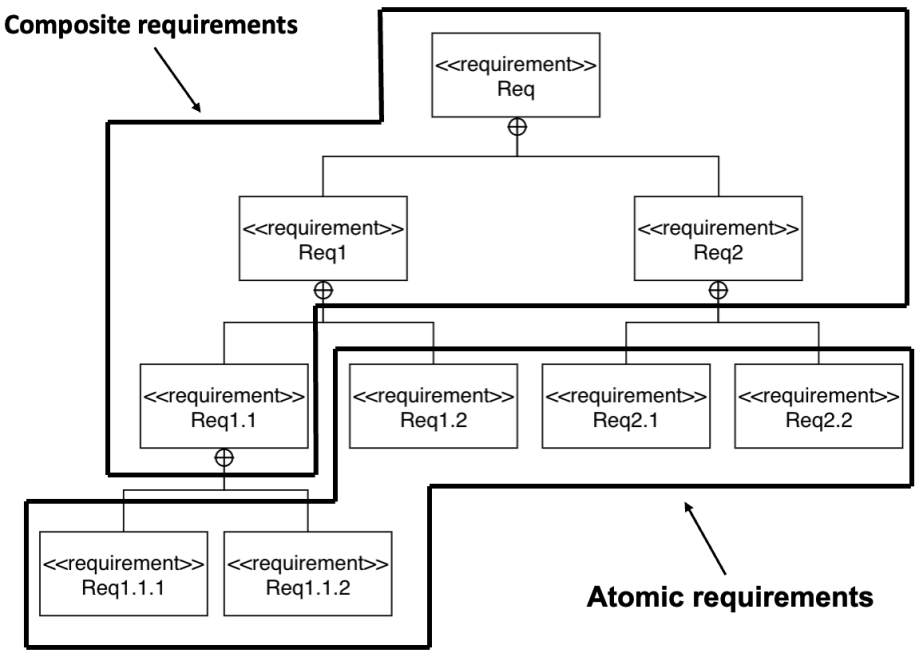

Fig. 3.3: Composite and atomic requirements in SysML v1.

3.2.5 Overview of Other Diagram Types in SysML v1

Although the BDD, IBD, and requirement diagrams are central to the structural consistency analysis proposed in this dissertation, SysML v1 includes additional diagrams for modeling behavior, logic, and constraints. A brief summary follows:

- **Activity Diagrams:** Describe control and data flows, supporting functional decomposition.
- **Sequence Diagrams:** Show interactions among parts via ordered message exchanges.
- **State Machine Diagrams:** Define the reactive behavior of blocks using state transitions.
- **Use Case Diagrams:** Model system functionality from the user's perspective.
- **Parametric Diagrams:** Express mathematical constraints among parameters, useful for trade-off analysis.
- **Package Diagrams:** Organize model elements into packages to support modularity and reuse.

These diagrams enhance the modeling expressiveness of SysML and are widely used in MBSE workflows.

3.3 SysML v2: Foundations and Modeling Constructs

SysML v2 [45] is the latest evolution of the Systems Modeling Language developed by the Object Management Group (OMG) to address the expressiveness and semantic limitations of SysML v1. This new version introduces a refined metamodel, a formal textual syntax, and enhanced semantic precision to support automation, traceability, and tool interoperability throughout the system engineering lifecycle.

Compared to its predecessor, SysML v2 offers several key improvements:

- A dual representation, textual and graphical, enabling seamless integration into modern development workflows.
- A modular and declarative structure based on definitions and instantiations to promote reuse and clarity.

- Formal typing and constraint mechanisms for improved consistency and verification of the model.
- Native support for expressing requirements, constraints, and behavioral logic within a unified framework.

In this dissertation, SysML v2 is widely used to model both the structure and behavior of Cyber-Physical Systems (CPS), enabling early verification and model transformation. The following subsections summarize the core constructs employed throughout this work, using both textual syntax and illustrative figures.

3.3.1 Part and Attribute Definitions

The entities of the system are defined using the **part def** construct, where structural features such as `attribute` are declared. Concrete usage within a model is instantiated by the **part** keyword. This separation promotes modularity and supports reuse. Listing 3.1 shows a simple part definition and instantiation. The corresponding graphical representation is given in Figure 3.4.

Listing 3.1: Part definition and instantiation

```
part def Heater {
attribute power: Real;
}
part mainHeater: Heater {
ref power = 1500.0;
}
```

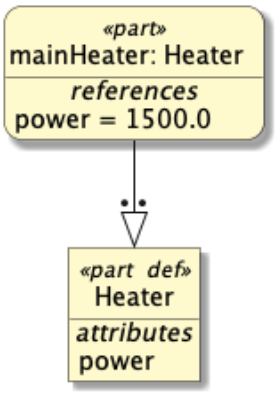

Fig. 3.4: Graphical representation of part and attribute definition.

3.3.2 Port and Connection Modeling

Communication between parts is defined using **port def** and **connection def**. These constructs allow specifying directionality (e.g., input, output), types of exchanged data, and synchronization constraints. Listing 3.2 provides a textual example, while Figure 3.5 illustrates the graphical equivalent of such a connection between components.

Listing 3.2: Example of port and connection definition

```
port def TempPort {
out tempReading: Real;
}
connection def TempLink {
end sensor: TempPort;
end controller: TempPort;
flow sensor.tempReading to controller.tempReading;
}
```

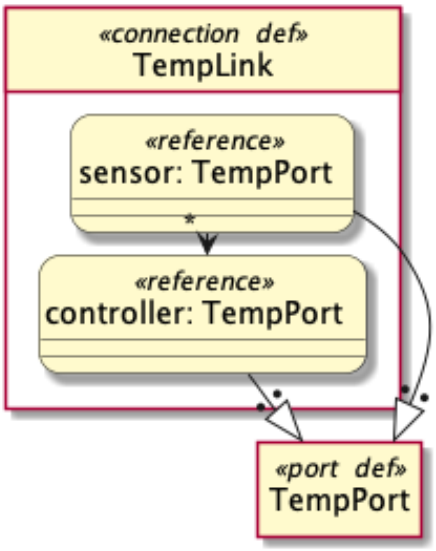

Fig. 3.5: Example of port and connection modeling in SysML v2.

3.3.3 Requirement and Constraint Modeling

Requirements are expressed using the **requirement** construct, which can include attributes, documentation, and embedded constraint logic. Reusable computations can be defined using **calc def**. An example combining a requirement, constraint and supporting calculation is shown in Listing 3.3, and its graphical form is shown in Figure 3.6.

Listing 3.3: Requirement and constraint with calculation

```
private import ScalarValues::*;
calc def sum {
in a: Real;
in b: Real;
return r: Real;
a + b
}

requirement safeTemp {
doc /* Total temperature must stay below 100 */
attribute limit: Real = 100.0;
constraint sumSensors { sum(sensor1.temp, sensor2.temp) < limit }
}
```

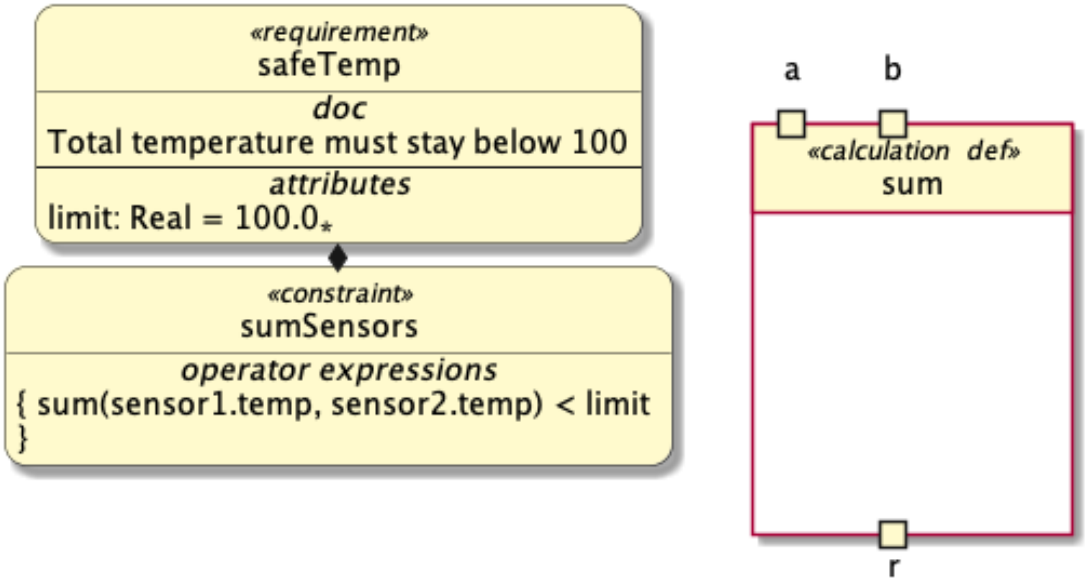

Fig. 3.6: Graphical representation of requirements and constraint modeling.

3.3.4 State Machines and Behavioral Modeling

SysML v2 supports behavioral modeling through constructs like **state def** and **transition**. These allow engineers to define reactive system logic, where events or guard conditions trigger transitions between states. Listing 3.4 and Figure 3.7 present a simple state machine example for a thermostat component.

Listing 3.4: Basic state machine example

```
state def ThermostatStates {
state Off;
state Heating;
transition turnOn
first Off
accept userRequest
then Heating;
}
```

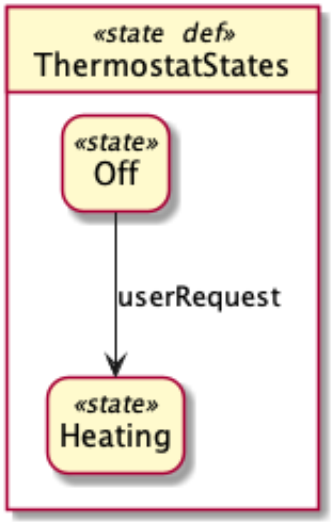

Fig. 3.7: Example of state machine modeling in SysML v2.

3.3.5 Satisfaction Links and Traceability

The `satisfy` relationship is used to link structural elements (e.g., parts) to the requirements they are designed to fulfill. This mechanism enhances traceability and completeness of the model. Listing 3.5 and Figure 3.8 shows an example where the `mainHeater` part satisfies the `safeTemp` requirement.

Listing 3.5: Satisfy relationship example

```
satisfy safeTemp by mainHeater;
```

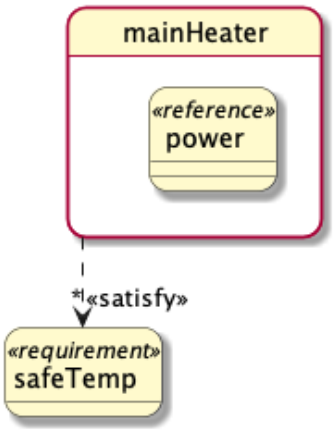

Fig. 3.8: Example of Satisfy relationship in SysML v2.

3.4 Behavior, Interaction, Priority (BIP) Framework

The *Behavior, Interaction, Priority (BIP)* framework [5] provides a rigorous, component-based modeling approach for designing and verifying complex heterogeneous systems, particularly Cyber-Physical Systems (CPS). Its key innovation lies in the separation of concerns into three orthogonal layers *Behavior*, *Interaction*, and *Priority*, which allows a modular and compositional modeling style while preserving formal semantics. This section reviews the core constructs of BIP (see Figure 3.9 for the three-layered BIP representation).

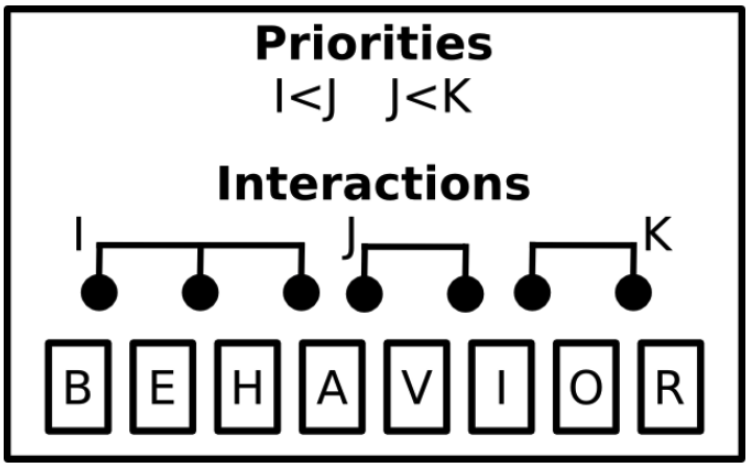

Fig. 3.9: The three-layered BIP representation: behavior, interaction, and priority [55].

3.4.1 Atomic Components and the Behavior Layer

The behavior layer of a BIP model encapsulates the internal logic of *atomic components*, often referred to as *atoms*. Each atomic component is a labeled transition system (LTS), defined by a finite set of states (called *places*), transitions guarded by boolean expressions and executable actions.

Communication with the environment occurs via explicitly declared *ports*. Each port may optionally carry data and participate in interactions defined in the next layer. The example in Listing 3.6 shows an atomic component with two states and a transition that is triggered by the reception of a signal on a port:

Listing 3.6: Example of a BIP Atomic Component

```
atom type Controller() {
  export port CommandPort_t commandIn;
  place Idle, Active;
  initial to Idle;
  on commandIn from Idle to Active do {
    printf("Controller␣activated.\n");
  }
}
```

This modular approach allows independent definition and reuse of behavior across different configurations, while maintaining clear execution semantics.

3.4.2 Connectors and the Interaction Layer

While atomic components define isolated behaviors, the interaction layer specifies how components coordinate and exchange data. This coordination is achieved through *connectors*, which are stateless entities that define sets of *interactions*, that is, synchronized executions involving a subset of component ports.

Each connector formalizes a pattern of synchronization by specifying:

- The ports involved in the interaction.
- A distinction between **triggers** (denoted with a quote ’ symbol) and **synchronous ports**:
  - A trigger initiates the interaction.
  - Synchronous ports must participate if the interaction is selected.

- Optional **guards** to restrict the execution of interactions based on port variables.
- Optional **data transfer functions**, divided into `up` and `down` blocks:
  - `up`: computes data values from component ports and stores them in connector-local variables.
  - `down`: distributes computed values back to the port variables involved in the interaction.

BIP supports different interaction styles, including:

- **Rendez-vous**: a strong synchronization pattern where *all* ports in the interaction must be enabled simultaneously.
- **Broadcast**: a weaker form where a single trigger activates an interaction with a subset of available receivers.

The following example (Listing 3.7) defines a broadcast-style connector. Here, the port `sender` is a trigger (noted with a quote), while `rec1` and `rec2` are optional synchrons. The interaction occurs when `sender` is enabled; the receivers participate only if they are also enabled.

Listing 3.7: Broadcast Connector in BIP

```
connector type Broadcast(BroadcastPort_t sender, CommandPort_t rec1,
   CommandPort_t rec2) {
define sender' rec1 rec2;

on sender rec1 rec2 down {
rec1.commandSignal = sender.commandSignal;
rec2.commandSignal = sender.commandSignal;
}
}
```

This mechanism supports flexible coordination structures while preserving modularity. By externalizing synchronization from the components to the connectors, BIP promotes separation of concerns between computation (behavior) and coordination (interaction).

3.4.3 Priority Layer and Conflict Resolution

When multiple interactions are enabled simultaneously, the *priority layer* is responsible for defining which one to execute. This layer defines a strict partial order over the enabled interactions using rules, as illustrated in Listing 3.8.

Listing 3.8: Static Priority Rule in BIP

```
priority scheduler brdABC:A.p,B.p < brdDEF:D.p,E.p,F.p
```

Dynamic priorities may depend on runtime conditions or guards, allowing the model to capture scheduling policies or context-sensitive control strategies. By isolating priority concerns in a dedicated layer, BIP avoids intermixing control logic with component behavior, promoting a clean separation of concerns and easier model evolution.

3.4.4 Execution Semantics and Tool Support

BIP models can be compiled and executed using the `bipc` compiler and simulation engine. The engine supports step-by-step execution, deadlock detection, invariant checking, and formal scheduling analysis. At runtime, enabled interactions are evaluated, and the one with the highest priority is selected for execution, respecting the compositional semantics.

3.5 Fuzzy Logic: A Foundation for Soft Requirement Modeling

Fuzzy logic, first introduced by Zadeh in 1965 [59], extends classical Boolean logic by allowing truth values to continuously range within the interval $[0, 1]$. This paradigm is especially valuable in engineering contexts where requirements are often vague, qualitative, or inherently imprecise, for example, “low energy consumption” or “acceptable temperature range.” In such cases, strict Boolean thresholds fail to capture real-world subtleties, and fuzzy logic provides a principled mechanism for modeling partial satisfaction.

3.5.1 Membership Functions

At the core of fuzzy logic lies the concept of a *membership function*, which maps a crisp input value to a degree of satisfaction within the $[0,1]$ interval. Several families of membership functions exist:

- **Triangular** and **trapezoidal**, commonly used for their simplicity and efficiency;
- **Gaussian** and **sigmoidal**, offering smoother transitions;
- **Piecewise or custom-defined** shapes, which allow domain-specific modeling.

In this dissertation, we primarily use trapezoidal membership functions because of their interpretability and computational tractability in system-level design. A trapezoidal function is parameterized by four values (a,b,c,d), as illustrated in Equation 3.1.

$$\mu(x) = \begin{cases} 0 & \text{if } x \leq a \text{ or } x \geq d \\ \frac{x-a}{b-a} & \text{if } a < x < b \\ 1 & \text{if } b \leq x \leq c \\ \frac{d-x}{d-c} & \text{if } c < x < d \end{cases} \tag{3.1}$$

The four parameters (a,b,c,d) determine the shape of the trapezoidal membership function as follows:

- a: the **start point**, where the membership begins to increase from 0.
- b: the **left top point**, where the membership reaches 1 for the first time.
- c: the **right top point**, where the membership stops being 1 and starts to decrease.
- d: the **end point**, where the membership goes back to 0.

Figure 3.10 shows an example of a trapezoidal membership function.

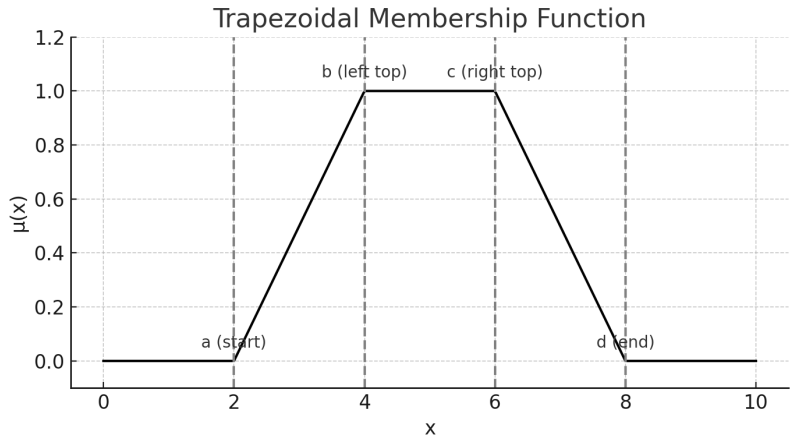

Fig. 3.10: Illustration of a trapezoidal membership function.

3.5.2 Example: Fuzzy Temperature Requirement

Consider a system requirement stating that the "room temperature should be approximately between 20°C and 25°C," with tolerances ranging from 18 ° C to 27 ° C. Table 3.1 shows sample evaluations of the membership function in this configuration.

Table 3.1: Trapezoidal Membership Evaluation for a Temperature Requirement

Temperature (x)	$\mu(x)$	**Explanation**
17°C	0	Outside tolerance ($x \leq a$)
19°C	0.5	Linear rise: $\frac{19-18}{20-18} = \frac{1}{2}$
22°C	1	Full satisfaction ($b \leq x \leq c$)
26°C	0.5	Linear decline: $\frac{27-26}{27-25} = \frac{1}{2}$
28°C	0	Outside tolerance ($x \geq d$)

This example illustrates how fuzzy logic captures graded satisfaction rather than binary classification, making it suitable for expressing soft constraints in system engineering.

3.6 OCaml

OCaml [33] is a functional programming language known for its strong static type system, expressiveness, and formal rigor. Widely used in the academic and industrial communities for developing reliable software. Its type system ensures that many

classes of errors are detected at compile time, which contributes to the reliability and correctness of formal specifications.

In addition to its type safety, OCaml emphasizes simplicity and readability [39]. Its concise syntax and powerful pattern matching constructs make it well-suited for expressing abstract syntax trees, recursive functions, and domain-specific representations in a clear and maintainable way. These characteristics make OCaml a natural choice for developing formal definitions and analysis tools in model-based engineering and verification.

In this section, we present the essential OCaml constructs relevant to our modeling context. We focus on defining types, records (structures), and functions, including examples of recursion, which is a key feature of the language.

3.6.1 Type in OCaml

In OCaml, we define custom types using the keyword **type**. For example, to define a type for a simple arithmetic expression. We use the following syntax:

```
type expr = | Int of int | Add of expr + expr | Mul of expr * expr
```

In this example, the expr type can represent integer values (Int of int), addition expressions (Add of expr + expr), and multiplication expressions (Mul of expr * expr).

3.6.2 Structure in OCaml

Structures in OCaml are defined using records. We illustrate this with a structure representing a person:

```
type person = {  name : string; age : int; }
```

This `person` structure includes fields for the person's name and age.

3.6.3 Function in OCaml

In OCaml, functions play a crucial role in structuring code and performing computations. To define a function in OCaml, use the keyword **fun** followed by the parameter and the expression to be computed. For instance, let us define a simple function that adds two numbers:

```
let add_numbers x y = x + y
```

In this example, `add_numbers` is a function that takes two parameters (`x` and `y`) and returns their sum. When a function calls itself during its execution, it is termed a recursive function. Recursion is a powerful concept that simplifies the code for operations that involve repeated or nested computations. Let's consider a classic example of a recursive function: calculating the factorial of a number.

```
let rec factorial n =  if n <= 1 then 1  else n * factorial (n - 1)
```

In this example, `factorial` is a recursive function that calculates the factorial of a given number `n`. It continues to call itself until it reaches the base case (`n <= 1`), at which point it returns 1. The final result is the product of all numbers from 1 to `n`.

3.7 Conclusion

This chapter has laid the theoretical and technical foundations for the modeling and verification methodologies presented in this dissertation. By introducing SysML v1 and v2, the BIP framework, fuzzy logic, and the OCaml language. we have covered a broad spectrum of modeling capabilities essential for the design and analysis of modern Cyber-Physical Systems (CPS).

Together, these modeling foundations provide a coherent methodological basis for the verification approaches developed in this thesis. Each subsequent chapter builds on these foundations to address specific verification challenges, ranging from structural consistency and interaction correctness to soft-requirements evaluation, thus contributing to the reliable design of CPS.

Part III

Research Contributions

Chapter 4

Structural Consistency Verification of SysML Models for Cyber-Physical Systems

4.1 Introduction

The structural integrity of a system model is a foundational aspect of reliable Cyber-Physical Systems (CPS) development. As these systems become increasingly complex, ensuring consistency between architectural components, interfaces, and requirements becomes a critical challenge in early design phases.

Model-Based Systems Engineering (MBSE) provides formal tools to manage this complexity, with SysML as a widely adopted language for specifying system structure, behavior, and requirements. SysML diagrams such as Block Definition Diagrams (BDD) and Internal Block Diagrams (IBD) enable engineers to represent the system's architectural elements. However, most modeling environments focus on syntax checking and do not verify the underlying structural semantics that ensure proper integration of components.

This chapter addresses the problem of structural consistency checking in SysML models. Rather than relying solely on visual inspection or syntactic validation, we aim to formalize the static semantics of the model and verify that the architecture adheres to a set of structural correctness rules. These include interface compatibility, valid connections, and traceable requirement allocations.

We begin by analyzing how structure is expressed in SysML and proceed to define a formal representation of its main modeling constructs. We then formulate consistency

rules and describe how they can be evaluated automatically. The method is illustrated through an industrial case study of an autonomous vehicle system. We conclude by discussing how the approach adapts to SysML v2 and supports scalable verification of CPS models.

4.2 Proposed approach

To provide clarity on our approach, we classify systems based on the complexity of their decomposition and assembly processes:

Single-Step Assembly (Simple System): In scenarios where the system to be built consists of straightforward and well-defined components, the assembly can be accomplished in a single step. These systems, referred to as "simple systems," involve directly assembling elementary components to meet the abstract specification (see Figure 4.1). This approach is typically applicable when the system's functionality and interactions are relatively uncomplicated, allowing immediate composition without intermediate refinement stages.

Multi-Step Assembly (Complex System): Conversely, more intricate systems, referred to as "complex systems" require a multi-step assembly process (see Figure 4.2). These systems involve the definition of abstract sub-components, which are subsequently refined by assembling other components. This hierarchical and iterative refinement process ensures that each sub-component meets its specified requirements before being integrated into the final system.

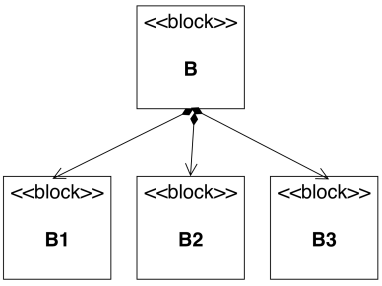

Fig. 4.1: Simple system

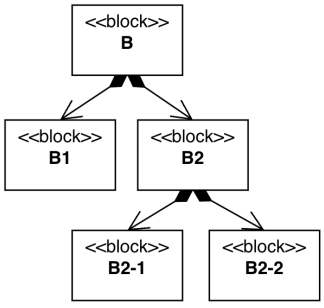

Fig. 4.2: Complex system

Definition 1 (Refinement of a composite SysML block with a composition of blocks)**.** Let B be an abstract block described with the block definition diagrams BDD_B and the internal block diagram IBD_B. Let $B_1, ..., B_n$ be the sub-blocks of B, described with their SysML models (see Figure 4.3). The composition of $B_1, ..., B_n$ refines the

abstract block A if, there is consistency at the structural level between the composition of the sub-blocks B_i and the composite block A.

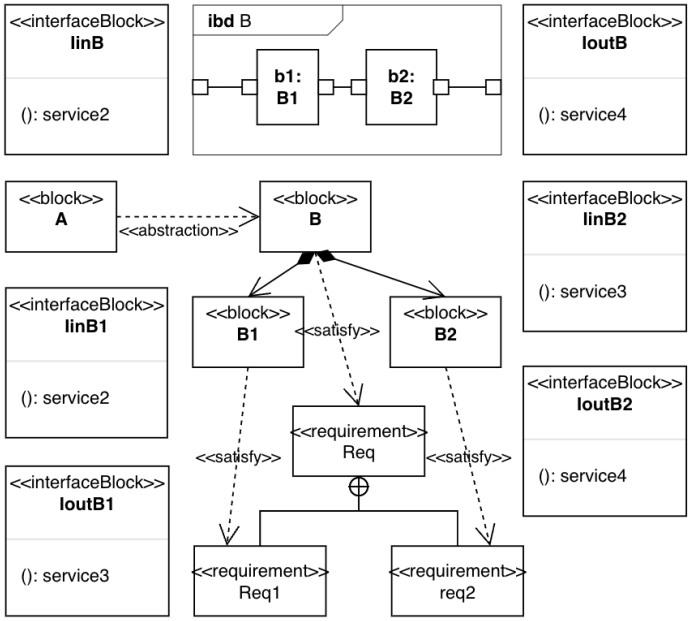

Fig. 4.3: The proposed structural architecture to model the abstract component

To verify the consistency, we consider the following three conditions:

- **Condition 1:**
  For every block B in `Set_Blocks`, such that B is composed of a set of blocks $\{B_1, \ldots, B_n\}$ and provides a set of services $\{S_1, \ldots, S_m\}$, then the blocks $\{B_1, \ldots, B_n\}$ must collectively provide at least the services $\{S_1, \ldots, S_m\}$.

- **Condition 2:**
  For every block B in `Set_Blocks`, such that B is composed of a set of blocks $\{B_1, \ldots, B_n\}$ and requires a set of services $\{S_1, \ldots, S_m\}$, then the sub-blocks $\{B_1, \ldots, B_n\}$ must collectively require at most the services $\{S_1, \ldots, S_m\}$.

- **Condition 3:**
  For every block $B \in$ `Set_Blocks`, let $\{B_1, \ldots, B_n\}$ be its sub-blocks. If B satisfies a composite requirement R composed of atomic requirements $\{R_1, \ldots, R_m\}$, then each atomic requirement R_i must be satisfied by at least one sub-block B_j:

$$\forall R_i \in \{R_1, \ldots, R_m\},\ \exists B_j \in \{B_1, \ldots, B_n\} \text{ such that } B_j \models R_i.$$

In order to confirm the first and second conditions, we will make use of the internal block diagrams and the connections between different parts via their respective ports. To verify the first condition, as shown in Figure 4.4, we need to ensure that the services provided by the interface block that types port PInA are included in the services provided by the interface block that types port PInB.

$$PInA \subseteq PInB$$

Similarly, for condition 2, we need to verify that the services offered in the interface block that types port PoutC are included in the service offered by the interface block that types port PoutA.

$$POutC \subseteq POutA$$

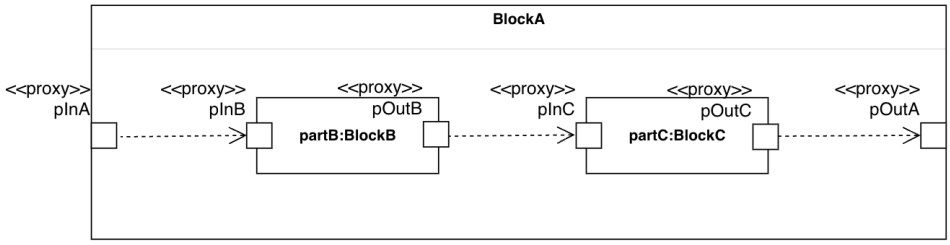

Fig. 4.4: Simple connection

When multiple input delegation ports are present, as illustrated in Figure 4.5, the services offered by the interface block typing port `PInA` must be included in those provided by the interface block typing the union of ports `PInB` and `PInC`.

$$PInA \subseteq PInB \cup PInC$$

Another relevant configuration involves multiple output delegation ports, as shown in Figure 4.6. In this case, the services provided by the interface blocks typing ports `POutB` and `POutC` must be included in those provided by the interface block typing port `POutA`.

$$POutB \cup POutC \subseteq POutA$$

Suppose there is an internal part offering a service required by another part linked by output delegation to the composite block. In that case, we remove the required services before verifying the inclusion, as shown in Figure 4.7. In this case, we have to verify that the services offered in the interface block, which type port POutC, minus the services offered in the interface block, which type port PInB, are included in the service offered by the interface block, which type port POutA.

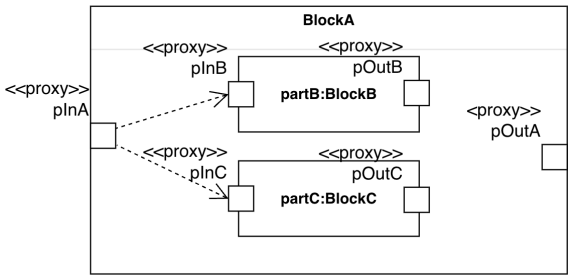

Fig. 4.5: Multiple In delegation connections

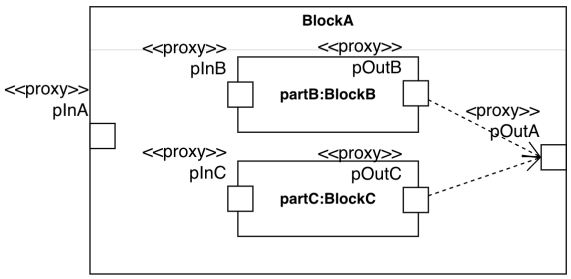

Fig. 4.6: Multiple Out delegation connections

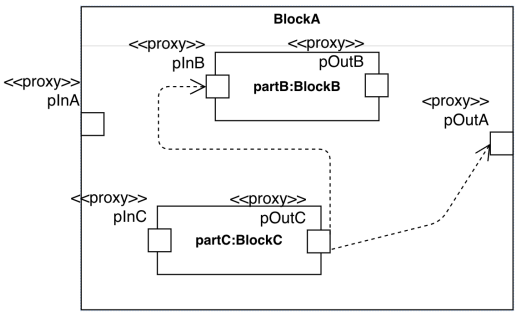

Fig. 4.7: Delegation connection with composition connection

$$POutC - PInB \subseteq POutA$$

Based on these conditions, we can deduce that checking the consistency of delegationINs is different from checking delegationOUTs.

As SysML lacks formalism and to automate verification, we need to formally define the SysML models specifying the architecture of a CPS. The proposed formal specification is defined in 4.3.

4.3 Formal specification of SysML architectural diagrams

4.3.1 Syntax of SysML structural diagrams

We propose a formal model for specifying SysML diagrams to analyze the architecture of a CPS and verify its structural consistency. We propose formalizing the block, interface block, port, requirement, block definition diagram, and requirement diagram.

Definition 2 (Formal Requirement)**.** A formal requirement is defined by the following type:

```
type requirement = { id : string ; name : string ; text : string; }
```

- *id* is the identifier of the requirement.

- *name* is the name of the requirement.

- *text* is the textual description of the requirement.

To model the requirement diagram, we first need to define a structure we will refer to as coupleReq, which is defined by a pair that associates a requirement with a list of requirements.

```
type coupleReq = requirement * requirement list
```

Definition 3 (Formal Requirement diagram)**.** A formal requirement diagram is defined by the following type:

```
type reqDiagram = { iR : requirement ; sR : requirement list ; relC : coupleReq list }
```

- iR is the name of the initial requirement of the diagram.

- sR is the finite set of requirements in the diagram.

- $relC \subseteq SR * P(SR)$ is the composition relation, where $P(SR)$ is the set of subsets of SR.

Definition 4 (Formal Interface block)**.** A formal model for a SysML interface block IB is defined by the following type:

```
type interfaceBlock = { nameIB : string ; opIB : string list}
```

- $namexIB$ is the name of the interface block,

- $opIB$ is the set of functions that the block requires or offers.

Definition 5 (Formal Port SysML)**.** The formal SysML port is defined by the following type:

```
type port = {nameP : string ; typeP : blocInterface ; directionP : string;
             connectorInterneSBC : string list }
```

- *nameP* is the name of the port.
- *typeP* is the interface block that types the port.
- *directionP* is the direction of the port (in or out).
- *connectorInterneSBC* is the set of functions required by the block which contains this port and which are satisfied internally by composition.

To model the block, we first define `couplePorts` as a pair that associates two ports. This structure is used to represent all port connections.

```
type couplePorts = port * port
```

Definition 6 (Formal SysML Block)**.** The formal SysML block B is defined by the following type:

```
type block = { name : string ; pinB : port ; poutB : port ; connectorsInB : port list;
               connectorsOutB : couplePorts list; sRB : requirement list}
```

- *name* is the name of the block B.
- *pinB* is the In port of the block B.
- *poutB* is the Out port of the block B.
- $connectorsInB = \{(p_i, p_j)\ such\ as\ (p_i, p_j) \in Ports \times Ports \wedge p_i.direction = p_j.direction = IN\}$, the set of connectors that link the ports of the sub-blocks (parts) with the ports of the composite block. They are called delegation IN connectors.
- $connectorsOutB = \{(p_i, p_j)\ \ such\ \ as\ \ (p_i, p_j) \in Ports \times Ports \wedge p_i.direction = p_j.direction = OUT\}$, the set of connectors that link the ports of the sub-blocks (parts) with the ports of the composite block. They are called delegation OUT connectors.
- sRB is the set of requirements that the block satisfies.

To model the Block Definition Diagram, we first introduce `coupleBlock`, a structure that associates a block with a set of sub-blocks. This allows us to explicitly represent the hierarchical decomposition of the system.

type coupleBlock = block * block list

Definition 7 (Formal Block Definition Diagram)**.** The formal BDD of a system S is defined by the following type:

type fBDD = {iBs : block ; sBs : block list ; suBs : coupleBlock list }

- iBs is the main block.

- sBs is the set of all blocks in the system.

- $suBs$ the set of couples that connect each block to its sub-blocks.

4.3.2 Static semantics

Static semantics define the meaning of constructs that are syntactically correct by enforcing additional rules beyond grammar. While syntax ensures that a model or program conforms to the structural rules of the language, such as the correct arrangement of elements or keywords, static semantics verify that these well-formed constructs are also semantically valid. This includes checking that every variable or model element is declared before use, that data types are compatible in expressions, and that component connections respect interface specifications. In the context of SysML, static semantics ensure, for example, that a required port is properly connected to a compatible provided port, or that a requirement allocated to a block is consistent with the overall model structure. These checks are performed at design time and help identify inconsistencies or modeling errors that cannot be detected through syntax alone. Ultimately, static semantics ensure the internal validity of the model by confirming its adherence to context-specific constraints.

We define three static semantic rules, denoted S_1, S_2, and S_3, as predicates parameterized by a given block b. Each rule verifies a specific aspect of structural consistency within the block hierarchy.

We begin with S_1, which verifies the consistency of service delegation through input ports of composite blocks.

$$S_1(b) \stackrel{\text{def}}{=} subB(b) \neq \emptyset \ \wedge \ incluList\,(b.pinB.\text{typeP.opIB},\ union\,(connectorsInB(b))) \tag{4.1}$$

Where:

- $subB(b)$: the set of sub-blocks of b.
- $b.pinB$: the input port of b through which it declares offered services.
- $b.pinB$.typeP.opIB : the services (operations) declared in the interface block that types b's input port.
- $connectorsInB(b)$: the set of input delegation connectors linking sub-blocks to b's input port.
- $union(connectorsInB(b))$: the union of all services offered via these input connectors.
- $incluList(L_1, L_2)$: true if all elements of L_1 are included in L_2.

The second rule, S_2, ensures that the sub-blocks of a composite block do not require more services than those declared as required by the composite block itself. This rule guarantees that any external services needed by the sub-blocks are explicitly declared at the composite level, assuming internal services are already satisfied.

$$S_2(b) \stackrel{\text{def}}{=} subB(b) \neq \emptyset \ \wedge incluList\Big(soustractionElement\big(union(b.connectorsOutB,\ b.connectorInterneSBC\big), b.poutB.\text{typeP.opIB}\Big) \tag{4.2}$$

Where:

- $b.connectorsOutB$: the set of output delegation connectors linking sub-block ports to b's output port.
- $union(b.connectorsOutB)$: the union of all services required via these output connectors.
- p.connectorInterneSBC : the set of services of port p already satisfied internally by composition.

- $soustractionElement(L_1, L_2)$: the set of elements in L_1 that are not in L_2.
- $b.poutB$.typeP.opIB : the services declared as required by b's output port.

The third rule, S_3, ensures the refinement of requirements: all atomic requirements associated with a composite block must be satisfied by its sub-blocks. This rule guarantees the traceability and distribution of requirements across the block hierarchy.

$$S_3(b, reqD) \stackrel{\text{def}}{=} subB(b) \neq \emptyset \wedge incluList\Big(AtomicReq(b, reqD), \bigcup_{sb \in subB(b)} AtomicReq(sb, reqD)\Big) \tag{4.3}$$

Where:

- $reqD$: the formal requirement diagram of the system.
- $AtomicReq(b, reqD)$: the set of atomic requirements associated with a block (or sub-block) b in $reqD$.

To systematically verify the structural consistency conditions previously defined, we propose Algorithms 1 and 2. Algorithm 1 checks conditions S_1 and S_2 for each composite block, while Algorithm 2 verifies condition S_3 against each block and its associated requirements.

Algorithm 1: Service Consistency

Input: Initial block `bc`, system BDD `bdd`
Output: Boolean result indicating whether `bc` and its sub-blocks satisfy conditions S1 and S2

Let $Sub_B \leftarrow Sub(bc, bdd)$;
Initialize $result \leftarrow$ `true`;
if $S1(bc)$ ***and*** $S2(bc)$ **then**
  $result \leftarrow$ `true`;
end
else
  $result \leftarrow$ `false`;
end
while $Sub_B \neq \emptyset$ **do**
  Select $b_i \in Sub_B$;
  if $Sub(b_i, bdd) \neq \emptyset$ **then**
    $result \leftarrow result$ **and** $serviceConsistency(b_i, bdd)$;
  end
  $Sub_B \leftarrow Sub_B \setminus \{b_i\}$;
end
return $result$;

Algorithm 2: Requirement Consistency

Input: Initial block `bc`, system BDD `bdd`, requirement diagram `reqD`
Output: Boolean result indicating whether requirement consistency (S3) holds
$Sub_B \leftarrow Sub(bc, bdd)$;
$finalListBc \leftarrow AtomicReq(bc)$;
$result \leftarrow$ `true`;
if $finalListBc = \emptyset$ **then**
  $result \leftarrow$ `false`;
end
while $Sub_B \neq \emptyset$ **do**
  Select $b_i \in Sub_B$;
  $finalListBi \leftarrow AtomicReq(b_i)$;
  $finalListBc \leftarrow finalListBc \setminus finalListBi$;
  if $Sub(b_i, bdd) \neq \emptyset$ **then**
    $result \leftarrow result$ **and** $requirementConsistency(b_i, bdd, reqD)$;
  end
  $Sub_B \leftarrow Sub_B \setminus \{b_i\}$;
end
if $result =$ ***true and*** $finalListBc = \emptyset$ **then**
  return *true*;
end
else
  return *false*;
end

4.4 Transformation tool

Eclipse Papyrus is an open-source modelling tool in the larger Eclipse Modeling Project. It provides a platform for creating various models, particularly those based on standard modelling languages such as the Unified Modeling Language (UML) and its profiles, SysML (Systems Modeling Language). Eclipse Papyrus typically saves SysML diagrams and models in the XMI (XML Metadata Interchange) format produced by the Object Management Group (OMG). The XMI format allows the interchange of objects and models through an XMI-formatted file.

Figures 4.8, 4.9, and 4.10 show an example of how SysML element data is represented in the XMI file. Figure 4.8 illustrates how requirements are represented. Figure 4.9 shows how the requirements hierarchy is represented. Figure 4.10 shows how the mapping between blocks and requirements is represented.

We developed a tool to automate the transformation of the SysML model into the proposed formal specification. However, some naming conventions must be respected during the modelling process with papyrus.

- **Convention 1:** Block names must end with the word "block".

```
<Requirements:Requirement xmi:id="_cS0RsN-LEeywT9d24_j_DQ" base_NamedElement="_cSorgN-LEeywT9d24_j_DQ" id="
00" text="Global energy consuming of the Cycab: the CyCab must not exceed an appropriate maximum energy
consumption" base_Class="_cSorgN-LEeywT9d24_j_DQ"/>
<Requirements:Requirement xmi:id="_f_D58N-LEeywT9d24_j_DQ" base_NamedElement="_f-5h4N-LEeywT9d24_j_DQ" id="
01" text="Global energy consuming of the Station: the Station must not exceed an appropriate maximum
energy consumption" base_Class="_f-5h4N-LEeywT9d24_j_DQ"/>
<Requirements:Requirement xmi:id="_kEsSkN-LEeywT9d24_j_DQ" base_NamedElement="_kEca8N-LEeywT9d24_j_DQ" id="
02" text="Global energy consuming of the Vehicle: the Vehicle must not exceed an appropriate maximum
energy consumption" base_Class="_kEca8N-LEeywT9d24_j_DQ"/>
<Requirements:Requirement xmi:id="_mp2-AN-LEeywT9d24_j_DQ" base_NamedElement="_mpsl8N-LEeywT9d24_j_DQ" id="
011" text="Global energy consuming of the Sensor&#xA;-the maximum energy consuming of the offered service
pos is limited to Opos&#xA;-the maximum energy consuming of the required service spos is limited to Ospos
&#xA;" base_Class="_mpsl8N-LEeywT9d24_j_DQ"/>
<Requirements:Requirement xmi:id="_nBHAYN-LEeywT9d24_j_DQ" base_NamedElement="_nBCu8N-LEeywT9d24_j_DQ" id="
012" text="Global energy consuming of the Computing Unit&#xA;-the maximum energy consuming of the required
service halt is limited to chall&#xA;-the maximum energy consuming of the offered service pos is limited
to Opos&#xA;-the maximum energy consuming of the required service far is limited to Ofar " base_Class="
_nBCu8N-LEeywT9d24_j_DQ"/>
```

Fig. 4.8: Requirement information representation in XMI file

```
<packagedElement xmi:type="uml:Class" xmi:id="_cSorgN-LEeywT9d24_j_DQ" name="GECC" isLeaf="true">
  <nestedClassifier xmi:type="uml:Class" xmi:id="_f-5h4N-LEeywT9d24_j_DQ" name="ECS">
    <nestedClassifier xmi:type="uml:Class" xmi:id="_nBCu8N-LEeywT9d24_j_DQ" name="ECCU"/>
    <nestedClassifier xmi:type="uml:Class" xmi:id="_mpsl8N-LEeywT9d24_j_DQ" name="ECSS"/>
  </nestedClassifier>
  <nestedClassifier xmi:type="uml:Class" xmi:id="_kEca8N-LEeywT9d24_j_DQ" name="ECV">
    <nestedClassifier xmi:type="uml:Class" xmi:id="_-YISQPiGEeyplPXJEsGKTg" name="ECSR"/>
    <nestedClassifier xmi:type="uml:Class" xmi:id="_Apyy4PiHEeyplPXJEsGKTg" name="ECEH"/>
    <nestedClassifier xmi:type="uml:Class" xmi:id="_18dtgPiGEeyplPXJEsGKTg" name="ECVC"/>
  </nestedClassifier>
</packagedElement>
```

Fig. 4.9: Relationship between requirements representation in XMI file

```
<Requirements:Satisfy xmi:id="_pwJA8PiOEeylPq067yBXHA" base_DirectedRelationship="_pv2tEPiOEeylPq067yBXHA"
base_Abstraction="_pv2tEPiOEeylPq067yBXHA"/>
<Requirements:Satisfy xmi:id="_3WilMPiOEeylPq067yBXHA" base_DirectedRelationship="_3WapYPiOEeylPq067yBXHA"
base_Abstraction="_3WapYPiOEeylPq067yBXHA"/>
<Requirements:Satisfy xmi:id="_38wQoPiOEeylPq067yBXHA" base_DirectedRelationship="_38t0YPiOEeylPq067yBXHA"
base_Abstraction="_38t0YPiOEeylPq067yBXHA"/>
<Requirements:Satisfy xmi:id="_4cs8APiOEeylPq067yBXHA" base_DirectedRelationship="_4cqfwPiOEeylPq067yBXHA"
base_Abstraction="_4cqfwPiOEeylPq067yBXHA"/>
<Requirements:Satisfy xmi:id="_5jaFYPiOEeylPq067yBXHA" base_DirectedRelationship="_5jYQMPiOEeylPq067yBXHA"
base_Abstraction="_5jYQMPiOEeylPq067yBXHA"/>
<Requirements:Satisfy xmi:id="_6IV-cPiOEeylPq067yBXHA" base_DirectedRelationship="_6ITiMPiOEeylPq067yBXHA"
base_Abstraction="_6ITiMPiOEeylPq067yBXHA"/>
<Requirements:Satisfy xmi:id="_65ThUPiOEeylPq067yBXHA" base_DirectedRelationship="_65RsIPiOEeylPq067yBXHA"
base_Abstraction="_65RsIPiOEeylPq067yBXHA"/>
```

Fig. 4.10: Mapping between blocks and requirements representation

- **Convention 2:** Interface block names must begin with the name of the associated block and end with either "InterfaceIn" or "InterfaceOut".

- **Convention 3:** Port names must end with "PortIn" or "PortOut".

- **Convention 4:** The name of the delegation connectors that start from the block to parts must end with "Delegation-IN".

- **Convention 5:** The name of the delegation connectors that start from the parts to block must end with "Delegation-OUT".

The first and second conventions are necessary to differentiate between blocks and interface blocks since the ".Uml" file generated by Papyrus uses the same tags for both. The third convention allows us to differentiate between port In and port Out since the ".Uml" file generated by Papyrus does not allow us to differentiate between them. The fourth and fifth conventions allow us to differentiate between the Assembly and delegation connectors.

4.5 Running example

To illustrate our proposal, we present the case study of a CyCab car, which is a component-based system [3], developed by INRIA [1], and considered as a case study in the ANR TACOS[2] project, which is a French research initiative. The ANR TACOS project aims to develop a component-based methodology for specifying high-safety systems, particularly in land transportation, from the initial requirement stages to the development of formal specifications. The CyCab is a small, electric, and automatic vehicle used as a means of transportation, designed primarily for autonomous transport. It allows users to move around via a set of pre-installed stations. A computer system controls it and can be automatically piloted in many modes.

One of the critical aspects of the CyCab system is its modular design, which enhances both maintenance and fault tolerance. By decomposing the CyCab system into distinct sub-systems, such as the station components, sensors, and computing units, we improve the ease of identifying and isolating faults. This decomposition not only supports more efficient troubleshooting and repairs but also allows for the seamless integration of new features without disrupting the existing system. This modular approach is vital for maintaining the reliability and safety of the CyCab, particularly given its role in autonomous transport.

To illustrate our approach, we consider the following constraints: A CyCab has a dedicated road where the stations are equipped with sensors and computing units. There are no obstacles on the road. The driving of the CyCab is guided by the information received from the stations, which allows the CyCab to be located relative to the stations. The (*Starter*) receives a signal from the station to activate the vehicle. The vehicle is also equipped with an emergency halt button associated with the Emergency Halt (EH) component. The emergency halt button can be activated anytime during the movement of the CyCab.

[1] Institut National de Recherche en Informatique et Automatique
[2] Trustworthy Assembling of Components: from requirements to Specification

4.5.1 Modeling

To specify the architecture of a CPS, in this case, the CyCab, we propose to use the following SysML diagrams: the Block Definition Diagram (BDD), the Internal Block Diagram (IBD), and the Requirement Diagram (REQD). Figure 4.11 represents the BDD of the composite block *CyCab*, which is connected by compositional relations with its sub-blocks. Moreover, the IBD of CyCab, Station, and vehicle blocks are illustrated in Figures 4.12, 4.13, and 4.14.

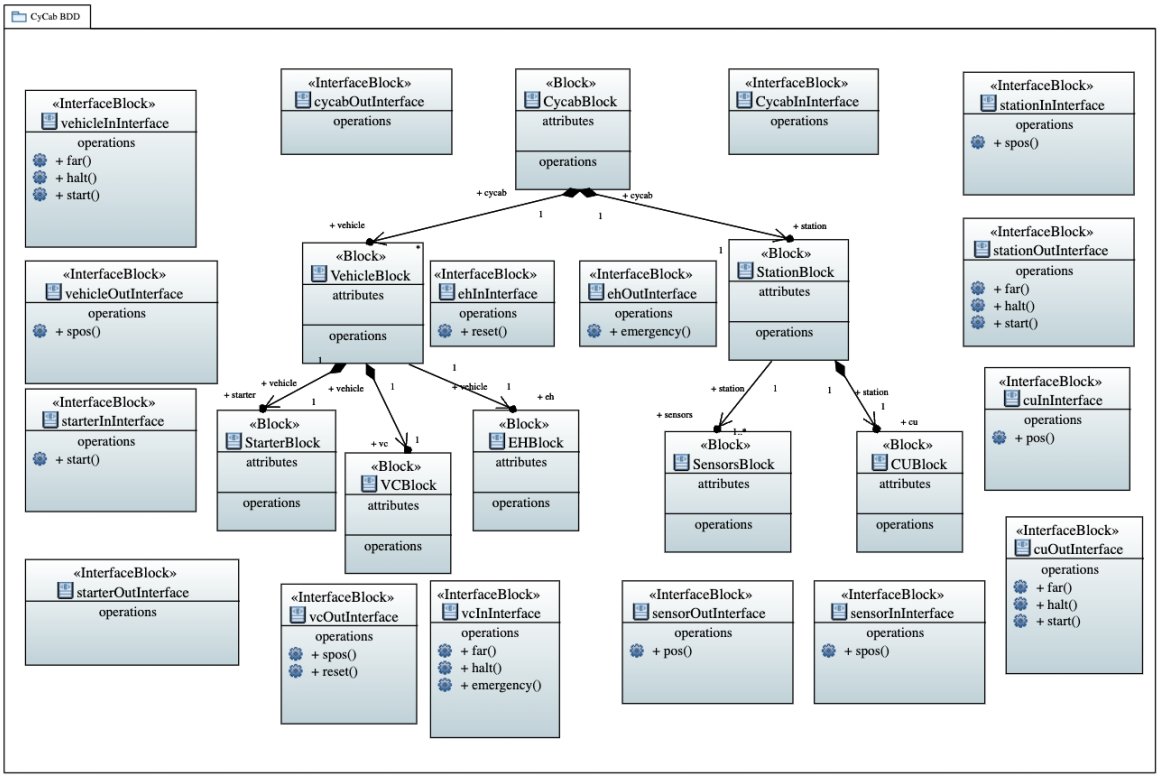

Fig. 4.11: Block Definition Diagram of the *CyCab* block

Figure 4.15 represents the requirement diagram of the system. The requirement GECC, Global Maximal Energy Consuming of CyCab, indicates that the CyCab must not exceed the energy consumption limit. This requirement contains the requirements for the ECS (Maximum Energy Consumption of Station Component) and ECV (Maximum Energy Consumption of Vehicle Component). The ECS requirement contains the requirements ECSS, the maximum energy consumption of the sensor component, and ECCU, the maximum energy consumption of the computing unit component. The ECV requirement contains the requirements ECVC, the maximum energy consumption of the vehicle core component; ECSR, the maximum energy consumption of the starter

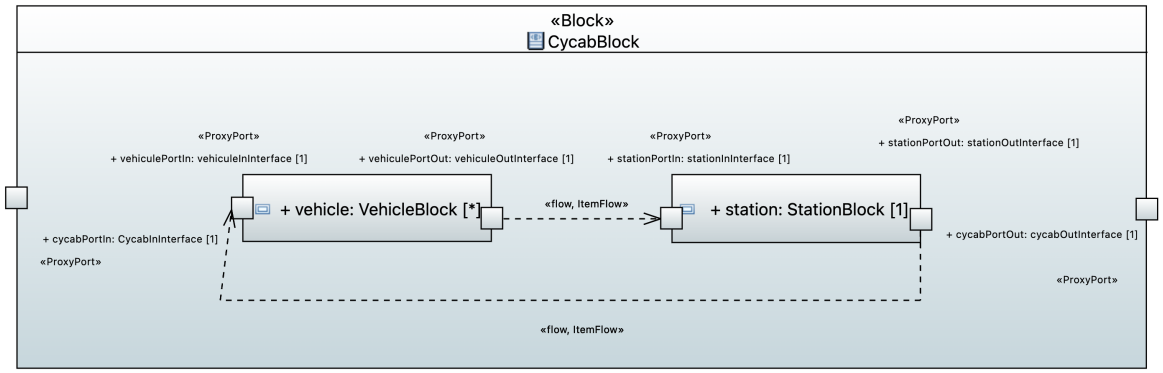

Fig. 4.12: Internal Block Diagram of the *CyCab* block

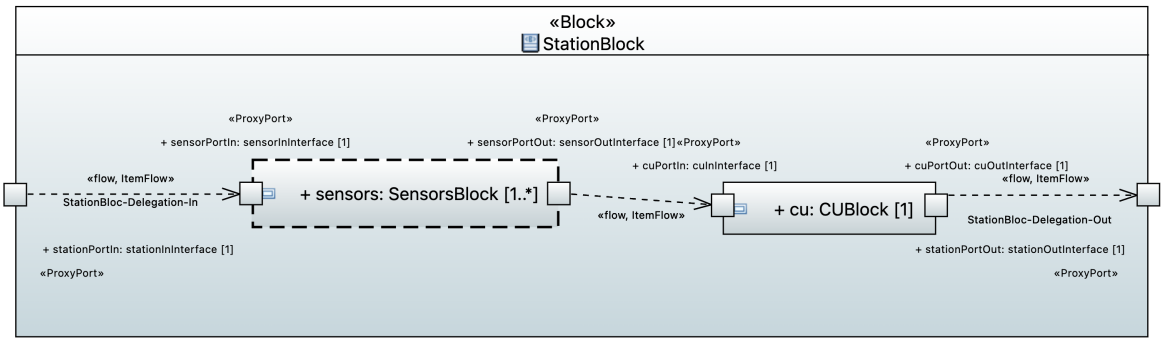

Fig. 4.13: Internal Block Diagram of the *Station* block

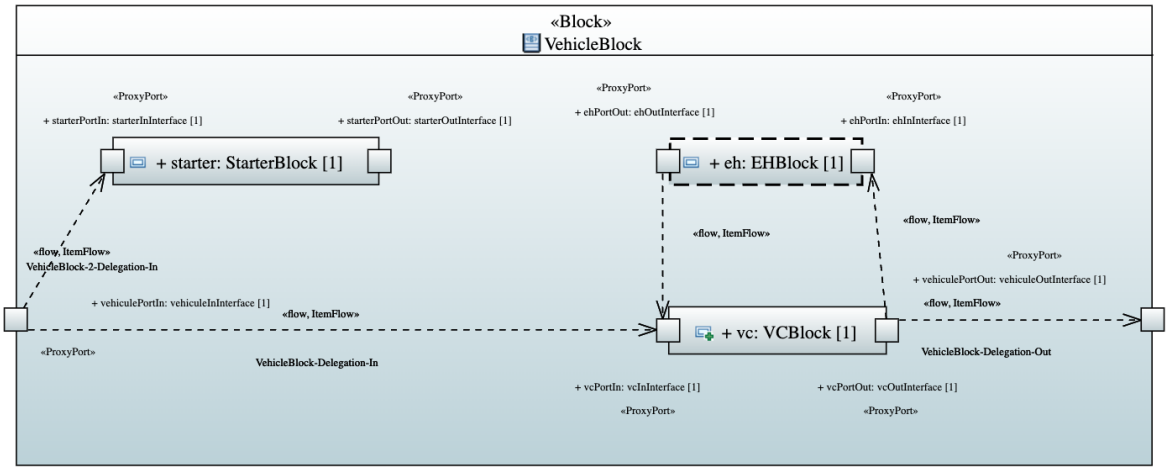

Fig. 4.14: Internal Block Diagram of the *Vehicle* block

component; and ECEH, the maximum energy consumption of the emergency stop component.

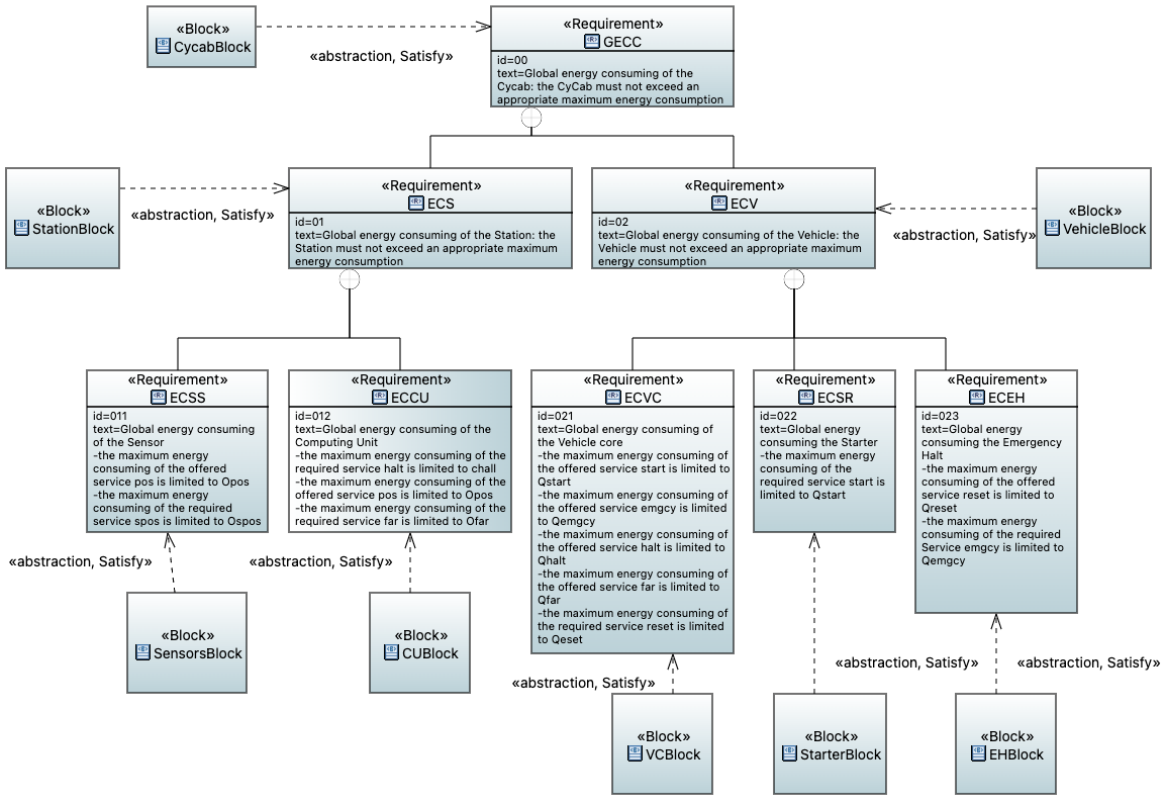

Fig. 4.15: The *CyCab* requirement diagram

4.5.2 Transformation

After modeling the SysML diagrams, we proceed to the transformation step. We introduce the file generated by Papyrus into the tool presented in Section 4.4, which produces the formal specification. However, due to the size of the case study, we only present a representative excerpt of the generated result.

We now present several specifications. Listing 4.1 shows the GECC, ECS, and ECV requirement specifications. Listing 4.2 provides the formal specification of the requirement diagram. Listing 4.3 shows the formal specifications for the vehicle's input and output interface blocks. Listing 4.4 contains the formal definitions of the vehicle's input and output ports as well as those of the VC. Listing 4.5 presents the formal specifications for the station and vehicle blocks. Finally, Listing 4.6 shows the formal specifications of the BDD.

Listing 4.1: Requirements

```
let gecc = { id = "00"; name = "GECC"; texte = "Global␣energy␣
   consuming␣of␣the␣Cycab:␣the␣CyCab␣must␣not␣exceed␣an␣appropriate␣
   maximum␣energy␣consumption"}
let ecs = { id = "01"; name = "ECS"; texte = "Global␣energy␣consuming
   ␣of␣the␣Station:␣the␣Station␣must␣not␣exceed␣an␣appropriate␣
   maximum␣energy␣consumption"}
let ecv = { id = "02"; name = "ECV"; texte = "Global␣energy␣consuming
   ␣of␣the␣Vehicle:␣the␣Vehicle␣must␣not␣exceed␣an␣appropriate␣
   maximum␣energy␣consumption"}
```

Listing 4.2: Requirement diagram

```
let reqD = { iR = gecc ;
sR = [ gecc ; ecs ; ecv ; ecss ; eccu ; ecvc ; ecsr ; eceh ] ;
relC = [ (gecc, [ecs ; ecv]) ;(ecs, [eccu ; ecss]) ;(ecv, [ecsr ;
   eceh ; ecvc])] ; }
```

Listing 4.3: Interface block

```
let vehicleininterface = { nameIB = "vehicleInInterface"; opB = [ "
   far" ;"halt" ;"start"  ]  }
let vehicleoutinterface = { nameIB = "vehicleOutInterface"; opB = [ "
   spos"  ]  }
```

Listing 4.4: Ports

```
let vehicleportin = {nameP = "vehicleportin" ;typeP =
   vehicleininterface ;directionP = "in" ;connectorInterneSBC = []}
let vehicleportout = {nameP = "vehicleportout" ;typeP =
   vehicleoutinterface ;directionP = "out" ;connectorInterneSBC =
   []}
let vcportout = {nameP = "vcportout" ;typeP = vcoutinterface ;
   directionP = "out" ;connectorInterneSBC = ["reset" ]}
```

Listing 4.5: Blocks

```
let stationblock = {name = "StationBlock" ;pinB = stationportin ;
   poutB = stationportout;connectorsOutB = [(cuportout,
   stationportout ) ] ; connectorsOutBIn = [(stationportin,
   sensorportin ) ] ; sRB = [ecs] ; }
let vehicleblock = {name = "VehicleBlock" ;pinB = vehicleportin ;
   poutB = vehicleportout;connectorsOutB = [(vcportout,
   vehicleportout ) ] ; connectorsOutBIn = [(vehicleportin,vcportin
   ) ;(vehicleportin,starterportin ) ] ; sRB = [ecv] ; }
```

Listing 4.6: Block definition diagram

```
let bddD = {
iBs =cycabblock;
sBs = [ cycabblock;stationblock;vehicleblock;cublock;sensorsblock;
    starterblock;vcblock;ehblock ] ;
suBs = [ (cycabblock,[stationblock;vehicleblock]) ; (stationblock,[
    cublock;sensorsblock]) ; (vehicleblock,[starterblock;vcblock;
    ehblock])   ] ;
}
```

4.5.3 Structural consistency checking

Finally, after transforming the SysML models into the proposed formal specification, we verify the structural consistency using the implementation of Algorithms 1 and 2. Our case study includes all the special cases considered in our approach.

The IBD of the station illustrates a simple case: the services offered by the station must be included in those offered by the sensors, and the services required by the CU must also be required by the station.

The IBD of the vehicle, on the other hand, contains two specific situations. First, the services offered by the vehicle must be included in the union of the services offered by the VC and the starter. Second, the services required by the VC must be included in those required by the vehicle. However, since the VC is connected to the EH input port, it is necessary to subtract the services provided by the EH from the list of services required by the VC before performing the inclusion check. Also, for the consistency of the mapping between blocks and requirements, we must verify that the atomic requirements that the GECC (ECSS, ECCU, ECVC, ECSR, ECEH) block verifies are included in the atomic requirements that the ECS(ECSS, ECCU) union ECV (ECVC, ECSR, ECEH) block verifies.

Figure 4.16 shows the result of the verification of our model. We can see that all the blocks are annotated with "consistency is verified". Therefore, we can say that our model is structurally consistent.

Now, if the Station block offers an additional service to switch the station off (see Figure 4.17), the formal specification of the input block interface is represented as follows:

```
let stationininterface = { namexB = "stationInInterface"; opB = [ "
    spos" ; "off" ]  }
```

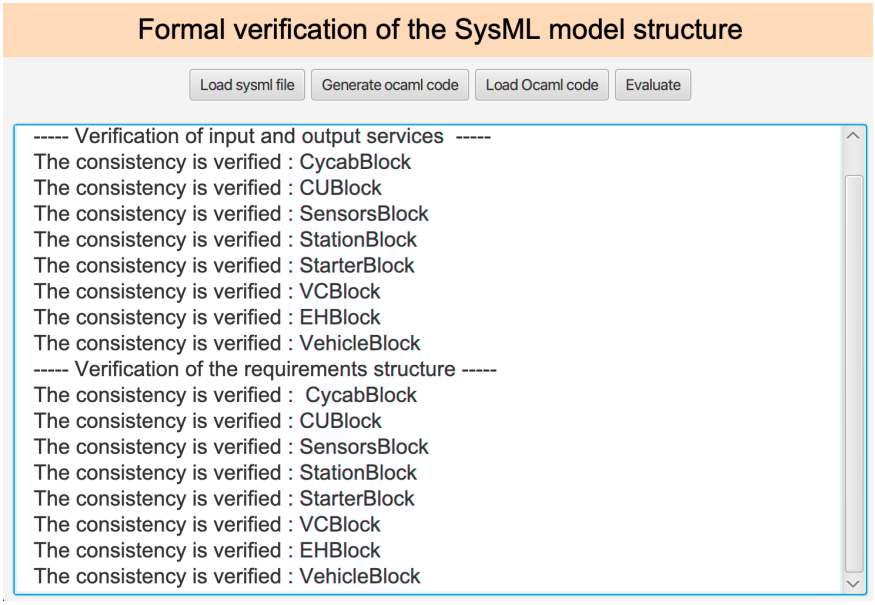

Fig. 4.16: Result of the consistency verification

Figure 4.18 shows the result of the verification of the model after the modification. We observe that the Station block is inconsistent, as its sub-components CU and Sensor do not provide the service required to switch off the station. Specifically, this inconsistency arises because neither sub-component offers the "off" service required by the Station block. To maintain consistency, at least one sub-component of Station connected via an input delegation port must provide the "off" service.

Our approach detects such inconsistencies by enforcing that all services required by a composite block are provided by at least one of its sub-components. Without this verification mechanism, the system may exhibit unexpected behavior or fail to perform essential operations, such as shutting down the station. By identifying these issues early in the design process, our method contributes to improving the system's dependability, particularly in terms of robustness (i.e., the ability to handle internal mismatches or degraded conditions) and reliability (i.e., the probability of operating correctly over time).

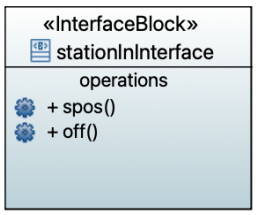

Fig. 4.17: Station block with OFF service

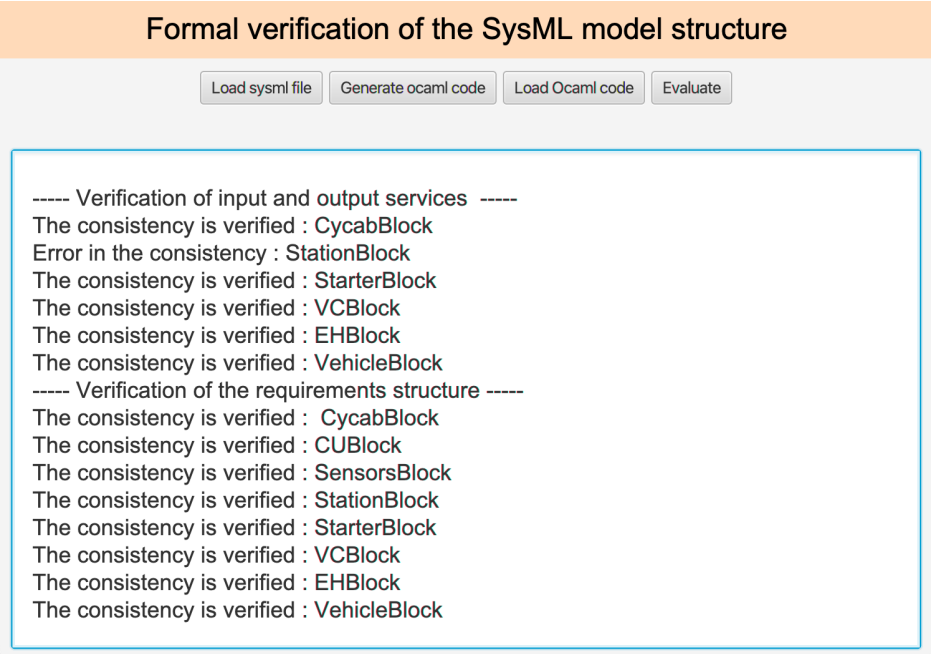

Fig. 4.18: Verification result after editing the station interface

4.6 Discussion about the applicability of our approach on SysML v2

In SysML v1, associations are commonly used to model port connections. However, these associations are abstract and do not provide details about the interactions between ports. This abstraction may lead to inconsistencies, as there is no explicit mechanism to ensure that the services required by one port are effectively provided by the connected port. SysML v2 addresses these limitations by introducing interface definitions (interface defs), which explicitly specify connection protocols and define the services that must be provided or consumed by connected ports. Table 4.1 presents some correspondences between SysML v1 and SysML v2 concepts.

Table 4.1: Mapping between SysML v1 and SysML v2

SysML v1	**SysML v2**
Part property / Block	Part / Part def
Proxy port / Interface block	Port / Port def
	Interface / Interface def

Although SysML v2 significantly improves modeling capabilities, particularly through the introduction of interface definitions that formalize port connections and service bindings, it does not detect the type of structural inconsistency addressed by our approach.

To illustrate this limitation, we modeled the structure of the CyCab's Station using SysML v2 (see Listings 4.7, 4.8, 4.9, and 4.10), and included an additional service named "off", which is provided by the Station block. Figure 4.19 shows the corresponding internal block diagram, generated from the textual syntax.

After building the project, no errors were reported, even though the Station block provides a service that is not offered by any of its sub-blocks (Sensor and CU). This reveals a critical limitation in SysML v2, which lacks native mechanisms for checking such structural inconsistencies.

Our verification approach addresses this issue by ensuring that composite blocks do not declare services unsupported by their internal components. While applicable to SysML v2, the approach must be adapted to accommodate its textual syntax and structural concepts, including interface definitions and explicit connections. These adjustments allow the integration of our method within the SysML v2 framework, thus ensuring model integrity and improving early validation.

Listing 4.7: Port def

```
port def IB_Station_in {
    in ref action pos {}
    out ref action halt {}
    out ref action far {}
    out ref action off {}}
port def IB_Station_out {
    out ref action halt {}
    out ref action far {}}
port def IB_Sensor_in {
    in ref action pos {}
    out ref action spos {}}
port def IB_Sensor_out {
   out ref action spos {}}
port def IB_CU_in {
   in ref action spos {}}
port def IB_CU_out {
   out ref action halt {}
   out ref action far {}
   }
```

Listing 4.8: interface def

```
interface def
   Station_Sensor_Interface{
    end port pp : IB_Station_in;
    end port pp_conj : IB_Sensor_in;
    bind pp_conj.pos = pp.pos;}

interface def Sensor_CU_Interface{
    end port pp : IB_Sensor_out;
    end port pp_conj : IB_CU_in;
    bind pp_conj.spos = pp.spos;}

interface def CU_Station_Interface{
    end port pp : IB_CU_out;
    end port pp_conj : IB_Station_in
       ;
    flow pp_conj.far to pp.far;
    flow pp_conj.halt to pp.halt;}
```

Listing 4.9: Part def

```
part def Station {
    port pStation : IB_Station_in;
    port pStation_out : IB_Station_out;
    part def Sensor{
        port pSensor : IB_Sensor_in;
        port pSensor_out : IB_Sensor_out;}
    part def CU{
        port pCU : IB_CU_in;
        port pCU_out : IB_CU_out;}
    connection def Station_Sensor {
        end part block1 : Station;
        end part block2 : Sensor;
        interface :Station_Sensor_Interface connect block1.pStation
           to block2.pSensor; }
    connection def Sensor_CU {
        end part block1 : Sensor;
        end part block2 : CU;
        interface :Sensor_CU_Interface connect block1.pSensor_out to
           block2.pCU;}
    connection def Cu_Station {
        end part block1 : CU;
        end part block2 : Station;
        interface :CU_Station_Interface connect block1.pCU_out to
           block2.pStation;} }
```

Listing 4.10: Sensor usage

```
part station : Station {
    part  sensor : Sensor{}
    part cu : CU{}

    interface : Station_Sensor_Interface connect pStation to sensor.
       pSensor;

    interface : Sensor_CU_Interface connect sensor.pSensor_out to cu.
       pCU;

    interface : CU_Station_Interface connect cu.pCU_out to
       pStation_out;
    }
```

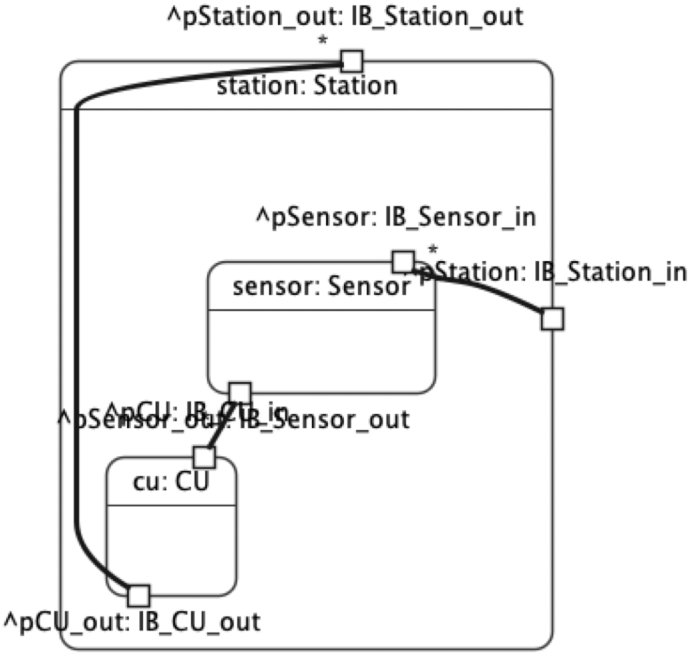

Fig. 4.19: Station IBD in SysML v2

4.7 Conclusion

In this chapter, we introduced a formal approach to verifying the structural consistency of SysML models for Cyber-Physical Systems (CPS). Unlike standard modeling environments, which typically validate only syntactic correctness, our method rigorously checks that component decompositions, service definitions, and requirement allocations satisfy well-defined consistency rules.

We proposed a formal specification for key SysML elements, blocks, ports, interfaces, and requirements. We defined a static semantics composed of three conditions (S1, S2, and S3) to ensure that services and requirements are consistently refined across hierarchical compositions. We demonstrated the feasibility of our approach through a transformation tool that extracts these formal specifications from SysML models.

Using the CyCab autonomous vehicle as a case study, we illustrate how our methodology can identify structural inconsistencies that are not detected by visual inspection or standard modeling tools. Furthermore, we discuss how our approach remains applicable to SysML v2, despite its richer syntax and improved semantics.

Ultimately, our method strengthens the assurance provided by MBSE practices, allowing early detection of structural inconsistencies in CPS architectures. This

contributes significantly to the reliability and maintainability of safety-critical systems by enabling early validation of architectural decisions.

In contrast to the custom formalism introduced in this chapter, the next chapter leverages the official SysML v2 specification to explore the modeling and formal verification of component interactions. In particular, it focuses on how SysML v2 can be integrated with the BIP framework to support rigorous analysis of coordination mechanisms.

Chapter 5

Modeling and Verification of Structured Interactions using SysML v2 and BIP

5.1 Introduction

Cyber-Physical Systems (CPS) are increasingly deployed in safety-critical domains such as autonomous transportation, aerospace, and industrial automation. These systems rely on the coordination of distributed hardware and software components that must interact in a predictable and reliable manner to ensure global correctness. Failures in coordination, such as deadlocks, race conditions, or inconsistent synchronization, can lead to unsafe system states and critical failures. Ensuring the correctness of component interactions is therefore a fundamental challenge in CPS engineering.

The Systems Modeling Language (SysML) is widely adopted for model-based systems engineering. Its recent evolution into SysML v2 introduces significant improvements in modularity, semantic precision, and textual expressiveness, making it a promising candidate for CPS design. However, SysML v2 remains descriptive and lacks formal execution semantics. In particular, it does not natively support the modeling or analysis of synchronization and communication between concurrent components, capabilities that are essential for verifying interaction correctness in CPS.

To address this limitation, we propose a structured interaction modeling approach based on a well-defined subset of SysML v2 constructs. This subset enables the explicit specification of coordination patterns using reusable connection definitions and port types. We focus on two fundamental mechanisms:

- **Rendez-vous connectors**, which enforce strict synchronization among multiple components, requiring all participants to be jointly enabled before progressing;
- **Broadcast connectors**, which support one-to-many communication, allowing receivers to respond asynchronously to emitted signals from a sender.

While these patterns can be described structurally in SysML v2, the language lacks operational semantics necessary for execution or formal verification. To address this limitation, we propose a transformation from SysML v2 models into the *Behavior, Interaction, Priority* (BIP) framework. BIP offers a rigorous formal semantics for coordination and interaction, along with tool-supported capabilities for model execution and verification. Through this transformation, the interaction structures specified in SysML v2 are preserved and enhanced with executable behavior, thereby enabling the verification of essential properties such as deadlock-freedom, synchronization correctness, and interaction liveness.

This chapter builds upon the foundational principles presented earlier in the dissertation. It demonstrates how SysML v2 can be extended with structured interaction modeling capabilities and systematically integrated with formal verification workflows using BIP. The proposed methodology is illustrated through a case study involving the coordination of a drone swarm.

5.2 Modeling Structured Interactions in SysML v2

This section presents our structured modeling strategy for capturing interaction semantics in SysML v2. Although SysML v2 offers precise syntax for defining parts, ports, and connections, it lacks executable semantics. This limitation makes it difficult to reason formally about the correctness of component coordination. To address this, we define a modeling subset that explicitly captures structured interactions using reusable connector definitions. Our approach focuses on two representative interaction patterns that are critical for coordinated behavior in CPS: *Rendez-vous* and *Broadcast*.

5.2.1 SysML v2 Constructs for Interaction Modeling

SysML v2 allows system designers to define modular components using `part definitions`, and to specify communication interfaces through `ports`. Interactions between components are defined structurally via `connection definitions` that link compatible ports. However, the semantics of these connections, particularly in concurrent settings,

are not formally specified. There is no native support for expressing synchronization constraints, optional participation, or execution atomicity.

To address this, we elevate connector definitions to first-class elements that encode specific interaction semantics. Each connector explicitly captures its coordination policy, which can then be preserved during transformation into executable semantics in BIP. This modeling pattern promotes clarity and modularity while laying the foundation for formal verification.

5.2.2 Rendez-vous Interactions

Rendez-vous interactions encode strong synchronization: all participating components must be simultaneously ready for the interaction to occur. This pattern is well-suited for applications requiring coordinated transitions, such as collective resets, synchronized state changes, or tightly coupled decision points.

To express rendez-vous interactions in SysML v2, we propose modeling them as connections between multiple ports of type `SyncPort`. In this approach, the interaction occurs only when all endpoints are simultaneously active. Listing 5.1 illustrates a typical example.

Listing 5.1: Rendez-vous Interaction in SysML v2

```
connection def RendezVous {
    end p1 : SyncPort;
    end p2 : SyncPort;
    end p3 : SyncPort;
}
```

Figure 5.1 illustrates the graphical representation of this connector. The interaction completes only when all three ports are simultaneously enabled.

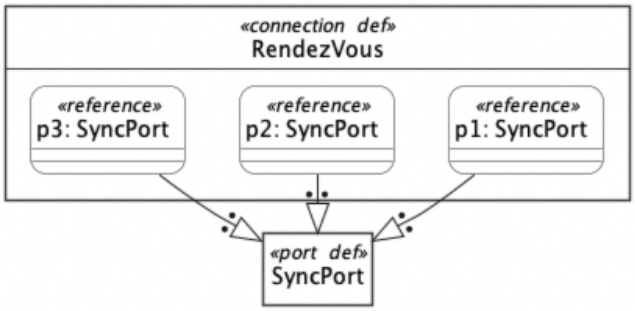

Fig. 5.1: Graphical Representation of a Rendez-vous Connector in SysML v2

The corresponding BIP translation is shown in Listing 5.2. All ports are treated symmetrically, and the interaction is executed atomically only when all involved components are ready.

Listing 5.2: Rendez-vous Connector in BIP

```
connector type RendezVous(SyncPort_t p1, SyncPort_t p2, SyncPort_t p3
    ) {
     define p1 p2 p3;
}
```

This synchronous connector enforces determinism and avoids race conditions in critical interaction regions. However, it may introduce blocking behavior, especially when one participant is unavailable. As such, rendez-vous connectors should be used judiciously and typically for operations where consistency must be guaranteed across multiple components.

5.2.3 Broadcast Interactions

Broadcast interactions model one-to-many communication in which a sender transmits an event or signal that multiple independent receivers may receive. Unlike rendez-vous, participation by receivers is optional: the sender can emit the broadcast without being blocked by unavailable recipients. This interaction mode is particularly suited for asynchronous coordination in loosely coupled systems.

In our approach, broadcast interactions are encoded in SysML v2 as `connection definitions` involving a sender of type `BroadcastPort` and multiple receivers of type `CommandPort`. Readiness constraints are explicitly declared to reflect optional participation. Listing 5.3 shows a simple example where a sender transmits a signal to three receivers:

Listing 5.3: Broadcast Interaction in SysML v2

```
connection def Broadcast {
    end sender : BroadcastPort;
    end receiver1 : CommandPort;
    end receiver2 : CommandPort;
    end receiver3 : CommandPort;
    constraint {sender.isReady}
    flow sender.Signal to receiver1.Signal;
    flow sender.Signal to receiver2.Signal;
    flow sender.Signal to receiver3.Signal;}
```

Figure 5.2 illustrates this interaction. The sender can proceed if ready, and each receiver reacts only if it is available.

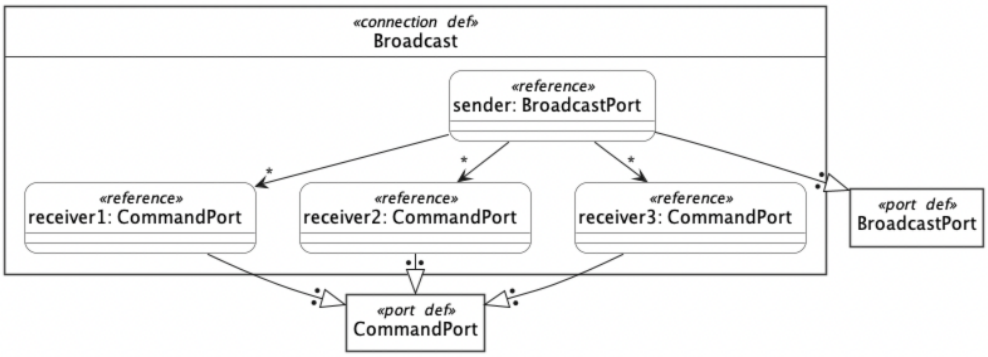

Fig. 5.2: Graphical Representation of a Broadcast Connector in SysML v2

In BIP, this connector is translated into a trigger-based interaction using the `define` and `on...down` clauses. The sender is marked as the trigger, and the interaction dynamically adapts to receiver availability, as shown in Listing 5.4.

Listing 5.4: Broadcast Connector in BIP

```
connector type Broadcast(BroadcastPort_t sender,
                         CommandPort_t receiver1,
                         CommandPort_t receiver2,
                         CommandPort_t receiver3) {
    define sender' receiver1 receiver2 receiver3;
    on sender receiver1 receiver2 receiver3 down {
        receiver1.Signal = sender.Signal;
        receiver2.Signal = sender.Signal;
        receiver3.Signal = sender.Signal;
    }
}
```

For more complex scenarios, broadcast connectors may include multiple constraints specifying subsets of participating receivers based on runtime readiness. This flexibility enables precise control over partial interactions, as shown in Listing 5.5.

Listing 5.5: Broadcast Interaction with Multiple Constraints in SysML v2

```
connection def Broadcast {
    end sender : BroadcastPort;
    end receiver1 : CommandPort;
    end receiver2 : CommandPort;
```

```
    constraint {sender.isReady and receiver1.isReady and receiver2.
       isReady}
    flow sender.Signal to receiver1.Signal;
    flow sender.Signal to receiver2.Signal;

    constraint {sender.isReady and receiver1.isReady}
    flow sender.Signal to receiver1.Signal;

    constraint {sender.isReady and receiver2.isReady}
    flow sender.Signal to receiver2.Signal;
}
```

This SysML v2 model is translated into BIP using `on` blocks that incorporate conditional guards, as illustrated in Listing 5.6.

Listing 5.6: Broadcast Connector with Multiple Constraints in BIP

```
connector type Broadcast(BroadcastPort_t sender,
                         CommandPort_t receiver1,
                         CommandPort_t receiver2) {
    define sender' receiver1 receiver2;

    on sender receiver1 receiver2 down {
        receiver1.Signal = sender.Signal;
        receiver2.Signal = sender.Signal;
    }
    on sender receiver1 down {
        receiver1.Signal = sender.Signal;
    }
    on sender receiver2 down {
        receiver2.Signal = sender.Signal;
    }
}
```

This mapping preserves the flexible execution semantics of broadcast interactions, enabling safe communication patterns without enforcing unnecessary synchronization. It also reflects realistic execution contexts where not all participants are always available, thus improving robustness and responsiveness in distributed CPS architectures.

5.3 Transformation from SysML v2 to BIP

To enable formal verification and execution semantics, we propose a systematic transformation from structured SysML v2 models into equivalent BIP specifications. This

transformation bridges the gap between high-level system architecture modeling and formally grounded execution semantics, while preserving both the structure and coordination logic of the original SysML v2 models.

The transformation process targets both synchronous and asynchronous interaction patterns and consists of three main phases: model element extraction, rule-based translation, and generation of an executable BIP model. These phases are guided by transformation rules that ensure a faithful mapping of structural, behavioral, and interaction semantics.

5.3.1 Overview of Transformation Principles

The transformation begins by extracting the key modeling constructs from the SysML v2 model, including:

- `port definitions` — specifying interface types and data;
- `part definitions` — describing the components;
- `state definitions` — capturing internal behavior;
- `connection definitions` — specifying coordination logic between components.

Each of these elements is mapped to its BIP counterpart:

- Ports are translated into BIP `port types`, preserving their directionality.
- State-based behaviors are captured by BIP `atom types`, composed of `places` (states) and guarded `transitions`.
- Interactions defined through connection definitions are translated into BIP `connector types`, with constraints and flows translated into interaction guards and data transfer actions.

If several transitions in SysML v2 share the same source and target states but differ in guard conditions, they are merged in BIP using conditional statements to retain behavioral distinctions.

5.3.2 Transformation Algorithm

The transformation procedure is summarized in Algorithm 3, which describes the translation from a SysML v2 model M_{SysML} to its executable BIP counterpart M_{BIP}.

Algorithm 3: Transformation from SysML v2 to BIP

Input: SysML v2 model M_{SysML}
Output: Equivalent BIP model M_{BIP}
Parse M_{SysML} to extract model elements:;
- Identify all `port def`, `part def`, `state def`, and `connection def`;
- Extract transitions from `state def`;
- Identify constraints and their associated `flow` statements;

Initialize an empty BIP model M_{BIP};
foreach *port definition in* M_{SysML} **do**
  Map to a BIP `port type` with attributes;
  Add the mapped port to M_{BIP};
end
foreach *part definition containing a* `state def` **do**
  Create a BIP `atom type` for the part;
  foreach *state definition in the part* **do**
    foreach *state in the* `state def` **do**
      Create a BIP `place` representing the state;
    end
    Convert transitions into BIP transitions;
  end
  Add the `atom type` to M_{BIP};
end
foreach *connection definition in* M_{SysML} **do**
  Create a BIP `connector type` with corresponding ports;
  Identify the port present in all constraints as the *trigger port* (marked with apostrophe ');
  foreach *constraint in the connection* **do**
    Translate the constraint into a BIP `on` condition;
    Identify associated `flow` statements;
    Group them under the matching `on` condition in the `down` section;
  end
  Add the connector to M_{BIP};
end
Transform the system composition:;
Identify the top-level part (composition root);
Map it to a BIP `compound type`;
Add it to M_{BIP};
foreach *contained part instance* **do**
  Add the corresponding `atom type` to M_{BIP};
end
foreach *connection inside the composition* **do**
  Add the corresponding BIP `connector type` to M_{BIP};
end
return M_{BIP}

5.3.3 Automation Potential and Tool Integration

Although the current transformation is carried out manually, the structured nature of SysML v2 models makes them highly suitable for automation. The standardized syntax

and semantics of the language, combined with the availability of the SysML v2 API, enable the programmatic extraction of model elements and the systematic application of transformation rules.

When integrated into a model-driven toolchain, this transformation process could support the automatic generation of executable BIP specifications directly from SysML v2 models. This would significantly reduce modeling effort, ensure consistency, and improve traceability between abstract architectures and their formally verified counterparts.

Ultimately, this integration reinforces the methodological link between descriptive modeling and formal analysis. It allows system engineers to fully exploit the expressiveness of SysML v2 while benefiting from the rigorous verification capabilities offered by the BIP framework.

5.4 Case Study: Swarm Drone Coordination with SysML v2 and BIP

Coordinating a fleet of autonomous drones is a fundamental challenge in cyber-physical system design, particularly in mission-critical applications requiring precise timing, synchronization, and robustness. This case study illustrates how structured interactions can be modeled in SysML v2 and transformed into an executable BIP model to ensure correctness, detect design flaws, and analyze system behavior.

The system under study consists of a central command station and three drones that must respond to broadcast commands and synchronize at specific moments. Two complementary coordination patterns govern their behavior:

- A **broadcast interaction**, where the station disseminates commands to all drones simultaneously;
- A **rendez-vous synchronization**, used to reset the entire swarm in a coordinated manner.

5.4.1 SysML v2 Specification

The system architecture is captured in SysML v2 using modular modeling constructs. We present each major model element individually to improve readability and traceability between the textual specification and its corresponding graphical representation.

Port Definitions. We begin by defining the communication ports for broadcasting, command reception, and synchronization. These ports are shown in Listings 5.7 to 5.9.

Listing 5.7: BroadcastPort Definition

```
port def BroadcastPort {
    out attribute commandSignal : String ;
    attribute isReady = true;
}
```

Listing 5.8: CommandPort Definition

```
port def CommandPort {
    in attribute commandSignal : String ;
    attribute isReady = true;
}
```

Listing 5.9: RendezVousPort Definition

```
port def RendezVousPort { }
```

Connector Definitions. The system includes two types of connectors: a broadcast connector and a rendezvous connector. Listings 5.10 and 5.11 show their definitions, followed by the corresponding diagrams.

Listing 5.10: Broadcast Connector

```
connection def Broadcast {
    end sender : BroadcastPort;
    end receiver1 : CommandPort;
    end receiver2 : CommandPort;
    end receiver3 : CommandPort;
    constraint{sender.isReady and receiver1.isReady and receiver2.
       isReady and receiver3.isReady}
    flow sender.commandSignal to receiver1.commandSignal;
    flow sender.commandSignal to receiver2.commandSignal;
    flow sender.commandSignal to receiver3.commandSignal;
}
```

The corresponding graphical representation of the `Broadcast` connector is shown in Figure 5.3.

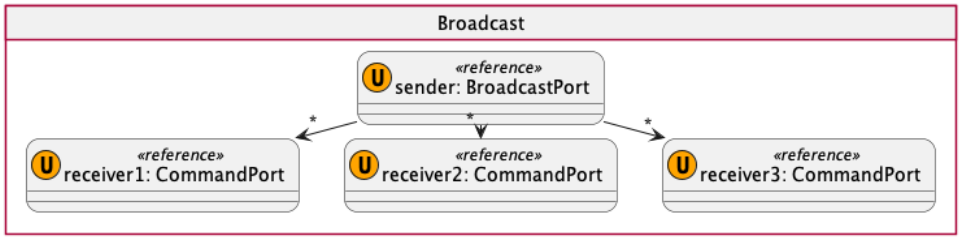

Fig. 5.3: Broadcast Connector in SysML v2

Listing 5.11: RendezVous Connector

```
connection def RendezVous {
    end d1 : RendezVousPort;
    end d2 : RendezVousPort;
    end d3 : RendezVousPort;
}
```

Figure 5.4 provides the graphical view of the `RendezVous` connector.

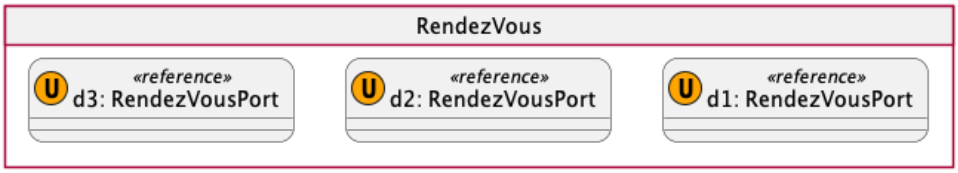

Fig. 5.4: Rendezvous Connector in SysML v2

Part Definitions. We now present the behavioral components of the system, namely the `Drone` and the `StationDeCommande`, which are defined in Listings 5.12 and 5.13.

Listing 5.12: Drone Part Definition

```
part def Drone {
    port commandIn : CommandPort;
    port rendezVous : RendezVousPort;
    attribute commandSignal : String ;
    state def DroneStates {
      entry; then Idle;
      state Idle; state Ready;
      state Flying; state Formation;
      transition 'Idle-Ready'
        first Idle accept  signal : String via commandIn
        then Ready {commandSignal = "Ready";}
      transition 'Ready-Flying'
        first Ready
```

```
        accept signal : String via commandIn
        then Flying {commandSignal = "InAir";}
      transition 'Flying-Formation'
        first Flying
        accept signal : String via commandIn
        then Formation {commandSignal = "InFormation";}
      transition 'Formation-Idle'
        first Formation
        accept signal : String via rendezVous
        then Idle;
    }
}
```

The behavioral structure of the `Drone` component is depicted in Figure 5.5.

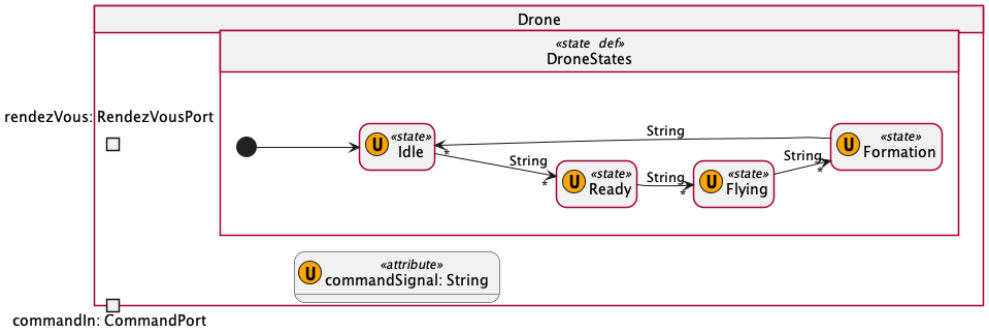

Fig. 5.5: Drone Part Definition in SysML v2

Listing 5.13: Command Station Part Definition

```
part def StationDeCommande {
    port stationCommand : BroadcastPort;
    state def StationStates {
      attribute phase : Integer = 0;
      attribute compteur : Integer = 0;
      attribute maxCycles : Integer = 10;
      state Active; state Stop;
      entry;  then Active;
      transition 'Active-Start'
        first Active
        accept Signal : String via stationCommand
        if phase == 0 and compteur < maxCycles
        then Active {
          phase = 1;
          compteur = compteur + 1;}
      transition 'Active-Takeoff'
```

```
        first Active
        accept Signal : String via stationCommand
        if phase == 1 and compteur < maxCycles
        then Active {
          phase = 2;
          compteur = compteur + 1;}
      transition 'Active-Formation'
        first Active
        accept Signal : String via stationCommand
        if phase == 2 and compteur < maxCycles
        then Active {
          phase = 0;
          compteur = compteur + 1; }
      transition 'Active-Stop'
        first Active
        if compteur >= maxCycles
        then Stop;
    }
}
```

The state machine and attributes of the `StationDeCommande` component are illustrated in Figure 5.6.

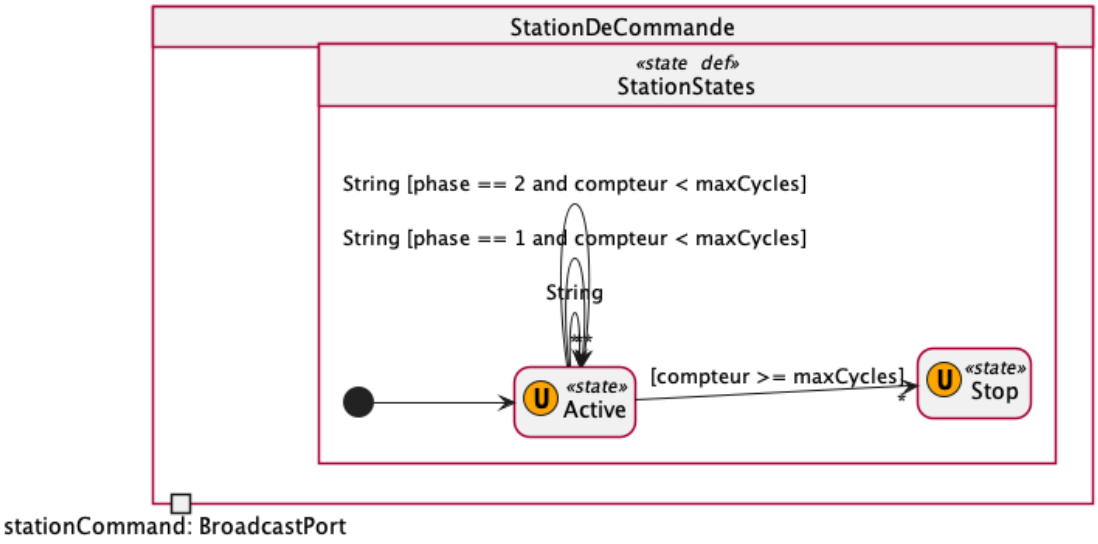

Fig. 5.6: Command Station Part Definition in SysML v2

System Assembly. Finally, Listing 5.14 shows the top-level system composition assembling all the parts into a drone swarm.

Listing 5.14: Drone Swarm System Assembly

```
part DroneSwarm {
    part station : StationDeCommande;
    part d1 : Drone;
    part d2 : Drone;
    part d3 : Drone;
    connection leaderCommand : Broadcast connect (station.
        stationCommand, d1.commandIn, d2.commandIn, d3.commandIn);
    connection rendezvousSync : RendezVous connect (d1.rendezVous, d2
        .rendezVous, d3.rendezVous);
}
```

The whole system composition is shown in Figure 5.7, which connects the station to three drone components.

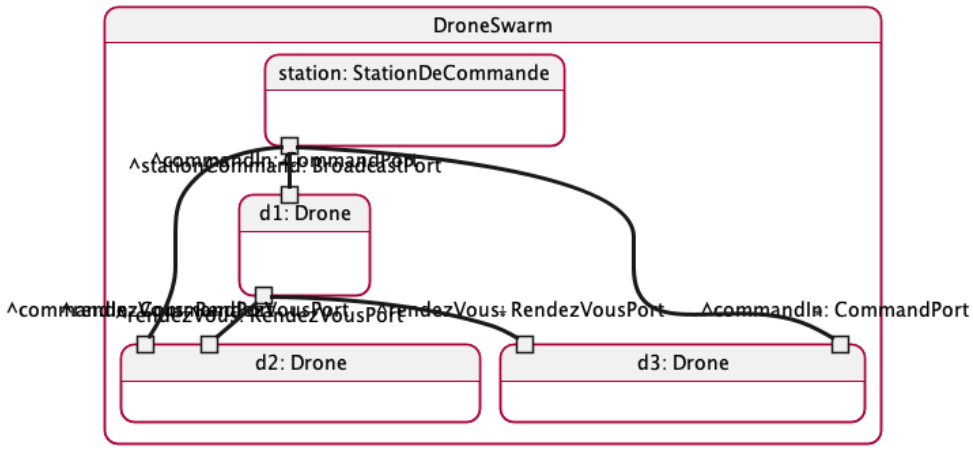

Fig. 5.7: System Assembly of the Drone Swarm in SysML v2

5.4.2 BIP Model and Execution

The corresponding BIP model is generated using the transformation strategy described in Section 5.3. All relevant SysML v2 elements such as ports, transitions, and connection constraints are mapped to their BIP equivalents. Additionally, trace outputs are embedded to monitor the evolution of component states during simulation.

We decompose the BIP model into its constituent parts for clarity: port types, atomic components, connectors, and compound system composition.

Port Type Definitions. Listing 5.15 defines the BIP port types used for communication between components.

Listing 5.15: Port Type Definitions in BIP

```
port type CommandPort_t(string commandSignal)
port type BroadcastPort_t(string commandSignal)
port type RendezVousPort_t()
```

Atomic Component: Drone. Listing 5.16 shows the BIP atomic component representing drone behavior.

Listing 5.16: Drone Component in BIP

```
atom type Drone(int id)
  data string commandSignal
  export port CommandPort_t commandIn(commandSignal)
  export port RendezVousPort_t rendezVous()
  place Idle, Ready, Flying, Formation
  initial to Idle do {
    commandSignal = "";}
  on commandIn from Idle to Ready do {
    commandSignal = "Ready";
    printf("Drone␣%d␣is␣Ready\n", id);}
  on commandIn from Ready to Flying do {
    commandSignal = "InAir";
    printf("Drone␣%d␣is␣Flying\n", id);}
  on commandIn from Flying to Formation do {
    commandSignal = "InFormation";
    printf("Drone␣%d␣is␣in␣Formation\n", id);}
  on rendezVous from Formation to Idle do {
    printf("Drone␣%d␣resets␣from␣Formation␣to␣Idle\n", id);}
end
```

Atomic Component: Command Station. The command logic is encoded in Listing 5.17.

Listing 5.17: StationDeCommande Component in BIP

```
atom type StationDeCommande()
  data string commandSignal
  data int phase
  data int compteur
  data int maxCycles
  export port BroadcastPort_t broadcastCommand(commandSignal)
  place Active, Stop
  initial to Active do {
```

```
    phase = 0; compteur = 0;
    maxCycles = 4;
    commandSignal = "";}
  on broadcastCommand from Active to Active provided (compteur <
     maxCycles) do {
    if (phase == 0) then
      commandSignal = "Start"; phase = 1;
    else  if (phase == 1) then
      commandSignal = "Takeoff"; phase = 2;
      else  if (phase == 2) then
          commandSignal = "Formation";
          phase = 0;
        fi fi fi
    compteur = compteur + 1;
    printf("Station: Sending %s Command\n", commandSignal); }
  internal from Active to Stop provided (compteur >= maxCycles) do {
    printf("Station: Max cycles reached. Stopping.\n", 0); }
end
```

Connector Definitions. The broadcast and rendezvous connectors are defined in Listings 5.18 and 5.19.

Listing 5.18: Broadcast Connector Definition in BIP

```
connector type Broadcast(BroadcastPort_t sender,
                         CommandPort_t receiver1,
                         CommandPort_t receiver2,
                         CommandPort_t receiver3)
  define sender' receiver1 receiver2 receiver3
  on sender receiver1 receiver2 receiver3 down {
    receiver1.commandSignal = sender.commandSignal;
    receiver2.commandSignal = sender.commandSignal;
    receiver3.commandSignal = sender.commandSignal; }
  on sender receiver1 down {
    receiver1.commandSignal = sender.commandSignal; }
end
```

Listing 5.19: RendezVous Connector Definition in BIP

```
connector type RendezVous(RendezVousPort_t d1,
                          RendezVousPort_t d2,
                          RendezVousPort_t d3)
  define d1 d2 d3
end
```

System Composition. Listing 5.20 shows the complete assembly of the system components in the ‘DroneSwarm‘ compound type.

Listing 5.20: Drone Swarm System Composition in BIP

```
compound type DroneSwarm()
  component StationDeCommande station()
  component Drone d1(1)
  component Drone d2(2)
  component Drone d3(3)
  connector Broadcast leaderCommand(station.broadcastCommand, d1.
     commandIn, d2.commandIn, d3.commandIn)
  connector RendezVous rendezvousSync(d1.rendezVous, d2.rendezVous,
     d3.rendezVous)
end
```

5.4.3 Simulation Results and Deadlock Resolution

The BIP model was executed to validate the coordination workflow. Figure 5.8 displays the resulting trace. The command station sequentially issues commands: `Start`, `Takeoff`, and `Formation`, leading drones through defined behavioral states. Once all drones reach the `Formation` state, a synchronized reset via rendez-vous returns them to the `Idle` state, confirming both interaction semantics and correct synchronization.

However, the initial configuration includes a stopping condition at the command station, triggered when a predefined number of cycles is reached. Once this threshold is met, no further interactions are possible, and the system enters a deadlock state.

To investigate this issue and evaluate its resolution, we modified the SysML v2 model to remove the stopping condition. The updated version of the `StationDeCommande` is shown in Listing 5.21, and its transformed BIP equivalent in Listing 5.22.

Listing 5.21: Modified Command Station Definition in SysML v2 (without stopping condition)

```
part def StationDeCommande {
    port stationCommand:BroadcastPort;
 state def StationStates {
    attribute phase : Integer = 0;
    state Active; entry; then Active;
    transition 'Active-Start'
        first Active
        accept Signal : String via stationCommand
        then Active { phase = 1; }
```

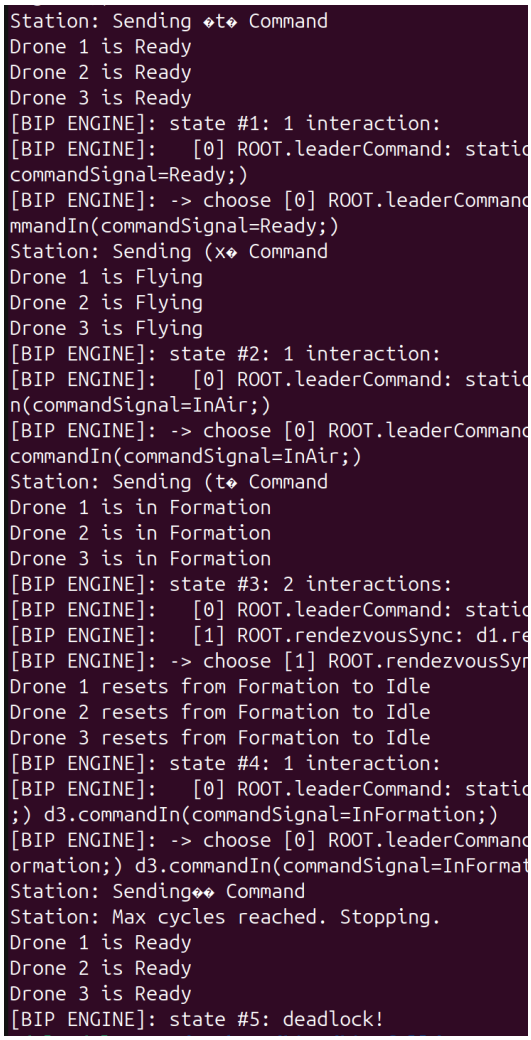

Fig. 5.8: Execution Trace of the BIP Model Showing State Transitions

```
    transition 'Active-Takeoff'
        first Active
        accept Signal : String via stationCommand
        then Active { phase = 2; }
    transition 'Active-Formation'
        first Active
        accept Signal : String via stationCommand
        then Active {phase = 0;}}}
```

Listing 5.22: Modified BIP Model for Continuous Execution

```
atom type StationDeCommande()
  data string commandSignal
  data int phase
```

```
  export port BroadcastPort_t broadcastCommand(commandSignal)
  place Active
  initial to Active do { phase = 0;
    commandSignal = ""; }
  on broadcastCommand from Active to Active do {
    if (phase == 0) then
      commandSignal = "Start";
      phase = 1;
    else if (phase == 1) then
        commandSignal = "Takeoff";
        phase = 2;
      else  if (phase == 2) then
        commandSignal = "Formation";
          phase = 0; fi fi fi
    printf("Station: Sending %s Command\n", commandSignal);}
end
```

Following this modification, the system performs cyclically without deadlock, demonstrating the importance of interaction structure and system-level behavior in early verification. This example highlights how structured modeling and formal execution can reveal and help correct latent coordination issues.

5.5 Conclusion

In this chapter, we presented a structured modeling and verification workflow that bridges high-level system modeling in SysML v2 with the formal semantics and analysis capabilities of the BIP framework. By focusing on two canonical interaction patterns, rendezvous and broadcast, we demonstrated how coordination logic in cyber-physical systems can be explicitly modeled, transformed, and verified.

Our methodology supports the use of native SysML v2 constructs, avoiding metamodel extensions while enabling formal execution through a well-defined transformation into BIP. This approach preserves both structural modularity and behavioral intent, facilitating early validation of design choices.

Through a case study involving swarm drone coordination, we illustrated the expressiveness and effectiveness of this modeling pipeline. The study highlighted how BIP simulation and trace analysis help identify design flaws, such as deadlock conditions, and guide model correction through incremental refinement. The corrected version of the command station model eliminated the stopping condition and restored continuous, deadlock-free execution.

This contribution complements other aspects of this dissertation by reinforcing the importance of structured interaction modeling and formal verification in the design of dependable CPS. The next chapter focuses on the modeling and verification of fuzzy requirements, addressing the challenges of handling imprecise or context-dependent specifications within the SysML v2 framework.

Chapter 6

Direct Modeling and Scenario-Based Verification of Fuzzy Requirements

6.1 Introduction

Cyber-Physical Systems (CPS) are increasingly deployed across critical domains such as intelligent transportation, energy management, aerospace, and smart buildings. These systems tightly integrate software components with physical processes and are subject to a wide range of requirements, both functional and non-functional. Unlike functional requirements, many non-functional requirements are inherently vague, context-dependent, or subjective. For example, a "comfortable temperature" or "low power usage" cannot be universally defined, and their interpretation often depends on environmental factors and human perception.

Model-Based Systems Engineering (MBSE) provides a unified framework to manage the complexity of CPS through formalized system specification, traceability, and early-stage verification. The recent evolution of the Systems Modeling Language (SysML) to version 2 (SysML v2) offers substantial improvements in semantic precision, modularity, and native support for constraint and expression evaluation. Despite these advancements, requirement verification in SysML v2 remains grounded in classical Boolean logic, where constraints are considered either satisfied or violated. This binary interpretation is inadequate when dealing with soft requirements that inherently admit partial satisfaction.

To address this limitation, we introduce a novel approach that integrates fuzzy logic into SysML v2 models using only native constructs. Our method enables engineers to specify fuzzy semantics through standard elements such as `calc def`, `attribute`, `requirement`, and `constraint`. Satisfaction degrees are computed using trapezoidal membership functions, producing values in the continuous range $[0, 1]$, and constraints assert thresholds for acceptability. This leads to a verification process that is not only more expressive and traceable but also provides better explainability when dealing with vague or imprecise requirements.

To complement this in-model evaluation, we propose an external batch-based simulation technique. This extension allows engineers to evaluate the satisfaction of fuzzy requirements under varying runtime conditions by generating multiple configurations through Python-based simulations. The resulting analyses help quantify system robustness, identify sensitivity to environmental changes, and guide design-space exploration in early development phases.

Together, these contributions demonstrate that SysML v2 can be effectively leveraged to support fuzzy requirement verification without modifying its metamodel or relying on external formal reasoning engines. This chapter builds upon the modeling foundations introduced in Chapter 3 and illustrates the proposed methodology through a case study involving a smart building HVAC system.

6.2 Modeling and Evaluating Fuzzy Requirements in SysML v2

This section presents our modeling approach for representing and verifying fuzzy requirements using only the standard constructs provided by SysML v2. Unlike previous extensions that modify the metamodel or rely on external reasoning engines, our method remains fully compliant with SysML v2's native capabilities. We begin by detailing how fuzzy semantics are implemented using reusable calculation definitions and constraints. We then illustrate the approach through a representative case study involving a smart building HVAC system, demonstrating that SysML v2 supports graded verification logic through its built-in expression evaluation.

6.2.1 Fuzzy Semantics with Native Constructs

To represent imprecise or context-dependent requirements, we adopt a fuzzy logic framework, where requirements are no longer interpreted as simply satisfied or not,

but instead evaluated along a continuum. Specifically, we use trapezoidal membership functions to map real-valued attributes (e.g., temperature, power consumption) to satisfaction degrees in the interval $[0,1]$. These values indicate the degree to which a particular requirement is fulfilled.

In our implementation, trapezoidal membership functions are defined using the `calc def` construct in SysML v2, which enables the definition of reusable logic applicable to multiple requirements. The function accepts the four parameters a, b, c, d that define the fuzzy interval, and computes the membership degree for an observed input x. The full definition of this reusable function is provided in Listing 6.1.

Listing 6.1: Trapezoidal Membership Function in SysML v2

```
calc def trapezoid {
    in x: Real;
    in a: Real;
    in b: Real;
    in c: Real;
    in d: Real;
    return r: Real;
    if x < a or x > d ? 0
    if x > a and x < b ? ((x - a) / (b - a))
    if x >= b and x <= c ? 1
    if x > c and x < d ? ((d - x) / (d - c))
    if x >= d ? 0
}
```

This function is then referenced in requirement blocks to calculate satisfaction degrees. Each requirement includes the observed input, the corresponding computed satisfaction degree, and a constraint that checks whether the degree exceeds a minimum acceptable threshold. This modeling pattern enables in-model evaluation of vague requirements, maintaining full compatibility with SysML v2's expression evaluation engine.

6.2.2 HVAC Case Study and Expression Evaluation

To illustrate the feasibility and clarity of our approach, we apply it to a smart building Heating, Ventilation, and Air Conditioning (HVAC) system. The system must simultaneously meet two fuzzy requirements: maintaining a thermally comfortable environment and ensuring energy efficiency.

The model defines an HVAC component with two attributes: temperature and power consumption. Two requirement blocks encode the fuzzy constraints using the

previously defined trapezoidal membership function. These requirements express tolerances over the expected operating ranges and specify acceptance thresholds.

The complete SysML v2 representation of the HVAC system is shown in Listing 6.2.

Listing 6.2: HVAC Fuzzy Requirements in SysML v2

```
part def HvacSystem {
    attribute temperature: Real;
    attribute power: Real;
}

part hvac: HvacSystem {
    ref temperature = 27.0;
    ref power = 227.5;
}

requirement comfortReq {
    doc /* The ambient temperature shall be comfortable. */
    attribute mu_comfort : Real = trapezoid(hvac.temperature, 18.0,
       22.0, 24.0, 29.0);
    assert constraint ComfortConstraint { mu_comfort > 0.6 }
}

requirement powerReq {
    doc /* The power usage shall remain reasonably low. */
    attribute mu_power : Real = trapezoid(hvac.power, 0.0, 100.0,
       200.0, 300.0);
    assert constraint PowerConstraint { mu_power > 0.6 }
}

satisfy comfortReq by hvac;

satisfy powerReq by hvac;
```

Figures 6.1 and 6.2 visualize the model before and after expression evaluation, illustrating how satisfaction degrees are calculated and interpreted directly within the SysML v2 environment.

Following the expression-based evaluation in SysML v2, the computed satisfaction degrees for the HVAC system are as follows:

- Comfort satisfaction degree: $\mu_{\text{comfort}} = 0.4$ (requirement **not satisfied**).
- Energy satisfaction degree: $\mu_{\text{power}} = 0.725$ (requirement **satisfied**).

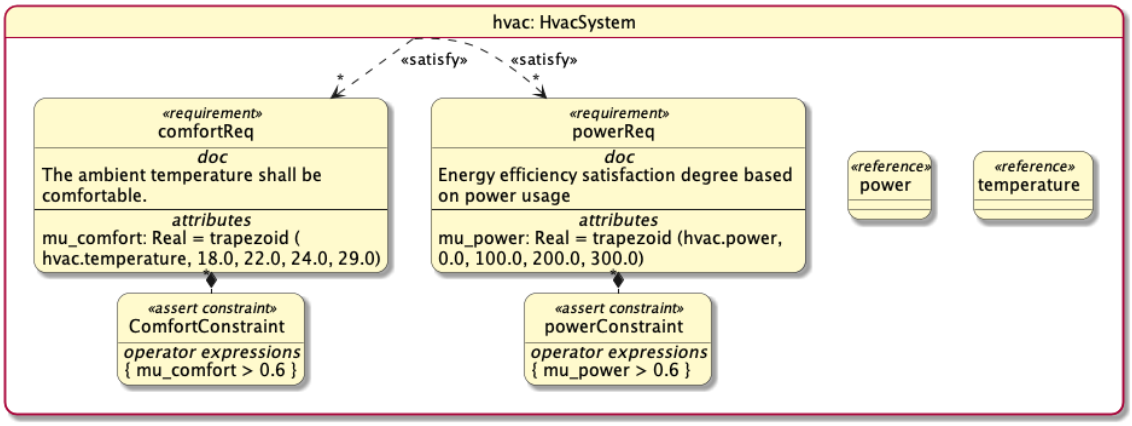

Fig. 6.1: SysML v2 Model Before Evaluation (Fuzzy Expressions)

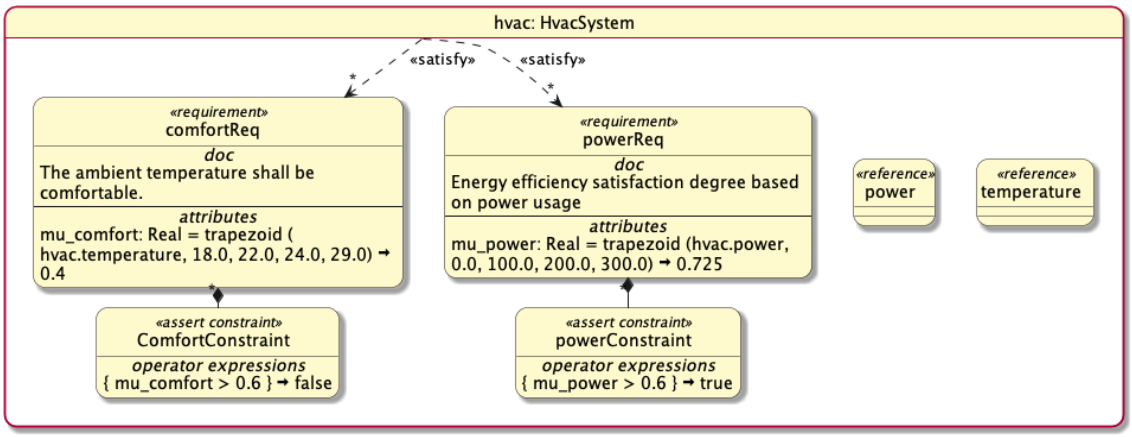

Fig. 6.2: SysML v2 Model After Evaluation (Computed Satisfaction Degrees)

These results indicate that the current temperature lies outside the defined fuzzy comfort zone, whereas energy consumption remains within the acceptable fuzzy interval.

6.2.3 Discussion

The integration of fuzzy requirement evaluation directly into SysML v2 brings several key benefits. First, it enables the precise and traceable modeling of soft requirements without relying on external plug-ins or altering the metamodel. Second, the computed satisfaction degrees provide designers with a quantitative view of requirement fulfillment, enabling more nuanced trade-off decisions early in the design cycle. Lastly, the use of native constraints supports seamless integration with SysML toolchains, preserving model consistency and reducing verification overhead.

Despite these advantages, this expression-based approach focuses on evaluating a single configuration. It does not address variability or the stochastic nature of CPS environments. In the next section, we extend the analysis by simulating system behavior under different runtime conditions using a batch-based exploration strategy.

6.3 Batch-Based Scenario Exploration

While the expression-based evaluation of fuzzy requirements in SysML v2 provides a precise and explainable verification mechanism, it is limited to assessing the system under a single, fixed configuration. However, in practice, Cyber-Physical Systems (CPS) operate in dynamic and uncertain environments. Environmental conditions (e.g., ambient temperature, occupancy), user behavior, and sensor variability can significantly affect system performance. Therefore, evaluating soft requirements across a diverse range of scenarios is essential to assess the robustness and reliability of the system design.

To address this limitation, we propose a complementary batch-based scenario exploration approach. This method extends the SysML v2 model by simulating the evaluation of fuzzy requirements across multiple runtime scenarios. These scenarios are generated externally, using Python, by sampling from probability distributions representing environmental variability. The key idea is to export only the essential elements from the SysML v2 model (such as trapezoidal membership functions and constraints), and reuse them in an external simulation framework without altering the model semantics.

6.3.1 Scenario Generation and Execution

We define three representative usage scenarios for the HVAC system, each reflecting different operational conditions:

- **S1 – Mild and stable:** Environmental parameters remain close to nominal values with minimal variability.
- **S2 – Variable conditions:** Mean values are similar to S1, but variability is increased to simulate more realistic fluctuations.
- **S3 – Stress scenario:** Conditions represent a worst-case environment with higher temperature and energy usage.

To simulate and evaluate the satisfaction of fuzzy requirements under these conditions, we implemented a Python script (Listing 6.3) that mirrors the trapezoidal membership functions defined in the SysML v2 model. This ensures semantic consistency between the requirements specification and the simulation logic.

The core of the script is the `trapezoid` function, which implements a standard trapezoidal membership curve. Two domain-specific fuzzy predicates are defined:

- `comfort_mu(temp)`: computes the degree of comfort based on the temperature.
- `power_mu(power)`: evaluates the degree of acceptable power consumption.

The scenarios (`S1`, `S2`, and `S3`) are instantiated using normally distributed random values for temperature and power to reflect different usage conditions. For each scenario, the script computes the degree of satisfaction for both fuzzy predicates across all samples.

Finally, the script outputs the average satisfaction levels for each scenario. These results provide a quantitative estimation of how well the system fulfills soft requirements under various environmental conditions. The use of simulation complements the model-based specification by enabling robustness and variability analysis through concrete execution traces.

Listing 6.3: Python-based Fuzzy Evaluation Script

```
import numpy as np
# Trapezoidal membership function
def trapezoid(a, b, c, d, x):
    if x <= a or x >= d: return 0.0
    elif a < x < b: return (x - a) / (b - a)
    elif b <= x <= c: return 1.0
    elif c < x < d: return (d - x) / (d - c)
    else: return 0.0
# Membership functions
def comfort_mu(temp): return trapezoid(18, 22, 24, 29, temp)
def power_mu(power): return trapezoid(0, 100, 200, 300, power)
# Define scenarios
scenarios = {
    "S1": {"temp": np.random.normal(23, 1, 10), "power": np.random.
       normal(120, 10, 10)},
    "S2": {"temp": np.random.normal(23, 3, 10), "power": np.random.
       normal(120, 30, 10)},
    "S3": {"temp": np.random.normal(27, 1, 10), "power": np.random.
       normal(250, 15, 10)}
}
```

```
# Evaluate
for name, sc in scenarios.items():
    comfort = [comfort_mu(t) for t in sc["temp"]]
    energy = [power_mu(p) for p in sc["power"]]
    print(f"{name}: Avg Comfort = {np.mean(comfort):.3f}, Avg Energy
        = {np.mean(energy):.3f}")
```

6.3.2 Interpretation and Robustness Analysis

The simulation produces average satisfaction degrees for each fuzzy requirement under every scenario. These results help evaluate how well the system maintains its goals under changing conditions:

- Under **S1**, both comfort and energy requirements are well satisfied, indicating robustness in nominal conditions.
- Under **S2**, the increased input variability causes a notable reduction in comfort satisfaction, while energy remains acceptable.
- Under **S3**, both satisfaction degrees drop significantly, exposing the system's vulnerability in extreme environments.

Figure 6.3 visualizes the average satisfaction degrees computed for each scenario, offering insight into the system's behavior under uncertainty.

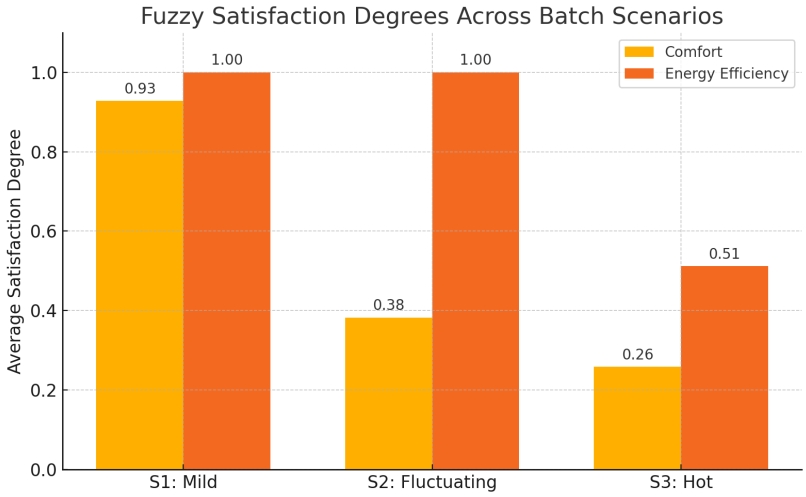

Fig. 6.3: Average fuzzy satisfaction degrees across batch-generated scenarios.

These observations can guide design decisions such as threshold adjustment, membership tuning, or the integration of adaptive control strategies.

6.3.3 Discussion

The batch-based scenario evaluation enhances SysML v2's expressiveness by supporting simulation-based analysis without requiring any change to the original model. The external Python script reuses the same fuzzy semantics defined within the SysML v2 model, ensuring consistency. Although performed outside the modeling tool, the results can be reintegrated to inform model evolution and to support trade-off and sensitivity analysis.

This hybrid approach allows engineers to assess the robustness and flexibility of CPS designs with respect to environmental fluctuations. It complements deterministic model evaluation with statistical insight and bridges the gap between structure-level consistency and runtime behavior analysis. In the next section, we conclude this contribution by summarizing the key insights and outlining potential directions for extension.

6.4 Conclusion

In this chapter, we introduced a model-based approach for specifying and verifying fuzzy requirements within the SysML v2 framework. Unlike traditional Boolean verification mechanisms, our method enables the formalization of soft, vague, and context-sensitive requirements using only native modeling constructs, namely, attributes, reusable calculation definitions, and constraints.

We demonstrated that fuzzy semantics can be effectively encoded using trapezoidal membership functions, allowing the evaluation of satisfaction degrees in the continuous interval $[0, 1]$. These values provide a nuanced, quantitative perspective on requirement satisfaction, which is particularly useful for early-stage design refinement and trade-off analysis. Through a representative case study involving a smart building HVAC system, we showed how SysML v2's native expression engine supports this evaluation directly within the model, without requiring metamodel extensions or external verification tools.

To address the limitations of static configuration evaluation, we extended our method with a lightweight scenario-based simulation mechanism. This complementary technique leverages a Python-based framework to simulate multiple environmental conditions, evaluate fuzzy satisfaction degrees under uncertainty, and assess system robustness across a spectrum of usage profiles. The batch evaluation results demonstrated how variability can impact satisfaction, revealing design vulnerabilities and guiding requirements threshold tuning.

Together, these contributions offer a unified and lightweight methodology for verifying fuzzy requirements in CPS. Our approach remains fully compliant with SysML v2, preserves model integrity, and supports both deterministic evaluation and variability-aware analysis. It thereby enhances the modeling and verification capabilities of MBSE workflows for CPS, particularly when dealing with qualitative performance aspects.

In the next chapter, we summarize the key contributions of this dissertation and outline future directions for extending our approach. These include the formalization of multi-objective fuzzy requirements, integration with co-simulation platforms, support for real-time monitoring of satisfaction at runtime, and application to additional domains such as robotics, healthcare, and smart energy systems.

Part IV

Conclusion and Future Work

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This dissertation has addressed several key challenges in the modeling and verification of Cyber-Physical Systems (CPS), with a focus on improving early correctness assurance through model-based techniques. In particular, it has proposed and evaluated three complementary verification approaches targeting structural consistency, dynamic interaction correctness, and the satisfaction of fuzzy or soft requirements.

First, a structural consistency verification method was introduced to detect architectural inconsistencies in SysML v1 models. This approach relies on a formalization of consistency rules derived from the structural semantics of block, port, and requirement diagrams. A dedicated verification tool was developed to automate the checking of these rules, and the approach was validated on the CyCab case study, illustrating its practical relevance for early detection of design errors.

Second, to address the verification of dynamic interactions, a novel integration between SysML v2 and the Behavior-Interaction-Priority framework was proposed. This method leverages the expressiveness of SysML v2 for component definition and the formal execution semantics of BIP for interaction analysis. A mapping from SysML v2 constructs to BIP was defined to enable model transformation, simulation, and verification of concurrent coordination patterns. The approach was applied to a swarm drone coordination system, demonstrating how BIP's interaction and priority layers can be used to detect synchronization issues, deadlocks, and scheduling anomalies.

Third, this work tackled the challenge of verifying imprecise or context-dependent requirements using fuzzy logic. A framework was developed for expressing fuzzy requirements directly within SysML v2 models, enabling both native expression-based evaluation and scenario-based simulation. A batch simulation tool was implemented

to explore system behavior under uncertainty, and the method was illustrated on an HVAC control system, showing how design decisions can be guided by degrees of satisfaction rather than binary constraints.

Taken together, these contributions offer a unified model-based verification workflow that integrates structural, behavioral, and non-functional aspects of CPS. The methods proposed in this dissertation enhance the rigor, automation, and coverage of verification tasks in early design stages, while remaining compatible with SysML-centered MBSE practices.

7.2 Future Work

This dissertation opens several promising directions for future research. Among them, three perspectives appear particularly relevant and impactful.

- **Refinement-based verification of dynamic behavior.** A first perspective is to extend the concept of refinement beyond structural consistency by focusing on dynamic behaviors. While the first contribution of this thesis verified whether a composition of components was structurally consistent with an abstract block, future work could aim to determine whether a composition satisfies the expected dynamic properties of an abstract specification. This involves reasoning about the temporal and interaction semantics of the system. The BIP framework, with its formal support for specifying and analyzing interactions and execution priorities, offers a promising foundation for developing such a refinement-based verification methodology.

- **Incremental verification strategies for evolving CPS models.** Cyber-Physical Systems are often developed iteratively, with frequent refinements and architectural changes. A compelling future research direction is to design verification strategies that support incremental evolution of models, avoiding the need to re-verify the entire system after each modification. This involves identifying verification artifacts that can be reused, detecting impacted elements, and isolating submodels for localized analysis. Combined with SysML v2's modular structure and BIP's compositional semantics, such strategies could enable scalable and efficient verification workflows that align with modern iterative development practices.

- **Adaptive verification of fuzzy requirements based on system context.** This perspective builds on the work conducted around fuzzy requirements by

introducing adaptability in how these requirements are evaluated. Rather than relying on fixed thresholds or static membership functions, future approaches could incorporate mechanisms that adjust these parameters in response to contextual information. For instance, a system could relax certain constraints when operating in degraded modes or tighten them in safety-critical conditions. This adjustment may depend on real-time sensor data, performance indicators, or environmental changes. Such an approach would enhance the relevance and realism of requirement evaluation, particularly in systems that must operate autonomously or adapt to changing conditions. It would support the development of context-aware Cyber-Physical Systems that continuously align their behavior with evolving operational needs and user expectations.

References

[1] Alur, R. and Dill, D. L. (1994). A theory of timed automata. *Theoretical computer science*, 126(2):183–235.

[2] Alvarez-Alvarado, M. S., Apolo-Tinoco, C., Ramirez-Prado, M. J., Alban-Chacón, F. E., Pico, N., Aviles-Cedeno, J., Recalde, A. A., Moncayo-Rea, F., Velasquez, W., and Rengifo, J. (2024). Cyber-physical power systems: A comprehensive review about drivers, standards, and future perspectives. *Standards, and Future Perspectives*.

[3] Baille, G., Garnier, P., Mathieu, H., and Pissard-Gibollet, R. (1999). *The INRIA Rhône-Alpes CyCab*. Technical report, INRIA. Describes the package natbib.

[4] Baresi, L., Pasquale, L., and Spoletini, P. (2010). Fuzzy goals for requirements-driven adaptation. In *2010 18th IEEE international requirements engineering conference*, pages 125–134. IEEE.

[5] Basu, A., Bozga, M., and Sifakis, J. (2006). Modeling heterogeneous real-time components in bip. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, pages 3–12. Ieee.

[6] Behrmann, G., David, A., and Larsen, K. G. (2004). A tutorial on uppaal. *Formal methods for the design of real-time systems*, pages 200–236.

[7] Bhave, A., Krogh, B. H., Garlan, D., and Schmerl, B. (2011). View consistency in architectures for cyber-physical systems. In *2011 IEEE/ACM second international conference on cyber-physical systems*, pages 151–160. IEEE.

[8] Bubenko, J., Rolland, C., Loucopoulos, P., and DeAntonellis, V. (1994). Facilitating" fuzzy to formal" requirements modelling. In *Proceedings of IEEE International Conference on Requirements Engineering*, pages 154–157. IEEE.

[9] Carrillo, O., Chouali, S., and Mountassir, H. (2013). Incremental modeling of system architecture satisfying sysml functional requirements. In *Formal Aspects of Component Software - 10th International Symposium, FACS 2013, Nanchang, China, October 27-29, 2013, Revised Selected Papers*, pages 79–99.

[10] Chanda, J., Kanjilal, A., Sengupta, S., and Bhattacharya, S. (2009). Traceability of requirements and consistency verification of uml use case, activity and class diagram: A formal approach. In *2009 Proceeding of International Conference on Methods and Models in Computer Science (ICM2CS)*, pages 1–4. IEEE.

[11] Cimatti, A., Dorigatti, M., and Tonetta, S. (2013). Ocra: A tool for checking the refinement of temporal contracts. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 702–705. IEEE.

[12] Cofer, D., Gacek, A., Miller, S., Whalen, M. W., LaValley, B., and Sha, L. (2012). Compositional verification of architectural models. In *NASA Formal Methods: 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings 4*, pages 126–140. Springer.

[13] Dagli, C. H., Singh, A., DAUBY, J. P., and Wang, R. (2009). Smart systems architecting: computational intelligence applied to trade space exploration and system design. In *Systems Research Forum*, volume 3, pages 101–119. World Scientific.

[14] Dassault Systèmes (2025). Cameo systems modeler. https://www.3ds.com/products/catia/no-magic/cameo-systems-modeler. Accessed: 2025-06-17.

[15] Demachy, R. and Guilmeau, S. (2022). Structural consistency of MBSE and MBSA models using Consistency Links. In *11th European Congress Embedded Real Time System (ERTS 2022)*, Toulouse, France.

[16] DionisioParaiba, J. and Martins, L. E. G. (2013). A proposal of requirements specification process for adaptive systems based on fuzzy logic and nfr-framework.

[17] Do, R. (1992). 178b/ed-12b. *Software Considerations in Airborne Systems and Equipment Certification". RTCA*.

[18] Dokter, K., Jongmans, S.-S., Arbab, F., and Bliudze, S. (2015). Relating bip and reo. *arXiv preprint arXiv:1508.04848*.

[19] Eclipse Foundation (2025). Eclipse papyrus. https://eclipse.dev/papyrus/index.html. Accessed: 2025-06-17.

[20] Egesoy, A. and Güzel, A. (2021). Fuzzy logic support for requirements engineering. *International Journal of Innovative Research in Computer Science & Technology*, 9(2):14–21.

[21] Feiler, P. H., Gluch, D. P., and Hudak, J. (2006). The architecture analysis & design language (aadl): An introduction.

[22] Graja, I., Kallel, S., Guermouche, N., Cheikhrouhou, S., and Hadj Kacem, A. (2020). A comprehensive survey on modeling of cyber-physical systems. *Concurrency and Computation: Practice and Experience*, 32(15):e4850.

[23] Group, O. et al. (2007). Modeling and analysis of real-time and embedded systems (marte).

[24] Gunes, V., Peter, S., Givargis, T., and Vahid, F. (2014). A survey on concepts, applications, and challenges in cyber-physical systems. *KSII Transactions on Internet and Information Systems (TIIS)*, 8(12):4242–4268.

[25] Han, D., Yang, Q., and Xing, J. (2014). Extending uml for the modeling of fuzzy self-adaptive software systems. In *The 26th Chinese Control and Decision Conference (2014 CCDC)*, pages 2400–2406. IEEE.

[26] Han, D., Yang, Q., Xing, J., Li, J., and Wang, H. (2016). Fame: A uml-based framework for modeling fuzzy self-adaptive software. *Information and Software Technology*, 76:118–134.

[27] Haque, S. A., Aziz, S. M., and Rahman, M. (2014). Review of cyber-physical system in healthcare. *international journal of distributed sensor networks*, 10(4):217415.

[28] Henzinger, T. A. (1996). The theory of hybrid automata. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, pages 278–292. IEEE.

[29] Incer, I., Badithela, A., Graebener, J., Mallozzi, P., Pandey, A., Yu, S.-J., Benveniste, A., Caillaud, B., Murray, R. M., Sangiovanni-Vincentelli, A., et al. (2023). Pacti: Scaling assume-guarantee reasoning for system analysis and design. *arXiv preprint arXiv:2303.17751*.

[30] INTO-CPS Project (2025). Into-cps project website. https://projects.au.dk/into-cps/. Accessed: 2025-06-17.

[31] Lee, E. (2012). Copyright cg2011-2012 edward ashford lee & sanjit arunkumar seshia all rights reserved.

[32] Lee, E. A. and Seshia, S. A. (2017). *Introduction to embedded systems: A cyber-physical systems approach*. MIT press.

[33] Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Sivaramakrishnan, K., and Vouillon, J. (2023). *The OCaml system release 5.1: Documentation and user's manual*. PhD thesis, Inria.

[34] Li, X., Liu, Z., and Jifeng, H. (2004). A formal semantics of uml sequence diagram. In *2004 Australian Software Engineering Conference. Proceedings.*, pages 168–177. IEEE.

[35] Liu, X. F. (1998). Fuzzy requirements. *IEEE Potentials*, 17(2):24–26.

[36] Lu, Y. (2017). Cyber physical system (cps)-based industry 4.0: A survey. *Journal of Industrial Integration and Management*, 2(03):1750014.

[37] Ma, Z. M., Yan, L., and Zhang, F. (2012). Modeling fuzzy information in uml class diagrams and object-oriented database models. *Fuzzy Sets and Systems*, 186(1):26–46.

[38] Magureanu, G., Gavrilescu, M., and Pescaru, D. (2013). Validation of static properties in unified modeling language models for cyber physical systems. *Journal of Zhejiang University SCIENCE C*, 14(5):332–346.

[39] Minsky, Y. (2011). Ocaml for the masses. *Communications of the ACM*, 54(11):53–58.

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

[41] Naufal, J. K., Camargo, J. B., Vismari, L. F., de Almeida, J. R., Molina, C., González, R. I. R., Inam, R., and Fersman, E. (2017). A 2 cps: A vehicle-centric safety conceptual framework for autonomous transport systems. *IEEE Transactions on Intelligent Transportation Systems*, 19(6):1925–1939.

[42] Nussbaumer, C. J. M. and Kieliger, L. (2017). Bidirectional transformation between bip and sysml for visualisation and editing.

[43] Object Management Group (2021). Semantics of a Foundational Subset for Executable UML Models (fUML) Version 1.5. https://www.omg.org/spec/FUML/1.5/About-FUML. Accessed: 2025-06-17.

[44] Object Management Group (OMG) (2012). OMG Systems Modeling Language SysML. Technical report.

[45] Object Management Group (OMG) (2024). Systems Modeling Language (SysML) v2 Beta 2 Specification: Language. https://www.omg.org/spec/SysML/2.0/Beta2/Language/PDF. Accessed Mars 2025.

[46] OMG, O. M. G. (1999). Unified modeling language specification UML, version 1.3 ©. Technical report.

[47] Opranescu, V. and Ionita, A. D. (2025). Review of cyber-physical systems modeling with uml, sysml and marte (november 2024). *IEEE Access*.

[48] Pedroza, G., Apvrille, L., and Knorreck, D. (2011). Avatar: A sysml environment for the formal verification of safety and security properties. In *2011 11th Annual International Conference on New Technologies of Distributed Systems*, pages 1–10. IEEE.

[49] Peterson, J. L. (1977). Petri nets. *ACM Computing Surveys (CSUR)*, 9(3):223–252.

[50] Rajkumar, R., Lee, I., Sha, L., and Stankovic, J. (2010). Cyber-physical systems: the next computing revolution. In *Proceedings of the 47th design automation conference*, pages 731–736.

[51] Tannoury, P., Chouali, S., and Hammad, A. (2022a). An incremental model-based design methodology to develop cps with sysml/ocl/reo. In *Journées du GDR GPL, Jun 2022, Vannes*.

[52] Tannoury, P., Chouali, S., and Hammad, A. (2022b). Model driven approach to design an automotive cps with sysreo language. In *Proceedings of the 20th ACM International Symposium on Mobility Management and Wireless Access*, pages 97–104.

[53] Taratuknin, V. and Yadgarova, Y. (2015). A fuzzy logic approach for product configuration and requirements management. In *2015 Annual Conference of the North American Fuzzy Information Processing Society (NAFIPS) held jointly with 2015 5th World Conference on Soft Computing (WConSC)*, pages 1–5. IEEE.

[54] Torre, D., Labiche, Y., Genero, M., and Elaasar, M. (2018). A systematic identification of consistency rules for uml diagrams. *Journal of Systems and Software*, 144:121–142.

[55] VERIMAG (2015). *BIP2 Documentation: Language Reference Guide*. Release 2015.04 (RC7). Available at https://www-verimag.imag.fr/TOOLS/DCS/bip/doc/latest/pdf/BIP2.pdf.

[56] Vidalie, J., Batteux, M., Mhenni, F., and Choley, J.-Y. (2022). Category Theory Framework for System Engineering and Safety Assessment Model Synchronization Methodologies. *Applied Sciences*, 12(12):5880.

[57] Warmer, J. and Kleppe, A. (1998). *The object constraint language: precise modeling with UML*. Addison-Wesley Longman Publishing Co., Inc.

[58] Yoo, Y.-Y. and Lee, J.-C. (2019). The improvement of maintainability evaluation method at system level using system component information and fuzzy technique. *Journal of the Korea Academia-Industrial cooperation Society*, 20(3):100–109.

[59] Zadeh, L. A. (1965). Fuzzy sets. *Information and control*, 8(3):338–353.