vehicles

MDPI

Article

Managing Design Variants in Formula Student Race Cars: A Digital Engineering Approach Across Multiple Teams

Julian Borowski [1,*], Hinrich Emsmann [2], Jannis Kneule [3], Rico Ruess [4] and Stephan Rudolph [5]

[1] IILS Ingenieurgesellschaft für Intelligente Lösungen und Systeme mbH, 70771 Leinfelden-Echterdingen, Germany
[2] Raceyard Kiel, Formula Student Team University of Applied Sciences Kiel, 24149 Kiel, Germany; hinrich.emsmann@raceyard.de
[3] Rennschmiede Pforzheim, Formula Student Team University of Applied Sciences Pforzheim, 75175 Pforzheim, Germany; jannis.kneule@rennschmiede-pforzheim.de
[4] FaSTDa Racing, Formula Student Team University of Applied Sciences Darmstadt, 64295 Darmstadt, Germany; rico.ruess@fastda-racing.de
[5] Design Theory and Similarity Mechanics Group, Institute of Aircraft Design, University of Stuttgart, 70569 Stuttgart, Germany; rudolph@ifb.uni-stuttgart.de
* Correspondence: julian.borowski@iils.de

Abstract

Increasing product complexity, shorter development cycles and cross-domain integration demands pose significant challenges for modern race car engineering teams. In Formula Student teams, heterogeneous toolchains, manual data exchange, late system integration, and high personnel turnover hinder efficient collaborative development and lead to repeated knowledge loss. This paper presents an integrated digital-engineering framework combining graph-based design languages (GBDL), model-to-text transformations, natural-language interactions via Large Language Models (LLMs), and Git-based version control to address these issues. By formalizing design knowledge and storing it in a centralized design graph, the framework ensures digital consistency of data and models, supports automated vehicle design variant generation, and enables seamless cross-domain integration. Through case studies of three Formula Student teams, the methodology demonstrates quantifiable reductions in design iteration time, enabling the evaluation of more than 10^4 suspension variants within days instead of a few dozen manually created variants, while reducing hands-on engineering effort from minutes per variant to a largely unattended optimization process. The results indicate that the approach not only enhances efficiency and collaboration but also preserves design knowledge for long-term knowledge management and reuse. Looking forward, this methodology provides a scalable route toward further engineering automation, systematic variant-driven development, and early-stage design optimization supported by design languages and integrated downstream toolchains.

Keywords: graph-based design language; design automation; digital engineering; Formula Student; systems engineering; design compiler; variant management; Large Language Model (LLM); design graph; Git version control

Academic Editor: Pak Kin Wong

Received: 18 December 2025
Revised: 13 February 2026
Accepted: 14 February 2026
Published: 23 February 2026

1. Introduction

The ongoing evolution of vehicle development is characterized by a marked increase in system complexity, a significant reduction in development timelines, and a rising demand for cross-domain integration [1–3]. These challenges are particularly pronounced

in the context of Formula Student (FS), where teams must design and manufacture high-performance vehicles under severe resource limitations and demanding requirements for safety, dynamic behavior, and system integration [4,5]. To achieve this objective, teams must handle mechanical, electrical, aerodynamic, and control sub-systems concurrently, often under stringent time constraints and with high staff turnover [6]. Such conditions further boost the existing fragmentation of tool landscapes, manual data exchange, and limited process interoperability among engineering sub-teams, ultimately impeding efficient system-level integration and the preservation of design knowledge [7].

As a way out of this situation, graph-based design languages (GBDLs) offer a formal, machine-readable and machine-executable framework for representing and processing engineering knowledge to address these challenges [8]. GBDLs encode design vocabulary, associated rules, and production systems within a unified, graph-based data model, which enables the systematic capture of system architecture, parameters, constraints, and interdisciplinary dependencies [9]. The resulting design graph serves as a central, tool-independent data model that ensures digital continuity across computer-aided design (CAD), numerical simulation, and design analysis from a single source of truth. This mitigates information loss, improves consistency, and supports scalable variant generation [10].

Despite these strategic advantages of GBDLs, adopting them in fast-paced, multidisciplinary development environments remains challenging. The underlying formalisms, which range from object-oriented modeling concepts to ontology-based knowledge representations and model-to-model transformations, typically require substantial prior knowledge [8]. This poses a particular barrier in FS teams, where members join with highly heterogeneous technical backgrounds, limited exposure to formal modeling methods, and the need to contribute effectively within a short timeframe [11].

Large Language Models (LLMs) offer a promising solution to bridge this gap [12]. The translation of the formal design graph into semantically rich text is a prerequisite for enabling natural-language interactions, allowing team members to ask design questions, trace dependencies, and explore cross-domain relationships using their familiar language, without the necessity of navigating the underlying graph structure or possessing knowledge of a graph query language [13]. This natural-language interface reduces the barrier to access the digital design model, accelerates decision-making, supports knowledge transfer between sub-teams, and enables early detection of integration issues [14].

In this paper, an integrated framework for Formula Student race car development is presented. Sections 3–5 first summarize the fundamental concepts of graph-based design languages, natural-language interaction with design graphs, and version control for graph-based engineering models based on the existing literature. Building on this foundation, the framework consists of a design compiler that utilizes graph-based design languages for modeling, uses the design graph for retrieval augmented context assembly to expose design semantics to an LLM and to enable natural-language queries for team members independently of their modeling expertise, and provides a Git-based version control to ensure traceability and maintainability of the evolving design models. The proposed concept is demonstrated through case studies from three FS teams and examines its benefits and limitations. Additionally, the potential of the concept to enhance vehicle design processes in highly collaborative and rapid-development environments is highlighted and discussed.

2. Challenges in Formula Student Race Car Development

The development of a FS race car is shaped by a highly fragmented tool landscape and a pronounced division of engineering tasks across multiple sub-teams [15]. Although this organizational structure is necessary to handle the complexity of modern race car

development, it introduces significant challenges in the overall vehicle design process. These challenges arise primarily from tool heterogeneity, manual data exchange, limited interoperability, short development cycles, and recurring knowledge loss due to high staff turnover. Together, these factors create a development environment with substantial inefficiencies and a high sensitivity to design inconsistencies [6].

2.1. Fragmented Toolchains and Limited Interoperability

FS teams rely on a wide range of domain-specific software tools, including CAD environments (e.g., Dassault [16], Siemens [17], Parametric Technology [18]), CFD for aerodynamic analysis, FEM for structural analysis, MBS simulations for suspension design, thermal and electrical system tools, and lap time simulations for overall vehicle evaluation [19]. Since each sub-team selects its own tools according to its respective domain, the resulting toolchain lacks digital continuity and interoperability. There are few interfaces between tools, meaning information must be transferred manually. For example, tire data evaluated in dedicated tire analysis tools must be manually entered into the global lap time simulator, and essential vehicle parameters such as mass distribution, gear ratios, torque curves and aerodynamic coefficients must be repeatedly re-entered into different tools. Furthermore, suspension kinematics are analyzed in MBS software, while packaging and collision checks are performed separately in CAD environments. Structural components are designed in 3D-CAD, exported for FEM analysis and reintegrated manually after several design iterations. This manual exchange of data between loosely coupled tools introduces redundancy and inconsistency, as well as an elevated risk of human error [20].

2.2. Sub-Team Separation and Late System Integration

In order to manage system complexity, FS teams usually divide themselves into specialised sub-teams, including those responsible for the chassis, aerodynamics, vehicle dynamics, electric powertrain, and others [15]. While this organizational structure enables specialization, it also isolates requirements and design decisions within each domain. Interactions between subsystems are usually discovered only during late integration in the CAD environment [21]. Consequently, critical packaging conflicts often remain undetected until late in the design process, and insights from one domain (e.g., aerodynamics) are incorporated inconsistently or too late into system-level models. Early-stage design iterations also cannot fully account for cross-domain dependencies. A notable example of this is the repeated occurrence of collisions between the suspension and the powertrain that remain often undetected [6]. These inconsistencies arise because dynamic states of vehicle movements are difficult to visualize and are rarely checked comprehensively during manual CAD integration.

2.3. Manual Processes and Time-Critical Development Cycles

FS teams develop an entirely new vehicle every year, typically within an intensive three-month-long design phase. In these circumstances, manual tasks that do not provide engineering insights, such as repeated data transfers, file management, and manual consistency checks, consume an excessive amount of available time. This has several consequences, including reduced iteration depth due to data management overheads, significant rework when late-stage CAD integration reveals collisions or outdated data, and the difficulty of maintaining permanently accurate model versioning across sub-teams. The impact of these inefficiencies becomes apparent when unexpected failures occur late in the development process [6]. For example, undetected collisions between suspension A-Arms and powertrain components have already forced last-minute redesigns and part replacements, often involving suboptimal design compromises due to time constraints.

2.4. Knowledge Loss and High Team Turnover

FS teams experience continuous staff turnover when students join or leave. While the teams do store the geometry and simulation models from previous seasons, much of the implicit design knowledge, such as the design rationale, assumptions, constraints and experienced cross-domain interactions, is often lost [22]. In practice, new members must reconstruct design intent from incomplete documentation, since sub-teams often rebuild models manually instead of extending existing ones. Therefore, iterative improvements across seasons are limited by missing historical context, and 'dead' CAD or simulation models persist without their underlying design logic [21,22]. This lack of formalized design logic and knowledge reduces the reusability of previous development cycles and forces teams to repeatedly solve similar problems each year.

2.5. Increased System Complexity and Regulatory Constraints

FS race cars operate under strict and continuously evolving design rules and face unique performance requirements. The continuous update of the FS regulations often introduces changes in key constraint domains, such as design space limits for aerodynamic devices (e.g., for the rear wing between the 2025 and 2026 seasons [23]), which increases design uncertainty and drives up the degree of interdependent decision-making across several subsystems. Additionally, the growing number of teams developing electric vehicles [24], either alongside or instead of classical internal combustion engine vehicles, further increases system complexity due to new subsystems such as battery management, electric drives, and power electronics.

The overall vehicle system comprises tightly coupled subsystems, including the drivetrain, aerodynamics, thermal management and cooling systems, braking system, electronics and control systems, as well as software. These subsystems interact in a highly integrated manner: For example, aerodynamic downforce affects tire load distribution and cooling airflow, which in turn impacts brake and powertrain thermal behavior, while control algorithms manage traction and stability across a wide range of dynamic driving conditions [25,26]. This high degree of subsystem coupling is further challenged by the specific characteristics of FS tracks: Due to tight and highly curved track layouts, lateral dynamics have a stronger influence on lap time than longitudinal acceleration [15]. Consequently, selecting the right tires and optimizing the kinematics are critical for mechanical grip, and the aerodynamic devices must operate efficiently at relatively low speeds to deliver significant downforce. In addition, vehicles must exhibit predictable, driver-friendly handling, since the drivers are students with varying levels of experience. Reliability is also crucial due to the importance of the endurance event in the overall scoring [6]. These multifaceted constraints increase the complexity of design trade-offs, making manual and isolated engineering workflows increasingly unsuitable for the overall race car design task.

2.6. Summary of Key Challenges

The combination of heterogeneous tools, manual data handling, isolated sub-team workflows, short design cycles, and recurring knowledge loss leads to:

- A lack of digital model consistency, software interoperability, and process continuity.
- Error-prone manual integration steps.
- Limited scalability of the engineering processes.
- Inefficiencies that reduce available time for design exploration.
- Repeated loss of engineering design and integration knowledge across seasons.

These challenges highlight the need for a formalized, holistic and digitally integrated approach that enables interoperability, automation, reuse of design knowledge, and consistent system-level verification throughout the entire development process.

3. Graph-Based Modeling and Knowledge Representation

The challenges outlined in Section 2 reveal a fundamental limitation of the current engineering workflow in FS race car development: design knowledge, dependencies, and variant relationships are distributed across heterogeneous tools, different sub-teams, and rarely documented manual processes. This fragmentation prevents a consistent system-level understanding of the vehicle and leads to data loss, integration errors, and inefficiencies in optimization processes [6]. To overcome these limitations, a formalized, machine-readable and machine-executable representation of engineering knowledge is required that captures the structure of processes, their parameters, constraints and interdisciplinary interactions in a unified model. Graph-based design languages (GBDLs) provide such a foundation. By encoding design knowledge in the form of design rules and vocabulary, GBDLs enable a digitally consistent, interoperable, and automatable representation of engineering systems [27]. Most notably, GBDLs offer the *digital consistency of data* and models built thereof, guarantee the *digital continuity of processes* and enable the *digital interoperability of tools* along the product life-cycle [28].

The following sections introduce the fundamental concepts, architectural principles, and knowledge representation mechanisms that underline graph-based design languages as a key enabler for digital engineering and variant management in FS race car development.

3.1. Fundamental Principles of Graph-Based Design Languages

Graph-Based Design Languages (GBDLs) represent a crucial methodology for the comprehensive description, formalization, and automation of engineering design tasks [29]. As a novel method, GBDLs enable the storage of engineering knowledge in a machine-readable and machine-executable format [28]. This approach serves as the foundational concept for design automation, aiming to manage the steadily increasing complexity observed in modern systems engineering and design [8].

GBDLs adhere to the fundamental composition scheme of natural languages, consisting of a vocabulary and rules that collectively form a grammar, as shown in Figure 1.

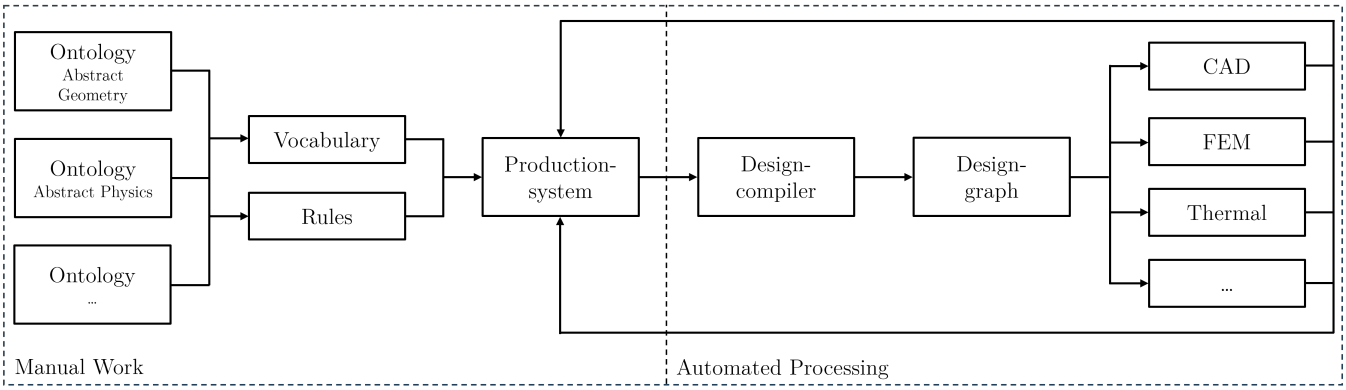

Figure 1. Fundamental information architecture of graph-based design languages [28,30].

In the context of engineering design, the core mechanism of GBDLs involves encoding any engineering concept of interest as an abstract graph node and expressing existing dependencies or couplings, whether disciplinary or multidisciplinary, by dedicated links between these nodes [8]. The result of processing a GBDL is the design graph, which is the holistic and consistent representation of the system in the form of a graph [28]. This approach offers a unified representation capable of modeling both parametrical and topological design information [31]. The ability to formalize design in a way that allows for digital programming leads to the concept of a machine-executable V-Model in Model-Based Systems Engineering (MBSE), enabling the automation of the design process [32].

3.2. Architecture and Knowledge Encoding

Figure 1 shows the information architecture of graph-based design languages, in which the vocabulary is instantiated from the available ontologies by rules. The rules combined with classical control structures from other programming languages such as `if`, `while` and `for` represents the source code referred to as the production system and defines the sequence of execution [28]. The production system encodes the complete design knowledge required to generate a complete system model. A design language compiler [30] compiles the production system into the design graph, which is later mapped to the domain-specific models (e.g., CAD, FEM, Thermal, etc.) by dedicated compiler plug-ins [28].

3.2.1. The Vocabulary and Ontology

As shown on the far left side of Figure 1, the vocabulary defines the total set of entities, or building blocks, allowed within the design. These entities form an ontology of the product. In GBDLs, the vocabulary is formally encoded in the class diagram. Each entity is represented as a class, encompassing specific attributes (i.e., parameters), symbolic equations, and relationships (i.e., associations) to other classes [29].

The design information can originate from various, heterogeneous spaces (e.g., verbal, logical, real-valued, and architecture spaces), where the GBDL acts as the common denominator for representation. For instance, concepts in verbal spaces, such as design requirements, can be encoded in an object-oriented model of that domain, i.e., an ontology, by defining characteristic dependencies between concepts in the form of links [8].

3.2.2. Rules and Model Transformations

The actual design synthesis is realized through rules, which encode design knowledge and define model transformations, as illustrated on the left side of Figure 1. These rules operate on the elements defined in the vocabulary by instructing their instantiation and subsequent manipulation.

A graphical rule structure generally consists of a left-hand side (LHS), which defines the pattern or conditions to be matched in the existing design graph, and a right-hand side (RHS), which specifies the modifications, such as adding or removing elements from the design graph, to be performed if the LHS pattern is matched [33]. In GBDLs, the sequences used to convert one model state into another are called transformations, which are ideally suited for digital formalization and programming to achieve a machine-executable process.

3.2.3. The Production System

The production system, depicted in the center of Figure 1, encodes the design process as a sequence of executed rules, commonly modeled as an activity diagram. The inclusion of control structures (e.g., `for`, `while` and `if`, etc.) enables the representation of any design decision or process, including more complex computational evaluations and repeated optimization loops.

The objective of the execution sequence is to achieve a consistent design that fulfills the given requirements in the best possible manner. In the production system, one, several, hundreds or even thousands or more model transformations may be used to digitally encode the design process knowledge during a detailed design phase [8]. The production system is typically executed by a design compiler (here the Design Cockpit 43® (DC43®) [30] is used) to automatically generate the complete design.

3.2.4. The Design Language

The design language itself is composed of the vocabulary and their underlying ontologies, the rules and model transformations, and the production systems. All these

three components (vocabularies, rules, production systems) together represent the design language, and as such, the digital DNA [34] of the product. In this respect, the entire design language represents a fully digital and machine executable representation of the design logic. The decomposition of the design languages into meaningful modules capable of designing and simulating the different system components of the Formula Student racing car, such as suspension, drivetrain, or aerodynamics, etc., permits the effortless generation of different design variants by exchanging the respective design language modules. The results of the execution of the overall design language using different modules for suspension, chassis, and aerodynamics are shown in Section 6.

3.3. Knowledge Representation via the Design Graph

As shown in the center-right of Figure 1, the final outcome of the GBDL compilation process is the design graph, that is, the instantiated abstract model. In this model, the nodes represent abstract objects (instantiated from the class diagrams), and the edges represent the relationships and dependencies between these objects. The design graph functions as the central data model or the holistic digital abstract model of the product, encompassing all parts, interconnections, parameters and topology. It is commonly implemented as a linked data structure in form of a graph [28]. The design graph is crucial for knowledge representation because:

- Consistency and single source of truth: By providing a central model, the design graph acts as a single source of truth, addressing the issue of inconsistent interlinked design models resulting from information loss between domain-specific tools. Furthermore, the centralized nature helps solve the digital consistency problem between models derived from it [31].
- Abstract, tool-independent storage: All design knowledge is stored abstractly in the design graph. This abstracts the knowledge away from vendor-specific, proprietary data formats or tools [29] in order to avoid vendor lock-in.
- Domain integration: From the abstract knowledge stored in the design graph, individual simulation plugins of the design compiler automatically map the abstract information into specific domain models (e.g., CAD, FEM, CFD; right side of Figure 1) for dedicated engineering analysis [33].

The graph-based representation is highly versatile. It can model abstract concepts, such as domain ontologies using abstract nodes and specialised links, in verbal spaces; boolean constraints in logical spaces; functional relationships in real-valued spaces; and system structures in functional, logical and physical architecture spaces. This makes it the common denominator across varying levels of abstraction and detail [8].

4. Natural Language Interaction with Design Graphs Using LLMs

Large Language Models (LLMs) offer a new avenue to access engineering knowledge stored in design graphs, enabling intuitive interaction through natural language. While graph-based design languages provide a rigorous and machine-executable representation of system knowledge, their creation and application traditionally require a substantial expertise in modeling formalisms, graph query languages and programming [35]. As engineering teams in multidisciplinary environments increasingly include participants without in-depth methodological knowledge [36], such as manufacturing planners, team managers and newcomer students, there is a need for a simplified, low-barrier access to the underlying design information [37]. Natural language interfaces aim to bridge this gap by allowing users to retrieve design graph content or ask design-related questions using a familiar natural language such as English.

In the present study, the natural-language interaction is implemented using the commercially available tool DC43® [30], which already provides an internal link to the commercial LLM Gemini 2.0 Flash of the Gemini family of Large Language Models [38]. No additional software component for the use of this internal link is necessary. However, as described in the next paragraph in more detail, only the design graph, or portions of it, are provided to the LLM as an additional context for each query.

4.1. Model-to-Text Transformation for LLM Integration

In order to enable interaction between an LLM and a design graph, the graph must first be transformed into a structured textual representation. Figure 2 illustrates this process in broad terms, but in practice it involves several stages of information extraction and abstraction.

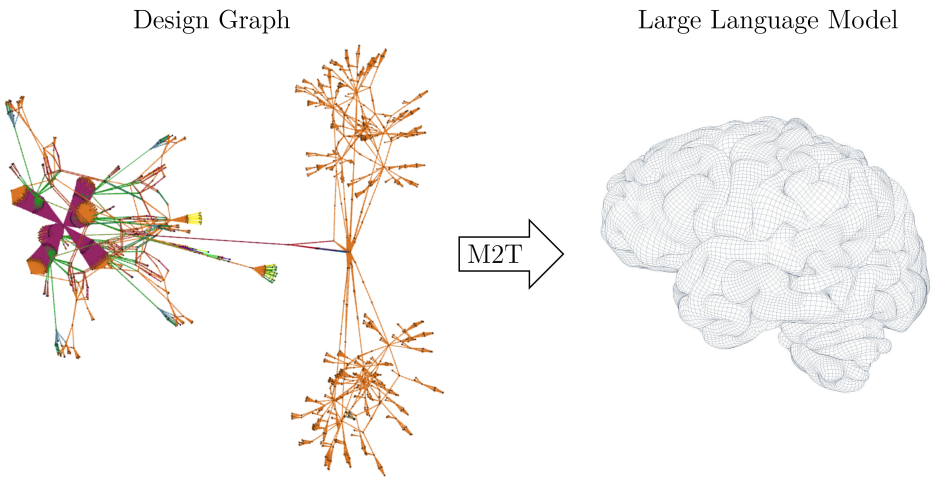

Figure 2. Model-to-Text (M2T) transformation of the design graph for RAG to the LLM.

First, the intended query scope is used to select relevant graph elements, such as nodes, their attributes, hierarchical decompositions and inter-node dependencies. These elements are then converted into natural-language or semi-formal statements that encapsulate the semantic meaning and structural context of the original graph entities [39]. Typical transformations include converting node classes into object descriptions, translating links into relational statements (e.g., 'Component A depends on Component B') and expressing parameter sets as readable attribute lists [40]. This creates a domain-specific corpus of text that conditions the LLM by embedding engineering semantics into its prompt. Rather than displaying the entire graph, the model-to-text workflow intentionally extracts an organized and comprehensible subset of information [41]. This reduces complexity, avoids token limitations and ensures that the essential system knowledge is preserved with sufficient detail for subsequent reasoning. The transformation may include object descriptions, hierarchical decompositions, parameter lists, constraint statements, or causal dependencies. By encoding the semantics of these graph elements as text, the LLM can leverage its pretrained language understanding to interpret engineering relationships, perform terminology mapping, and reason across the contextualized system information [42].

Within the described method, this selective extraction is realized through retrieval-augmented generation (RAG) techniques. In the present implementation, the natural language interaction is realized using the Gemini family of Large Language Models [38] (Gemini 2.0 Flash), selected primarily for its extended context window (up to 10^6 tokens), which allows substantial portions of the textualized design graph to be provided directly within the prompt. The design graph is transformed into a compact, domain-specific textual schema inspired by Cypher-like graph query languages, encoding nodes, attributes,

hierarchical relations, and dependencies in a machine-optimized but LLM-interpretable format. Query relevance is handled through deterministic preselection of graph fragments based on the user's query scope, eliminating the need for an external embedding model or vector database while preserving the functional role of retrieval-augmented context assembly. Instead of transmitting the complete textualized design graph to the LLM, only those preselected graph fragments that are semantically relevant to the current user query are included in the prompt [43]. These fragments may include object descriptions, parameters, constraints, or dependency statements. By anchoring the LLM's context in authoritative graph-derived information, this retrieval-augmented prompting significantly mitigates the probability of hallucinations and enhances factual consistency [44]. Moreover, the selective inclusion of graph-derived context ensures that the prompt remains compact and query-specific, enabling efficient token usage while preserving the completeness and traceability of the underlying system knowledge. As a main result, the described approach provides a robust link between the structured graph representation and the adaptive reasoning capabilities of the LLM, enabling reliable and contextually accurate natural-language access to the design graph [45].

4.2. Interpretation of Graph Knowledge by LLMs

Once the textual context is embedded in the prompt, the LLM interprets the design knowledge through several mechanisms:

- Semantic mapping: The LLM aligns natural language queries with the textualized graph content. For instance, vague user questions ('How is the battery cooled?') are mapped to specific model elements (e.g., node classes, design parameters, or links) based on linguistic similarity and context awareness [46].
- Implicit query generation: Although no explicit graph query language is used, the LLM effectively generates an internal search process. It identifies relevant graph fragments, extracts parameter values or structural relationships, and reconstructs an answer in natural language [47].
- Contextual reasoning across disciplines: Engineering design graphs often combine structural, functional, topological, geometric and parametric information. LLMs can integrate these heterogeneous elements by leveraging their pretrained reasoning abilities, enabling explanations that cross disciplinary boundaries (e.g., linking mechanical constraints with electrical architecture) [36,48].

In practice, users can ask domain-specific engineering questions (e.g., 'Which are the main coordinates that have an impact on the dynamic toe angle change?') and the LLM derives its answers directly from the textualised graph context, as shown in Figure 3.

This means that even complex relationships in suspension kinematics can be understood without having to look at the formal graph structure directly.

4.3. Motivation for Natural Language Access

Natural language interaction offers a significant advantage for engineering workflows by providing intuitive access to complex system models without requiring prior knowledge of graph-based design languages, code syntax, or specialized modeling environments [49]. This lowers the entry barrier for new team members and domain novices, enabling them to explore system structures, parameter definitions, and design rationales through familiar linguistic interaction [49] in their mother tongue. At the same time, routine information retrieval, such as identifying component dependencies, accessing parameter values or comparing design variants, becomes substantially more efficient. This supports faster decision-making during iterative development cycles. The ability to uncover implicit relationships within the system model through natural-language queries enhances trans-

parency and enables earlier detection of inconsistencies or overlooked couplings [50]. In settings involving multiple teams, such as Formula Student, where different sub-teams must work closely together, a natural-language interface offers a unified and accessible entry point to the shared design graph. This makes collaboration easier, facilitates and improves knowledge transfer and reduces the communication overheads usually associated with highly specialised digital engineering tools [36].

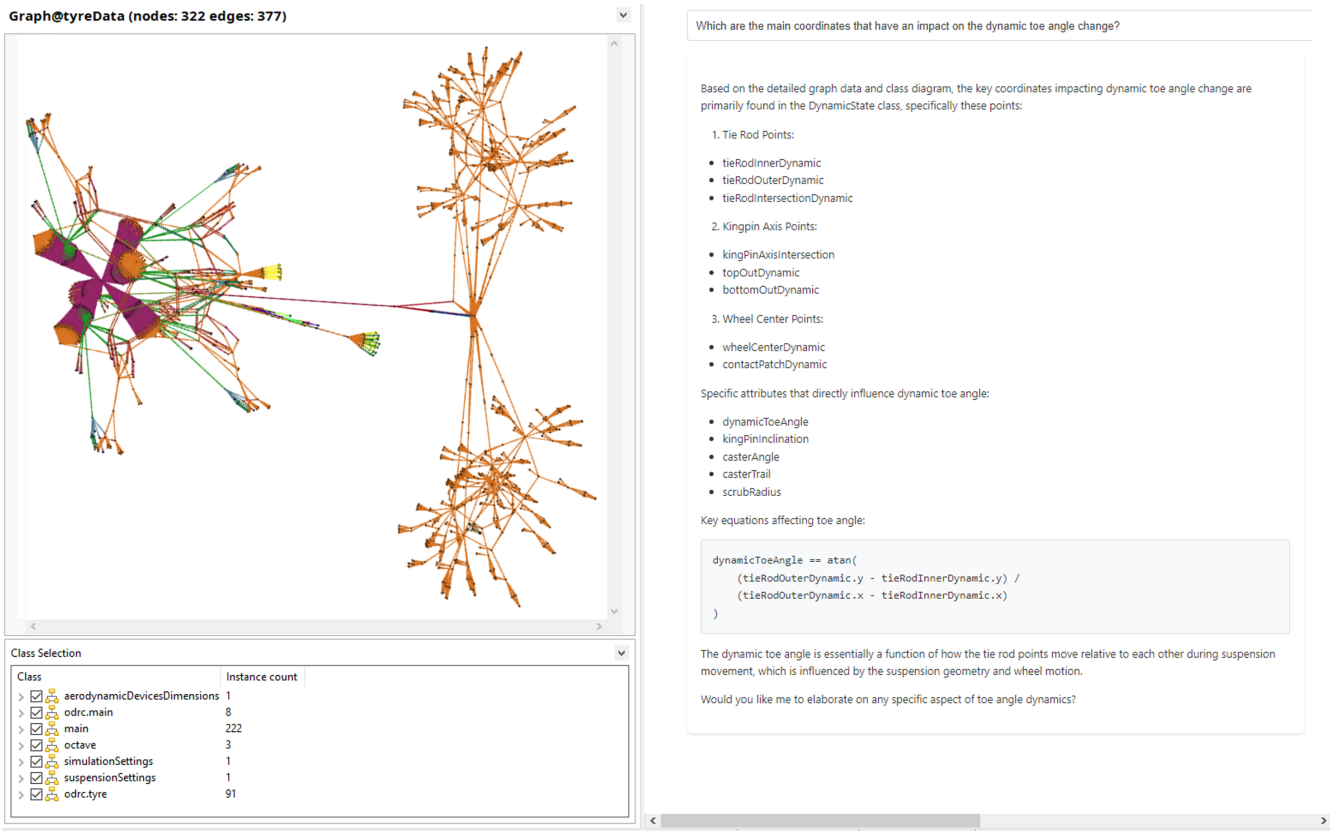

Figure 3. Natural-language user interaction enabled by linking the design graph to the LLM.

5. Version Control for Graph-Based Engineering Models

Graph-Based Design Languages (GBDLs) formalize engineering knowledge in a machine-readable and machine-executable graph format, enabling automated model generation and the systematic creation of design variants. As these artifacts increasingly serve as digital assets analogous to software source code, effective version control becomes essential [51]. Similar challenges to those found in software engineering arise from the evolution of vocabularies, rules, production systems and instantiated design graphs, such as the need to manage complexity, track changes and coordinate contributions across multiple developers or sub-teams [52] working in different locations and time zones. These challenges are particularly pronounced in environments involving multiple development teams, such as FS, where different sub-teams are working together on various systems of the race car. Furthermore, short development cycles, high staff turnover and frequent updates to design configurations require a robust mechanism for maintaining consistency and traceability of the design progress [53].

In the present study, the version control mechanism is realized using a Git [54] integration natively embedded in DC43® [30]. Beyond the classical functionalities Git offers and which are described in more detail in the following paragraph, no further features for version control were implemented or used.

5.1. Git as a Version Control Mechanism for Graph-Based Design Languages

Because GBDL model elements (e.g., ontologies, vocabulary, rules and productions systems) are available as formal, structured, and file-based representations, they can be stored and managed using Git repositories in direct analogy to software development projects [55]. All elements of a design language are stored as files on a computer, similar to source files of software programs. This includes the class diagram that defines the vocabulary, the object-diagram-based rules, and the activity-diagram production system. Graph instances, representing the generated product configurations, can likewise be stored as a file containing a structured graph. This file based approach makes the full functionality of modern version control applicable for GBDLs as well [56]. Git branches can be used as parallel development paths for different design variants, enabling teams to explore alternative configurations without interfering with the main development line. Branches may contain parameter sweeps, design iterations of a FS vehicle, or experimental redesigns of sub-systems. Git merging strategies can then be used to reintegrate divergent development stages, ensuring that validated improvements are consolidated into the main branch [54]. Furthermore, Git commits preserve a complete history of how vocabularies, rules, or production systems evolve over time, providing essential traceability for engineering teams at all times.

Git is used for version-control of the GBDLs only, and includes vocabularies, rules, and production systems, while the instantiated design graphs become obsolete after model generation. Furthermore, design graphs can be regenerated by executing the corresponding version of the design language. Storing design graphs only makes sense, if they represent the result of a huge accumulation of runtime, as it might occur after time consuming optimization runs. In engineering practice, generated design graph instances may thus be stored for documentation and traceability purposes, particularly for different design variants obtained after extended optimization cycles, but have then to be treated as immutable artifacts. In consequence, no merge operations on complex graph structures are required. Parallel development and merging are therefore performed only at the level of the design language, where the usual Git merge strategies are both effective and efficient.

5.2. Comparing Design Versions and Managing Variants

The formalization inherent to GBDLs enables systematic comparison between different design versions. Since all design knowledge is encoded using machine readable structures and executable graph transformations, differences between model states can be evaluated at the level of code structure, design parameters, or design logic. For example, two graph versions may be compared by analyzing added or removed nodes, modified relationships, or altered parameter values [57]. Furthermore, given that design variants are generated by executing the same rule set under different conditions, Git can be used to track the exact parameter sets or boundary conditions that led to each variant. This ensures that even large design spaces, such as thousands of vehicles or sub-system configurations, remain reproducible and can be systematically evaluated.

5.3. Advantages for Traceability and Collaboration

The integration of Git into graph-based engineering workflows enhances traceability by capturing each GBDL design state as an explicit and reproducible commit. This facilitates the reconstruction of the design history of vocabularies, rules, and production systems at any specified point in time [58]. In the context of multi-team environments, such as FS, with dedicated sub-teams for chassis, suspension, or aerodynamics, Git facilitates coordinated development by enabling the concurrent design of various systems and sub-systems of the vehicle, without risking inconsistent or overwritten design states. Moreover, Git's

mechanisms for change tracking, branching, and comparison provide a transparent basis for reviewing and integrating modifications, thereby strengthening collaborative design processes and ensuring continuity across development cycles.

6. Application: Formula Student Case Studies

To demonstrate the practical applicability of the methodology outlined in Sections 2–5, the following section presents selected case studies from three FS teams. These examples illustrate the advantages of integrating graph-based design languages, natural-language interaction through LLMs, and Git as a version control system into the development of high-performance race cars.

6.1. Formula Student Team Kiel

Since the 2025 season, the Formula Student Team Kiel has made use of graph-based design languages and the corresponding compiler DC43® [30] as the main tool to design and optimize the suspension and steering kinematics. The development was carried out as an iterative, target-driven process, which is version-controlled using Git, in order to ensure traceability of geometrical and topological changes, different optimization steps, and their respective effects on suspension characteristics throughout the design process.

Within the suspension design, dynamic camber change was defined as the primary optimization objective. Key kinematic parameters, such as roll center height, scrub radius, and caster angle, were implemented as boundary constraints to guide the optimization process in order to achieve a desired dynamic camber characteristic. Each design iteration was automatically evaluated within the software for potential component interference, enabling early identification and mitigation of packaging conflicts. A similar workflow was applied to the steering system, with a focus on achieving an appropriate Ackermann characteristic according to the team's vehicle dynamics targets.

Quantitative assessment of the iterative optimization procedure was conducted using an integrated lap time simulation, complemented by analysis of the tool's generated suspension characteristics plots, for instance dynamic toe and camber change, roll center variation, total weight transfer or steering characteristics. This provided immediate feedback on the dynamic implications of design decisions, enabling a rapid convergence toward performance-oriented vehicle configurations, as illustrated in Figure 4.

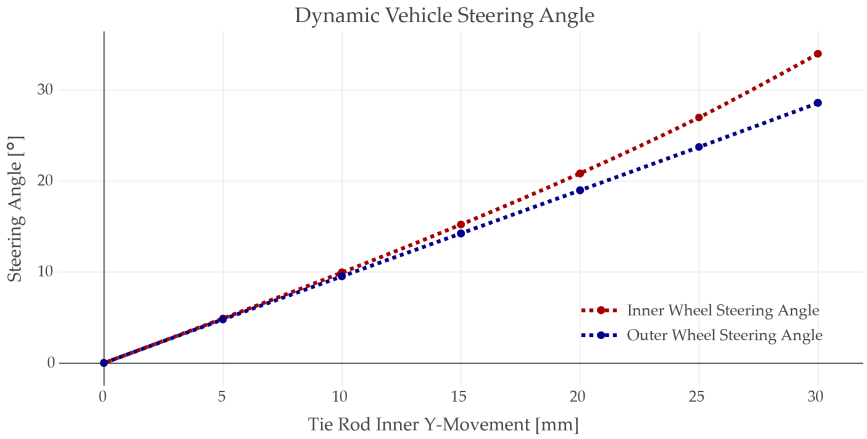

Figure 4. Vehicle steering characteristics.

Additional analyses were carried out for the 2026 season regarding the anti-roll bar (ARB) configuration, including the optimization of the ARB blade geometry depending on the target vehicle's roll stiffness [19]. The outcome of this optimization process is shown in Figure 5, which visualizes the resulting vehicle model, including the optimized

suspension and steering geometry. To support subsequent component design, the loads calculated and stored in the design graph were compared and combined with empirical load measurements acquired by the team during on-track operations at the FS competitions.

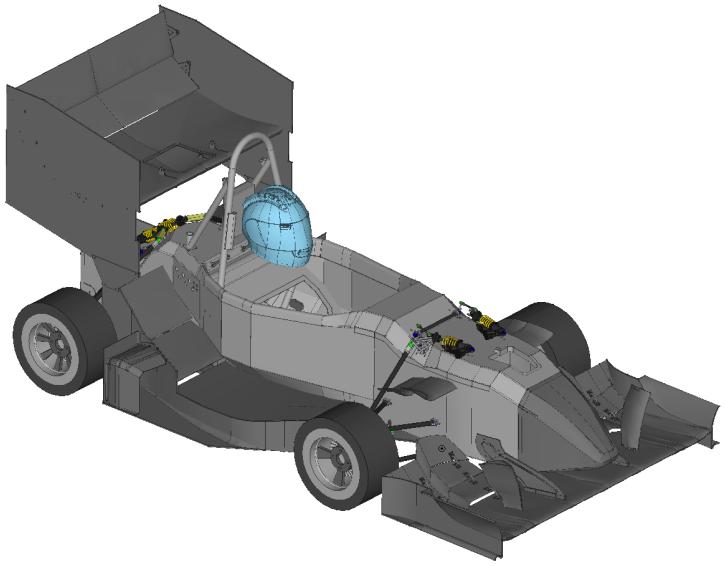

Figure 5. Optimized and collision-free *Formula Student Team Kiel* vehicle model in DC43®.

By managing design variants and their corresponding graphs through Git, the team was able to systematically compare alternative concepts and capture design knowledge for subsequent iterations. In combination with the suspension optimization algorithm [19] and the integrated collision analysis, this approach facilitated a significantly faster development of a kinematically optimal and collision-free suspension configuration within the defined design space, reducing the number of manual design iterations required to reach a viable solution from several dozen to fewer than ten target-driven optimization cycles.

6.2. Formula Student Team Pforzheim

The development of the suspension kinematics of the 2025 FS vehicle (called Rosie), which is the team's inaugural all-wheel drive (AWD) prototype, was characterized by severe time constraints and the engineering imperative to transition from a proven rear-wheel drive (RWD) concept (referencing the predecessor, Jade) to an AWD architecture. The situation was further exacerbated by the sudden turnover of the lead suspension designer and the absence of an adequate documentation of the development process.

At the onset of the season, a dual software approach was adopted in an intensive two-week development phase to identify the most efficient path to a viable design solution. On the one hand, the team used a state of the art MBS tool, which is widely used within the FS community. That enabled rapid design of the initial model but revealed technical bottlenecks during implementation, particularly the implementation of a decoupled spring-damper system proved unfeasible within the allocated timeframe. Furthermore, the workflow was deemed inefficient regarding the rapid definition of new performance targets and the onboarding of new engineering team members. On the other hand, the team employed the aforementioned technology of graph-based design languages and the corresponding design compiler. Consequently, the workflow was significantly streamlined by the software's capability to import CAD coordinates directly, generating immediate model iterations with integrated result analysis, enabling the team to hit new performance targets within a span of a week. Therefore, the technology of graph-based design languages has been selected for the purpose of analyzing and optimizing the vehicles' suspension kinematics.

The primary engineering challenge was to define and adapt suspension parameters (e.g., roll center height, bump steer, ride frequency) within the geometric constraints of the existing RWD suspension coordinates on the monocoque. These coordinates were retained due to budgetary limitations that precluded the production of a new monocoque mold. The decisive advantage of the approach was the open representation of all model equations combined with integrated sensitivity analyses, which enabled a substantially improved understanding of the suspension behavior, rather than relying on labor-intensive manual derivations. Further efficiency gains were achieved through integrated LLM-based assistance within the software environment. These capabilities facilitated faster onboarding, as new team members could query specific kinematic problems and receive step-wise explanations and corresponding analytical outputs. Moreover, targeted parameter sensitivity analyses supported design iterations by clarifying relevant governing equations and identifying the parameters with the highest influence on selected performance targets, thereby enabling focused design adjustments. This AI-augmented workflow enabled the team to develop and implement a valid, functional kinematic design within two weeks, a process that typically spans several months for comparable FS teams. From an engineering-effort perspective, the time reduction is mainly driven by a substantial decrease in manual iteration work. In previous seasons, developing a new suspension kinematic concept typically required more than 150 person-hours due to extensive manual trial-and-error iterations. Using the GBDL-based workflow with integrated optimization, comparable designs were achieved within approximately 20–30 person-hours, as initial hardpoint placement could be guided by packaging constraints and subsequently refined automatically. Overall, a suspension design loop that previously spanned three to four weeks was reduced to a process of a few days.

To mitigate the risk of knowledge loss associated with personnel turnover, a structured framework for data management and traceability was established, as these aspects are essential for enabling continuous performance improvements across seasons. In this context, the archiving and versioning of vehicle models transitioned from a static procedure to a dynamic, server-based system. During the initial phase, models were serialized as compressed archives, which allowed the preservation of design rationale and facilitated comparative analyses between vehicle generations but proved inefficient for iterative development due to the need for manual model retrieval and evaluation. Subsequently, the workflow was enhanced through the integration of a centralized Git-server with the native version control capabilities of DC43®, ensuring clear differentiation and traceability of model states throughout the development cycle and providing an infrastructure which supports also the inter-seasonal knowledge transfer. This systematic versioning approach was further expanded and refined in the subsequent 2026 season.

In contrast to the development of Rosie, which was geometrically constrained by the reuse of legacy monocoque hardpoints, the 2026 development cycle provided complete design freedom. The process, shown in Figure 6, comprised the parallel development of four suspension design variants exhibiting substantial kinematic differences. Within the 2026 versioning tree, specific design targets were defined and iterative optimization loops were executed for each vehicle variant. This systematic evaluation ultimately led to the selection of a redirected spring-damper configuration combined with a Z-shaped anti-roll bar, illustrated in Figure 7.

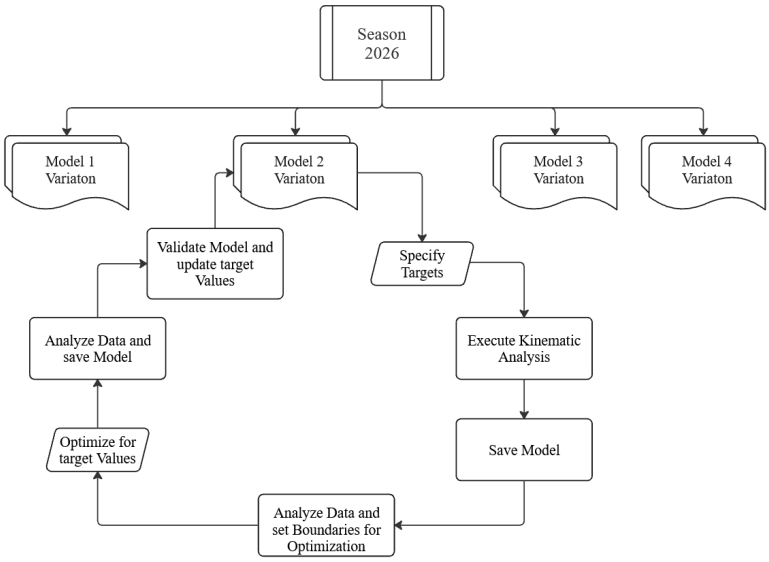

Figure 6. Suspension design variants optimization cycle.

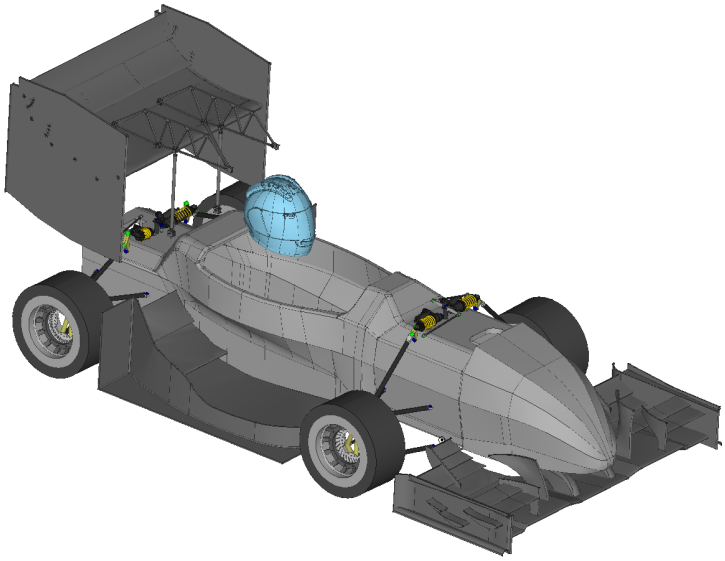

Figure 7. Vehicle model *Formula Student Team Pforzheim* in DC43® with Z-shaped anti-roll bar.

The suspension kinematics, developed using the aforementioned methodology, were empirically validated during the 2025 FS competition season with the all-wheel-drive vehicle Rosie. The vehicle demonstrated stable and predictable handling characteristics across all dynamic disciplines. The suspension design was a significant contributing factor to the team's most successful season to date, culminating in first place in Engineering Design, second place in Acceleration, second place Overall, and the receipt of the *All Star Excellence Award* at the FS competition in France.

6.3. Formula Student Team Darmstadt

In the 2026 season, the Formula Student FaSTDa-Racing team of the Darmstadt University of Applied Sciences employed graph-based design languages and the corresponding compiler DC43® as the central element of the suspension design. These tools were utilized for kinematic analysis and optimization. As an input of the design process, the coordinates of the vehicles' suspension kinematics are entered for the front and rear axle in two separate design rules. These input files serve as a central interface between the model, the optimization process and the design evaluation. The team originally created and adjusted these setup files manually, always depending on the results of the kinematic analysis.

However, this process proved to be time-consuming and error-prone, as several errors occurred when changing the coordinates and naming the variants by the user manually. Therefore, the decision was made to automate the entire process as much as possible by using a combination of Python [59] scripts, Windows batch files, and the Git version control system. The aim is to create a reproducible, traceable and above all, efficient workflow in which:

- Optimization results are automatically transferred to new design variants.
- Variants to be used can be selected from a drop-down menu.
- Input data and results (design graphs and plots) are archived consistently with the design variants in their description.
- Git is used for seamless version management.

The resulting work environment is structured into three functional areas. First, the graph-based design language comprises two main modules: the suspension kinematics analysis and the kinematics optimization [19]. Second, a set of automation scripts facilitates the handling of model variants. In particular, one script is used to select the kinematics configuration for the front axle, a second scripts creates new setup variants based on the optimization output, and a third script is used to archive the results obtained from the kinematics analysis. Third, version management and evaluation are supported through a Git repository, archiving the input data, design graph, and relevant plots for each vehicle design variant, which visually depict the key characteristics of the suspension kinematics. As illustrated in Figure 8, a typical workflow for a simulation run proceeds as follows.

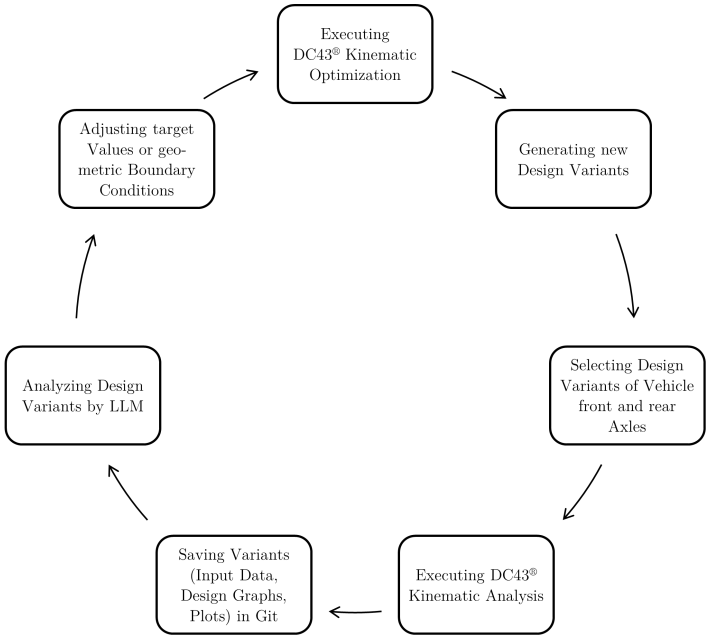

Figure 8. Suspension kinematic design process.

At first, the DC43® kinematic optimization design language generates result files within the *kinematicsOptimization* directory. Then several Python scripts process the results and produce a new setup variant in a dedicated *designVariants* directory. Additional scripts enable selecting an existing design variant, which is subsequently executed within the suspension analysis. Following the suspension analysis, all generated plots are copied into a run-specific archive folder. Finally, the complete state of the project is recorded in Git, including input data, results, and plots within a dedicated commit, thus ensuring traceability and reproducibility. A representative optimized vehicle model obtained through this workflow is shown in Figure 9.

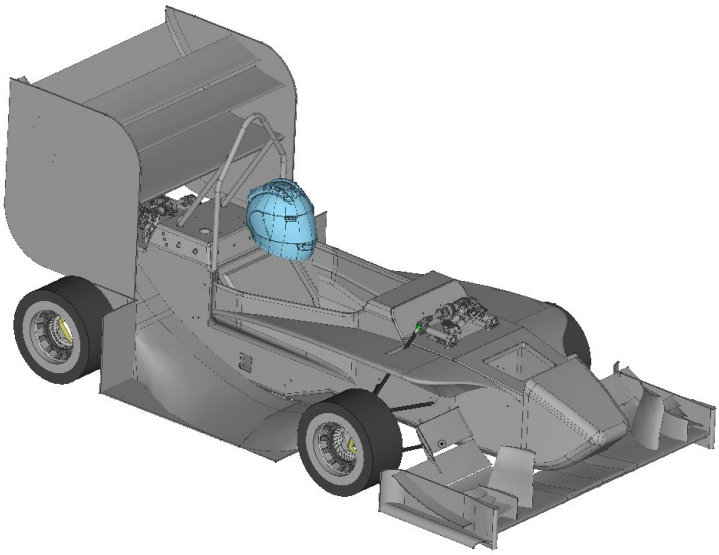

Figure 9. Vehicle model *Formula Student Team Darmstadt* in DC43® after optimization.

Due to the automated workflow, 16 optimization runs were executed within three days, implicitly exploring approximately 15,360 suspension variants. Each optimization run required less than five minutes of manual interaction (less than 80 min in total), while the remaining computation time was fully unattended. By contrast, the previously used manual process, making use of a state of the art MBS tool, typically allowed only about 25 variants to be analyzed, each requiring repeated user interaction and approximately three minutes of hands-on time per variant, making large-scale manual variant exploration almost impractical.

The simulation results stored within the design graph are automatically analyzed by making use of an integrated Large Language Model to compare several design parameters of the different vehicle variants. The proposed methodology enables the direct identification of developments in values and trends across simulation cycles, thereby eliminating the necessity of manually evaluating each of the 107 characterizing data points (front and rear axles combined) of the suspensions kinematics. Within the predefined design space, the generated vehicle variants were compared based on selected performance targets, including anti-dive characteristics, heave- and roll-frequencies, and roll center positions. This evaluation provided a consistent basis for assessing differences between variants and enabled the identification of the most suitable vehicle design within the given constraints.

7. Discussion

The presented case studies of the three FS teams from Kiel, Pforzheim and Darmstadt demonstrate the practical benefits of integrating GBDLs, natural-language interfaces via LLMs, and Git-based version control into the race car development process. Across all teams, the proposed methodology enabled design automation and traceable management of complex engineering knowledge in suspension kinematics and vehicle system design. A key advantage of this approach is the centralization of engineering knowledge in a machine-readable design graph, which acts as a single source of truth. By providing a unified representation of components, parameters, and interdisciplinary relationships, teams were able to eliminate inconsistencies arising from fragmented toolchains and manual data transfer. Most notably:

- The Formula Student Team Kiel demonstrated that integrating graph-based design languages with a Git-supported workflow enables rapid, traceable design and optimization of suspension and steering kinematics. This allows for the efficient evaluation

of alternative vehicle variants and fast convergence toward performance-oriented, collision-free configurations.

- The Formula Student Team Pforzheim showed in particular that GBDL-enabled design iterations could be finished in a fraction of the normal development time. This was possible even under constraints such as legacy monocoque hardpoints or a sudden turnover of team members.
- The Formula Student Team Darmstadt similarly leveraged automation scripts in combination with Git to ensure reproducible and traceable handling of multiple design variants, enabling systematic comparison of suspension kinematic variants.

All in all, the integration of the LLM further enhanced accessibility and collaboration inside the multidisciplinary teams. By transforming design graphs into structured, semantically rich textual representations, team members without prior experience in formal modeling or programming could query the system, trace dependencies, and explore cross-domain interactions. The existence of the LLM allowed team members to query the current design via the LLM with questions, formulated in natural language, beyond the area of their own expertise, in case of unavailability of the appropriate specialist. This helped to reduce situations where people get completely stuck when working on their own. However, no specific claim can be made on the quality of the content of the answers generated by the LLM, due to their current black box nature. Overall, the LLM lowered the psychological entry barrier for users unfamiliar with graph-based design languages by reducing the effort required to search documentation and interpret model structures, thereby increasing confidence in making modifications to the code, exploring design alternatives, and improving acceptance of the automated workflow. As such, the onboarding of new members was eased.

Furthermore, by supplying the LLM with structured data derived from model-to-text transformations of the design graph, the system grounds its content generation in verified engineering information, which significantly improves response accuracy and reduces hallucination. Finally, the methodology calls for a systematic approach to version control, including defined procedures for branching and merging, to ensure consistency across concurrent design variants.

While the above points reflect some of the key findings and user stories experienced by the three FS teams when using the novel graph-based engineering approach, a key observation of the two organizers in form of the first and last author of this work is the fact that convincing the first two or three FS teams was much more difficult than to convince team number four and five, who had already had a word of mouth. We attribute this to the fact that quite many prefer a known but manually tedious way of working to an unknown but much more effective and efficient automation opportunity simply to the fact that the new method and tools do not match their educational background. In this respect, classical engineering curriculae urgently need to include modern object-oriented modeling, object-oriented programming, visual modeling languages and techniques as well as fundamental artificial intelligence methods as a must in order to educate and prepare students for the foreseeable future of digitalization.

8. Conclusions

This paper has presented a comprehensive framework for managing design variants in Formula Student race cars through digital engineering methods. Through the use of graph-based design languages, a natural-language interaction with LLMs and a Git-based version control is enabled. The proposed approach addresses core challenges in FS development, including tool heterogeneity, manual data exchange, fragmented sub-team workflows, and knowledge loss due to student turnover and insufficient documentation.

The applications of the three FS teams from *Kiel*, *Pforzheim* and *Darmstadt* demonstrated that the framework enables faster, more reliable suspension design iterations, improves cross-domain collaboration, and provides a robust infrastructure for knowledge preservation. The combination of GBDLs and LLM-based natural-language interfaces proved particularly effective in lowering the barrier to accessing complex design knowledge, supporting decision-making, and accelerating the design process.

Looking to the future, the presented methodology offers a scalable route toward increased engineering automation, systematic variant management, and early-stage design optimization enabled by graph-based design languages, model-to-text transformations, and associated design compiler technologies. Due to the fact that an LLM is seen as an additional add-on and not a black box with no alternatives, the engineering experts who would like to inspect the consequences of their decisions are able to trace the design evolution forward or backward, profiting off the *engineering as code* design philosophy and modeling approach to the maximum.

Author Contributions: Conceptualization, J.B. and S.R.; methodology, J.B. and S.R.; design and dimensioning, J.B., H.E., J.K. and R.R.; simulation, J.B., H.E., J.K. and R.R.; analysis and evaluation, J.B. and S.R.; writing—original draft preparation, J.B., H.E., J.K., R.R. and S.R.; writing—review and editing, J.B. and S.R. All authors have read and agreed to the published version of the manuscript.

Funding: The authors would like to thank the Ministry of Science, Research and Arts of the Federal State of Baden-Württemberg for the financial support of the projects BUP47–STRIKE-FS and BUP56–BUP-Speed Transfer within the InnovationCampus Future Mobility.

Data Availability Statement: The datasets presented in this article are not readily available, as they are the property of the cooperating Formula Student teams and subject to confidentiality agreements. Requests to access the datasets should be directed to the corresponding author.

Acknowledgments: The authors wish to thank the cooperating Formula Student Teams of the University of Applied Sciences Kiel, the University of Applied Sciences Pforzheim and the University of Applied Sciences Darmstadt for making use of graph-based design languages and the corresponding compiler DC43® in their development process as well as sharing their design data and development experience with us.

Conflicts of Interest: The corresponding author is an external doctoral candidate and employed by IILS Ingenieurgesellschaft für Intelligente Lösungen und Systeme mbH and the research described in this paper makes use of the company's proprietary software, DC43®, which was provided by the employer. The study received no additional financial support beyond the doctoral candidate's regular salary. Neither the candidate nor the supervisor receive royalties, fees, or any other personal remuneration related to the software's commercial exploitation for this paper. All data collection, analysis, and reporting were conducted independently of commercial interest, and the findings presented herein reflect the authors' unbiased scientific conclusions. Furthermore, the software license used for this paper was provided free of charge, and the corresponding Formula Student teams were likewise granted complimentary access to the software license as well as free design coaching.

References

1. Sabadka, D.; Molnár, V.; Fedorko, G. Shortening of Life Cycle and Complexity Impact on the Automotive Industry. *TEM J.* **2019**, *8*, 1295–1301. [CrossRef]
2. Rizvi, A.; Hepp, D.; Jana, P.; Buechner, F.; Geßler, F.; Feike, J. Traditional OEMs Are on a Mission to Close the Gap in Automotive Software Development with Innovative Market Entrants. Targeted Improvements Could Significantly Accelerate Their Pace. *McKinsey* **2025**. Available online: https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/winning-the-automotive-software-development-race? (accessed on 29 November 2025).
3. Tyson, S. Overcoming EV and AV Complexity with Model-Based Systems Engineering. *Siemens* **2024**. Available online: https://blogs.sw.siemens.com/en-US/automotive-transportation/2024/08/27/overcoming-ev-and-av-complexity-with-model-based-systems-engineering/? (accessed on 29 November 2025).

4. García-Manrique, J.A.; Peña-Miñano, S.; Rivas, M. Manufacturing to motorsport by students. *Procedia Eng.* **2015**, *132*, 259–266. [CrossRef]
5. Schommer, A.; Collier, G.; Norris, R.; Morrey, D.; Nesi Maria, L.; Johnston, C. System architecture of a four-wheel drive Formula Student vehicle. *arXiv* **2022**, arXiv:2211.10476. [CrossRef]
6. Borowski, J. Digitaler Entwurf eines Formula Student Fahrzeugs am Beispiel von Fahrwerk und Antriebsstrang zur 3D-Geometrie- und Kinematikerstellung. Ph.D. Thesis, Department of Aerospace Engineering, University of Stuttgart, Stuttgart, Germany, 2026.
7. Guérineau, J.; Bricogne, M.; Rivest, L.; Durupt, A. Organizing the fragmented landscape of multidisciplinary product development: A mapping of approaches, processes, methods and tools from the scientific literature. *Res. Eng. Des.* **2022**, *33*, 307–349. [CrossRef]
8. Rudolph, S. On Some Artificial Intelligence Methods in the V-Model of Model-Based Systems Engineering. In Proceedings of the 12th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2024), Rome, Italy, 21–23 February 2024; pp. 386–393.
9. Schmidt, L.C.; Cagan, J. GGREADA: A graph grammar-based machine design algorithm. *Res. Eng. Des.* **1997**, *9*, 195–213. [CrossRef]
10. Dunbar, D.; Hagedorn, T.; Blackburn, M.; Dzielski, J.; Hespelt, S.; Kruse, B.; Verma, D.; Yu, Z. Driving Digital Engineering Integration and Interoperability Through Semantic Integration of Models with Ontologies. *arXiv* **2022**, arXiv:2206.10454. [CrossRef]
11. Bischof, G.; Bratschitsch, E.; Casey, A.; Lechner, T.; Lengauer, M.; Millward-Sadler, A.; Rubeša, D.; Steinmann, C. The Impact of the Formula Student Competition on Undergraduate Research Projects. In Proceedings of the 39th IEEE Frontiers in Education Conference, San Antonio, TX, USA, 18–21 October 2009; pp. 1–6.
12. Gomez, A.P.; Krus, P.; Panarotto, M.; Isaksson, O. Large language models in complex system design. *Proc. Des. Soc.* **2024**, *4*, 2197–2206. [CrossRef]
13. Shahriari, R.; Ragan, E.D.; Ruiz, J. Natural Language Interaction for Editing Visual Knowledge Graphs. In Proceedings of the Knowledge Capture Conference 2025 (K-CAP '25), Dayton, OH, USA, 10–12 December 2025.
14. Liang, S.; Stockinger, K.; de Farias, T.M.; Anisimova, M.; Gil, M. Querying knowledge graphs in natural language. *J. Big Data* **2021**, *8*, 3. [CrossRef]
15. Rouelle, C. Formula Student Car Design Process. *OptimumG Vehicle Dynamics Solutions*. 2018. Available online: https://www.formulabharat.com/wp-content/uploads/2018/02/Formula-Student-Car-Design-Process.pdf (accessed on 23 November 2025).
16. Dassault Systemes. Available online: https://www.3ds.com/de/products/catia/catia-v5 (accessed on 15 January 2026).
17. Siemens Digital Industries Software. Available online: https://plm.sw.siemens.com/de-DE/nx/ (accessed on 15 January 2026).
18. Parametric Technology Corporation (PTC). Available online: https://www.ptc.com/de/products/creo/parametric (accessed on 15 January 2026).
19. Borowski, J.; Rudolph, S. Automation and Optimization of the Design and Development Process for a Formula Student Racing Car Suspension. In *2025 Stuttgart International Symposium*; SAE Technical Paper 2025-01-0270; SAE International: Warrendale, PA, USA, 2025. [CrossRef]
20. Pirklbauer, G.; Ramler, R.; Zeilinger, R. An Integration-Oriented Model For Application Lifecycle Management. In Proceedings of the 11th International Conference on Enterprise Information Systems—Information Systems Analysis and Sepcification, Milan, Italy, 6–10 May 2009; pp. 399–402.
21. Mihailidis, A.; Samaras, Z.; Nerantzis, I.; Fontaras, G.; Karaoglanidis, G. The design of a Formula Student race car: A case study. *Proc. Inst. Mech. Eng. Part D J. Automob. Eng.* **2009**, *223*, 805–818. [CrossRef]
22. Nordvall, E.; Tarkian, M. Knowledge based engineering for formula student. In Proceedings of the DS 118: Proceedings of NordDesign 2022, Copenhagen, Denmark, 16–18 August 2022.
23. Formula Student Germany—Rules. Available online: https://www.formulastudent.de/fsg/rules (accessed on 15 January 2026).
24. Institution of Mechanical Engineers. More Electric Cars Than Ever at Formula Student 2025—And Number Set to Rise. Available online: https://www.imeche.org/news/news-article/more-electric-cars-than-ever-at-formula-student-2025-and-number-set-to-rise? (accessed on 15 January 2026).
25. Piechna, J. A Review of Active Aerodynamic Systems for Road Vehicles. *Energies* **2021**, *14*, 7887. [CrossRef]
26. Hu, Q.; Amini, M.R.; Wang, H.; Kolmanovsky, I.; Sun, J. Integrated Power and Thermal Management of Connected HEVs via Multi-Horizon MPC. In Proceedings of the 2020 American Control Conference (ACC), Denver, CO, USA, 1–3 July 2020; pp. 3053–3058.
27. Rudolph, S. On design process modelling aspects in complex systems. In *13th NASA-ESA Workshop on Product Data Exchange (PDE 2011), Cypress, CA, USA*; National Aeronautics and Space Administration: Washington, DC, USA, 2011; pp. 1–28.
28. Rudolph, S. Digital Continuity, Consistency and Interoperability Along the Product Life-Cycle Using Graph-Based Design Languages. *Global Product Data Interoperability Summit (GPDIS 2023)*. 2023. Available online: https://gpdisonline.com/wp-content/uploads/2023/10/IILSmbH-StephanRudolph-DigitalContinuityConsistencyAndInteroperabilityAlongThePLC-MBSE-Open.pdf (accessed on 16 November 2025).

29. Vogel, S.; Arnold, P. Towards a more complete object-orientation in graph-based design languages. *SN Appl. Sci.* **2020**, *2*, 1235. [CrossRef]
30. Ingenieurgesellschaft für Intelligente Lösungen und Systeme mbH. Design Cockpit 43® (DC43®). Available online: https://www.iils.de/en/product/design_cockpit_43/ (accessed on 21 November 2025).
31. Arnold, P.; Rudolph, S. Bridging the gap between product design and product manufacturing by means of graph-based design languages. In Proceedings of the TMCE 2012, Karlsruhe, Germany, 7–11 May 2012.
32. Borowski, J.; Rudolph, S. A Digital Machine-Executable V-Model for a Formula Student Racing Car. In *2024 Stuttgart International Symposium on Automotive and Engine Technology, ISSYM 2024. Proceedings*, 1st ed.; Kulzer, A.C., Reuss, H.C., Wagner, A., Eds.; Springer: Wiesbaden, Germany, 2024.
33. Voss, C.; Petzold, F.; Rudolph, S. Graph transformation in engineering design: An overview of the last decade. *Artif. Intell. Eng. Des. Anal. Manuf.* **2023**, *37*, e5. [CrossRef]
34. Borowski, J.; Rudolph, S. Graph-Based Design Languages for Engineering Automation: A Formula Student Race Car Case Study. *Vehicles* **2026**, *8*, 24. [CrossRef]
35. Hornsteiner, M.; Kreussel, M.; Steindl, C.; Ebner, F.; Empl, P.; Schönig, S. Real-Time Text-to-Cypher Query Generation with Large Language Models for Graph Databases. *Future Internet* **2024**, *16*, 438. [CrossRef]
36. Khan, M.T.; Chen, L.; Feng, W.; Ki Moon, S. Large Language Model Powered Decision Support for a Metal Additive Manufacturing Knowledge Graph. In Proceedings of the 11th International Conference of Asian Society for Precision Engineering and Nanotechnology (ASPEN 2025), New Taipei City, Taiwan, 25–28 November 2025.
37. Witschel, H.; Riesen, K.; Grether, L. Natural Language-based User Guidance for Knowledge Graph Exploration: A User Study. In Proceedings of the 13th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, Virtual, 25–27 October 2021; Volume 1: KDIR, pp. 95–102.
38. Google Gemini 2.0 Flash. Available online: https://deepmind.google/models/gemini/ (accessed on 9 January 2026).
39. Zhao, J.; Zhuo, L.; Shen, Y.; Qu, M.; Liu, K.; Bronstein, M.; Zhu, Z.; Tang, J. GraphText: Graph Reasoning in Text Space. *arXiv* **2023**, arXiv:2310.01089. [CrossRef]
40. Perevalov, A.; Both, A. Towards LLM-driven Natural Language Generation based on SPARQL Queries and RDF Knowledge Graphs. In Proceedings of the Joint Proceedings of the 3rd International Workshop on Knowledge Graph Generation from Text (TEXT2KG) and Data Quality Meets Machine Learning and Knowledge Graphs (DQMLKG) Co-Located with the Extended Semantic Web Conference (ESWC 2024), Hersonissos, Greece, 26–30 May 2024. Available online: https://ceur-ws.org/Vol-3747/text2kg_paper14.pdf (accessed on 24 November 2025).
41. Ji, S.; Liu, L.; Xi, J.; Zhang, X.; Li, X. KLR-KGC: Knowledge-Guided LLM Reasoning for Knowledge Graph Completion. *Electronics* **2024**, *13*, 5037. [CrossRef]
42. Shi, X.; Xia, Z.; Cheng, P.; Li, Y. Enhancing text generation from knowledge graphs with cross-structure attention distillation. *Eng. Appl. Artif. Intell.* **2024**, *136*, 108971. [CrossRef]
43. Zhu, X.; Xie, Y.; Liu, Y.; Li, Y.; Hu, W. Knowledge Graph-Guided Retrieval Augmented Generation. In Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies, Albuquerque, New Mexico, 29 April–4 May 2025; Volume 1, pp. 8912–8924.
44. Lewis, P.; Perez, E.; Piktus, A.; Petroni, F.; Karpukhin, V.; Goyal, N.; Küttler, H.; Lewis, M.; Yih, W.-T.; Rocktäschel, T.; et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. Available online: https://arxiv.org/pdf/2005.11401 (accessed on 24 November 2025).
45. Linders, J.; Tomczak, J.M. Knowledge Graph-extended Retrieval Augmented Generation for Question Answering. *arXiv* **2025**, arXiv:2504.08893. [CrossRef]
46. Kandula, K. Integrating Knowledge Graphs with Large Language Models for Natural Language Querying. Available online: https://corescholar.libraries.wright.edu/cgi/viewcontent.cgi?article=4063&context=etd_all (accessed on 27 November 2025).
47. He, X.; Tian, Y.; Sun, Y.; Chawla, N.V.; Laurent, T.; LeCun, Y.; Bresson, X.; Hooi, B. G-Retriever: Retrieval-Augmented Generation for Textual Graph Understanding and Question Answering. *arXiv* **2024**, arXiv:2402.07630. [CrossRef]
48. Wang, Q.; Lyu, D.; Chen, Q. Uncovering novel scientific insights with a synergistic GNN-LLM framework. *Knowl.-Based Syst.* **2025**, *330*, 114527. [CrossRef]
49. Ozsoy, M.; Messallem, L.; Besga, J.; Minneci, G. Text2Cypher: Bridging Natural Language and Graph Databases. *arXiv* **2024**, arXiv:2412.10064. [CrossRef]
50. Zhu, T.; Cordeiro, C.; Sun, Y. ReqInOne: A Large Language Model-Based Agent for Software Requirements Specification Generation. Available online: https://arxiv.org/html/2508.09648v1 (accessed on 28 November 2025).
51. Taentzer, G.; Ermel, C.; Langer, P.; Wimmer, M. A fundamental approach to model versioning based on graph modifications: From theory to implementation. *Softw. Syst. Model.* **2012**, *13*, 239–272. [CrossRef]

52. Ruhroth, T.; Gaertner, S.; Buerger, J.; Juerjens, J.; Schneider, K. Versioning and Evolution Requirements for Model-Based System Development. Available online: https://fb-swt.gi.de/fileadmin/FB/SWT/Softwaretechnik-Trends/Verzeichnis/Band_34_Heft_2/RuhrothCVSM2014.pdf (accessed on 30 November 2025).
53. Esser, S.; Vilgertshofer, S.; Borrmann, A. Graph-based version control for asynchronous BIM level 3 collaboration. In *EG-ICE 2021 Workshop on Intelligent Computing in Engineering, Proceedings*; Universitätsverlag der Technischen Universität Berlin: Berlin, Germany, 2021; pp. 98–107.
54. GitLab Inc. What Are Git Version Control Best Practices? Available online: https://about.gitlab.com/topics/version-control/version-control-best-practices/ (accessed on 6 December 2025).
55. Wu, S.; Wang, G.; Lu, J.; Huang, J.; Qiao, J.; Yan, Y.; Kiritsis, D. Cognitive digital thread tool-chain for model versioning in model-based systems engineering. *Adv. Eng. Inform.* **2025**, *67*, 103490. [CrossRef]
56. Bajczi, L.; Szekeres, D.; Siegl, D.; Molnár, V. Enhancing MBSE Education with Version Control and Automated Feedback. Available online: https://bibbase.org/network/publication/bajczi-szekeres-siegl-molnr-enhancingmbseeducationwithversioncontrolandautomatedfeedback-2024 (accessed on 27 November 2025).
57. Stephan, M.; Cordy, J. A Survey of Model Comparison Approaches and Applications. In Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development—Volume 1: MODELSWARD, Barcelona, Spain, 19–21 February 2013; pp. 265–277.
58. Esser, S.; Vilgertshofer, S.; Borrmann, A. Graph-based version control for asynchronous BIM collaboration. *Adv. Eng. Inform.* **2022**, *53*, 101664. [CrossRef]
59. Python Software Foundation. Available online: https://www.python.org/ (accessed on 7 December 2025).