vehicles

MDPI

Article

Graph-Based Design Languages for Engineering Automation: A Formula Student Race Car Case Study

Julian Borowski [1,*] and Stephan Rudolph [2]

[1] IILS Ingenieurgesellschaft für Intelligente Lösungen und Systeme mbH, 70771 Leinfelden-Echterdingen, Germany
[2] 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

The development of modern vehicles faces an increase in complexity, as well as a need for shorter development cycles and a seamless cross-domain integration. In order to meet these challenges, a graph-based design language which formalizes and automates engineering workflows is presented and applied in a design case study to a Formula Student race car suspension system. The proposed method uses an ontology-based vocabulary definition and executable model transformations to compile design knowledge into a central and consistent design graph. This graph enables the automatic generation of consistent 3D CAD models, domain-specific simulations and suspension kinematic analyses, replacing manual and error-prone tool and data handover processes. The design language captures both the structural and dynamic behavior of the suspension, supports variant exploration and allows for integrated validation, such as 3D collision detection. The study illustrates how graph-based design languages can serve as 'digital DNA' for knowledge-based product development, offering a scalable, reusable platform for engineering automation. This approach enhances the digital consistency of data, the digital continuity of processes and the digital interoperability of tools across all relevant engineering disciplines in order to support the validation of early-stage designs and the optimization of complex systems.

Keywords: graph-based design language; design automation; digital engineering; ontology modeling; suspension kinematics; Formula Student; systems engineering; design compiler; variant generation

Check for updates

Academic Editor: Mohammed Chadli

Received: 30 December 2025
Revised: 9 January 2026
Accepted: 20 January 2026
Published: 22 January 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

Modern vehicle development is undergoing a profound transformation driven by increasing system complexity, shorter development cycles, and the growing need for cross-domain integration [1,2]. Traditional vehicle design processes, which rely heavily on isolated domain-specific tools and informal knowledge exchange, often result in fragmented models, inconsistent data flows, and time-consuming manual iterations [3]. These inefficiencies are particularly pronounced in the context of lightweight design, electrification, and aerodynamically optimized systems, where mechanical, thermal, and control aspects are deeply interlinked [4–6]. At the same time, advances in model-based design, systems engineering, and knowledge formalization offer new opportunities to rethink how engineering processes are structured and executed [7–9]. In particular, graph-based design languages provide a formal, executable framework that captures product knowledge, design logic, and process rules in a unified and reusable format [10]. Unlike conventional

toolchains, which require engineers to translate information manually across domains, graph-based design languages enable automated generation of consistent CAD, simulation, and analysis models from a single, central source of truth [11].

The present paper introduces selected topics of a Formula Student race car design automation [12], based on a novel implementation of the design process using graph-based design languages. The shown methodology focuses on the suspension system of a Formula Student race car, using this as a simplified representative case study in the context of overall vehicle design. The proposed approach uses an ontology-driven vocabulary [13] and model transformations as design rules to encode structural, geometrical, and behavioral knowledge about the suspension system and the overall vehicle. The design language allows a unified representation of both topological and parametric properties of the design and incorporates, via model-transformations, a unique mechanism how the representation can be modified topologically and parametrically [10]. The design language is then compiled using a design compiler software platform to generate a central design graph consisting of all the product's relevant design knowledge, including component relationships, physical properties and dynamic simulation results [14]. In this respect, design languages follow almost in a one-to-one mapping the concept of formal programming languages with a source code level (i.e., the design language), the object code level (i.e., the design graph), and the executable code level (i.e., the domain-specific languages) as inputs for all subsequent analysis tools. These modeling, simulation, and execution capabilities for the most relevant engineering disciplines, such as 3D CAD geometry generation, multi-body and finite-element simulation, as well as thermal analysis, and many more are a unique property of the used software platform DC43® [15] and is, as such, subject of ongoing research and development, but not in the scope of the present paper.

The main scientific contribution of the present work lies therefore in demonstrating how graph-based design languages can transform the conventional vehicle design process into a knowledge-driven, automated workflow. In principle, this requires a complete methodological redesign of the formerly mostly manually driven vehicle design process into a new, purely digital and machine-executable process, using objects instantiated from background ontologies and design rules as model-to-model transformations to create an explicit digital model representation in form of a design graph, as an equivalent to the formerly implicit data structures created when using CAD, MBS, and FEM programs interactively [12]. Furthermore, it is not only necessary to model the product and its components, but also the logic of how the race car is synthesized, evaluated, and optimized. As such, a complete re-executable design process, in form of a language representation, which can be compiled and executed by a machine, has to be created. Hereby ensures the design language approach the digital consistency of data and models created thereof, the digital interoperability of the software tools, as well as the digital continuity throughout the development process [10]. As a further difference to other MBSE tools, which follow a visual modeling paradigm using standards such as SysML® [16], design languages possess a string-based source representation, which is automatically mapped within the development environment DC43® to the corresponding visual representations.

The remainder of this paper is structured as follows: Section 2 reviews current design processes and introduces the concept of graph-based design languages. Section 3 details the implementation of the method for modeling suspension systems. Section 4 presents the results in terms of the generated design graph, kinematic analyses, and variant exploration. Section 5 discusses the broader implications for vehicle design automation, followed by conclusions and outlook in Section 6.

2. Design Processes

Contemporary engineering design processes are often fragmented and rely on a multitude of isolated tools and models. This fragmentation leads to inefficiencies, inconsistencies, and increased error potential, especially as products become more complex and development cycles shorter. At the same time, advances in computational modeling and formal design methods offer new opportunities to rethink how engineering knowledge is captured and applied [17]. This section introduces graph-based design languages as a model-driven approach that formalizes design knowledge, automates model generation, and enables consistent, cross-domain engineering workflows.

2.1. Current Design Processes

In current engineering practice, product knowledge is typically distributed across various sources, ranging from technical documentation and industry standards to the personal experience of engineers and data from previous designs [18]. This fragmented knowledge landscape poses significant challenges in the context of complex, multidisciplinary product development [19]. To design a product, this knowledge must be translated into domain-specific models using specialized tools such as CAD (Computer-Aided Design), FEM (Finite Element Method), or CFD (Computational Fluid Dynamics) software [20–23], as shown in Figure 1. Each engineering discipline relies on its own toolchain, resulting in multiple, heterogeneous representations of the product across different domains [11].

These models are inherently interdependent, which means that they are dependent on each other for their existence and functionality. A change in one domain, for example a modification in the CAD geometry, often necessitates updates in other domains, such as structural analysis or thermal simulation [24]. Consequently, data must be exchanged and converted between tools frequently throughout the design process [25]. However, these data exchanges and model updates are largely manual, making them time-consuming and error-prone [26]. Each conversion step risks data loss, misinterpretation, or version mismatch, ultimately leading to inconsistencies between the domain models [27]. As product complexity increases, these inconsistencies can accumulate and severely impact the overall development process [28].

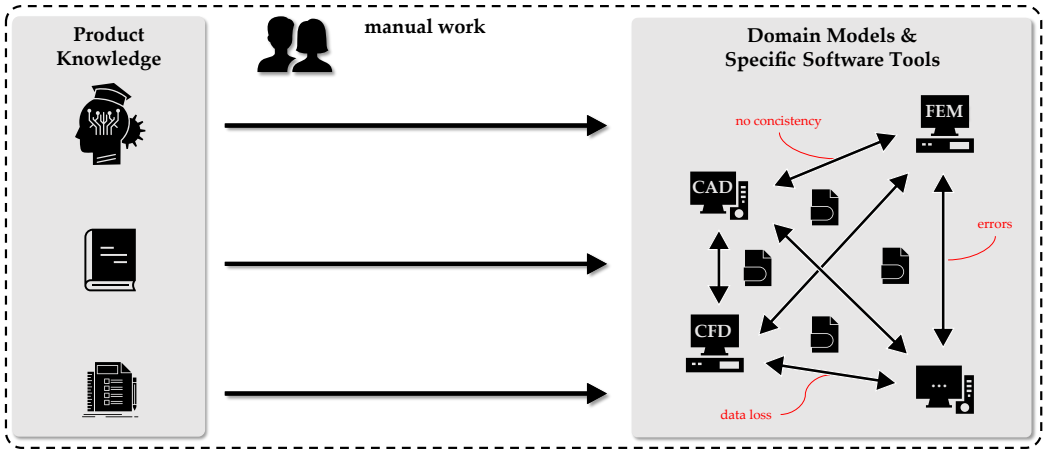


Figure 1. Current manual design process.

The fundamental problem here is the absence of a central product dataset or data model that captures all relevant design information consistently and coherently [29]. Without such a central model, engineering teams are forced to maintain multiple partial representations in parallel, an approach that is both inefficient and vulnerable to error propagation.

This observation motivates the need for a more integrated and automated approach to product modeling [30], as will be discussed in the following section.

2.2. Design Processes with Graph-Based Design Languages

The key for overcoming the challenges posed by fragmented tools and inconsistent models is not to manage complexity manually, but to avoid inconsistencies through a dedicated system information architecture [31]. This can be achieved by making use of a fundamentally different modeling approach that eliminates the need for repeated conversions between the domain-specific tools [32]. Instead of working with isolated models across different disciplines, the use of a central, holistic product model is proposed as the single source of truth. All domain-specific representations, such as 3D CAD geometry, 3D field simulations and further simulation parameters, can be consistently derived from this central model. This approach ensures the digital consistency of data and models, the digital continuity of processes, and the digital interoperability of software tools [10]. It also reduces the risk of manual errors across all domains.

A key question is how such a comprehensive model can be efficiently created and maintained. This is addressed by generating the model automatically, rather than relying on a manual design. In order to achieve this, the product knowledge must be captured in a format that is both computer-readable and executable [10,33]. This leads to the concept of a graph-based design language, a formal and executable modeling approach inspired by natural languages. In this framework, engineers describe the system architecture, components, parameters, and constraints using a defined vocabulary and explicit transformation rules. The design language is processed by a dedicated compiler, which instantiates an abstract design graph. This graph serves as the central representation of the system, capturing all relevant components, their interdependencies, and both functional and physical properties [11]. From this central design graph, the compiler automatically derives a unified, central product model, and all necessary domain-specific models (e.g., CAD, FEM, thermal simulations). This process is illustrated in Figure 2.

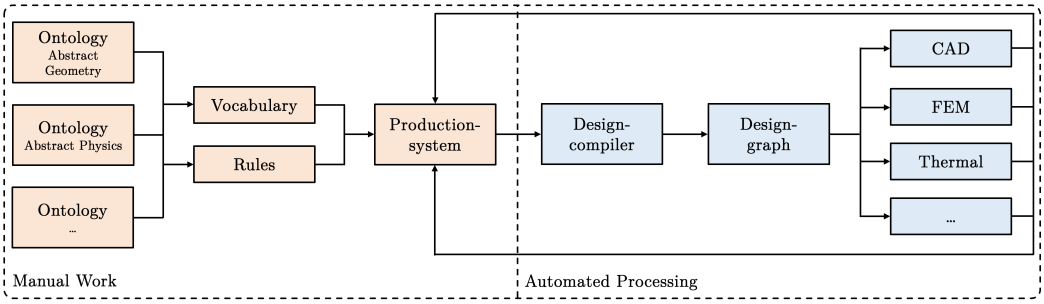


Figure 2. Information flow and model generation with graph-based design languages [10].

The use of a graph-based design language not only minimizes manual effort and error potential, but also establishes the foundation for advanced engineering automation, where models evolve together in a synchronized, rule-based manner. This compiler-driven approach shifts the engineer's focus from model synchronization to the definition of reusable design knowledge, thereby enabling a high degree of automation and adaptability.

Building on this foundation, a graph-based design language can be understood as a formalized and executable representation of engineering knowledge. It captures not just the static structure of a product, but also the logic needed to synthesize and adapt it automatically. Such a design language typically consists of three core elements, a vocabulary, several rules and a production system:

- Vocabulary (in UML an instantiated object): A set of semantically typed elements that constitute the building blocks of the design language. Defined as an ontology, the vocabulary is typically formalized in a class diagram. These elements represent domain-relevant entities such as subsystems, components, and other granular entities.

The classes in the diagram are enriched with properties that can represent, for instance, physical or cost variables.

- Rules (in UML model-to-model transformations): Transformation instructions, formulated in a simplified Java-based language, which define valid combinations, behaviors, and derivation logic between vocabulary elements.
- Production System (in UML an activity diagram): An executable orchestration of rules, visualized as an activity diagram, that controls how the design language is processed. It governs rule execution order, iteration, subprograms, and decision logic, producing the design graph from which domain-specific models are automatically derived.

These three core elements of the design language are created manually by the engineer, and collectively they form the basis for an automated product development process. Once the graph-based design language is defined, it can be executed automatically by a design compiler [11]. This compiler interprets the language and builds the design graph, which is the central, digital representation of the product.

The design graph serves as a holistic representation of the product that integrates all relevant aspects of the design in a consistent, structured way. From this central graph, all necessary domain-specific models, such as 3D CAD geometry, field simulations, or documentation can be derived automatically and consistently. The design graph thus becomes the single source of truth, ensuring coherence across disciplines and minimizing the need for manual transformation steps, as illustrated in Figure 3. However, the design process does not end with model generation. In engineering practice, especially for complex systems, design is inherently iterative. Therefore, the analysis of design graphs and domain-specific simulation results often indicates the need for modifications, refinements, or extensions. This feedback loop is directly supported by the design language. Engineers can refine the language itself, by extending the vocabulary, modifying rules, or integrating optimization algorithms.

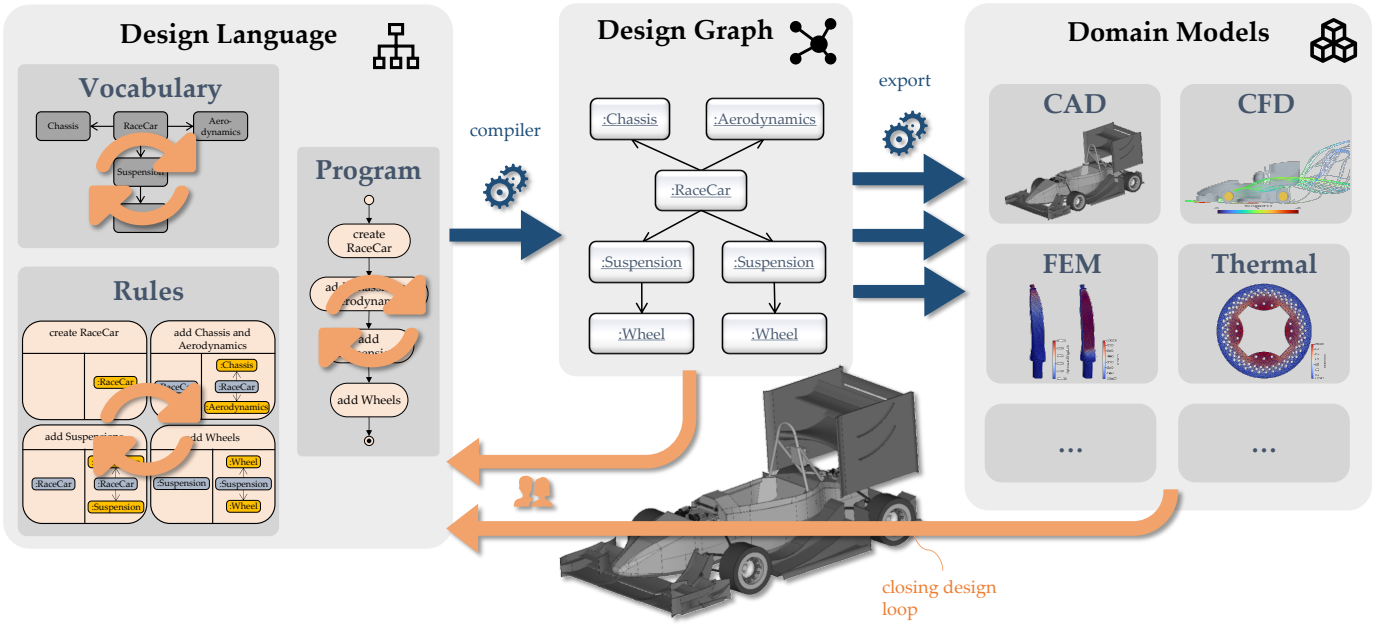


Figure 3. Design process enabled by graph-based design languages.

The production system further supports advanced control structures such as loops, conditionals, and subprograms, conceptually inspired by UML activity diagrams, enabling the specification of more complex and flexible design generation logic. In this way, graph-based design languages not only automate model generation but also support continuous

iteration and evolution, aligning well with modern engineering workflows that require both consistency and adaptability.

The terminology 'design language', 'design compiler', and 'design graph' reflects the terminology of high-level programming languages, where the (design) language represents the source code, the (design) compiler represents the translation machine, and the (design) graph stands for the intermediate object code, which is mapped by the compiler backend to the targets, in form of the domain-specific languages of the attached programs in Figure 3.

3. Formula Student Design Language Implementation

The following section describes how the suspension kinematics of the Formula Student race car is implemented using graph-based design languages. Modeling is carried out using the Design Cockpit 43® (DC43®) [15] software environment, which facilitates the structured definition, compilation, and execution of graph-based design languages.

3.1. Ontology and Class Diagram Modeling

Implementation begins with the creation of ontology-based class diagrams, which define the formal vocabulary of the design language. These diagrams represent the semantic background knowledge of the suspension system, capturing all the components, parameters and interdependencies required for kinematic analysis. The components of the actual race car suspension system are then created by instantiation and stored in the design graph for further processing.

3.1.1. Class Diagram *suspensionData* Package

The *suspensionData* package (Figure 4) defines the domain-specific ontology required to formally model the suspension subsystem of the Formula Student race car.

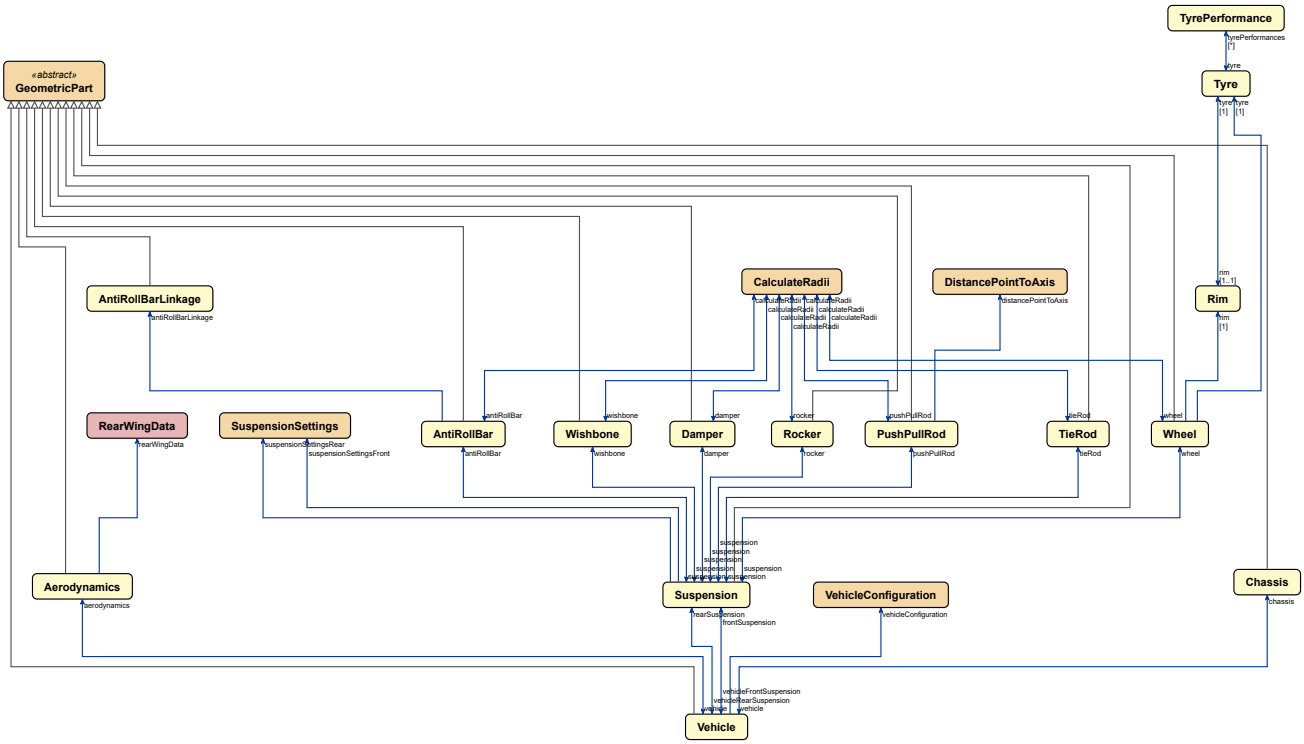


Figure 4. Class diagram of the *suspensionData* package.

At the core of the model is the *Vehicle* class, which acts as the top-level aggregation of all relevant subsystems including the *Chassis*, *Aerodynamics*, and both *frontSuspension* and *rearSuspension*. It contains global vehicle parameters such as center of gravity, weight

distribution, total mass, and wheelbase. These attributes are directly linked through physical relationships, expressed as equations, which support the consistent and automated derivation of key characteristics. The suspension itself is modelled using the *Suspension* class, which is linked to all the main components required for a double-wishbone architecture. These components include the *Wishbone*, *Wheel*, *Damper*, *PushPullRod*, *Rocker*, *AntiRollBar* and *TieRod*. Each component is defined as a subclass of *GeometricPart*, allowing spatial definition via 3D coordinate vectors and parametric control over geometry, stiffness, inertia, and other relevant physical properties [34].

Critical geometric parameters such as bearing diameters, wall thicknesses, and gluing dimensions are explicitly defined and linked via equations to ensure internal consistency. Subcomponents like *Wishbone* and *PushPullRod* include both analytical expressions (e.g., moments of inertia, stiffness calculations) and physical placement logic. In addition, aerodynamic and tire-related data are integrated through the *Aerodynamics*, *Tyre*, *Rim*, and *TyrePerformance* classes. These classes enable analysis of the race car in the context of aerodynamic loads and tire dynamics, including force transmission, slip behavior and rolling resistance, all of which are crucial for lap time simulation [35,36]. Especially for determining tire performance based on test data, a dedicated artificial intelligence feature, recently developed in the publicly funded research project KIEZ4.0, based on symbolic regression [37], is used to extract a symbolic relationship between tire slip angle and lateral forces as a function of normal load. In this example, this feature allows a seamless adaptation of the lap time simulation to any tire type, for which such experimental data are available. In general, this artificial intelligence feature allows to deduce a symbolic expression for the functional relationship implicitly encoded by a set of given data points [37].

The interconnections between components are expressed using the *opposite* keyword, which ensures bidirectional traceability and enables full graph traversal during execution. This means that each physical component is geometrically and semantically embedded within the system architecture, allowing for automated, rule-based instantiation and navigation within the design graph.

3.1.2. Class Diagram *suspensionStates* Package

The *suspensionStates* package (Figure 5) builds on the static definitions offered by the *suspensionData* package, modeling the dynamic states of suspension components during vehicle operation. It captures the kinematic behavior of each suspension element, enabling detailed analysis of suspension characteristics and their impact on vehicle handling.

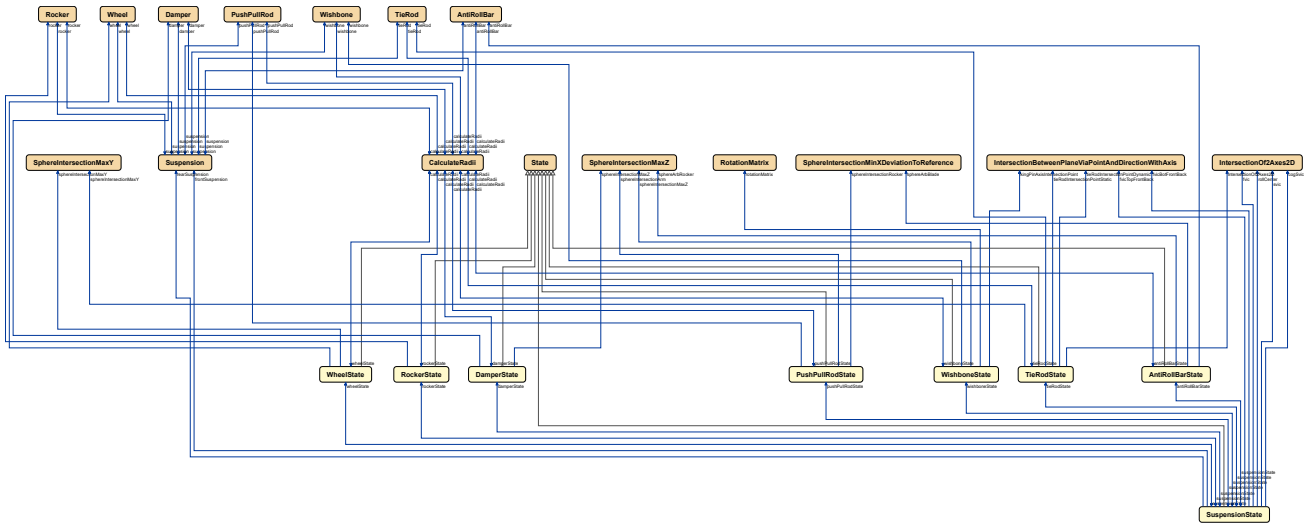


Figure 5. Class diagram of the *suspensionStates* package and its connections to other packages.

The steering linkage is represented by the *TieRodState* class, which dynamically calculates tie rod length and angular displacement relative to the chassis. This allows for an accurate mapping of steering inputs to wheel steering angles and the evaluation of toe variation caused by suspension motion.

Similarly, the *DamperState* class models the current displacement of the damper, providing essential information for simulating spring and damping forces under different suspension states [38]. Figure 6 illustrates the properties and the symbolic equations of the *DamperState* class.

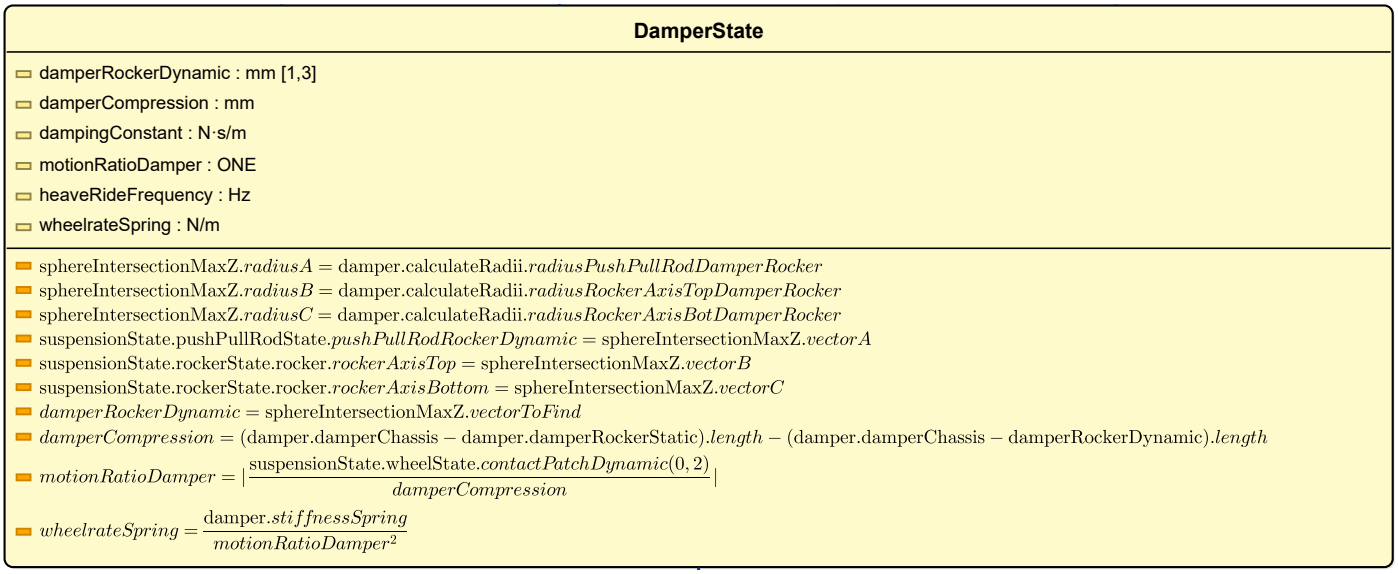


Figure 6. Properties and symbolic equations of the *DamperState* class.

The solution path generator of the compiler solves these symbolic equations when executing the design language. This allows the declarative definition of the symbolic equations in the class diagrams, while the solution procedure is automatically computed internally by the graph algorithms of the solution path generator. To calculate the dynamic damper position on the rocker, the intersection of the spherical coordinates of the *PushPullRod* fixed to the rocker and the *Rocker*'s two axis points are used. To find the correct intersection, the coordinate with the greatest z-direction value is chosen. The current damper compression is then calculated by subtracting the initial damper length from the current damper length. Due to this general form of modeling, the calculation of the dynamic damper state can be reused for different vehicle variants, without requiring additional modifications to the design language or the underlying computational modeling approach.

The anti-roll bar's behavior is captured by the *AntiRollBarState* class, which computes angular displacement and the resulting torque based on suspension roll angles, thereby quantifying the contribution of the anti-roll bar to vehicle roll stiffness and roll stiffness distribution. To gain a deeper understanding of the vehicle's roll characteristics, the package includes calculations for the virtual instantaneous centers of the suspension and the roll center positions. These geometric constructions are derived from specific calculations of line-line and plane-axis intersections and provide an important understanding of the distribution of lateral and vertical suspension stiffness and its effect on vehicle handling [36].

Finally, the *SuspensionState* class provides an integrative framework that combines the dynamic states of all individual components to create a full set of suspension characteristics.

3.1.3. Class Diagram *mathematicalOperations* Package

The *mathematicalOperations* package is used to perform all the necessary mathematical calculations involved in the modeling and analysis of the suspension kinematics. It performs essential geometric and algebraic operations, such as rotation transformations, intersection computations and distance measurements between all relevant components and coordinates. Figure 5 shows the connections between the *mathematicalOperations* package and the other classes, as does the class diagram for the Formula Student race car example in Figure A1.

The *RotationMatrix* class (Figure A2) is central to these operations, as it calculates the rotation of a point in three-dimensional space around an arbitrary axis by a specified angle. First, the class derives a normalized rotation axis vector from the two input vectors that define the axis line. It then uses the Rodrigues rotation formula [39,40] to construct a 3×3 rotation matrix, utilising trigonometric functions of the rotation angle. This matrix is then applied to transform the input point relative to the rotation axis, yielding the rotated point in global coordinates. Further classes address the problem of finding the intersection of three spheres, which is a common geometric operation in the analysis of suspension kinematics. These classes formulate the intersection condition as the set of points that are equidistant from three given centres, each with a specified radius [41]. Selection criteria such as minimizing the deviation in the x-coordinate from a reference point or maximizing the y- or z-coordinate of the solution ensure that the correct intersection point is chosen from among the multiple mathematical solutions. The *CalculateRadii* class performs calculations for various important radii within the suspension system. These include the distances between essential components such as wishbone joints, tie rod connections, push- and pull-rods, rocker arms, dampers and anti-roll bars. These radii represent the fixed lengths of links that are essential for subsequent kinematic analyses. Finally, classes such as *IntersectionOf*2*Axes*2*D* handle intersection calculations between lines and planes, as well as between two linear axes, in both two- and three-dimensional contexts.

Together, these classes form an extensive toolkit for addressing the geometric transformations and spatial intersection issues commonly faced in the design and analysis of various suspension systems.

3.2. Production System Definition

The next logical step is to define the vocabulary and implement a set of transformation rules and programs within the production system. This system governs the execution logic of all relevant rules and subprograms [10].

As shown in Figure 7, the execution logic within the production system (visualized as activity diagram) begins by loading predefined tire data and collecting user-defined simulation parameters via the *userInput* rule. Key settings such as the suspension analysis type (e.g., vertical or steering kinematics), geometry generation and optional optimization steps are then configured and stored in the *SimulationSettings* object.

The *initializeVehicle* program then instantiates all the necessary vehicle components for further computation, as illustrated in more detail in Figure 7. This includes setting up general vehicle data, aerodynamics, the chassis, front and rear suspension, and tires. The program also performs consistency checks by removing unused tire and tire performance entries.

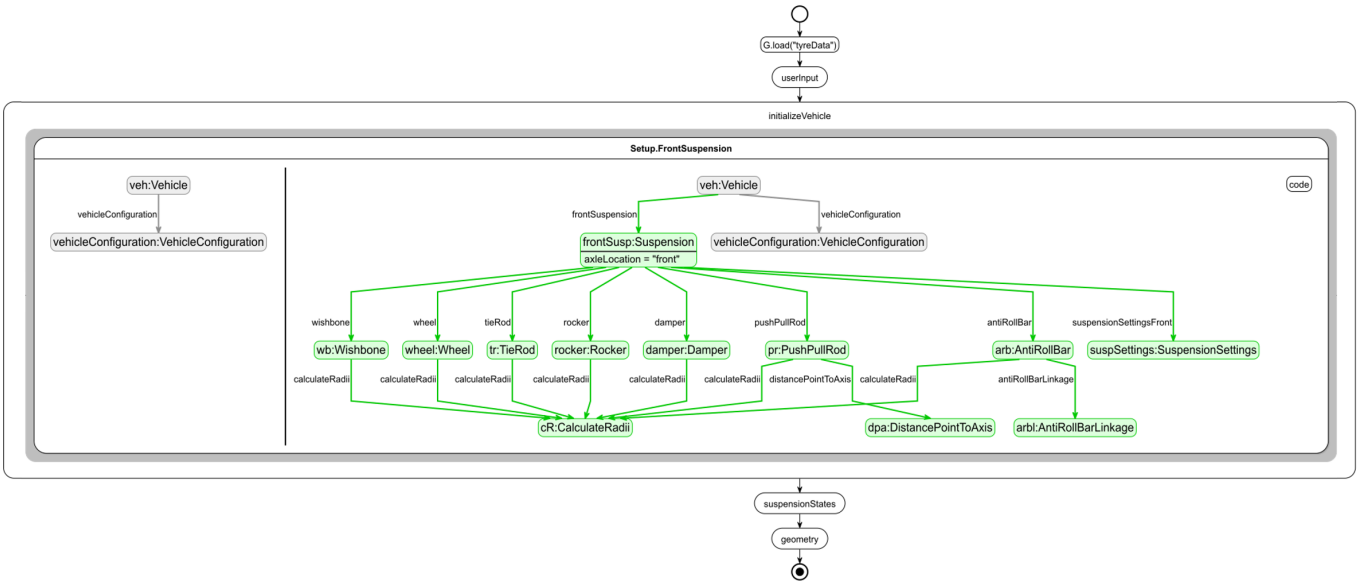


Figure 7. Activity diagram of the Formula Student race car suspension design language (excerpt).

Depending on the chosen simulation type, the *suspensionStates* rule triggers a specific state generation program for either vertical motion analysis, steering kinematics or a zero state by default. In all cases, this is followed by a connection process (*connectMath*), which establishes the links required for the subsequent mathematical computations. Finally, if geometry generation is enabled, the *geometry* rule is executed. This involves a structured sequence of subprograms that are responsible for creating domain-specific representations of suspension components such as wishbones, uprights, dampers, tie rods, and anti-roll bars, to name the most important ones.

4. Results

After executing the underlying design language, the design graph is obtained as the final, comprehensive representation of the vehicle design. In this context, the design graph is a formal, interconnected model of the suspension system that combines structural, behavioural and computational information within a single data structure. It captures the physical configuration of the race car's suspension components and the results of the kinematic analysis, forming the basis for subsequent design validation, evaluation, and optimization [34].

At the core of the design graph in Figure 8 are the instantiated representations of the vehicle's various subsystems. These include the chassis, suspension, aerodynamics and drivetrain.

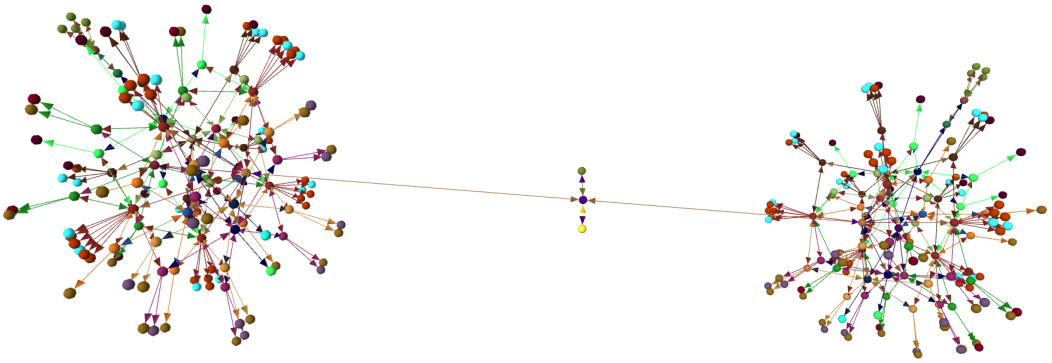


Figure 8. Resulting design graph of the Formula Student race car suspension design language.

Each is modeled with relevant physical parameters, such as geometry, kinematics and system behavior. Associated performance data, such as tire characteristics or stiffness properties, is calculated and then stored in the nodes of the design graph. In addition to the properties of each class, the graph stores a complete set of calculation results specific to the suspension state. Dedicated instances contain the resulting positions, orientations and joint conditions of all relevant components for each defined motion state, such as steering or vertical movement. This enables the suspension system's behaviour to be simulated accurately and quickly under varying vehicle configurations. The primary purpose of Figure 8 is its use as a machine-readable data structure to ensure data consistency across the design process. A human analysis of the design graph is only intended for interactive inspections and therefore depends on additional filtering options available in the design language development environment DC43®. It should be clear that a human interpretation of such a data structure requires an in-depth knowledge of the ontologies from which the graph nodes together with their association and inheritance links are instantiated.

Furthermore, the design graph inherits procedural elements that perform the mathematical operations required for analysis, such as computing geometric intersections, axis-based transformations or projections. These functions are modelled in different classes, as explained in Section 3.1.3, and are directly linked to the component and state instances on which they operate. Consequently, all derived quantities, such as joint trajectories, collisions or motion limits, remain fully traceable and reproducible within the model. Beyond that, the design graph illustrated in Figure 8 serves as the central product model and the 'single source of truth' for the Formula Student race car. Within this holistic digital representation, nodes represent instantiated objects (building blocks) from the domain vocabulary, while edges denote multidisciplinary couplings and functional dependencies. The dense clusters visible in the graph reflect the complex system integration of the suspension, synthesized through automated model transformations.

As a consistent global data structure, the design graph enables the automatic derivation of various domain-specific models, such as parametric CAD or FEM simulations, ensuring multidisciplinary model consistency. In the design graph example shown in Figure 8, the central node in the visualized graph represents the vehicle instance and serves as the root element of the entire structural hierarchy. The dense clusters of connected nodes and edges to the left and right correspond to the front and rear axles, respectively. The stored data from the design graph is then processed further and used to visualize the kinematic characteristics of the racing car. Figure 9 shows a plot of the vehicle track depending on the vertical movement of the wheels for a given Formula Student vehicle.

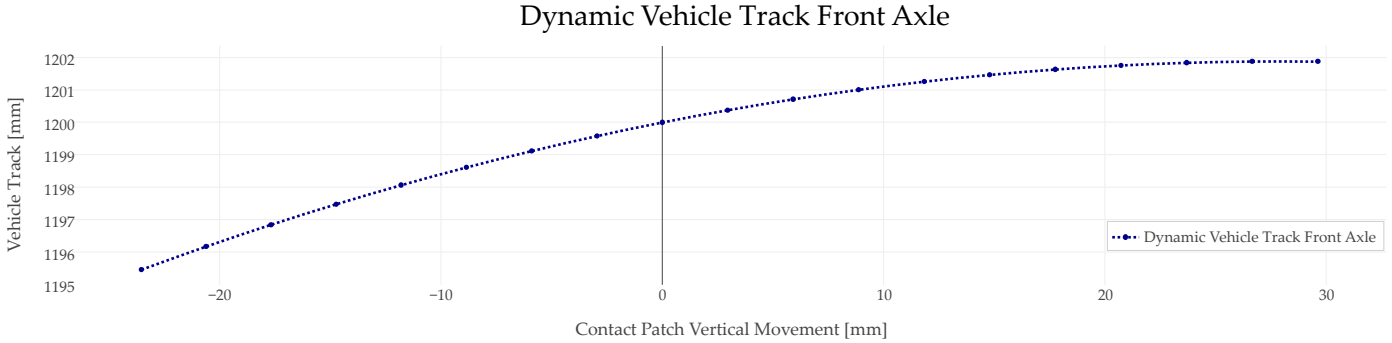
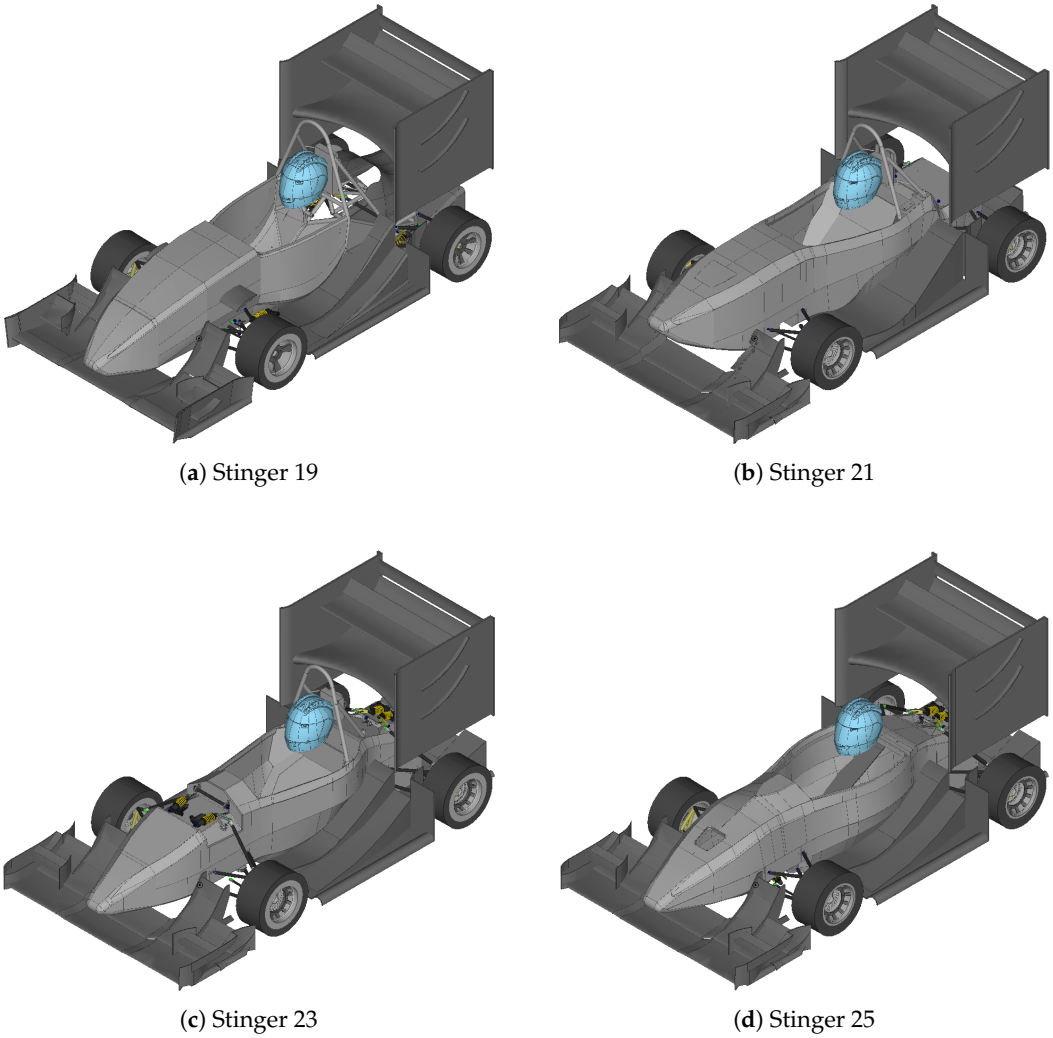


Figure 9. Dynamic change in vehicle track width during heave movement.

The digital modeling framework used to compile and execute graph-based design languages also offers code-based geometry modeling. In this case, a boundary surface model (BREP) [42] is used for geometric representation. Alongside the abstract design of

the vehicle, a corresponding 3D CAD model can be generated and exported in various formats [43]. In addition, components can be directly simulated during modeling through integrated thermal and finite element analysis [44,45]. Beyond single-model simulation, the digital framework enables the automated generation and optimization of multiple vehicle variants. These variants may differ not only in general geometry but also in their underlying topologies, such as the arrangement of suspension and damping systems. By systematically varying design parameters and topology configurations, the framework supports comparative analysis and design evaluation at an early stage of development. Figure 10 shows four automatically generated vehicle variants with different suspension layouts, based on the design of the Formula Student Team Weingarten [46], illustrating the impact of topology changes on the overall vehicle and suspension system design.



(**a**) Stinger 19 (**b**) Stinger 21

(**c**) Stinger 23 (**d**) Stinger 25

Figure 10. Different Formula Student race car design variants.

Since all results from the kinematic analysis are stored in the central design graph, they can be used directly to generate the vehicle geometry for all relevant states of motion. This allows for integrated, dynamic collision detection to verify the current race car design. To accomplish this, the boundary sourface model (BREP) is employed and then triangulated. Based on these triangles, it is determined whether two parts interpenetrate. Figure 11, for example, illustrates a detected interference between the inner wheel and the tie rod geometry during a steering maneuver. The affected components are automatically highlighted in a red color scheme to provide immediate visual feedback. In addition to

visualization, collision information is persistently stored within the design graph and can be used in subsequent steps for automated validation and targeted design optimization.

In summary, the presented results demonstrate the capability of the graph-based design language approach to holistically model, analyze, and evaluate complex suspension systems within a single, unified framework. By integrating geometric modeling, kinematics simulation and topology variation, the approach enables automated early-stage design exploration and validation, as well as the detection of potential issues such as 3D component collisions. The design graph, which contains all the design knowledge, ensures the full traceability (both forward and backward, if need be) of computed properties and provides a consistent basis for subsequent processes, such as optimization of vehicle suspension kinematics or 3D CAD geometry generation.

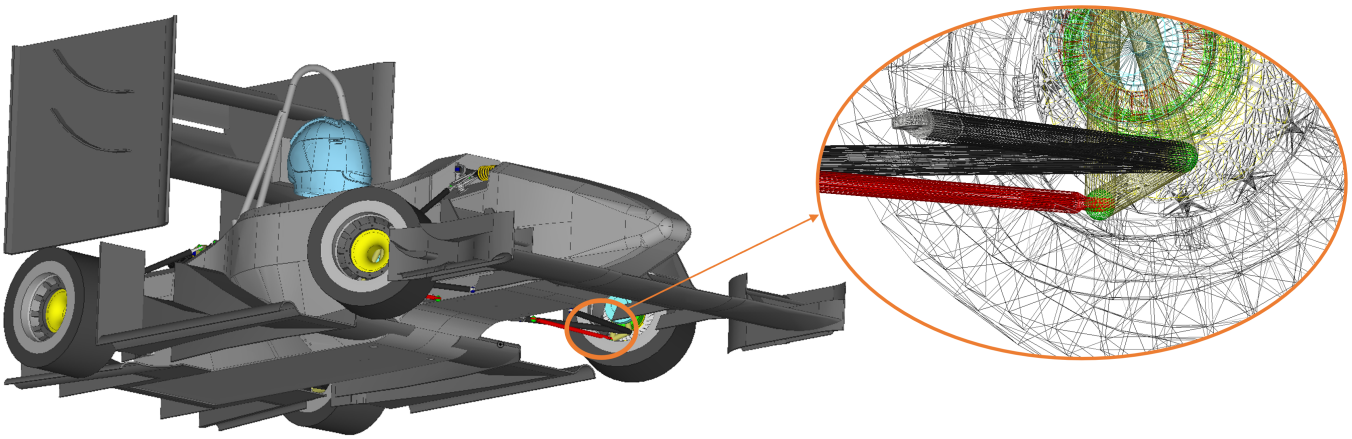


Figure 11. Example of a detected collision (colliding components are marked in red) [34].

Due to automation, the duration of the design process is reduced in principle to the execution time of the algorithms. In this application, one iteration of the Formula Student race car design takes about five minutes on an average PC, and includes suspension kinematics analysis, 3D geometry generation, and collision detection. As a consequence, a time reduction of a formerly manual design task taking several hours, to five minutes only, is achieved. In practice, this reduces the overall kinematic design process of a Formula Student race car from formerly several months to a time period of only two weeks [47].

5. Discussion

This case study demonstrates the effectiveness and advantages of using graph-based design languages to automate and integrate complex engineering tasks. Modeling the Formula Student suspension system within a formal, ontology-driven framework enabled the automatic generation of a design graph that consistently captures behavioral and parametric information across multiple engineering domains.

A key benefit of this approach lies in its ability to encapsulate design knowledge in a form that is both executable and reusable. Unlike traditional CAD- or script-based automation tools, the proposed method allows domain-specific logic, constraints, and structural rules to be expressed declaratively. This promotes greater transparency, traceability, and maintainability. Furthermore, compiler-driven execution ensures the consistent derivation of CAD models, simulation inputs, and design metrics, thereby eliminating the need for error-prone manual handovers between domain-specific software tools. The central design graph provides a unified representation of all relevant engineering data, supporting both forward synthesis and feedback-based refinement. This facilitates iterative design processes, such as adapting geometry based on collision detection or suspension characteristics. By integrating procedural rule sets and optimization algorithms into the same

language environment, the approach enables advanced design space exploration and early validation of alternative topologies.

Analogous to DNA in biology, which encodes the complete blueprint of an organism [48], a graph-based design language can be understood as the 'digital DNA' of a company [49]—or, in this case, of a Formula Student race car [34]. This 'digital DNA' not only formalizes what is designed, but also how it is designed, offering a reproducible, evolvable, and scalable foundation for knowledge-based product development. Once established, such a design language becomes a strategic asset, enabling the reuse of design logic across vehicle generations and the continuous improvement of engineering workflows over several student generations. However, despite all these advantages, several important challenges remain, of which we mention the most important three.

The first challenge, the creation of ontology models and transformation rules, requires an additional initial invest of time and domain expertise, particularly in the case of multidisciplinary system design. Moreover, although the compiler ensures internal consistency, the quality of the generated models depends on the expressiveness and completeness of the physics encoded in the design language. Compared to the manual design process, where a single specific situation and its boundary conditions are in focus, the programming of the general, generic case, which covers a multitude of such situations with almost arbitrary boundary conditions, requires much more intellectual modeling effort and time. It is therefore clear that this extra effort represents an upfront investment which is only compensated for when the so called 'break-even-point' is reached later, after a certain number of automated program executions. However, since it is in generally agreed on that design is iterative by nature, it is very likely that any such 'break-even-point' will be reached either sooner or later.

The second challenge lies on the side of the user: abstract thinking, generic modeling, and a compile step between the source code representation and the resulting models require unfamiliar levels of abstraction, which mechanical engineers are typically neither used to nor educated for. This might hinder an immediate and widespread acceptance of the graph-based design language approach at present and limits it currently to these audiences, which do not experience this as an obstacle. Similarly, the third challenge also lies on the side of the user: object-oriented analysis (OOA), and object-oriented programming (OOP) skills are the new ingredients for success in engineering automation. If not already present, this has to be adequately reflected in future university curricula, which need to include such basics of computer science in their engineering education.

Beyond this need for an initial invest, the case study shows that graph-based design languages provide a robust and scalable approach to managing design complexity, are capable to push automation to a new level and to reduce development time at the same time, thus enabling a machine-readable and machine-executable knowledge-based product development and innovative knowledge management.

As an outlook in view of future-oriented applications of graph-based design languages, the externalized data model in form of the design graph allows for the straightforward integration of several AI-techniques into the shown methodology. Most notably, an integration of Large Language Models (LLMs) using the design graph as input for retrieval augmented generation (RAG) techniques to further improve the answer precision when using the LLM for a complex graph query [47]. Furthermore, as it is straightforward to loop around a fully executable code sequence for optimization purposes, the design language can be embedded in an optimizer, such as particle multi-swarm optimization (PMSO), which has been explicitly chosen due to the unavailability of gradient information (such as collision detection or FEM simulation results) encoded in external process chains [34]. Finally, to mention, the future prospects of graph-based design languages for an improved

knowledge management based on the 'design-as-code' paradigm and supported by the integrated design language version management in the Git-source repository [47].

6. Conclusions

This paper has presented a graph-based design language approach for automating and integrating complex engineering workflows, demonstrated through a case study of a Formula Student race car suspension system design. By formalizing product knowledge within a structured ontology and compiling it into a central design graph, the method enables consistent generation of CAD models, simulation inputs, and design metrics across multiple domains. The results show that this approach significantly reduces manual modeling effort, enables full forward and backward traceability, and supports iterative, knowledge-based development processes. The concept of a graph-based design language as the 'digital DNA' of an engineering organization highlights its potential as a reusable and evolvable foundation for design logic, methodology, and domain integration. Looking to the future, this methodology offers a scalable route towards further engineering automation, variant-driven development and early-stage design optimization based on formal languages and associated language compiler technologies.

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

Funding: This work was supported by the Federal Ministry for Economic Affairs and Climate Action (now: Federal Ministry for Economic Affairs and Energy) as part of the project 'Künstliche Intelligenz Europäisch Zertifizieren unter Industrie 4.0' (KIEZ4-0), conducted within the grant 20D1914C.

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 in order to preserve the competitiveness of the teams. Requests to access the datasets should be directed to the corresponding author.

Acknowledgments: The authors thank the two anonymous reviewers for their constructive comments, which helped to clarify several important aspects of the paper. The authors also wish to thank the cooperating Formula Student Teams of the Ravensburg-Weingarten University, the University of Applied Sciences Pforzheim, the University of Applied Sciences Kiel 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 interests, 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.

Appendix A

Figure A1 shows the class diagram of the Formula Student race car example focussing on the suspension design and analysis.

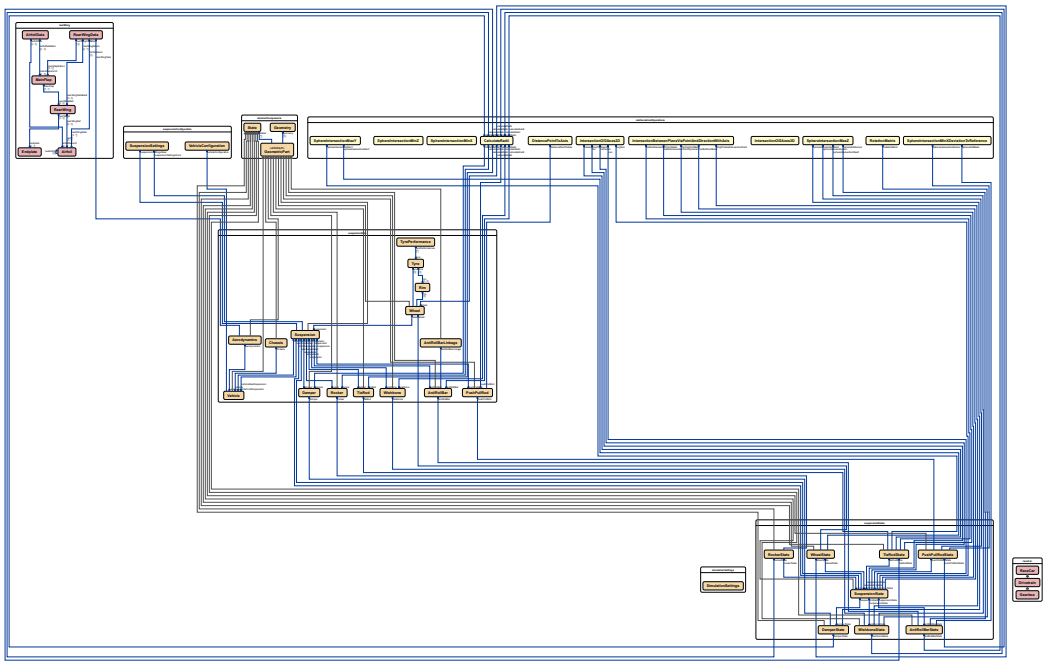


Figure A1. Class Diagram of the Formula Student race car suspension design language.

Appendix B

The *RotationMatrix* class (Figure A2) is used to calculate the rotation of a point in three-dimensional space around an arbitrary axis by a specified angle.

RotationMatrix

- rotationAngle : deg
- rotationAxis : mm [1,3]
- axisVectorA : mm [1,3]
- axisVectorB : mm [1,3]
- pointToRotate : mm [1,3]
- rotatedPoint : mm [1,3]
- transformedPoint : mm [1,3]
- transformedRotatedPoint : mm [1,3]
- rotationMatrix : Double [3,3]
- rotMatrix11 : Double
- rotMatrix12 : Double
- rotMatrix13 : Double
- rotMatrix21 : Double
- rotMatrix22 : Double
- rotMatrix23 : Double
- rotMatrix31 : Double
- rotMatrix32 : Double
- rotMatrix33 : Double

$$rotationAxis = (\text{axisVectorA} - \text{axisVectorB}).normalized$$

$$transformedPoint = pointToRotate - axisVectorA$$

$$rotMatrix11 = \frac{rotationAxis(0,0)^2}{1[\text{mm}^2]} \cdot (1 - \cos(rotationAngle)) + \cos(rotationAngle)$$

$$rotMatrix12 = \frac{rotationAxis(0,0) \cdot rotationAxis(0,1)}{1[\text{mm}^2]} \cdot (1 - \cos(rotationAngle)) - \frac{rotationAxis(0,2) \cdot \sin(rotationAngle)}{1[\text{mm}]}$$

$$rotMatrix13 = \frac{rotationAxis(0,0) \cdot rotationAxis(0,2)}{1[\text{mm}^2]} \cdot (1 - \cos(rotationAngle)) + \frac{rotationAxis(0,1) \cdot \sin(rotationAngle)}{1[\text{mm}]}$$

$$rotMatrix21 = \frac{rotationAxis(0,1) \cdot rotationAxis(0,0)}{1[\text{mm}^2]} \cdot (1 - \cos(rotationAngle)) + \frac{rotationAxis(0,2) \cdot \sin(rotationAngle)}{1[\text{mm}]}$$

$$rotMatrix22 = \frac{rotationAxis(0,1)^2}{1[\text{mm}^2]} \cdot (1 - \cos(rotationAngle)) + \cos(rotationAngle)$$

$$rotMatrix23 = \frac{rotationAxis(0,1) \cdot rotationAxis(0,2)}{1[\text{mm}^2]} \cdot (1 - \cos(rotationAngle)) - \frac{rotationAxis(0,0) \cdot \sin(rotationAngle)}{1[\text{mm}]}$$

$$rotMatrix31 = \frac{rotationAxis(0,2) \cdot rotationAxis(0,0)}{1[\text{mm}^2]} \cdot (1 - \cos(rotationAngle)) - \frac{rotationAxis(0,1) \cdot \sin(rotationAngle)}{1[\text{mm}]}$$

$$rotMatrix32 = \frac{rotationAxis(0,2) \cdot rotationAxis(0,1)}{1[\text{mm}^2]} \cdot (1 - \cos(rotationAngle)) + \frac{rotationAxis(0,0) \cdot \sin(rotationAngle)}{1[\text{mm}]}$$

$$rotMatrix33 = \frac{rotationAxis(0,2)^2}{1[\text{mm}^2]} \cdot (1 - \cos(rotationAngle)) + \cos(rotationAngle)$$

$$rotationMatrix = [[rotMatrix11, rotMatrix21, rotMatrix31], [rotMatrix12, rotMatrix22, rotMatrix32], [rotMatrix13, rotMatrix23, rotMatrix33]]$$

$$transformedRotatedPoint = transformedPoint \cdot rotationMatrix$$

$$rotatedPoint = transformedRotatedPoint + axisVectorA$$

Figure A2. Properties and symbolic equations of the *RotationMatrix* class.

References

1. Vogel-Heuser, B.; Kowalewski, S. Cyber-physische Systeme. *Automatisierungstechnik* **2013**, *61*, 667–668. [CrossRef]
2. Wiecher, C.; Mandel, C.; Günther, M.; Fischbach, J.; Greenyer, J.; Greinert, M.; Wolff, C.; Dumitrescu, R.; Mendez, D.; Albers, A. Model-based Analysis and Specification of Functional Requirements and Tests for Complex Automotive Systems. *arXiv* **2022**, arXiv:2209.01473. [CrossRef]
3. 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. *Syst. Eng.* **2023**, *26*, 365–378. [CrossRef]
4. Vogelsang, A.; Fuhrmann, S. Why Feature Dependencies Challenge the Requirements Engineering of Automotive Systems: An Empirical Study. In Proceedings of the 21st IEEE International Requirements Engineering Conference (RE), Rio de Janeiro, Brazil, 15–19 July 2013; pp. 267–272.
5. Ferraris, A.; Cupis, D.; Pinheiro, H.; Messana, A.; Sisca, L.; Airale, A.; Carello, M. Integrated Design and Control of Active Aerodynamic Features for High Performance Electric Vehicles. In Proceedings of the 2020 SAE Brasil Congress and Exhibition, Sao Paulo, Brazil, 1 December 2020.
6. Hick, H.; Küpper, K.; Sorger, H. *Systems Engineering for Automotive Powertrain Development*; Springer Nature: Cham, Switzerland, 2021.
7. Pohl, K.; Hönninger, H.; Achatz, R.; Broy, M. *Model-Based Engineering of Embedded Systems: The SPES 2020 Methodology*, 1st ed.; Springer: Berlin/Heidelberg, Germany, 2012.
8. Estefan, J. Survey of Model-Based Systems Engineering (MBSE) Methodologies. International Council on Systems Engineering (INCOSE). INCOSE-TD-2007-003-02. 2008. Available online: https://www.omg.org/sysml/MBSE_Methodology_Survey_RevB.pdf (accessed on 23 July 2025).
9. Santos, C.; Kalume Neto, M.; Kanieski, J.M.; Rufino Júnior, C.A.; Schweiger, H.-G.; Zanin, H. A multidimensional assessment of electrification in automotive powertrains: Technical, operational, and strategic perspectives. *Results Eng.* **2025**, *28*, 107377. [CrossRef]
10. Rudolph, S. Digital continuity, consistency and interoperability along the product life-cycle using graph-based design languages. In Proceedings of the Global Product Data Interoperability Summit (GPDIS 2023), Phoenix, AZ, USA, 18–21 September 2023. Available online: https://gpdisonline.com/wp-content/uploads/2023/10/IILSmbH-StephanRudolph-DigitalContinuityConsistencyAndInteroperabilityAlongThePLC-MBSE-Open.pdf (accessed on 16 July 2025).
11. Schmidt, J.; Rudolph, S. Graph-Based Design Languages: A Lingua Franca for Product Design Including Abstract Geometry. *IEEE Comput. Graph. Appl.* **2016**, *36*, 88–93. [CrossRef]
12. Borowski, J. Digitaler Entwurf eines Formula Student Fahrzeugs am Beispiel von Fahrwerk und Antriebsstrang zur 3D-Geometrie- und Kinematikerstellung. Ph.D. Thesis, University of Stuttgart, Stuttgart, Germany, 2026.
13. Yang, L.; Cormican, K.; Yu, M. Ontology-based systems engineering: A state-of-the-art review. *Comput. Ind.* **2019**, *11*, 148–171. [CrossRef]
14. Schmidt, J.; Rudolph, S. Gaining system design knowledge by systematic design space exploration with graph based design languages. *AIP Conf. Proc.* **2014**, *1618*, 390–393.
15. Ingenieurgesellschaft für Intelligente Lösungen und Systeme mbH. Available online: https://www.iils.de/en/product/design_cockpit_43/ (accessed on 21 July 2025).
16. Object Management Group. Available online: https://www.omg.org/spec/SysML/ (accessed on 6 January 2026).
17. Vizitiu, C.; Dinculescu, A.C.; Marin, M.G.; Nistorescu, A.I.; Dominey, K.A.; Kristály, D.M. Optimizing early-stage efficiency in complex system development via model-based systems engineering (MBSE) and concurrent engineering (CE) integration. *Res. Eng. Des.* **2025**, *36*, 12. [CrossRef]
18. Pereira Pessôa, M.V.; Jauregui Becker, J.M. Smart design engineering: A literature review of the impact of the 4th industrial revolution on product design and development. *Res. Eng. Design.* **2020**, *31*, 175–195. [CrossRef]
19. Vinck, D. Engineering Practices. Revue d'Anthropologie des Connaissances. 2014. Available online: https://doi.org/10.3917/rac.023.0226 (accessed on 24 July 2025).
20. Ahmed Khan, W.; Esat, V.; Hammad, M.; Ali, H.; Qasim Zafar, M.; Ali, R. *Computer Aided Engineering Design and Manufacturing*; Springer: Cham, Switzerland, 2025.
21. Unnikrishnan Nair, S.; Somanath, S. *Introduction to Finite Element Analysis*; Springer: Singapore, 2024.
22. Kim, N.-H. *Introduction to Nonlinear Finite Element Analysis*; Springer: Cham, Switzerland, 2026.
23. Majumder, P.; Choudhury, M.R.; Karmaka; G.; baishya, P.; Singh, S. Optimizing aerodynamics: Numerical flow analysis for drag and lift reduction in SUV designs. *Interactions* **2024**, *245*, 338. [CrossRef]
24. Teschemacher, T.; Bauer, A.M.; Aristio, R.; Meßmer, M.; Wüchner, R.; Bletzinger, K. Concepts of data collection for the CAD-integrated isogeometric analysis. *Eng. Comput.* **2022**, *38*, 5675–5693. [CrossRef]
25. Li, M.; Lin, C; Chen, W.; Liu, Y.; Gao, S.; Zou, Q. XVoxel-Based Parametric Design Optimization of Feature Models. *Comput.-Aided Des.* **2023**, *160*, 103528. [CrossRef]

26. Rinker, F.; Waltersdorfer, L.; Meixner, K.; Winkler, D.; Lüder, A.; Biffl, S. Traceable Multi-view Model Integration: A Transformation Pipeline for Agile Production Systems Engineering. *SN Comput. Sci.* **2023**, *4*, 205. [CrossRef]
27. Torres, W.; van den Brand, M.G.J.; Serebrenik, A. A systematic literature review of cross-domain model consistency checking by model management tools. *Softw. Syst. Model.* **2020**, *20*, 897–916. [CrossRef]
28. Sztipanovits, J.; Koutsoukos, X.; Karsai, G. Toward a Science of Cyber-Physical System Integration. *Proc. IEEE* **2012**, *100*, 29–44. [CrossRef]
29. Gujarathi, G.P.; Ma, Y.-S. Parametric CAD/CAE integration using a common data model. *J. Manuf. Syst.* **2011**, *30*, 118–132. [CrossRef]
30. Wong, F.S.; Wynn, D.C. A systematic approach for product modelling and function integration to support adaptive redesign of product variants. *Res. Eng. Des.* **2023**, *34*, 153–177. [CrossRef]
31. Jinzhi, L.; Junda, M.; Zheng, X.; Wang, G.; Kiritsis, D. Design Ontology Supporting Model-based Systems-engineering Formalisms. arXiv **2020**, arXiv:2010.07627. [CrossRef]
32. Vogel, S.; Arnold, P. Towards a more complete object-orientation in graph-based design languages. *SN Appl. Sci.* **2020**, *2*, 1235. [CrossRef]
33. Zhong, S.; Scarinci, A.; Cicirello, A. Natural Language Processing for systems engineering: Automatic generation of Systems Modelling Language diagrams. *Knowl.-Based Syst.* **2023**, *259*, 110071. [CrossRef]
34. Borowski, J.; Rudolph, S. Automation and Optimization of the Design and Development Process for a Formula Student Racing Car Suspension. In Proceedings of the 2025 Stuttgart International Symposium, Stuttgart, Germany, 2–3 July 2025.
35. Bakker, E.; Nyborg, L.; Pacejka, H. Tyre modelling for use in vehicle dynamics studies. In Proceedings of the 1987 SAE International Congress and Exposition, Detroit, MI, USA, 23–27 February 1987.
36. Frömmig, L. *Grundkurs Rennwagentechnik: Einführung in das Zusammenwirken von Reifen, Fahrwerk, Aerodynamik, Differenzialsperren und Rahmen*, 1st ed.; Springer Vieweg: Wiesbaden, Germany, 2019.
37. Anselment, M.; Neumaier, M.; Rudolph, S. Systematic tree search for symbolic regression: Deterministically searching the space of dimensionally homogeneous models. *CEAS Aeronaut. J.* **2025**. [CrossRef]
38. Borowski, J.; Stetter, R.; Rudolph, S. Design, Dimensioning and Simulation of Inerters for the Reduction of Vehicle Wheel Vibrations—Case Studies. *Vehicles* **2020**, *2*, 424–437. [CrossRef]
39. Gray, J.J. Olinde Rodrigues' paper of 1840 on transformation groups. *Arch. Hist. Exact Sci.* **1980**, *21*, 375–385. [CrossRef]
40. Murray, R.M.; Li, Z.; Sastry, S.S. *A Mathematical Introduction to Robotic Manipulation*; CRC Press: Boca Raton, FL, USA, 1994; Volume 1.
41. Gründemann, H. *Geometrische Strukturen der Kontinuumsphysik: Raum-Zeit-Exkursionen von der Euklidischen zur Riemannschen Geometrie*, 1st ed.; Springer Spektrum: Berlin/Heidelberg, Germany, 2022.
42. Hoffmann, C.M. *Geometric and Solid Modeling*, 1st ed.; Morgan Kaufmann Publishers: San Mateo, CA, USA, 1989.
43. Vukasinovic, N.; Duhovnik, J. *Advanced CAD Modeling—Explicit, Parametric, Free-Form CAD and Re-engineering*, 1st ed.; Springer Nature: Cham, Switzerland, 2019.
44. Munjiza, A.; Rougier, E.; Knight, E.E.; Lei, Z. Discrete Element and Particle Methods. In *Encyclopedia of Continuum Mechanics*, 1st ed.; Altenbach, H., Öchsner, A., Eds.; Springer: Berlin/Heidelberg, Germany, 2020.
45. 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 Vieweg: Wiesbaden, Germany, **2024**.
46. Formula Student Team Weingarten. Available online: https://www.fsteamweingarten.de/fahrzeuge21/ (accessed on 24 July 2025).
47. Borowski, J.; Rudolph, S. Managing Design Variants in Formula Student Race Cars: A Digital Engineering Approach Across Multiple Teams. In *Vehicle Design Processes*, 3rd ed.; MDPI: Basel, Switzerland, 2026; ISSN 2624-8921.
48. Woody, S.; Himelblau, E. Understanding and Teaching Genetics Using Analogies. *Am. Biol. Teach.* **2013**, *75*, 664–669. [CrossRef]
49. Schumacher, S.; Stetter, R.; Till, M.; Laviolette, N.; Algret, B.; Rudolph, S. Simulation-Based Prediction of the Cold Start Behavior of Gerotor Pumps for Precise Design of Electric Oil Pumps. *Appl. Sci.* **2024**, *14*, 6723. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.