

TOWARDS AN ONTOLOGY-BASED APPROACH FOR DISCIPLINE-SPECIFIC ENGINEERING VIEWS IN ENGINE DESIGN

Kai Becker[1], Jakob P. Haug[1] & Matthias Schuff[1]

[1]German Aerospace Center (DLR), Institute of Test and Simulation for Gas Turbines, Forschungsallee 1, D 86159 Augsburg, Germany

Abstract

For future improvements in the design of aero engines, various, often conflicting objectives, such as high thermal efficiency and long component lifetimes, have to be achieved. Accordingly, technical solutions must address these complex and multi-disciplinary challenges and integrate them into a holistic approach. A major challenge is to provide a global data model for the engine design to process the data generated throughout their entire life cycle, from the early design phase to the end of their service life. A monolithic model suitable for all involved disciplines will ultimately be difficult to maintain and also to develop further. During development, disciplines in interdisciplinary areas would have to agree on terminology. Problems might arise as these notations must also be accepted in the everyday work of engineers. In addition, the introduction of new topics or disciplines may require renewed discussions about terminology. There are currently many approaches that aim to use ontologies to merge heterogeneous data into a large knowledge graph. This paper presents an approach that uses a basic ontology for storing data in nodes and their relationships and uses translators to convert the data and information into a discipline-specific (hierarchical) view. The idea is presented using the example of simulations in the context of engine development. Advantages and challenges are discussed from various perspectives, each with a focus on a different aspect.

Keywords: Knowledge Graph, Ontology-Based Modeling, Discipline-Specific Engineering Views, Simulation Data Management

1. Introduction

The development of modern aero engines is characterized by a growing level of technical complexity and an increasing consideration of multi-disciplinary aspects. For example, aerodynamic performance, structural integrity, thermal behavior and system-level constraints must be considered in a joint approach. Increasingly stringent performance and safety requirements within limited development timeframes as well as financial constraints therefore require close cooperation between engineers from different fields. As a result, computer-assisted collaboration has become an essential element of contemporary aerospace engineering practice.

A key enabler for such collaboration is the effective integration of engineering data and models across disciplines and abstraction levels. Engineers typically work with heterogeneous models that reflect domain-specific viewpoints, ranging from conceptual system architectures to detailed geometric, aerodynamic and structural analyses. To support multi-disciplinary reasoning and coordinated decision-making, these models must be consistently related and interpretable across domains. Model-level data integration, rather than mere file exchange, is therefore critical for enabling and enhancing computer-assisted engineering and design in complex aerospace projects.

A common approach to achieve such integration is to define a general, shared data model that serves as a common foundation for the considered disciplines. While this strategy is conceptually appealing,

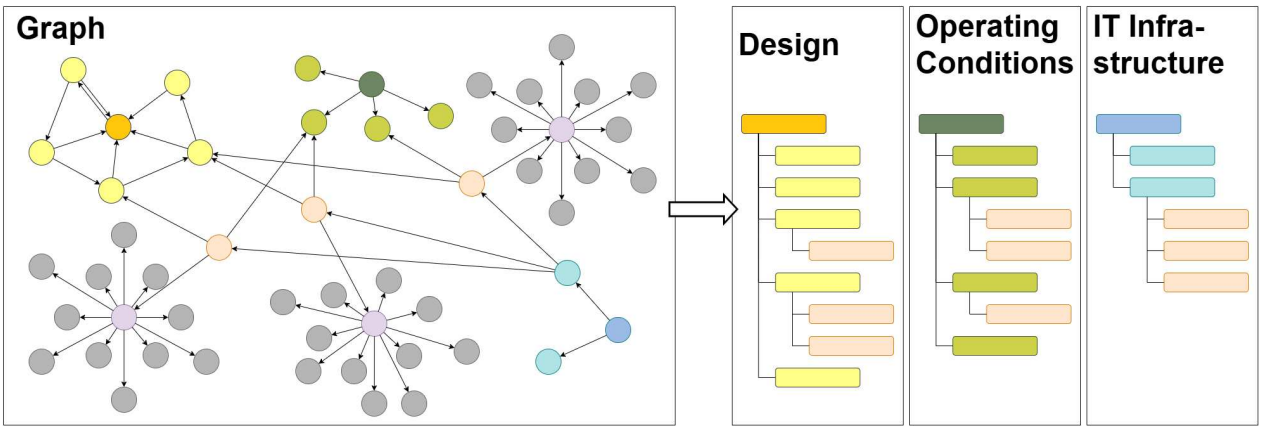


Figure 1 – Idea of different views derived from the same graph.

it often leads to the gradual growth of large, monolithic data models. As further disciplines contribute their specific concepts, parameters and constraints, the resulting models become increasingly complex, difficult to navigate and cumbersome to use in everyday engineering work.
Over time, this growing complexity can negatively impact usability and acceptance. Engineers are confronted with data models that expose far more information than is relevant for their specific tasks. This makes it difficult to identify the concepts and relationships that matter within a given discipline. As a result, motivation to work with the shared models decreases and engineers tend to revert to discipline-specific tools and informal workarounds. This undermines the original goal of integrated, collaborative engineering and reintroduces fragmentation across domains.
These challenges highlight the need for integration approaches that support structured, discipline-specific views on a shared semantic foundation. These approaches do not rely on ever-expanding monolithic data models. This paper focuses on the ontologies used for simulations and their application in creating hierarchical and domain-specific views for the collaborative engine design.
Figure 1 illustrates the basic concept using a simplified representation of simulations. The left graph shows the entire knowledge graph, featuring specific data models for representing design (yellow), operating conditions (green), the IT infrastructure (blue) and simulations (beige), along with the corresponding results (violet, gray). In principle, the individual data models can also be structured hierarchically, as such structures can be represented in graphs using their parent-child relationships. In this example, the numerical model of a simulation is derived from a specific design. It is run on a computer for a specific operating point and generates simulation results. Different task requirements can lead to differently structured hierarchical representations.
The right-hand side shows perspectives that focus either on the design, the operating conditions or the IT landscape in use, with the simulations sorted into different positions within the tree. This immediately shows, for example, which simulations were performed for which design element. Using graphs, it is not necessary to anchor simulations at an exact position in the data system. Furthermore, to add additional properties, it is sufficient to link the respective nodes via a relationship. For this example, it might be important to also record the solver settings. It is then up to the derived perspectives to decide whether they will incorporate this new information.
This approach can be applied to significantly more complex processes, such as the various disciplines involved in engine design. The idea here is to enable engineers to continue working with their familiar and preferred view of the data while also being able to easily integrate and keep synchronized data and information from other areas that were previously difficult and time-consuming to access. Therefore, a data management system is presented here that focuses on ontology-based graphs and is designed to handle heterogeneous data. The discipline-specific views are achieved by tailored representations matching their respective perspectives and abstraction levels.

2. Ontology-based Data Modeling

Across the literature, several topics consistently emerge in the management of data over the complete product life cycle. Ontologies have been providing a rigorous foundation for consistent knowledge representation in engineering, enabling reuse and semantic interoperability for more than 30 years now ([1, 2, 3]). Within engineering, they emerged as a response to growing complexity, heterogeneity of tools and the need for semantic interoperability in concurrent designs. They have the potential to bridge the gap between otherwise incompatible technical representations. In aero engine design, for example, the same turbine blade can appear as CAD geometry in the design phase, as a flow boundary in CFD simulation, as a meshed volume model in structural analysis and as a thermal component in cooling simulations. An ontology can link these domain-specific views by defining their shared meaning, such as the blade as a technical component with geometry, material properties, loads, temperatures and operating conditions, while each simulation perspective retains its own representation. Especially, the complexity and safety-critical nature of the products in aerospace offers considerable scope and flexibility for the application of ontologies. In recent years, frameworks for developing ontologies for technical design have been introduced, particularly for the aerospace industry, with a focus on modularity, coverage of the entire lifecycle and integration into design processes ([4, 5]). Zamboni et al. outlined a knowledge-based engineering framework that enables the rapid generation of novel aircraft configurations and facilitates the transparent digitization of knowledge from the design phase, using Semantic Web technologies for its interoperability [6].

Ontologies have the potential to help formalize design knowledge, managing configuration alternatives and assisting traceability across design phases. They support consistent terminology, unified modeling concepts and semantically defined relationships between artifacts such as requirements, geometric features, loads, boundary conditions and performance metrics. This ability is a key factor in the multi-disciplinary design environment ([7]). The collaborative design approach for modern aero engines comprises contributions from multiple disciplines, each with its own modeling assumptions, design abstractions and analysis methodologies. Back in 1996, Rosenman and Gero introduced the idea of multiple abstraction hierarchies in collaborative design ([8]). Involved engineering teams should keep their discipline-specific views that are nonetheless connected through a coherent underlying representation. Mayer et al. demonstrated in 2008 how meta-modeling techniques are able to bridge representational and semantic differences between analysis disciplines and execution environments ([9]).

Nowadays, Model-Based Systems Engineering (MBSE) has become a dominant paradigm for managing the complexity of aerospace systems. Because of the heterogeneity of the various modeling languages, integration challenges arise. Ontology-based MBSE seeks to provide a formal semantic layer linking otherwise disparate system models.

2.1 Hierarchical Perspective

In order to demonstrate the strengths of ontology-based modeling, a classic hierarchical representation is first discussed. Consider the task of gathering simulations of various geometric variations of a small subcomponent in an aero engine as an example, a classic geometric model is used as the basis as in [10]. An engine comprises several components, such as the high-pressure compressor, the combustion chamber and the high-pressure turbine. The secondary air system can be considered a further component, which in turn consists of several components, including so-called labyrinth seals. Such a labyrinth seal will be used as an example in the following explanations. The use case shall further include geometric variations of this subcomponent. This raises the question of whether a completely new design must be created for every variant that is more than 99% identical or whether there is another way to handle this. For simplicity, the various geometric variations are saved under the component and the simulations - along with their settings and results - are saved under the respective geometric variations. The information about the setup must also be recorded in the representation. The question, however, is how and where this should be done. If, for example, the solver settings are saved for each simulation, it can be assumed that they are identical for each simulation here. The operating points, considered for each geometry variation, will also be largely identical. This results in a significant amount of information being stored redundantly in the model.

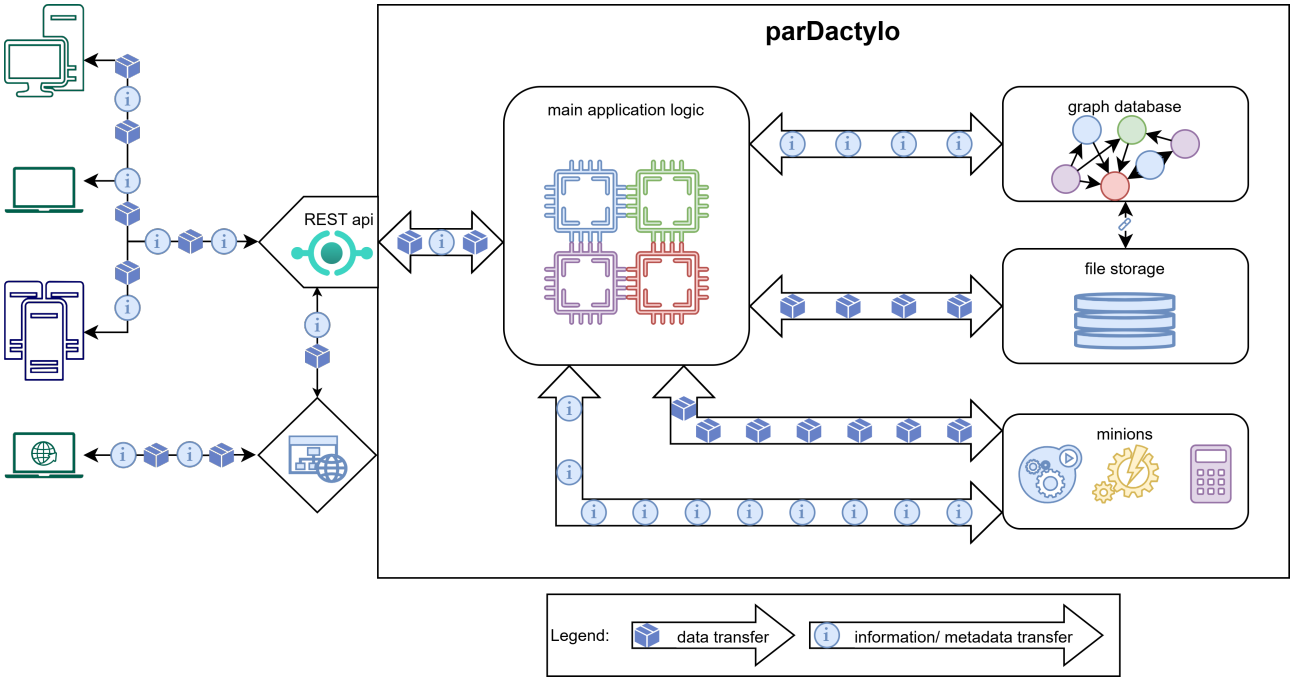


Figure 2 – Structure of the data management system parDactylo ([11])

One possible solution would be to store the information only once and then create links to it in the respective simulations. This would still allow the hierarchical structure to be maintained.
But what if the user wants to view all simulations for a specific operating point because the user is curious which design performed best here? This view would require a different hierarchical sorting of the data. The data described here contains a large number of cross-references to other data. Therefore, there is a strong case for storing the data directly as a graph and providing an appropriate view depending on the query as already explained in Figure 1. This is precisely the approach being investigated in more detail in this paper. The necessary components for this are first presented below.

2.2 Implementation of the Data Management System parDactylo

At the institute of test and simulation for gas turbines, the data management system (DMS) parDactylo is being developed for the collaborative architectural design for aero engines. parDactylo relies on ontologies as a semantic foundation and uses them as the basis for data and process modeling. Large amounts of heterogeneous data are generated during the design process, distributed across the various process steps. This data comprises a wide variety of data sizes and types and results from the many different involved disciplines requiring a consistent formulation of the data including everything from the preliminary design processes to detailed high-fidelity simulations of individual components or component groups. Linking all this information in one central data management system is the key to even more efficient, future aero engines with less environmental impact and lifetime costs by reusing all information along the full design process chain with a holistic approach. To provide some background information on the overall system despite this paper's focus on using ontologies and generating source code from them as foundation for discipline-specific engineering views, a brief overview on the structure of parDactylo is given in the following paragraph.
The basic components of parDactylo, illustrated in Figure 2, are a graph database, the file storage, the interface and main application logic and modular functionalities as well as on-demand data processing (minions). parDactylo is based on a knowledge graph which contains information about all files and information while also keeping track of all actions which create, modify or alter data. Within the knowledge graph extensive annotation of data enables complex queries and searches on existing data and thus easy access, reuse and re-contextualization of available knowledge in current design processes. Different stakeholders, regardless of their physical location, are able to access all data possibly necessary and useful for their design tasks. This holistic approach drastically drives the continuous improvement of design processes as all generated data is available to all participants,

whether human or machine, to be used in their processes. The underlying concept of parDactylo was introduced in recent publications [11, 12] and led to a knowledge-graph-backed layer with a modular, extensible distributed file storage architecture.
In parDactylo, every asset, be it a file, a parameter set, or a derived result, is represented as a node in the graph, while the graph's edges encode semantic relationships. The actual binary files are stored on heterogeneous storage backends and referenced in the graph via immutable links. This design enables automatic tracking of every modification, access, or transformation, thereby providing full data-provenance capabilities and allowing users to efficiently query the entire data ecosystem without the need to retrieve and open the files. The files only need to be accessed when the provided metadata is not sufficient for querying. Consequently, parDactylo not only fulfills the imperative to expose every data set but also turns the wealth of existing information into a catalyst for continual methodological advancement. Disturbances of workflows are reduced by enabling fast access to the available knowledge. Due to the modular structure, different data storage solutions, even already deployed and used ones, may be used with parDactylo as long as a unique link to the stored files is available.
In Figure 2, the data and information transfers between these internal components as well as from and to the inquirer are shown. parDactylo's capabilities are briefly demonstrated using the example of downloading a file or getting information of that file to the user's local system. As a starting point, the inquirer needs the universally unique identifier (UUID) by which the file is identified in the graph database. The request is then processed with the following steps:

1. The UUID is transferred to parDactylo via the REST API and passed on to the main application logic.

2. The main application logic queries the graph database (graph container) to get the node and the related information in the graph database with the provided UUID. The details for querying the file management system are extracted from the node's properties.

3. Using this information, the data storage is queried for the respective file. This file is then fetched from the data storage.

4. If the inquirer requested some further processing, e.g. extracting data from the file, these jobs are now executed by the respective minions, orchestrated by the minion coordinator.

5. The file and/or possible files and information provided by the minions are sent back to the user via the REST API.

When uploading files to parDactylo these steps are executed in the reverse order, adding the file to the data storage first and then creating corresponding nodes in the graph database with the link to the file in the file storage. Queries that only involve metadata or information involve only the graph database with no need to access the data storage part of parDactylo.

2.3 Data Provenance

The graphs as the key feature in parDactylo are based on a foundational ontology, shown in Figure 3, that is derived from the PROV ontology (PROV-O)[13]. The base class is called *DatabaseEntry* and contains all important data properties for identification (name, UUID and type), for time tracking (creation date, date of last modification) as well as a field for a JSON array for optional additional properties. The three main PROV classes, *Activity*, *Agent* and *Entity*, are derived from the base class DatabaseEntry.
Besides these PROV core classes, an additional abstract class, denoted *Abstract*, is defined. This class is used to model conceptual reference entities, e.g. physical units, which are represented by a single element in the graph. Besides the relationships defined in PROV-O, the base class provides two additional general relationships for parent-child relationships (*has*) and dependencies (*requires*). All relationships are written in the present tense, as the ontology is used to create data records and not just to track the data provenance. Ontologies describing specific use case scenarios are derived from this baseline ontology.

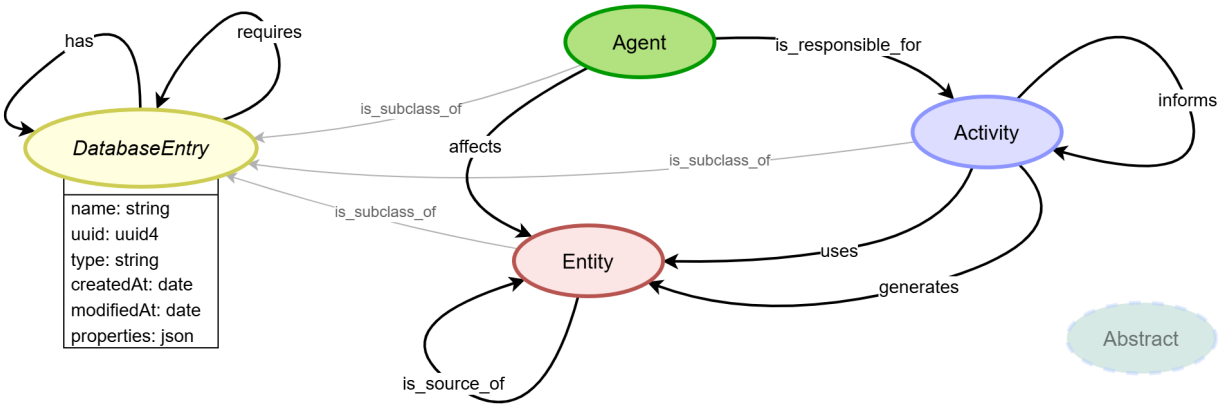


Figure 3 – Basic ontology of database elements based on [13].

To describe hierarchical relationships in this paper, a path notation inspired by property-graph query languages is used. Each view element is denoted by the ontology class or role used for its representation in the hierarchy. Directed relationships are represented by `-[:r^k]->`, where `r` denotes the relationship type and k denotes the number of graph traversal steps. For example, `-[:has_domain^1]->` denotes a direct `has_domain` relationship, whereas `-[:has_domain^+]->` denotes one or more successive `has_domain` relationships.

2.4 Simulation Topology

The simulation topology was introduced by Schuff to extend an existing data model of the engine allowing to manage complex simulation configurations ([14, 15]). The model is designed to link computational domains with the geometric features found in predesign and allows to integrate 3D high-fidelity simulations into the overall design considerations. The domain-based description of the simulation topology allows to couple different domains across disciplines and fidelity levels in the collaborative engine design. The goal is to overcome the shortcomings of a geometry-based description of the individual components by closing the gap between predesign and high-fidelity simulations.
The base entities of the simulation topology are illustrated in Figure 4. All entities are derived from *Entity* in section 2.3. In Figure 4, this is indicated by a red border. The two main entities are domain and topology. The topology comprises the region of interest for a simulation and includes all other entities. The domains are single-discipline parts of the topology and can themselves be subdivided into further domains. Domains are connected by interfaces to each other and / or to the system boundaries. Each interface needs one or more boundaries and is designed for the data exchange between two or more domains. Depending on the type of boundary, a different exchange is required. Surfaces describe walls to which, e.g., pressure is applied or deformation and heat flux is exchanged. In con-

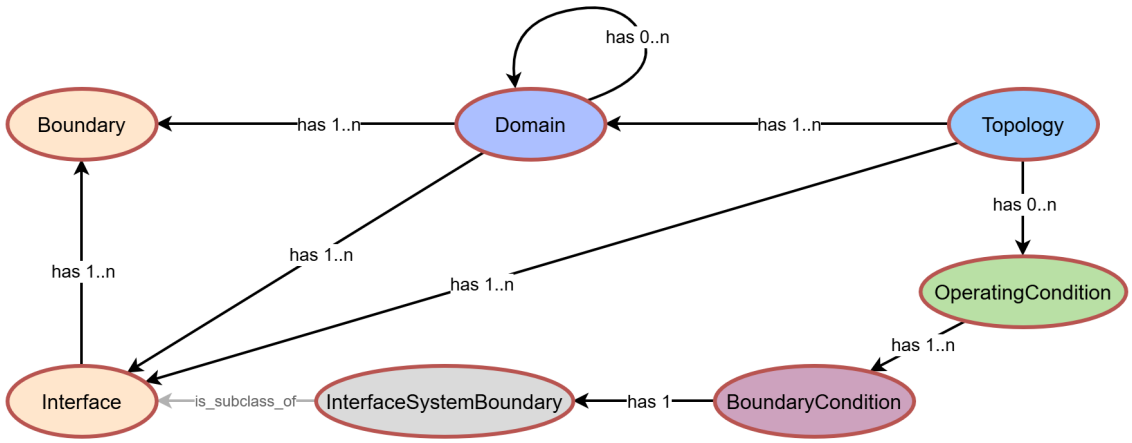


Figure 4 – Baseline ontology of the simulation topology

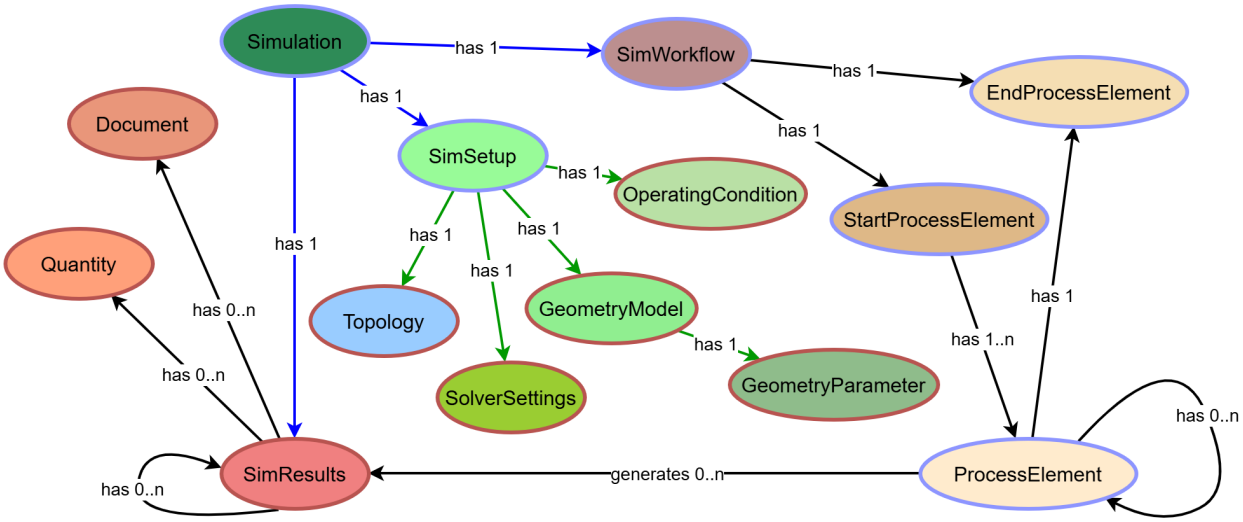


Figure 5 – Baseline ontology for simulations

trast, fluid ports are designed to allow fluid to flow through them. A special interface type is the system boundary interface (*InterfaceSystemBoundary*). The system boundary interfaces also serve as the interfaces of the underlying topology to the outside (system boundaries), each requiring a specific boundary condition. To perform a simulation, boundary values (*Boundary*) have to be prescribed for each system boundary (*InterfaceSystemBoundary*). Therefore, an appropriate set of boundary conditions (*BoundaryCondition*) is clustered as an operating condition (*OperatingCondition*). Conversely, every connected operating condition must contain a valid set of boundary conditions by providing one boundary condition for every present system boundary.

2.5 Simulation Representation

Figure 5 shows the ontology for a complex simulation associated with the already described simulation topology. The simulation is derived from the activity marked by its blue border whereas entities have a red border. The simulation is linked to two other activities, a setup and a workflow of any complexity as well as the entity for the results. The setup is connected further to four entities, the geometric model, the solver settings and the aforementioned topology as well as one operating condition. For the scope of the paper, a highly simplified geometric model is considered, in which only model variations are captured using individual geometric parameters. In the actual design process, the geometric and data models are far more detailed, as described in [10, 16], for example. The numerical model for the simulation is created from the geometric model and the topology in the setup process. The numerical settings are configured using the solver settings and the selected operating condition. The specified operating condition must be a valid set for the used topology and must therefore already have a connection to topology, as shown in Figure 4. Therefore, the number of possible simulations sums up to the combination of available topologies, operating conditions, solver settings and geometry variations.

The description of workflows follows the Business Process Model and Notation (BPMN) [17]. This implies that the actual workflow is enclosed by start and end elements. The process in between can take any form and include elements such as tasks, decisions or events. The process elements generate results, which may consist of values and/or files. The result nodes themselves can be nested. For the data provenance all elements and the files, that are marked as results via an appropriate relationship, will be saved in the DMS as shown by Haug et al..

2.6 Ontology-driven Code Generation

Ontologies are the backbone of the DMS presented here. They are not only used to classify data and to connect data and information. As high-level domain models, they form the foundation to generate application-level classes automatically. The risk of inconsistencies between the ontologies and the

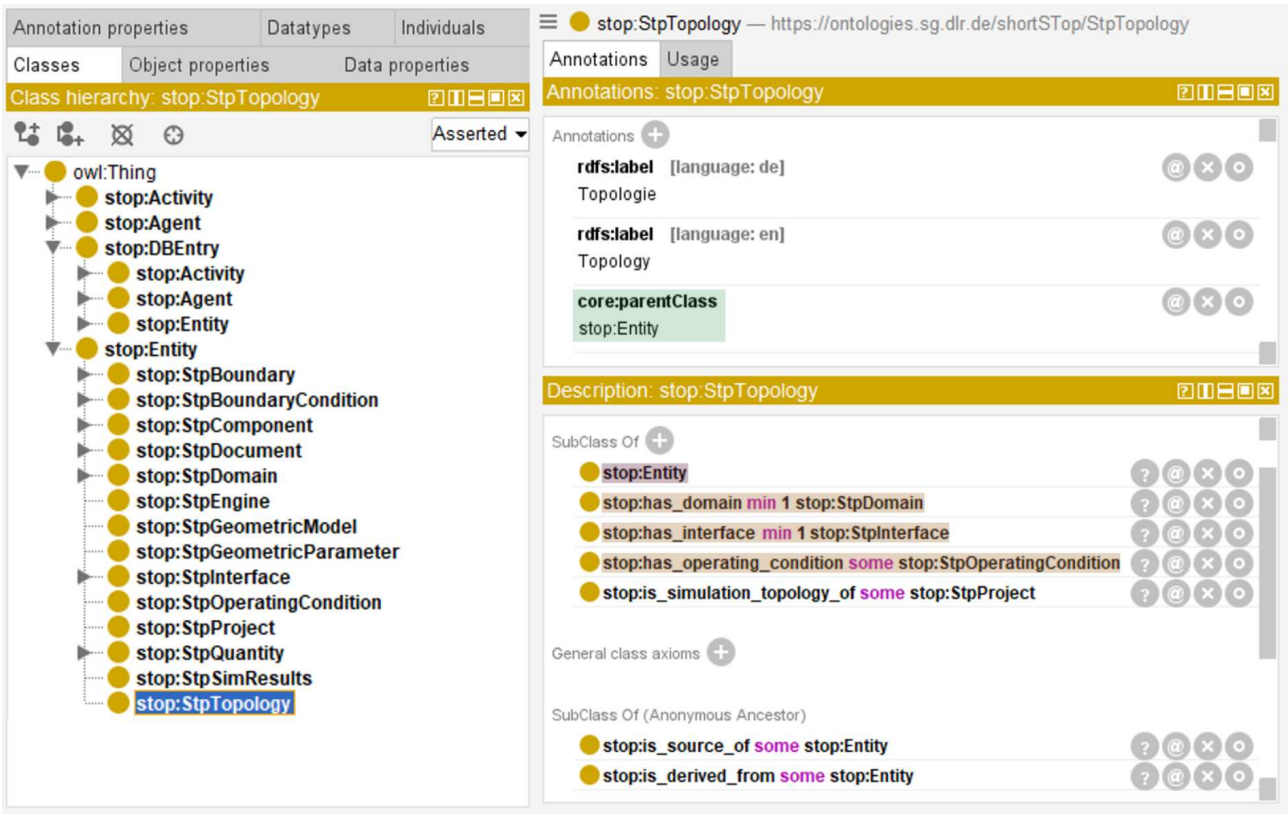


Figure 6 – Details of the entity *Topology* in protégé [18]

implementation is kept to a minimum, because the class definitions are derived directly from the web ontology language (OWL) representation. This prevents the unnecessary manual duplication of the ontology concepts in source code and provides a bridge between the semantic modeling and the software development.

The presented tool bridges OWL-based knowledge representation and object-oriented software development by generating programming-language classes from ontology definitions. In the OWL format, classes define the relevant types of entities in a domain, as described in Figure 5, for example. Object properties describe semantic relationships between instances of these classes following the subject–predicate–object pattern. For example, a simulation has results or the simulation topology is part of the simulation setup. The subject is one of the classes defined as the property's domain. The object belongs to the class or classes defined as its range. Data properties assign literal values to individuals, such as numerical parameters, for example. Annotation properties provide additional metadata for ontology elements, such as labels or comments. They are converted to class constants during the code generation. Given OWL ontologies, the tool reads the ontology classes, object properties and data properties and transforms them into an intermediate representation. This representation abstracts from OWL-specific serialization details and captures the information required for code generation, including names, IRI (Internationalized Resource Identifier [19]), domains and ranges of object properties as well as datatypes and annotations. To avoid confusion, the OWL definitions are written in bold in the following remarks.

Backend generators then translate the intermediate representation into target-language artifacts. As a prerequisite, the target architecture must provide base classes for entities and relationships, from which the automatically generated program classes are then derived. The current implementation supports Python and Qt, enabling ontology-derived structures to be used directly in application code and user-interface frameworks. This approach supports consistency between the ontology and the implementation, reduces manual modeling effort and provides a foundation for ontology-driven application development. The generated code is not intended to replace OWL reasoning. Rather, it provides a structural projection of the ontology into a programming environment. The most important OWL constructs can be represented in the target language without modification and are therefore

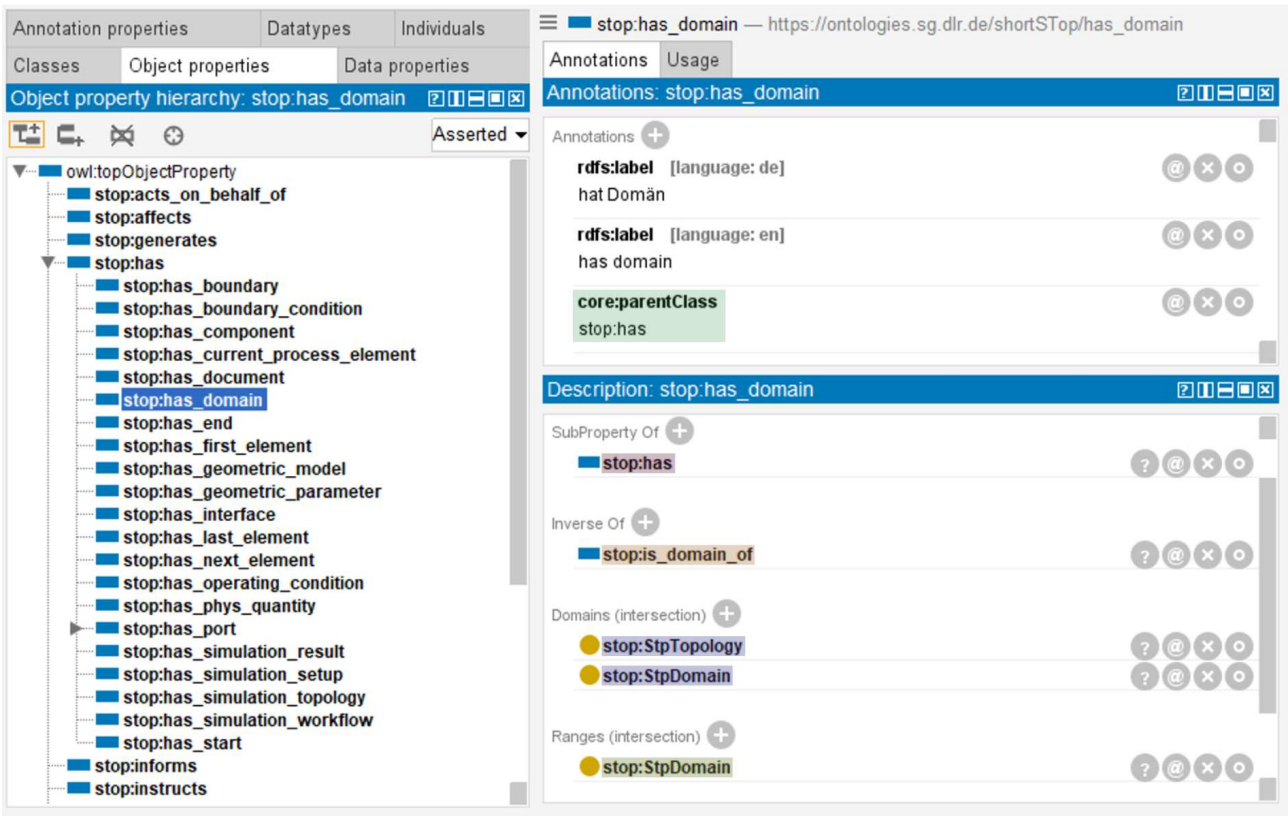


Figure 7 – Details of the object property *has_domain* in protégé [18]

translated directly into source code. More expressive axioms can be retained as metadata, comments, validation rules, or extension points. The program code, into which the auto-generated source code is to be integrated, must include base classes for entities and relationships, as these are not part of the auto-generated source code. For the use cases discussed later, a module was implemented for the software framework GTlab (Gas Turbine Laboratory [20]) that includes all necessary base classes and simple visualization options. GTlab enables multidisciplinary, collaborative research in the field of aircraft propulsion and also supports, for example, the design and simulation of individual components at various levels of detail throughout their entire life cycle.

For the code-generation process, a few additional settings have to be included. The most important assumption is the closed-world assumption, which means that only the defined relationships are allowed. In the left window of Figure 6, a hierarchical view of the entities, also referred to here as classes, is shown in the program protégé [18]. Entities can appear in multiple locations within the tree view. Consequently, their position in the tree is not necessarily unique. For example, the class *Entity* appears in two places, directly under the root element and under *DBEntry*. Therefore, it is necessary to introduce an annotation property that specifies a unique location for generated files in the source code. This property was named *parentClass*, marked in green in this figure. As class, *Topology* has been selected. The annotation properties of the entity are listed in the upper-right window. In the lower-right window, under **SubClass Of** all classes are listed on which the respective class depends. In this case, only one parent class is listed, highlighted in red. Additionally, all object properties are also listed in this window. The three *has* relationships shown in Figure 4 are listed here, marked in orange:

- `Topology -[:has_domain^+]-> Domain`
- `Topology -[:has_interface^+]-> Interface`
- `Topology -[:has_operating_condition^*]-> OperatingCondition`

Figure 7 shows a snapshot of the object properties. In contrast to the representation of the simulation topology in Figure 4, the *has* relationship is further subdivided in this OWL representation. As shown

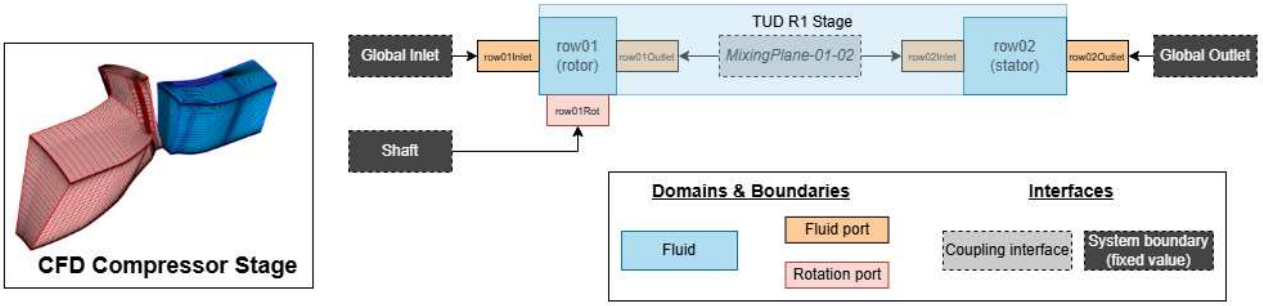


Figure 8 – Simulation topology of a single compressor stage (derived from [15])

in the left window of Figure 7, there are many specific variants of the *has* relationship. For example, the *has_domain* relationship is selected here. According to the description of the simulation topology, both a *topology* and a *domain* can have a *has* relationship with other *domains*. Consequently, this determines the valid entities for both sides of the relationship. For *has_domain*, the **domains** comprises *topology* and *domain*, marked in blue, while the **range(s)** comprises only *domain*, highlighted in olive. The program logic is structured such that every relationship also requires an inversion. In this way, the relationship can be addressed from both sides. Here the inverse relationship is *is_domain_of*, highlighted in orange. When constructing the program classes for the relationships, the annotation property *parentClass* is also used. While both sides of the relationship are defined in OWL, in the source code both sides are combined into a single relationship class. For this reason, the property is set only on one side, in this case *has_domain*. By specifying relationships specifically, the program code can more precisely define which classes can be linked together for each relationship and which classes are expected on the other side. The inheritance strategy still enables all *has* relationships to be collected at once.

3. Use Case - Simulation Topology

The first use case considers the simulation topology of a single compressor stage. Although comparatively compact, this example already illustrates the structural limitations of a purely hierarchical representation and the advantages of a graph-based approach. Figure 8 shows two representations of the simulation topology for a steady-state CFD simulation. The compressor stage is represented by one main domain, which is subdivided into two subdomains corresponding to the rotor and the stator. The topology further comprises three system boundaries: inlet, outlet and shaft. The rotor is connected to the inlet by its upstream fluid port, whereas the stator is connected to the outlet by its downstream fluid port. In addition, the rotor is connected to the shaft. The remaining fluid ports of rotor and stator are connected by an interface. For the steady-state case considered here, this interface is represented by a mixing plane.

In the initial GTlab implementation, the simulation topology was represented as a strict hierarchy. This implementation imposes the restriction that each object can be assigned to only one parent. The corresponding explorer view is shown in Figure 9a. Below the simulation topology, separate sections are provided for domains, interfaces, system boundaries, including inlet boundary conditions and operating points. The domains are themselves arranged hierarchically. Their associated boundaries and interfaces are listed below the respective domain. However, the interface between rotor and stator, highlighted in the figure as `MixingPlane-row01-row02`, naturally belongs to both domains. Since the tree-based representation does not allow one object to appear below multiple parents, the interface has to be assigned to a single parent domain. This illustrates a typical compromise that is required when inherently graph-like engineering data are forced into a fixed hierarchical structure.

The underlying graph representation is shown in Figure 10. The graph can be divided into six clusters. Cluster 1 comprises the simulation topology and the setup information for the simulations, whereas clusters 2 - 6 contain the results of five executed simulations, including folders and files. The dense interconnections within cluster 1 indicate that there is no unique hierarchical ordering of the data. In particular, simulation results are generated by process steps and are related to setup in-

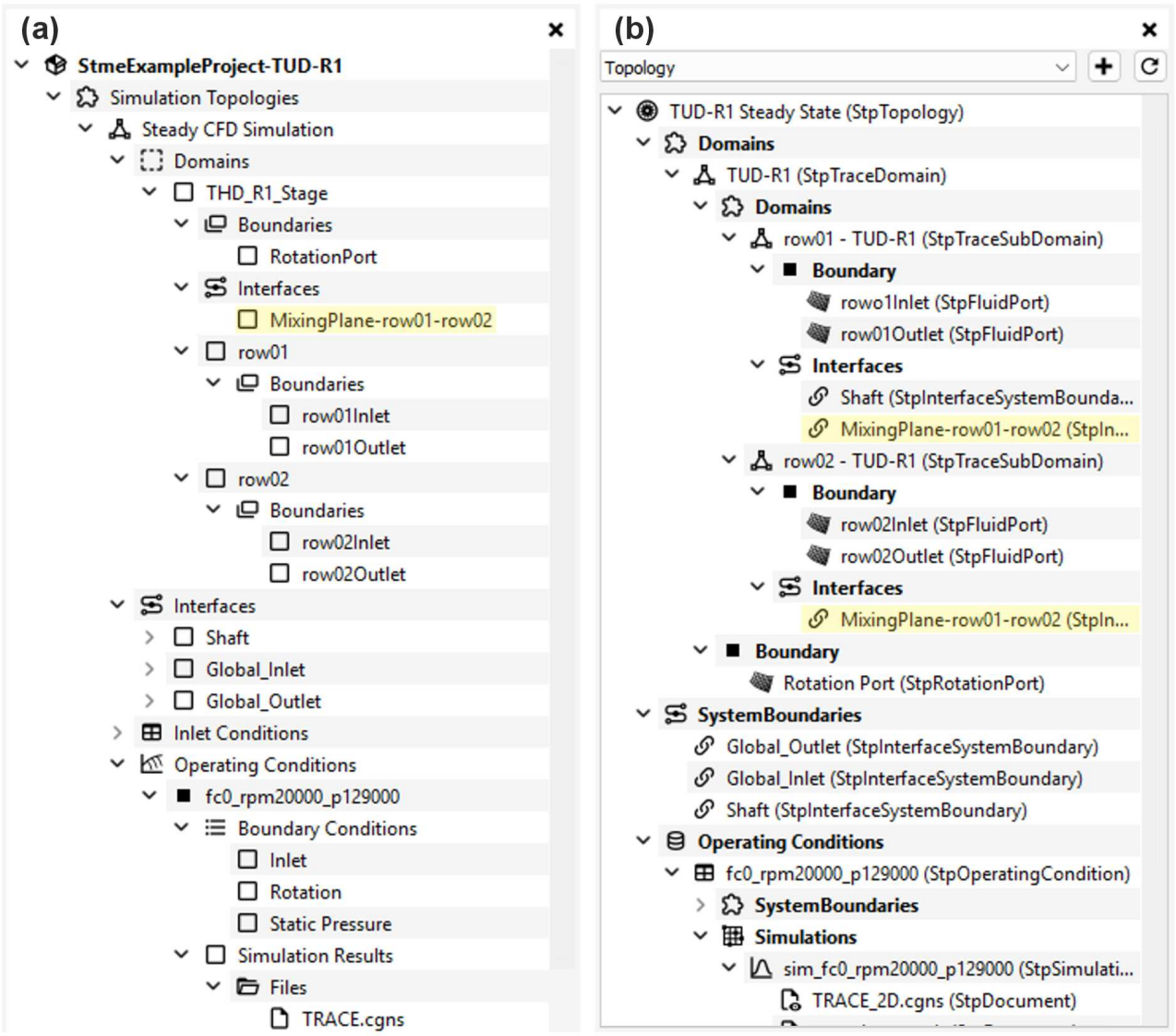


Figure 9 – Illustration of the simulation topology including simulation results: a) hierarchically organized approach, b) graph-based approach

formation, operating conditions and topology information at the same time. In a strict hierarchy, such relationships would have to be represented by additional references or duplicated information, which would increase the complexity of navigation and maintenance. In contrast, the graph representation allows such relationships to be encoded directly and can be extended by adding further nodes and relationships without changing a predefined global hierarchy.

A hierarchical view can nevertheless be derived from the graph by defining root classes and traversal paths from these roots to the elements that shall be displayed. For the topology-oriented perspective shown in Figure 9b, the simulation topology is used as the root element. Domains, system boundaries and operating conditions are displayed as subelements of the topology. For domains, the corresponding subdomains, boundaries and interfaces are shown. Most of these elements can be retrieved by direct relationships. System boundaries, however, may be connected either directly to the topology or indirectly through an arbitrary number of intermediate domains. The corresponding view definition must therefore allow paths of variable length.

The following path definitions illustrate the topology-oriented view:

- **Domains:** `Topology -[:has_domain]-> Domain`

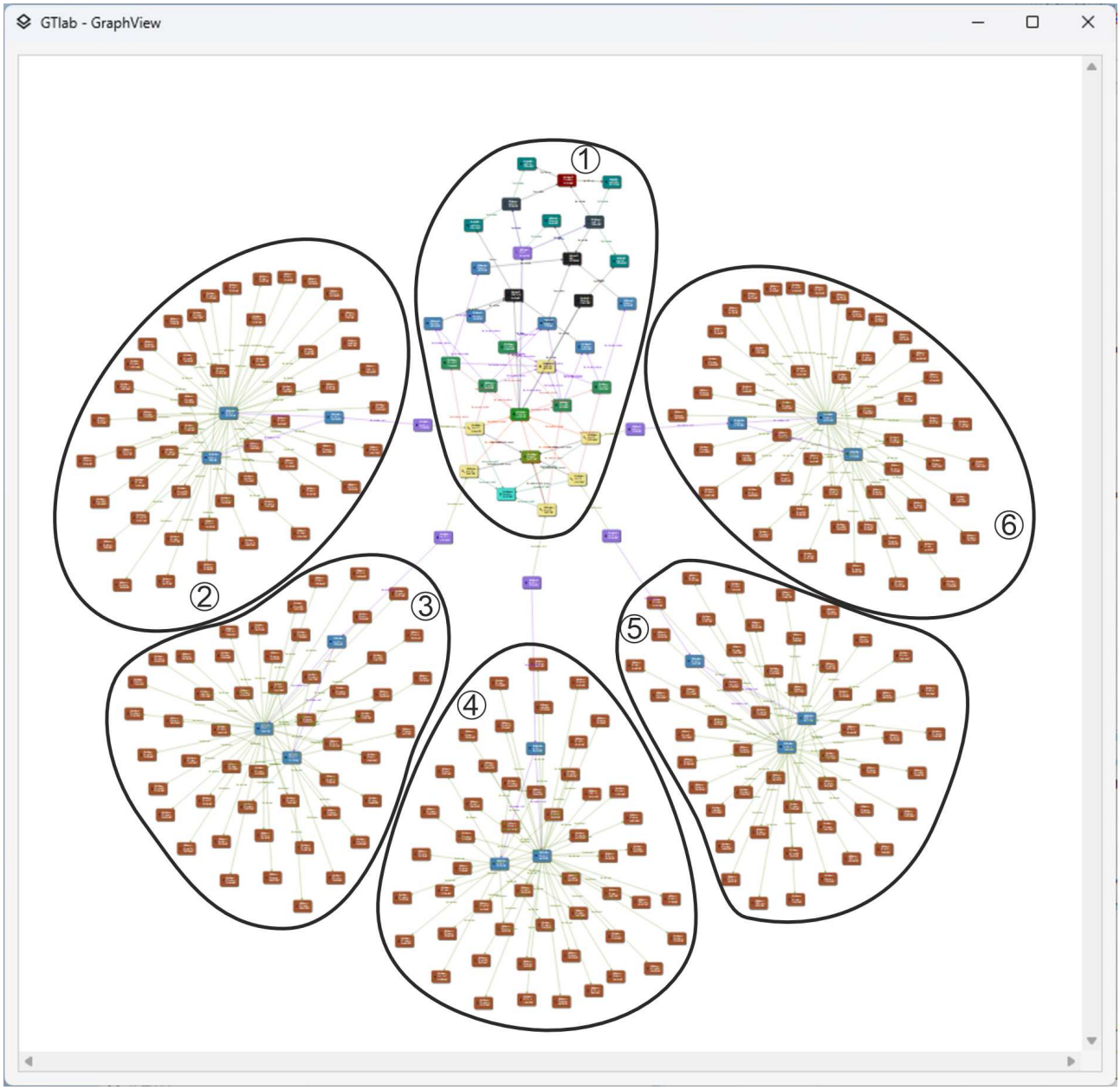


Figure 10 – Graph representation of the data used in the compressor-stage use case

  - *Domains:* `Domain -[:has_domain]-> Domain`
  - *Boundary:* `Domain -[:has_boundary]-> Boundary`
  - *Interfaces:* `Domain -[:has_interface]-> Interface`

- **System Boundaries:** `Topology -[:has_domain]-> Domain -[:has_interface]-> InterfaceSystemBoundary`

  - *System Boundaries:* `OperatingCondition -[:has_boundary_condition]-> BoundaryCondition`

- **Operating Conditions:** `Topology -[:has_operating_condition]-> OperatingCondition`

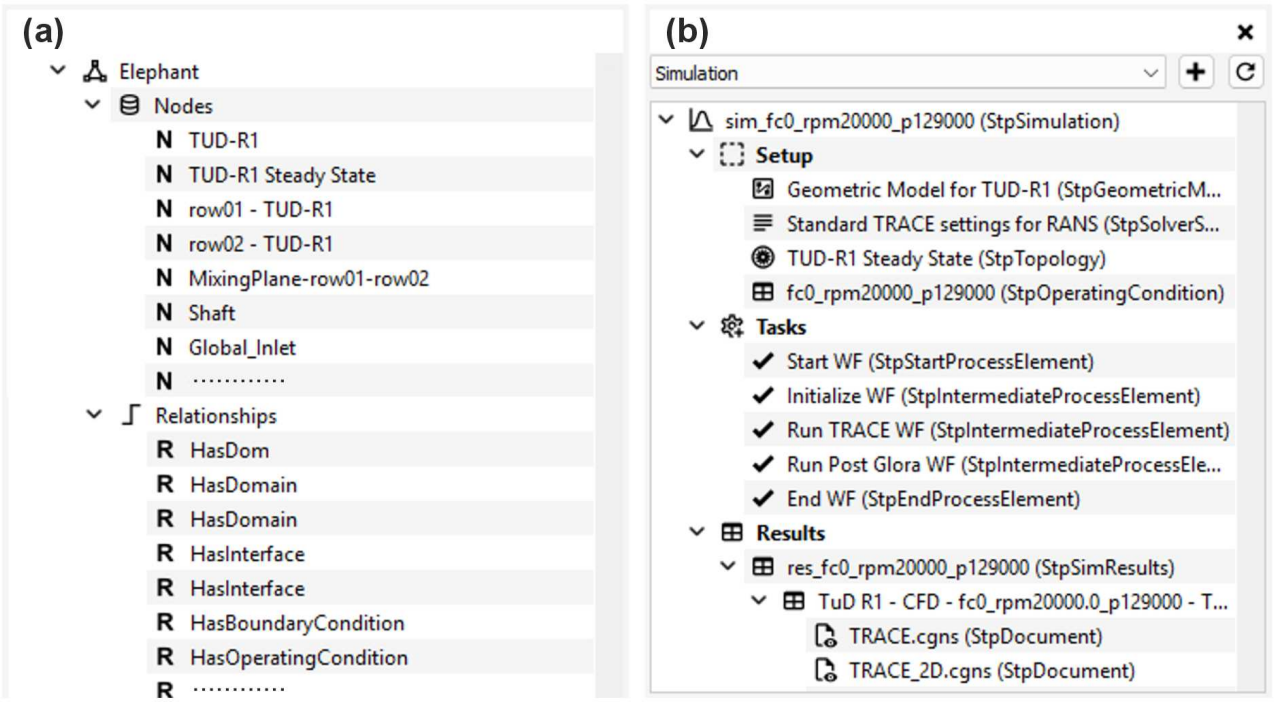


Figure 11 – Details of graph-based implementation: (a) simulation topology, (b) simulation

- *Setup:*
  - `OperatingCondition <-[:has_operating_condition]- SimSetup`
    `SimSetup -[:has_simulation_solver_settings]-> SolverSettings`
  - `OperatingCondition <-[:has_operating_condition]- SimSetup`
    `SimSetup -[:has_geometric_model]-> GeometricModel`
- *Results:*
  - `OperatingCondition <-[:has_operating_condition]- SimSetup`
    `SimSetup <-[:has_simulation_setup]- Simulation`
    `Simulation -[:has_simulation_result]-> SimResults`
    `SimResults -[:has_document]-> Document`

Compared with the original hierarchical representation, the graph-derived view avoids assigning shared elements to an artificial parent. The interface between rotor and stator can be shown below both domains that it connects, because the underlying graph stores the relationship independently of the displayed hierarchy. Similarly, operating conditions can be enriched with their corresponding system boundaries and related simulations without duplicating data. Only the grouping elements of the view have to be created specifically for the user interface; the displayed domain objects, simulation objects and result objects remain references to the corresponding graph elements.
The current graph-based implementation in GTlab still has to be embedded in the hierarchical storage structure of the software environment. For this reason, two technical groups are introduced: one for graph nodes and one for graph relationships, as shown in Figure 11a. Figure 11b shows an additional simulation-oriented perspective derived from the same underlying graph. In this view, each simulation is displayed together with its setup, executed process steps and generated results. The focus of this perspective is therefore not the topology itself, but the execution of simulations and the traceability of their results. Further relationships, such as the link between a result and the process step that generated it, can be added to the view definition if required.
This example therefore serves as a controlled case for separating the underlying graph representation from the hierarchical views presented to the user. It demonstrates that a graph-based data model can preserve relationships that are difficult to represent in a strict tree, while still allowing engineers to interact with the data through familiar hierarchical structures. At the same time, the

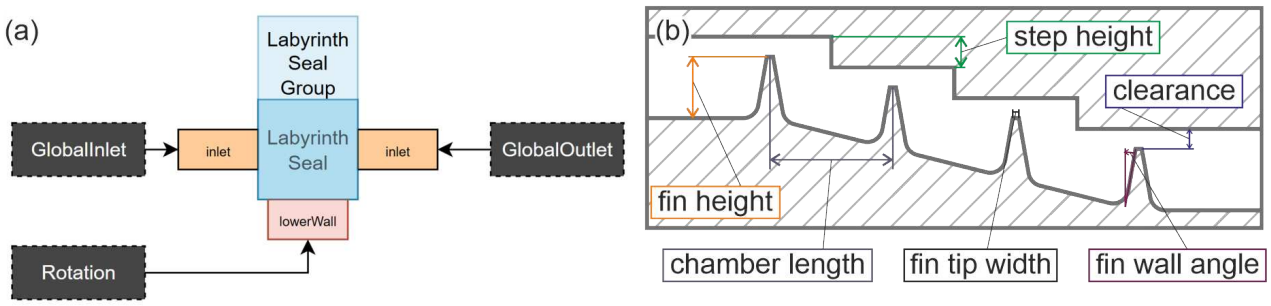


Figure 12 – Details of the stepped labyrinth seal: (a) simulation topology, (b) geometric parameters (derived from [21])

example remains limited in scope: the topology is fixed and the main variation is associated with the operating conditions and the resulting simulation data. In practical simulation campaigns, however, additional dimensions such as geometric variants, solver settings, workflow states and storage locations become equally relevant. The following use case therefore extends the discussion from a topology-centered view to a broader simulation-management scenario, in which several alternative perspectives are derived from the same underlying knowledge graph.

4. Use Case – Simulations

The second use case extends the previous example by considering a larger set of simulations that differ not only in their operating conditions, but also in their geometric models and solver settings. According to the simulation ontology introduced in Figure 5, a simulation setup is defined by four main components: the geometric model used for mesh generation, the selected simulation topology, the solver settings and the operating condition. Each of these components provides a possible criterion for organizing and exploring the resulting simulation data. While the first use case varied only the operating conditions, the present use case investigates how the same graph-based approach can support multiple alternative perspectives on a more heterogeneous simulation dataset.

The example is based on simulations of a labyrinth seal, whose topology and geometric parameters are shown in Figure 12. Labyrinth seals are important elements of the secondary air system of gas turbines, where they regulate internal airflows and reduce leakage [22]. The secondary air system supports several safety-critical functions, including component cooling, prevention of hot-gas ingestion, clearance control and bearing chamber sealing. Due to the strong interconnection of the secondary air system, local changes in a seal configuration may influence the behavior of the overall engine system. At the same time, the flow inside labyrinth seals is physically complex despite their comparatively simple geometry. Seal performance depends on several coupled parameters, including tooth geometry, surface condition, rotation, heat transfer and the swirl of the incoming flow [23].

Figure 12a shows the simulation topology used for the labyrinth-seal example. The lower wall of the computational domain is rotating, whereas the tip wall is stationary. The base geometry is derived from the convergent configuration with a smooth tip wall described by Denecke [21]. Figure 12b shows the geometric parameters that are varied in the simulations. In the example considered here, these parameters include the number of teeth (`not`), the step height (`step_height`) and the chamber length (`width`).

In total, 19 simulations are considered. The dataset includes three solver settings, several geometric models and multiple operating conditions. Figure 13 illustrates different graph-derived perspectives on this dataset. In Figure 13a, the simulations are organized according to the geometric model within a simplified engine data model. The different geometric variations of the labyrinth seal are associated with the secondary air system. This view demonstrates that a graph-based representation can embed detailed simulation models of a single component into a higher-level engine context, even if the surrounding engine model is not yet geometrically complete. The varied geometric parameters are attached directly to the corresponding geometry variations, which makes the differences between configurations explicit.

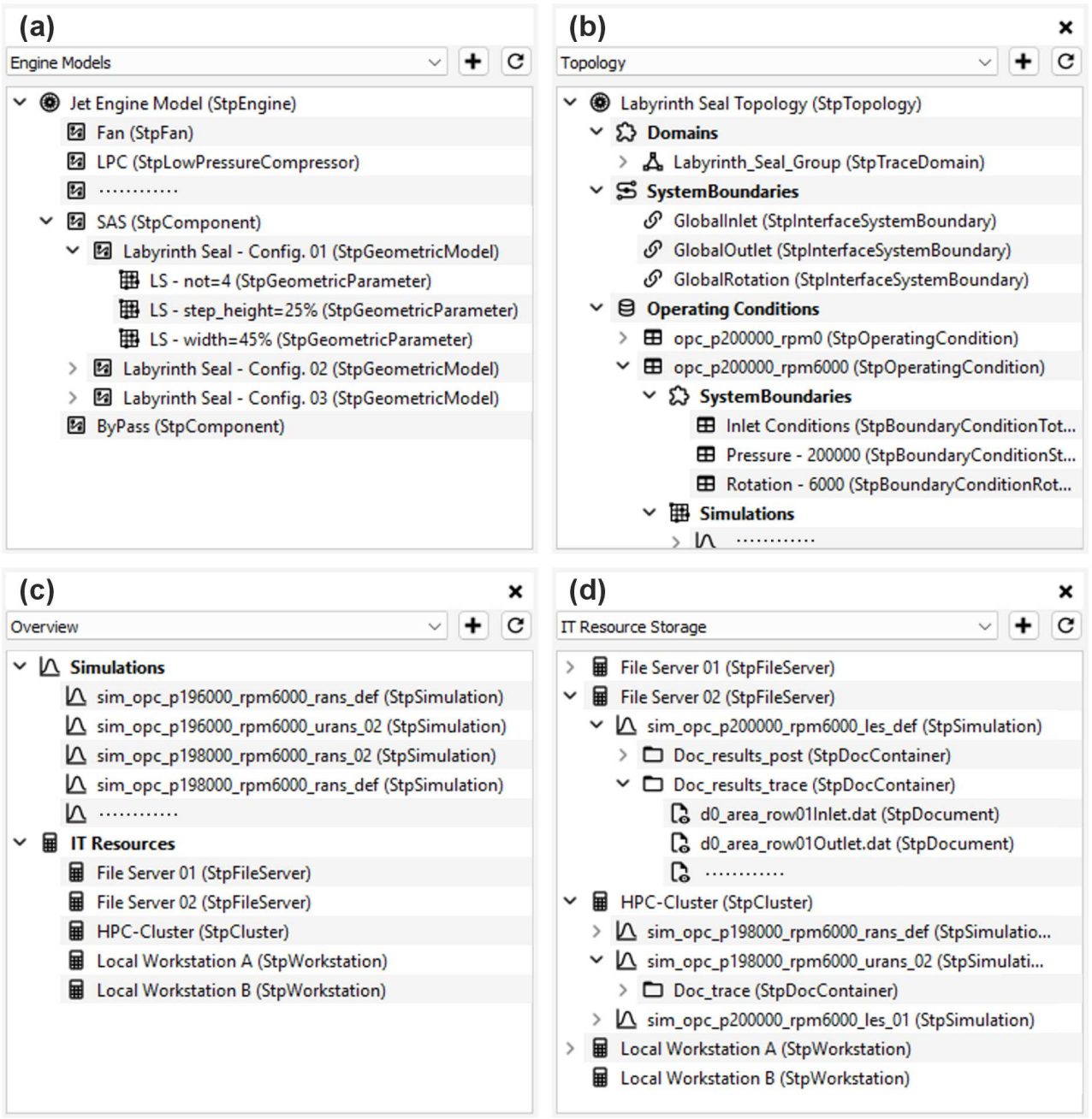


Figure 13 – Perspectives derived from the graph representation with different roots: (a) engine model, (b) simulation topology, (c) overview, (d) IT resource storage

Figure 13b shows a topology-oriented perspective similar to the one introduced in the first use case. In this view, the simulations are grouped by simulation topology and operating condition. Since the topology in Figure 12a contains three system boundaries, each operating condition must provide corresponding boundary conditions for these boundaries. The view therefore supports the analysis of which simulations were performed for a given topology and operating point.

A further perspective concerns the storage location of simulation data, as shown in Figure 13d. Executing simulations produces large amounts of data, which may be distributed across different storage systems. By linking simulation results and files to storage resources in the graph, it becomes possible to inspect where the data associated with a simulation are located. For example, run scripts and log files may remain on the HPC cluster, whereas larger result files may be stored on a dedicated file server. This perspective is particularly useful for assessing data distribution and storage load across the available infrastructure.

Simulations can also be organized according to solver settings. Figure 14a shows such a solver-

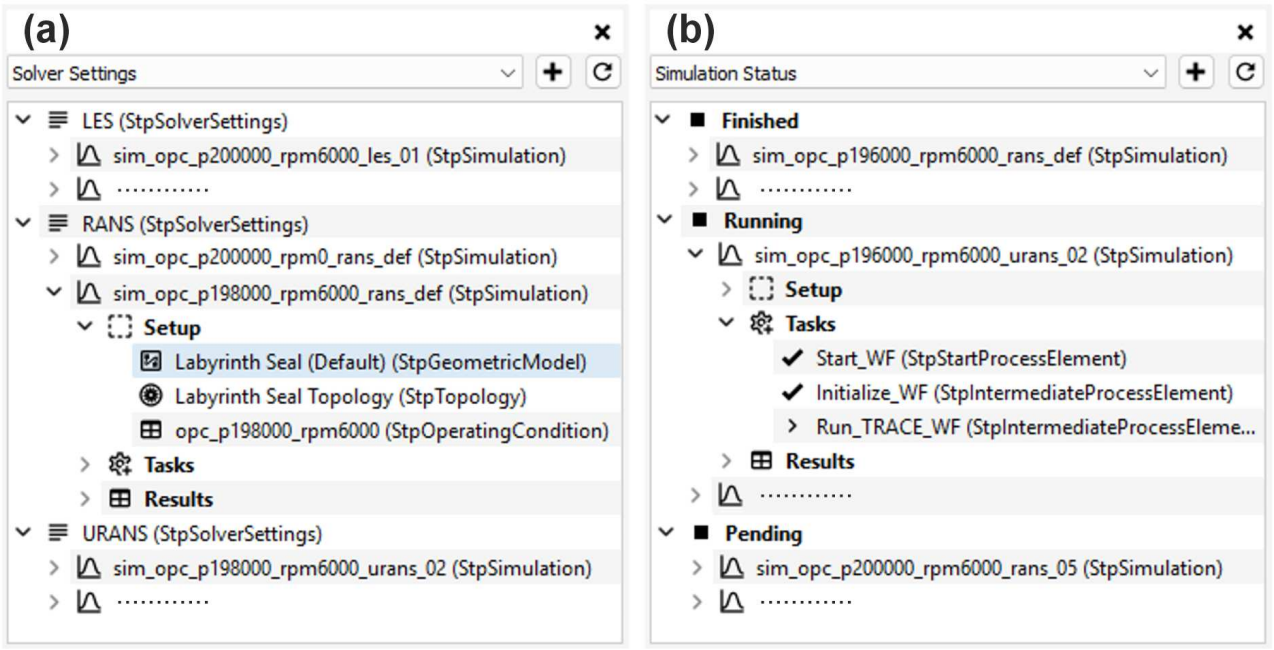


Figure 14 – Perspectives derived from the graph representation with different roots: (a) solver settings, (b) simulation status

oriented perspective. In the present example, three solver settings are used, each corresponding to a different level of accuracy. Higher-accuracy simulations typically require longer runtimes and generate larger amounts of data, but they may also serve as references for validating simulations with lower accuracy. The graph-derived view therefore provides a compact overview of which simulations were performed with which solver configuration.

In addition to organizing existing data, the graph can also be used to infer the status of simulations. For this purpose, a relationship is maintained that points to the currently active workflow element. As long as this relationship does not point to the end element of the workflow, the simulation is considered active. If no such relationship exists, the simulation has not yet been started. If the relationship points to the end element, the simulation is considered finished. Figure 14b shows the corresponding categorization into finished, running and pending simulations. The expanded simulation is currently located at its third workflow step, as indicated by the corresponding icon.

The complete graph stored in parDactylo for this use case is shown in Figure 15. Although certain structural patterns are visible, large parts of the graph are strongly interconnected. This confirms that a single fixed hierarchy would be insufficient to represent the dataset without extensive cross-references or duplicated information. The graph-based representation, by contrast, preserves the relationships between topology, geometry, solver settings, operating conditions, workflow state, result data and storage resources. Different hierarchical views can then be derived from the same underlying graph according to the task-specific information need of the engineer.

The second use case therefore extends the observations from the compressor-stage example to a more heterogeneous simulation-management scenario. Whereas the first use case focused on the representation of a single simulation topology, the second use case demonstrates that the same underlying graph can support several complementary organizational principles. Simulations can be explored with respect to geometric variants, operating conditions, solver settings, workflow states or storage locations without changing or duplicating the underlying data. This illustrates the central advantage of the proposed approach: the knowledge graph preserves the semantic relationships between engineering artifacts, while the derived hierarchical views adapt the presentation of these relationships to the task at hand. At the same time, the example also shows that the usefulness of such views depends on appropriate view definitions, consistent relationship modeling and sufficient metadata quality. These aspects are therefore important considerations for applying the approach to larger parts of the engine design process.

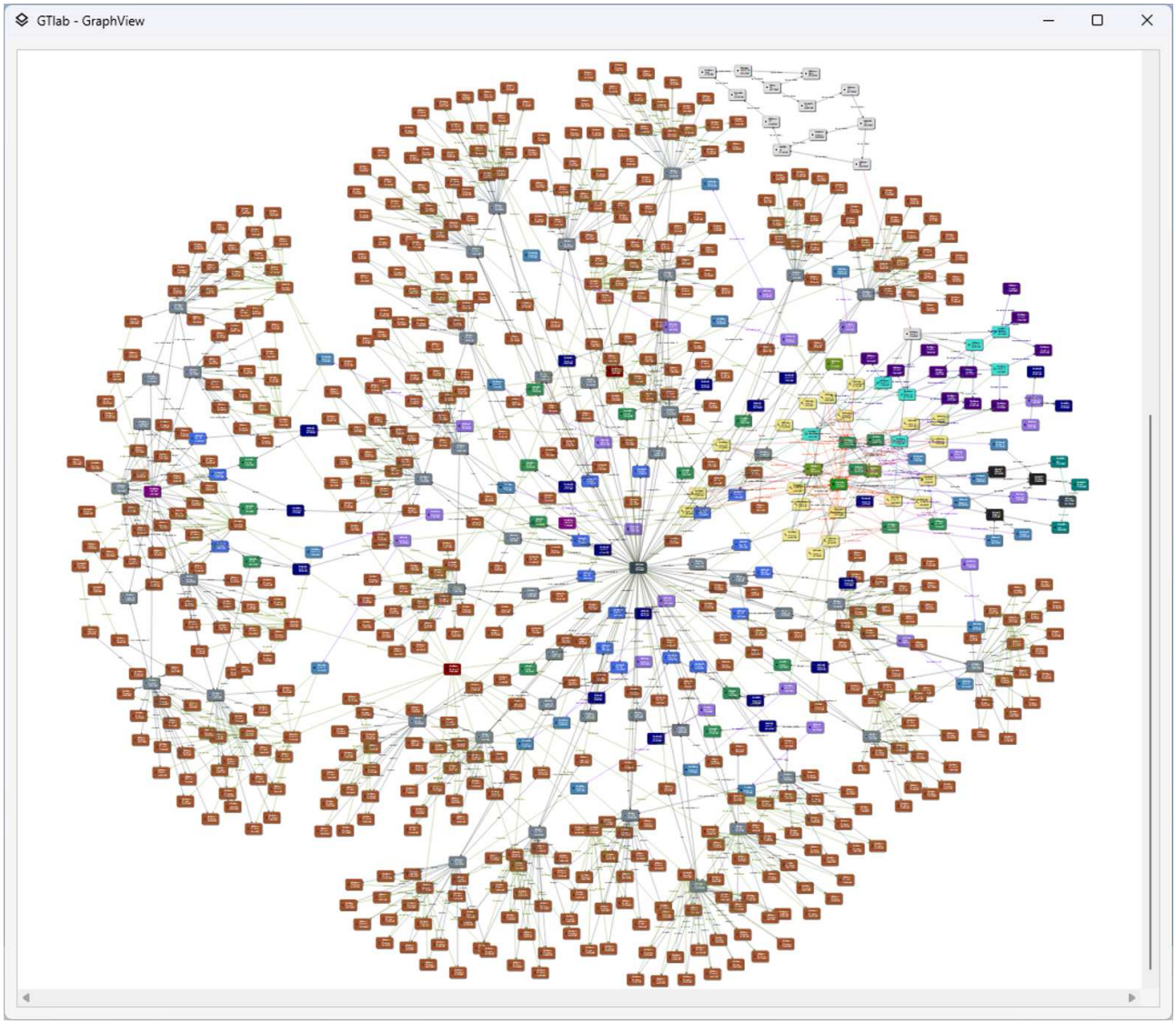


Figure 15 – Graph representation of the data used in the labyrinth-seal use case as stored in parDactylo

5. Conclusion and Outlook

This paper presented an ontology-based approach for deriving discipline-specific hierarchical views from a shared knowledge graph. The approach addresses a central limitation of fixed hierarchical data models: engineering artifacts are often related to several contexts at the same time and therefore cannot always be assigned to a single natural position in a tree. By storing the data and their semantic relationships in a graph and deriving task-specific views from this graph, the proposed approach enables engineers to work with familiar hierarchical representations without requiring a single global hierarchy.

The two use cases demonstrated this principle at different levels of complexity. The compressor-stage example showed how a graph-based representation avoids structural compromises for shared elements such as interfaces, while still supporting a topology-oriented hierarchical view. The labyrinth-seal example extended this idea to a broader simulation campaign and showed that the same dataset can be organized according to geometry, topology, operating condition, solver settings, simulation status and storage location. Together, these examples indicate that graph-derived views can improve the accessibility of complex engineering data while preserving the semantic consistency of the underlying model.

A further contribution of the presented approach is the integration of ontology-driven code generation. By deriving application-level classes from OWL-based ontology definitions, the implementation

can remain closely aligned with the semantic model. This reduces manual duplication of modeling concepts and supports the integration of ontology-based data structures into engineering software environments such as GTlab. The generated classes are not intended to replace ontology reasoning, but they provide a practical projection of ontology concepts into the application layer.
The current implementation also has limitations. The definition of discipline-specific views is still largely configuration-driven and requires knowledge of the underlying ontology and relationship structure. Future work should therefore investigate methods for supporting users in defining, validating and adapting such views. Further work is also required to assess the scalability of the approach for larger industrial knowledge graphs, to evaluate query performance and to study how engineers interact with the generated views in practical design tasks.
In the longer term, the combination of knowledge graphs and AI-based methods offers additional potential. AI agents may support the identification of missing information, suggest links between existing data or trigger processes that generate missing artifacts. However, in safety-critical engineering domains such as aero engine design, such extensions must remain traceable, verifiable and under human responsibility. For this reason, future developments should focus not only on autonomous graph enrichment, but also on provenance tracking, validation mechanisms and transparent user-facing views. The concept of deriving multiple task-specific views from a shared knowledge graph provides a promising foundation for keeping complex engineering data searchable, manageable and understandable throughout the design process and across the product lifecycle.

6. Contact Author Email Address

Kai Becker: kai.becker@dlr.de
German Aerospace Center (DLR), Institute of Test and Simulation for Gas Turbines, Augsburg, Germany

7. Copyright Statement

The authors confirm that they and/or their company or organization, hold copyright on all of the original material included in this paper. The authors also confirm that they have obtained permission, from the copyright holder of any third party material included in this paper, to publish it as part of their paper. The authors confirm that they give permission, or have obtained permission from the copyright holder of this paper, for the publication and distribution of this paper as part of the ICAS proceedings or as individual off-prints from the proceedings.

References

[1] Thomas Robert Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993. doi: 10.1006/knac.1993.1008.

[2] Mike Uschold and Michael Gruninger. Ontologies: principles, methods and applications. *The Knowledge Engineering Review*, 11(2):93–136, 1996. doi: 10.1017/S0269888900007797.

[3] Fatima Zahra Amara, Meriem Djezzar, Mounir Hemam, Sanju Tiwari, and Mohamed Madani Hafidi. Unlocking the Power of Semantic Interoperability in Industry 4.0: A Comprehensive Overview. In Fernando Ortiz-Rodriguez, Boris Villazón-Terrazas, Sanju Tiwari, and Carlos Bobed, editors, *Knowledge Graphs and Semantic Web*, pages 82–96, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-47745-4.

[4] I. O. Sanya and Essam Shehab. A framework for developing engineering design ontologies within the aerospace industry. *International Journal of Production Research*, 53(8):2383–2409, 2015. doi: 10.1080/00207543.2014.965352. URL https://doi.org/10.1080/00207543.2014.965352.

[5] Yassine Ghanjaoui, Jasper Bussemaker, Jörn Biedermann, and Björn Nagel. An Ontology-based Approach for the Co-Development and Optimization of Aircraft Cabin Design and Assembly Architectures. In *34th Congress of the International Council of the Aeronautical Sciences, ICAS 2024*, September 2024. URL https://elib.dlr.de/207156/.

[6] Jacopo Zamboni, Arthur Zamfir, Erwin Moerland, and Björn Nagel. A semantic knowledge based engineering framework for the rapid generation of novel air vehicle configurations. In *33rd Congress of the International Council of the Aeronautical Sciences, ICAS 2022*, September 2022. URL https://elib.dlr.de/192117/.

[7] Ian R. Grosse, John M. Milton-Benoit, and Jack C. Wileden. Ontologies for supporting engineering analysis models. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 19(1):1–18, 2005. doi: 10.1017/S0890060405050018.

[8] Michael A. Rosenman and John S. Gero. Modelling multiple views of design objects in a collaborative cad environment. *Computer-Aided Design*, 28(3):193–205, 1996. ISSN 0010-4485. doi: 10.1016/0010-4485(96)86822-9. URL https://www.sciencedirect.com/science/article/pii/0010448596868229. Artificial Intelligence in Computer-Aided Design.

[9] Wolfgang Mayer, Arndt Mühlenfeld, and Markus Stumptner. Using Ontologies to Optimise Design-Driven Development Processes. In Richard Curran, Shuo-Yan Chou, and Amy Trappey, editors, *Collaborative Product and Service Life Cycle Management for a Sustainable World*, pages 451–459, London, 2008. Springer London. ISBN 978-1-84800-972-1.

[10] Stanislaus Reitenbach, Alexander Krumme, Thomas Behrendt, Markus Schnös, Thomas Schmidt, Sandrine Hönig, Robert Mischke, and Erwin Mörland. Design and Application of a Multidisciplinary Predesign Process for Novel Engine Concepts. *Journal of Engineering for Gas Turbines and Power*, 141(1):011017, 09 2018. ISSN 0742-4795. doi: 10.1115/1.4040750. URL https://doi.org/10.1115/1.4040750.

[11] Jakob P. Haug, Kai Becker, Matthias Schuff, Martin Siggel, and Stanislaus Reitenbach. Data Management in a Collaborative Design Architecture, Part A: The Basic Concept of the Underlying Data Management System. In *AIAA Scitech 2025 Forum*, 2025. doi: 10.2514/6.2025-1370. URL https://arc.aiaa.org/doi/10.2514/6.2025-1370.

[12] Martin Siggel, Stanislaus Reitenbach, Marius Bröcker, Matthias Schuff, and Jakob P. Haug. Data Management in a Collaborative Design Architecture, Part B: Remote processing and analysis of simulation data. In *AIAA Scitech 2025 Forum*, 2025. doi: 10.2514/6.2025-1371. URL https://arc.aiaa.org/doi/10.2514/6.2025-1372.

[13] Timothy Lebo, Satya S. Sahoo, Deborah L. McGuinness, Paul Groth, Khalid Belhajjame, James Cheney, David Corsar, José Manuel Gómez-Pérez, Stian Soiland-Reyes, and Stephan Zednik. PROV-O: The PROV Ontology. W3c recommendation, World Wide Web Consortium (W3C), 2013. URL https://www.w3.org/TR/2013/REC-prov-o-20130430/.

[14] Matthias Schuff. Towards Multi-Fidelity Simulations in the Context of the Virtual Engine. In *Deutscher Luft- und Raumfahrtkongress 2023, Stuttgart*, September 2023. doi: 10.25967/610297. URL https://elib.dlr.de/200756/.

[15] Matthias Schuff, Jakob P. Haug, Kai Becker, Martin Siggel, Jens Schmeink, and Stanislaus Reitenbach. Data Management in a Collaborative Design Architecture, Part C: Simulation Topology. In *AIAA Scitech 2025 Forum*, 2025. doi: 10.2514/6.2025-1372. URL https://arc.aiaa.org/doi/10.2514/6.2025-1372.

[16] Martin Siggel, Stanislaus Reitenbach, Mladen Banovic, and Andreas Pahs. Closing the gap between data model driven geometry generation and hifi cad for automatized and consistent design analysis. In *AIAA SciTech 2024 Forum*, Januar 2024. ISBN 978-162410711-5. doi: 10.2514/6.2024-2897. URL https://elib.dlr.de/209849/.

[17] Layna Fischer, editor. *BPMN 2.0 Handbook: Second Edition: Updated and Expanded: Methods, Concepts, Case Studies and Standards in Business Process Modeling Notation (BPMN)*. Future Strategies Inc., Lighthouse Point, Florida, 2 edition, 2012. ISBN 978-0-9849764-1-6.

[18] Mark A. Musen. The protégé project: a look back and a look forward. *AI Matters*, 1(4):4–12, 2015. doi: 10.1145/2757001.2757003. URL https://doi.org/10.1145/2757001.2757003.

[19] Martin J. Duerst and Michel Suignard. Internationalized Resource Identifiers (IRIs). RFC 3987, Internet Engineering Task Force, January 2005. URL https://www.rfc-editor.org/rfc/rfc3987.

[20] Stanislaus Reitenbach, Jens Schmeink, Marius Bröcker, Marvin Nöthen, and Martin Siggel. GTlab, April 2025. URL https://zenodo.org/records/15129162. Software.

[21] Jens Denecke. *Rotierende Labyrinthdichtungen mit Honigwabenanstreifbelägen - Untersuchung der Wechselwirkung von Durchflussverhalten, Drallverlauf und Totaltemperaturänderung*. PhD thesis, Karlsruhe Institute of Technology, 2008.

[22] Raymond E. Chupp, Robert C. Hendricks, Scott B. Lattime, and Bruce M. Steinetz. Sealing in Turbomachinery. *Journal of Propulsion and Power*, 22(2):313–349, 2006. doi: 10.2514/1.17778. URL https://doi.org/10.2514/1.17778.

[23] Adele Nasti, Ivan I. Voutchkov, David J. J. Toal, and Andrew J. Keane. Multi-fidelity simulation for secondary air system seal design in aero engines. In *Proceedings of the ASME Turbo Expo 2022: Turbomachinery Technical Conference and Exposition*. ASME, 2022. doi: 10.1115/GT2022-80391.