33RD CONGRESS
OF THE INTERNATIONAL COUNCIL
OF THE AERONAUTICAL SCIENCES
STOCKHOLM, SWEDEN, 4-9 SEPTEMBER, 2022

ICAS 2022 SWEDEN

A SEMANTIC KNOWLEDGE BASED ENGINEERING FRAMEWORK FOR THE RAPID GENERATION OF NOVEL AIR VEHICLE CONFIGURATIONS

Jacopo Zamboni[1], Arthur Zamfir[1], Erwin Moerland[1] & Björn Nagel[1]

[1]German Aerospace Center (DLR), Institute of System Architectures in Aeronautics, Hamburg, Germany

Abstract

The demands posed on modern aircraft design encourages research into new tools and methodologies that allow the rapid generation of digital prototypes which can represent fundamentally different configurations. High-level design decisions are generally made within the conceptual design phase and require a solid understanding of the technology impacts on the overall aircraft system. This paper presents a recently developed, novel knowledge-based engineering framework, which allows for the transparent digitization of conceptual design knowledge and its interconnection using semantic-web technologies. It focuses on describing an effective structure for capturing and storing the expert's knowledge and on the procedures required to analyse and solve the resulting complex system. By flexibly recombining the knowledge patterns concerning e.g. different propulsion architectures or lifting-surface configurations, the resulting system provides a significantly reduced effort to generate fundamentally different air vehicle configuration prototypes and allows for covering the vast design space typically considered during the conceptual design phase. The discussion is supported by examples for the automated creation of multi-trapezoidal lifting surfaces and the creation of different combat air vehicles architectures starting from the same knowledge base.

Keywords: Knowledge-Based-Engineering, Semantic Web Technologies, Conceptual design, Collaborative engineering

Nomenclature

Abbreviations

Codex	Collaborative Design & Exploration
CPACS	Common Parametric Aircraft Configuration Schema
DLR	German Aerospace Center
DOE	Design Of Experiments
DSL	Domain Specific Language
KBE	Knowledge Based Engineering
OOP	Object Oriented Programming
OWL	Web Ontology Language
RDF	Resource Description Framework
SWT	Semantic Web Technologies
SysML	Systems Modeling Language
UCAV	Unmanned Combat Air Vehicle
UML	Unified-Modeling-Language

Basic modeling concept

Ontology	A set of concepts and categories in a domain showing the properties and the in-between relations
Query	A formal question requesting information. When applied to the context of graphs, it attempts to identify an explicit pattern within the graph
Triple	A statement made of a subject, a predicate and an object. Forms the base unit of knowledge in SWT modeling

1. Introduction

To cope with the demanding environmental and defense challenges currently faced within the air transport system, there is a growing need for methods to effectively analyse the effect of introducing novel technologies and configurations on an overall system level. To enable the integration of the available knowledge concerning promising future technologies, the collaboration between a multitude of engineers - generally having heterogeneous backgrounds and ways of working - is required. In support of this, the continued digitization of the aircraft design procedure allows the consideration of a larger number of disciplines and technologies already early in the design process. While enabling more in-depth analyses, the complexity of the aircraft conceptual design phase however increases. Several tools have been created in the past decade in an effort to automate and streamline the conceptual and preliminary phases of the aircraft design. Among these, the DLR-developed Knowledge-Based Engineering (KBE) tools VAMPzero [1] and VAMPzeroF [2] for the initiation and synthesis of transport and military aircraft models respectively, have been developed and successfully used within a series of air vehicle design projects [3, 4, 5]. Both implementations have shown how a flexible design tool can improve the integration of a multitude of disciplinary domains in the design process. The matured experience in the application of these kinds of tools, combined with the necessity for improved integration between the several disciplines that characterise the aircraft design process, has however highlighted a series of improvements that would allow for an even more flexible usage of the incorporated digital knowledge-bases, ultimately enabling more thorough searches for promising solutions within the vast design spaces considered. Transferring from a frame-based to a non-frame-based modelling approach is one of the most promising improvements under investigation. Most KBE-tools use a so-called "frame-based" [6] language for modeling, such as python, C#, or UML for ParaPy, PaceLab APD and the Design Cockpit 43 respectively. These "frames" are often called *classes* in these languages and are usually structured in hierarchical parent-child relationships, i.e. a class can have multiple child-classes but just one parent-class. Classes are declared as part of the *meta-model* and are used to create new instances in the *model*, which will reflect the shape/type of that class and cannot take any other shape once instantiated. This approach to classification and instantiation makes it straightforward to create and reason using the model within the context of a specific domain, since the instances are structured in a way that reflects the real-world knowledge of that domain domain, e.g. an A320 is a type of aircraft which is a sub-class of vehicle. While this approach to creating a model is intuitive from the perspective of a *single domain*, it can complicate modeling and data-integration in a *collaborative multi-domain* environment. For example, an engine can be seen as a "thrust-producer" from the propulsion-domain and as an "electric-energy-producer" from the systems-domain. Integrating both views in a single model would require both domain-experts to agree on a single class that covers both perspectives. A not strictly frame-based modeling approach, as presented in this paper, where instances can change their apparent classes depending on the specific domain-perspective taken, could provide an improved collaborative KBE-experience in a multi-domain environment - while still allowing for an effective way of capturing domain-specific knowledge.

The Codex Framework [7] aims to provide a non-frame-based environment for knowledge capture, formalization and execution in which knowledge from several competence specialists can be collaboratively added and integrated. The ability to quickly adapt or re-use the implemented knowledge-bases results in a model of the aircraft system that can then be tailored to the external requirements and designers inputs, ultimately allowing for quick design space explorations and comparisons between several technologies. Using this capability, the basis for decision-making in early design stages can be significantly improved, as discussed in [8]. Using the Codex Framework, a semantic KBE application for the conceptual design of combat aircraft is being created. The resulting application Codex-FighterDesign serves as candidate to succeed VAMPzeroF.

This paper starts by elaborating the structuring of knowledge for effective multi-domain collaboration in section 2. Based on this, the Codex Framework is introduced in section 3. including its architecture and how it enables the creation of semantic KBE tools by combining ontologies and sets of parametric and topological rules. Two KBE tool implementations created using the framework are described in section 4. The first describes the creation of a multi-trapezoidal wing and shows how the resulting knowledge-base can be used to create both combat aircraft and wind turbine geometrical models. On the basis of combat aircraft implementations of conventional fighter jet, UCAV and a flying wing UCAV, the second implementation shows the ease in which architectural changes for quickly exploring design spaces can be implemented using the modelling approach. The paper ends with a conclusion and outlook in section 5.

2. Effective multi-domain knowledge modeling

This section provides the theoretical background concerning methods for knowledge modeling, whilst having its integration in a collaborative engineering context in mind. It starts by providing an overview of modeling using Knowledge-Based Engineering (KBE) techniques and how the Codex framework leverages these. Thereafter, a promising method for structuring knowledge for effective collaboration is introduced.

Using KBE techniques for multi-domain knowledge modeling

The core task of any KBE tool is to provide an effective means of capturing knowledge, i.e. creating the domain-specific ontologies and models. Many KBE tools leverage the already existing grammar of a specific programming language for this task, such as Python or C# for ParaPy or PaceLab APD respectively. While object-oriented (frame-based) programming languages do provide an unambiguous formal language for capturing domain knowledge, they also require that the domain-expert is either proficiently skilled in the language, or (s)he needs to team-up with a knowledge engineer to effectively model the knowledge representing his/her domain. Finally, the grammar of programming languages is not always guaranteed to remain the same over time as new non backward-compatible versions can be released. On the long run, this might lead to high maintenance cost, incompatibility of the KBE application or to missing out on useful features or libraries which are only available for the newest versions of the language.
To cope with this, a more sustainable approach is to use a modeling language that is independent from any programming language and is purely focused on the formalization of knowledge. An example for this is the *Design Cockpit 43* [9], which uses the Unified-Modeling-Language (UML) [10] to digitise domain knowledge. While UML is mostly used in the context of defining the structure and behavior of software, it is successfully utilised in the *Design Cockpit 43* as a generic language to model any kind of domain entities and their relationships to each other, i.e. the ontology of a domain. The frame-based nature of UML models makes these easy to create and understand, however this language lacks certain fundamental aspects of non-frame-based (i.e. graph-based) languages that make them uniquely suitable for multi-domain collaboration and knowledge integration.
The Codex Framework presented in section 3.of this paper leverages Semantic-Web-Technologies (SWT) [11] for capturing domain knowledge in a standardised way. Like for many other standardised web-technologies, SWT is based on the assumption of being able to integrate all exiting information in a collaborative and decentralised network, making it an excellent fit for the intentions of the Codex Framework. Using SWT completely decouples the captured knowledge from the specific implementation required to execute it, which makes it easier to maintain as well as easier to share and integrate with other applications. This however might come at the cost of added complexity in the development and implementation of the underlying framework and methodology. To enable coupling all different kinds of knowledge, often stemming from different sources and representing heterogeneous domains, the language used in SWT is relatively abstract and generic. It is not tailored to the creation of engineering applications and might show reduced performance and ease of use compared to the frame-based languages used by other KBE frameworks. Codex intends to cope with this by providing an environment in which the complexity is being reduced to a minimum, allowing its users to leverage the more flexible modeling language without having to give up on the conveniences typical of modern KBE frameworks.

Structuring Knowledge for effective collaboration

The way the human mind understands its environment is by grouping entities into categories, also known as frames or classes. Based on common characteristics of these entities, this categorization is usually done subconsciously. It helps to make quick decisions based on the learned categories of an

entity. For example, both a tiger and an active volcano are categorised as dangerous and therefore we intuitively avoid it. This categorization is context-dependent and in a different context a completely different categorization scheme for the same entities can be created: a tiger is an organism and a volcano a geological feature.
In essence, knowledge can be effectively structured in categorical hierarchies that are context-dependent, i.e. each context has a different hierarchy of categories that may contain some of the same entities. Visually, these hierarchies can be imagined as layers of domain specific knowledge that, when taken collectively, will result in a multi-domain knowledge graph. A visualisation of this idea is shown in figure 1.

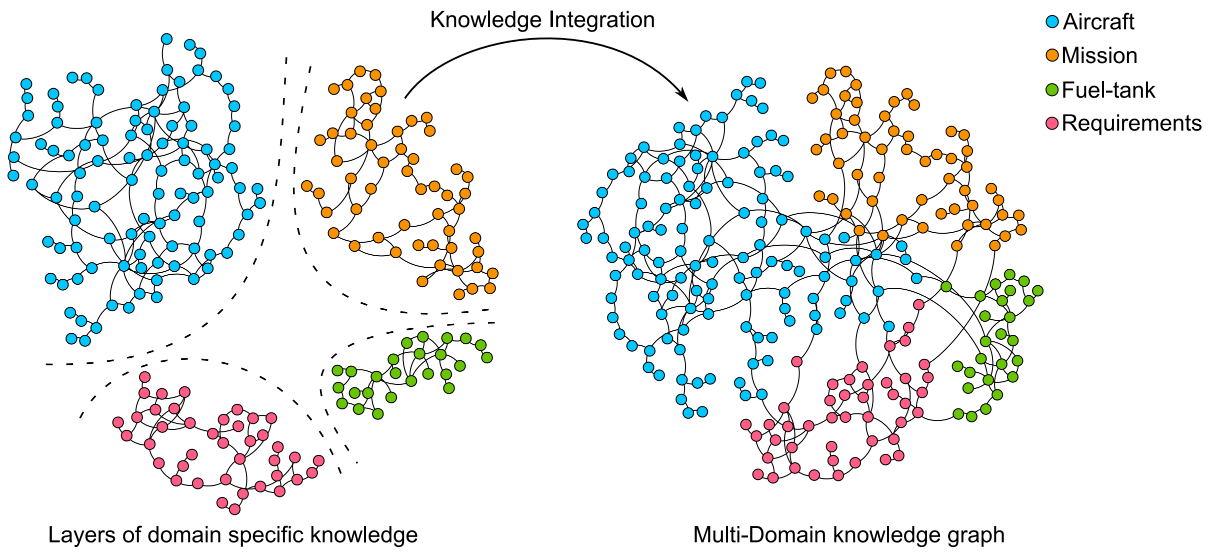


Figure 1 – Domain specific knowledge integrated into multi-domain knowledge graph [7].

The capability to model knowledge for a specific domain is generally well covered by the application of dedicated domain-specific tools (such as finite element solvers for structural modeling or a KBE tool for mass estimations, etc.). Enabling multi-domain knowledge modeling is usually approached by connecting the domain specific tools through ad-hoc connections of the corresponding input and output data or - to reduce the amount of interfaces and create a common language - through usage of a central data exchange format. The Common Parametric Aircraft Configuration Schema (CPACS), initiated by DLR in 2005 and currently developed and applied by an increasing community of aerospace engineers, is an example of a successfully applied schema for parametric air vehicle design [12]. It provides a hierarchical structure in which the individual components of an air vehicle can be described and combined into overall vehicle concepts and allows for the exchange of analysis results of the tools involved in the process. This *schema-based* approach is useful for creating a common language among the involved specialists and enables tool-integration in simulation workflows for collaborative simulation tasks [13]. The development of a schema does however require all participants to agree on a location and hierarchy for their common entities, which can lead to time-consuming schema management efforts.
As an alternative to schema-based multi-domain collaboration, a *schema-less* approach can be used, which closer mimics the aforementioned knowledge graph that can support multiple contexts at the same time [7]. A model that has been created using a schema-less modeling approach can be constrained to represent a very domain-specific view [14], i.e. schema-less modeling can still have the advantages of schema-based modeling while having the potential to allow for more flexible multi-domain collaboration.Another application where schema-less might be beneficial is within early design stages, typically featuring large design spaces and a large amount of configuration options that might fit the requirements. The flexibility of schema-less multi-domain collaboration has the potential to enable a large amount of newly defined concepts to be quickly combined into overall vehicle representations, as depicted in figure 1.

3. The Codex Framework

Codex aims to provide an environment for knowledge capture, formalization and execution in which knowledge from several competence specialists can be collaboratively added and integrated. The ability to quickly adapt or re-use the implemented knowledge-bases results in a model of the aircraft system that can then be tailored to the external requirements and designers inputs, ultimately allowing for quick design space explorations and comparisons between several technologies.
Within this section, first the architecture of the Codex Framework is presented. Thereafter, the creation of domain specific languages (DSL) to ease the modeling using semantic web technologies is described and finally it is shown how parametric and production rules can be used to automatically expand or modify the model graphs.

3.1 Architecture of the Codex Framework

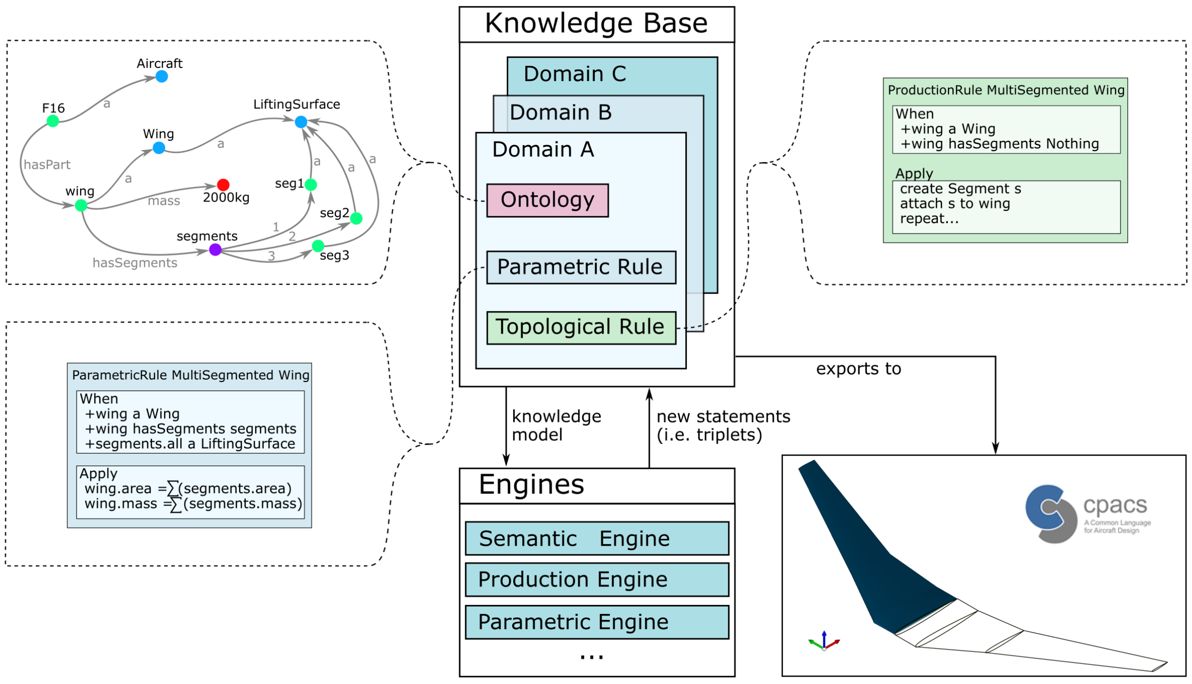
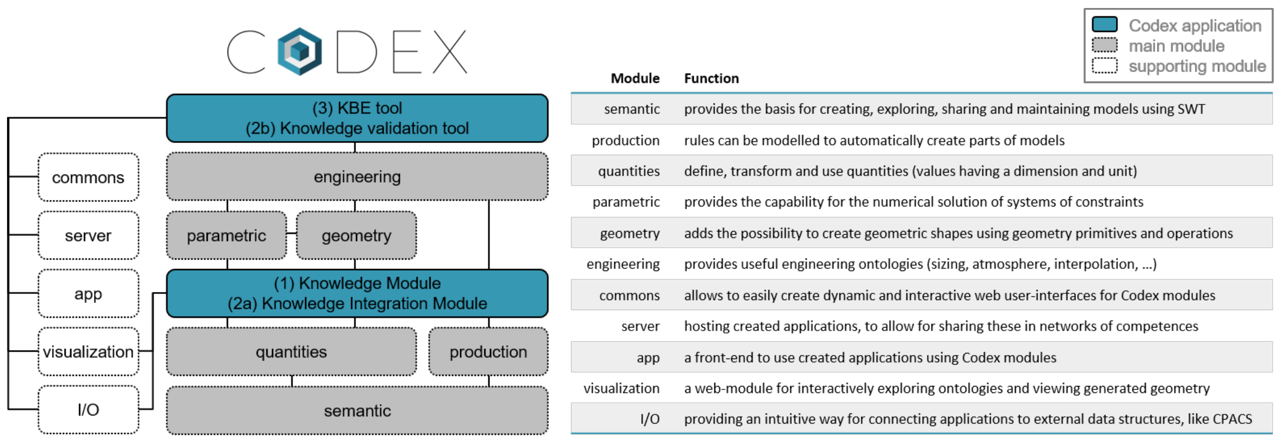


Figure 2 – Codex's knowledge representation and architecture.

Figure 2 shows how knowledge is structured in Codex. A knowledge base contains several domains that have an ontology and a set of rules at their core. Leveraging SWT standards [11], the knowledge is stored as a graph in which all entities are interconnected. Within the graphs, the smallest unit of knowledge is a triple, containing a `subject`, a `predicate` and an `object` (e.g. "wing - is a -> lifting surface" or "wing - has mass -> 2000 kg"). The predicates creating the connections are called `properties` in SWT and exist independently from the other entities available in the graph. In other words, these do not belong to any class or other meta-model structure. This is not the case in frame-based KBE applications, where an instance or property is dependent on the class from which it inherits its characteristics. Thus, a benefit of the graph-based approach is that whenever part of the knowledge base needs to be reused, only the necessary triples are transferred, leaving out any superfluous information. By connecting the nodes of graphs through semantic links, the information available in the graph is augmented and new meaning emerges.
Next to defining the ontology representing a specific domain, competence specialists can formalise procedures to manipulate and extend the graph using rules. These rules are independent from the ontologies. Rules are made of two parts: a *when* side describing query which can match an explicit pattern within the graph and an *apply* side that contains the actions to be performed in case a match is found. A rule is executed whenever its when side matches a corresponding part of the complete knowledge graph. Its results are then added as extra information into the complete graph, which might

lead to matching subsequent rules available within the knowledge-base and further augment the overall knowledge available in the graph. Rules are subdivided according to their nature. *Parametric rules* generate parametric constraints that describe the relationships among the values characterizing the different knowledge nodes. *Production rules* are used to alter the topological information in the knowledge graph by adding or removing entities and connections between them.
Supporting the representation based on ontologies and rules, three types of engines take care of applying the rules and allow for the automatic inference of new knowledge:

- *semantic engines* can infer new knowledge by logically combining the assertions made in the different ontologies (e.g. if `wheel` is part of `landing gear` and `landing gear` is part of `aircraft`, then `wheel` is part of `aircraft`).
- *parametric engines* analyse and numerically solve the systems of parametric constraints providing results to yet unknown parameters that define the different components of the aircraft/model (e.g. computing the mass of the aircraft as a function of its requirements and missions).
- *production engines* execute the production rules and modify the knowledge graph by generating new or removing obsolete entities. An example of such production rules is the automated creation and disposition of seats in a passenger cabin as a function of ergonomic rules and information contained in a knowledge base and its model.



Module	Function
semantic	provides the basis for creating, exploring, sharing and maintaining models using SWT
production	rules can be modelled to automatically create parts of models
quantities	define, transform and use quantities (values having a dimension and unit)
parametric	provides the capability for the numerical solution of systems of constraints
geometry	adds the possibility to create geometric shapes using geometry primitives and operations
engineering	provides useful engineering ontologies (sizing, atmosphere, interpolation, ...)
commons	allows to easily create dynamic and interactive web user-interfaces for Codex modules
server	hosting created applications, to allow for sharing these in networks of competences
app	a front-end to use created applications using Codex modules
visualization	a web-module for interactively exploring ontologies and viewing generated geometry
I/O	providing an intuitive way for connecting applications to external data structures, like CPACS

Figure 3 – Overview of the stack of Codex's modules and application cases (blue)

Figure 3 provides an overview of the modules and application cases currently available in the Codex framework. Three major types of applications of the framework are distinguished:

1. *Knowledge Module* representing ontology of a specific domain.
2. (a) *Knowledge Integration Module* capable of integrating multiple knowledge modules into a multi-domain model.
2. (b) *Knowledge Validation tool* capable of validating e.g. parameter values within an ontology against requirements from another ontology based on a set of executable rules.
3. *KBE tool* representing the ontologies and set of executable rules within a single or multiple domains of application

In the basis, the capability for the **semantic integration** of knowledge is provided, represented by applications (1) and (2a). Next to integrating the knowledge and rules of different domains, the framework allows for **knowledge validation** and the **creation of KBE tools**, represented by applications (2b) and (3) respectively. Representing the very basis of the framework, the *semantic* module implements the semantic engines as described above. Similarly and based on this capability, the *production* module implements production engines. The *quantities* module enables a formal way to define, transform and use quantities. The latter is especially valuable for integrating knowledge within the engineering domain, where knowledge from several sources potentially applying different unit systems can be combined. Using these three modules, the application cases concerning semantic knowledge integration are enabled. In this, the framework provides supporting modules for connecting to existing data structures through its *I/O* module and supports the engineer by *visualization* of the resulting graphs. If users would like to use the framework for knowledge validation or the creation of KBE tools, the parametric, geometry and engineering modules can be used. The *parametric* module implements the parametric engines as described above. A key functionality of KBE applications is the generation of geometries that can be visualised and used for further modeling [8]. Therefore, Codex offers a *geometry* module based on the OpenCASCADE kernel [15], allowing users to easily include geometric knowledge in the models. This module is presented in [16]. Finally, the *engineering* module provides a set of ontologies and rules commonly used within KBE applications within the engineering domain, e.g.: a representation of the standard atmosphere, a sizing routine for mass, center of gravity and inertia calculations and interpolation capabilities. Finally, the supporting modules *commons*, *app* and *server* support the user in sharing the KBE tool within networks of competences.

3.2 Modeling using Semantic Web Technologies

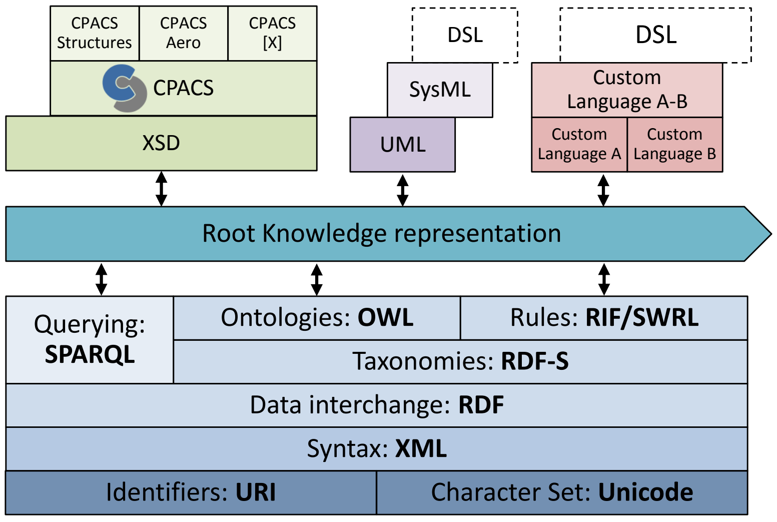


Figure 4 – Codex language layers [7].

As shown in figure 4, the core layer of the Codex knowledge representation is based on SWT standards. As mentioned in section 2. the resulting knowledge graph is characterised by being non-hierarchical and non-domain specific and offering many advantages such as being easy extendable, allowing for transparent data integration and thereby contributing to the ease of collaboration. This representation is highly abstract, a plus when it comes to multi-domain modeling, however it can prove difficult to express highly specific concepts in this way as even a concept that appears simple, could require several statements (i.e. triples) to be formalised. For this reason, in Codex, it is possible to create more expressive Domain Specific Languages (DSL) that simplify the modeling tasks. Such DSL can be implementations of existing languages (e.g. CPACS and SysML) or they can be user-defined for a specific application. An example of such a DSL is provided by the geometry module [16]:

Listing 1: Codex geometry syntax example, syntax elements highlighted in purple.

```
// Create a point and ellipse using geometry syntax
using(GeometrySyntax) {
    point = createPoint(0.0[m], 0.0[m], 0.0[m])
    createEllipse(point, 4.0[ft], 1.0[m], AxisZ, AxisY)
}
```

The two geometry functions (createPoint and createEllipse) result in several statements being added to the model. An excerpt of the added statements is visualised in figure 5 in the form of a semantic graph. The complexity of this graph highlights the importance of adding a specialised language, making it much easier for the user to add specific knowledge to the system.

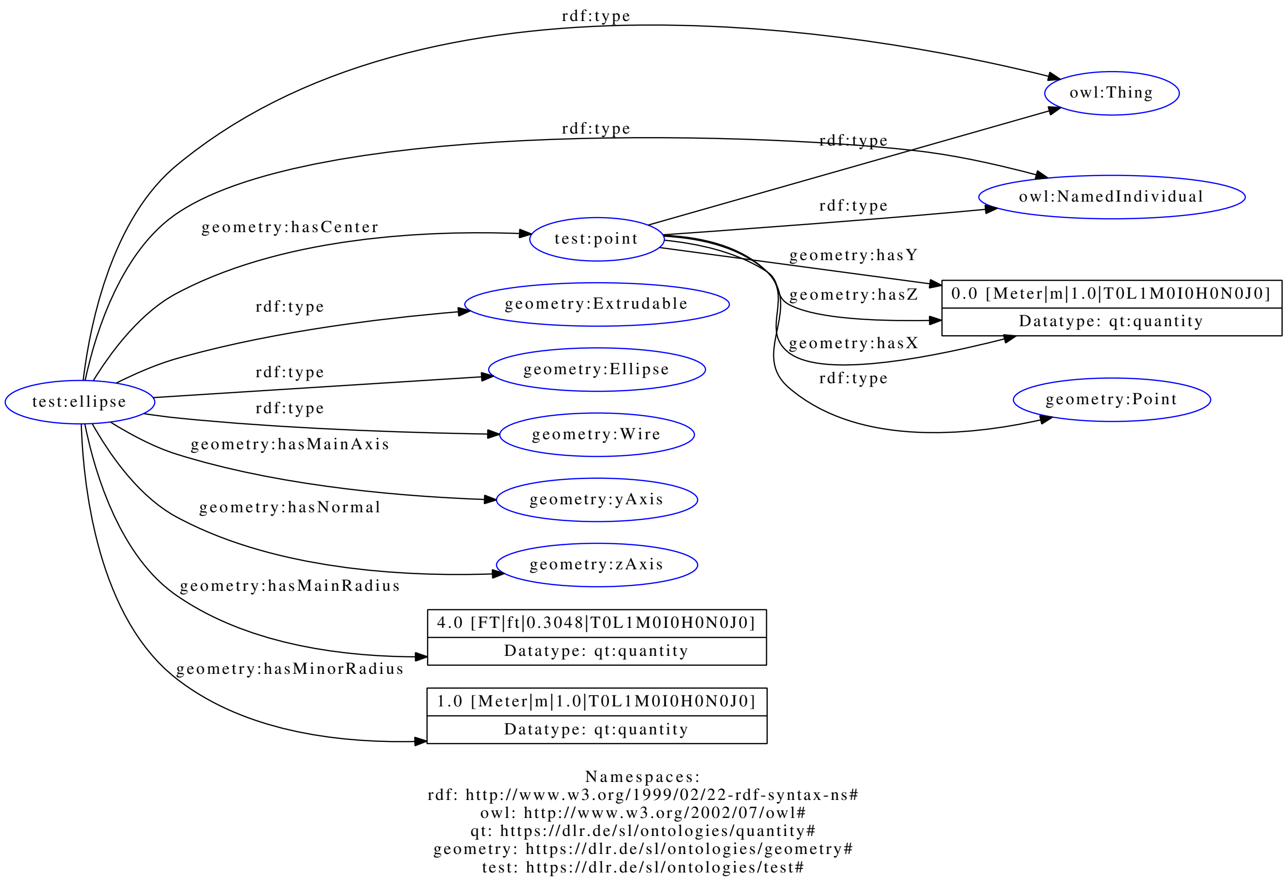


Figure 5 – Semantic graph resulting from the geometry syntax in listing 1, visualised using RDFGrapher [17]. Every linked pair of nodes is the visual representation of a triple. In front of the colon, the name of the domain in which the entity is stored is provided (e.g. rdf, geometry). Blue round nodes identify a resource (i.e. individual or class) and the rectangular nodes are literals (i.e. values).

In a graph, any entity can be pointed at or point to another one, whereas in frame-based modeling, most characteristics can only be accessed by adhering to the strict hierarchical structure of the model. Regularly, this limitation extends to the direction of such a path (i.e. the parent knows about its children but not the other way around). This difference has many consequences that impact the ease of storing and re-using knowledge in a KBE application.

One consequence is that the indirect path taken to access data in frame-based applications can be tedious to implement/remember and it can also lead to implementation fragility. In fact, if any of the elements in the hierarchical path change (e.g. a property is removed from a class or an entity is simply renamed), all entities using such connection need to be updated. On the contrary, in a graph-based model, the path can be direct and minimal to no maintenance is necessary when the ontology is changed.

Another consequence of using a hierarchical modeling structure is that information that is needed in more parts of the model ends up being repeated, increasing modeling and maintenance complexity. For example, imagine a wing made of several segments connected to another, where each segment is characterised by a root and tip airfoil. In a hierarchical model, each segment must have the two airfoils as children according to the meta-model description. Any interface between segments will see a duplication of data as the tip of a segment is identical to the root of the next one. Parametric constraints must then be implemented to ensure that such connections persist when values are recomputed. In the modeling way central to Codex, an intermediate airfoil entity can be pointed to by both segments using distinct properties. The first could point to the airfoil with the object property *hasRootAirfoil* while the following one could point with the semantic connection *hasTipAirfoil*. The fact that only one node exist for the airfoil means that the model is comparatively less complex and the system of parametric constraints is not weighted down by superfluous equality constraints. Furthermore, if data duplication is unwanted, the strict hierarchy can leave an expert conflicted about where it is best to store information, since this might differ depending on the point of view (i.e. domain).

3.3 Using rules to automatically expand or modify graphs

Rules supplement the modeling capabilities offered in Codex by automatically adding, removing or modifying the information contained in the knowledge graph.

Parametric rules

An example of a parametric rule is shown in listing 2. This matches any resource of type `tp:HTP` that is connected to a resource `wing` of type `w:Wing` through the semantic link `tp:hasReferenceWing`. Whenever such a sub-graph is found in the complete knowledge graph, a parametric constraint of type equation is generated and added to the system of parametric constraints[1]. In particular, the equation represents the volume coefficient method commonly used in low fidelity aircraft conceptual design for determining control surface reference areas [18]. It can be noticed that the equation is not written similarly to a value assignment (e.g. y = x.pow(2) + 1) as the parametric constraints in Codex are a-causal, implying that the equation can be used to compute any of its variables depending on the problem boundary conditions. This simplifies the knowledge formalization task, as an equation can be written as preferred by the user without having to worry about which variable should be computed from which given constraints. The task of determining if the system of constraints is well-determined is fulfilled by the Parametric Analyser shipped as part of the `Codex-Parametric` module. The Analyser is able to determine the degrees of freedom or over-determination characterizing a system of equations and communicates to the user possible solutions to the issue. The algorithms used to match variables to constraints and identify irregularities are described in [19] and [20].

Listing 2: Example of parametric rule.

```
/*
Legend:
    rdf:type: Property used to semantically describe a type
              of a given resource, part of the SWT standard
    g: Geometry domain
    ls: Lifting Surface domain
    tp: Tail Plane domain
    w: Wing domain
*/

When{ htp, wing ->
    +(htp..rdf:type..tp:HTP)
    +(htp..tp:hasReferenceWing..wing)
    +(wing..rdf:type..w:Wing)
} Apply { htp, wing ->
    eq(lhs = htp[g:hasArea]*htp[tp:hasACDistance]/htp[tp:hasVolumeCoeff],
       rhs = wing[g:hasArea]*wing[ls:hasMAC])
}
```

Production rules

As shown in figure 2, the production rules complete the set of concepts making up a domain alongside ontologies and parametric rules. Production rules, also referred as *topological rules* in Codex, share the same "when-side" query matching principle as its parametric counterpart, however differ in what is allowed on the "apply-side". Whereas parametric rules only allow for modification to the parametric

[1]lhs and rhs are respectively the left-hand and right-hand sides of the equation.

system by adding or removing constraints, a production rule is much more versatile and permits to execute any block of code, even allowing to call routines external to Codex. Topological rules are more commonly used to directly modify the semantic graph by creating new or connecting existing entities, thus, as the name suggests, changing the graph on a topological level. An example of such a production rule is provided below:

Listing 3: Example of production rule.

```
/*
Legend:
    rdf:type: Property used to semantically describe a type of a given resource
    ls:       Lifting Surface domain
    p:        Performance domain
*/

When{ airfoil , designCondition ->
    +(airfoil..rdf:type..ls:Airfoil)
    +(airfoil..p:hasDesignCondition..designCondition
    +(airfoil..ls:hasNACACode..Anything)
    +(airfoil..ls:hasAerodynamic..Nothing)
} Apply { airfoil , designCondition ->
  // Run external routine xFoil with the airfoil NACA code and design
  // condition (i.e. Mach and Reynolds numbers)
  characteristics = xFoil(airfoil[ls:hasNACACode], designCondition)

  // Link and store results in the knowledge graph
  addAssertion(airfoil , ls:hasAerodynamic, characteristics)
}
```

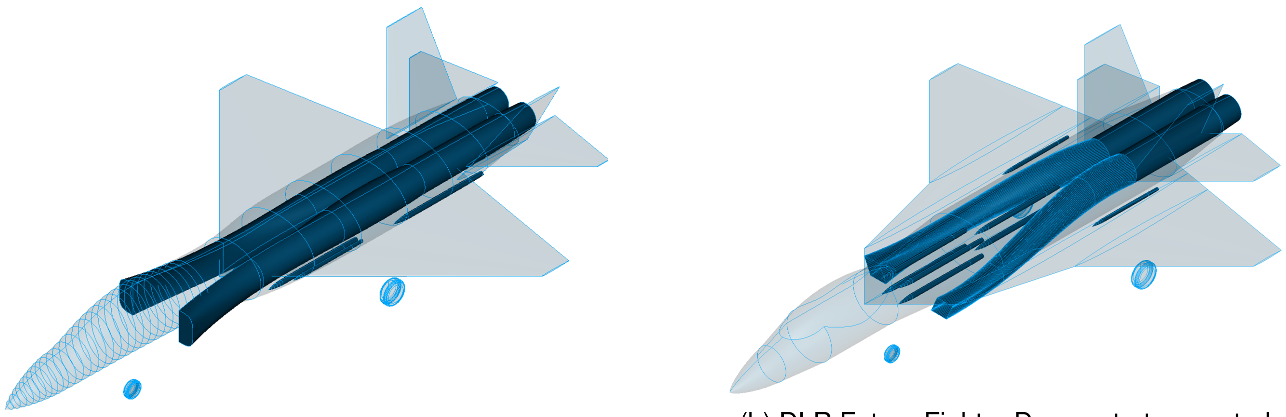
Combining parametric and production rules

While the definition of parametric and production rules is done independently, the execution of a parametric rule can trigger a production rule and vice-versa. For example, a production rule could have a certain condition on its "when-side" that is only satisfied if the value of a specific parameter is above a given threshold. The value of this parameter will only be assigned if a parametric rule involving that parameter is applied and consequently computed by the parametric solver, thus resulting in the indirect triggering of that production rule. Conversely, triggering of a parametric rule after the execution of a production rule is the default case, since parametric rules are always dependent on the topology of the model. While the declarative definition of rules removes the need to specify an execution order, special care has to be taken when orchestrating parametric and production rule-sets to avoid unintended behavior, such as infinite recursion where, for example, the execution of `ruleA` triggers the execution of `ruleB` which in turn modifies the graph in such a way, that it matches `ruleA` again.

Advantages of the presented rule-based approach

Separating rules from the other components of a knowledge base is an advantage when re-using knowledge as only the rules that are actually required can be shared between applications. However, this approach complicates the execution of the KBE application as simply initialising an element will not automatically apply all the rules that would be inherited as would be the case within a frame-based environment. As manually instantiating and connecting the constraints to the model elements would also be unfeasible, Codex solves this issue by leveraging the possibility to match a query in the knowledge graph and use this as trigger for the apply side of a rule. This tends to be more computationally expensive as the graph needs to be continuously observed for changes as the model is mutated. A certain degree of experience in ordering the "when-side" queries can however significantly improve the performance. For this reason, it is intended to support users in implementing efficient queries within the Codex Framework.

Another difference of the proposed approach when it comes to rules is that it follows the declarative programming paradigm [21]. In declarative programming, the user states *what* the program should do in opposition to imperative programming where the *how* it should be done is stated instead. Although all code at the end boils down to imperative instructions for the machine, the extra layer of abstraction offered by declarative programming can be useful in a KBE environment as it allows the specialist to fully focus on formalizing his/her knowledge while not requiring them to also optimise their instructions for performance or providing extensive "if..else.." chains that cover all the possible permutations expected during code execution. These tasks become an implementation detail and can be left to the developers of the different engines/modules that interact with the semantic model, thereby splitting the role of disciplinary expert from the one of knowledge engineer.



(a) DLR Future Fighter Demonstrator created using the VAMPzeroF [2] application.

(b) DLR Future Fighter Demonstrator created using the Codex-Fighter application.

Figure 6 – Improvements to automated geometry generation after transferring the knowledge from VAMPzeroF to the Codex-Fighter application. Among others, the improved representation of the multi-segment wing and inlets can be observed.

As final advantage, improved control over the level of detail of geometry generated within the design approach can be mentioned. A difficulty experienced through the usage of the KBE application VAMPzeroF is the limited control available on complex geometry generation. Both VAMPzeroF and its successor Codex-FighterDesign make use of the same geometry kernel [15], so in principle, the same geometrical operations are available. However, the combination of a geometry DSL and the ability to use production rules to precisely describe geometries at an element level in Codex, allows the user more control on the end result. As exemplified in figure 6, where the same requirements have been used to generate a fighter jet in both KBE applications, this can considerably reduce the effort required to connect the low- to medium-fidelity tool to high-fidelity analyses requiring detailed geometry definitions.

Including rule-based topological changes in designs of experiments

The increased flexibility allowed in modeling and the automation provided by the production rules expand the design space readily explorable from within the same application and knowledge base. This also translates into improvements of what can be studied in a Design Of Experiments (DOE). Experience matured while using the in-house conceptual design tool VAMPzeroF [2] and industry renowned PaceLab APD [22] in several large DOE has shown that performing analysis by exclusively varying the input parameters (i.e. parametric studies) limits the extent of the design space that can be effectively explored. The ability of topologically manipulating the model in an automated way through production rules, opens up the possibility to expand the type of studies to also include architectural changes. These can be as simple as adding or removing some propulsion units from a design but, given the support of tools and methodologies to determine feasible architectures as described in [23], they could open the way to automated system engineering DOE. This is still merely a hypothesis, however some preliminary outlook on automated architectural changes is provided in sub-section 4.2

4. Implementing KBE tools using the Codex Framework

The following implementations provide an insight in how Codex can be used to create a knowledge base, execute its rules and finally visualise the resulting geometries. The first example focuses on an application for the design of multi-trapezoidal lifting surfaces. It presents the steps typically required when creating a completely new KBE application in the Codex framework. Space is also given to the discussion of best practices and pitfalls when modeling using a graph based language. The second example focuses on the way knowledge can be re-used and on how the use of production rules can facilitate the creation of architecturally distinct designs. This discussion is supported by the description and visualization of three combat aircraft designs.

4.1 Multi-trapezoidal lifting surface domain

In this section, an application for the automatic generation of multi-trapezoidal lifting surfaces is described to provide insights into how the Codex framework can be used to generate KBE applications. This particular example has been chosen as it represents a graspable model which leverages all the capabilities of the framework discussed in section 3. of this paper. The aim of this use-case is to formally describe a parametric and geometric model of a lifting surface made of a flexible amount of segments, which are continuously connected to one another.

When it comes to building a new KBE application, first a domain needs to be initialised. Here is where information about `Types` (i.e.: classes) and properties divided in `Object` and `Data-properties` are stored. Object properties connect nodes to an entity of type `individual`, while the data properties connect to a `literal` (i.e. a node representing some value). Contrary to frame-based approaches, the creation of classes is not a prerequisite for defining individuals. These are applied to individuals to classify them to be of certain `type`.

Listing 4 shows an excerpt from the lifting surface domain central to the current example. In the first section we see the definition of types/classes. E.g. the line "LiftingSurface a Class" declares a new `owl:Class` that can be referenced in code and simultaneously adds the statement "ls:LiftingSurface - rdf:type -> owl:Class" to the semantic domain model "[2].

In the case of the `ls:AerodynamicCenter` type, extra information in the form of a comment, a WikiData entry [24] and a source are added to the semantic graph - providing other users a better idea of the semantic meaning of the type. Other annotations exist such as labels, author and version info, which are generally used to enhance the transparency of the created models.

Listing 4: Excerpt from Lifting Surface domain: Classes.

```
LiftingSurface a Class
AerodynamicCenter a Class {
    addComment("The␣Aerodynamic␣center␣is␣the␣point␣at␣which␣the
␣␣␣␣␣␣␣␣pitching␣moment␣coefficient␣does␣not␣vary␣with␣lift
␣␣␣␣␣␣␣␣coefficient␣(i.e.␣angle␣of␣attack).␣Its␣longitudinal
␣␣␣␣␣␣␣␣position␣is␣typically␣at␣around␣25%␣of␣the␣MAC."
    )
    addSeeWikiData("Q1229549")
    addSource("https://en.wikipedia.org/wiki/Aerodynamic_center")
}
```

Listing 5 provides an excerpt of object and data properties declarations. Whereas the object properties will be used to link to individuals, e.g. `ls:hasAerodynamicCenter` will link to an individual that represent the aerodynamic center of the entity being connected, the data properties provide additional information in the form of values, for example `ls:hasNACACode` will link to a string that identifies a NACA airfoil, while `ls:hasAspectRatio` and `ls:hasDihedral` will point to numerical values. Like the classes, also properties can be annotated to better reflect the intentions of the knowledge engineer.

Listing 5: Excerpt from Lifting Surface domain: Properties.

[2]As in previous examples, the names in front of the colon denotes the domain of a given entity. In particular, ls is the Lifting Surface domain, rdf and owl denotes the SWT core domains.

```
// Object Properties
hasAerodynamicCenter a ObjectProperty
hasSegments a ObjectProperty

// Data Properties
hasNACACode a DataProperty {
    addSource("https://en.wikipedia.org/wiki/NACA_airfoil")
}
hasAspectRatio a ParameterProperty(Length.pow(2) / Area) {
    lower(0.0[m2/m2])
    initial(7.5[m2/m2])
    upper(60.0[m2/m2])
}
hasDihedral a ParameterProperty(Angle) {
    unit(deg)
    initial(0.0[rad])
}
```

When it comes to data properties, an extra type of semantic link, the `ParameterProperty`, has been included in Codex. When used as shown in the example below, it still results in a data property but it also allows the expert to add important information for the parametric system modeling and numerical computation. In particular, every `ParameterProperty` requires the user to declare the dimension of the value to which the property will point at. In the listing, `Length` and `Length.pow(2) / Area` are two such dimensions (with the former being a base dimension and the latter a derived one). These are used whenever a constraint is added to the parametric system to check that the parameters used respect the dimensional analysis rules of the different arithmetic operations. Furthermore, a user can also define lower, initial and upper boundary quantities for a given property. These are useful for improving the convergence of the numerical computation but are not strictly necessary. Finally, convenience functionalities such as the function "unit(deg)" are provided to improve the application ease of use. In particular, this function stores the request of the user of converting any result that uses the `ls:hasDihedral` property from the default SI-unit *radians* to *degrees* instead. The implementation of quantities is of great benefit to an application aimed at the engineering sector as it allows a seamless use of different unit systems while ensuring the dimensional analysis and unit conversions are automatically taken care of. In order to provide the aforementioned capabilities, Codex directly implements QUDT [25], an open-source set of ontologies based on SWT for the formal description of dimensions, units and quantities. Codex offers the user around 250 different dimensions and 270 units allowing also the creation of new derived ones through arithmetic operations (e.g. "Length.pow(2) / Area" and [m2 / m2]).
As explained in sections 3.3and 3.3 rules should be created alongside the ontology to completely define a domain. The rule shown in 6 takes care of finding two individuals of type `ls:Segment` that are next to each other in the multi-segmented wing. When found, it adds a triple to the semantic model that states that the tip airfoil of the former segment is the same entity as the root airfoil of the latter. This way of sharing entities is very effective as any parameter that is computed for the tip airfoil is also available to the root airfoil and vice-versa. This single rule connects all common airfoils shared by connected segments across the entire multi-segmented wing.

Listing 6: Excerpt from Lifting Surface domain: Production Rule.

```
/*
Legend:
    ls: Lifting Surface domain
    e: Engineering domain
    rdf: RDF standard domain
    owl: OWL standard domain
*/

When { tipAirfoil , rootAirfoil ->
    +(previous..rdf.type..ls.Segment)
    +(previous..e.hasNext..next)
    +(previous..ls.hasTipAirfoil..tipAirfoil)
    +(next..ls.hasRootAirfoil..rootAirfoil)
} Apply { tipAirfoil, rootAirfoil ->
    addAssertion(tipAirfoil, owl.sameAs, rootAirfoil)
}
```

Another example of production rule created as part of the lifting surface domain is shown in listing 7. This generates a geometrical wire anytime the query matches an individual of type `af:Airfoil` that has some NACA series value and is connected to an individual of type `g:Point`.

Listing 7: Excerpt from Lifting Surface domain: Geometry Rule.

```
/*
Legend:
    ls: Lifting Surface domain
    e: Engineering domain
    g: Geometry domain
    rdf: RDF standard domain
*/

When { airfoil, refPoint ->
    +(airfoil..rdf.type..ls.Airfoil)
    +(airfoil..ls.hasNACACode..Anything)
    +(airfoil..e.hasRefPoint..refPoint)
} Apply { airfoil, refPoint ->
    using(GeometrySyntax) {
      // Create a wire given airfoil characteristics and reference point
      // then link the resulting properties to the airfoil individual
      createWireFromNACACode(airfoil, refPoint)
    }
}
```

Such a rule shows an interesting capability that emerges from the way rules are set up in Codex: since nothing happens until the when-side is completely matched to a part of the semantic graph, the knowledge engineer can use such a characteristic to have a level of fidelity that dynamically changes depending on the contents of the model. In this case, if an airfoil is being given a NACA code, it is automatically generated by the geometry plugin and can be immediately visualised to provide the user instant feedback on model changes.
An experienced knowledge engineer can leverage this characteristic of Codex, by defining rules in such a way that the fidelity and complexity of the model can be easily adjusted by merely providing some elements to the graph. There is no need to create and maintain complex if-else chains covering all possible cases. Application of such a strategy enables:

- Automated activation of geometry generation only when entities contain all the necessary geometrical information as prerequisites.
- Automated switching between internal models and higher fidelity response surfaces information, the moment this is provided, e.g.: replacing empiric correlations for determining aerodynamic coefficients with results from CFD analyses. Useful to use the same KBE application both as an *initialiser* and *synthetiser* depending on the variables provided upon execution. This proves to be a key feature when creating large multi-fidelity simulation workflows as described in e.g. [26].
- Automated increase of fidelity the applied product development process, allowing for the seamless switch from top-level requirement driven design to the detailed sizing of sub-system architectures. This can provide support for system of systems studies as discussed in [27].

With a defined domain consisting of ontologies and rules, a user can start to apply the formalised knowledge to model new ideas. This is done in the same way as building the ontology itself, i.e. by adding statements to the knowledge graph until it matches the intentions of the user. The listings 8

and 9 present some lines of code that use a combination of Manchester [28], Semantic, Geometry and Parametric DSLs to initialise the model of a multi-trapezoidal wing.
After generating a new knowledge base and importing the ontologies that are used in the model creation, the user can create new individuals:

Listing 8: Model initialization: Ontology.

```
seg0 a Individual; seg1 a Individual; seg2 a Individual; winglet a Individual

semiWing a Individual {
  Types(ls:LiftingSurface, ls:Asymmetric)  // Add rdf:type assertions
  Facts{
    ls:hasSegments(
      createSequence(segment0, segment1, segment2, winglet)
    )
  }
}

wing a Individual {
  Facts{
    w:hasSemiWing(semiWing)
    e:hasReferencePoint(createPoint(0.0[m], 0.0[m], 0.0[m]))
    e:hasSymmetryPlane(PlaneXZ)
  }
}
```

Even when no extra-information is given, rules can support the model creator by inferring not explicitly stated knowledge. For example, by having a semantic rule in the supporting ontology that states that any entity connected via the `ls:hasSegments` property must be of type `ls:Segment`, the workload of model initialization can be reduced considerably.
With an established basic model structure in place, the user can add further information to the knowledge graph such as the data properties below:

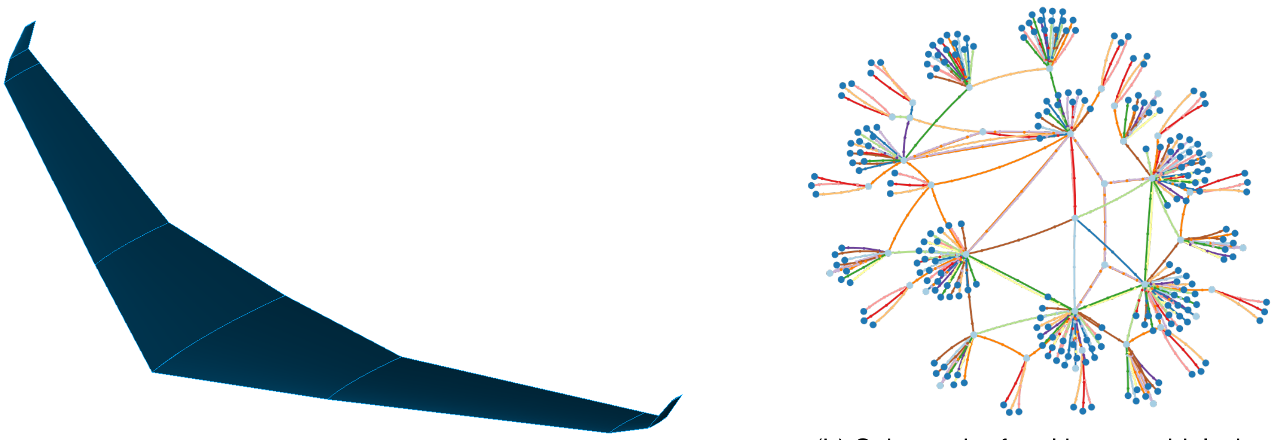
Listing 9: Model initialization: Rules.

```
// using "segment0..ls:hasRootAirfoil" to get the airfoil through query
addAssertion(segment0..ls:hasRootAirfoil, ls.hasNACACode, "NACA23015")
addAssertion(segment0..ls:hasTipAirfoil, ls.hasNACACode, "NACA23015")
... // omitted data properties for brevity

// Use convenience of Parametric syntax to create data properties
// that point to values of type quantity
ParameterConstraints{
  wing[g.hasArea] = 1320.0[ft2]
  wing[ls.hasAspectRatio] = 10.3[ul]  // ul: unitless

  segment0[ls.hasSweepLE] = 25.0[deg]
  segment0[ls.hasTaperRatio] = 0.55[ul]
  ...  // omitted boundary constraints for brevity
}
```

Once the execution starts, the data-properties pointing to numerical values are automatically transformed into boundary constraints and added to the system of equations that will be computed by the internal numerical solver. When the model is executed and solved, it results in the multi-trapezoidal wing shown in figure 7a. This wing is characterised by a semi-wing made of four segments lofted through a total of five airfoils (three shared among segments and two as end-caps) and mirrored with respect to the xz-plane. Figure 7b shows a sub-graph of the resulting semantic model where only the individuals (light-blue nodes), the parametric quantities (dark-blue nodes) and the properties among them (i.e. the graph edges/links) are included.
Codex aims to facilitate cross-domain integration and knowledge re-use. To show this, the knowledge available in the multi-trapezoidal lifting surface domain can be used to generate the two geometries

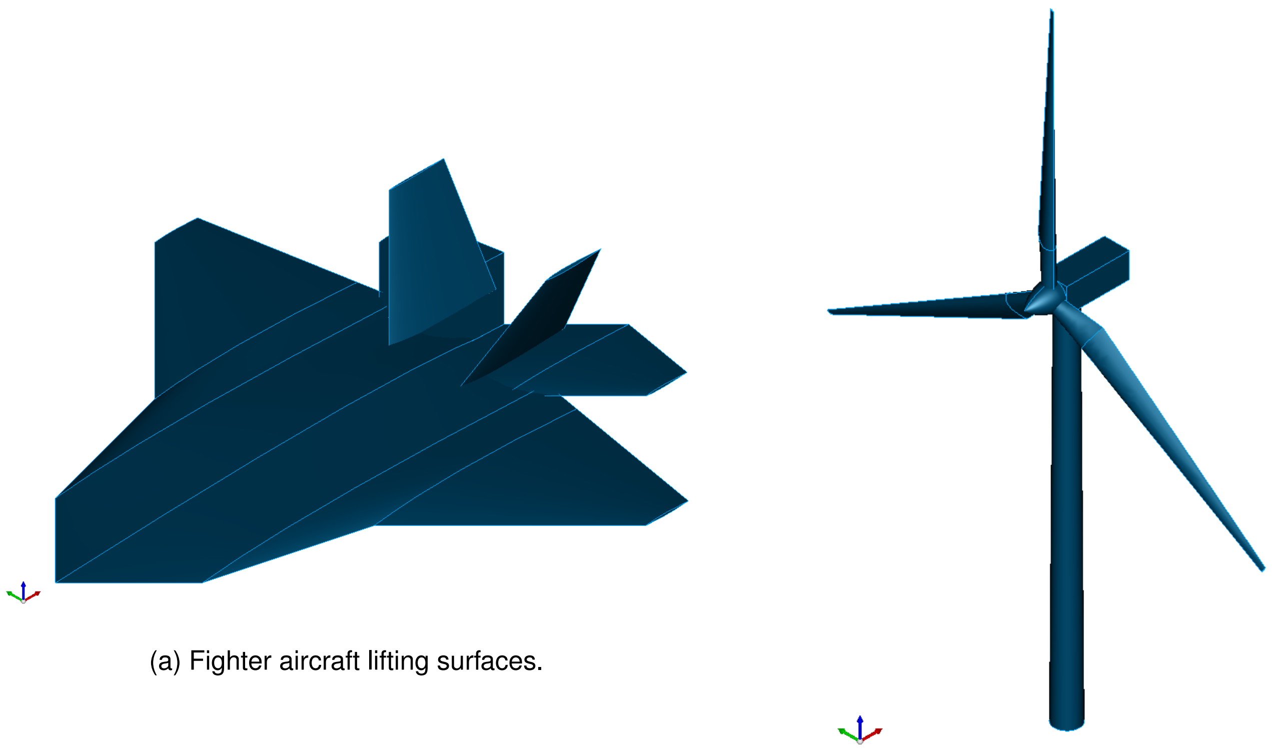


(a) Multi-trapezoidal wing.

(b) Sub-graph of multi-trapezoidal wing model.

Figure 7 – Multi-trapezoidal wing result visualization.

shown in figures 8a and 8b depicting the lifting surfaces of a high-velocity fighter aircraft configuration and a model of a wind turbine. To create the first geometry, the newly generated domain has been connected to knowledge concerning tail plane sizing (e.g. parametric rule 2) and wing mass estimation to allow its use in the conceptual design of the combat aircraft shown in figure 9. To create the wind turbine geometry, the production rule mirroring the lifting surface with respect to a plane has been substituted by a rule providing the ability to radially mirror a shape - allowing the rapid generation of multi-bladed wind turbines.



(a) Fighter aircraft lifting surfaces.

(b) Conceptual design of wind turbine.

Figure 8 – Re-using the knowledge base covering multi-trapezoidal lifting surfaces within differing KBE applications.

4.2 Architectural changes

While modern KBE applications allow users to quickly explore the design space by varying the parametric system, limits exist when these studies should include more complex changes - such as the automated generation of topologically differing configurations or sub-system architectures. In this application, the same rule-base within Codex is used to generate three combat aircraft designs as shown in figures 9, 10 and 11. This clearly shows how Codex can be utilised to re-use knowledge while at the same time expanding the design space that can be explored.
The main characteristics of each design are:

- Conventional fighter jet: has canopy and cockpit, multi-trapezoidal wing with conventionally located tail planes, two engines fed by two S-shaped inlets, conventional engine nozzle.
- Conventional UCAV: has no canopy or cockpit but uses the internal space for added communication sub-systems, single trapezoid wing and conventional tail planes, single engine fed by a straight Y-shaped inlet, conventional engine nozzle.
- Flying-wing UCAV: no canopy, cockpit and fuselage, flying wing architecture with no tail planes, single engine fed by straight dorsal inlet, 2D engine nozzle.

Although these designs are visually distinct, they share the same knowledge base. They differ thanks to the activation of separate rules due to the dissimilar input provided by the user, reflecting in the created models. For example, when it comes to the inlets, the user needs only to state few characteristics to then generate the geometries seen in the three figures. In particular, aside from the sizing characteristics of the inlet that depend on the engine and performance requirements, and are thus computed, the model of each design needs information about the number of inlets entry ports, the number of engines, the shape of inlet (S-shaped or straight in this example) and the vertical location (typical choices are ventral, side or dorsal). Codex can use these conditions to further infer knowledge that then triggers other parts of the knowledge base. For example, a two-ports single engine inlet will be inferred as being of a Y-shape type thus activating the corresponding rules for its sizing and positioning in space.

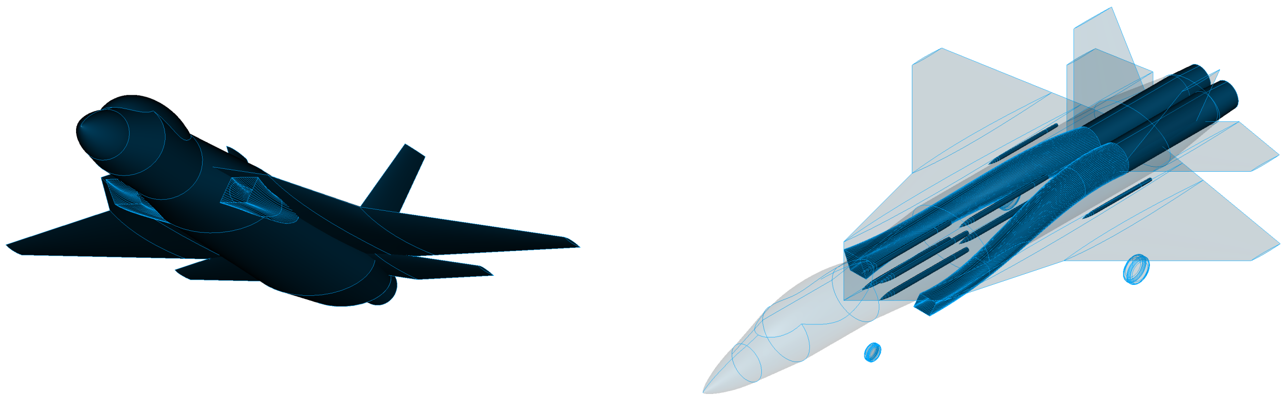


Figure 9 – Conventional fighter jet.

A particularly interesting case is the flying wing design. While the fighter and conventional UCAV share many similarities making the conversion between them simpler, the flying wing does not posses either tail planes or fuselage. The lack of the latter creates a small obstacle in that the many rules are written with reference to the fuselage sizes. For example, both weapon bays and landing gear locations typically depend on the fuselage geometry. This is easily solved in Codex as the design builder can state that the wing individual is also of type fuselage. Furthermore, a new semantic rule that asserts that any individual of type wing and fuselage is also of type wing-body could be added. In this way, this peculiar combination of characteristics can be named and made more easy to find when querying the model. Finally, even when asserting that the wing is a fuselage, the user would not have to worry that the wing-body individual "inherits" useless characteristics (e.g. fuselage diameter or fuselage internal structures would not make sense for such an entity), as classes and properties are fully independent in Codex' semantic model.

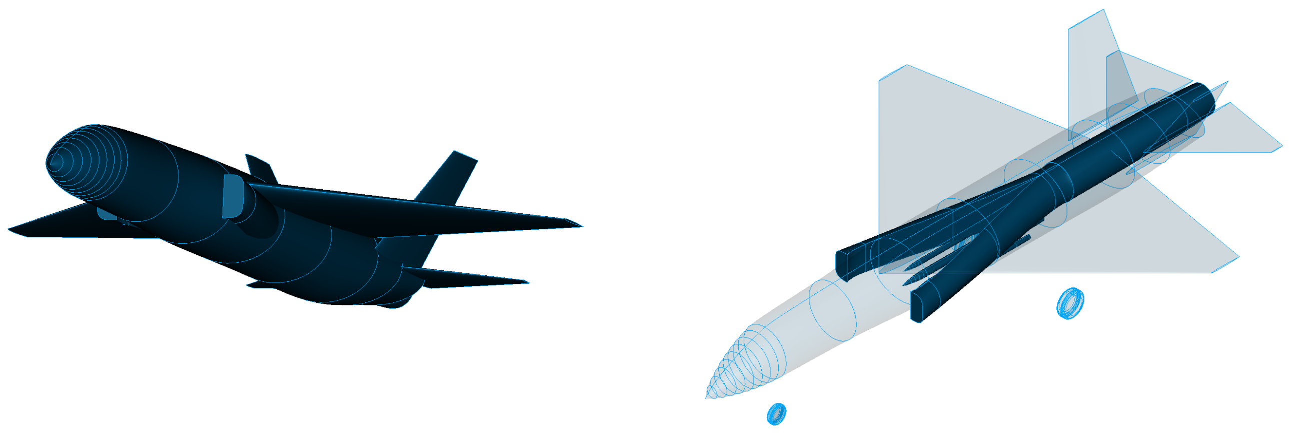


Figure 10 – Conventional UCAV.

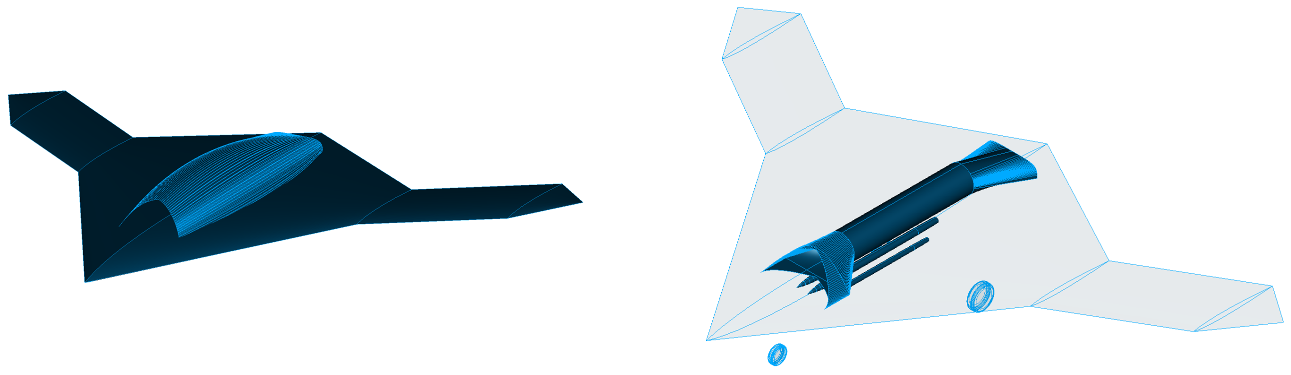


Figure 11 – Flying-Wing UCAV (external inlet structure removed for clarity).

5. Conclusions and Outlook

Knowledge-based engineering offers methods to cope with the complexity of exploring the vast design space that characterises modern air vehicles concepts. This paper discusses how the adoption of a graph-based, modeling focused standard such as SWT can further improve KBE capabilities by:

- improving multi-domain knowledge integration with minimal to no repetition of information
- enhancing collaboration between disciplinary experts by providing an abstract and domain-agnostic language to integrate knowledge while maintaining ease of use and of knowledge formalisation thanks to using domain-specific languages
- allowing more effective re-use of knowledge across applications by embracing the widely accepted SWT standards as the core modeling language
- offering an effective strategy to automate architectural and configurations changes via a rule-based system that can both topologically change the modeling graph and fill-in the knowledge gaps of unknown parametric variables
- decreasing implementation fragility and improving KBE application maintainability by separating the formalization of knowledge from the rules allowing manipulation of knowledge and by avoiding the identified pitfalls encountered in frame-based hierarchical modeling approaches

Although the proposition of extending knowledge-based engineering with semantic web technologies shows great promise, the approach also presents some challenges. Firstly, moving to a graph-based approach to modeling means also losing the main advantage of the frame-based ones such as the ability to directly inherit all the properties, methods and characteristics of the instantiated classes. Furthermore, thinking about knowledge less as a hierarchical structure and more as a lattice/graph requires some adaptation and work from the experts as the complexity of the model increases. Finally, while a generic and abstract modeling language is advantageous for cross-domain integration, its lack of specificity makes it more verbose and difficult to use by the competence specialists.

The Codex framework is being developed to implement the proposition of using SWT in a KBE framework and to address the aforementioned challenges. It mainly does so by providing an interface between the highly abstract and generic modeling language while providing the experts with the tools required to create new KBE applications. It currently allows to model, compute and generate geometries using a tailored programming language. This is purposefully done to prove the framework is indeed capable of providing the flexibility and modularity the authors hypothesised. However, the authors are well-aware that an extension is needed, allowing competence specialists to model their knowledge themselves within the framework - without having to learn any detailed programming language or routines. Therefore, a web-based user-interface is currently being developed, providing a visually supported environment for knowledge digitization. Next to allowing a more intuitive environment for knowledge formalisation, execution and exploration than can be offered by a programming language, its largest contribution will be the ability for multiple users to concurrently contribute to the development of the same knowledge base.

Further future tasks related to the Codex Framework development are the continuous development of the several KBE applications that will be used within the multitude of ongoing and planned collaborative air vehicle design projects both within DLR and within national and international consortia. In particular, it is the intention of the authors to complete the knowledge base of Codex-FighterDesign and use it as a model generator for agent-based system of systems studies. Finally, the authors aim at open-sourcing part of the Codex framework to allow interested parties in using this novel approach to KBE and possibly contributing to its growth.

6. Main author contact details

Jacopo Zamboni
German Aerospace Center (DLR) – Institute of System Architectures in Aeronautics
Email: jacopo.zamboni@dlr.de
Telephone: +49 (0)40 2489 641 340

7. Copyright Statement

References

[1] S. Woehler, G. Atanasov, D. Silberhorn, B. Fröhler, and T. Zill. Preliminary aircraft design within a multidisciplinary and multifidelity design environment. In *Aerospace Europe Conference 2020*, April 2020.

[2] A. Mancini, J. Zamboni, and E. Moerland. A knowledge-based methodology for the initiation of military aircraft configurations. In *AIAA AVIATION 2021 FORUM*, August 2021.

[3] E. Moerland, S. Deinert, J. Dornwald, and A. Defence. Collaborative aircraft design using an integrated and distributed multidisciplinary product development process. In *30th Congress of the International Council of the Aeronautical Sciences*, pages 1–12, 2016.

[4] P. S. Prakasha, P. D. Ciampa, L. Boggero, M. Fioriti, B. Aigner, A. Mirzoyan, A. Isyanov, K. Anisimov, I. Kursakov, and A. Savelyev. Collaborative system of systems multidisciplinary design optimization for civil aircraft: Agile eu project. In *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, 2017.

[5] E. Moerland, T. Pfeiffer, D. Böhnke, J. Jepsen, S. Freund, C. M. Liersch, G. P. Chiozzotto, C. Klein, J. Scherer, Hasan Y. J., and J. Flink. On the design of a strut-braced wing configuration in a collaborative design environment. In *17th AIAA Aviation Technology, Integration, and Operations Conference*, 2017.

[6] Marvin Minsky. A framework for representing knowledge. *MIT AI Memo 306*, June 1974.

[7] J. Zamboni, A. Zamfir, and E. Moerland. Semantic knowledge-based-engineering: The codex framework. In *Proceedings of the 12th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*, volume 2, pages 242–249, November 2020.

[8] G. La Rocca. Knowledge based engineering: Between AI and CAD. Review of a language based technology to support engineering design. *Advanced Engineering Informatics*, 26(2):159–179, 2012.

[9] IILS mbh. Design Cockpit 43. `https://iils.de/`, 2020. Online; accessed Jul 27, 2020.

[10] Object Management Group (OMG). Unified Modeling Language (UML). `https://www.omg.org/spec/UML/2.5.1/About-UML/`.

[11] World Wide Web Consortium. Semantic Web. `https://www.w3.org/standards/semanticweb/`, 2015. Online; accessed Jul 27, 2020.

[12] M. Alder, E. Moerland, J. Jepsen, and B. Nagel. Recent advances in establishing a common language for aircraft design with cpacs. In *Aerospace Europe Conference 2020, Bordeaux, France*, 2020.

[13] E. Moerland, A. Mancini, and B. Nagel. Towards a seamless simulation of the air transport system. In *33rd Congress of the International Council of the Aeronautical Sciences, Stockholm, Sweden*, 2022.

[14] World Wide Web Consortium. Shapes Constraint Language (SHACL). `https://www.w3.org/TR/shacl/`, July 2017. Online; accessed Mar 23, 2022.

[15] OPEN CASCADE S.A.S. Open CASCADE Technology. `https://dev.opencascade.org/doc/occt-6.7.0/overview/html/index.html`, 2014.

[16] B. Boden, Y. Cabac, T. Burschyck, and B. Nagel. Rule-based verification of a geometric design using the codex framework. In *33rd Congress of the International Council of the Aeronautical Sciences*, pages 1–15, 2022.

[17] RDF Grapher. `https://www.ldf.fi/service/rdf-grapher`. Online; accessed March 25, 2022.

[18] D. Scholz. Empennage sizing with the tail volume complemented with a method for dorsal fin layout. *INCAS bulletin*, 13(3):149–164, 2021.

[19] H. Yusan and S. Rudolph. On systematic knowledge integration in the conceptual design phase of airships. In *13th Lighter-Than-Air Systems Technology Conference*, Reston, Virigina, 1999. American Institute of Aeronautics and Astronautics.

[20] S. Rudolph and M. Bölling. Constraint-based conceptual design and automated sensitivity analysis for airship concept studies. *Aerospace Science and Technology*, 8(4):333–345, 2004.

[21] D. Fahland, D. Lübke, J. Mendling, H. Reijers, B. Weber, M. Weidlich, and S. Zugal. Declarative versus imperative process modeling languages: The issue of understandability. In *Enterprise, Business-Process and Information Systems Modeling*, pages 353–366, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[22] J. Zamboni, R. Vos, M. Emeneth, and A. Schneegans. A method for the conceptual design of hybrid electric aircraft. In *AIAA Scitech 2019 Forum*, AIAA SciTech Forum. American Institute of Aeronautics and Astronautics, 2019.

[23] J. Bussemaker and P. D. Ciampa. Mbse in architecture design space exploration. In Azad M. Madni, Norman Augustine, and Michael Sievers, editors, *Handbook of Model-Based Systems Engineering*. Springer, April 2022.

[24] WikiMedia. WikiData. `https://www.wikidata.org/wiki/Wikidata:Main_Page`.

[25] FAIRsharing Team. Fairsharing record for: Quantities, units, dimensions and types.

[26] Moerland, E., Pfeiffer, T., Böhnke, D. et al. On the design of a strut-braced wing configuration in a collaborative design environment. In *17th AIAA Aviation Technology, Integration, and Operations Conference*, page 4397, 2017.

[27] L. Knöös Franzén. *An Ontological and Reasoning Approach to System of Systems*, volume 1907. Linköping University Electronic Press, Linköping, 2021.

[28] M. Horridge and P. Pater-Schneider. Manchester syntax.