A Model-Based Systems Engineering Framework for developing Knowledge Based Engineering Applications

Master Thesis

Fábio M. Fernandes

TU Delft

A Model-Based Systems Engineering Framework for developing Knowledge Based Engineering Applications

by

Fábio M. Fernandes

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday July 6, 2023 at 09:30 AM.

Student number:	4921437	
Project duration:	April, 2022 – July, 2023	
Thesis committee:	Dr. ir. G. la Rocca,	TU Delft, Chair
	Dr. ir. A. R. Kulkarni,	TU Delft, Supervisor
	Dr. ir. M. F. M. Hoogreef,	TU Delft, Examiner
	Dr. J. Guo,	TU Delft, External Examiner

TUDelft

Acknowledgements

This thesis marks the end of my Master's Degree in Aerospace Engineering at TU Delft, and with it the end of my academic life as a student. I would like to take this moment to express my gratitude to all those who provided unwavering support and contributed to the realization of this achievement.

I am most thankful to my supervisors, Dr. ir. Gianfranco La Rocca, and Dr. ir. Akshay Kulkarni, whose support, guidance and continuous feedback on this thesis were the key to the progress and success of my work. I would also like to thank Dr. ir. Maurice Hoogreef and Dr. Jian Guo for their availability as external members of my graduation committee. Furthermore, I am grateful to Darpan Bansal, for his feedback on the draft version of this thesis.

I would like to extend my sincere thanks to the members of the DEFAINE project, in particular to Robin Augustinus and Bram Timmer from GKN Aerospace Fokker Elmo for their support and insights during our collaboration; and to Max Baan and Reinier van Dijk from ParaPy for their support with the ParaPy software and availability to discuss the progress of my thesis, providing guidance in shaping its direction.

Over the last years in Delft, I feel like I have grown the most, not only academically, but also on a personal level. For the latter, I must thank my friends for all the experiences we've shared. Getting to know amazing people like you, with different cultures, interests, and ideas is what made me keep evolving with an open mind, and has made this time in Delft so enjoyable. I won't mention names because I would definitely miss someone, but thank you all very much for sharing with me this incredible moment of my life, you were the ones who made it truly memorable. A special mention goes out to Cristian, Rubén, and Josema, for all the amazing moments we've shared and the great friendships we've built. Regardless of now being in different parts of the world, we always find ways to connect.

To my friends from Portugal, I don't have to mention you all by name, you know who you are. Thank you for your true friendship and for being there during the good and the hard moments. There is one person, however, to whom I must make a special mention, Manuel, we are the proof that brotherhood has nothing to do with blood ties.

Last but definitely not least, I would like to thank my family for their support, your help is always there even when I don't ask. In particular to my parents, there are not enough words to express how grateful I am to you for your unconditional love, support, and sacrifices. Without you, none of this would have been possible.

I could not finish writing this without an extra special thanks to my wonderful girlfriend, Joana, for her love, friendship, patience, and wisdom. Without you I would be a lesser person.

Fábio M. Fernandes
Delft, July 2023

Abstract

Knowledge Based Engineering (KBE) is a particularly relevant technology for addressing the increasing complexity of engineering systems, the need for rapid time-to-market, and the need for achieving reductions in the costs of product development. KBE applications can be effective means of automating repetitive engineering design tasks, enabling engineers to enhance their designs through optimization and innovation. However, the current development process of KBE applications can be improved, as it has shortcomings that limit a wider adoption of KBE technology. Currently, two primary approaches are employed in the development of KBE applications. The first approach involves directly coding the engineering knowledge within the application itself, while the second approach entails modeling the engineering knowledge outside the application and subsequently converting it into executable code, manually. Both approaches result in applications that are perceived, to varying degrees, as “black boxes”. This makes it challenging to understand how the application reaches its conclusions, which can hinder the end-user’s trust in the application and limit its acceptance. To date, a suitable methodology to effectively support KBE app development is lacking, which has considerable implications on the time required for application development, as well as the quality of the applications in terms of traceability of requirements and domain knowledge within the KBE application code, the applications’ maintainability and scalability, and (eventually) the ability to preserve and efficiently reuse engineering knowledge.

To address the outlined shortcomings of the current KBE app development process, this thesis proposes a novel framework for the development of KBE applications, based on Model-Based Systems Engineering (MBSE) concepts, to model domain knowledge and requirements, and to support (semi-)automatic generation of KBE apps through visual editing, as opposed to standard coding. The key objectives of this framework are to improve knowledge capture and formalization, requirements traceability, and knowledge reuse in KBE applications. In the proposed framework, the knowledge required for developing a KBE application is first captured in a formal knowledge model that uses the industry-standard Systems Modeling Language (SysML). Source code is then automatically generated for the targeted KBE system (ParaPy) using a model-to-code tool developed in this research. Traceability of requirements onto the various elements of the KBE app architecture is also provided, thereby reducing the typical *black-box* effect of KBE applications. Furthermore, the framework allows to reuse knowledge from previously generated knowledge models, enabling effective project-to-project knowledge transfer.

This thesis presents the development of three distinct KBE applications using the proposed framework, with the aim of evaluating it in terms of ease of modeling, development time, and quality of the automatically generated (skeleton) code. Preliminary results show that the learning curve to modeling is intuitive and easy enough to learn; the time required for generating the knowledge models is lower than current modeling processes; the automatically generated code is error-free, well-structured, and complies with existing coding standards, providing a correct starting point for further app development, while resulting in time savings in the development of the app skeleton.

Contents

List of Figures

List of Tables

1 Introduction

In today's competitive global marketplace, organizations are under pressure to deliver innovative products to market quickly. As modern engineering systems become increasingly complex, often involving numerous interacting components, intricate design requirements, and multidisciplinary considerations, traditional design processes struggle to handle this complexity efficiently and accurately [1]. Traditional design processes often involve repetitive tasks and redundant efforts, resulting in prolonged development cycles and increased development costs [2, 3].

Knowledge Based Engineering (KBE) provides a systematic approach to manage and leverage knowledge effectively, enabling engineers to tackle the complexities involved in the design and analysis of modern engineering systems, while enhancing the engineers' productivity. KBE involves the use of dedicated software tools called KBE systems, to capture and reuse product and process engineering knowledge in order to reduce the time and costs of product development. KBE achieves these goals by automating repetitive, non-creative, design tasks and supporting multidisciplinary design optimization in all the phases of the design process [4].

Engineering applications developed to automate design tasks using KBE systems are called KBE applications. The development of a KBE application encompasses the acquisition of necessary product and process knowledge from the domain experts, which is followed by the translation of this knowledge into code using the programming language provided by the KBE system at hand.

There are several methodologies available in the literature that attempt to formalize and improve the KBE application development process. The most well-known and established is the Methodology and software tools Oriented to Knowledge-based engineering Applications, also known as, MOKA [5]. The main objectives of MOKA are to reduce the lead times and costs of developing KBE applications, by providing a consistent way of developing and managing KBE applications.

However, MOKA has several shortcomings that hinder a wider adoption of KBE technology [6]. Firstly, the application of MOKA requires the involvement of specialized knowledge engineers, whose availability is scarce, posing challenges for its practical implementation. Additionally, MOKA primarily caters to support knowledge engineers rather than domain experts or developers. Moreover, MOKA does not entirely identify knowledge representation mechanisms and supporting tools. Consequently, the methodology is perceived as too difficult and complex, especially for small teams [7, 8], and as a result it is rarely used in practice.

Currently, two primary approaches are employed in the development of KBE applications [9]. The first approach involves directly coding the engineering knowledge within the application itself and generating a separate knowledge book, using the so-called Rapid Application Development (RAD) methods. However, this creates an application that functions as a “black box” (i.e., generating output from input without providing insight into the underlying reasoning or decision-making process), making it challenging to understand how the application reaches its conclusions, which can hinder the end-user’s trust in the application and limit its acceptance. The second approach entails modeling the engineering knowledge outside the application and subsequently converting it into executable code, manually. This approach leads to a disconnection between the modeled knowledge and its implementation in the application. As a result, it becomes difficult to comprehend how the modeled knowledge has been incorporated into the application, reinforcing its black-box perception.

Based on the current KBE application development process, the following limitations can be identified:

1. The current process of encapsulating knowledge within source code for KBE applications relies heavily on manual efforts, resulting in a significant dependence on effective communication and understanding between domain experts and developers. However, this manual approach is highly susceptible to human errors. Consequently, the extensive knowledge transfer combined with identifying and rectifying the errors within the KBE application leads to long development times.
2. It is common for the source code of KBE applications to be extensive. With manual programming, the knowledge and understanding of the source code are typically limited to the original developer. Consequently, the task of tracing errors within the code, extracting the embedded knowledge rules, and ensuring compliance with requirements becomes challenging if the original developer is unavailable during project execution. This situation effectively transforms the KBE application into a “black box”, where its internal workings and reasoning processes are not transparent or accessible to other stakeholders involved in the project.
3. In case a formal knowledge model is not developed prior to (manual) code generation, permanent knowledge loss can occur if the domain expert and the programmer leave the organization, which can severely impede the transfer of knowledge from one project to another, hindering the continuity and effectiveness of knowledge (re)utilization in subsequent endeavors.

To improve the current KBE app development process, the aforementioned challenges must be addressed. Therefore, the key motivation for the work presented in this thesis is the development of a novel methodological approach for the development of KBE applications. This novel methodological approach shall improve the transparency of KBE applications and reduce their development time.

Model-Based Systems Engineering (MBSE) is a promising engineering approach that has the capacity to solve a key challenge in KBE applications: their black-box perception. MBSE uses graphical and textual models (in contrast to the document-based approach of traditional systems engineering) to support the design, analysis, verification, and validation of complex systems, throughout the system development lifecycle [10].

As discussed in the following chapters, MBSE can aid in modeling domain knowledge and requirements, facilitating knowledge capture and formalization, requirements traceability, and knowledge reuse within KBE applications. Furthermore, MBSE can solve the current problem of the disconnection between the modeled knowledge and the KBE application encoding that knowledge, thanks to its support for automatic generation of source code from knowledge models.

Consequently, the adoption of an MBSE-based KBE app development methodology holds potential to make the knowledge encoded within the applications more explicit and transparent, and reduce the development time of KBE applications. This should increase engineers' trust and acceptance of KBE applications, potentially improving the adoption of KBE technology in the industry.

The structure of this thesis report is then organized as follows. Chapters 2 and 3 provide a theoretical background on KBE and MBSE, respectively, while overviewing the state of the art in these fields and setting the foundation for the subsequent chapters. Chapter 4 presents the formulated research objective and the research questions addressed in this thesis. Chapter 5 describes the methodological approach used in this research work to develop a novel framework for developing KBE applications based on MBSE concepts. This leads into Chapter 6, where the methods used to verify and validate the developed framework are described. Subsequently, Chapter 7 presents a case study in which the developed framework was employed in the development of a real-world KBE application. Finally, conclusions and recommendations for future work are addressed in Chapter 8.

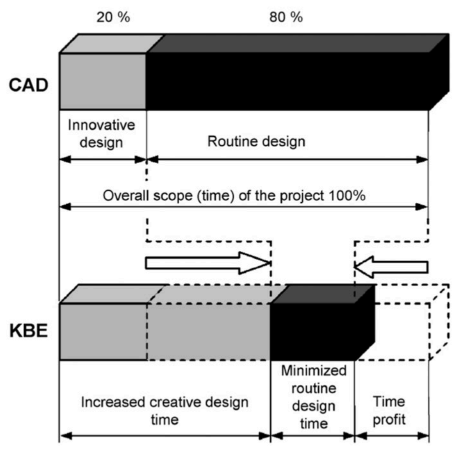
2 Knowledge Based Engineering

This chapter provides essential theoretical background relevant to understand the need for the research work carried out. It begins by introducing the concept of Knowledge Based Engineering and highlighting its advantages. Subsequently, an analysis of the state of the art in the development of KBE applications is presented, followed by an identification of the existing limitations within the field.

2.1. What is KBE?

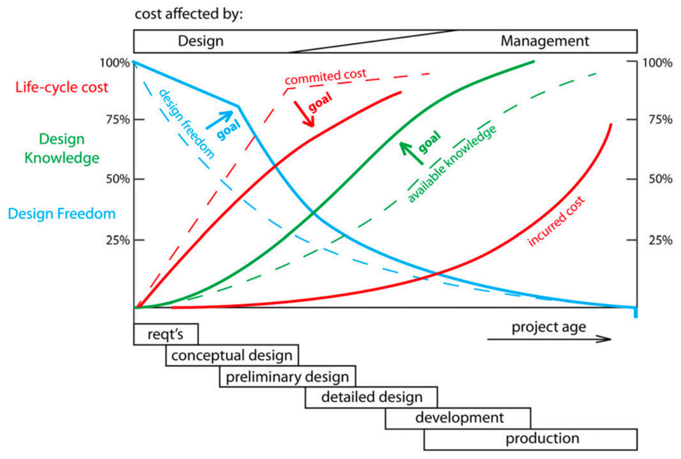
Knowledge Based Engineering is defined, according to La Rocca [4], as "*a technology based on the use of dedicated software tools called KBE systems, which are able to capture and reuse product and process engineering knowledge, with the final goal of reducing time and costs of product development by means of [...] automation of repetitive and non-creative design tasks [and] support of multidisciplinary design optimization in all the phases of the design process*" (p. 161).

There are several arguments for adopting Knowledge Based Engineering. Most of these arguments are based on the opportunities offered by KBE in the formalization of knowledge and in the automation of design in the conceptual and preliminary design phases [6]. A major advantage in adopting KBE is the considerable reduction in time and costs in the product development process, while allowing more time for innovative work, thanks to its automation of repetitive and non-creative design tasks that can make up to 80% of the typical design process [5, 11, 12]. Moreover, the increased time for creative design, provided by the adoption of KBE, allows the engineers to explore a larger part of the design envelope, which is essential before making decisions with a high impact on committed costs, thus allowing for a lower percentage of committed cost at the end of the preliminary design phase. These advantages are illustrated in Figure 2.1. Various examples where KBE has been successfully deployed with benefits in time savings and cost reduction, in different domains, can be found in the literature [5, 13–15].



20 %
80 %
CAD
Innovative design
Routine design
Overall scope (time) of the project 100%
KBE
Increased creative design time
Minimized routine design time
Time profit

(a) Achievable reduction in design time. Source: [16]



(b) Achievable reduction in committed cost. Source: [6]

Figure 2.1: Advantages of adopting KBE on the product design process

KBE applications are key enablers towards the automation of complex systems design as they allow users to exploit the object-oriented programming (OOP[1]) paradigm, provide runtime caching[2] and dependency tracking[3] capabilities, demand-driven evaluation[4], and have a tight integration with a Computer-Aided Design (CAD) kernel [17]. KBE systems allow capturing engineering knowledge into powerful automated solutions for product configuration, design space exploration and multidisciplinary design optimization [17].

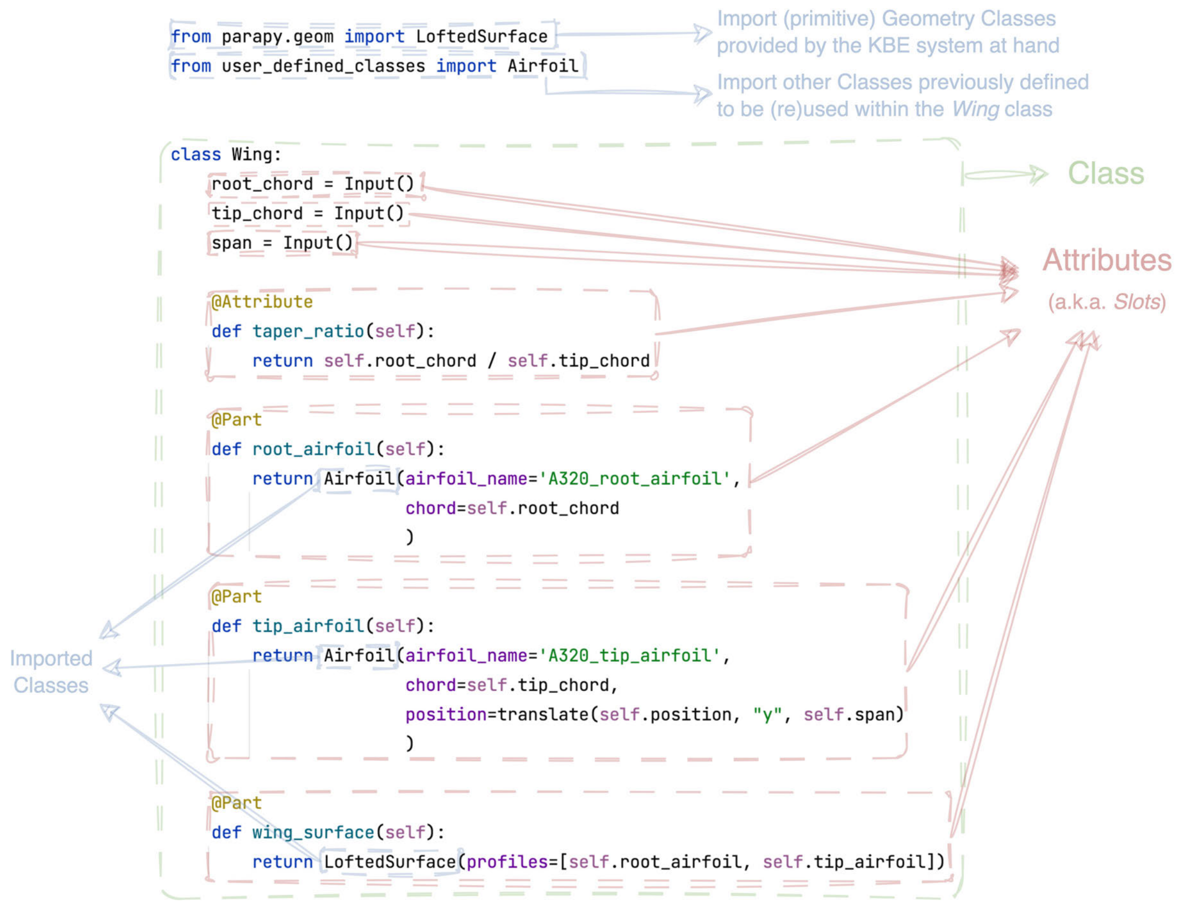
Following the principles of OOP, KBE applications are predominantly composed of class declarations that conform to a standardized structure for defining generic object types. These classes encompass attributes that provide detailed specifications of their characteristics. This standardized structure holds significant value for the purpose of automating the generation of source code for KBE applications (this capability will be further explored in this research - see Section 5.4). A snippet of the source code from a simplistic KBE application used for wing design is presented in Figure 2.2.

[1]A software development approach that organizes data and behavior into reusable entities called objects. This paradigm promotes code reusability, modularity, and extensibility, making it suitable for building complex and scalable software systems.

[2]The capability to store and retain computed results during runtime, commonly known as caching, allowing for subsequent reuse without the need for redundant re-computation, unless necessary.

[3]The capability of continuously monitoring the current validity of cached values, invalidating them once they are deemed invalid, and re-computing them selectively, only if they are requested again.

[4]A computational approach where the evaluation of expressions or computations is deferred until their results are explicitly requested by the user, or indirectly requested while trying to satisfy a user demand.



```
from parapy.geom import LoftedSurface
from user_defined_classes import Airfoil

class Wing:
    root_chord = Input()
    tip_chord = Input()
    span = Input()

    @Attribute
    def taper_ratio(self):
        return self.root_chord / self.tip_chord

    @Part
    def root_airfoil(self):
        return Airfoil(airfoil_name='A320_root_airfoil',
                       chord=self.root_chord
                       )

    @Part
    def tip_airfoil(self):
        return Airfoil(airfoil_name='A320_tip_airfoil',
                       chord=self.tip_chord,
                       position=translate(self.position, "y", self.span)
                       )

    @Part
    def wing_surface(self):
        return LoftedSurface(profiles=[self.root_airfoil, self.tip_airfoil])
```

Import (primitive) Geometry Classes provided by the KBE system at hand
Import other Classes previously defined to be (re)used within the *Wing* class
Class
Attributes
(a.k.a. *Slots*)
Imported Classes

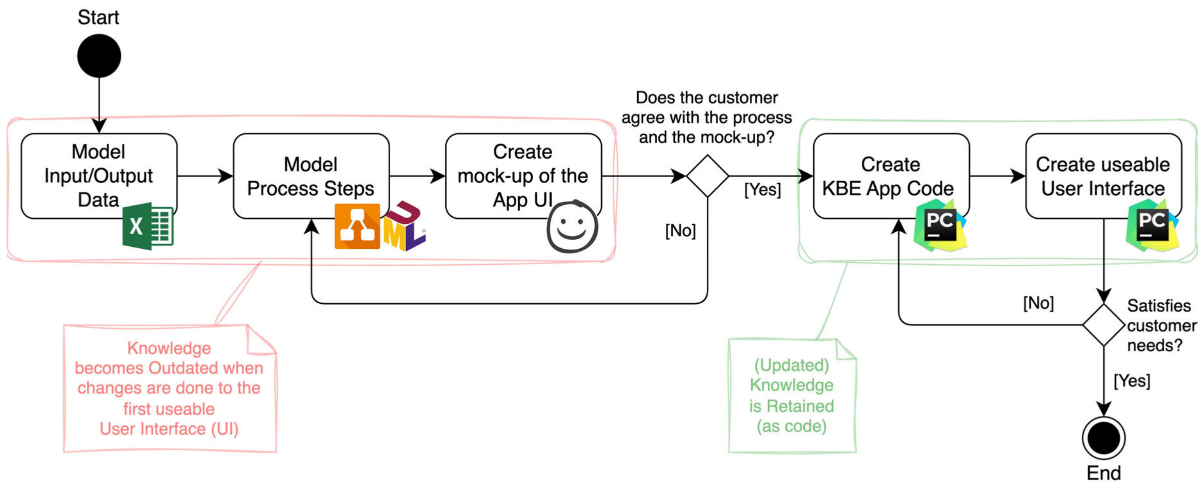
Figure 2.2: Snippet of the Source Code from a simplistic KBE Application used for Wing design. The standardized structure of KBE apps is highlighted by the green and red boxes, representing the definition of the class and its attributes, respectively.

The process of developing a KBE application involves acquiring the necessary product and process knowledge from domain experts, and subsequently translating it into executable code, using the programming language supported by the specific KBE system at hand. This is a complex process that requires expertise both in knowledge acquisition and modeling, as well as in code generation and verification, and involves communication among several stakeholders: the domain expert, the knowledge engineer, the developer, the project lead, and the end-user. The domain expert has the knowledge required to develop a KBE application, usually it's an engineer specialized in a certain field which typically has no experience in developing KBE applications. The knowledge engineer is a specialized engineer that captures and formalizes the knowledge required to develop the KBE application. The developer is the engineer that reads the formal models created by the knowledge engineer and develops the KBE application. The project lead is the engineer that oversees the development of the KBE application. The end-user is the engineer that uses the KBE application after it has been developed, often the domain expert is also the end-user.

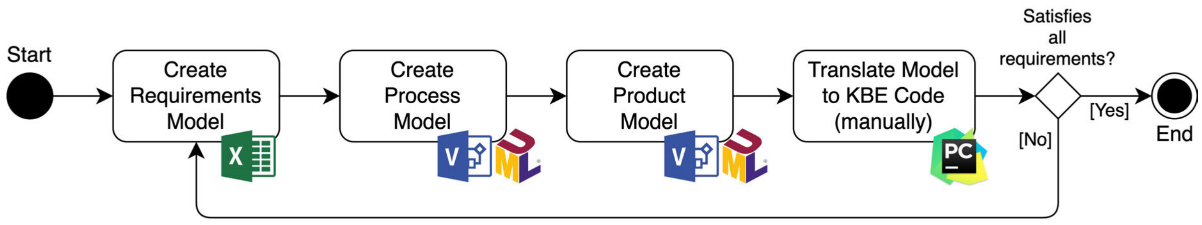
2.2. State of the Art in KBE App Development

Over the years, numerous KBE platforms have emerged, employing distinct programming languages. Nevertheless, a common characteristic among these platforms is their utilization of an OOP language. However, one drawback of KBE platforms is that they often employ either a difficult to learn or platform-specific programming language, posing a limitation. A significant advancement in the realm of KBE languages was introduced in 2016 by ParaPy[5], which leverages the widely adopted Python programming language, enhancing the appeal and accessibility of the platform.

At present, development teams involved in KBE application development employ a diverse array of tools for capturing requirements, desired functionalities, architectural design, and the user interface of a KBE application. However, the utilization of multiple tools hampers the ability of these teams to collaboratively create and share a unified and cohesive application design among themselves and with the customer. Consequently, this results in the development of applications that do not align with customer needs, necessitating substantial modifications after the initial software release. Furthermore, the absence of a structured approach during the codification phase creates time constraints and adds pressure on the development timeline. The KBE application development process currently employed at ParaPy is schematically represented in Figure 2.3a. Additionally, the current KBE app development process employed at GKN Aerospace Fokker Elmo[6] for developing ParaPy-based KBE applications is schematically represented in Figure 2.3b.


Start
Model Input/Output Data
Model Process Steps
Create mock-up of the App UI
Does the customer agree with the process and the mock-up?
[Yes]
[No]
Create KBE App Code
Create useable User Interface
[No]
Satisfies customer needs?
[Yes]
End
Knowledge becomes Outdated when changes are done to the first useable User Interface (UI)
(Updated) Knowledge is Retained (as code)

(a) Current KBE App Development Process at ParaPy


Start
Create Requirements Model
Create Process Model
Create Product Model
Translate Model to KBE Code (manually)
Satisfies all requirements?
[Yes]
[No]
End

(b) Current KBE App Development Process at GKN Aerospace Fokker Elmo

Figure 2.3: Examples of Current KBE App Development Processes. The icons indicate the tool used to perform each task. The tasks are performed by a team of domain experts and KBE developers. Note that the figures shows a generalization of the tasks that are usually performed, but often the order of these tasks can be interchanged depending on the project and the expertise of the people involved, which can make the overall knowledge acquisition process less structured/consistent.

The current practice in the development of KBE applications at ParaPy typically consists of the following tasks:

1. The input and output (I/O) data of the application is modeled in Microsoft Excel[7].

[5]`https://parapy.nl` (Accessed: 15/06/2023)
[6]https://www.gknaerospace.com/en/about-us/fokker-technologies/ (Accessed: 15/06/2023)
[7]`https://www.microsoft.com/en-us/microsoft-365/excel` (Accessed: 15/06/2023)

2. The process steps are modeled in draw.io[8] using UML[9] activity diagrams.
3. A mock-up of the user interface (UI) of the application is created in Balsamiq[10]. This task is performed as an extra means to elicit customer requirements (e.g., functionality, I/O management, user experience), however a model of the app's requirements is not created.
4. The knowledge required to develop the KBE application is directly implemented in the KBE application code (manually) using the PyCharm[11] integrated development environment (IDE).

The current practice in the development of KBE applications at GKN Aerospace Fokker Elmo typically consists of the following tasks:

1. The requirements of the application are modeled in Microsoft Excel.
2. A model of the process steps is created in Microsoft Visio[12] using UML activity diagrams.
3. A model of the product features is created in Microsoft Visio using a combination of UML class diagrams and composite structure diagrams.
4. The generated product model is manually translated to KBE code using the PyCharm IDE.

Figure 2.3 helps highlight the major shortcomings of the current KBE app development process, namely: captured knowledge becoming outdated, multiple tools needed (leading to a steep learning curve), no consistency between knowledge models, duplication of work (in the form of knowledge modeling and coding), and lack of overview.

Currently, there is a need to develop a framework capable of addressing the aforementioned challenges, in order to enhance the efficiency of designing KBE applications, while offering a structured approach to the knowledge acquisition phase. This framework should allow developers/users to (formally) capture the different aspects of an application within a unified environment, alleviating the existing issue of fragmentation. Furthermore, it should expedite the codification process through automatic code generation and, additionally, facilitate review by users who lack expertise in the KBE language.

To date, a suitable development methodology to effectively support the development of KBE applications is lacking, which has considerable implications on the time required for application development, as well as the quality of the applications in terms of requirement compliance, traceability, maintainability, scalability, and (eventually) the ability to preserve and efficiently reuse engineering knowledge.

2.2.1. Current Practice in Knowledge Modeling

There are several methodologies available in the literature that attempt to formalize and improve the KBE application development process. Examples of these methodologies are CommonKADS [18], DEKLARE [19], KOMPRESSA [20], KCM [21], KNOMAD [22], and MOKA [5]. Although several methodologies exist, two of the most well-know, developed as part of European Union projects, are the Common Knowledge Acquisition and Design Support (CommonKADS), and the Methodology and tools Oriented to Knowledge-based engineering Applications (MOKA). However, CommonKADS lacks specialization to engineering design, focusing on the organizational aspect of Knowledge Based Systems (KBS) development and the communication and interrelationships between the different agents involved, making it a generic-purposed framework [23]. Thus, the most accepted and established methodology for KBE app development appears to be MOKA, with several research works attempting

[8]`https://app.diagrams.net` (Accessed: 15/06/2023)
[9]Unified Modeling Language - `http://www.uml.org` (Accessed: 15/06/2023)
[10]`https://balsamiq.com` (Accessed: 15/06/2023)
[11]`https://www.jetbrains.com/pycharm/` (Accessed: 15/06/2023)
[12]`https://www.microsoft.com/en-ww/microsoft-365/visio/flowchart-software` (Accessed: 15/06/2023)

to extend this methodology [24–26].

The primary objectives of the MOKA research project[13] were to reduce the lead times and costs of developing KBE applications, to provide a consistent way of developing and maintaining KBE applications, and to develop a methodology which would form the basis of an international standard [27]. One of the main achievements of the MOKA research project was the identification of the typical KBE life cycle, which is presented in Figure 2.4.

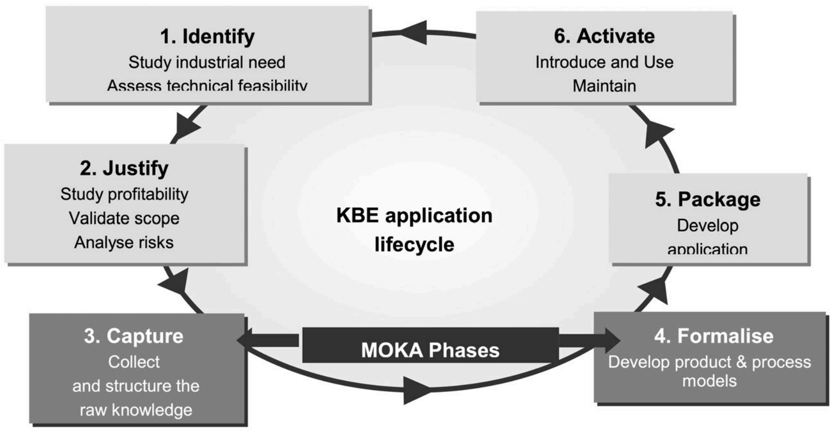


Figure 2.4: The KBE life cycle identified in the MOKA project.
Source: [27]

The MOKA methodology focuses mainly on the *Capture* and *Formalise* phases of the KBE application life cycle. It deliberately avoids the *Package* phase in order to preserve MOKA's neutrality to KBE platforms [27].

The MOKA model resulting from the *Capture* and *Formalise* phases is composed of three submodels, as shown in Figure 2.5. It involves the generation of an informal knowledge model, followed by the formulation of a formal knowledge model, which is then stored in a neutral language knowledge model [28]. Finally, the neutral language knowledge model is converted to KBE application code.

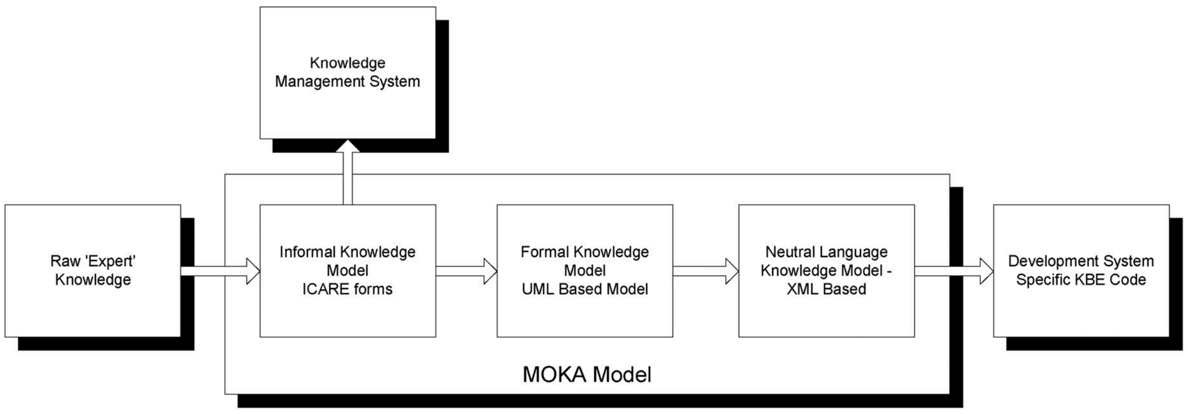
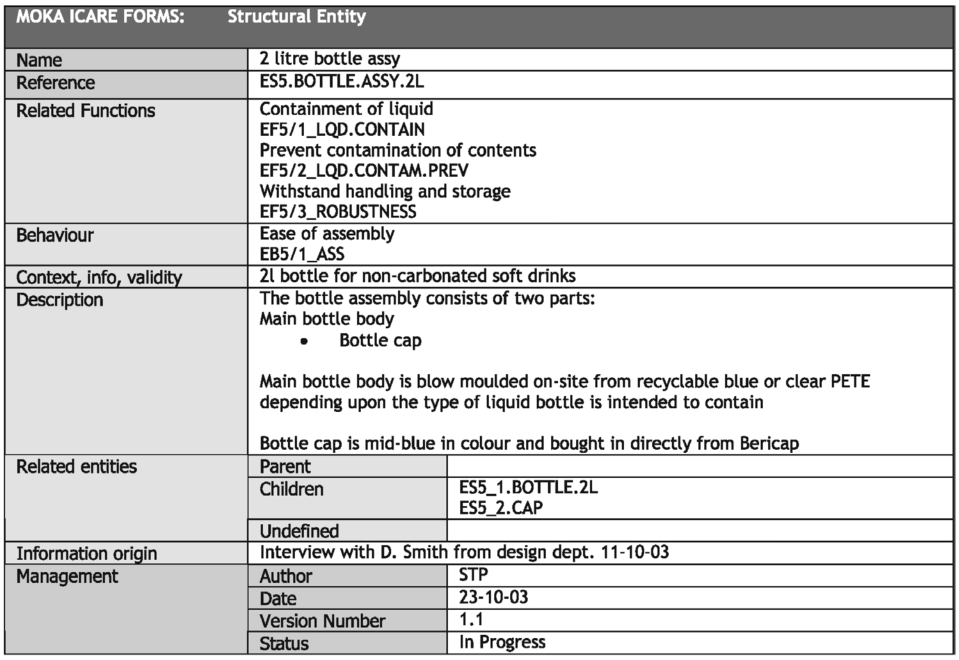


Figure 2.5: MOKA knowledge models.
Source: [28]

The informal knowledge model is structured upon a set of forms called "ICARE forms". The acronym ICARE stands for Illustration, Constraints, Activities, Rules, and Entities, representing the 5 different

[13]MOKA European project (No. 25418) conducted within the ESPRIT-IV framework

kinds of forms that can be used to capture and structure engineering knowledge. An example of an Entity form is shown in Figure 2.6. These forms allow the structuring of product and process knowledge at an informal level that is easily understood by both the knowledge engineer and the domain expert, which is essential since the domain expert must be able to validate the knowledge before it can be formalized in the next phase [24]. However, the application of the ICARE forms for informal knowledge modeling may not always be intuitive to domain experts, who primarily work in terms of requirements, processes, and products. This is why the assistance of a knowledge engineer is often required during MOKA's knowledge capture phase. Domain experts perceive the conversion of their knowledge into the ICARE format as an additional pedagogical endeavor, which may not yield substantial returns in terms of the invested time.



MOKA ICARE FORMS:	Structural Entity	
Name	2 litre bottle assy	
Reference	ES5.BOTTLE.ASSY.2L	
Related Functions	Containment of liquid EF5/1_LQD.CONTAIN Prevent contamination of contents EF5/2_LQD.CONTAM.PREV Withstand handling and storage EF5/3_ROBUSTNESS	
Behaviour	Ease of assembly EB5/1_ASS	
Context, info, validity	2l bottle for non-carbonated soft drinks	
Description	The bottle assembly consists of two parts: Main bottle body • Bottle cap Main bottle body is blow moulded on-site from recyclable blue or clear PETE depending upon the type of liquid bottle is intended to contain Bottle cap is mid-blue in colour and bought in directly from Bericap	
Related entities	Parent	
	Children	ES5_1.BOTTLE.2L ES5_2.CAP
	Undefined	
Information origin	Interview with D. Smith from design dept. 11-10-03	
Management	Author	STP
	Date	23-10-03
	Version Number	1.1
	Status	In Progress

Figure 2.6: Example of MOKA (ICARE) Entity Form of a bottle assembly.
Source: [5]

The subsequent stage involves utilizing the informal knowledge model to develop a formal knowledge model, comprising of two parts: the Product Model and the Design Process Model. The Product Model is used to represent the object-level knowledge within the domain, encompassing various aspects such as structures, functions, behaviors, and geometry, along with their associated attributes, relations, and constraints. On the other hand, the Design Process Model is used to represent the problem-solving activities, control knowledge, and the interconnections to the Product Model [27]. A general example of these models is presented in Figure 2.7. In this stage, the information from the ICARE forms is manually converted by a knowledge engineer into a formal model notation. The MOKA Modelling Language (MML), an extension of UML, is employed to represent the knowledge in a format that is both understandable by knowledge engineers and software engineers, and is suitable for the subsequent development of the KBE application [6].

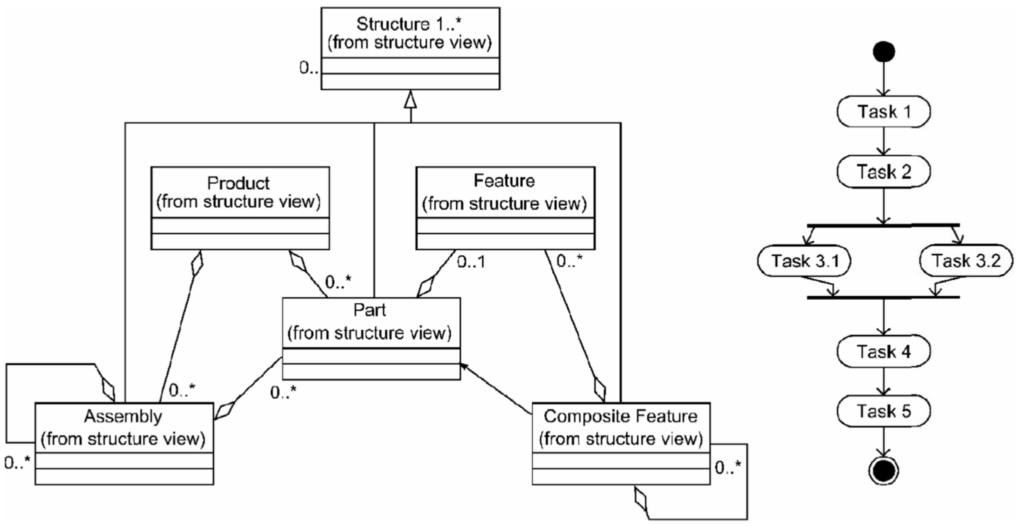


Figure 2.7: Example of MML Product Model (left) and Design Process Model (right).
Source: [9]

Finally, the formal model is converted to a neutral language knowledge model, providing a crucial link between the MML formal model and the subsequent KBE application code. The objective is to develop a model which can readily be translated - possibly with a high degree of automation - into the KBE development platform source code [28]. MOKA recommends the use of Extensible Markup Language (XML) in order to build the neutral model. However, due to its aim of being a system-independent methodology, MOKA does not provide a detailed mechanism to create this model [9, 28]. This is one of the key shortcomings of MOKA.

2.2.2. Current Practice in KBE Code Generation

Since MOKA does not specify how to create the neutral language knowledge model nor provides any automation scripts or software programs to assist (even partially) in generating the code, developers are responsible for manually converting the formal knowledge model into source code. This means that the developer has complete discretion over the architecture of the code and the organization of rules and requirements within it. However, developers can often use an implicit programming style, which may be difficult for other developers and domain experts to understand, making it challenging to ensure that all knowledge rules and requirements are complied with. Additionally, the manual conversion of the formal model to source code can result in human errors. Furthermore, this leads to duplication of work in the form of knowledge modeling and coding, resulting in a disconnection between the knowledge that has been captured in the formal model and the knowledge that has been encoded in the KBE application. All these factors can ultimately lead to (significant) knowledge losses, particularly if the developer of the original source code is unavailable for consultation during the project where the KBE application must be executed.

To address the disconnection between knowledge models and KBE application code, an effective approach is to utilize tools that automatically generate source code from knowledge models. Several examples of such tools are available in the literature.

For instance, Dewitte [29] developed an online platform that enables the creation of UML-like class diagrams and the automatic generation of code for each element in the diagrams. This platform specifically targets the Genworks GDL[14] KBE system. However, a limitation of this platform is that it only allows to model product-related knowledge, overlooking the need to model also requirements and process-related knowledge.

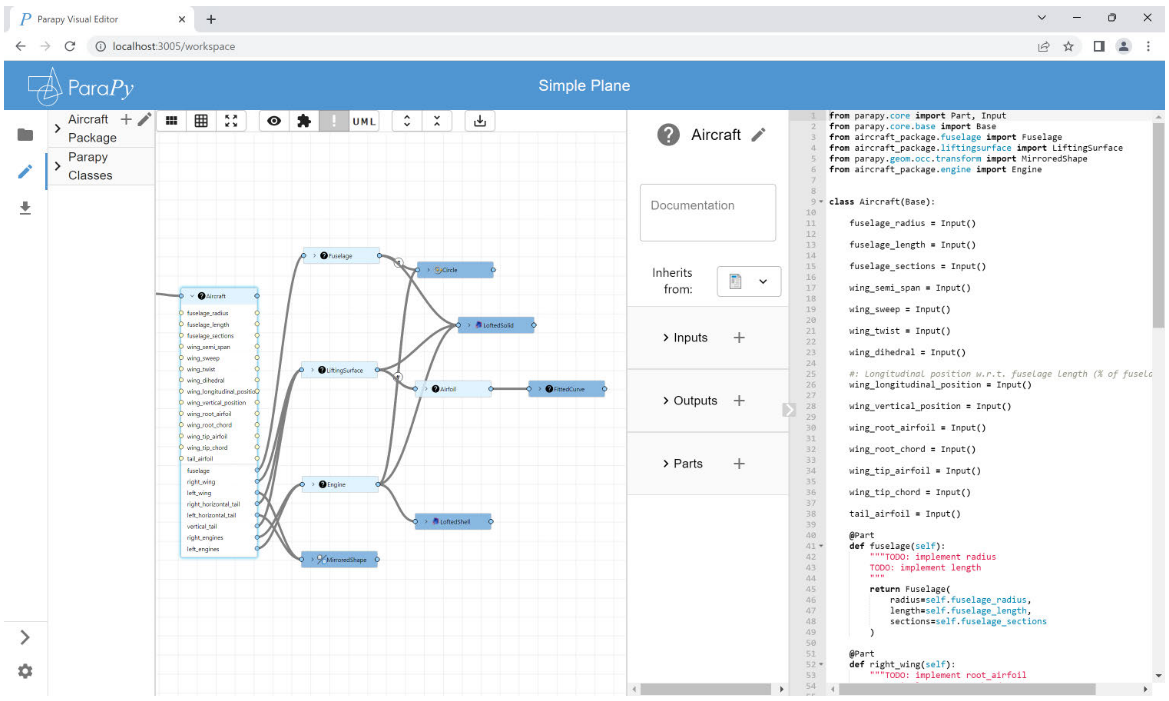
Similarly, industry examples exist, such as ParaPy's Visual Editor tool, depicted in Figure 2.8. The Visual Editor allows the use of UML-like class diagrams to create knowledge models. The knowledge models are then stored in a JSON[15] format from which application source code is automatically generated for the ParaPy KBE system, using a dedicated translation engine. However, this tool exhibits shortcomings that impede its effectiveness in addressing the present limitations of the KBE app development process.

For example, ParaPy's Visual Editor, only allows for the modeling of product knowledge, overlooking the modeling of requirements and processes. As a result, multiple tools are still needed to capture the various aspects of the knowledge required for the development of KBE applications.

Additionally, although the model can be exported to UML class diagrams, there is no support for exporting it to an industry-standard neutral language knowledge model. Consequently, the tool lacks system-independence and restricts direct conversion to ParaPy KBE code, potentially limiting code generation for other KBE languages.

Furthermore, while the modeling process is relatively straightforward to learn, it relies on a custom modeling language blending UML-like class diagrams and composite structure diagram features. Utilizing an industry-standard modeling language may be more appropriate for this purpose in order to facilitate industry adoption of the tool.

Moreover, the tool confines knowledge modeling to a single drawing canvas, limiting model organization, particularly in complex KBE applications.



```
from parapy.core import Part, Input
from parapy.core.base import Base
from aircraft_package.fuselage import Fuselage
from aircraft_package.liftingsurface import LiftingSurface
from parapy.geom.occ.transform import MirroredShape
from aircraft_package.engine import Engine


class Aircraft(Base):

    fuselage_radius = Input()

    fuselage_length = Input()

    fuselage_sections = Input()

    wing_semi_span = Input()

    wing_sweep = Input()

    wing_twist = Input()

    wing_dihedral = Input()

    #: Longitudinal position w.r.t. fuselage length (% of fusela
    wing_longitudinal_position = Input()

    wing_vertical_position = Input()

    wing_root_airfoil = Input()

    wing_root_chord = Input()

    wing_tip_airfoil = Input()

    wing_tip_chord = Input()

    tail_airfoil = Input()

    @Part
    def fuselage(self):
        """TODO: implement radius
        TODO: implement length
        """
        return Fuselage(
            radius=self.fuselage_radius,
            length=self.fuselage_length,
            sections=self.fuselage_sections
        )

    @Part
    def right_wing(self):
        """TODO: implement root_airfoil
```

Figure 2.8: ParaPy's Visual Editor showcasing the (product) model of a KBE application used for designing Simple Airplanes. The central area depicts the drawing canvas where the model is constructed. On the right, the KBE code automatically generated from the model is presented.

[14]`https://www.genworks.com` (Accessed: 15/06/2023)
[15]`https://www.json.org/json-en.html` (Accessed: 15/06/2023)

2.2.3. Stakeholder's Trust in KBE applications

The KBE application development process can be categorized into three main scenarios. Each of these will be discussed in the following paragraphs.

The first scenario is the one described in the MOKA methodology, where there is a knowledge engineer available to structure and formalize the knowledge from the domain expert (which usually has no coding experience) into models, so that it's understandable by the application developer. This scenario is more common in large companies that have already been using KBE for some time, and it is schematically represented in Figure 2.9.

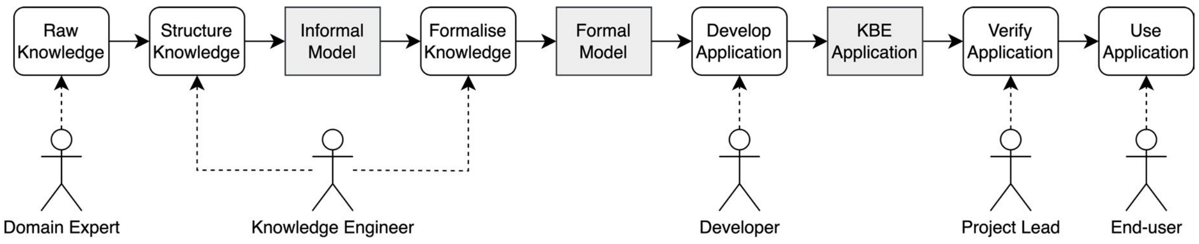


Figure 2.9: Current steps of scenario 1 in the KBE app development process.
Source: Adapted from [25]

In this scenario, the domain expert is expected to populate a knowledge base to develop an application. However, the domain expert is not able to understand how the knowledge has been implemented in the KBE application (i.e., the expert is unable to trace where the rules and requirements are placed), resulting in a lack of trust in the KBE application [30]. Moreover, the lack of transparency in KBE applications gives the domain expert a feeling of exclusion, causing insufficient participation in the development of a KBE application which could hinder adoption of KBE technology [31].

The second and third scenarios are more common in Small and Medium-sized Enterprises (SMEs), or in conceptual design projects (having small design teams and short lead-time), where often a single engineer fulfils multiple roles (e.g., as a domain expert and as a developer), and there is no specialist knowledge engineer available to structure and formalize the knowledge from the domain expert into models.

The second scenario is the one where the domain expert has little to no experience in coding (software engineering). In this case, the domain expert provides the required knowledge for the KBE application to the developer via a set of unstructured informal models/documents. This scenario is schematically represented in Figure 2.10.

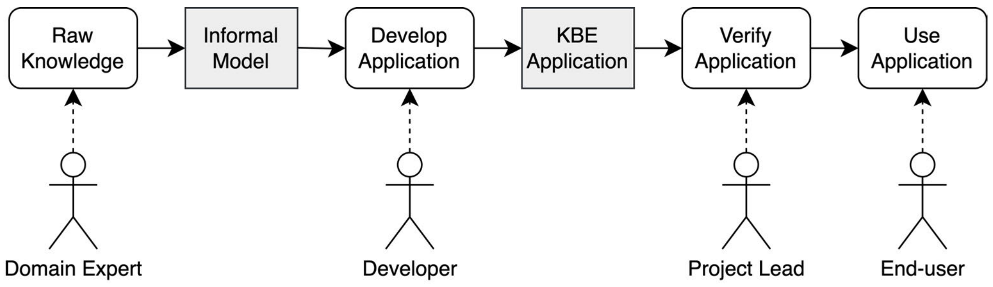


Figure 2.10: Current steps of scenario 2 in the KBE app development process

Here again, the domain expert faces the same challenges as in case of scenario 1. Furthermore, the developer has additional challenges as there are no formal models available. The lack of an overarching architectural view can hinder developers' efficiency and effective collaboration with their colleagues, as a clear overview of the interactions between different parts of the KBE application is not provided. Additionally, understanding a colleague's application code can be challenging as most of the knowledge embedded in the code is represented by formulas and data without clear context [32], making it difficult to process. Moreover, the lack of formal models to compare the code with can further aggravate the black-box perception. Without any formal model, the developer will be prone to making errors in case the comprehension from the informal model is not correct.

The third scenario is similar to the second scenario. However, in this case, the domain expert also has experience in coding (software engineering) and therefore is also the developer of the KBE application. In this case, the domain expert usually skips the knowledge modeling step and implements the required knowledge directly into the KBE application. This scenario is schematically represented in Figure 2.11.

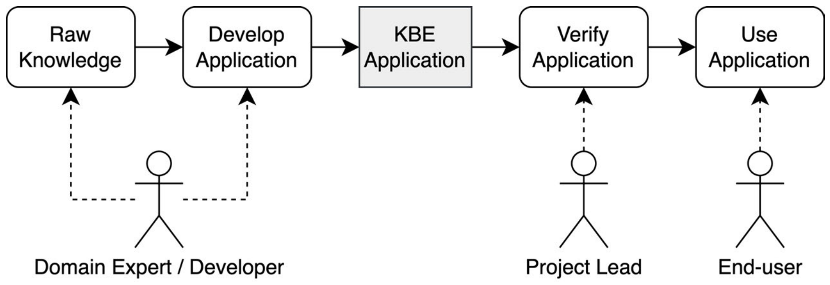


Figure 2.11: Current steps of scenario 3 in the KBE app development process

Here, if the domain expert leaves the company (or is unavailable for consultation in the project), the knowledge contained in the expert is lost since there are no structured and formalized models of the knowledge used to develop the KBE application. Moreover, other domain experts cannot understand the knowledge written as application code, which is also hard to find due to the usually large amount of application code, enforcing an application with a black-box behavior [25].

This is the most acute problem when one engineer takes on multiple roles of developer, knowledge engineer and domain expert. Furthermore, motivating the domain expert/developer to formalize their knowledge into a model is challenging because they do not find directly any returns on the time invested in formalization.

In all the scenarios mentioned above, the project lead has to verify that the developed application complies with the requirements. However, due to the black-box nature of the source code, it becomes challenging for the project lead to determine whether the knowledge has been implemented correctly, or if the application is generating the intended output. The only way to verify if the application is functioning as required is by running it and testing with some values. However, even if the application produces the desired outcome, it does not necessarily mean that it has done so by deriving the correct information. This makes it extremely challenging to validate the application, and in turn, challenging to convince the customer that it is performing as intended [25].

The end-user is the stakeholder that uses the KBE application to automate (part of) the design process. Often, the domain expert is also the end-user of the KBE application. Thus, depending on the scenario, the end-user's trust on the code might vary. In scenarios 1 and 2, the end-user might have a lack of trust in the application as they are not able to determine how the application is computing the results. Here, the application just "magically" computes an output from some inputs [25]. In such cases, adoption of KBE applications in the design process becomes challenging. This is also the case if the end-user is not necessarily the domain expert. In scenario 3, there is no trust-deficit in the developed application. However, the maintainability and scalability of the code in the absence of the original developer becomes challenging.

2.3. Limitations of the Current KBE App Development Process

The main issues resulting from the current practice in the development of KBE applications can be found in the literature [6]. Based on the current KBE application development process, the main limitations can be summarized as follows:

Case-based, ad hoc development

Due to the shortcomings of existing KBE app development methodologies, such as MOKA's, rather than employing a structured methodology to develop KBE applications, the development of these apps is mostly case-based and *ad hoc*. Developers tend to improvise KBE solutions based on custom development processes, without formally modeling the knowledge before (manually) generating the application source-code. This is a serious problem if the domain expert and the developer leave the organization, since it can lead to permanent knowledge loss, misuse and under-utilization, as well as higher maintenance costs due to the non-standard development of the application.

Black-box Perception

In general, KBE applications tend to have a large source-code. When the source-code is manually developed, the information contained within the code is known only to the developer. Consequently, detecting errors in the code, extracting the embedded knowledge rules, and ensuring that the KBE application complies with requirements becomes challenging in the absence of the original developer. Therefore, KBE applications tend to acquire a black-box perception. One of the main reasons that further enforces this perception is that, in the traditional KBE practice, the information models are contained within the applications and, most of the times, the captured knowledge is represented as context-less data and formulas. Although this approach is effective for the fast development of *ad hoc* KBE applications, it hinders the transparency of the applications and the traceability of requirements, which is problematic for the validation of the applications, as well as the long-term maintenance, sharing and reuse of the underlying knowledge.

Knowledge Reuse

The previously presented shortcomings are closely tied with the difficulty of reusing knowledge in KBE systems since case-based black-box applications are not well suited for knowledge reuse. Another factor that further hinders knowledge reuse is the lack of data exchange standards for KBE in order to facilitate the interoperability between different KBE applications and platforms.

Longer Development Times

The current process of incorporating knowledge into source code for KBE applications predominantly involves manual efforts, relying on effective communication and understanding between domain experts and developers. However, this manual approach is prone to human errors, resulting in prolonged development times due to the need for extensive knowledge transfer and error identification and rectification within the KBE application.

Additionally, as noted by Baxter et al. [33], approximately 20% of a designer's time is dedicated to searching for and assimilating information, and 40% of all design requirements are satisfied by personal stores, despite the possible existence of more appropriate information sources. This indicates that design knowledge is not available in a shared and easily accessible knowledge repository. In such situations, reusing knowledge with the help of an established KBE app development framework could lead to substantial savings in time and effort. The lack of knowledge reuse in KBE systems contributes to a longer development time of applications.

Lack of Supporting Tools

Verhagen et al.[6] identify key issues and under-developed areas of research that could help move from case-based to methodology-guided development approaches, namely, the low availability of supporting tools and technologies for KBE implementation and translators to support the automatic conversion of formal models into KBE application code.

Although commercial tools like Visual Paradigm[16] exist for converting UML models into source code, they are currently limited to traditional programming languages and do not support KBE languages. On the other hand, tools like ParaPy's Visual Editor enable the conversion of models into KBE source code, but they have their own limitations. For instance, these tools lack the capability to fully capture the knowledge necessary for developing KBE applications, as they only focus on modeling product-related knowledge while neglecting the importance of modeling requirements and process-related knowledge.

[16]`https://www.visual-paradigm.com` (Accessed: 16/06/2023)

3 Model-Based Systems Engineering

This chapter introduces Model-Based Systems Engineering, addressing its key concepts, benefits, and challenges, while giving an outlook on the state of the art in this field. The presented theoretical background on MBSE is essential to understand the rationale behind the decision-making process during the development of the novel framework for KBE app development.

3.1. What is MBSE?

Model-Based Systems Engineering is defined, according to the International Council on Systems Engineering (INCOSE) [34], as "*the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases*" (p. 15). MBSE emphasizes the usage of digital models, in contrast to the document-based approach of traditional systems engineering.

The importance of MBSE in modern engineering is driven by the growing complexity of systems in various domains, such as aerospace, defense, automotive, healthcare, and others. These systems often consist of thousands of components and interact with other systems, making it difficult to manage their design, implementation, and operation. MBSE provides a structured and formalized approach to systems engineering, which can help engineers manage this complexity, ensuring that the system will meet all requirements.

In MBSE, a system is represented as a set of interrelated (digital) models that capture different aspects of the system's behavior, structure, and requirements [34]. A model is a simplified representation of a system, which captures its essential features and interactions in a formalized way. The model can be viewed in the form of diagrams, tables, or reports generated by querying the model repository, enabling understanding and analysis of different aspects of the same system model. In this approach, documents can still serve as a means for reporting information, however the information contained in the documentation is generated from the model. The model provides more precise control of the information than is available in a document-based approach, where the information may be spread across various documents and the relationships may not be explicitly defined [10].

The key aspects captured by the system model are in fact the same key concepts that are relevant when dealing with the development of KBE applications (Requirements, Behavior/Process, and Structure/Product), and this relationship is an essential factor for considering the possibility of using an MBSE approach for the development of KBE applications.

3.2. Benefits of MBSE

MBSE offers several benefits for engineering complex systems over traditional document-based systems engineering. Some of these benefits include [10, 35]:

- Improved Communication - MBSE emphasizes the use of models to represent system components and their interactions, which can improve communication among the team, and other stakeholders who may have different backgrounds and perspectives. Models provide a common language for system design, which can reduce misunderstandings and errors.
- Enhanced System Understanding - By allowing to capture and manage complex system requirements, which are often difficult to express in natural language, and providing more rigorous traceability between requirements, design, analysis, and testing, MBSE can help engineers better understand the system being designed.
- Reduced Errors - By using models to represent system components and their interactions, engineers can identify errors and inconsistencies early on, reducing the cost and risk of system development.
- Increased Efficiency - MBSE can lead to increased efficiency in system development, as models can be reused across multiple projects and used to automate certain design and testing tasks.
- Improved System Design - MBSE allows engineers to explore different design alternatives and evaluate the impact of design decisions on the system as a whole. This can lead to better designs and more efficient use of resources.

3.3. Challenges of MBSE

There are several challenges that organizations may face when implementing MBSE that must be addressed to effectively implement the approach. Some of these challenges include [10, 36]:

- Paradigm Shift - MBSE requires a shift in mindset from traditional document-based processes to a model-based approach. This can be difficult for some organizations that are used to traditional methods and may not have the necessary training or expertise in modeling languages.
- Tool Selection and Integration - The process of selecting and integrating tools to support MBSE can be complex. Different tools may use different modeling languages and formats, posing challenges when attempting to integrate them with existing tools and workflows. The lack of standardization in modeling tools and practices can lead to interoperability issues and make it difficult to share models across different organizations and domains. Organizations may need to invest in new tools and ensure that they are compatible with existing ones to ensure seamless integration.
- Modeling Complexity - MBSE can be a complex process, and the models used to represent the system can be equally complex. This can make it difficult to manage and understand the models,

and can lead to errors or omissions in the model. Organizations need to invest in proper modeling techniques and methodologies to ensure that the models are accurate and reliable.

- Cost and Resources - Implementing MBSE can be costly and requires considerable resources, including expertise in modeling languages, tools, and methodologies. Organizations need to invest in training and hiring staff with the necessary expertise to ensure successful implementation.
- Adoption and Acceptance - Finally, MBSE adoption and acceptance can be a challenge. Some stakeholders may be resistant to change and may not see the value of MBSE. Organizations need to invest in stakeholder engagement and education to ensure that all stakeholders understand the benefits of MBSE and are willing to adopt it.

3.4. Key Concepts in MBSE

In order to implement Model-Based Systems Engineering a modeling language, tool, and method are necessary. These concepts are described in the following sections.

3.4.1. Modeling Language

A modeling language is a formal (graphical) language used to describe, specify, and represent different aspects of a system. Modeling languages can be used to create models that capture system requirements, behavior, structure, and other important aspects of the system. The purpose of a modeling language is to provide a standardized way for stakeholders to communicate about the system being modeled. Modeling languages allow stakeholders to create models that can be easily understood by others, regardless of their background or expertise. This enables stakeholders to collaborate more effectively and make better decisions about the system being modeled.

There are several modeling languages used in MBSE, each with its own strengths and weaknesses. Some of the most commonly used modeling languages in MBSE include SysML[1] (Systems Modeling Language), UML (Unified Modeling Language), and BPMN[2] (Business Process Model and Notation). SysML is designed specifically for systems engineering, while UML is more general-purpose and widely used in software engineering, and BPMN is used to model business processes and workflows.

SysML

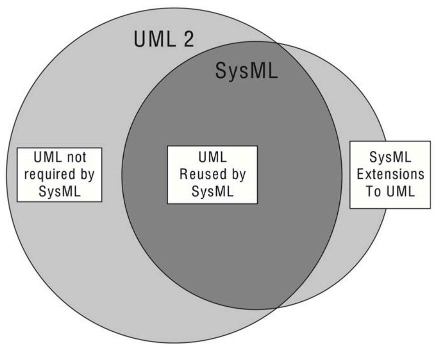
Anticipating the selected modeling language to be used in the novel framework for the development of KBE applications (see Section 5.3.1), a brief introduction to SysML is presented here.

SysML is a general-purpose graphical modeling language that supports the analysis, specification, design, verification, and validation of complex systems. The modeled systems may include hardware, software, data, procedures, people, facilities, and other elements [10]. Furthermore, SysML supports modeling of requirements, structure, behavior, and parametrics (the so-called “4 pillars of SysML”) in order to provide a complete description of a system, its components, and its environment. The language includes 9 types of diagrams, presented in Figure 3.1b, that can be used to describe different views of the system being modeled [10].

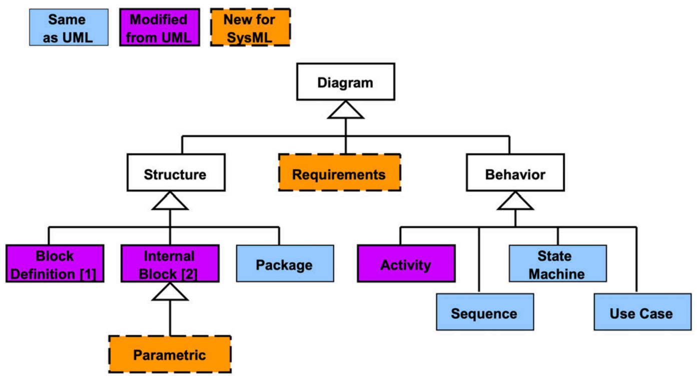
SysML is defined as a lightweight profile of UML 2.x, the industry standard modeling language for software-intensive applications, as illustrated in Figure 3.1a. It is designed with relatively small modifications to the underlying UML language, using simple stereotypes, tagged values, and constraints

[1] `https://www.omg.org/spec/SysML/` (Accessed: 16/06/2023)
[2] `https://www.bpmn.org` (Accessed: 16/06/2023)

[37]. This approach allows for the reuse of the mature notation and semantics of UML 2.x, which is well established among software engineers and already implemented by many modeling tool vendors. SysML's status as a UML profile has advantages in terms of facilitating its adoption and implementation.



(a) Relation between SysML and UML. Source: [37]



(b) SysML Diagram Taxonomy. Source: Adapted from [38]

Figure 3.1: Overview of the Systems Modeling Language

SysML inherits many of the UML diagrams, these are either reused without modification or slightly modified with lightweight customizations. In addition, SysML adds two new diagrams specifically designed for modeling requirements and parametrics. The presence of a dedicated diagram for modeling requirements sets SysML apart from UML, and is a decisive factor that allows SysML to be selected as the key enabling modeling language of the developed framework. Some of these diagrams will be discussed in more detail in the following chapters, and examples of the diagrams used in the novel framework for KBE app development are presented in Section 6.2.1. Furthermore, SysML targets systems engineers, while UML targets software engineers [39]. Thus, SysML provides advantages for defining complex systems by offering additional features such as satisfaction and allocation matrices, which are essential for modeling the information and relationships related to the requirements, process, and product knowledge needed to develop KBE applications.

3.4.2. Modeling Tool

A modeling tool is a software application used to create, edit, and manage models. Modeling tools provide a graphical user interface (GUI) and other features that enable stakeholders to create and manipulate models using a modeling language. Modeling tools can be used to create different types of models, such as requirements models, design models, and behavioral models. The purpose of a modeling tool is to enable stakeholders to create, manage, and analyze complex systems using a standardized modeling language. Modeling tools can help stakeholders to create more accurate and complete models, identify and resolve issues more quickly, and collaborate more effectively with other stakeholders in the system development process. Modeling tools can also provide features for simulation and analysis, enabling stakeholders to evaluate the behavior and performance of the system being modeled.

There are several Commercial Off-The-Shelf (COTS) as well as Free and Open-Source Software (FOSS) modeling tools available for implementing MBSE. Some of the most commonly used nowadays are presented in Table 3.1. In addition to modeling knowledge, some of these tools also provide a mechanism to automatically convert diagrammatic knowledge models into neutral language knowledge models that use an open standard file format called XML Metadata Interchange (XMI[3]), developed by the Object Management Group (OMG). As a result, neutral language knowledge models in this format can be opened and edited in other SysML modeling tools that support the XMI standard. This is an important enabling step in speeding up the KBE application development process, which was not previously possible with MOKA. Moreover, some modeling tools, such as Magic Systems of Systems Architect (MSoSA), also support the automated translation of UML models into programming languages such as Java or C++.

Table 3.1: Main Modeling Tools currently available for implementing MBSE

Modeling Tool	Tool Vendor	License Type
MSoSA[a] / CSM[b]	Dassault Systèmes	COTS[h]
ESDR[c]	IBM	COTS
EA[d]	Sparx Systems	COTS
Papyrus[e]	Eclipse	FOSS[i]
Capella[f]	Eclipse	FOSS
Modellio[g]	Modeliosoft	FOSS

[a] MSoSA: Magic Systems of Systems Architect (Accessed: 20/03/2023)
[b] CSM: Cameo Systems Modeler (Accessed: 20/03/2023)
[c] ESDR: Engineering Systems Design Rhapsody (Accessed: 20/03/2023)
[d] EA: Enterprise Architect (Accessed: 20/03/2023)
[e] `https://www.eclipse.org/papyrus` (Accessed: 20/03/2023)
[f] `https://www.eclipse.org/capella` (Accessed: 20/03/2023)
[g] `https://www.modeliosoft.com/en` (Accessed: 20/03/2023)
[h] COTS: Commercial Off-The-Shelf
[i] FOSS: Free and Open-Source Software

When selecting a modeling tool, there are several assessment criteria that should be considered. These criteria are as follows [10]:

1. The **functionality** of the tool should be assessed to ensure it has the necessary features and capabilities to create, manage, and analyze models, and to support specific modeling languages and standards.
2. The **usability** of the tool should be evaluated to ensure that it is easy to use and intuitive, with a clear and easily navigable user interface.
3. The **scalability** of the modeling tool should be considered to ensure that it can handle projects of different sizes and complexities.
4. The **integration** of the tool should be consideration, as the modeling tool should be able to integrate with other tools and systems used in the development process.
5. The **customization** capabilities are also important, as the tool should allow for customization and configuration to meet the specific needs of the project, such as the ability to create custom templates, add custom model elements, or create custom reports.
6. Lastly, a comprehensive documentation and **support** system should be available for the modeling tool, including user manuals, online forums, and customer support, to ensure that any issues that arise during the modeling process can be quickly resolved.

[3] `https://www.omg.org/spec/XMI/` (Accessed: 16/03/2023)

Magic Systems of Systems Architect / Cameo Systems Modeler

To identify the most suitable modeling tool for this research, a brief evaluation was conducted on three widely used commercial modeling tools. This assessment was guided by recommendations from the literature[4] [40] and involved hands-on experimentation with the tool most frequently recommended (Magic Systems of Systems Architect / Cameo Systems Modeler). The author's evaluation of these tools against the aforementioned selection criteria is presented in Table 3.2. FOSS SysML modeling tools were not included in this assessment. This decision was based on the fact that COTS SysML modeling tools generally provide superior support and documentation, are well established in the industry, and offer a stable platform with fewer bugs in comparison.

Table 3.2: Assessment of COTS modeling tools against specific selection criteria

Assessment Criteria	MSoSA[a]	ESDR[b]	EA[c]
Functionality	+	+/-	-
Usability	+/-	+/-	+/-
Scalability	+	+	+
Integration	+	-	+/-
Customization	+/-	+/-	-
Support	+	+/-	+/-

[a] MSoSA: Magic Systems of Systems Architect (by Dassault Systèmes)
[b] ESDR: Engineering Systems Design Rhapsody (by IBM)
[c] EA: Enterprise Architect (by Sparx Systems)

While evaluating the three SysML-compliant COTS modeling tools, Magic Systems of Systems Architect (MSoSA), Engineering Systems Design Rhapsody (ESDR), and Enterprise Architect (EA), each tool demonstrated strengths and weaknesses. However, MSoSA stood out as a pragmatic choice for implementing MBSE, as it strictly enforces SysML's syntax and semantics, provides support for basic requirements traceability, and allows to automatically export the SysML models to neutral language models using the XMI industry-standard format. Despite the noted challenges at times with its complex user interface, the tool's extensive functionality and support made it a strong contender for the framework's specific requirements. In comparison, ESDR's weak support for Activity diagrams and lack of integration with multiple platforms, and EA's limited enforcement of SysML well-formedness rules, and crude extension mechanisms made them less favorable options. Therefore, based on the evaluation of the key assessment criteria, Magic Systems of Systems Architect / Cameo Systems Modeler was selected as the preferred modeling tool for formalizing KBE specific knowledge using SysML, as it offers the most robust standards-compliant SysML solution. This is in alignment with the prevalent industry practices [40].

3.4.3. Modeling Method

A modeling method is a systematic approach to creating, managing, and analyzing models that comprehensively represent the various aspects of a complex system. It establishes a structured framework for capturing the system's requirements, architecture, behavior, and other relevant elements using modeling languages like SysML. By offering guidelines, principles, and best practices, modeling methods ensure the creation of accurate, complete, and consistent models. These methods define the

[4]`https://sysml.org/sysml-tools/`

specific activities involved in modeling, including the production and development of artifacts such as diagrams [10]. The purpose of a modeling method is to ensure that models are created in a consistent and repeatable way, enabling stakeholders to effectively communicate and collaborate with each other. Modeling methods can help stakeholders improve the quality of the models being created, and ensure that the models are aligned with the goals and objectives of the system being developed.

There are several modeling methods available in the literature, with several examples documented by Estefan [41] in a Survey of Model-Based Systems Engineering Methodologies. Some of the most notable modeling methods in MBSE are presented in Table 3.3.

Table 3.3: Notable Modeling Methods available for implementing MBSE

Modeling Method	**Method Developer**
OOSEM[a]	INCOSE
OPM[b]	Dori
RUP SE[c]	IBM
MagicGrid[d]	Dassault Systèmes

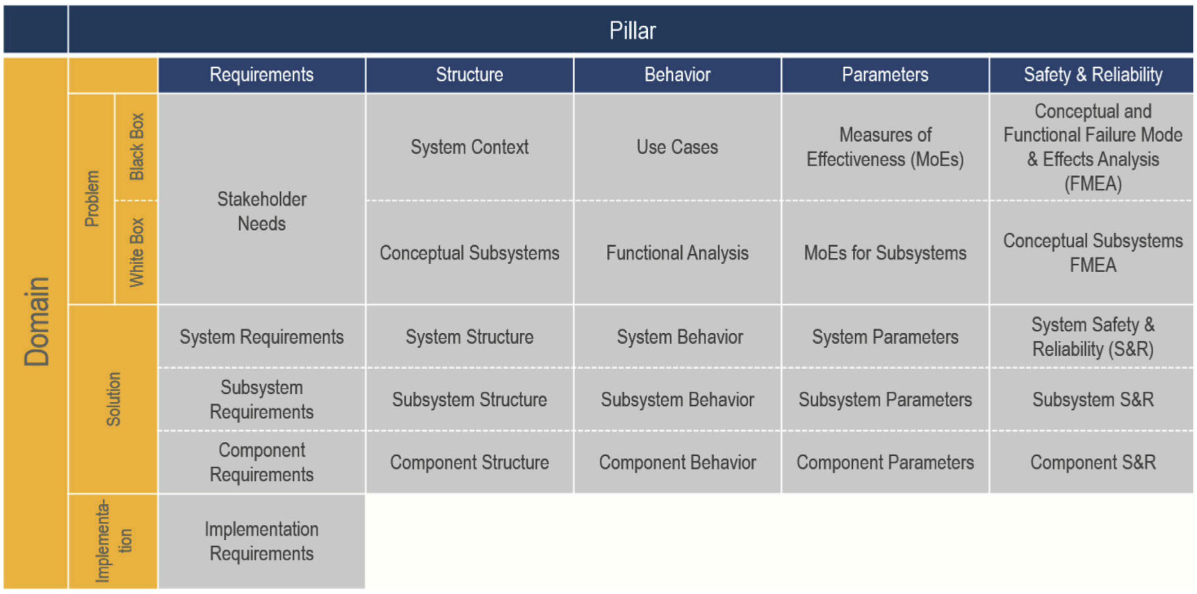
[a] OOSEM: Object-Oriented Systems Engineering Method [10, 35]
[b] OPM: Object-Process Methodology [42]
[c] RUP SE: Rational Unified Process for Systems Engineering [43]
[d] MagicGrid [44]

MagicGrid

Anticipating the selected modeling method to be used in the novel framework for the development of KBE applications (see Section 5.3.3), a brief introduction to the MagicGrid method is presented here. The MagicGrid method is a culmination of best practices from numerous MBSE adoption projects and is designed to be modified or extended to support specific customer needs [45].

The method's compatibility with vanilla SysML, tool-independence (as long as the tool supports SysML), and capability to be modified to support the specific scope being addressed are key factors that justify its selection as the chosen modeling method to be implemented in the novel framework.

The MagicGrid method can be represented as a Zachman style matrix (see Figure 3.2) and is designed to guide the engineers through the modeling process of complex systems.



Domain			Pillar				
			Requirements	Structure	Behavior	Parameters	Safety & Reliability
	Problem	Black Box	Stakeholder Needs	System Context	Use Cases	Measures of Effectiveness (MoEs)	Conceptual and Functional Failure Mode & Effects Analysis (FMEA)
		White Box		Conceptual Subsystems	Functional Analysis	MoEs for Subsystems	Conceptual Subsystems FMEA
	Solution		System Requirements	System Structure	System Behavior	System Parameters	System Safety & Reliability (S&R)
			Subsystem Requirements	Subsystem Structure	Subsystem Behavior	Subsystem Parameters	Subsystem S&R
			Component Requirements	Component Structure	Component Behavior	Component Parameters	Component S&R
	Implementa-tion		Implementation Requirements				

Figure 3.2: The MagicGrid framework represented as a Zachman style matrix.
Source: [44]

As illustrated in Figure 3.2, the MagicGrid method defines the Problem, Solution, and Implementation domains for the development of the system of interest (SoI). Each domain is represented as an individual row within the MagicGrid matrix. The Problem domain row is further divided into two segments to indicate that defining the problem domain entails examining the SoI from both a black-box and white-box perspective. The Solution domain row is subdivided into multiple inner rows to emphasize the possibility of specifying the solution architecture at various levels of granularity. The Implementation domain row is not fully highlighted like the upper rows, denoting that, apart from the implementation requirements specification, other elements within this domain are not a part of MBSE and therefore appear outside the scope of the matrix [44].

Each domain's definition encompasses distinct aspects of the SoI, aligning with the four pillars of SysML: Requirements, Structure, Behavior, and Parameters (also referred to as Parametrics). Additionally, the Safety & Reliability aspect is also considered. These aspects are represented as columns within the matrix.

A cell located at the intersection of a specific row and column represents a view of the system model, which may consist of one or more presentation artifacts. These artifacts can take the form of diagrams, matrices, maps, or tables. The modeling workflow is defined by the order of the cells in a left-to-right and top-to-bottom manner.

4 Research Objective and Research Questions

This chapter presents the research objective and research questions that guide the performed research work.

From the review on the state of the art of existing methodologies and current practices in the development of KBE applications, it can be concluded that the existing methodologies have shortcomings that consequently impact the time required to develop applications, their quality (i.e., requirement compliance & traceability, maintainability, scalability, etc.), and the eventual capability to preserve and efficiently reuse engineering knowledge. To improve the development of KBE applications and tackle their current limitations, the following research objective of this thesis work is defined.

Research Objective

Create a methodological approach to improve KBE applications in terms of transparency, requirements traceability, reuse of knowledge, and development time.

Considering MBSE's promising capabilities in tackling the aforementioned challenges, the aim is to achieve the research objective by creating a framework for developing KBE applications, that implements an MBSE-based methodological approach.

In order to verify the attainment of the research objective, the main research question is formulated as:

Q.1. Main Research Question

"What is the impact of using MBSE to support the development of KBE applications in terms of their transparency, requirements traceability, knowledge reuse, and development time?"

To guide the research process, a set of sub-questions, derived logically from the main question, is formulated. The main research question can initially be approached by answering the following sub-question:

Q.1.1. Sub-question

“To which phases of the KBE app development process can MBSE be applied, and which customizations are necessary to support it?”

As explained in Chapter 3, MBSE is primarily concerned with creating digital knowledge models of complex systems. Hence, it can be expected that MBSE will be used in the proposed framework, during the knowledge modeling (i.e., capture and formalization) phase of the KBE app development process, to create digital knowledge models that capture the knowledge required for developing KBE applications. As such, the following sub-question that could be addressed is:

Q.1.2. Sub-question

“How to bridge the gap between the knowledge modeling phase and the code formulation phase in the KBE app development process?”

As mentioned in Section 2.2.2, an effective approach to address the disconnection between knowledge models and KBE application code, is the utilization of tools that automatically generate source code from knowledge models. Therefore, this approach will be implemented in the current reserach work. As such, a sub-question that originates naturally from this, can be formulated as:

Q.1.3. Sub-question

“What is the maturity of the automatically generated KBE code in terms of completeness, reliability, and quality?”

Furthermore, it is necessary to address considerations related to the maintenance and future developments of KBE applications. As such, the following sub-question must be answered:

Q.1.4. Sub-question

“How to guarantee the synchronization and consistency between the knowledge models and the KBE application encoding that knowledge?”

5 Development of the Novel MBSE-for-KBE Framework

This chapter describes the development of the novel KBE app development framework based on MBSE concepts. It begins by outlining the proposed key solutions to the identified challenges in KBE app development. Subsequently, it provides an overview of the developed framework, highlighting its main components and functionalities. The chapter then delves into a detailed explanation of the process employed in creating the framework. Finally, a new KBE application development process is proposed, utilizing the developed framework as a central component.

5.1. Framework Overview

In order to improve the current KBE app development process, it is necessary to address the challenges summarized in Section 2.3. The proposed key solutions to those challenges can be summarized as follows:

1. To enhance the involvement of domain experts in the development of KBE applications, a new knowledge formalization technique should be implemented, instead of the more complex ICARE forms which are not intuitive for domain experts. The new knowledge formalization technique should capture the KBE application's requirements and associate them with appropriate process steps and product features (i.e., a Requirement-Product-Process Ontology as shown in Figure 5.1), which is a more intuitive approach for domain experts.
2. The outcome of the knowledge capture and formalization phase should consist of digital models (instead of documents) that can be used to automatically generate neutral language knowledge models. This will provide a direct connection between the formal knowledge models and the KBE application code, thereby enabling automatic generation of code from the models.
3. The neutral language knowledge model should be utilized to automatically generate (part of) the KBE application code. This will justify the effort spent on creating the formal knowledge model, as developers will save time by generating the KBE application source code automatically.

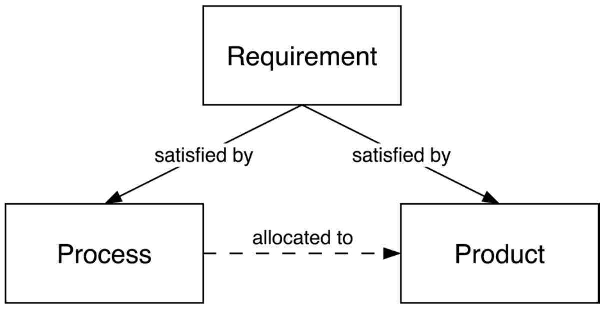


Figure 5.1: Relationships between Requirement, Process, and Product. The requirements can be satisfied by process steps or product features. The process steps are associated to relevant product features by the allocation relationship of SysML.
Source: Adapted from [46]

A structured three-part research methodology was employed to implement the previously identified key solutions. The first part involved defining an ontology (i.e., set of concepts and relationships between them) to effectively describe the knowledge involved in the development of KBE applications namely, its requirements, processes, and products. This ontology is then mapped to the ontology of a target KBE system, to establish a correspondence between the modeled knowledge and the language of the KBE system. This is an essential step to support the automatic generation of application code from a knowledge model. The second part identified a suitable modeling language, tool, and method for capturing the engineering knowledge used in KBE app development. Finally, the third part involved the development of translation engines that could be used to automatically generate skeleton code from the knowledge models. It is important to note that the development process was not strictly linear but rather iterative, with feedback and refinement loops between the three parts. The outcome of this research methodology was the development of an MBSE-based KBE app development framework that is schematically represented in Figure 5.2. In this research work the target KBE system was ParaPy.

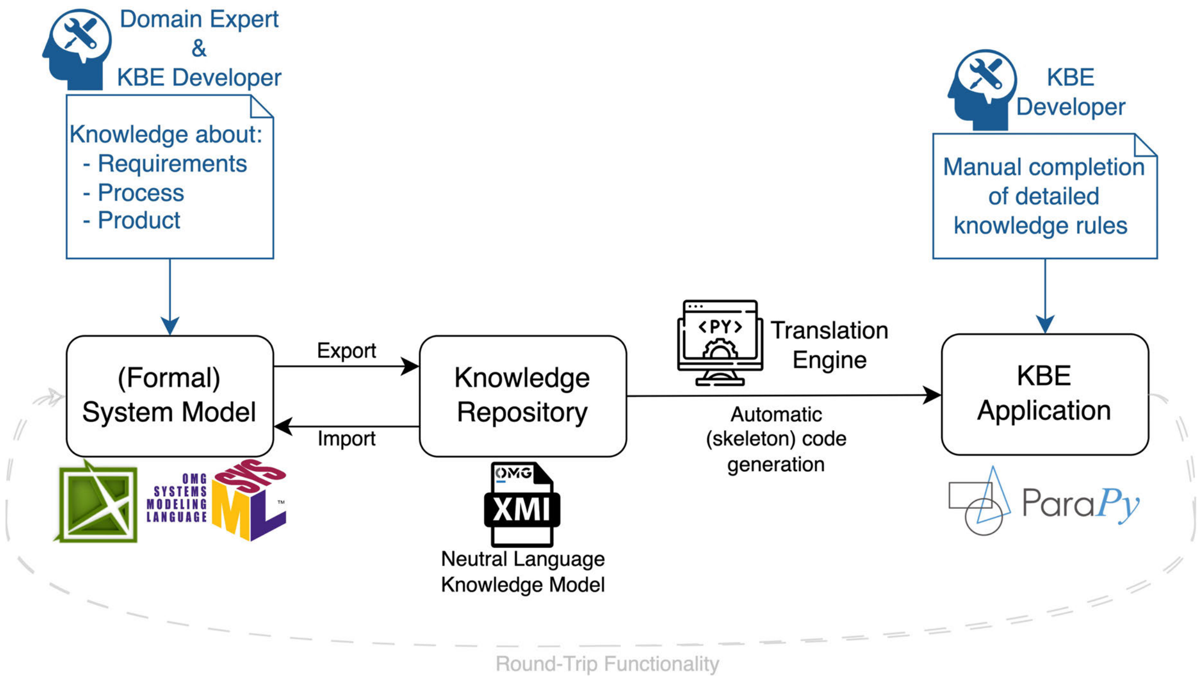


Figure 5.2: Overview of the MBSE-based Framework for KBE App Development. The dashed lines represent proposed functionalities for future works, aimed at enhancing the capabilities of the framework beyond the scope of this research.

The main components of the developed framework are the following:

- **Formal System Model:** A formal model of the system knowledge is created by a team of domain experts and KBE developers. This model encompasses information about the system's (i.e., KBE app) requirements, processes, and products. In this research, SysML is chosen as the modeling language, and MSoSA is selected as the modeling tool, based on the criteria discussed in Section 5.3.1 and Section 5.3.2, respectively.
- **Knowledge Repository:** Once the knowledge model reaches an adequate level of maturity, it is exported and stored in a Knowledge Repository. The knowledge models are exported in the neutral XML Metadata Interchange (XMI) standard, enabling them to be opened and edited in other SysML modeling tools that support this standard.

  The knowledge repository includes a collection of (primitive) geometry knowledge provided by the target KBE system, as well as knowledge captured from previous projects and stored in the XMI format. During the development of the formal system model in the modeling tool, existing knowledge in the repository can be imported, facilitating effective knowledge reuse.
- **Translation Engine:** A Python-based code generator was developed in this research, the translation engine. Its function is to automatically generate KBE application code by parsing neutral language knowledge models stored in the XMI format. By leveraging the mapping between the knowledge model and the target KBE system (see Section 5.2.3), the translation engine produces the appropriate KBE code.

  Currently, the translation engine is specifically designed to work with ParaPy. However, the framework's design, which incorporates a neutral language knowledge model and an explicit mapping between the knowledge model and the KBE system, allows for easy extension to other KBE languages. This can be achieved by establishing a similar mapping between the knowledge model and the ontology specific to the new KBE language.
- **KBE Application:** Depending on the level of detail in the system knowledge model, the translation engine can generate either a code skeleton or the complete application code. The code skeleton consists of key classes, attributes, and their relationships in a correct file structure, while the complete application code is ready for immediate use.

  In the case of the code skeleton, KBE developers can augment it by manually adding the programming-intensive aspects using specialized IDEs like PyCharm, which are better suited for coding than SysML modeling tools.

Currently, the framework developed in this research is limited to generating application skeleton code, whose development is acknowledged to be a critical part in the overall KBE app development process. The skeleton code is then manually completed by the KBE application developer. However, a potential issue arises when modifications are made to the knowledge model after manual code additions, as regenerating the code from the model would result in the loss of the previously added code.

To prevent this knowledge loss, there is a need for capabilities that enable the generation of the knowledge model from the application source code, commonly known as reverse engineering [47]. Reverse engineering can be useful in the process of formalizing manually added code within the knowledge model, as well as generating knowledge models from pre-existing KBE applications.

Furthermore, a "round-trip" functionality, which combines code generation and reverse-engineering capabilities, could be developed. This functionality would enable domain experts and developers to seamlessly transition between code and models, facilitating the validation, updating, and reuse of the underlying knowledge, while ensuring consistency and synchronization between the models and the application code. However, the development of this functionality was considered beyond the scope of this research and will be addressed in the future recommendations presented in Section 8.2.

The proposed framework does not seek to reject any ideas from the MOKA methodology, but rather aims to extend and tailor them for a more comprehensive system description. This is achieved by utilizing a more expressive formal model using SysML, as opposed to MOKA's formal model using MML. Additionally, the proposed framework aims to streamline the KBE application development process by eliminating the need for an informal knowledge model (MOKA's ICARE forms) and the involvement of a specialized knowledge engineer. The proposed framework has the potential to allow for a knowledge modeling step in the KBE application development process, while solving the problem of the disconnection between the knowledge models and the application code.

The three phases of the structured research methodology employed in the development of the framework are described in the following sections.

5.2. Ontology Definition

An ontology is *"an explicit description of concepts in a specific domain, defining a formal domain vocabulary. An ontology defines relationships between the concepts in the domain, and adds attributes to further specify a concept, making it a suitable means of capturing and reusing knowledge"* [14].

The first part of the framework development focused on defining an ontology capable of describing the knowledge involved in KBE applications and allow for an effective knowledge capture and formalization. This ontology, referred to as the "Knowledge Ontology", aims to define exactly which concepts and relationships between those concepts can be captured in a formal knowledge model of a KBE application. This gives domain experts and KBE developers a clear formal domain vocabulary for capturing the knowledge systematically in digital models.

The definition of the Knowledge Ontology involved several key steps. Firstly, the foundational concepts necessary for KBE application development were identified based on the top-level concepts (Requirements, Process, and Product) illustrated in Figure 5.1. Secondly, the ontology of the target KBE system, ParaPy, was examined, and the identified concepts and relationships were integrated into the Knowledge Ontology. This integration is crucial for facilitating the mapping between the two ontologies, in order to enable automatic code generation for the target KBE system from the modeled knowledge. The third step involved identifying and incorporating specific elements and terminology from the chosen modeling language, SysML, into the Knowledge Ontology. This step aimed to ensure that the ontology aligned with the modeling language and allowed for the effective capture and representation of knowledge using SysML. Finally, the last step consisted of mapping the Knowledge Ontology to the ontology of the target KBE system. This is an essential step to enable automatic code generation, as it gives information on what form the knowledge takes when represented as application code. Additionally, this mapping serves as a validation mechanism to ensure that all elements in the knowledge model can be effectively represented in the application code, and vice versa. An overview of the ontologies and their mapping is presented in the following sections.

5.2.1. Knowledge Ontology

Semantic Triples were employed as the chosen representation format for the developed Knowledge Ontology. This decision was motivated by the format's ability to express knowledge in a clear, unambiguous, and machine-readable manner [48], which aligns well with the goal of formalizing knowledge in a digital model. In semantic triples, a subject is connected to an object via a predicate, as illustrated in Figure 5.3.

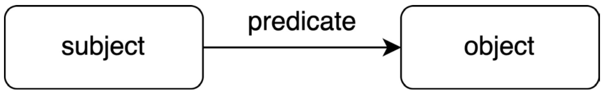


Figure 5.3: Semantic Triples. Used to define how two concepts relate to each other.
Source: Adapted from [48]

In the context of KBE applications, the primary focus of domain experts and KBE developers revolves around Requirements, Process, and Product. Hence, these concepts were established as the top-level components of the Knowledge Ontology. To comprehensively capture the knowledge necessary for modeling and developing a complete KBE application, these top-level components were further expanded into more specific concepts. The initial version of the Knowledge Ontology, encompassing the foundational concepts pertinent to KBE application development, is depicted in Figure 5.4.

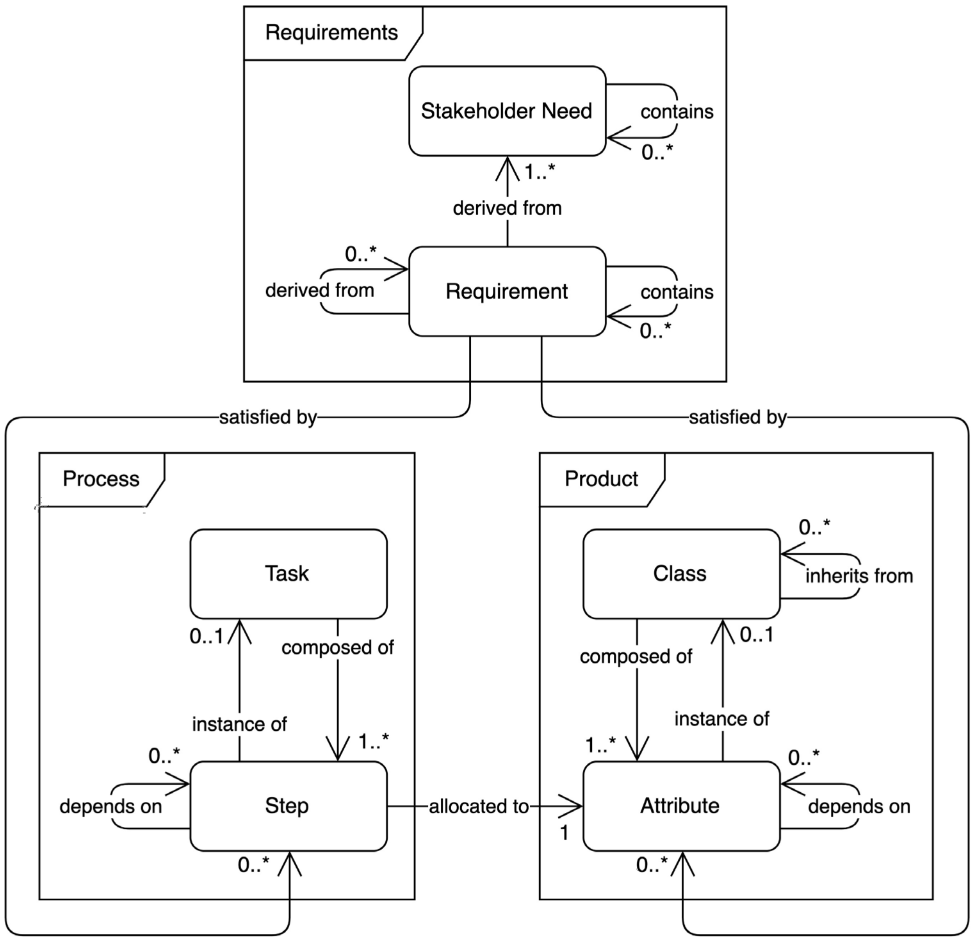


Figure 5.4: Initial Definition of the Knowledge Ontology considering only the foundational concepts involved in the development of KBE applications

In the initial definition of the Knowledge Ontology, within the Requirements sub-ontology, two foundational concepts were identified, namely, *Stakeholder Needs* and (system) *Requirements*. Stakeholder needs represent the users' needs for the KBE application in an unstructured and general (ambiguous) manner. On the other hand, requirements add structure and rules to those needs so that they are clear, unambiguous, and verifiable.

Within the Process sub-ontology, two foundational concepts were identified, namely, *Tasks* performed by the KBE application and the *Steps* required to complete each task. Tasks represent the distinct activities undertaken by the KBE application, while Steps outline the necessary actions or stages involved in accomplishing each task.

Within the Product sub-ontology, two foundational concepts were identified, namely, *Classes* and *Attributes*. Classes denote the different types of entities or objects relevant to the KBE application, while Attributes specify the characteristics or properties associated with each class.

Through multiple iterations, the Knowledge Ontology was refined and expanded by incorporating relevant concepts from the target KBE system and the modeling language. This iterative process aimed to ensure the ontology's completeness and suitability for its intended purpose. The final version of the Knowledge Ontology, achieved after these refinement iterations, is presented in Figure 5.5. Additionally, Table 5.1 provides definitions for the relationships among the various elements within the ontology.

Table 5.1: Definitions of the Relationships between Elements of the Knowledge Ontology and mapping of those Relationships to SysML syntax.
Source: Adapted from [49]

Knowledge Ontology	**Type of Relationship**	**Brief semantics**	**SysML syntax**
			source *target*
"imports" "derived from" "depends on"	Dependency	The source element depends on the target element and may be affected by changes to it.	- - - - - - - - - ->
"satisfied by" "allocated to" "represented as"	Association	The description of a set of links between objects.	————————
"composed of"	Composition	The target element is a part of the source element.	◆————————
"contains"	Containment	The source element contains the target element.	⊕————————
"specialization of" "inherits from"	Generalization	The source element is a specialization of the more general target element.	————————▷
"instance of"	Realization	The source element guarantees to carry out the contract specified by the target element.	- - - - - - - - - -▷

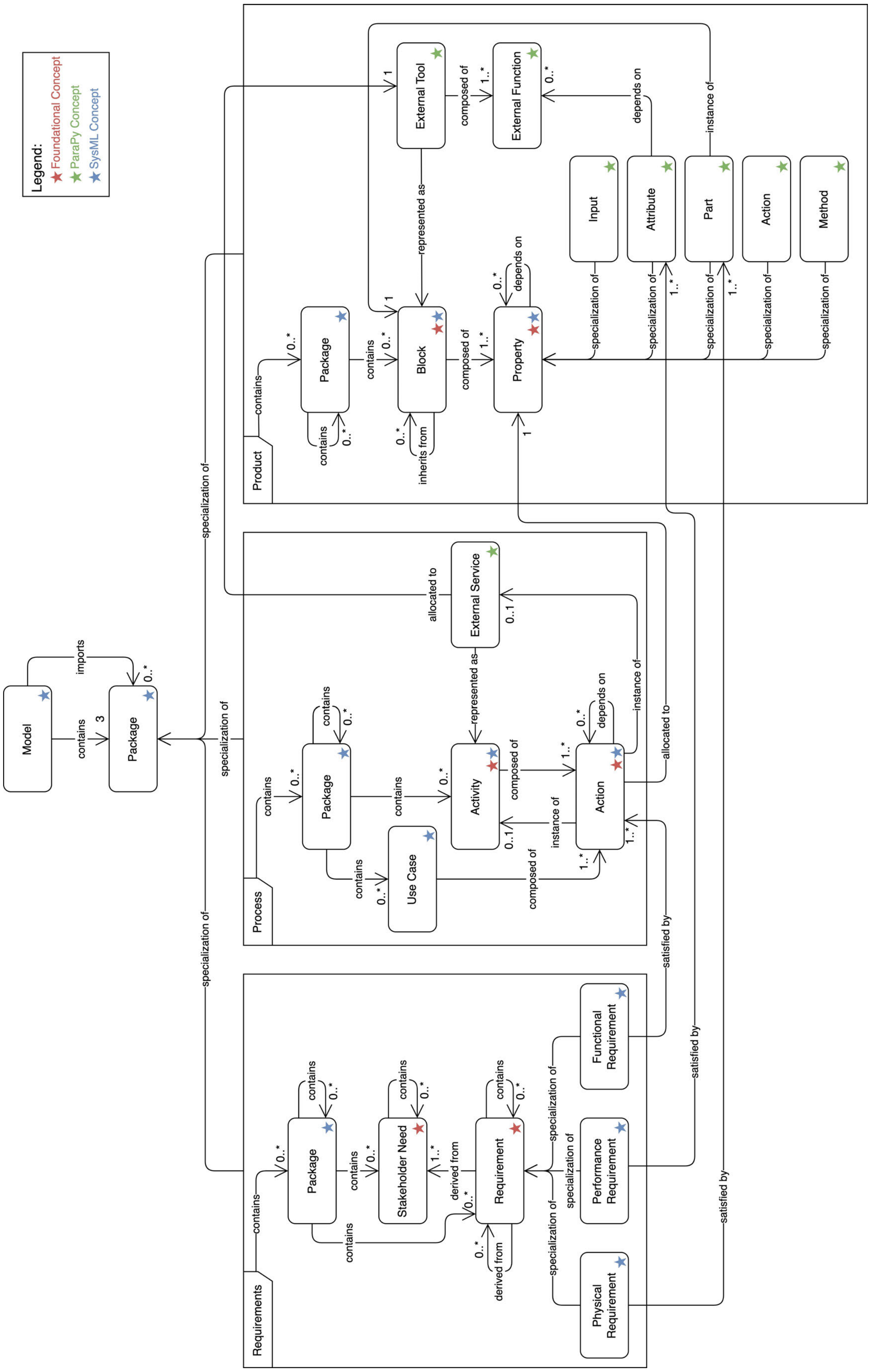


Figure 5.5: Final Definition of the Knowledge Ontology after incorporating the concepts from the target KBE system (ParaPy) and the modeling language (SysML). The foundational concepts previously identified (with different terminology) are represented with a red star, the concepts incorporated from the target KBE system ontology are represented with a green star, and the concepts incorporated from the modeling language are represented with a blue star.

The main element of the final Knowledge Ontology is the *Model*, which contains all the knowledge required to build the KBE application. A Model must contain at least three *Packages*, namely, *Requirements*, *Process*, and *Product* - the top-level elements of the ontology. Each of these packages may contain other packages to help better organize the model. A Model may also import Packages from other Models, facilitating project-to-project knowledge transfer. These top-level elements and their more specific concepts that must be considered in order to capture the knowledge required to model/develop a complete KBE application are briefly explained in the following paragraphs.

Requirements Package

The Requirements package is used to capture stakeholder needs and (system) requirements. Each *Requirement* must be (implicitly) derived from at least one *Stakeholder Need* and can also be derived from other requirements. A requirement must be satisfied by elements from either the Process or Product packages. Three specific types of requirements are considered, namely *Physical Requirement*, *Performance Requirement*, and *Functional Requirement*.

Process Package

The Process package is used to represent the behavior of the system (i.e., KBE application) by capturing its *Use Cases* and its *Activities*. Activities and Use Cases are composed of smaller atomic elements called *Actions*, which capture a basic step of the system functionality within the specific Activity or Use Case. Actions must be allocated to elements of the Product package that are responsible for performing them. Additionally, activities that are performed outside the KBE app are also considered, which were named External Services. It must be noted that KBE applications do not need to have their behavior/process explicitly defined, because it is implicitly handled by the dependency tracking capabilities of KBE systems. The main goal of modeling the Process of a KBE application is to increase the transparency and traceability of knowledge within the KBE app, thereby decreasing its black-box perception.

Product Package

The Product package is used to represent the structure of the system by capturing the conceptual subsystems that are part of the KBE application. These subsystems are captured in what SysML calls *Blocks* (equivalent to UML's *classes*). Blocks are composed of smaller atomic elements called *Properties*, which may depend on each other. Five specific types of Properties were considered, namely *Input*, *Attribute*, *Part*, *Action*, and *Method*. These types are directly associated with the *slots* of the ParaPy ontology (see Figure 5.6). Additionally, tools external to the KBE app are also considered, which were named *External Tools*. External Tools are composed of *External Functions*, which can be connected to Attributes.

5.2.2. ParaPy Ontology

The ontology of the target KBE system (in this case ParaPy) was also identified, and is presented in Figure 5.6. This is an essential step so that later the Knowledge Ontology can be mapped to the ontology of the target KBE system.

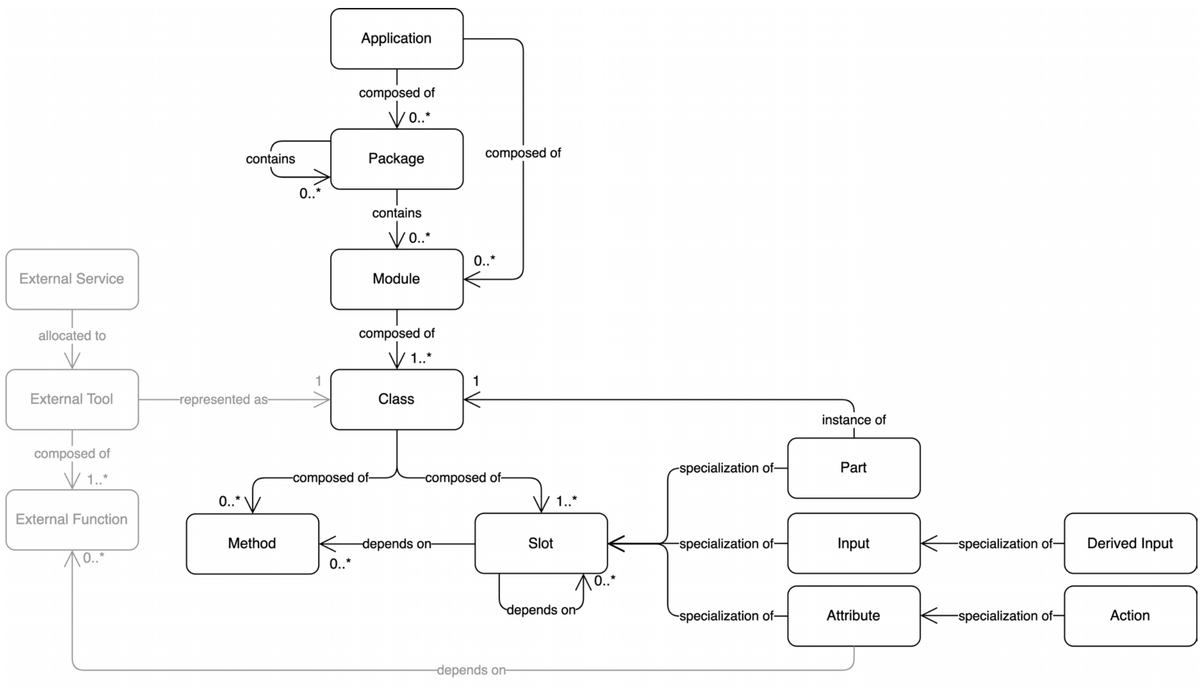


Figure 5.6: Definition of the ParaPy Ontology. The elements represented in gray are external to the ParaPy KBE system, but must still be considered during the development of a KBE application since the application interacts with them.

The main element of the ParaPy Ontology is the (KBE) *Application*. The Application can be composed of multiple *Packages*, each of which can contain other Packages, to help organize the Application into logically cohesive groups. Packages may also contain *Modules* (.py files) that contain the application source-code. Each Module is composed of at least one *Class*. Classes are composed of *Slots*, i.e., attributes from the OOP paradigm that specify the characteristics of each class (see Figure 2.2). Additionally, Classes may also be composed of *Methods*, i.e., normal Python class methods. Slots may depend on other Slots or Methods. The three main types of Slots are *Input*, *Attribute*, and *Part*. Additionally, *Derived Input* and *Action* are special types of Input and Attribute, respectively.

5.2.3. Mapping between the Knowledge Ontology and the ParaPy Ontology

Finally, the Knowledge Ontology was mapped to the ParaPy Ontology in order to enable knowledge traceability from the Knowledge Base to the KBE application code. As previously mentioned, this constitutes an essential step to enable (semi) automatic generation of KBE applications from knowledge models. This mapping is presented in Figure 5.7.

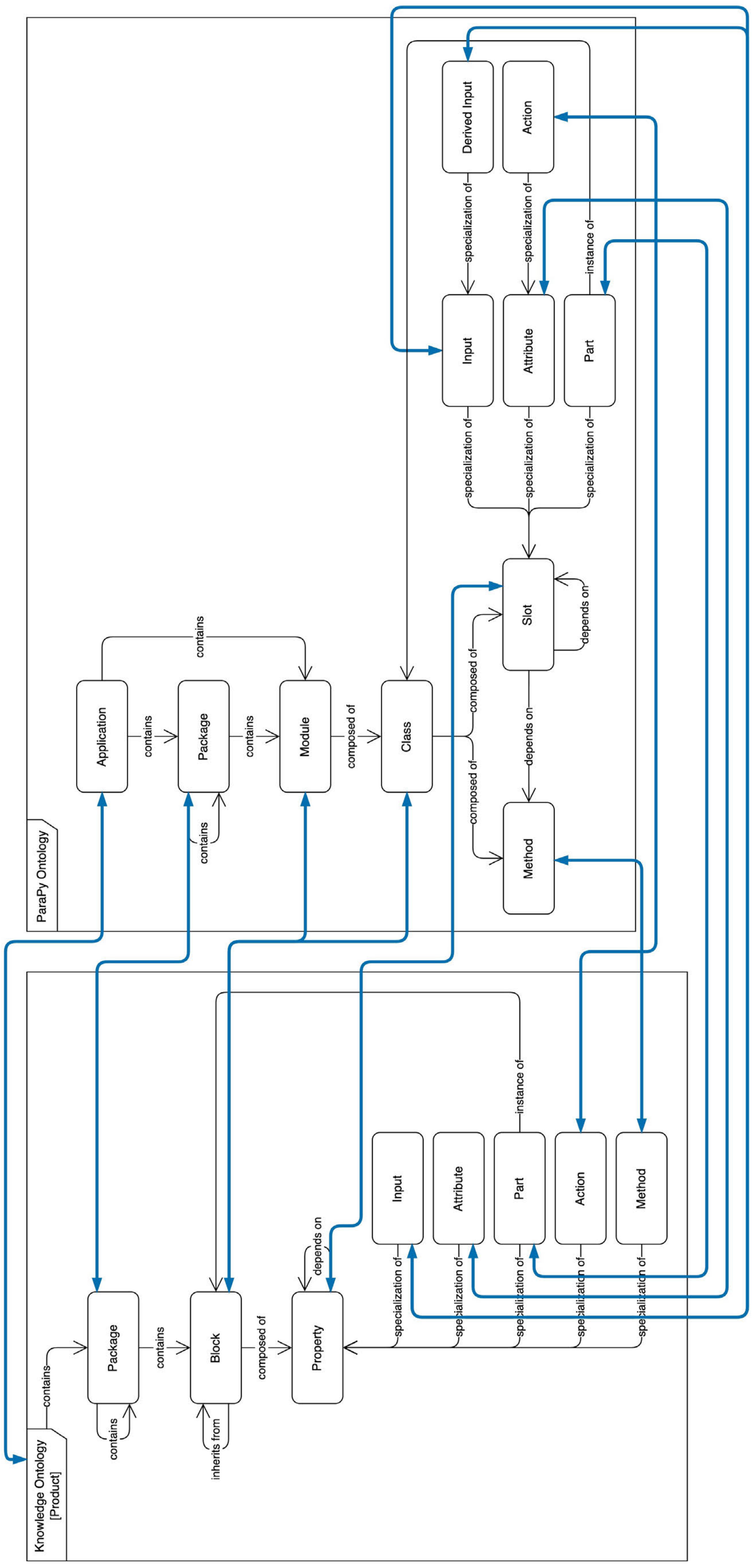


Figure 5.7: Mapping of the Knowledge Ontology to the ParaPy Ontology. The ontologies are simplified, representing only the elements for which code can be automatically generated.

It is important to highlight that the mapping process between the Knowledge Ontology and the ParaPy Ontology focused solely on the Product package. The decision to exclude the mapping of elements from the Process package was based on the intrinsic capabilities of ParaPy-based KBE applications. As previously explained, these applications possess a dependency-tracking mechanism, which determines the appropriate sequence of slot evaluations. Consequently, there is no need to explicitly map the elements from the Process package to achieve the desired behavior in the application. The dependency tracking mechanism in ParaPy handles the sequencing of evaluations, ensuring that the necessary processes are executed in the correct order.

Furthermore, it is important to highlight that most elements of the ontologies have a one-to-one mapping, except for the Block and Input elements of the Knowledge Ontology.

The Block element of the Knowledge Ontology maps to the Module and Class elements of the ParaPy Ontology because each Block generates a Module with a single Class inside of it. It should be noted that while most programming languages adhere to this one-class-per-module structure, Python allows for the definition of multiple classes within a single module. Consequently, this presents a current limitation within the developed framework, which will be addressed in the future recommendations outlined in Section 8.2. To overcome this limitation in the current framework, a workaround can be applied by grouping all Blocks that define Classes intended to be within the same Module inside a single Package. This ensures that the desired structure and organization are maintained, despite the constraint imposed by the framework.

The Input element of the Knowledge Ontology maps to the Input and Derived Input elements of the ParaPy Ontology. However, this does not constitute a limitation because the Input element of the Knowledge Ontology was defined in a way that can automatically handle both cases of the ParaPy Ontology.

5.3. Selection and Tailoring of MBSE Key Components

The second part of the framework development focused on selecting the three pillars required in order to effectively apply Model-Based Systems Engineering to produce and control a coherent system model. Moreover, this phase involved tailoring these pillars for the purpose of supporting the development of KBE applications. The three pillars are the modeling language, tool, and method. Their description and selection criteria will be further discussed in the following subsections.

5.3.1. Modeling Language

For the purpose of this research, the Systems Modeling Language (SysML) was selected as the graphical modeling language for MBSE due to its extensive usage and established reputation for meeting industry standards. The rich semantic and expressiveness of the SysML diagrams, together with their flexibility but conceptual rigor, make it a suitable modeling language for the purpose of modeling the knowledge required for developing KBE applications. SysML offers three major types of diagrams that support the modeling of Requirements, Behavior (Process information), and Structure (Product structure), allowing to fully capture the top-level concepts of the previously defined Knowledge Ontology.

SysML itself is extendible to support domain-specific needs or a specific methodology. This is essential in order to tailor it to specific needs of KBE application development. SysML includes several diagrams that are not relevant for developing KBE applications. Thus, with the goal of tailoring the language to the precise scope being addressed, a subset of SysML called *SysML-for-KBE* (adapted from the *SysML-Lite* originally proposed by Friedenthal et al. [10]) is proposed as the modeling language to be used to support formal knowledge modeling in the development of KBE applications.

SysML-for-KBE includes five of the nine SysML diagrams, and a small subset of the available language features for each diagram kind. The included diagrams are Requirements, Use Case, Activity, Block Definition, and Internal Block, and are briefly explained as follows:

1. The **Requirements Diagram** is used to represent text-based requirements and their relationships to other requirements, design elements, and test cases, supporting requirements traceability.
2. The **Use Case Diagram** is used to represent the functionality of a system in terms of how external entities, also known as actors, use it to achieve a specific set of objectives.
3. The **Activity Diagram** is used to represent behavior in terms of the order in which actions execute, based on inputs, outputs, and control, as well as how the actions transform inputs to outputs.
4. The **Block Definition Diagram** is used to represent structural elements called blocks, and their composition and classification.
5. The **Internal Block Diagram** is used to represent the interconnections and interfaces between the components of a block.

Detailed notation tables that describe the symbols used in these diagrams can be found in [10].

Additionally, SysML-for-KBE includes two special matrices that help engineers in assessing the compliance of requirements and guaranteeing that all the tasks that must be performed by a KBE application are incorporated in the source code. These matrices are the following:

1. The **Satisfy Requirements Matrix**, which specifies the product and process elements that satisfy a given requirement.
2. The **Allocation Matrix**, which identifies the product elements that perform a given task/action.

Examples of each diagram kind and matrix utilized in SysML-for-KBE are presented in Section 6.2.1.

Furthermore, SysML-for-KBE was tailored through the definition of specific stereotypes. These stereotypes were introduced to enhance the ability of the modeling language to capture the knowledge necessary for developing KBE applications. In alignment with the concepts identified in the Knowledge Ontology, custom stereotypes were created (see Figure 5.8) to represent the following concepts:

- Input, Attribute, Part, Action, Method, and External Function. These stereotypes were defined by extending the Property Metaclass of SysML.
- External Tool and External Service. These stereotypes were defined by extending the Element Metaclass of SysML.
- Stakeholder Need. This stereotype was defined by specializing the Requirement Stereotype of SysML.

The selection of the appropriate metaclass for each stereotype was carefully made to restrict their application to specific SysML elements. This deliberate choice ensures the proper usage of the stereotypes by domain experts and KBE developers during the modeling process. The correct usage of these stereotypes in the models is essential to enable the automatic translation of knowledge from the models into KBE application code (further details on this will be discussed in Section 5.4.1).

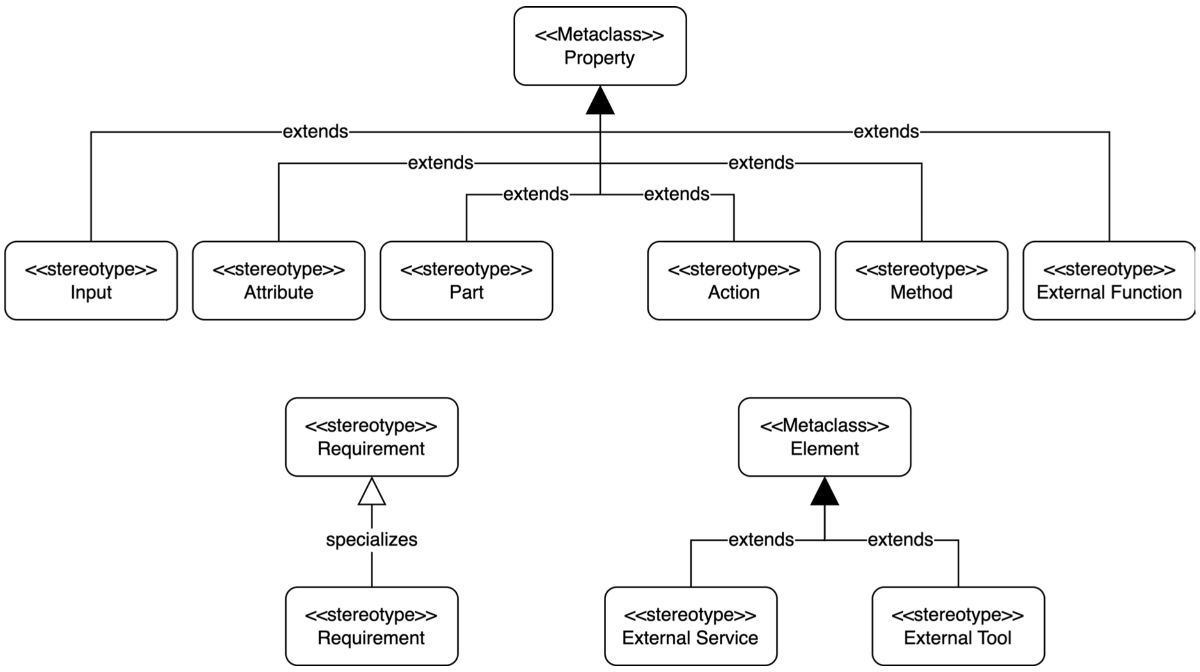
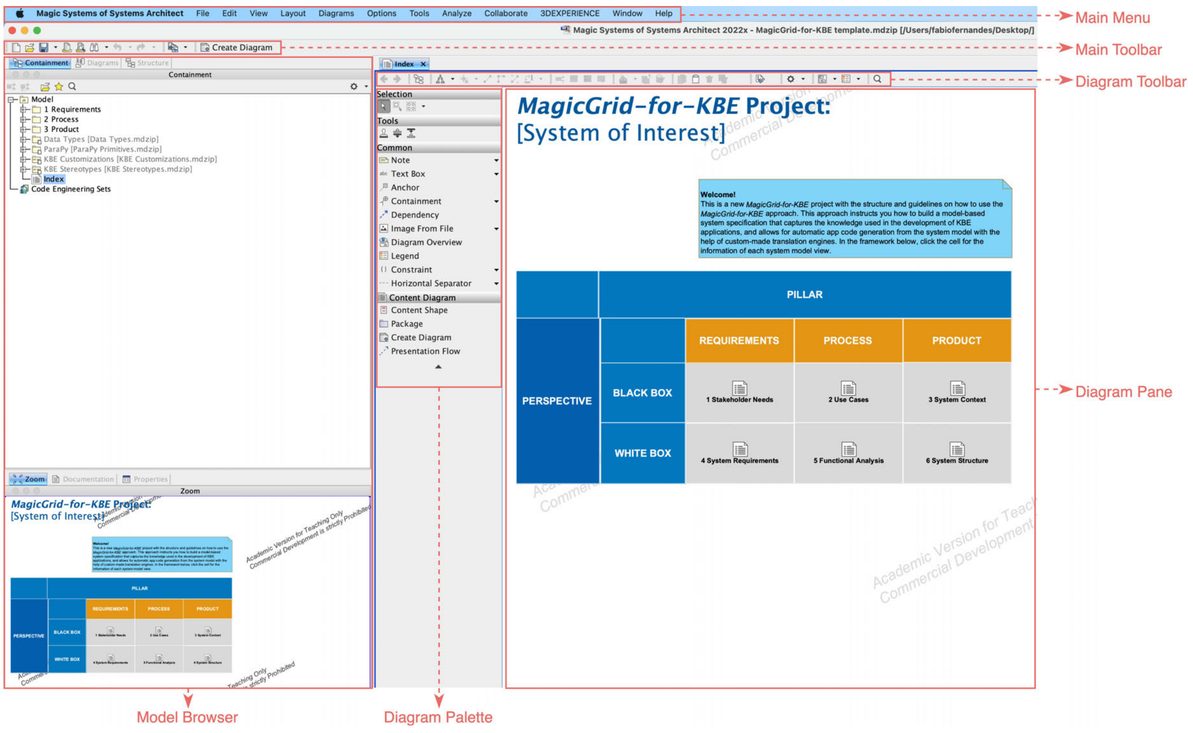


Figure 5.8: Definition of the KBE-specific stereotypes for SysML

5.3.2. Modeling Tool

For the purpose of this research, any SysML-compliant modeling tool could have been selected. Based on advice from the literature (see Section 3.4.2), the chosen modeling tool was the industry-leading Magic Systems of Systems Architect (MSoSA) / Cameo Systems Modeler (CSM) developed by Dassault Systèmes. A key feature of MSoSA that motivated its selection as the chosen modeling tool for the developed framework was its capability to automatically export the SysML models to neutral language models in the XMI industry-standard format. This feature played a crucial role in enabling the subsequent automatic generation of KBE application code from the models.

A breakdown of the user interface of MSoSA is presented in Figure 5.9. The main concepts presented are also common to most modeling tools currently available.



Welcome!
This is a new *MagicGrid-for-KBE* project with the structure and guidelines on how to use the *MagicGrid-for-KBE* approach. This approach instructs you how to build a model-based system specification that captures the knowledge used in the development of KBE applications, and allows for automatic app code generation from the system model with the help of custom-made translation engines. In the framework below, click the cell for the information of each system model view.

Figure 5.9: Screenshot of the User Interface of Magic Systems of Systems Architect highlighting the main components of the modeling tool

In summary, the typical user interface of an MBSE modeling tool is composed of various components such as the main menu, main toolbar, diagram toolbar, model browser, diagram palette, and diagram pane. These components are described as follows:

- The **main menu** is the top-level menu in a modeling tool that typically contains options for opening, saving, and closing models, as well as options for configuring the tool's settings, accessing help documentation, and performing other administrative tasks.
- The **main toolbar** is a collection of buttons or icons located below the main menu in a modeling tool that provides quick access to commonly used commands, such as creating new diagrams, adding elements to the model, and saving changes.
- The **diagram toolbar** is a collection of buttons or icons located above a diagram pane in a modeling tool that provides quick access to commands specific to the currently selected diagram, such as changing the view or layout of the diagram, adding annotations or text labels, and adjusting the zoom level.
- The **model browser** is a pane or window in a modeling tool that displays a hierarchical view of the model's elements and their relationships. It is typically used to navigate and manage the model's contents, and may include features such as filtering, searching, and grouping elements by type or other criteria.
- The **diagram palette** is a collection of pre-defined shapes or symbols that can be used to create diagrams in a modeling tool. It is typically displayed as a pane or window that can be opened or closed as needed, and may include multiple categories of shapes or symbols based on the modeling language or notation being used.
- Finally, the **diagram pane** is the main area of a modeling tool where diagrams are created, edited,

and viewed. It typically occupies the largest portion of the tool's interface and can be customized to show or hide various elements, such as grid lines, rulers, and object snap settings.

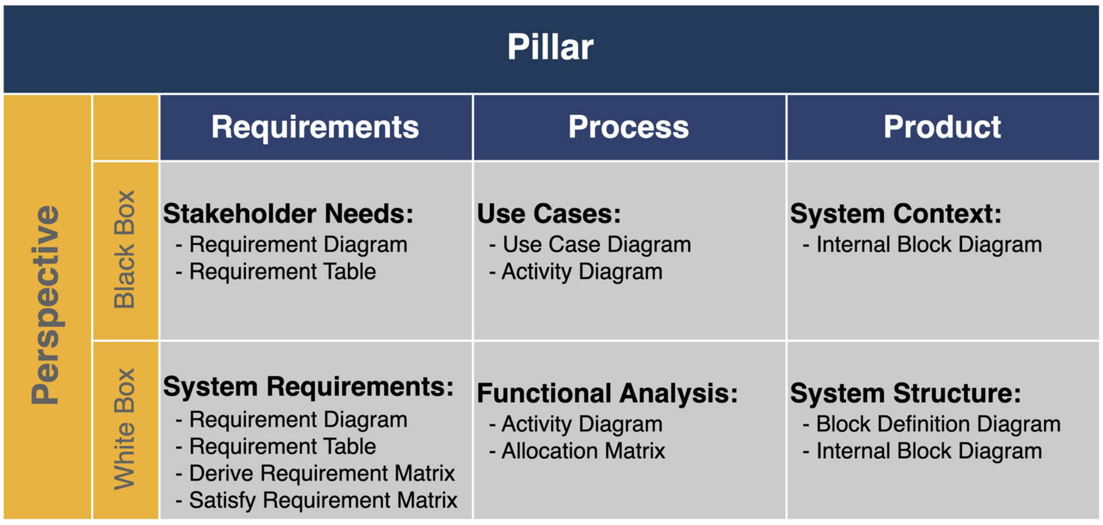
Understanding these different components of an MBSE modeling tool's user interface is essential for efficient and effective modeling.

5.3.3. Modeling Method

Effective use of a modeling language requires a well-defined MBSE method. SysML has been developed with the aim of supporting multiple systems engineering methods. The selection of a particular method is based on various criteria, including the method's ease of use, ability to address the range of systems engineering concerns, and the level of tool support [10]. Taking these criteria into account, the MagicGrid method [44] was adopted and tailored for the purpose of this research work.

The customizations made to MagicGrid aimed to tailor the method for the development of KBE applications and to simplify the modeling process by reducing the number of concepts, constructs, and steps required. This led to the development of *MagicGrid-for-KBE*, a modified version of the MagicGrid method that is intended to provide guidance to engineers as they navigate the modeling process for developing KBE applications. MagicGrid-for-KBE answers important questions such as how to organize the model, what the modeling workflow is, what artifacts should be produced in each step of the workflow, and how these artifacts are connected.

The MagicGrid-for-KBE method can be represented as a simplified Zachman style matrix. This matrix consists of the three main ontological concepts that are essential when dealing with KBE applications, namely Requirements, Process, and Product, which are represented as columns. Furthermore, the method considers two perspectives of the problem domain, namely Black Box and White Box, which are represented as rows. The Black Box perspective focuses on the system's external behavior and functionality, without delving into its internal workings or implementation details. The White Box perspective involves a more detailed examination of the system, focusing on its internal structure, components, and inner workings. Each element in the matrix represents a view of the system model, which can consist of one or more presentation artifacts. A presentation artifact can be a (SysML) diagram, or matrix. The MagicGrid-for-KBE method is depicted in Figure 5.10, providing a visual illustration of the method's structure and organization, as well as the presentation artifacts that each system model view may consist of. The modeling workflow is defined by the order of the cells in a left-to-right and top-to-bottom manner.



		Pillar		
Perspective		Requirements	Process	Product
	Black Box	**Stakeholder Needs:** - Requirement Diagram - Requirement Table	**Use Cases:** - Use Case Diagram - Activity Diagram	**System Context:** - Internal Block Diagram
	White Box	**System Requirements:** - Requirement Diagram - Requirement Table - Derive Requirement Matrix - Satisfy Requirement Matrix	**Functional Analysis:** - Activity Diagram - Allocation Matrix	**System Structure:** - Block Definition Diagram - Internal Block Diagram

Figure 5.10: Matrix representation of the MagicGrid-for-KBE method illustrating the different system model views. The gray cells highlight the system model views, indicated by the text in bold. The dashed points within each cell represent the various types of presentation artifacts that can be included in each view.
Source: Adapted from [45]

In the problem domain analysis, two phases are carried out to define the System of Interest (SoI), i.e., the KBE application. In the first phase, the SoI is treated as a black box, focusing on how it interacts with the environment without getting any knowledge about its internal structure and behavior. In the second phase, the SoI is considered from a white-box perspective, allowing for an understanding of its expected behavior and conceptual structure. Both phases involve defining the requirements, process, and product of the SoI, with the only difference being the perspective taken. The following paragraphs provide a description of each view of the system model, presented in the order that corresponds to the modeling workflow. An overview of these system model views, with (simplified) examples of the presentation artifacts that each view may contain, is illustrated in Figure 5.12.

1. Stakeholder Needs

The Stakeholder Needs cell represents information gathered from various stakeholders of the SoI, including primary user requirements, government regulations, policies, procedures, and more. The later refinements in the model make these stakeholder needs structured and formalized.

SysML requirements diagrams or tables, or both, are used to capture and document stakeholder needs.

2. Use Cases

The Use Cases cell identifies how the SoI is expected to interact with the user and other systems. In this cell, functional stakeholder needs are refined by defining one or more use cases and use case scenarios. In comparison to stakeholder needs, use cases provide a more detailed and precise description of the desired behavior of the system and the intended outcomes of its use. Each use case must belong to one or more system contexts defined in the System Context cell. Each use case must have a primary scenario and can also include alternative scenarios.

SysML use case diagrams are used to capture and document use cases, while SysML activity diagrams are used to capture and document use case scenarios.

3. System Context

The System Context cell defines an external view of the SoI at a high level of abstraction. It outlines all external entities that interact with the system but are not part of it. In addition to the SoI itself, the elements of a specific system context can encompass external systems and users that interact with the SoI. These entities and their interactions are considered as inputs and outputs of the SoI and are crucial for understanding the system's behavior in its operational environment.

SysML internal block diagrams are used to capture and document the system context.

4. System Requirements

The System Requirements cell specifies the technical requirements that the system must satisfy. These requirements are derived from stakeholder needs during the definition of the functional analysis and system structure view specifications, and can be updated at any time during the development of the system architecture. Unlike stakeholder needs, system requirements are verifiable, and traceability relationships must be established to assert which elements of the system architecture fulfill which system requirements.

SysML requirements diagrams or tables, or both, are used to capture and document system requirements, while SysML satisfy requirements matrices are used to capture and document which elements of the systems architecture fulfill each requirement.

5. Functional Analysis

The Functional Analysis cell is the logical continuation of the Use Cases cell. It involves decomposing each top-level function performed by the SoI, previously identified during the black-box analysis, and repeating this process iteratively until the desired level of detail is reached for the given project. This breakdown allows for the identification of the conceptual subsystems, or functional blocks, responsible for performing each function. The goal is to ensure that all functions required by the system are identified and analyzed thoroughly, allowing for a comprehensive understanding of the system's behavior.

SysML activity diagrams are used to capture and document the functional analysis. From the resulting model of the expected system behavior, it is possible to extract an activity decomposition map that begins with the high-level goals of the SoI and ends with its expected functions.

6. System Structure

While the functional analysis cell helps identify conceptual subsystems, the System Structure cell captures them in the model. These conceptual subsystems are a group of interconnected and interactive parts that perform one or more expected functions of the SoI. Each conceptual subsystem should be decomposed into a more elementary structure, until the desired level of detail is reached for a given project.

A combination of both SysML block definition diagrams and SysML internal block diagrams are used to capture and document this view.

Upon successful capture of the conceptual subsystems in the model, functions are allocated to each subsystem in order to specify which functions they perform. It is important to note that the granularity of expected system behavior and structure must be consistent in each level of detail, as illustrated in Figure 5.11. SysML allocation matrices are used to capture and document the traceability between elements of the functional analysis and elements of the system structure.

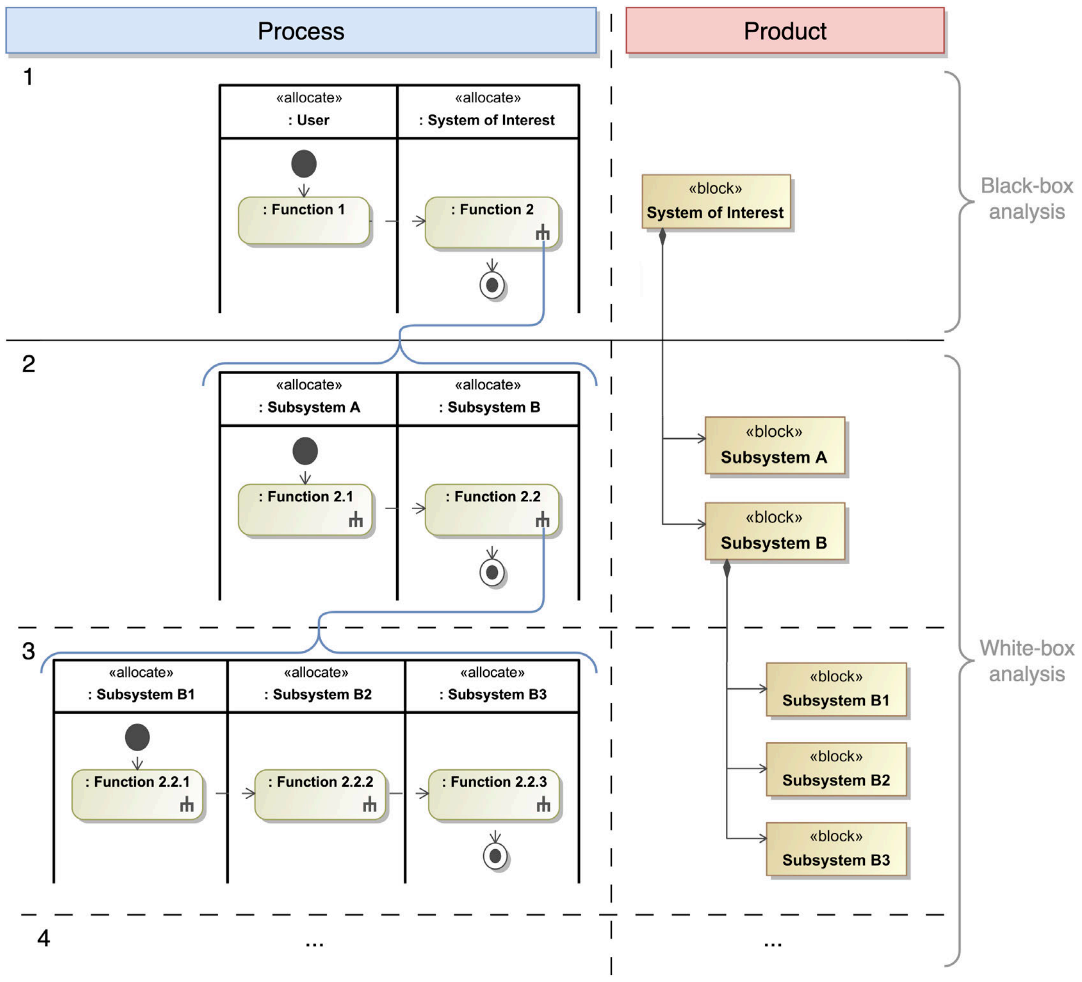


Figure 5.11: Example of consistency in the granularity between the system behavior and structure at each level of detail. Source: Adapted from [44]

A template has been developed within the modeling tool (MSoSA) to facilitate the implementation of the MagicGrid-for-KBE method. This template offers users the advantage of skipping the setup process for a new project and provides them with an appropriate starting point for their modeling activities. Furthermore, the template serves as a guide throughout the modeling process by offering direct access to the various system model views and providing textual descriptions of the expected knowledge to be captured in each view. Furthermore, it ensures the proper organization of the model and includes dedicated tables and maps that give the user a better overview of the model. The initial frame/page of the template can be seen within the diagram pane shown in Figure 5.9.

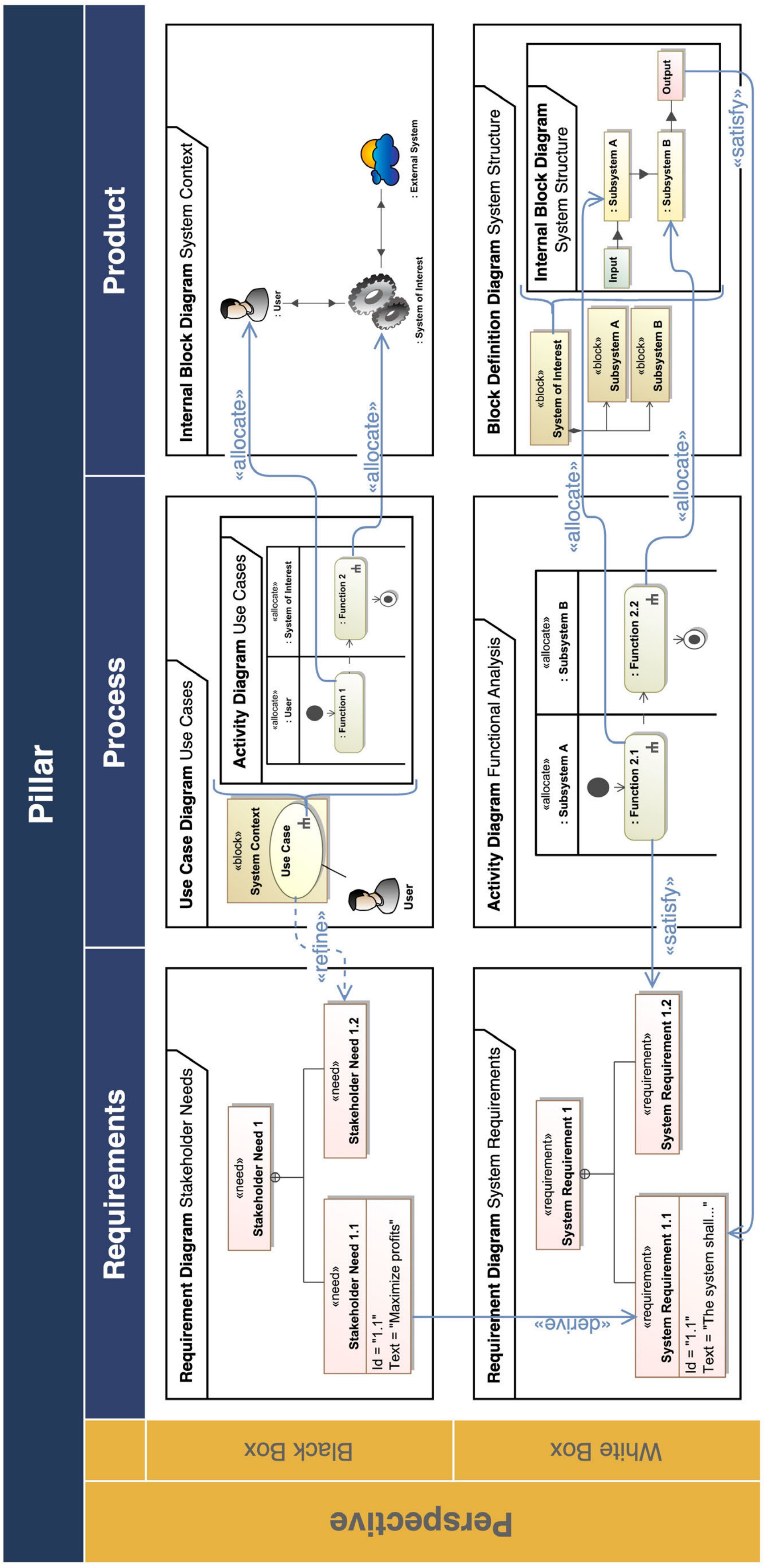


Figure 5.12: Overview of the MagicGrid-for-KBE method illustrating (simplified) examples of the presentation artifacts contained in each system model view.
Source: Adapted from [45]

5.4. Automatic Generation of KBE Application Code

The third part of the framework development focused on developing dedicated translators to parse the knowledge model and automatically generate source code for the targeted KBE system. Furthermore, a Graphical User Interface (GUI) was developed to facilitate the process of translating the different knowledge representations, and to increase the transparency of KBE applications by allowing traceability of requirements from the knowledge model to the application code.

5.4.1. Translation Engines

Following the generation of a (complete) SysML model in MSoSA, the information contained within can be exported to a file format that is compatible with other tools for additional manipulations and transformations. To enable this, MSoSA employs the use of XML Metadata Interchange (XMI), a standard established by the Object Management Group (OMG) for metadata exchange using Extensible Markup Language (XML). By selecting the "*UML 2.5 XMI File*" option from MSoSA's export menu, an XML file (in the XMI standard) is created containing all the information present in the SysML model, which serves as the main entry point to the developed technological solution.

The developed technological solution for translating the various knowledge representations is composed of two translation engines created using Python. The first translation engine converts the XML file exported from MSoSA to a intermediate file in JSON format, while the second translation engine converts the JSON file to the target KBE language (ParaPy). This translation process is illustrated in Figure 5.13.

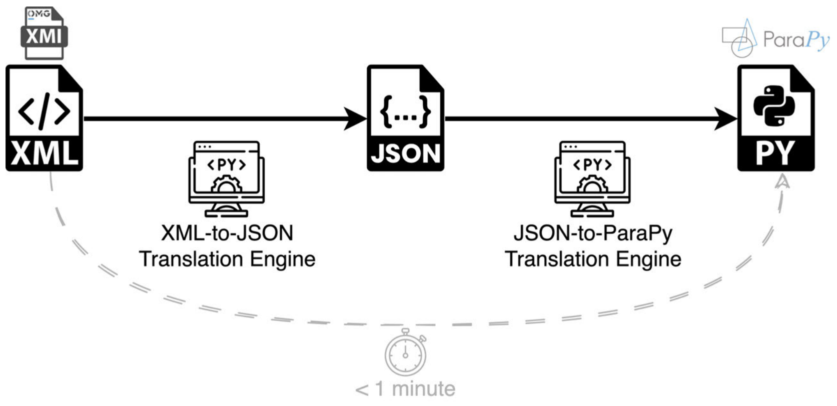

OMG XMI
XML
JSON
PY
ParaPy
<PY>
XML-to-JSON Translation Engine
<PY>
JSON-to-ParaPy Translation Engine
< 1 minute

Figure 5.13: Illustration of the translation process from MSoSA's XMI to ParaPy's Python-based code

During the development of the translation engines, the following best practices for software development were followed:

1. The primary focus was on writing code that is clear and easily understandable, striving for self-documentation whenever possible. This involved using meaningful names for variables and functions, dividing the code into small, modular functions, adhering to a consistent coding style following PEP8 conventions, employing a logical and meaningful structure for organizing the code, and ensuring that the code was concise and expressive.
2. In instances where the code remained unclear or could not be further simplified, comments were included to provide additional clarity and explanations.

3. Regardless of the code's clarity and the presence of comments, functions and methods were accompanied by docstrings, which serve as documentation to describe their purpose, parameters, and return values.

By implementing these practices, the code aimed to convey its intentions and functionality effectively, minimizing the need for extensive external documentation.

The elements involved in the knowledge translation process are discussed in the following subsections.

Intermediate JSON File

The main reason for converting the XML file exported from MSoSA to an intermediate file in JSON format was to facilitate the practical implementation of code generation. While it is technically possible to directly convert the XML file to ParaPy code, this approach is considerably more complex.

The XML file contains a substantial amount of information that is irrelevant for automatically generating the KBE application source code, such as geometrical and positional details about diagrams and their elements. Moreover, the pertinent information for source code generation is scattered across different sections of the large XML file. This poses challenges in parsing the file to extract the necessary information while simultaneously writing the ParaPy code. Consequently, the translation engine's code would become overly intricate and difficult to comprehend and develop.

To address these complexities, a two-step process is adopted. First, the necessary information is extracted from the XML file and stored in an intermediate JSON file. Then, this intermediate file is utilized to generate the ParaPy code.

The decision to employ JSON as the intermediate file format instead of XML was based on two considerations. Firstly, JSON offers improved human-readability compared to XML, facilitating the identification of information within the file and simplifying the development of the second translation engine responsible for translating the information into ParaPy code. Secondly, JSON aligns with the format used by ParaPy's Visual Editor to store model information (albeit with a slightly different schema due to the differing nature of knowledge captured in ParaPy's Visual Editor models). Therefore, JSON emerged as a viable option for this research.

Furthermore, the existence of an intermediate file (in JSON format) offers the advantage of facilitating future expansions to the developed framework by providing an additional entry point to the knowledge translation process.

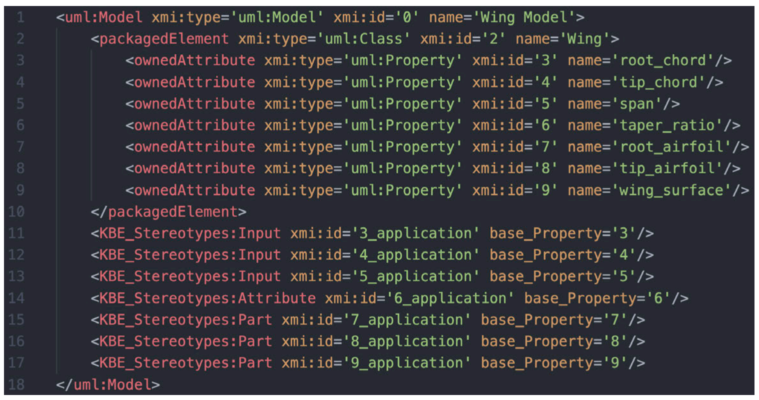
In the future, users of the MBSE-for-KBE framework may not be restricted to using MSoSA as the exclusive modeling tool for creating their SysML models and exporting XML files. They may have the flexibility to choose alternative tools capable of exporting files in different formats. Developing a translation engine capable of directly converting these formats to KBE code would likely be complex, as previously explained.

Therefore, if these users develop their own translation engine that can convert the format exported from their modeling tool into a JSON format adhering the JSON schema defined in this research, they can utilize the JSON-to-ParaPy translator developed in this research to automatically generate source code for the ParaPy KBE system. This flexibility allows users to leverage various modeling tools based on their preferences or project requirements, provided they also create the *[exported_file_format]*-to-JSON translator.

Another benefit of having an intermediate JSON file is that users who employ MSoSA as their modeling tool but target a different KBE system can still export their models to the XML format provided by MSoSA. Subsequently, they can extract the relevant information from these files into a JSON file using the XML-to-JSON translation engine developed in this research. These users would then need to develop their own translation engine to automatically generate code for their target KBE system from the JSON file (i.e., JSON-to-*[target_KBE_system_format]* translation engine. It is important to note that in this approach, modifications or extensions may be necessary to the schema of the JSON file to accommodate the ontology of the specific KBE system being targeted. This implies that the XML-to-JSON translation engine developed in this research would require some changes or additions. On the positive side, these users would already have the XML-to-JSON translator developed in this research as a foundation, thereby eliminating the need to create their own XML-to-JSON translator from scratch.

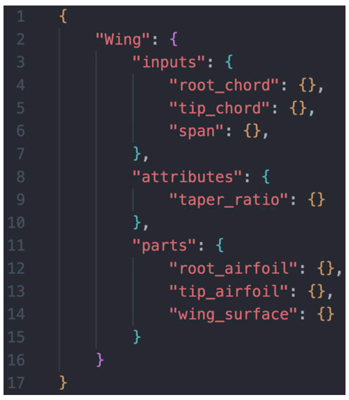
XML-to-JSON Translation Engine

The XML-to-JSON translation engine parses the XML file and identifies the various model elements (i.e., data within the file) relevant for generating the source code of the KBE application based on their tags and attributes. Additionally, it determines the type of each element based on the stereotype applied to it during the modeling process, enabling the translation of the elements into the appropriate ParaPy slots. Finally, it consolidates this extracted information into a well-structured, tree-like format represented in JSON. This consolidation significantly reduces the file size compared to the original XML file while facilitating easy access and management of the information within the JSON structure. A simplified example of the data contained in an XML file is presented in Figure 5.14a. And the corresponding JSON file is presented in Figure 5.14b.



```xml
<uml:Model xmi:type='uml:Model' xmi:id='0' name='Wing Model'>
    <packagedElement xmi:type='uml:Class' xmi:id='2' name='Wing'>
        <ownedAttribute xmi:type='uml:Property' xmi:id='3' name='root_chord'/>
        <ownedAttribute xmi:type='uml:Property' xmi:id='4' name='tip_chord'/>
        <ownedAttribute xmi:type='uml:Property' xmi:id='5' name='span'/>
        <ownedAttribute xmi:type='uml:Property' xmi:id='6' name='taper_ratio'/>
        <ownedAttribute xmi:type='uml:Property' xmi:id='7' name='root_airfoil'/>
        <ownedAttribute xmi:type='uml:Property' xmi:id='8' name='tip_airfoil'/>
        <ownedAttribute xmi:type='uml:Property' xmi:id='9' name='wing_surface'/>
    </packagedElement>
    <KBE_Stereotypes:Input xmi:id='3_application' base_Property='3'/>
    <KBE_Stereotypes:Input xmi:id='4_application' base_Property='4'/>
    <KBE_Stereotypes:Input xmi:id='5_application' base_Property='5'/>
    <KBE_Stereotypes:Attribute xmi:id='6_application' base_Property='6'/>
    <KBE_Stereotypes:Part xmi:id='7_application' base_Property='7'/>
    <KBE_Stereotypes:Part xmi:id='8_application' base_Property='8'/>
    <KBE_Stereotypes:Part xmi:id='9_application' base_Property='9'/>
</uml:Model>
```

(a) Example of the data contained in an XML file. The tags, attribute names, and attribute values of each model element are denoted by red, yellow, and green text, respectively.

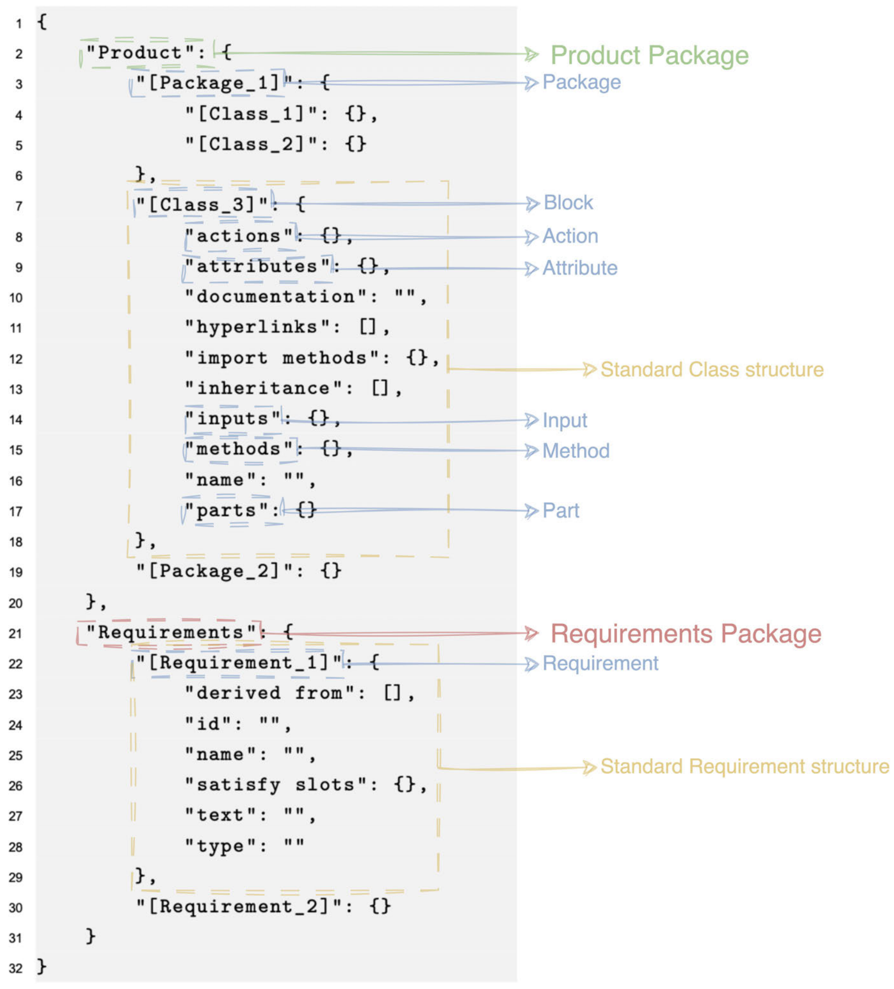


```json
{
    "Wing": {
        "inputs": {
            "root_chord": {},
            "tip_chord": {},
            "span": {},
        },
        "attributes": {
            "taper_ratio": {}
        },
        "parts": {
            "root_airfoil": {},
            "tip_airfoil": {},
            "wing_surface": {}
        }
    }
}
```

(b) Example of the data contained in the corresponding JSON file

Figure 5.14: Comparison of the data contained in an XML file and its corresponding JSON representation after its translation by the XML-to-JSON translation engine

To illustrate the link between the Knowledge Ontology developed in this research and the structure of the JSON file, a slightly more elaborate (general) example of the data structure of the JSON file that is automatically generated by the XML-to-JSON translation engine is shown in Figure 5.15. The complete formal description of the data schema of the JSON files is presented in Appendix B.



```
{
    "Product": {
        "[Package_1]": {
            "[Class_1]": {},
            "[Class_2]": {}
        },
        "[Class_3]": {
            "actions": {},
            "attributes": {},
            "documentation": "",
            "hyperlinks": [],
            "import methods": {},
            "inheritance": [],
            "inputs": {},
            "methods": {},
            "name": "",
            "parts": {}
        },
        "[Package_2]": {}
    },
    "Requirements": {
        "[Requirement_1]": {
            "derived from": [],
            "id": "",
            "name": "",
            "satisfy slots": {},
            "text": "",
            "type": ""
        },
        "[Requirement_2]": {}
    }
}
```

Figure 5.15: Example of the data structure of the JSON file. The dashed boxes in red, green, and blue illustrate the relationship between the elements of the file and the elements of the Knowledge Ontology. The dashed boxes in yellow illustrate the standardized representation of different types of model elements contained in the JSON file.

JSON-to-ParaPy Translation Engine

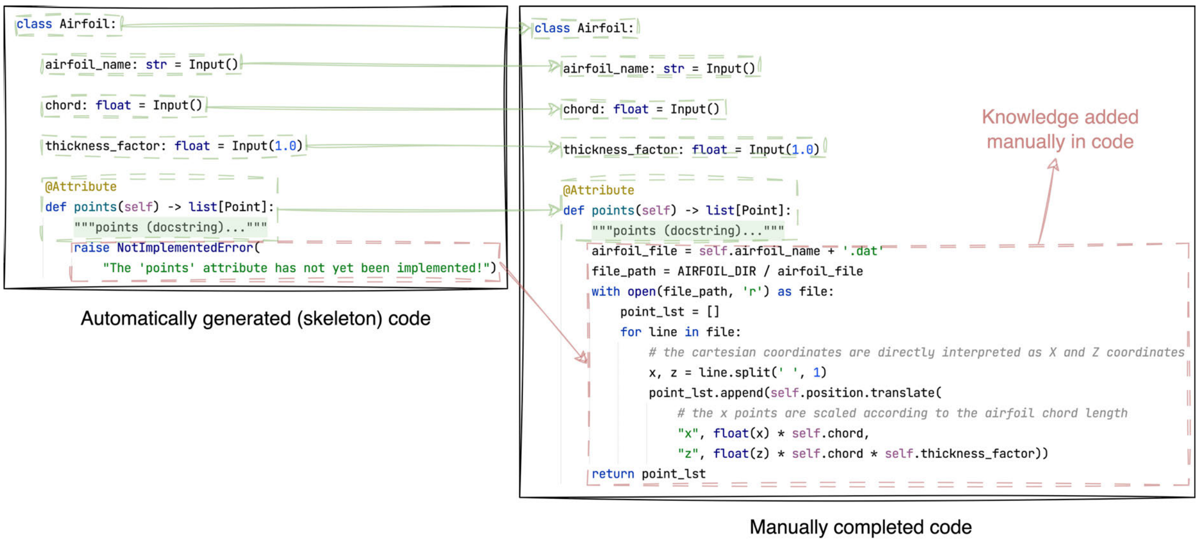
The JSON-to-ParaPy translation engine parses the JSON file generated by the XML-to-JSON translation engine. It identifies the contents of the JSON file by analyzing their internal (standardized) structure and generates Python modules (files) with the necessary ParaPy code. Each module corresponds to the information extracted from a *Class* defined within the JSON file due to the mapping established between the Knowledge Ontology and the ParaPy Ontology (as explained in Section 5.2.3).

Furthermore, this translation engine also generates the Packages defined in the JSON file and places the Modules inside the corresponding *Package*, ensuring the correct organization of the KBE application source code. The internal structure of the KBE application that would have been generated from the example JSON file presented in Figure 5.15 is illustrated in the directory tree presented in Figure 5.16. Note that only elements of the Product package are used to generate the KBE application. No application code is generated from the elements of the Requirements package, as these elements are only used for providing requirements traceability through the Graphical User Interface developed in this research (see Section 5.4.2).

```
KBE_Application
 ├─package_1
 │  ├─class_1.py
 │  └─class_2.py
 ├─class_3.py
 └─package_2
```

Figure 5.16: Directory tree showcasing an example of the internal structure of a (generic) KBE application automatically generated by the JSON-to-ParaPy translation engine

Depending on the level of detail of the system knowledge model, and thus in the JSON file, the JSON-to-ParaPy translation engine can generate either a code skeleton (i.e., key classes, slots and their relationships in a correct file structure) or the complete application code that is ready for use. An example highlighting the similarities and differences between the automatically generated skeleton code and the manually complete code is presented in Figure 5.17.



```python
class Airfoil:

    airfoil_name: str = Input()

    chord: float = Input()

    thickness_factor: float = Input(1.0)

    @Attribute
    def points(self) -> list[Point]:
        """points (docstring)..."""
        raise NotImplementedError(
            "The 'points' attribute has not yet been implemented!")
```

Automatically generated (skeleton) code

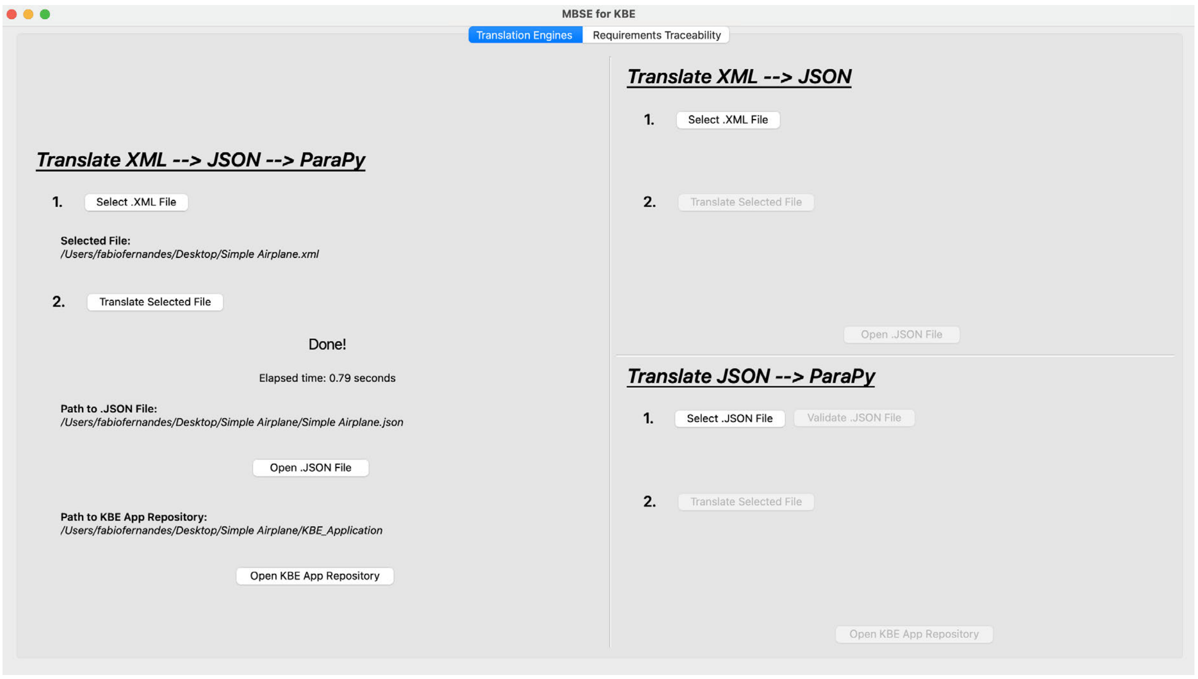
```python
class Airfoil:

    airfoil_name: str = Input()

    chord: float = Input()

    thickness_factor: float = Input(1.0)

    @Attribute
    def points(self) -> list[Point]:
        """points (docstring)..."""
        airfoil_file = self.airfoil_name + '.dat'
        file_path = AIRFOIL_DIR / airfoil_file
        with open(file_path, 'r') as file:
            point_lst = []
            for line in file:
                # the cartesian coordinates are directly interpreted as X and Z coordinates
                x, z = line.split(' ', 1)
                point_lst.append(self.position.translate(
                    # the x points are scaled according to the airfoil chord length
                    "x", float(x) * self.chord,
                    "z", float(z) * self.chord * self.thickness_factor))
        return point_lst
```

Manually completed code

Figure 5.17: Comparison between the automatically generated skeleton code (on the left) and the manually completed code (on the right). The green and red dashed boxes highlight the similarities and differences, respectively, between the two codes.

5.4.2. Graphical User Interface

To enhance the usability of the translation engines and provide additional functionalities for traceability of requirements, a graphical user interface (GUI) was developed. The GUI consists of two main tabs: the Translation Engines tab and the Requirements Traceability tab.



Selected File:
/Users/fabiofernandes/Desktop/Simple Airplane.xml

Done!

Elapsed time: 0.79 seconds

Path to .JSON File:
/Users/fabiofernandes/Desktop/Simple Airplane/Simple Airplane.json

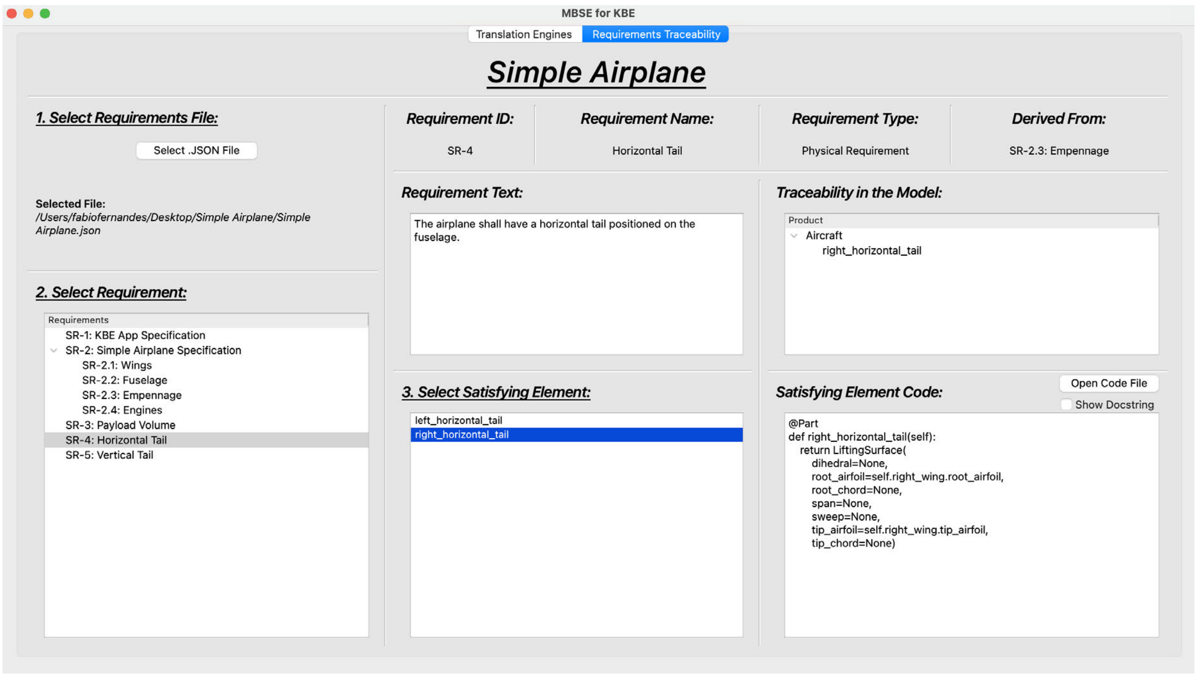
Path to KBE App Repository:
/Users/fabiofernandes/Desktop/Simple Airplane/KBE_Application

Figure 5.18: GUI Translation Engines Tab

The Translation Engines tab, depicted in Figure 5.18, is divided into three sections corresponding to different types of possible file translations/conversions. This division was implemented to offer flexibility and modularity, allowing for future developments as discussed above. These sections are described as follows:

- The left section, **Translate XML** → **JSON** → **ParaPy**, serves as the main functionality of the framework, enabling the user to directly translate an XML file to JSON and create the KBE Application simultaneously.
- The top right section, **Translate XML** → **JSON**, enables the translation of XML files into the intermediate file in JSON format. The idea behind this section is to accommodate users who generate SysML models using MSoSA but are not using ParaPy as their target KBE system, and thus intend to use a different translation engine to generate their KBE Application.
- The bottom right section, **Translate JSON** → **ParaPy**, enables the automatic creation of a KBE Application from a intermediate file in JSON format. This section intends to accommodate users whose target KBE system is ParaPy but are not using MSoSA to generate their SysML models. Instead, these users can employ alternative modeling tools that export models in a format different from the XMI used in this research. These models, in a different format, can then be translated into a JSON file containing the relevant model information using translation engines that the users need to develop. Users can then leverage this section to translate the JSON file and generate a ParaPy-based KBE Application.

  Additionally, to ensure the validity of a JSON file that was not automatically generated using the provided XML-to-JSON translation engine, users can validate the data structure of their JSON file by clicking the *'Validate .JSON File'* button, which utilizes the JSON Schema outlined in Appendix B to inform the user whether the file adheres to the required schema and is deemed valid.


MBSE for KBE

Translation Engines | Requirements Traceability

Simple Airplane

1. Select Requirements File:

Select .JSON File

Selected File:
/Users/fabiofernandes/Desktop/Simple Airplane/Simple Airplane.json

2. Select Requirement:

Requirements
- SR-1: KBE App Specification
- SR-2: Simple Airplane Specification
  - SR-2.1: Wings
  - SR-2.2: Fuselage
  - SR-2.3: Empennage
  - SR-2.4: Engines
- SR-3: Payload Volume
- SR-4: Horizontal Tail
- SR-5: Vertical Tail

Requirement ID:	*Requirement Name:*	*Requirement Type:*	*Derived From:*
SR-4	Horizontal Tail	Physical Requirement	SR-2.3: Empennage

Requirement Text:

The airplane shall have a horizontal tail positioned on the fuselage.

Traceability in the Model:

Product
- Aircraft
  - right_horizontal_tail

3. Select Satisfying Element:

- left_horizontal_tail
- right_horizontal_tail

Satisfying Element Code:

Open Code File

Show Docstring

```
@Part
def right_horizontal_tail(self):
    return LiftingSurface(
        dihedral=None,
        root_airfoil=self.right_wing.root_airfoil,
        root_chord=None,
        span=None,
        sweep=None,
        tip_airfoil=self.right_wing.tip_airfoil,
        tip_chord=None)
```

Figure 5.19: GUI Requirements Traceability Tab

The Requirements Traceability tab, depicted in Figure 5.19, provides functionality for tracing requirements from the knowledge model to the application code. To initiate the process of tracing requirements, users are required to:

1. Select in section **'1. Select Requirements File'** a JSON file containing information about the product and requirements of the KBE application (this is the same JSON file that was used to previously generate the KBE app using the appropriate translation engine).

2. After selecting a file, the **'2. Select Requirement'** tree widget is automatically populated, allowing users to choose a specific requirement. Upon selecting a requirement, detailed information about the selected requirement is then displayed to the user, including the requirement's ID, name, type, text, and any requirements from which it is derived.

3. The **'3. Select Satisfying Element'** list widget is then automatically populated with elements that satisfy the selected requirement. Users are expected to choose one of the elements from this list. Upon selection, the path to the element in the SysML model is displayed in the 'Traceability in the Model' tree widget, and the code corresponding to that element is extracted from the Python Module that defines its containing class. The extracted code is presented in the 'Satisfying Element Code' text box. Additionally, users can choose to view the documentation of the code slot by selecting the 'Show Docstring' checkbox.

4. Finally, users have the option to open the Python module/file that contains the displayed code in their preferred integrated development environment (IDE) by clicking the *'Open Code File'* button.

Overall, the developed GUI facilitates the utilization of the translation engines and incorporates features that enhance traceability of requirements from the knowledge model to the application code.

5.5. Proposed KBE Application Development Process

To improve the current KBE app development process, a new development process that utilizes the developed framework as a central component is proposed. The newly proposed KBE app development process for ParaPy-based KBE applications is schematically represented in Figure 5.20.

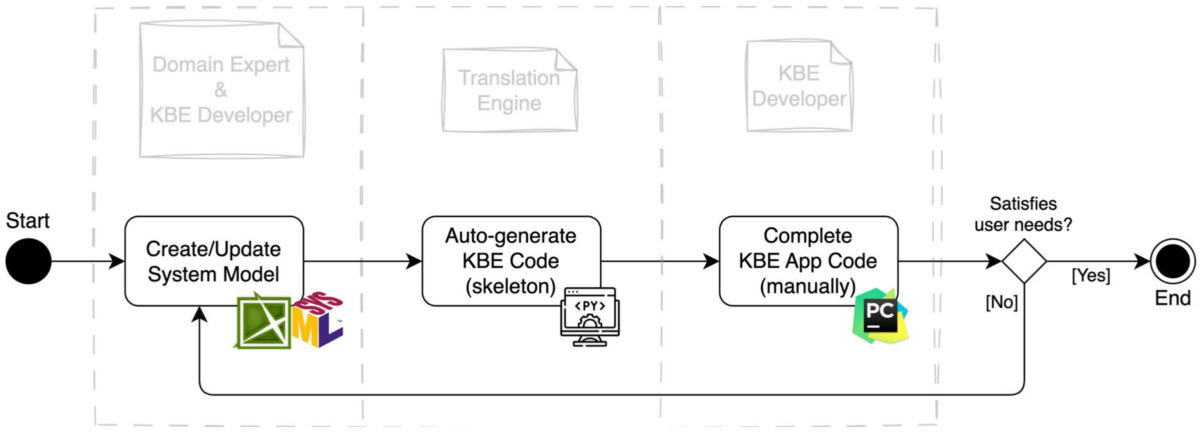

Domain Expert & KBE Developer
Translation Engine
KBE Developer
Start
Create/Update System Model
Auto-generate KBE Code (skeleton)
Complete KBE App Code (manually)
Satisfies user needs?
[Yes]
[No]
End

Figure 5.20: Newly Proposed KBE Application Development Process. The gray notes indicate the stakeholders responsible for performing each task. The icons indicate the tool used to perform each task.

The proposed development process for KBE applications using the framework developed in this research is described as follows:

1. A team of domain experts and KBE developers create a System Model of the KBE application in Magic Systems of Systems Architect, using SysML as the modeling language and MagicGrid-for-KBE as the modeling method.
2. Once the system model reaches the desired level of maturity, the translation engines developed in this research should be used automatically generate the (skeleton) code of the KBE application.
3. Finally, the detailed knowledge and rules of the KBE application should be manually completed by a team of KBE developers in an IDE like PyCharm. Although it is possible to directly embed this information within the Internal Block Diagrams of the system model, the author recommends using an integrated development environment (IDE) such as PyCharm for this purpose. The rationale behind this recommendation stems from the fact that MSoSA does not provide the same functionalities for writing code as IDEs, which greatly facilitate coding by offering features such as syntax highlighting, code completion, formatting, analysis, and refactoring.

In the proposed KBE app development process, a potential issue arises when modifications are made to the knowledge model after manual code additions, as regenerating the code from the model would result in the loss of the previously added code. This constitutes a current limitation of the developed framework and will be addressed in the future recommendations presented in Section 8.2.

The essential set of steps from the MagicGrid-for-KBE method required to successfully develop KBE applications using the framework developed in this research are presented as an activity diagram in Figure 5.21.

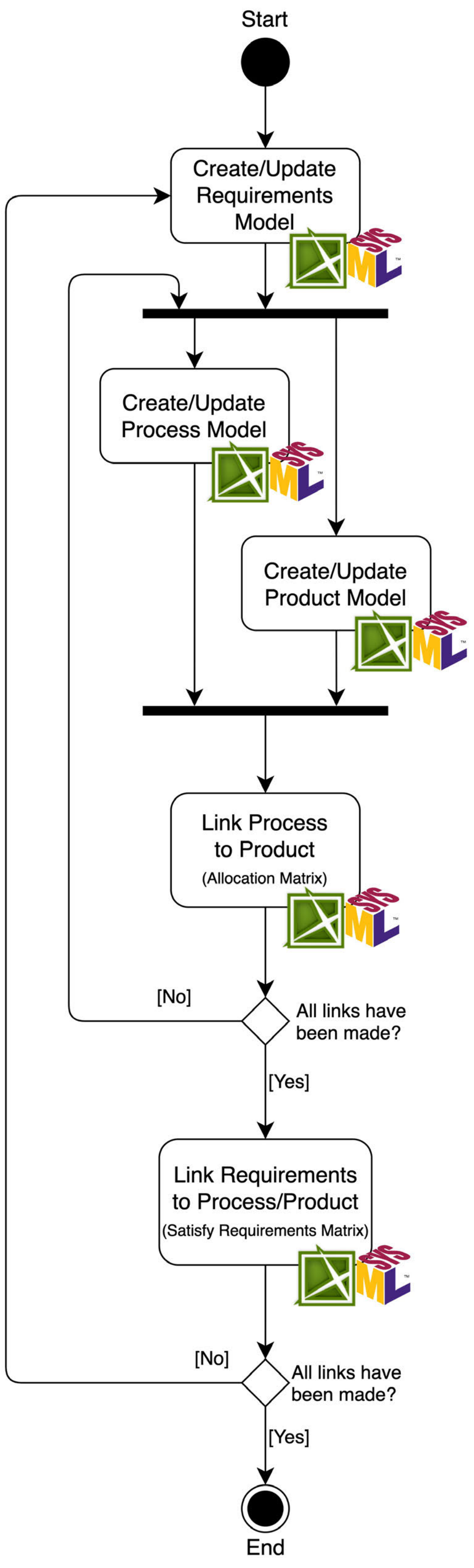

Start
Create/Update Requirements Model
Create/Update Process Model
Create/Update Product Model
Link Process to Product
(Allocation Matrix)
[No]
All links have been made?
[Yes]
Link Requirements to Process/Product
(Satisfy Requirements Matrix)
[No]
All links have been made?
[Yes]
End

Figure 5.21: Activity diagram showing the essential steps required to successfully create a Newly Proposed KBE Application Development Process. The gray notes indicate the stakeholders responsible for performing each task. The icons indicate the tool used to perform each task.

6
Verification & Validation of the Developed MBSE-for-KBE Framework

This chapter discusses the testing campaign used to verify and validate the developed MBSE-for-KBE framework.

6.1. Description of the Testing Campaign

The testing campaign used to verify and validate the proposed MBSE-for-KBE framework involved the development of two distinct KBE Applications using the proposed framework. The development of these KBE applications aimed to assess the framework in terms of ease of modeling, quality of the automatically generated (skeleton) code, support for requirements traceability, and application development time.

The first KBE application developed using the proposed framework focused on the development a KBE app for automating the design of Simple Airplane configurations. The basis for the development of this app were the examples of the pre-existing “Primi Plane” KBE application used in the tutorial lessons of the KBE course at TU Delft. The main goals of developing this app were manifold, and are described as follows:

1. Firstly, the aim was to verify and validate the knowledge capture and formalization capabilities of the developed framework through comprehensive testing of the (knowledge) ontology, modeling language, modeling tool, and modeling method. This involved assessing the effectiveness of these framework components in capturing and representing accurately the required domain knowledge for developing such an application.
2. Secondly, the aim was to verify and validate the automatic code generation capabilities of the framework by testing the translation engines developed in this research. This involved evaluating the accuracy and efficiency of the code generation process, in order to assess the framework’s ability to transform the modeled knowledge into functional code.
3. Thirdly, the aim was to verify and validate the requirements traceability capabilities provided by

the framework. This involved assessing the correct functionality of the GUI developed in this research to provide traceability of requirements from the models to the application code.

4. Lastly, the aim was to serve as a proof of concept, demonstrating the feasibility and practicality of the proposed methodology for developing KBE applications.

The second KBE application developed using the proposed framework focused on the development of a KBE app for automating the design and selection of off-the-shelf components of Modular Unmanned Aerial Vehicles (UAVs). The basis for the development of this application was a pre-existing KBE app previously developed by the author for the KBE course at TU Delft. The main goals of this case study were twofold, and are described as follows:

1. Firstly, the aim was to model (and develop) a more complex KBE app (compared to the Simple Airplane app) in order to test additional knowledge capture capabilities and functionalities of the proposed framework that were not assessed in the development of the Simple Airplane app. This included assessing the remaining elements of the Knowledge Ontology that had not previously been needed to model the Simple Airplane app. Furthermore, the capability of reusing knowledge from the knowledge repository developed in this research was also tested. This knowledge repository contained a package with preset product and process knowledge about commonly used methods and tools for preliminary analysis of propeller performance, as well as knowledge stored from previous projects, such as the Simple Airplane app.

2. Secondly, the aim was to evaluate the development time of KBE applications using the proposed framework. To achieve this, a comparative time study was conducted, focusing on the development time of the Modular UAV app. The study involved comparing the time required to develop the application using the proposed framework with the time taken for the traditional manual approach previously employed during its development for the KBE course.

The subsequent sections discuss the results obtained from the test campaign conducted for each developed KBE application.

6.2. Simple Airplane KBE Application

The capabilities of the MBSE-for-KBE framework tested during the development of the Simple Airplane KBE application are detailed in the following sections.

6.2.1. Knowledge Capture and Formalization

The development process of the Simple Airplane KBE application followed the proposed KBE app development process outlined in Figure 5.20, albeit with a small difference. To create the system model for the Simple Airplane app the MagicGrid-for-KBE method was used. However, in addition to defining the skeleton structure of the KBE application in MSoSA, the detailed knowledge and rules necessary to develop a complete KBE application were directly embedded in the system model using MSoSA's functionality to write Python code within the specification of each *Property*. This alternative approach, enabled by the relatively small size of the KBE application in question, served three main purposes. Firstly, it aimed to test the functionality provided by MSoSA, i.e., the capability to write code within the specification of each property. Secondly, it aimed to evaluate the quality of the automatically generated code in cases where the detailed knowledge and rules were written within MSoSA. Lastly, it aimed to assess the feasibility of generating a fully functional KBE application directly from the model using translation engines, eliminating the need for the subsequent (manual) addition of detailed knowledge and rules via PyCharm, thereby effectively reducing the number of tools required to implement the proposed framework.

This section presents only the main diagrams of the formal knowledge model of the Simple Airplane app that are required to provide more specific details about the features implemented and the functionalities tested in the development of this app. It should be noted that these diagrams are not (necessarily) presented in the order in which the knowledge was modeled. The modeling order corresponds to the one defined in the MagicGrid-for-KBE method presented in Section 5.3.3.

It should be noted that the focus of this case study was not to generate highly realistic descriptions of the requirements, processes, and products involved in the development of a real airplane. Instead, contrived descriptions that allowed to thoroughly test the knowledge capturing capabilities of the framework were used. Consequently, the definitions of the requirements, processes, and products were deliberately simplified.

Requirements Sub-ontology

To assess the applicability of the Requirements sub-ontology to the development of KBE applications, the requirements diagrams depicted in Figures 6.1 and 6.2 were created. These diagrams define stakeholder needs, as well as functional, physical, and performance requirements, along with containment and derivation relationships between requirements.

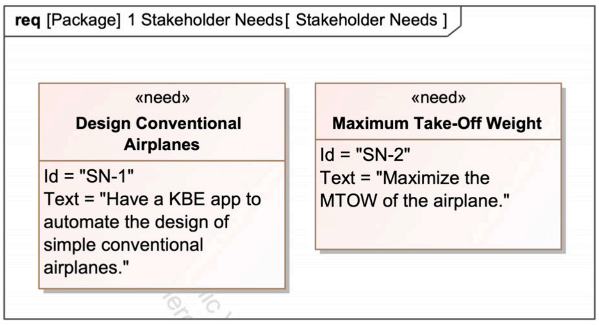

req [Package] 1 Stakeholder Needs[Stakeholder Needs]
«need»
Design Conventional Airplanes
Id = "SN-1"
Text = "Have a KBE app to automate the design of simple conventional airplanes."
«need»
Maximum Take-Off Weight
Id = "SN-2"
Text = "Maximize the MTOW of the airplane."

Figure 6.1: SysML Requirements Diagram representing the Stakeholder Needs of the Simple Airplane app. Note the use of the *«need»* stereotype defined in SysML-for-KBE.

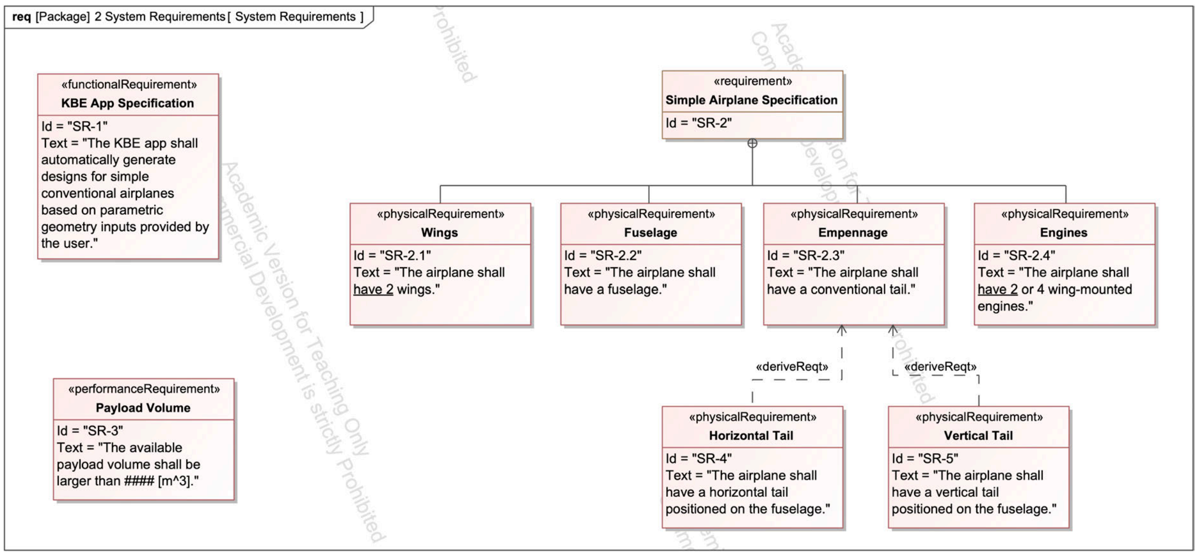

req [Package] 2 System Requirements [System Requirements]
«functionalRequirement» KBE App Specification
Id = "SR-1"
Text = "The KBE app shall automatically generate designs for simple conventional airplanes based on parametric geometry inputs provided by the user."
«requirement» Simple Airplane Specification
Id = "SR-2"
«physicalRequirement» Wings
Id = "SR-2.1"
Text = "The airplane shall have 2 wings."
«physicalRequirement» Fuselage
Id = "SR-2.2"
Text = "The airplane shall have a fuselage."
«physicalRequirement» Empennage
Id = "SR-2.3"
Text = "The airplane shall have a conventional tail."
«physicalRequirement» Engines
Id = "SR-2.4"
Text = "The airplane shall have 2 or 4 wing-mounted engines."
«deriveReqt»
«deriveReqt»
«performanceRequirement» Payload Volume
Id = "SR-3"
Text = "The available payload volume shall be larger than #### [m^3]."
«physicalRequirement» Horizontal Tail
Id = "SR-4"
Text = "The airplane shall have a horizontal tail positioned on the fuselage."
«physicalRequirement» Vertical Tail
Id = "SR-5"
Text = "The airplane shall have a vertical tail positioned on the fuselage."

Figure 6.2: SysML Requirements Diagram representing the System Requirements of the Simple Airplane app. Note the use of the *«requirement»*, *«physicalRequirement»*, *«performanceRequirement»*, and *«functionalRequirement»* stereotypes provided by MSoSA by default. Note also the use of containment and derivation relationships between requirements.

Process Sub-ontology

To assess the applicability of the Process sub-ontology to the development of KBE applications, the activity diagrams depicted in Figure 6.3 were created. These diagrams define activities and actions, as well as flow relationships between actions.

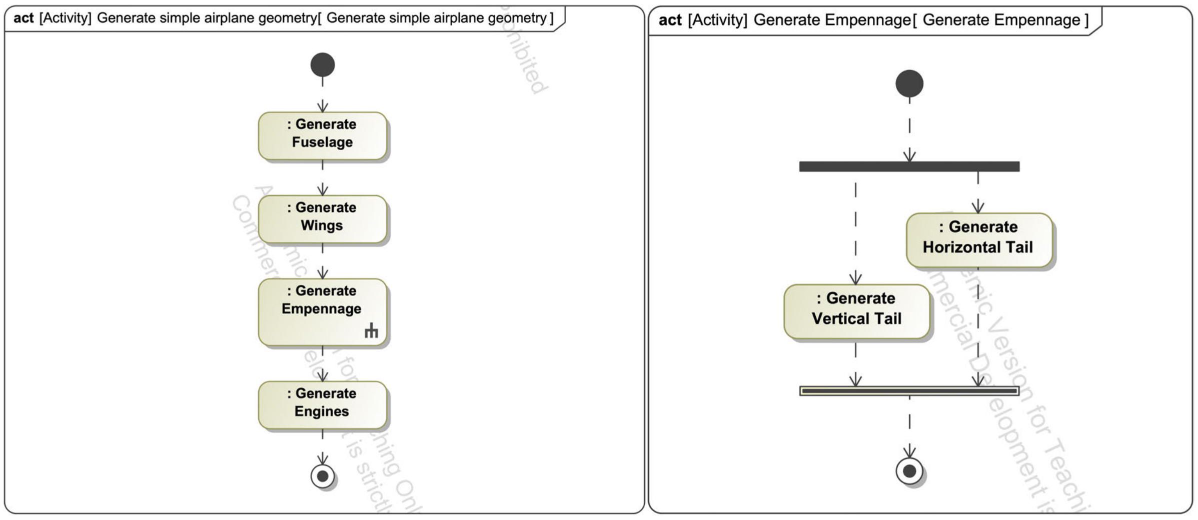

act [Activity] Generate simple airplane geometry[Generate simple airplane geometry]
: Generate Fuselage
: Generate Wings
: Generate Empennage
: Generate Engines
act [Activity] Generate Empennage[Generate Empennage]
: Generate Horizontal Tail
: Generate Vertical Tail

Figure 6.3: SysML Activity Diagram representing the Top-level Process of the Simple Airplane app. In the diagram on the left, note the rake symbol on the bottom right corner of the "*: Generate Empennage*" action, denoting that the "*Generate Empennage*" activity (shown on the diagram on the right) is further decomposed in the model an has a direct link/navigation to it.

Product Sub-ontology

To assess the applicability of the Product sub-ontology to the development of KBE applications, the block definition diagram (BDD) depicted in Figure 6.4 and the internal block diagrams (IBD) depicted in Figure 6.5 and Figure 6.6, were created. The BDD defines blocks, including their composition and inheritance relationships. On the other hand, the IBDs define inputs, attributes, parts, and methods, along with the connections between these elements. Note that in Figure 6.6 only a few elements of the diagram are presented, highlighting the definition of methods, as the entire diagram was too extensive and irrelevant for this section.

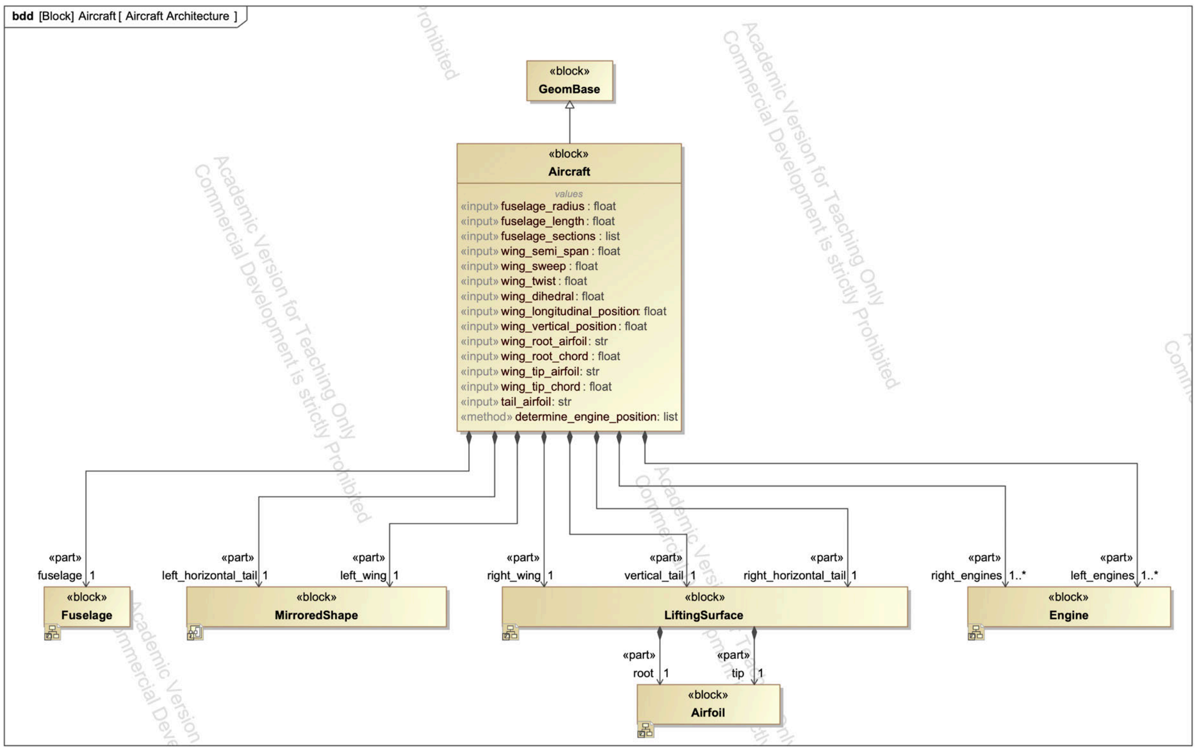

bdd [Block] Aircraft [Aircraft Architecture]
«block» GeomBase
«block» Aircraft
values
«input» fuselage_radius : float
«input» fuselage_length : float
«input» fuselage_sections : list
«input» wing_semi_span : float
«input» wing_sweep : float
«input» wing_twist : float
«input» wing_dihedral : float
«input» wing_longitudinal_position: float
«input» wing_vertical_position : float
«input» wing_root_airfoil : str
«input» wing_root_chord : float
«input» wing_tip_airfoil : str
«input» wing_tip_chord : float
«input» tail_airfoil : str
«method» determine_engine_position : list
«part» fuselage 1
«part» left_horizontal_tail 1
«part» left_wing 1
«part» right_wing 1
«part» vertical_tail 1
«part» right_horizontal_tail 1
«part» right_engines 1..*
«part» left_engines 1..*
«block» Fuselage
«block» MirroredShape
«block» LiftingSurface
«block» Engine
«part» root 1
«part» tip 1
«block» Airfoil

Figure 6.4: SysML Block Definition Diagram representing the architecture of the Aircraft class. Note the use of the *«block»* stereotype provided by SysML by default and the use of the *«input»*, *«method»*, and *«part»* stereotypes defined in SysML-for-KBE. The small squares on the bottom left corner of the blocks indicate that these blocks have an Internal Block Definition diagram associated to them and provide a direct link/navigation to that diagram.

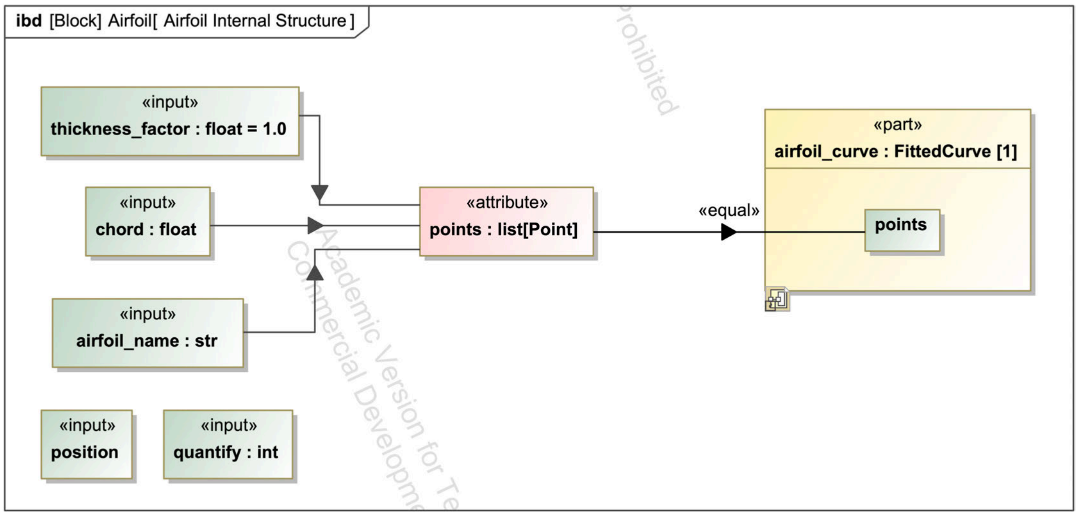

ibd [Block] Airfoil[Airfoil Internal Structure]
«input» thickness_factor : float = 1.0
«input» chord : float
«input» airfoil_name : str
«input» position
«input» quantify : int
«attribute» points : list[Point]
«equal»
«part» airfoil_curve : FittedCurve [1]
points

Figure 6.5: SysML Internal Block Diagram representing the internal structure of the Airfoil class. Note the use of the *«input»*, *«attribute»*, and *«part»* stereotypes defined in SysML-for-KBE. The assignment of these stereotypes to the model elements automatically applies the defined symbol styles in the diagram.

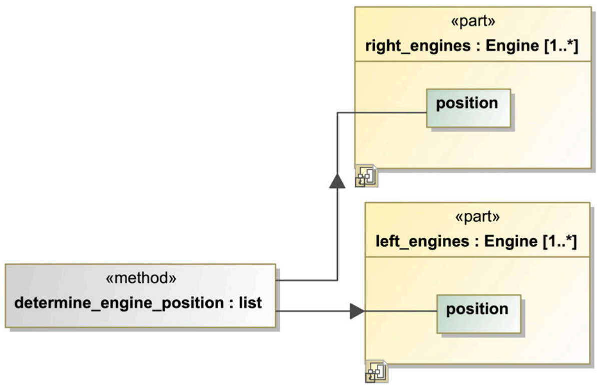
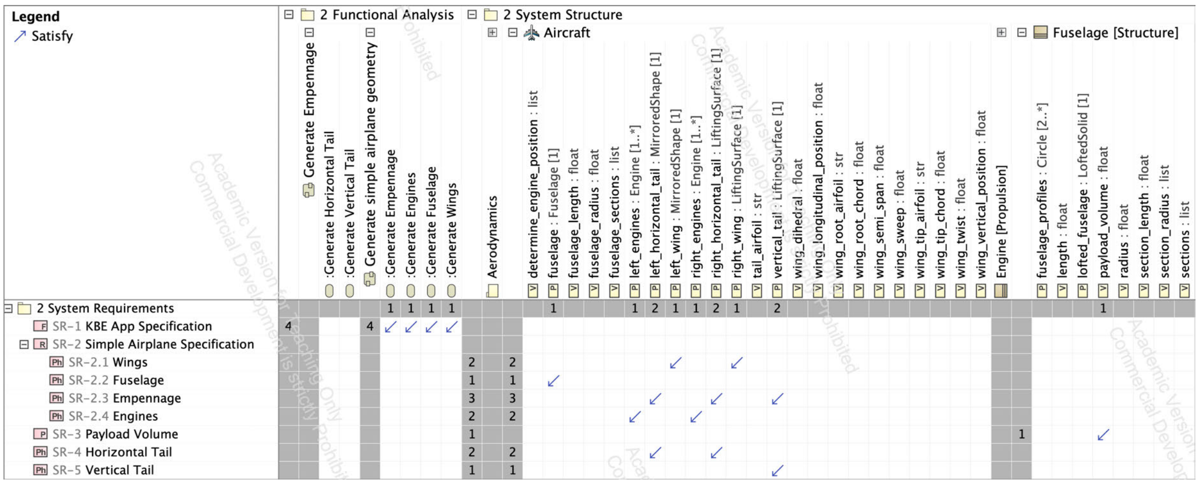

«part»
right_engines : Engine [1..*]
position
«part»
left_engines : Engine [1..*]
position
«method»
determine_engine_position : list

Figure 6.6: Snippet of the Internal Block Diagram of the Aircraft class representing some of its elements. Note the use of the *«method»* stereotype.

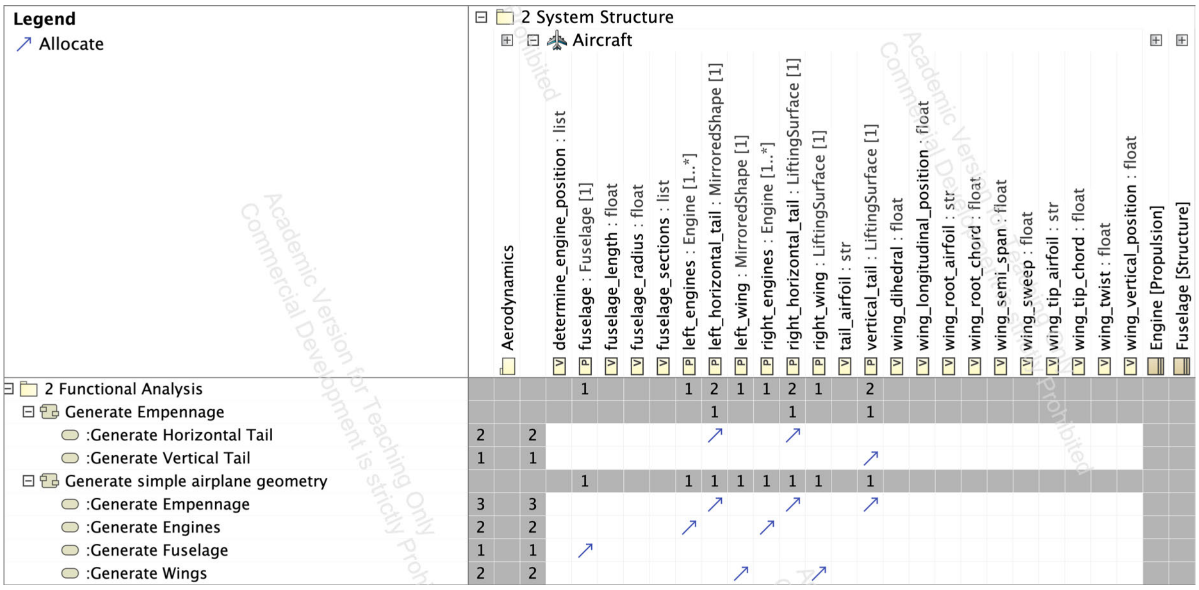
Interrelationships between Sub-ontologies

To assess the applicability of the defined interrelationships between the sub-ontologies of the Knowledge Ontology, the satisfy requirements matrix and the allocation matrix depicted in Figure 6.7 and Figure 6.8, respectively, were created. The satisfy requirements matrix defines satisfy relationships between functional requirements and activities, performance requirements and attributes, and physical requirements and parts, respectively. On the other hand, the allocation matrix defines allocation relationships between actions and properties.


Legend
↗ Satisfy

	2 Functional Analysis	Generate Empennage	:Generate Horizontal Tail	:Generate Vertical Tail	Generate simple airplane geometry	:Generate Empennage	:Generate Engines	:Generate Fuselage	:Generate Wings	2 System Structure	Aerodynamics	Aircraft	determine_engine_position : list	fuselage : Fuselage [1]	fuselage_length : float	fuselage_radius : float	fuselage_sections : list	left_engines : Engine [1..*]	left_horizontal_tail : MirroredShape [1]	left_wing : MirroredShape [1]	right_engines : Engine [1..*]	right_horizontal_tail : LiftingSurface [1]	right_wing : LiftingSurface [1]	tail_airfoil : str	vertical_tail : LiftingSurface [1]	wing_dihedral : float	wing_longitudinal_position : float	wing_root_airfoil : str	wing_root_chord : float	wing_semi_span : float	wing_sweep : float	wing_tip_airfoil : str	wing_tip_chord : float	wing_twist : float	wing_vertical_position : float	Engine [Propulsion]	Fuselage [Structure]	fuselage_profiles : Circle [2..*]	length : float	lofted_fuselage : LoftedSolid [1]	payload_volume : float	radius : float	section_length : float	section_radius : list	sections : list
2 System Requirements						1	1	1	1					1				1	2	1	1	2	1		2																1				
SR-1 KBE App Specification	4				4	↙	↙	↙	↙																																				
SR-2 Simple Airplane Specification																																													
SR-2.1 Wings										2		2								↙			↙																						
SR-2.2 Fuselage										1		1		↙																															
SR-2.3 Empennage										3		3							↙			↙			↙																				
SR-2.4 Engines										2		2						↙			↙																								
SR-3 Payload Volume										1																											1				↙				
SR-4 Horizontal Tail										2		2							↙			↙																							
SR-5 Vertical Tail										1		1													↙																				

Figure 6.7: SysML Satisfy Requirements Matrix of the Simple Airplane app. Note the functional requirement *SR-1* which is satisfied by various actions, the physical requirement *SR-2.2* which is satisfied by the *fuselage* part, and the performance requirement *SR-3* which is satisfied by the *payload_volume* attribute.



Legend
↗ Allocate

2 System Structure / Aircraft	2 System Structure	Aerodynamics	Aircraft	determine_engine_position : list	fuselage : Fuselage [1]	fuselage_length : float	fuselage_radius : float	fuselage_sections : list	left_engines : Engine [1..*]	left_horizontal_tail : MirroredShape [1]	left_wing : MirroredShape [1]	right_engines : Engine [1..*]	right_horizontal_tail : LiftingSurface [1]	right_wing : LiftingSurface [1]	tail_airfoil : str	vertical_tail : LiftingSurface [1]	wing_dihedral : float	wing_longitudinal_position : float	wing_root_airfoil : str	wing_root_chord : float	wing_semi_span : float	wing_sweep : float	wing_tip_airfoil : str	wing_tip_chord : float	wing_twist : float	wing_vertical_position : float	Engine [Propulsion]	Fuselage [Structure]
2 Functional Analysis					1				1	2	1	1	2	1		2												
Generate Empennage										1			1			1												
:Generate Horizontal Tail	2		2							↗			↗															
:Generate Vertical Tail	1		1													↗												
Generate simple airplane geometry					1				1	1	1	1	1	1		1												
:Generate Empennage	3		3							↗			↗			↗												
:Generate Engines	2		2						↗			↗																
:Generate Fuselage	1		1		↗																							
:Generate Wings	2		2								↗			↗														

Figure 6.8: SysML Allocation Matrix of the Simple Airplane app. The elements of the Process package are displayed in rows and the elements of the Product package are displayed in the columns.

6.2.2. Automatic Generation of KBE Application Code

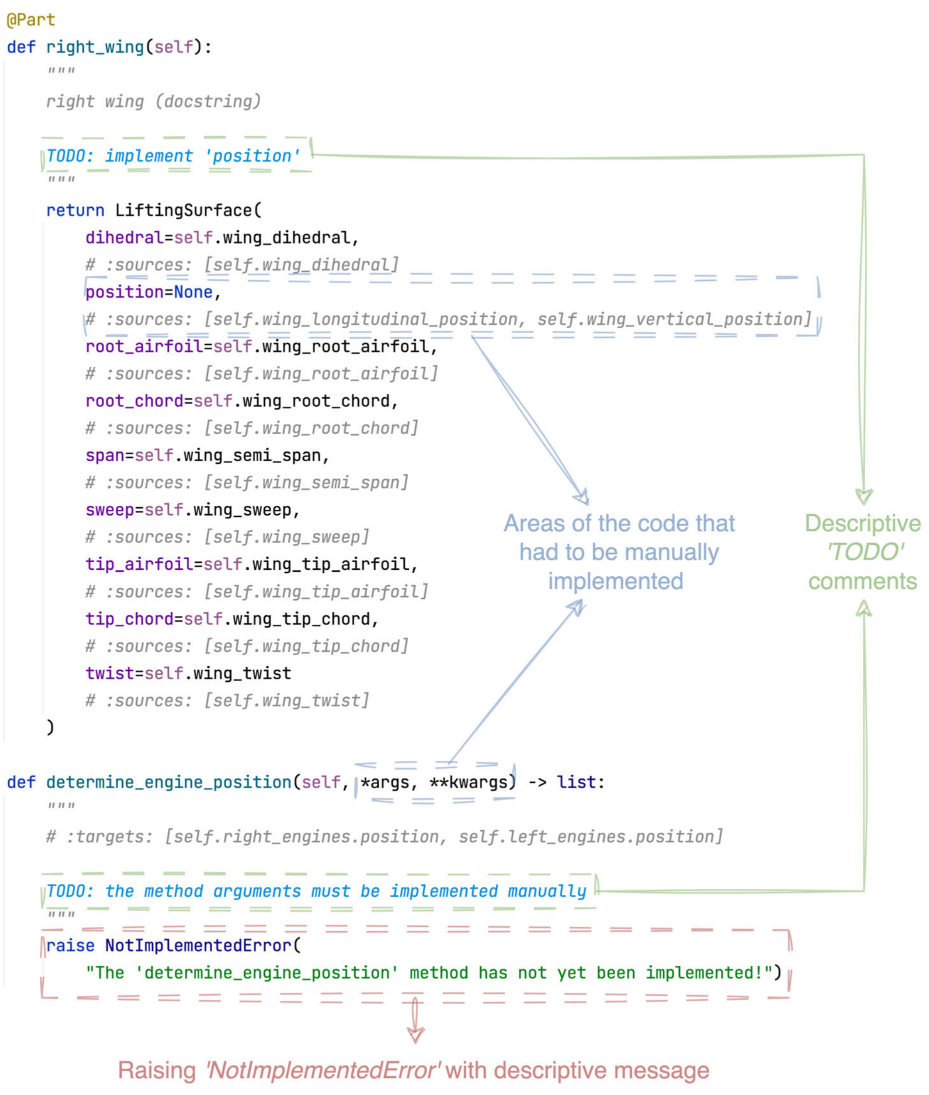
Following the generation of the model in MSoSA, an XML file containing all its information was exported. The XML file contained approximately 57,000 lines of data. Using the developed translation engines, this XML file was successfully translated into an intermediate file in JSON format, and the Simple Airplane KBE application code was created from the JSON. The whole translation process took less than one second. The resulting JSON file contained approximately 1,800 lines of data, representing a reduction in size of nearly 97% compared to the original XML file. Despite this reduction in size, the JSON file retained all the essential knowledge required to automatically generate the code of the KBE application and provide traceability within it. The internal structure of the resulting KBE application consisted of the directory tree shown in Figure 6.9, effectively generating all the packages defined in the model as folders and all the classes as Python files. It is important to note that the ‘MirroredShape’ class shown in Figure 6.4 is a *ParaPy Geometry Primitive*. As such, there is no need to generate a dedicated Python file for its definition, as it is already predefined within ParaPy. Therefore, it only needs to be imported into the classes that require its functionality.

```
Simple_Airplane
├── __init__.py
├── aircraft.py
├── aerodynamics_
│   ├── __init__.py
│   ├── airfoil.py
│   └── lifting_surface.py
├── propulsion_
│   ├── __init__.py
│   └── engine.py
└── structure_
    ├── __init__.py
    └── fuselage.py
```

Figure 6.9: Directory tree of the Simple Airplane KBE application showcasing its internal folders and files automatically generated by the translation engine developed in this research

The automatically generated code demonstrated good quality and a high level of completeness, requiring only minor manual interventions, through the addition of code in areas that were clearly identified with descriptive *'TODO'* comments, in order to enable the successful execution of a fully functioning KBE application. These minor interventions consisted of adding code to parts of the KBE app that are significantly more programming-intesive and challenging to implement automatically than other parts of the code, such as, positioning geometry components and defining normal Python methods. Examples of these areas where code had to be manually implemented are presented in Figure 6.10.

The code adequately fulfilled the intended requirements and specifications of the KBE app, with all expected features implemented and operating correctly. Moreover, the code was well organized and structured, easily readable, well documented, and adhered to PEP8 coding standards.



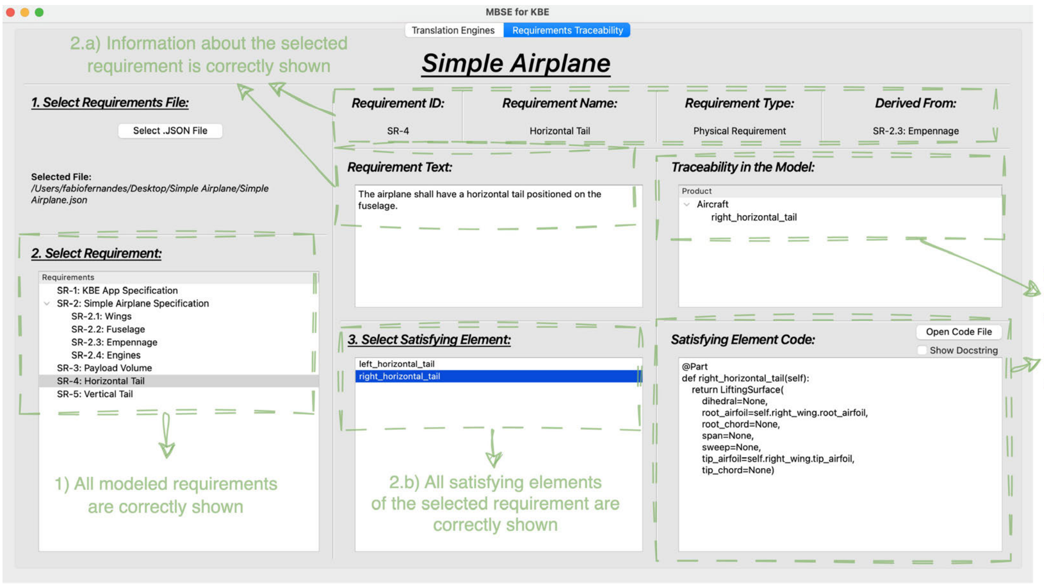
```
@Part
def right_wing(self):
    """
    right wing (docstring)

    TODO: implement 'position'
    """
    return LiftingSurface(
        dihedral=self.wing_dihedral,
        # :sources: [self.wing_dihedral]
        position=None,
        # :sources: [self.wing_longitudinal_position, self.wing_vertical_position]
        root_airfoil=self.wing_root_airfoil,
        # :sources: [self.wing_root_airfoil]
        root_chord=self.wing_root_chord,
        # :sources: [self.wing_root_chord]
        span=self.wing_semi_span,
        # :sources: [self.wing_semi_span]
        sweep=self.wing_sweep,
        # :sources: [self.wing_sweep]
        tip_airfoil=self.wing_tip_airfoil,
        # :sources: [self.wing_tip_airfoil]
        tip_chord=self.wing_tip_chord,
        # :sources: [self.wing_tip_chord]
        twist=self.wing_twist
        # :sources: [self.wing_twist]
    )

def determine_engine_position(self, *args, **kwargs) -> list:
    """
    # :targets: [self.right_engines.position, self.left_engines.position]

    TODO: the method arguments must be implemented manually
    """
    raise NotImplementedError(
        "The 'determine_engine_position' method has not yet been implemented!")
```

Areas of the code that had to be manually implemented
Descriptive 'TODO' comments
Raising 'NotImplementedError' with descriptive message

Figure 6.10: Snippet of the areas of the code of the Simple Airplane app where minor interventions were needed through manual completion of the code

6.2.3. Requirements Traceability

The requirements traceability functionality of the developed GUI was also tested, and it was verified that all its capabilities, previously described in Section 5.4.2, worked as expected. This is illustrated in Figure 6.11.


MBSE for KBE
Translation Engines
Requirements Traceability
2.a) Information about the selected requirement is correctly shown
1) All modeled requirements are correctly shown
2.b) All satisfying elements of the selected requirement are correctly shown
3) Traceability of the selected satisfying element in the model and in the code is correctly shown
Select .JSON File
Open Code File
Show Docstring

Simple Airplane

1. Select Requirements File:

Selected File:
/Users/fabiofernandes/Desktop/Simple Airplane/Simple Airplane.json

2. Select Requirement:

Requirements
SR-1: KBE App Specification
SR-2: Simple Airplane Specification
SR-2.1: Wings
SR-2.2: Fuselage
SR-2.3: Empennage
SR-2.4: Engines
SR-3: Payload Volume
SR-4: Horizontal Tail
SR-5: Vertical Tail

Requirement ID:	Requirement Name:	Requirement Type:	Derived From:
SR-4	Horizontal Tail	Physical Requirement	SR-2.3: Empennage

Requirement Text:

The airplane shall have a horizontal tail positioned on the fuselage.

Traceability in the Model:

Product
Aircraft
right_horizontal_tail

3. Select Satisfying Element:

left_horizontal_tail
right_horizontal_tail

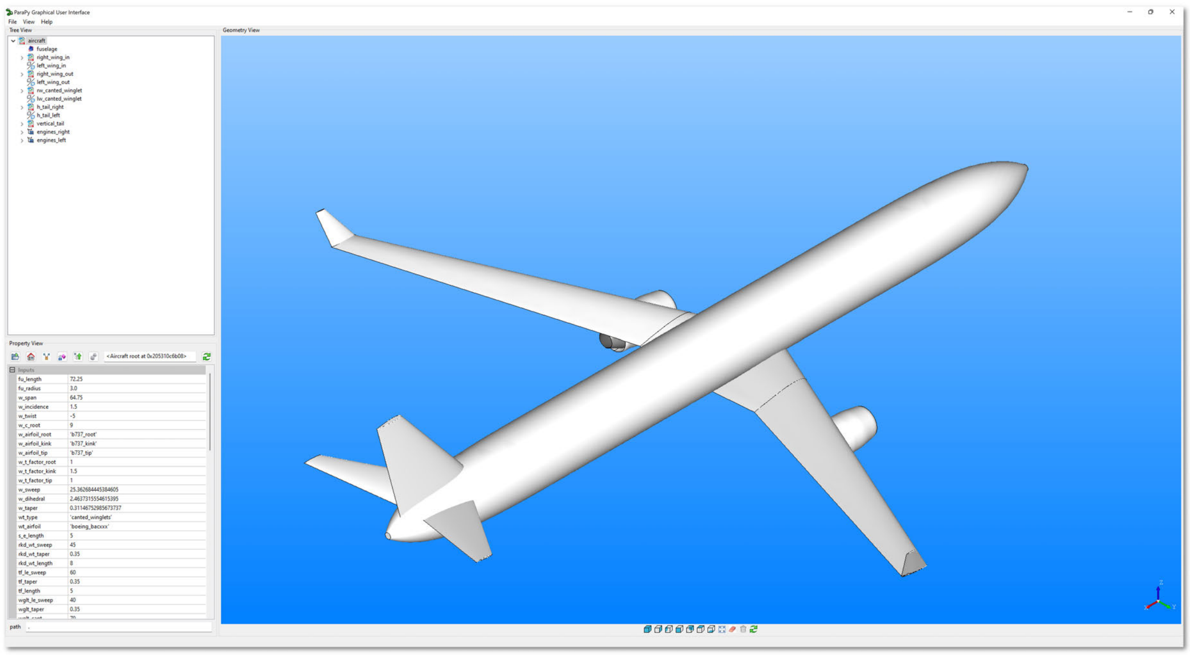
Satisfying Element Code:

```
@Part
def right_horizontal_tail(self):
    return LiftingSurface(
        dihedral=None,
        root_airfoil=self.right_wing.root_airfoil,
        root_chord=None,
        span=None,
        sweep=None,
        tip_airfoil=self.right_wing.tip_airfoil,
        tip_chord=None)
```

Figure 6.11: Requirements Traceability tab of the developed GUI highlighting the verification of its capabilities

6.2.4. Final Working Application

The final Simple Airplane KBE application developed using the proposed MBSE-for-KBE framework is shown in Figure 6.12.



Inputs	
fu_length	72.25
fu_radius	3.0
w_span	64.75
w_incidence	1.5
w_twist	-5
w_c_root	9
w_airfoil_root	'b737_root'
w_airfoil_kink	'b737_kink'
w_airfoil_tip	'b737_tip'
w_t_factor_root	1
w_t_factor_kink	1.5
w_t_factor_tip	1
w_sweep	25.362684445384605
w_dihedral	2.4637315554615395
w_taper	0.31146752985673737
wt_type	'canted_winglets'
wt_airfoil	'boeing_bacxxx'
s_e_length	5
rkd_wt_sweep	45
rkd_wt_taper	0.35
rkd_wt_length	8
tf_le_sweep	60
tf_taper	0.35
tf_length	5
wglt_le_sweep	40
wglt_taper	0.35
wglt_cant	70

Figure 6.12: Screenshot showing the ParaPy GUI for the Simple Airplane KBE application developed using the proposed MBSE-for-KBE framework

6.3. Modular UAV KBE Application

The capabilities of the MBSE-for-KBE framework tested during the development of the Modular UAV KBE application are detailed in the following sections.

6.3.1. Knowledge Capture and Formalization

The development process of the Modular UAV KBE application followed the proposed KBE app development process outlined in Figure 5.20. Firstly, the skeleton structure of the KBE application was defined in MSoSA using the MagicGrid-for-KBE method. Next, the skeleton code was automatically generated from the model using the translation engines developed in this research. Finally, the detailed knowledge and rules were manually added to the skeleton code using PyCharm.

This section presents only the main diagrams of the formal knowledge model of the Simple Airplane app that are required to provide more specific details about the features implemented and the functionalities tested in the development of this app.

Process Sub-ontology

To assess the remaining elements of the Process sub-ontology that were not assessed in the previous case study, the activity diagram depicted in Figure 6.13 was created. In addition to the elements already defined in the previous case study, this diagram also defines external services (and their allocation to external tools). Furthermore, the diagram demonstrates the functionality of hyperlinking files or webpages to different model elements, which are represented by the white file icon on the bottom left corner of each element. This feature enhances the documentation capabilities of the framework by storing all the links to the relevant documentation in a convenient and accessible location (i.e., the element model that implements the knowledge contained within the respective documentation) and allowing the user to open the associated files directly through the model element.

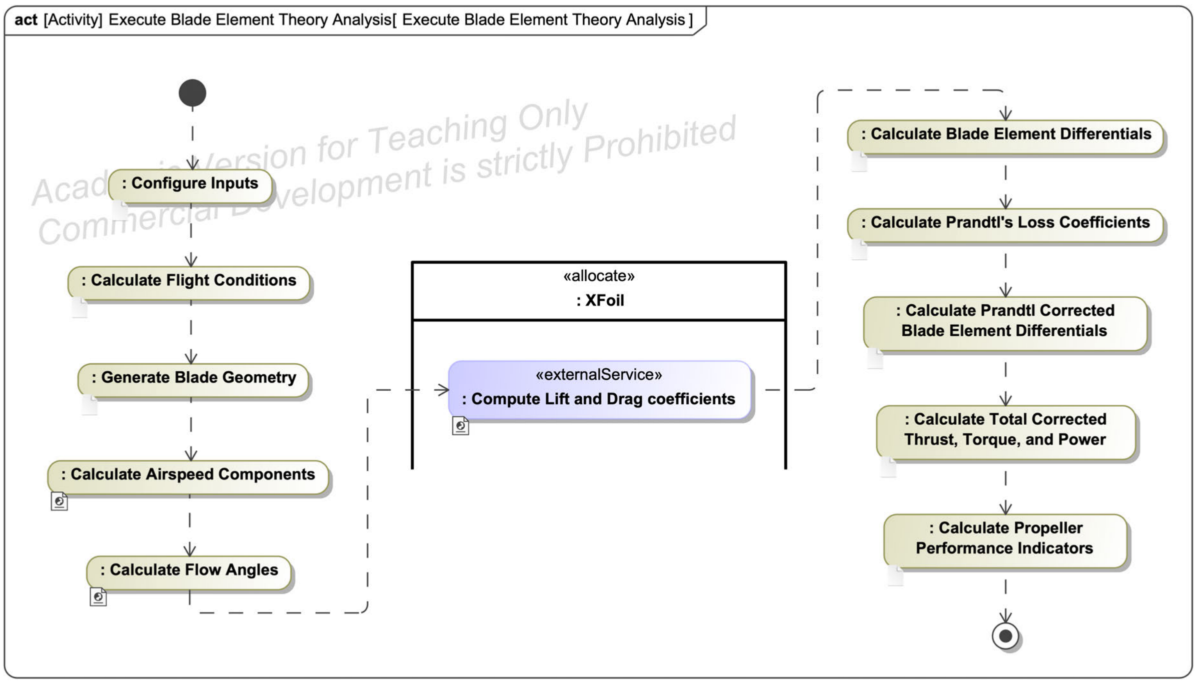

act [Activity] Execute Blade Element Theory Analysis[Execute Blade Element Theory Analysis]
: Configure Inputs
: Calculate Flight Conditions
: Generate Blade Geometry
: Calculate Airspeed Components
: Calculate Flow Angles
«allocate»
: XFoil
«externalService»
: Compute Lift and Drag coefficients
: Calculate Blade Element Differentials
: Calculate Prandtl's Loss Coefficients
: Calculate Prandtl Corrected Blade Element Differentials
: Calculate Total Corrected Thrust, Torque, and Power
: Calculate Propeller Performance Indicators
Academic Version for Teaching Only
Commercial Development is strictly Prohibited

Figure 6.13: SysML Activity Diagram representing the "*Execute Blade Element Theory Analysis*" activity from the Modular UAV app. Note the use of the stereotype *«externalService»* defined in SysML-for-KBE. The white icons on the bottom left corner of the activities denote that they have an hyperlinked (external) file and provide a direct link/navigation to that file.

Product Sub-ontology

To assess the remaining elements of the Product sub-ontology that were not assessed in the previous case study, the block definition diagram depicted Figure 6.14 and the internal block diagrams depicted in Figure 6.15 and Figure 6.16, were created. In addition to the elements already defined in the previous case study, these diagrams also define external tools, external functions, and actions. Note that in Figure 6.16 only a few elements of the diagram are presented, highlighting the definition of actions, as the entire diagram was too extensive and irrelevant for this section.

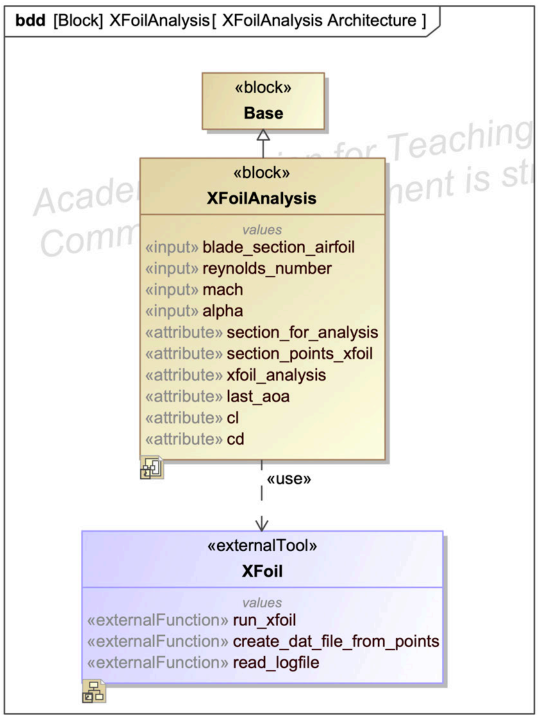

bdd [Block] XFoilAnalysis[XFoilAnalysis Architecture]
«block» Base
«block» XFoilAnalysis
values
«input» blade_section_airfoil
«input» reynolds_number
«input» mach
«input» alpha
«attribute» section_for_analysis
«attribute» section_points_xfoil
«attribute» xfoil_analysis
«attribute» last_aoa
«attribute» cl
«attribute» cd
«use»
«externalTool» XFoil
values
«externalFunction» run_xfoil
«externalFunction» create_dat_file_from_points
«externalFunction» read_logfile

Figure 6.14: SysML Block Definition Diagram representing the architecture of the XFoilAnalysis class. Note the use of the *«externalTool»* stereotype defined in SysML-for-KBE.

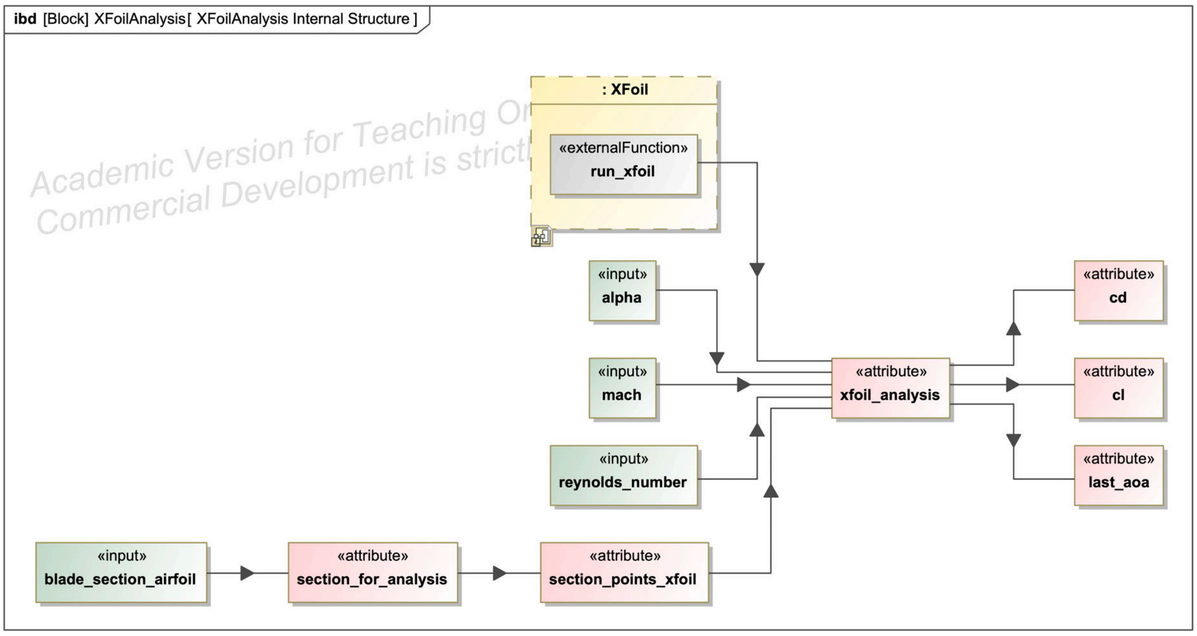

ibd [Block] XFoilAnalysis[XFoilAnalysis Internal Structure]
: XFoil
«externalFunction» run_xfoil
«input» alpha
«input» mach
«input» reynolds_number
«attribute» xfoil_analysis
«attribute» cd
«attribute» cl
«attribute» last_aoa
«input» blade_section_airfoil
«attribute» section_for_analysis
«attribute» section_points_xfoil

Figure 6.15: SysML Internal Block Diagram of the XFoilAnalysis class representing some of its elements. Note the use of the *«externalFunction»* stereotype defined in SysML-for-KBE.

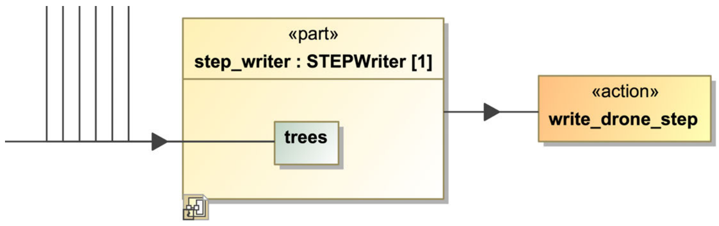


Figure 6.16: Snippet of the Internal Block Diagram of the Drone class representing some of its elements. Note the use of the *«action»* stereotype defined in SysML-for-KBE.

Knowledge Reuse

Finally, the framework's knowledge reuse capabilities were evaluated through two different approaches. Firstly, the *'Execute Blade Element Theory Analysis'* activity and the *'XFoil'* external tool were directly imported from a presets package stored in the knowledge repository. This package is automatically imported at the beginning of each new project using the MagicGrid-for-KBE method template defined in MSoSA. Secondly, project-to-project knowledge transfer was explored by manually creating a copy (via copy/paste) of the *'Airfoil'* class from the Simple Airplane app. The distinction between these approaches lies in the preservation of connections. In the former approach, the imported elements maintain their connection to the original elements in the presets package. In the latter approach, the copied element becomes independent of its source.

In the case of the *'Airfoil'* class, similarly to its usage in the Simple Airplane app where it was employed in the *'LiftingSurface'* class to generate the root and tip airfoils of wings, in the Modular UAV app it was employed in the *'Blade'* class to generate the root and tip airfoils of propeller blades.

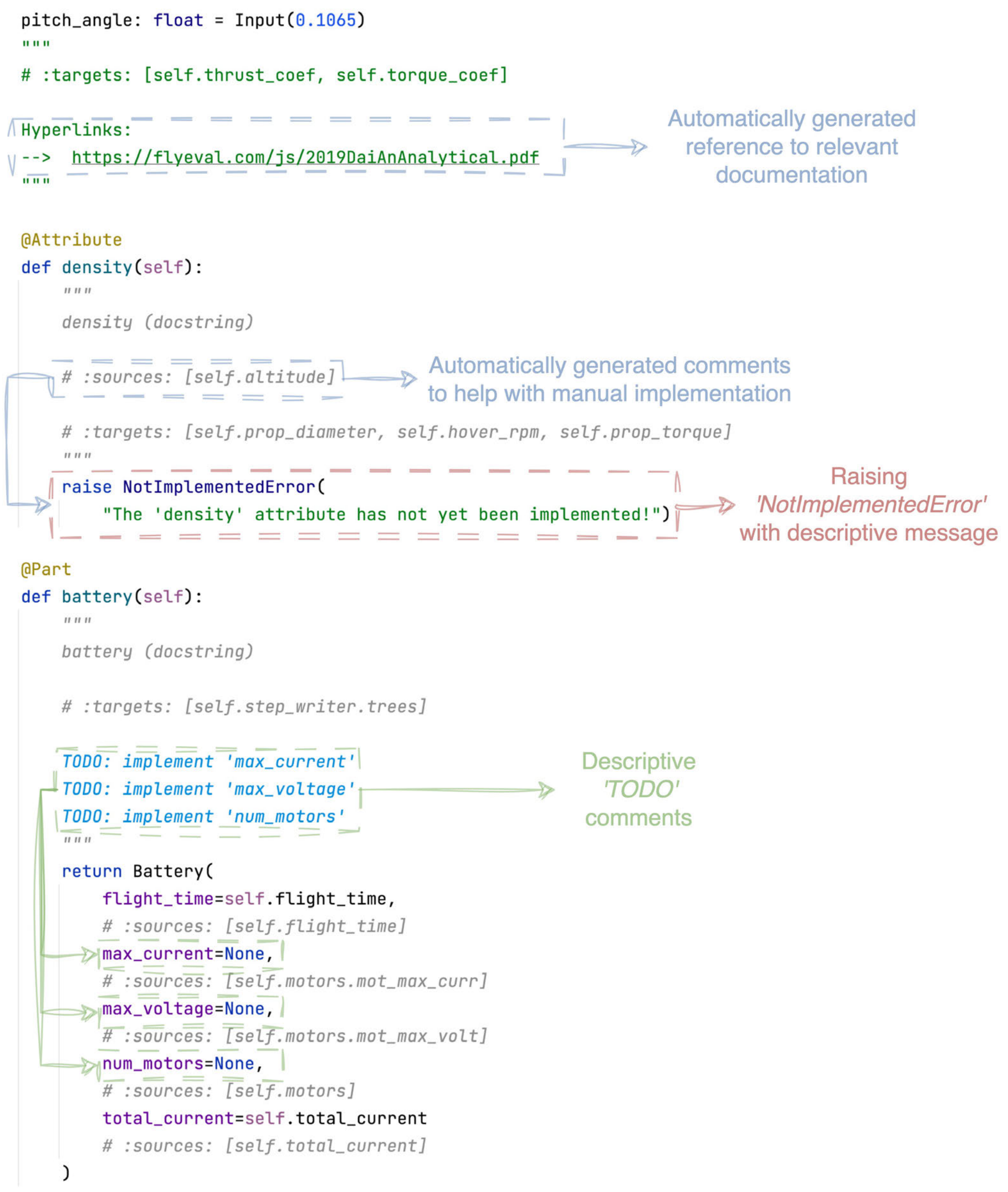
In both cases, the imported elements did not require any modifications. However, if modifications had been necessary, the impact would have varied based on the approach. For the *'Execute Blade Element Theory Analysis'* activity and the *'XFoil'* external tool, any changes made within the MSoSA project of the Modular UAV would have been reflected back in the presets package from which these elements were imported (and vice-versa). This was possible due to the maintained connection between them. Conversely, modifications made to the 'Airfoil' class would not affect its original definition in the Simple Airplane MSoSA project (and vice-versa), as the copied element remained independent from its source.

This ability to reuse existing knowledge and the flexibility to modify it to suit the specific requirements of the KBE app, within SysML models, has the potential to significantly reduce the development time of KBE applications.

6.3.2. Automatic Generation of KBE Application Code

Following the generation of the model in MSoSA, an XML file containing all its information was exported. The XML file contained approximately 127,000 lines of data. Using the developed translation engines, this XML file was successfully translated into an intermediate file in JSON format, and the Modular UAV KBE application (skeleton) code was created from the JSON. The whole translation process took around three seconds. The resulting JSON file contained approximately 5,500 lines of data, representing a reduction in size of nearly 96% compared to the original XML file. Once again, despite this reduction in size, the JSON file retained all the essential knowledge required to automatically generate the skeleton code of the KBE application and provide traceability within it.

Similarly to the Simple Airplane case study, the automatically generated skeleton code was of good quality, correctly implementing all expected features. The main difference was that, in this case, the automatically generated code consisted only of the app's skeleton, and the detailed rules and expressions used to compute each property had to be manually implemented in PyCharm. This manual implementation was facilitated by the presence of easily identifiable and descriptive comments in the code that had been automatically generated during the translation process. Specifically, in the elements whose detailed expression still had to be implemented, a 'NotImplementedError' was raised, accompanied by a descriptive error message, and the source and target elements of each respective element were written in its docstring. In cases where this approach was not applicable, 'None' was returned by the elements, and descriptive 'TODOs' were included in their respective docstrings, indicating the necessary tasks that needed to be completed. This is illustrated in Figure 6.17.



```
pitch_angle: float = Input(0.1065)
"""
# :targets: [self.thrust_coef, self.torque_coef]

Hyperlinks:
-->  https://flyeval.com/js/2019DaiAnAnalytical.pdf
"""

@Attribute
def density(self):
    """
    density (docstring)

    # :sources: [self.altitude]

    # :targets: [self.prop_diameter, self.hover_rpm, self.prop_torque]
    """
    raise NotImplementedError(
        "The 'density' attribute has not yet been implemented!")

@Part
def battery(self):
    """
    battery (docstring)

    # :targets: [self.step_writer.trees]

    TODO: implement 'max_current'
    TODO: implement 'max_voltage'
    TODO: implement 'num_motors'
    """
    return Battery(
        flight_time=self.flight_time,
        # :sources: [self.flight_time]
        max_current=None,
        # :sources: [self.motors.mot_max_curr]
        max_voltage=None,
        # :sources: [self.motors.mot_max_volt]
        num_motors=None,
        # :sources: [self.motors]
        total_current=self.total_current
        # :sources: [self.total_current]
    )
```

Automatically generated reference to relevant documentation
Automatically generated comments to help with manual implementation
Raising 'NotImplementedError' with descriptive message
Descriptive 'TODO' comments

Figure 6.17: Snippet of the code of the Modular UAV app showcasing the presence of easily identifiable and descriptive comments

6.3.3. Application Development Time

To assess the development time of the Modular UAV app using the MagicGrid-for-KBE methodology, a time study was conducted. The results of this time study are presented in Table 6.1. It is important to mention that the values presented reflect only the required modeling/coding effort, the time spent on knowledge acquisition is not accounted for.

Table 6.1: Time Study of the Modular UAV app

Task	Time [HH:mm]
1. Set up the project in MSoSA	00:01
2. Create Requirements Model	00:30
3. Create Process Model	01:00
4. Create Product Model	10:00
5. Link Process to Product	00:20
6. Link Process/Product to Requirements	00:10
7. Translate the SysML model to KBE (skeleton) code	00:01
8. Add details to the code	08:30
TOTAL	20:32

The first task, setting up the project in MSoSA, was straightforward and took less than one minute to complete since a pre-existing template, the MagicGrid-for-KBE method template, was utilized.

The second and third tasks involved creating the requirements and process models, respectively. These models were deliberately kept simple as the formalization of this knowledge had not been previously done during the manual development of the application for the KBE course. Therefore, limited comparisons could be drawn regarding the time spent on these tasks. The purpose of these tasks was to enhance the model's completeness, thereby improving the knowledge captured during the development of the KBE app.

The fourth task, creating the product model, proved to be the most time-consuming, requiring approximately 10 hours to complete. This was already expected, considering that developing the product model requires the biggest modeling effort as it encapsulates all the knowledge that is automatically translated into the KBE application code.

Tasks five and six involved linking processes to the products to which they are allocated, and linking processes and products to the requirements they satisfy, respectively. These tasks were straightforward due to the simplified top-level requirements and processes defined earlier in tasks two and three. Given that these steps were not performed during the manual development of the Modular UAV app for the KBE course, limited comparisons could be made regarding the time required to complete each task. The purpose of these tasks was to provide traceability of the captured requirements and processes to the elements that satisfy/implement them in the application code, thereby improving the overall transparency of the KBE app.

Task seven consisted of translating the SysML model into skeleton KBE code. This process involved exporting the SysML model to the XML format provided by MSoSA and utilizing the developed translation engines to automatically generate the application's skeleton code. The entire process took less than a minute.

Finally, task eight involved adding the details to the product model by manually writing the expressions for the code elements using PyCharm. This was a relatively time-consuming process due to the size of the considered KBE app, taking approximately 8.5 hours to complete. However, the excellent structure and organization of the automatically generated skeleton code, the presence of descriptive 'TODO' comments, and helpful documentation within the code significantly facilitated this process.

The development process for the Modular UAV app, utilizing the MBSE-for-KBE framework, involved a total effort of approximately 20 hours and 30 minutes, combining both modeling and coding activities. In contrast, the manual approach for creating the app during the KBE course required an estimated 3 hours for modeling (specifically, generating a UML class diagram) and 20 hours for coding, resulting in a combined effort of 23 hours.

These results demonstrate an 11% reduction in development time when employing the MBSE-for-KBE framework. Additionally, the framework offers substantial benefits that contribute to a more comprehensive and detailed knowledge capture, improved requirements traceability, and enhanced transparency within the application. These factors contribute to the reduction of the black-box perception associated with KBE apps, which, in turn, could lead to decreased costs in future tool development and maintenance. Furthermore, reducing the black-box perception of KBE applications can lead to higher trust in the application by the various stakeholders, promoting the adoption of KBE technology in the industry. Moreover, the utilization of the proposed framework promotes knowledge reuse in the development of subsequent KBE apps, potentially resulting in further reductions in development times.

To provide a more accurate evaluation of the time efficiency of the new approach, a comparative analysis was conducted by focusing only on tasks 4, 7, and 8, which align better with those performed during the manual development of the Modular UAV app for the KBE course. The total time required to complete these tasks using the proposed framework amounted to approximately 18.5 hours, while the manual approach took 23 hours, demonstrating a reduction in the time that it took to develop a KBE app with the same functionality as the previous one in approximately 20%.

Two possible limitations of this time study must be acknowledged. Firstly, is the fact that the time study focuses only on the modeling and coding aspects of creating the KBE application, while assuming that the time invested in knowledge acquisition would be consistent across both approaches. The presented case demonstrated that developing KBE applications via a combination of modeling and coding - instead of just coding - offers benefits such as a better overview of the app and easier consolidation of knowledge while generating the model, potentially resulting in faster knowledge acquisition. However, this could not be tested in this case study because it is important to consider that during the author's second attempt at creating the application using the proposed framework, he was already more familiar with its expected structure compared to their initial manual development of the app for the KBE course. This increased familiarity could have positively influenced the time taken to develop the application for the second time, and thus the results of such time study could have been biased. Secondly, it is worth mentioning that the author's lack of experience with MSoSA, in general, could have negatively impacted the time required to develop the application using the newly proposed framework. At the time of conducting this time study, the author was not aware of certain "modeling shortcuts" that could have expedited the modeling process. In comparison, the author had significantly more coding experience when he initially developed the KBE application for the KBE course, which could have made them more efficient in that aspect. However, it is challenging to quantify the influence or impact of these factors on the development of the app based solely on this single case.

6.3.4. Final Working Application

The final Modular UAV KBE application developed using the proposed MBSE-for-KBE framework is shown in Figure 6.18.

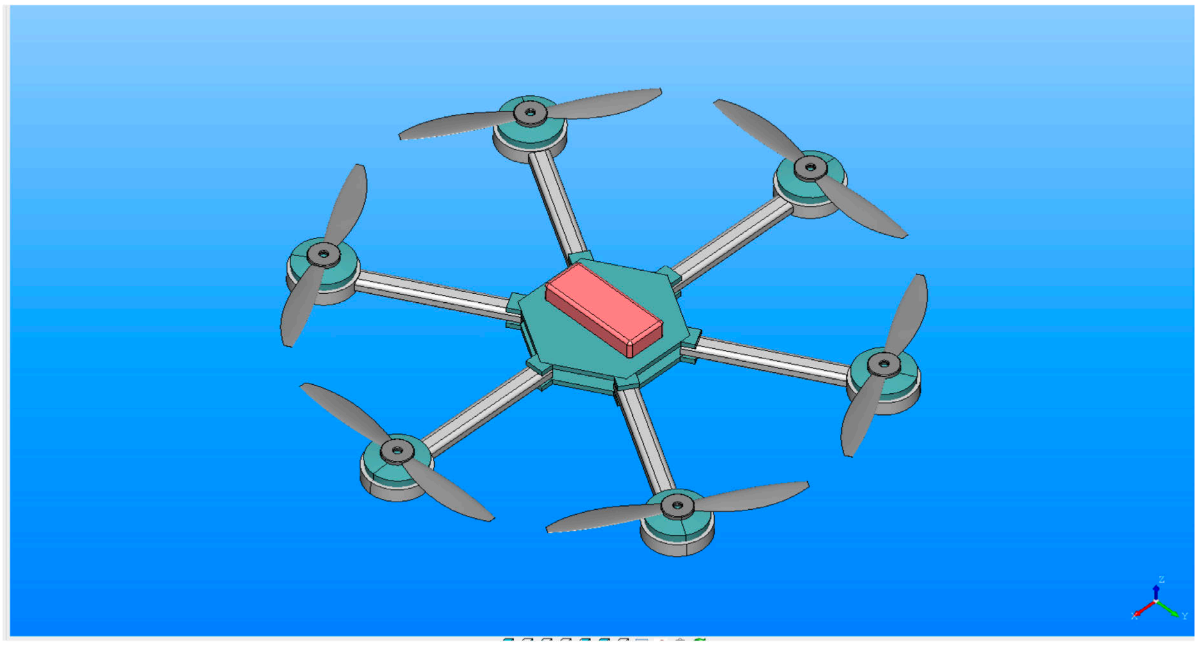


Figure 6.18: Screenshot showing the Geometry View of the ParaPy GUI for the Modular UAV KBE application developed using the proposed MBSE-for-KBE framework

7 Case Study

This chapter presents the case study used to assess the applicability of the developed MBSE-for-KBE framework in the development of real-world KBE applications.

7.1. GKN Aerospace Fokker Elmo - EWIS Architecture Modeler

This case study, conducted in collaboration with GKN Aerospace Fokker Elmo experts, focused on the development of a KBE application for automating the design of Electrical Wiring Interconnection Systems (EWIS) architectures. The main goals of this case study were threefold:

1. Firstly, to apply the developed framework to a real-world scenario during the early phases of the conceptual design stage, in order to assess the framework's performance in situations where knowledge acquisition was ongoing and where knowledge was highly prone to change.
2. Secondly, to compare the development time of the KBE app skeleton using the proposed framework against the time required by the industry experts using the traditional manual approach.
3. Thirdly, to obtain comprehensive feedback from industry experts regarding the performance of the developed framework and its level of satisfaction in meeting their KBE app development requirements.

In order to compare the development time of the EWIS Architecture Modeler app skeleton code using the proposed framework versus the traditional approach employed at Fokker Elmo, two time studies were conducted. The first study, conducted by the author, involved developing the app skeleton using the proposed MBSE-for-KBE framework. The second study, carried out by a KBE expert at Fokker Elmo, involved developing the app skeleton using the manual approach traditionally used at the company (previously described in Figure 2.3b). Table 7.1 presents the results of both time studies, with relevant information in the table footnotes.

Table 7.1: Comparative Time Study of the development of the EWIS Architecture Modeler app (Proposed Framework vs. Traditional Approach)

Task	Proposed Framework[a] Time [HH:mm]	Traditional Approach[a] Time [HH:mm]
1. Create Requirements Model[b]	00:30	00:30
2. Create Process Model[c]	03:00	04:00
3. Create Product Model[c]	03:00	04:00
4. Link Process to Product	00:10	01:30[d]
5. Link Process/Product to Requirements	00:10	00:45[d]
6. Translate the Product model to KBE (skeleton) code[e]	00:01	02:00
TOTAL	06:51	12:45

[a] The tasks performed using the proposed framework were performed by the author, which at the time possessed limited experience with the modeling tool (MSoSA). Conversely, the tasks performed using the traditional approach were performed by a KBE expert at Fokker Elmo. As a result, there is a slight mismatch in the learning curve position for each user in the respective cases, suggesting that there is potential for an even bigger time difference between the two approaches (i.e., the proposed framework might offer even greater time savings than those reported).

[b] The requirements model created using the traditional approach consisted only of a Microsoft Excel table format. Conversely, the requirements model created using the proposed framework encompassed both a table and a diagram. The diagram was generated (semi-)automatically from the table, offering an additional perspective of the model.

[c] The models generated in both approaches contained exactly the same information, with the only difference between them being that the models created using the proposed framework were SysML models and the models created using the traditional approach were UML models. The time difference between the two approaches lies in the fact that the modeling tool used in the proposed framework (MSoSA) offers certain functionalities that make the modeling process faster than in the modeling tool used in the traditional approach (Microsoft Visio), such as, automatic diagram layout based on the type of diagram and better interconnection between elements represented across multiple diagrams. Furthermore, Visio does not enforce UML/SysML well-formedness rules as well as MSoSA, which potentially might lead to errors in the model requiring further re-work.

[d] The times presented for the traditional approach are estimates, as these tasks are not actually performed in the traditional approach. However, performing these tasks is essential to guarantee traceability of requirements and processes to the KBE application code. In order to perform these tasks in the traditional approach, tables would have to be manually created in Excel. It is important to consider that these tables are static and disconnected from the models, unlike the matrices that are automatically created (and updated) in the proposed framework. Thus, in the traditional approach, any changes made in the models would have to be manually propagated to the Excel tables, a process which is highly time-consuming and error-prone.

[e] The SysML/UML Product model to be translated contained a total of 27 blocks. Out of these, 12 contained an average of 5 ParaPy slots (inputs, attributes, and parts), while the remaining 15 were left empty (i.e., contained no slots).

The results of the comparative time study show a substantial reduction in the time required for the initial knowledge model development and the code skeleton generation using the proposed framework. The study revealed a considerable 46.3% decrease in the time required to generate knowledge models and skeleton code through the implementation of the proposed framework. The translation of models into application skeleton code in the traditional approach took 2 hours, accounting for approximately 15.7% of the total development time. In contrast, the automatic translation of the SysML model into skeleton code took less than 1 minute, demonstrating the efficiency and time-saving advantages of the MBSE approach.

These improvements in development time highlight the potential of the proposed framework in streamlining the development process of KBE applications, reducing the manual effort required, and expediting the generation of application code.

According to feedback received from KBE experts at Fokker Elmo, the proposed framework offers several notable benefits. These are described as follows:

1. Firstly, the established connection among requirements, process, and product models, enhances the traceability of knowledge within the KBE app, reducing its black-box perception.
2. Secondly, the direct connection between the knowledge models and the application code, facilitated by the translation engines, mitigates the occurrence of inconsistencies between the models and the code. This ensures a higher level of coherence and accuracy throughout the development process, especially during the conceptual design phase where the modeled knowledge is highly prone to change.
3. Thirdly, the automatically generated code demonstrates a quality comparable to that manually produced by a KBE expert, alleviating concerns regarding the reliability and competence of the automatically generated code.

Furthermore, one of the KBE experts at Fokker Elmo conveyed their endorsement of the proposed framework, expressing a willingness to recommend it to all colleagues, except for the most experienced KBE developers. This exception stems from their already established proficiency in creating KBE apps using their existing methods, which might make them less inclined to embrace a new approach and deviate from their established practices. This is directly related to the adoption and acceptance challenge of implementing MBSE outlined in Section 3.3.

8 Conclusion

This chapter summarizes the key findings and outcomes of the thesis work. Section 8.1 describes the main conclusions drawn from the presented research work and answers the research questions presented in Chapter 4. In Section 8.2 recommendations for future work are proposed, highlighting potential avenues for further investigation and suggesting areas for improvement or development based on the identified limitations of the current work.

8.1. Conclusions

8.1.1. General Overview and Main Conclusions

Knowledge Based Engineering (KBE) applications offer a promising solution to address the increasing complexity of engineering systems, the need for rapid time-to-market, and the need for achieving reductions in the costs of product development. However, to date, existing KBE development methodologies have several limitations that impact the time needed for application development, the quality of the resulting applications, and the ability to effectively preserve and reuse engineering knowledge.

In this research work, we propose the use of a Model-Based Systems Engineering (MBSE) based framework to support the development of KBE applications, with the aim of improving knowledge capture and formalization, requirements traceability, knowledge reuse, and the development time of KBE applications.

The proposed framework involves generating a formal knowledge model using the industry-standard Systems Modeling Language (SysML), where the knowledge required for the application development is captured in a digital model using multiple interconnected and synchronized views. Source code is then automatically generated for the targeted KBE system by dedicated translators that parse the knowledge model. Traceability of requirements onto the various elements of the KBE app architecture is also provided, thereby reducing the typical black-box perception of KBE applications. Furthermore, the framework allows to reuse knowledge from previously generated knowledge models, enabling effective project-to-project knowledge transfer.

The KBE applications developed during the verification and validation phase, and the case study carried out, prove the applicability of the framework to the development of KBE applications. Preliminary results show that the learning curve to modeling is fairly intuitive and easy to learn; the time required for generating the knowledge models is lower than current modeling processes; the automatically generated code is error-free, well-structured, and complies with existing coding standards, providing a correct starting point for further app development, while resulting in time savings in the development of the app skeleton.

The main contributions of this research project can be summarized as follows:

- A consistent way of capturing and representing requirements, process, and product knowledge for KBE applications, supported by a graphical modeling language called SysML-for-KBE, based on the Systems Modeling Language (SysML).
- A software tool to automatically generate KBE code from SysML models and assist in providing traceability of requirements within KBE applications.

The developed framework directly targets the ParaPy KBE language, however the results from this research work can be extrapolated and adapted to suit a broader spectrum of KBE languages, simply by modifying the translation engines to account for the different ontology mapping required to map the knowledge base ontology to the specific KBE language ontology.

8.1.2. Answers to Research Questions

Based on the research work carried out, answers can now be given to the initial research questions. Starting by answering the main research question:

Q.1. ***"What is the impact of using MBSE to support the development of KBE applications in terms of their transparency, requirements traceability, knowledge reuse, and development time?"***

MBSE provides additional rigor in the specification and development process of KBE applications. The increased knowledge formalization provided by MBSE results in a more comprehensive and detailed knowledge capture, enables traceability of requirements, and improves the transparency of the applications. These factors contribute to the reduction of the black-box perception associated with KBE apps, which, in turn, promotes knowledge reuse in the development of subsequent applications, potentially resulting in reduced development times. Moreover, this should lead to decreased costs in future tool development and maintenance.

Furthermore, the visual modeling approach offered by SysML enables product architects with limited programming skills to generate application skeletons, which can then be further developed by KBE developers in the target KBE development environment. This accessibility to KBE technology for less IT-specialized engineers, coupled with the reduction of the black-box perception of KBE applications, enhances trust and promotes industry acceptance of KBE technology.

However, a limitation of the present research lies in the difficulty of quantitatively measuring the degree of improvement in application transparency, requirements traceability, knowledge reuse, and validity of the KBE applications compared to the traditional *ad hoc* approach. Obtaining more comprehensive insights would require studying additional cases over an extended period and developing appropriate metrics to accurately evaluate these factors. Addressing this limitation is crucial for a deeper understanding of the effectiveness and impact of the proposed approach.

Answering the sub-questions that were derived from the main research question:

Q.1.1. ***"To which phases of the KBE app development process can MBSE be applied, and which customizations are necessary to support it?"***

MBSE can be applied to the knowledge capture and formalization phases of the KBE app development process. In order to achieve this, certain customizations need to be made to the modeling language, modeling tool, and modeling method to ensure their suitability for supporting KBE application development and reducing the learning curve for domain experts and other users of the methodology.

In terms of the modeling language, it is essential to streamline the concepts (diagrams and elements) utilized and define new stereotypes that precisely capture the knowledge required for KBE application development. In this research work, a specialized modeling language called SysML-for-KBE was developed based on the widely adopted Systems Modeling Language (SysML).

In terms of the modeling tool, it is necessary to simplify its functionalities and optimize the modeling process for KBE application development. Typically, modeling tools offer a plethora of features and details that may not be directly relevant to develop KBE applications. By customizing the tool, unnecessary complexities can be eliminated, allowing users to focus on the essential aspects of KBE modeling without being overwhelmed by irrelevant functionalities.

In terms of the modeling method, it should be specifically adapted to guide engineers through the modeling process required for developing KBE applications. In this research work, a tailored method known as the MagicGrid-for-KBE method was developed based on the foundation of the existing MagicGrid method.

Q.1.2. ***"How to bridge the gap between the knowledge modeling phase and the code formulation phase in the KBE app development process?"***

Bridging the gap between the knowledge formalization phase and the code formulation phase in the KBE app development process can be achieved through several key steps.

Firstly, it is essential to define an ontology that represents the knowledge used in KBE applications. This ontology serves as a formal and structured representation of the domain-specific knowledge. In the context of this research work, the 'Knowledge Ontology' was specifically designed and developed.

Additionally, the ontology of the target KBE system must be defined. This ontology captures the specific concepts, relationships, and rules that govern the targeted KBE system. In this research work, the 'ParaPy Ontology' was identified to encapsulate the key aspects of the target KBE system.

Once the knowledge ontology and the target KBE system ontology have been established, the next step is to define a mapping between the two. This mapping links the relevant concepts and relationships from the knowledge ontology to their corresponding counterparts in the target KBE system ontology. In this research work, a mapping between the 'Knowledge Ontology' and the 'ParaPy Ontology' was defined.

Finally, the development of translation engines plays a vital role in bridging the gap between knowledge formalization and code formulation. These translation engines automate the process of converting the knowledge ontology into the ontology of the target KBE system. This research work included the development of translation engines that automatically generate the ParaPy code that embodies the captured knowledge and enables the functionality of the KBE application, ensuring a seamless transition of knowledge from the formalized representation to the KBE system.

Q.1.3. ***“What is the maturity of the automatically generated KBE code in terms of completeness, reliability, and quality?”***

The maturity of the automatically generated KBE application can vary depending on the level of detail incorporated into the knowledge model. At the very least, the automatically generated KBE application encompasses the essential skeleton code that forms the foundation of the application, whose development is acknowledged to be a critical part in the overall KBE app development process. However, a more mature app can also be achieved if the user chooses to model the detailed engineering rules and expressions in the knowledge model.

Nevertheless, the automatically generated code skeleton is of good quality, comparable to that achievable through manual creation by a KBE expert. Additionally, it establishes a well-structured and accurate starting point for further application development, leading to time savings in the development of the application’s skeleton code.

Q.1.4. ***How to guarantee the synchronization and consistency between the knowledge models and the KBE application encoding that knowledge?***

Ensuring synchronization and consistency between knowledge models and the encoding of that knowledge in the KBE application can be achieved through the development of dedicated translators that facilitate the seamless conversion of the knowledge model into source code for the targeted KBE system.

When modifications are made to the knowledge model, the dedicated translators automatically generate a new KBE skeleton code, thus guaranteeing synchronization between the model and the code. This process enables developers to maintain consistency and coherence between the knowledge representation and its implementation in the KBE application. However, the current workflow lacks the capability to propagate changes made directly in the code back into the knowledge model, hindering the bidirectional synchronization. To address this limitation, the development of “round-tripping” capabilities is suggested. These capabilities should empower developers to incorporate changes made in the KBE app code back into the knowledge model, allowing for efficient updates and modifications. Additionally, this functionality should enable the creation of knowledge models from existing KBE apps, facilitating knowledge extraction.

8.2. Future Recommendations

Considering the conclusions derived from the present research and the identified limitations, several recommendations for future work can be suggested. These recommendations can be summarized as follows:

1. Create or customize a modeling tool, such as ParaPy’s Visual Editor or Magic Systems of Systems Architect, to enhance the KBE application development process. The customization should focus on adding or modifying specific features to better support the KBE modeling needs, as well as removing unnecessary features to reduce the complexity of these tools.

2. Incorporate a round-trip functionality to establish bidirectional synchronization between the knowledge model and the generated code. This functionality would enable changes made in the KBE application code to be automatically reflected in the knowledge model, ensuring consistency and maintaining synchronization.

3. Implement protected blocks in the automatically generated code to safeguard user-written attributes and critical code sections. For instance, special comments, such as *'#start_protected_block'* and *'#end_protected_block'*, could be used to designate editable areas when re-importing the code into the SysML model or re-generating the skeleton code after modifications to the knowledge model. The inclusion of such functionality, similar to the capabilities provided by tools like Eclipse Acceleo[1], would enhance the flexibility and maintainability of the automatically generated code.

4. Create a Process traceability tab within the graphical user interface (GUI) to facilitate traceability between activities, actions, and product elements in the KBE application. This tab would allow engineers to ensure that each function of the KBE application is appropriately allocated to a corresponding product element, providing a comprehensive view of the implementation, and aiding in validation.

5. Recreate the ParaPy Primitives, including their associated inputs and attributes, in a format that can be automatically imported at the start of each new project in MSoSA, for example using the XMI standard. This could involve creating a SysML model containing all existing ParaPy Primitives to expedite the modeling workflow and provide a solid foundation for further development. Alternatively, if a round-trip functionality is available, it could be used to parse the existing ParaPy code base and automatically generate a neutral format that can be imported into MSoSA.

6. Expand the current database of preset processes and products available within the developed knowledge repository. Increasing the repertoire of pre-existing processes and associated products would provide users with a broader range of preset options and facilitate more efficient development of KBE applications.

7. Expanding the current capabilities of the developed translation engines in order to support the definition of more than one class per module.

8. Explore the development of translation engines that support other KBE systems beyond the capabilities developed in this research. This expansion would enable the generation of code for a broader range of KBE platforms, enhancing the applicability and versatility of the developed framework.

9. Define an ontology that encompasses the modeling of user interface (UI) elements in KBE applications. By incorporating UI modeling into the knowledge model, translation engines can be developed to automatically convert these models into executable code, facilitating the implementation of the UI aspects of KBE applications.

[1]`https://www.eclipse.org/acceleo/` (Accessed: 20/06/2023)

References

[1] DEFAINE Consortium. *DEFAINE - Design Exploration Framework Based on AI for froNt-loaded Engineering - Full Project Proposal Annex*. Mar. 2021.

[2] S. Thomke and T. Fujimoto. “The Effect of “Front-Loading” Problem-Solving on Product Development Performance”. In: *Journal of Product Innovation Management* 17.2 (Mar. 2000), pp. 128–142. ISSN: 0737-6782. DOI: 10.1016/S0737-6782(99)00031-4.

[3] A. R. Kulkarni et al. “A Knowledge Based Engineering Tool to Support Front-Loading and Multidisciplinary Design Optimization of the Fin-Rudder Interface”. In: *Aerospace Europe 6th CEAS Conference*. Oct. 2017.

[4] G. La Rocca. “Knowledge Based Engineering: Between AI and CAD. Review of a Language Based Technology to Support Engineering Design”. In: *Advanced Engineering Informatics*. Knowledge Based Engineering to Support Complex Product Design 26.2 (Apr. 2012), pp. 159–179. ISSN: 1474-0346. DOI: 10.1016/j.aei.2012.02.002.

[5] M. Stokes, ed. *Managing Engineering Knowledge: MOKA Methodology for Knowledge Based Engineering Applications*. London Bury St. Edmunds, 2001. ISBN: 978-1-86058-295-0.

[6] W. J. C. Verhagen et al. “A Critical Review of Knowledge-Based Engineering: An Identification of Research Challenges”. In: *Advanced Engineering Informatics*. Network and Supply Chain System Integration for Mass Customization and Sustainable Behavior 26.1 (July 2011), pp. 5–15. ISSN: 1474-0346. DOI: 10.1016/j.aei.2011.06.004.

[7] P. Bermell-Garcia. “A Metamodel to Annotate Knowledge Based Engineering Codes as Enterprise Knowledge Resources”. In: (Apr. 2007). (Visited on 06/21/2023).

[8] C. Van der Velden, C. Bil, and X. Xu. “Adaptable Methodology for Automation Application Development”. In: *Advanced Engineering Informatics* 26 (Apr. 2012), pp. 231–250. DOI: 10.1016/j.aei.2012.02.007.

[9] C. Chapman et al. “Utilising Enterprise Knowledge with Knowledge-Based Engineering”. In: *International Journal of Computer Applications in Technology* 28.2/3 (Apr. 2007), pp. 169–179. ISSN: 0952-8091. DOI: 10.1504/IJCAT.2007.013354.

[10] S. Friedenthal, A. Moore, and R. Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. 3rd edition. Boston: Morgan Kaufmann, 2015. ISBN: 978-0-12-800202-5.

[11] C. Chapman and M. Pinfold. “Design Engineering—a Need to Rethink the Solution Using Knowledge Based Engineering”. In: *Knowledge-Based Systems* 12.5 (Oct. 1999), pp. 257–267. ISSN: 0950-7051. DOI: 10.1016/S0950-7051(99)00013-1.

[12] S. Cooper, I. S. Fan, and G. Li. *Achieving Competitive Advantage Through Knowledge-Based Engineering - A Best Practice Guide*. 2001.

[13] A. H. Van der Laan and M. J. L. Van Tooren. “Parametric Modeling of Movables for Structural Analysis”. In: *Journal of Aircraft* 42.6 (Nov. 2005), pp. 1605–1613. DOI: 10.2514/1.9764.

[14] B. Vermeulen. “Knowledge Based Method for Solving Complexity in Design Problems”. PhD thesis. Delft University of Technology, 2007. (Visited on 01/16/2022).

[15] G. Frank et al. “Towards a Generic Framework of Engineering Design Automation for Creating Complex CAD Models”. In: *International Journal of Advances in Systems and Measurements* 7 (June 2014), pp. 179–192.

[16] W. Skarka. “Application of MOKA Methodology in Generative Model Creation Using CATIA”. In: *Engineering Applications of Artificial Intelligence*. Soft Computing Applications 20.5 (Aug. 2007), pp. 677–690. ISSN: 0952-1976. DOI: `10.1016/j.engappai.2006.11.019`.

[17] G. La Rocca. “Knowledge Based Engineering Techniques to Support Aircraft Design and Optimization”. PhD thesis. Delft University of Technology, 2011.

[18] G. Schreiber et al. “CommonKADS: A Comprehensive Methodology for KBS Development”. In: *IEEE Expert* 9 (Dec. 1994), pp. 28–37. ISSN: 2374-9407. DOI: `10.1109/64.363263`.

[19] J. Forster, I. Arana, and P. Fothergill. “Re-Design Knowledge Representation with DEKLARE”. In: *Proceedings of KEML 1996 6th Workshop on Knowledge Engineering: Methods & Languages, Paris, France*. Citeseer, 1996.

[20] P. J Lovett, A Ingram, and C. N Bancroft. “Knowledge-Based Engineering for SMEs — a Methodology”. In: *Journal of Materials Processing Technology* 107.1 (Nov. 2000), pp. 384–389. ISSN: 0924-0136. DOI: `10.1016/S0924-0136(00)00728-7`.

[21] JP Terpenny, S. Strong, and J. Wang. “A Methodology for Knowledge Discovery and Classification”. In: *Proc. 10th Flexible Autom. Intell. Manuf. Conf.* Citeseer, 2000, pp. 22–32.

[22] R. Curran et al. “A Multidisciplinary Implementation Methodology for Knowledge Based Engineering: KNOMAD”. In: *Expert Systems with Applications*. Advances in Aligning Knowledge Systems, Improving Business Logistics, Driving Innovation and Adapting Customer Centric Services 37.11 (Nov. 2010), pp. 7336–7350. ISSN: 0957-4174. DOI: `10.1016/j.eswa.2010.04.027`.

[23] I. O. Sanya and E. M. Shehab. “An Ontology Framework for Developing Platform-Independent Knowledge-Based Engineering Systems in the Aerospace Industry”. In: *International Journal of Production Research* 52.20 (Oct. 2014), pp. 6192–6215. ISSN: 0020-7543, 1366-588X. DOI: `10.1080/00207543.2014.919422`.

[24] P. K. M. Chan. “A New Methodology for the Development of Simulation Workflows: Moving Beyond MOKA”. MSc Thesis. Delft University of Technology, 2013.

[25] K. Wheeler. “Methodological Support for Knowledge Based Engineering Application Development: Improving Traceability of Knowledge into Application Code”. MSc Thesis. Delft University of Technology, 2020.

[26] C. A. R. Torres. “A Knowledge-Based Engineering System Framework for the Development of Electric Machines”. PhD thesis. Mondragon Unibertsitatea. Goi Eskola Politeknikoa, 2019.

[27] MOKA Consortium. *Methodology and Tools Oriented to Knowledge-based Engineering Applications Task 4.3 Final Synthesis DELIVERABLE D4.3*. June 2000.

[28] S. Preston et al. “Knowledge Acquisition for Knowledge-Based Engineering Systems”. In: *International Journal of Information Technology and Management* 4.1 (Mar. 2005), pp. 1–11. ISSN: 1461-4111. DOI: `10.1504/IJITM.2005.006401`.

[29] P. J. A. R. Dewitte. “Development and Reuse of Engineering Automation”. MSc Thesis. Delft University of Technology, 2014.

[30] I. S. Fan and P. Bermell-Garcia. “International Standard Development for Knowledge Based Engineering Services for Product Lifecycle Management”. In: *Concurrent Engineering* 16.4 (Dec. 2008), pp. 271–277. ISSN: 1063-293X. DOI: 10.1177/1063293X08100027.

[31] R. Curran, W. J. C. Verhagen, and M. J. L. Van Tooren. “The KNOMAD Methodology for Integration of Multidisciplinary Engineering Knowledge Within Aerospace Production”. In: *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*. Aerospace Sciences Meetings. American Institute of Aeronautics and Astronautics, Jan. 2010. DOI: 10.2514/6.2010-1315.

[32] W. J. C. Verhagen and R. Curran. “Knowledge-Based Engineering Review: Conceptual Foundations and Research Issues”. In: *New World Situation: New Directions in Concurrent Engineering*. Ed. by J. Pokojski, S. Fukuda, and J. Salwiński. London: Springer London, 2010, pp. 267–276. ISBN: 978-0-85729-024-3.

[33] D. Baxter et al. “An Engineering Design Knowledge Reuse Methodology Using Process Modelling”. In: *Research in Engineering Design* 18.1 (May 2007), pp. 37–48. ISSN: 0934-9839, 1435-6066. DOI: 10.1007/s00163-007-0028-8. (Visited on 02/25/2023).

[34] International Council on Systems Engineering (INCOSE). *Systems Engineering Vision 2020*. Sept. 2007.

[35] International Council on Systems Engineering (INCOSE). *Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities*. Ed. by D. D. Walden et al. 4th edition. Hoboken, New Jersey: Wiley, 2015. ISBN: 978-1-118-99941-7 978-1-119-01512-3.

[36] C. Haskins. “4.6.1 A Historical Perspective of MBSE with a View to the Future”. In: *INCOSE International Symposium* 21.1 (June 2011), pp. 493–509. ISSN: 23345837. DOI: 10.1002/j.2334-5837.2011.tb01220.x.

[37] J. Holt and S. Perry. *SysML for Systems Engineering: A Model-Based Approach*. 3d edition. IET Professional Applications of Computing Series 20. Stevenage: The Institution of Engineering and Technology, 2018. ISBN: 978-1-78561-554-2.

[38] C. Moreland. *An Introduction to the OMG Systems Modeling Language (OMG SysML™)*. 2008.

[39] S. Sint et al. “Thirteen Years of SysML: A Systematic Mapping Study”. In: *Software & Systems Modeling* 19 (Jan. 2020). DOI: 10.1007/s10270-019-00735-y.

[40] R. Cloutier and M. Bone. “Compilation of SysML RFI-Final Report”. In: *Stevens Institute of Technology, School of Systems & Enterprises* (2010).

[41] J. A. Estefan. “Survey of Model-Based Systems Engineering (MBSE) Methodologies”. In: *INCOSE MBSE Initiative* 25.8 (2007), pp. 1–12.

[42] D. Dori. *Object-Process Methodology: A Holistics Systems Paradigm*. Berlin ; New York: Springer, 2002. ISBN: 978-3-540-65471-1.

[43] P. Kruchten. *The Rational Unified Process: An Introduction*. 3rd ed. The Addison-Wesley Object Technology Series. Boston: Addison-Wesley, 2004. ISBN: 978-0-321-19770-2.

[44] A. Aleksandravičienė and A. Morkevičius. *MagicGrid Book of Knowledge: A Practical Guide to Systems Modeling Using MagicGrid from Dassault Systèmes*. 2nd edition. Kaunas, Lithuania: Vitae Litera, 2021. ISBN: 978-609-454-554-2.

[45] A. Morkevicius et al. “MBSE Grid: A Simplified SysML-Based Approach for Modeling Complex Systems”. In: *INCOSE International Symposium* 27 (July 2017), pp. 136–150. DOI: 10.1002/j.2334-5837.2017.00350.x.

[46] B. Yu and H. Zhao. “A Semantic Ontology of Requirement – Product – Process – Resource for Modeling of Product Lifecycle Information”. In: *The 19th International Conference on Industrial Engineering and Engineering Management*. Ed. by E. Qi, J. Shen, and R. Dou. Berlin, Heidelberg: Springer, 2013, pp. 319–330. ISBN: 978-3-642-37270-4. DOI: 10.1007/978-3-642-37270-4_31.

[47] S. Rugaber and K. Stirewalt. “Model-Driven Reverse Engineering”. In: *IEEE Software* 21.4 (July 2004), pp. 45–53. ISSN: 1937-4194. DOI: 10.1109/MS.2004.23.

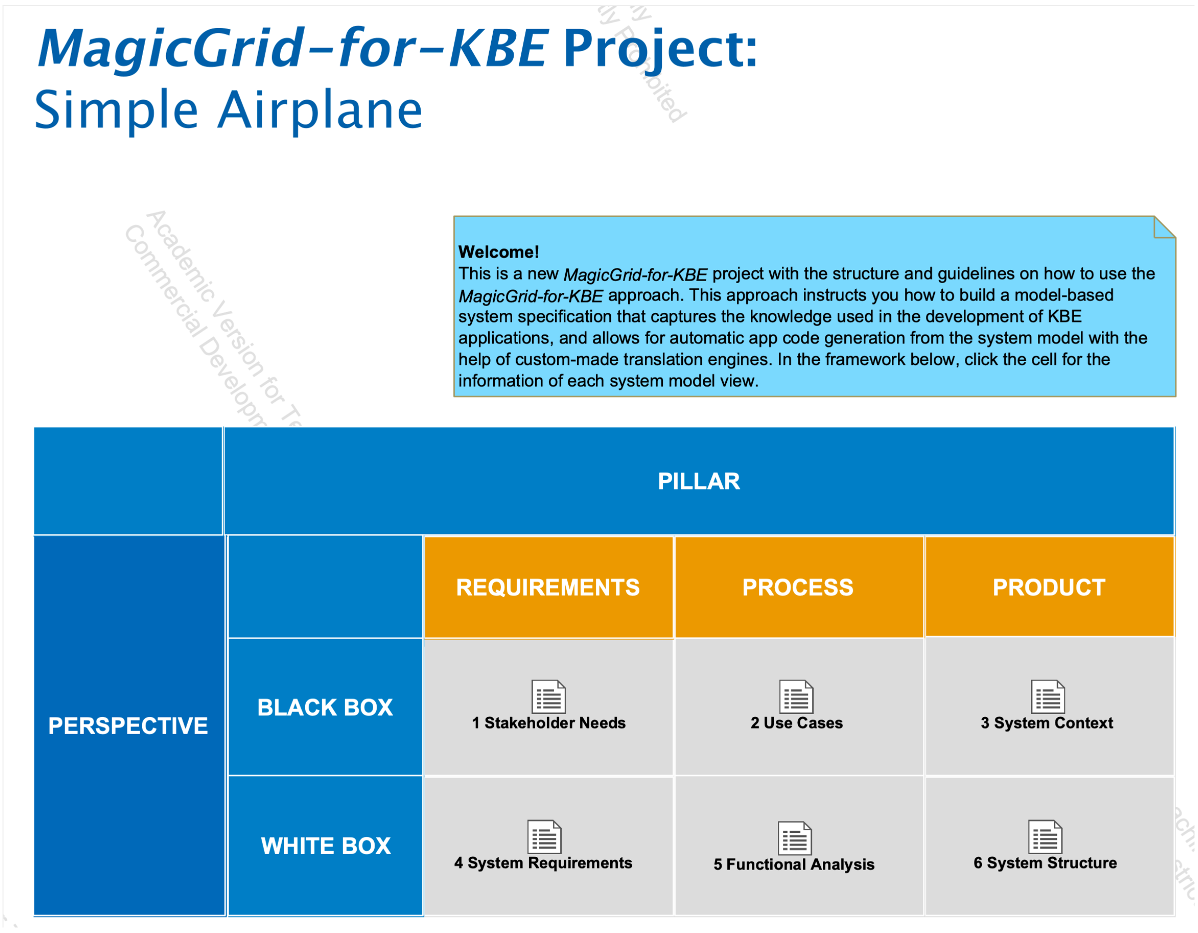
[48] G. Lapalme. “XML: Looking at the Forest Instead of the Trees”. In: *Disponible sur http://www. iro. umontreal. ca/~ lapalme/ForestInsteadOfTheTrees* (2007).

[49] J. Arlow and I. Neustadt. *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*. 2nd ed. Upper Saddle River, NJ: Addison-Wesley, 2005. ISBN: 978-0-321-32127-5.

Appendices

A
System Model Views of the MagicGrid-for-KBE Framework

The various system model views defined by the MagicGrid-for-KBE method, and implemented as a template in Magic Systems of Systems Architect to facilitate the modeling process of KBE Applications are depicted here. The presented examples are from the system model of the Simple Airplane KBE application developed in this research.

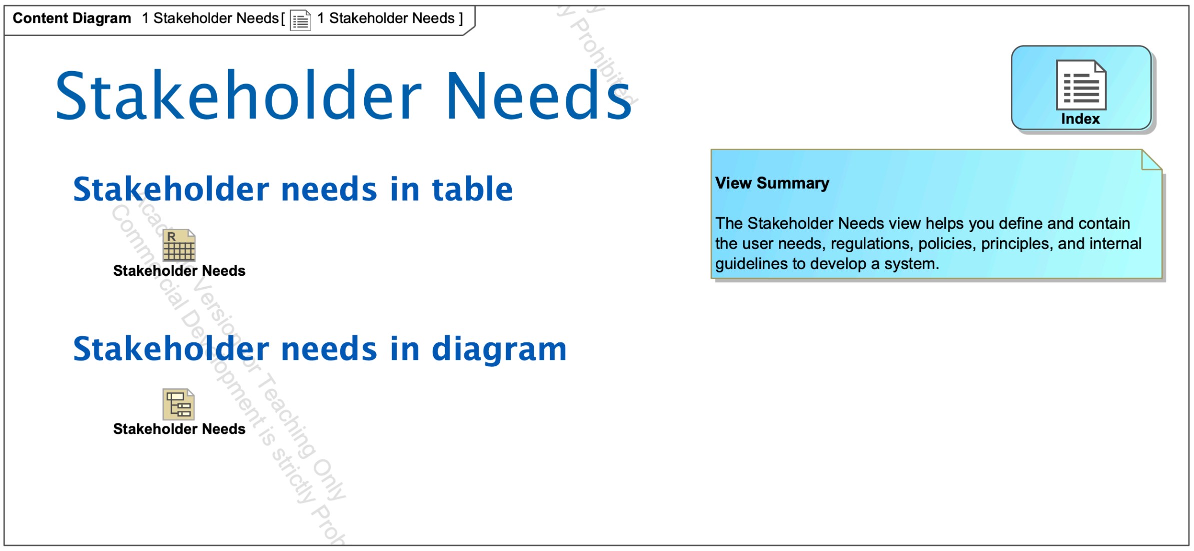


***MagicGrid-for-KBE* Project:**
Simple Airplane

Welcome!
This is a new *MagicGrid-for-KBE* project with the structure and guidelines on how to use the *MagicGrid-for-KBE* approach. This approach instructs you how to build a model-based system specification that captures the knowledge used in the development of KBE applications, and allows for automatic app code generation from the system model with the help of custom-made translation engines. In the framework below, click the cell for the information of each system model view.

		PILLAR		
		REQUIREMENTS	PROCESS	PRODUCT
PERSPECTIVE	BLACK BOX	1 Stakeholder Needs	2 Use Cases	3 System Context
	WHITE BOX	4 System Requirements	5 Functional Analysis	6 System Structure

Figure A.1: Start Page of the MagicGrid-for-KBE model template



Stakeholder Needs

Stakeholder needs in table

Stakeholder needs in diagram

View Summary

The Stakeholder Needs view helps you define and contain the user needs, regulations, policies, principles, and internal guidelines to develop a system.

Figure A.2: Stakeholder Needs View

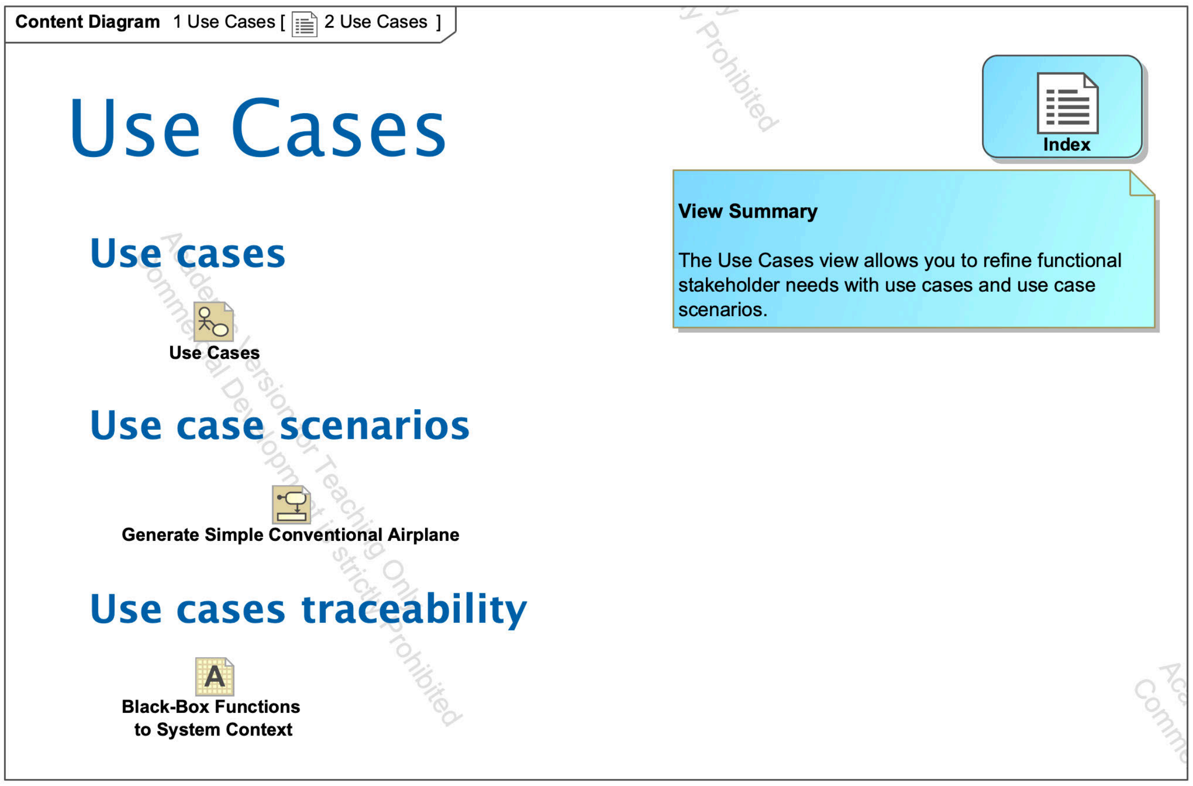

Content Diagram 1 Use Cases [2 Use Cases]
Use Cases
Use cases
Use Cases
Use case scenarios
Generate Simple Conventional Airplane
Use cases traceability
Black-Box Functions to System Context
Index
View Summary
The Use Cases view allows you to refine functional stakeholder needs with use cases and use case scenarios.

Figure A.3: Use Cases View

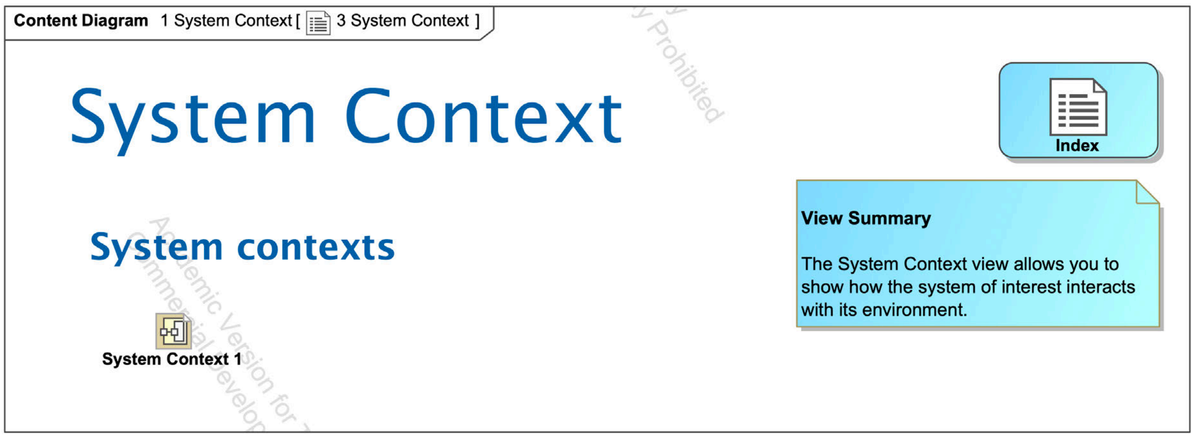
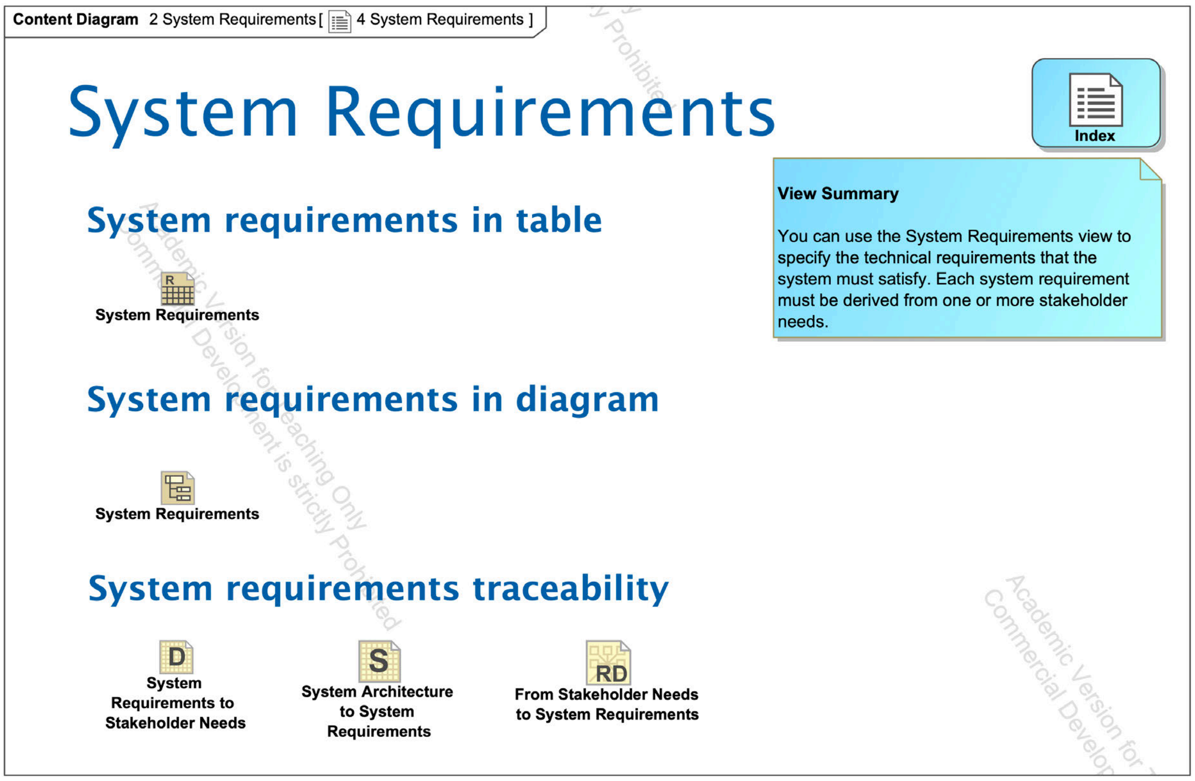

Content Diagram 1 System Context [3 System Context]
System Context
System contexts
System Context 1
Index
View Summary
The System Context view allows you to show how the system of interest interacts with its environment.

Figure A.4: System Context View



Content Diagram 2 System Requirements [4 System Requirements]

System Requirements

System requirements in table

System Requirements

System requirements in diagram

System Requirements

System requirements traceability

System Requirements to Stakeholder Needs

System Architecture to System Requirements

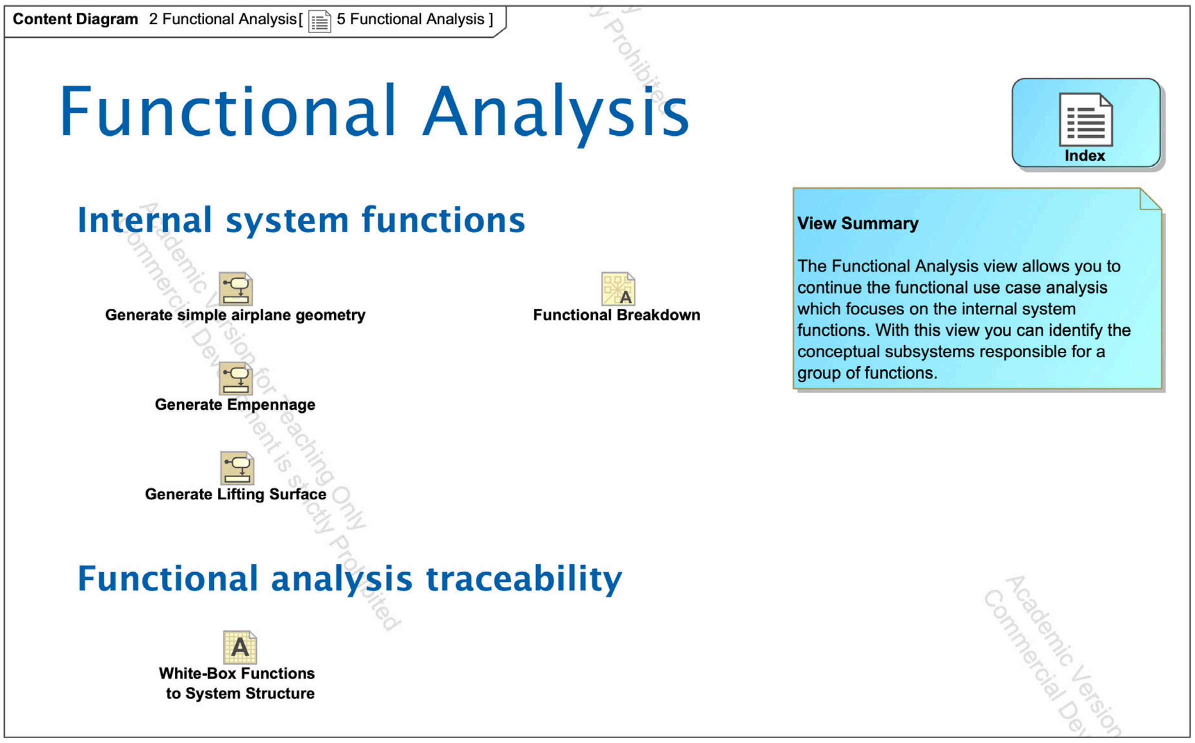
From Stakeholder Needs to System Requirements

Index

View Summary

You can use the System Requirements view to specify the technical requirements that the system must satisfy. Each system requirement must be derived from one or more stakeholder needs.

Figure A.5: System Requirements View



Content Diagram 2 Functional Analysis [5 Functional Analysis]

Functional Analysis

Internal system functions

Generate simple airplane geometry

Functional Breakdown

Generate Empennage

Generate Lifting Surface

Functional analysis traceability

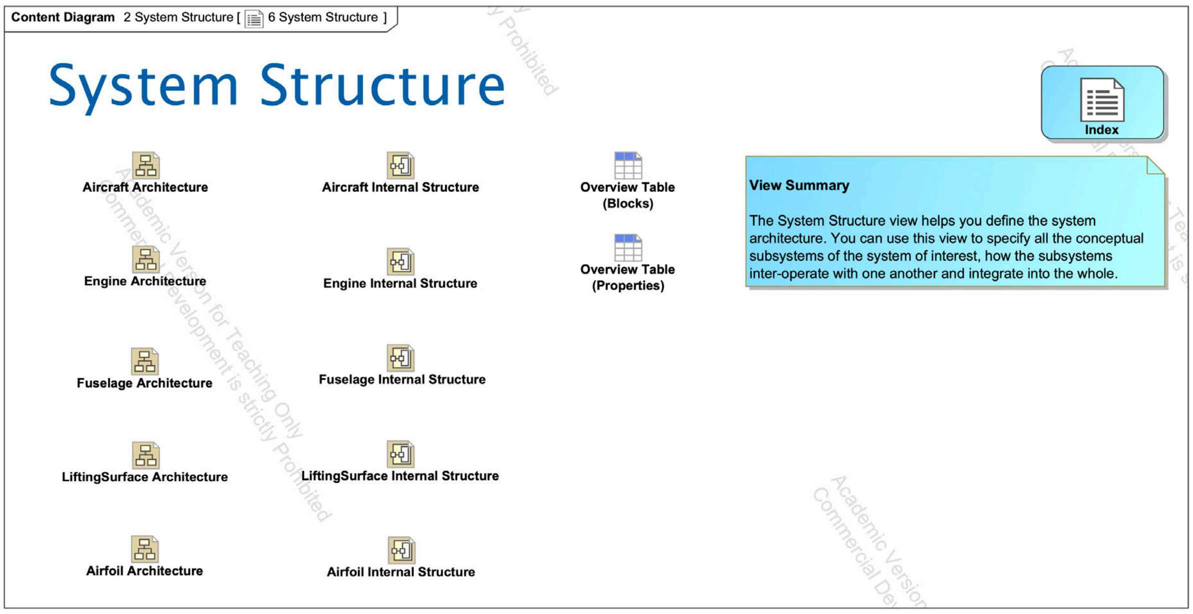
White-Box Functions to System Structure

Index

View Summary

The Functional Analysis view allows you to continue the functional use case analysis which focuses on the internal system functions. With this view you can identify the conceptual subsystems responsible for a group of functions.

Figure A.6: Functional Analysis View


Content Diagram 2 System Structure [6 System Structure]
System Structure
Index
Aircraft Architecture
Aircraft Internal Structure
Overview Table (Blocks)
Engine Architecture
Engine Internal Structure
Overview Table (Properties)
Fuselage Architecture
Fuselage Internal Structure
LiftingSurface Architecture
LiftingSurface Internal Structure
Airfoil Architecture
Airfoil Internal Structure

View Summary

The System Structure view helps you define the system architecture. You can use this view to specify all the conceptual subsystems of the system of interest, how the subsystems inter-operate with one another and integrate into the whole.

Figure A.7: System Structure View

B

JSON Schema of the Neutral Language Knowledge Model

The JSON Schema of the Neutral Language Knowledge Model is defined across several files which are stored in a directory with the following tree structure:

```
[JSON Schema]
├── JSON_for_KBE.json
└── [sub-schemas]
    ├── Requirement_Schema.json
    ├── Package_Schema.json
    ├── Class_Schema.json
    └── [class sub-schemas]
        ├── Input_Schema.json
        ├── Attribute_Schema.json
        ├── Part_Schema.json
        ├── Action_Schema.json
        └── Method_Schema.json
```

The contents of each file are presented ahead:

Listing B.1: JSON_for_KBE.json

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "Main_Schema.json",
    "title": "JSON Schema for KBE",
    "description": "The schema of the JSON file used for automatically generating KBE (ParaPy
        ) source code via the developed translation engine",
    "type": "object",
    "properties": {
        "Product": {
            "description": "The package that contains all the information about the structure
                 of the KBE application and that later will be translated into the source
                code of the app",
            "type": "object",
            "additionalProperties": {
                "oneOf": [
                    {"$ref": "./sub_schemas/Package_Schema.json"},
                    {"$ref": "./sub_schemas/Class_Schema.json"}
                ]
            }
        },
        "Requirements": {
            "description": "The package that contains all the information about the
                requirements of the KBE application and that later will be used to allow
                traceability between the model and the code",
            "type": "object",
            "additionalProperties": {
                "$ref": "./sub_schemas/Requirement_Schema.json"
            }
        }
    },
    "required": [
        "Product",
        "Requirements"
    ],
    "additionalProperties": false
}
```

Listing B.2: Requirement_Schema.json

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "Requirement_Schema.json",
    "title": "Requirement Schema",
    "description": "The schema of a single Requirement element",
    "type": "object",
    "properties": {
        "derived from": {
            "description": "The requirement(s) from which the given requirement is derived
                from",
            "type": "array",
            "items": {
                "type": "array",
                "items": {
                    "type": "string"
                },
                "minItems": 2,
                "maxItems": 2
            }
        },
        "id": {
            "description": "The id of the given requirement",
            "type": "string"
        },
        "name": {
            "description": "The name of the given requirement",
            "type": "string"
        },
        "satisfy slots": {
            "description": "The KBE slots that satisfy the given requirement",
            "type": "object",
            "additionalProperties": {
                "$ref": "#/$defs/satisfy_slot"
            }
        },
        "text": {
            "description": "The text of the given requirement",
            "type": "string"
        },
        "type": {
            "description": "The type of the given requirement",
            "type": "string"
        }
    },
    "required": [
        "derived from",
        "id",
        "name",
        "satisfy slots",
        "text",
        "type"
    ],
    "additionalProperties": false,
    "$defs": {
        "satisfy_slot": {
            "description": "A slot that satisfies the given requirement",
            "type": "object",
            "properties": {
                "slot name": {
                    "description": "The name of the given slot",
                    "type": "string"
                },
                "slot owner class": {
                    "description": "The name of the class that owns the given slot",
                    "type": "string"
                },
                "slot path in model": {
                    "description": "The path to the given slot in the product model",
                    "type": "array",
                    "items": {
                        "type": "string"
                    }
                },
                "slot type": {
                    "description": "The type of the given slot",
                    "type": "string"
                }
            },
            "required": [
                "slot name",
                "slot owner class",
                "slot path in model",
                "slot type"
            ]

        }
    }
}
```

Listing B.3: Package_Schema.json

```json
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "Package_Schema.json",
    "title": "Package Schema",
    "description": "The schema of a single Package element",
    "type": "object",
    "not": {
        "type": "object",
        "properties": {
            "name": {
                "type": "string"
            }
        },
        "required": [
            "name"
        ]
    },
    "additionalProperties": {
        "description": "A Package may contain other packages or classes",
        "oneOf": [
            {"$ref": "#"},
            {"$ref": "Class_Schema.json"}
        ]
    }
}
```

Listing B.4: Class_Schema.json

```json
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "Class_Schema.json",
    "title": "Class Schema",
    "description": "The schema of a single Class element",
    "type": "object",
    "properties": {
        "actions": {
            "description": "The action slots of the given KBE class",
            "type": "object",
            "additionalProperties": {
                "$ref": "./class_sub_schemas/Action_Schema.json"
            }
        },
        "attributes": {
            "description": "The attribute slots of the given KBE class",
            "type": "object",
            "additionalProperties": {
                "$ref": "./class_sub_schemas/Attribute_Schema.json"
            }
        },
        "documentation": {
            "description": "The description/docstring of the given KBE class",
            "type": "string"
        },
        "hyperlinks": {
            "description": "The hyperlinks to the given KBE class",
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "import methods": {
            "description": "The methods imported by the given KBE class",
            "type": "object"
        },
        "inheritance": {
            "description": "The classes from which the given KBE class inherits from",
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "inputs": {
            "description": "The input slots of the given KBE class",
            "type": "object",
            "additionalProperties": {
                "$ref": "./class_sub_schemas/Input_Schema.json"
            }
        },
        "methods": {
            "description": "The methods of the given KBE class",
            "type": "object",
            "additionalProperties": {
                "$ref": "./class_sub_schemas/Method_Schema.json"
            }
        },
        "name": {
            "description": "The name of the given KBE class",
            "type": "string"
        },
        "parts": {
            "description": "The part slots of the given KBE class",
            "type": "object",
            "additionalProperties": {
                "$ref": "./class_sub_schemas/Part_Schema.json"
            }
        }
    },
    "required": [
        "actions",
        "attributes",
        "documentation",
        "hyperlinks",
        "import methods",
        "inheritance",
        "inputs",
        "methods",
        "name",
        "parts"
    ],
    "additionalProperties": false
}
```

Listing B.5: Input_Schema.json

```json
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "Input_Schema.json",
    "title": "Input Schema",
    "description": "The schema of a single Input element of a KBE class",
    "type": "object",
    "properties": {
        "default": {
            "description": "The default value of the given input",
            "type": ["number", "string", "boolean", "object", "array", "null"]
        },
        "description": {
            "description": "The description/docstring of the given input",
            "type": "string"
        },
        "hyperlinks": {
            "description": "The hyperlinks to the given input",
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "input kind": {
            "description": "The kind of the given input (i.e, 'required', 'optional', or '
                derived')",
            "type": "string"
        },
        "name": {
            "description": "The name of the given input",
            "type": "string"
        },
        "sources": {
            "description": "The source elements of the given input",
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "targets": {
            "description": "The target elements of the given input",
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "type": {
            "description": "The type of the given input",
            "type": "string"
        }
    },
    "required": [
        "default",
        "description",
        "hyperlinks",
        "input kind",
        "name",
        "sources",
        "targets",
        "type"
    ],
    "additionalProperties": false
}
```

Listing B.6: Attribute_Schema.json

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "Attribute_Schema.json",
    "title": "Attribute Schema",
    "description": "The schema of a single Attribute element of a KBE class",
    "type": "object",
    "properties": {
        "code": {
            "description": "The code that allows to compute the expression of the given
                attribute",
            "type": "string"
        },
        "description": {
            "description": "The description/docstring of the given attribute",
            "type": "string"
        },
        "hyperlinks": {
            "description": "The hyperlinks to the given attribute",
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "name": {
            "description": "The name of the given attribute",
            "type": "string"
        },
        "sources": {
            "description": "The source elements of the given attribute",
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "targets": {
            "description": "The target elements of the given attribute",
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "type": {
            "description": "The type of the given attribute",
            "type": "string"
        }
    },
    "required": [
        "code",
        "description",
        "hyperlinks",
        "name",
        "sources",
        "targets",
        "type"
    ],
    "additionalProperties": false
}
```

Listing B.7: Part_Schema.json

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "Part_Schema.json",
    "title": "Part Schema",
    "description": "The schema of a single Part element of a KBE class",
    "type": "object",
    "properties": {
        "description": {
            "description": "The description/docstring of the given part",
            "type": "string"
        },
        "hyperlinks": {
            "description": "The hyperlinks to the given part",
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "name": {
            "description": "The name of the given part",
            "type": "string"
        },
        "part inputs":{
            "description": "The input elements of the given part",
            "type": "object",
            "additionalProperties": {
                "$ref": "#/$defs/part_input"
            }
        },
        "part name": {
            "description": "The name of the classifier of the given part",
            "type": "string"
        },
        "targets": {
            "description": "The target elements of the given part",
            "type": "array",
            "items": {
                "type": "string"
            }
        }
    },
    "required": [
        "description",
        "hyperlinks",
        "name",
        "part inputs",
        "part name",
        "targets"
    ],
    "additionalProperties": false,
    "$defs":{
        "part_input": {
            "description": "An input element of the given part",
            "type": "object",
            "properties": {
                "expression": {
                    "description": "The code that allows to compute the expression of the
                        given part input",
                    "type": "string"
                },
                "sources": {
                    "description": "The source elements of the given part input",
                    "type": "array",
                    "items": {
                        "type": "string"
                    }
                },
                "targets": {
                    "description": "The target elements of the given part input",
                    "type": "array",
                    "items": {
                        "type": "string"
                    }
                }
            },
            "additionalProperties": false,
            "required": [
                "expression",
                "sources",
                "targets"
            ]
        }
    }
}
```

Listing B.8: Action_Schema.json

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "Action_Schema.json",
    "title": "Action Schema",
    "description": "The schema of a single Action element of a KBE class",
    "type": "object",
    "properties": {
        "code": {
            "description": "The code that allows to compute the expression of the given
                action",
            "type": "string"
        },
        "description": {
            "description": "The description/docstring of the given action",
            "type": "string"
        },
        "hyperlinks": {
            "description": "The hyperlinks to the given action",
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "name": {
            "description": "The name of the given action",
            "type": "string"
        },
        "sources": {
            "description": "The source elements of the given action",
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "targets": {
            "description": "The target elements of the given action",
            "type": "array",
            "items": {
                "type": "string"
            }
        }
    },
    "required": [
        "code",
        "description",
        "hyperlinks",
        "name",
        "sources",
        "targets"
    ],
    "additionalProperties": false
}
```

Listing B.9: Method_Schema.json

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "Method_Schema.json",
    "title": "Method Schema",
    "description": "The schema of a single Method element of a KBE class",
    "type": "object",
    "properties": {
        "code": {
            "description": "The code that allows to compute the expression of the given
                method",
            "type": "string"
        },
        "description": {
            "description": "The description/docstring of the given method",
            "type": "string"
        },
        "hyperlinks": {
            "description": "The hyperlinks to the given method",
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "name": {
            "description": "The name of the given method",
            "type": "string"
        },
        "targets": {
            "description": "The target elements of the given method",
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "type": {
            "description": "The type of the given method",
            "type": "string"
        }
    },
    "required": [
        "code",
        "description",
        "hyperlinks",
        "name",
        "targets",
        "type"
    ],
    "additionalProperties": false
}
```