Methodological Support for Knowledge Based Engineering Application Development

Improving Traceability of Knowledge into Application Code

K. Wheeler

Technische Universiteit Delft

TU Delft Delft University of Technology

Methodological Support for Knowledge Based Engineering Application Development

Improving Traceability of Knowledge into Application Code

by

K. Wheeler

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday March 9, 2020 at 13:30.

Student number:	4280806	
Project duration:	September 1, 2018 – March 9, 2020	
Supervised by:	Ir. R. E. C. van Dijk,	ParaPy
	Dr. Ir. G. la Rocca,	TU Delft
Thesis committee:	Prof. Dr. Ir. L. L. M. Veldhuis,	TU Delft, Chair
	Dr. Ir. G. la Rocca,	TU Delft, First
	Dr. Ir. O. A. Sharpanskykh,	TU Delft, Second

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Abstract

Knowledge Based Engineering (KBE) applications reduce the amount of repetitive work and allow more time for innovative design as engineering knowledge is translated into application code. The problem is that these KBE applications are non-transparent and are perceived as a black box as they generate an output from some input, but it is unclear how they came to their conclusion. One of the well known KBE development methodologies, MOKA, has proposed solutions to structure knowledge that have proven to have a positive impact on the development of KBE applications. However, knowledge structured with MOKA is disconnected from the application code, making it difficult to trace how knowledge has been implemented which decreases the transparency of KBE applications. Moreover, knowledge seems to not be modelled at all, enforcing the black box perception. A methodology like MOKA is not being used due to one of the assumptions that a domain expert, who owns the knowledge, is a different person than the developer that develops the application. This requires a specialist called a knowledge engineer, which many companies do not have. This specialist can translate the knowledge from the domain expert into models that can be understood by developers. A field study showed that in contrast to the assumption made by MOKA, the domain expert is often also the developer. This opens up possibilities for a new methodology that bypasses the necessity of these specialists. This in combination with the original motivation to improve the transparency of KBE application by improving the traceability of knowledge into application code made it possible to formulate the research question of this thesis: to what extent can a methodology to structure knowledge improve the traceability of knowledge onto KBE application code without the need of specialised knowledge engineers?

An ontology called the Traceability Model (TM) has been developed that comprises two ontologies: a Knowledge Ontology and a ParaPy Ontology. The Knowledge Ontology defines different classes of Knowledge Forms (KF) that each capture a different aspect of knowledge: e.g. product hierarchy, design steps, restrictions. A KF has pre-defined fields that must be filled in by a domain expert, and it can be related to other KFs to create models. The ParaPy Ontology defines the different Code Objects (CO) that can be found in ParaPy application code, such as `Classes` and `Inputs`. The TM explains how knowledge connects to application code by defining a connection between the Knowledge Ontology and the ParaPy Ontology. This enables the traceability of knowledge into application code. Furthermore, a platform called the Knowledge Portal (KP) has been developed that enables the creation of KFs, automatically parses application code into COs, and allows users to connect KFs to COs. It also automatically generates several knowledge representations to help a user understand the knowledge that has been captured. This platform assists domain experts to structure knowledge and helps to improve the transparency of KBE applications. To test the proposed solutions, two experiments have been performed. The first experiment tested whether participants were better prepared to develop application code using the KP, while the second experiment tested whether participants were better able to verify whether a developed application complies to the requirements. Results of the first experiment shows that developers are better prepared to develop application code in a collaborative environment as they developed the application twice as fast and with less errors using the KP compared to without. The results of the second experiments shows that projects leads are twice as more likely to correctly assess whether the KBE application complies to requirements using the KP compared to without, as they have a better understanding of how and where knowledge is implemented in application code.

From this research, it can be concluded that the traceability of knowledge into application code can be established by defining different classes of KFs and their relation to different types of COs. Moreover, the KP successfully implements the TM to explain knowledge embedded in application code and to help participants understand the knowledge captured, this can be concluded based on the results of the experiments. The experiments further show the importance of modelling knowledge and enabling a connection between knowledge and code, as the transparency improves greatly. However, another experiment is needed to test whether knowledge can actually be structured without the necessity of knowledge engineers.

Contents

Nomenclature

AI	Artificial Intelligence
AJAX	Asynchronous JavaScript and XML
AST	Abstract Syntace Tree
CAD	Computer Aided Design
CO	Code Object - An object found in application, automatically parsed from application code by the Knowledge Portal
DPM	Design Process Model - One of MOKA's formal models to capture knowledge about the design process steps and the order of execution
ICARE	Illustration Constraint Activity Rule Entity
JSON	JavaScript Object Notation
KBE	Knowledge Based Engineering
KBS	Knowledge Based System
KF	Knowledge Form - A form used to structure knowledge in such an extent it can be used in a KBE application
KP	Knowledge Portal - The developed platform that assists the stakeholders to perform their task in the development of a KBE application
MOKA	Methodology and software tools Oriented to Knowledge based engineering Applications
OOP	Object Oriented Programming
PM	Product Model - One of MOKA's formal models to capture knowledge about the product
RDF	Resource Description Framework
SME	Small to Medium-sized Enterprises
TM	Traceability Model - The developed ontology which cconnects the Knowledge Ontology to the ParaPy Ontology
UI	User Interface
UML	Unified Modelling Language
UX	User eXperience

List of Figures

List of Tables

1

Introduction

Knowledge Based Engineering (KBE) applications are effective at automating time-consuming and knowledge intensive activities [33] as product and process knowledge is translated into application code. These activities are often repetitive and tedious, such as geometric modelling or pre-processing for analyses. Other advantages that KBE has includes knowledge re-use, no knowledge loss when employees leave the company, a decrease in human errors, and an increase in design space. Due to the reduction in routine design time, it also gives engineers more time to spend on innovative designs. These advantages are achieved by merging Object Oriented Programming (OOP), Artificial Intelligence (AI), and Computer Aided Design (CAD).

During the development of a typical KBE application, five types stakeholders can be determined: domain experts, knowledge engineers, developers, project leads, and end-users. The domain experts possess the knowledge that is used in the KBE application. They are not directly involved in the development of the KBE application, but their input is essential to build a working KBE application. The goal of the knowledge engineer is to translate the knowledge of the domain expert into models that can be understood by developers, who will develop the KBE application. Once a KBE application is developed, the project lead has to make sure that it complies to the requirements. Only then can it be delivered to the customer so that the KBE application can be put to use. The end-user is the stakeholder that uses the KBE application to automate (part of) their design process.

Without the use of a proper knowledge modelling technique during the development of a KBE application, it is perceived as a black box. This is due to the fact that a KBE application is non-transparent: it generates an output from some input, but it is not possible to understand how it came to its conclusion. This is often due to the fact that the knowledge from the domain expert is directly translated to application code, which means that the knowledge is merely represented by context-less data and formulas [40]. Only a developer experienced in using the coding language can attempt to understand how the knowledge is implemented, making it inaccessible for all the other stakeholders. However, KBE application development is often a collaborative development process causing developers to have difficulties understand application code of colleagues even though they can understand the coding language.

The other stakeholders are negatively affected as well. Domain experts have difficulties trusting the KBE application as they cannot understand for themselves how their knowledge is implemented. They also feel excluded from the development process, which decreases their motivation to participate and to help with providing knowledge for the KBE application. A project lead is not able to understand the application code, and has to rely on running the KBE application to test its functionalities. However, the fact that the KBE application gave a seemingly correct output does not mean it did so by deriving the correct facts. This makes it difficult to verify whether an application complies to the requirements. Finally, the end-user has to have blind trust in the KBE application they are using, as they are not able to understand for themselves how the KBE application is working.

There are several KBE methodologies available to support the application development process to improve the transparency. One of the well known methodologies is MOKA [8]. MOKA describes several steps to model knowledge, which has proven to be successful. However, the models created with MOKA are disconnected from application code. This means that it is not possible to understand how the knowledge is implemented in the application, which enforces the black box perception. Therefore,

this thesis focussed on solutions to improve the traceability of knowledge into application code by extending one of the models of MOKA.

A field study was performed to discover how KBE applications are developed in the industry. One of the findings was that a KBE methodology such as MOKA was often not used during the application development process. The main reason seems to be that MOKA assumes that a knowledge engineer is required that will translate the knowledge from a domain expert into models that can be understood by developers. However, results from the field study show that most engineers in a KBE project are both the developer as well as the domain expert. This gives the developer the feeling that the knowledge modelling step is not necessary, as they might as well directly implement their knowledge into application code. In a way this is possible, but as knowledge is not modelled the KBE application becomes non-transparent, causing the negative effects explained above. Moreover, it will not be possible to establish traceability of knowledge into application code if knowledge is not modelled at all.

Therefore, this thesis proposes a new ontology called the Traceability Model (TM) which aims to remove the necessity of knowledge engineers. This is possible as it is assumed that the domain expert is also the developer, meaning that they would understand how to model knowledge for use in a KBE application. The TM defines several classes of knowledge to help guide the structure of knowledge by the domain expert. Moreover, the TM aims to establish traceability of knowledge into application code. This is achieved by defining how classes of knowledge can be connected to pieces of code. Moreover, a platform called the Knowledge Portal (KP) has been developed to provide an environment to structure the knowledge. The KP will also be able to parse application code automatically, to allow a developer to connect the developed code to the knowledge that has been structured.

This thesis report starts off by presenting the reader with the results of the field study which made it possible to formulate the research aim, as discussed in chapter 2. Then, the necessary theoretical background is given in chapter 3. This helps the reader understand the methodology of this thesis, presented in chapter 4. Afterwards, the developed ontology is discussed in chapter 5 as well as the developed platform in chapter 6. Once the solutions have been presented, the experiments are presented in chapter 7, as well as the results and the discussion of the experiments. Finally, the report finished with a conclusion in chapter 8 and presents some recommendations for further work.

2

Problem Identification

KBE applications are perceived as a black box which is mainly caused by neglecting a knowledge modelling step in the application development process. And when knowledge is modelled, there is still the problem that the created models are disconnected from the application code. Knowledge seems to be implemented directly in the application code. This would make it only available for KBE developers that can understand the coding language. However, even for the developers it would be difficult to understand how the application is working as application code is not made to be human readable. This problem is worse for the other stakeholders and hinders adoption of KBE technologies. To assess how the black box is perceived in the industry, a field study has been performed. From this field study it was possible to formulate a research question that helped guide the direction of the thesis.

2.1. Field Study

Before working on a solution to improve the traceability of knowledge into application code, interviews were held with 11 industry members experienced in developing KBE application. The industries were distinctive: e.g. aerospace, oil & gas, logistics, and academia. The aim of the interview was to pinpoint the roots of the problem concerning the lack of traceability of knowledge into application code. It was investigated what their KBE application development process was, and which steps were taken to capture, structure, and formalise knowledge. Knowing which steps are (not) performed could give an indication of what is required before being able to connect knowledge to code. Moreover, the participants were asked which roles were defined in the project, and how many roles a single engineer fulfils. For current KBE methodologies, the assumption is made that the domain expert and the developer are two separate engineers. However, the hypothesis is that modern engineers fulfil both the roles of the domain expert as well as the developer. For more information about the roles in the development of a KBE application, see subsection 3.2.3. The interviews were a combination of structured and unstructured questions, allowing room for the industry member to tell their narrative in their own way. The full set of structured questions can be found in Appendix A.

Roles in KBE Development

It became clear that 81% of the participants mentioned that a single engineer fulfils the role of the domain expert and the developer. This is possible as modern engineers learn basic IT skills as it is becoming part of curricula in most degrees. Most participants had a technical degree instead of a IT related degree, meaning they joined the project as a domain expert and learned to be a KBE developer. This means that an engineer is able to understand how a KBE application must be developed while possessing the required knowledge. This allows the same engineer to both structure knowledge and develop application code while possessing the knowledge themselves. However, it was mentioned that for more complex matters, the assistance of a dedicated domain expert would be needed.

In only two case engineers had a single role during the development. These two cases were also the only participants to mention that there were knowledge engineers available. These two cases were at large aerospace companies, while the rest could be classified as a SME (Small to Medium-sizes Enterprise). This shows that SMEs neither have engineers dedicated to a single role, nor have the re-

sources for a knowledge engineer. Whereas larger companies are able to dedicate whole departments for a single role.

Steps in the KBE Application Development Process

The steps in the KBE application development process were investigated as well. Figure 2.1 shows the steps that became apparent from the interviews, and put into chronological order. The percentage shows how many participants usually perform that step during the application development process. Table 2.1 shows the reasoning why certain steps are not performed (any more), or what problems are experienced during those steps.

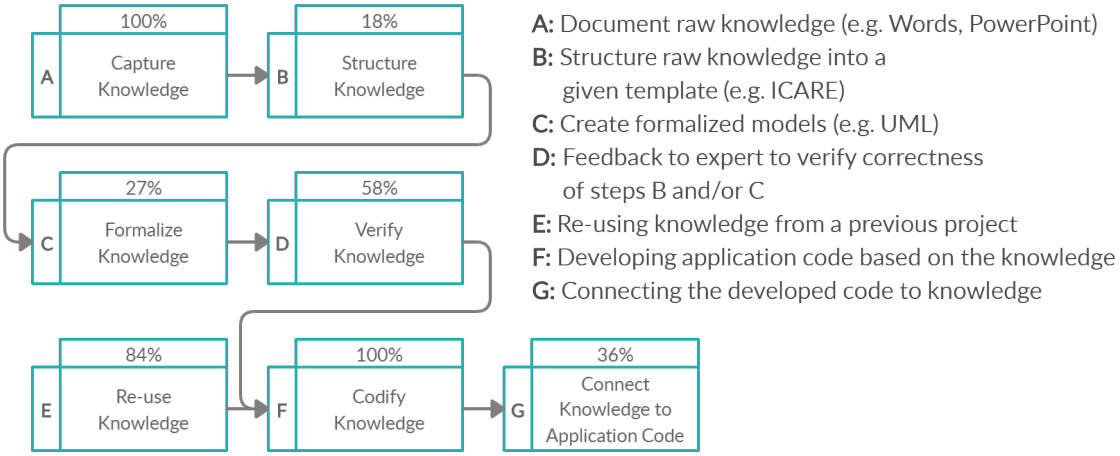


Figure 2.1: The steps in application development that emerged from the interviews, with the percentage of participants mentioning to perform that step.

Table 2.1: List of problems experienced, and the number of participants mentioning the problem during which steps. The numbers represent how many times a participant mentioned the problem regarding that step.

Problem	Step							Total
	A	B	C	D	E	F	G	
Step requires specialist	7	7	4	0	0	0	0	18
No adequate platform	4	4	5	0	0	0	4	17
Results of step are lost	5	5	2	0	0	2	0	14
Results of step not maintained	5	5	0	0	0	0	0	10
Too complex to perform / implement	1	3	1	1	1	0	2	9
Too time-consuming	2	2	3	1	0	0	0	8
Too difficult to find what is wanted	0	2	1	0	0	3	0	6
Step performed inadequately	1	1	0	2	0	0	0	4
No idea how to perform / implement	1	2	0	0	0	0	0	3
Advantages not clear	1	1	0	0	0	0	0	2
Total	27	32	16	4	1	5	6	91

From the interviews it seems that capturing, structuring, and formalising knowledge are the steps where most issues are found. The two biggest issues with these steps is that a specialist is required to perform the step correctly, and there is no platform to support engineers with these steps. This is reflected in [41] where it is mentioned that there is an improvement necessary in the methodological support for KBE. Without sufficient support, it will not be possible to model knowldge correctly and it remains too time-consuming or complex to perform the steps. Furthermore, the participants mentioned that there is a feeling that the results of steps A, B, and C are often lost, forgotten, or are not in sync with application code any more. This further enforced the feeling that the steps are useless and thus are not performed.

The findings further suggest that 36% of the participants have connected knowledge to code. However, the way this is achieved is primitive and is concluded to provide little to no benefits. An example is that application code is sometimes documented within the application code itself, but this does not enable an engineer to find out the source of the knowledge, and it is off limits for someone who cannot understand the coding language. Another example of 'connecting captured knowledge to code' was that a piece of code referred to a certain meeting where it was discussed. However, when looking up the meeting notes, the discussed knowledge could not be found. This circles back to the fact that knowledge has not been structured and formalised correctly, making it difficult to connect captured knowledge to code. It also shows that knowledge is seemingly lost, as the meeting notes could not be found any more. It seems that a lack of a modelling technique to structure and formalise knowledge is the root cause of a lack of traceability of knowledge to application code. As knowledge is still in its raw form, the connection between knowledge and code simply cannot be made.

Usage of KBE Methodologies

The two cases with dedicated departments were the participants that mentioned to have experience with KBE methodologies such as MOKA. It is interesting, as MOKA currently defines a life-cycle where the life-cycle steps target each role separately. Moreover, these were the two participants that mentioned to have knowledge engineers, meaning that they were not hindered by the fact that a KBE methodology requires knowledge engineers.

When the other participants were asked for reasons for not using a KBE methodology, such as MOKA, it was mentioned that the requirement for a knowledge engineer is the main reason. Moreover, it is the developers that have to structure knowledge first before developing application code. There seems to be a lack of motivation to go through two extra steps to structure knowledge instead of directly developing the application code. Skipping the knowledge modelling step seems to work for the developers though; however, not structuring knowledge hinders many aspects of a KBE application that provides benefits: traceability of knowledge into application code, re-use of knowledge, applications become more transparent, and it prevents other problems that will be explained in chapter 3. It is important to help the stakeholders understand the benefits of modelling knowledge.

As an engineer now fulfils the role of a domain expert as well as the developer, it means that knowledge can be structured more formally in one step instead of two steps as proposed by MOKA. This should help remove the extra work load put onto the engineers to model knowledge and should increase motivation. Moreover, having a platform where modelling knowledge can be performed should also increase the motivation to model knowledge. Many participants mentioned that having visual feedback of the structured knowledge would be positive.

2.2. Research Aim

The goal of the thesis is to increase the transparency of KBE applications by focussing on the traceability of knowledge into application code, as there seems to be a disconnectedness between captured knowledge and the application code. The focus is put on Small to Medium-sized Enterprises (SMEs) where engineers fulfil multiple roles during the development of an application. This choice has been made as the adoption of KBE technology at SMEs seems to be the most problematic as the requirement for a specialised knowledge engineer is hindering KBE adoption. It also became clear from the field study that most participants have a similarly structured project as SMEs, where a single engineer fulfils multiple roles: e.g. as a domain expert and as a developer. From the results from the field study, it can be concluded that there is a lack of traceability of knowledge into application code due to three reasons: there is a lack of a knowledge modelling technique, with the consequence that knowledge is not structured or formalised; there is a lack of a platform to structure, formalise, and connect knowledge to code, ensuring that the benefits do not outweigh the costs; and there is a necessity of specialised knowledge engineers, causing the adoption of KBE technologies for SMEs to be too difficult. To tackle these three points, the following three goals have been set:

1. Develop an ontology that defines a classification of knowledge and how it connects to application code
2. Develop a platform to structure & model knowledge, and connect knowledge to application code
3. Remove the necessity of specialised knowledge engineers

To guide the course of the thesis, the following research question has been formulated.

”To what extent can a methodology to structure knowledge improve the traceability of knowledge onto KBE application code without the need of specialised knowledge engineers?”

3

Theoretical Background

This chapter presents the reader with some theoretical background to help understand the choices made during this thesis. It starts with an explanation of what knowledge is, and what types of knowledge exist. Afterwards, there is a section explaining what KBE is, how it uses knowledge, and what the advantages are that it can bring. Next, two modelling techniques are presented, which were used to understand how knowledge can actually be modelled correctly. Then, the black box perception is discussed. This is a problem that is common for applications, and this section expands on the effects it has on the KBE application and its stakeholders. Commonly, the black box perception is the result of a lack of modelling knowledge. Therefore, the final chapter introduces the MOKA methodology, which proposes a few models to structure knowledge for use in a KBE application that have proven to be effective.

3.1. Engineering Knowledge

Knowledge plays an important role in KBE applications, as engineers developing a KBE application translate engineering knowledge into application code. This engineering knowledge is implemented in the application code as engineering rules and they control the flow of the application to come to a design. Therefore it is important to define the concept of knowledge and the different types of knowledge that exist in general, this is explained in the first section. Knowing what knowledge is, it is possible to classify the different types of engineering knowledge that can be found specifically in a KBE application.

3.1.1. Concept of Knowledge

According to Milton [29] there are many ways to explain the concept of knowledge: a highly structured form of information, what is needed to think like an expert, what separates non-experts from experts, or what is required to perform complex tasks. Milton suggests that ”knowledge is a machine or an engine in our heads”. Knowledge is often interchanged with information, however this is not correct as information is a building block of knowledge as it adds to or changes knowledge by means of a flow of messages or meanings [32]. According to Boateng [4], knowledge can be seen as ”the processing of information” and as ”a skill based on previous understanding, procedures, and experience”. Information itself can be seen as data that is ”in formation”, meaning data that has been processed. This gives three layers of how information can be perceived, and Figure 3.1 summarises nicely how these layers compare to each other. The pyramid shows that going from a lower to a higher level shifts the focus from content to context, and the volume decreases. Human involvement is what makes information shift up a layer, and not only increases the value of it but makes it harder to manage and to transfer the knowledge (Davenport, cited in [4]).

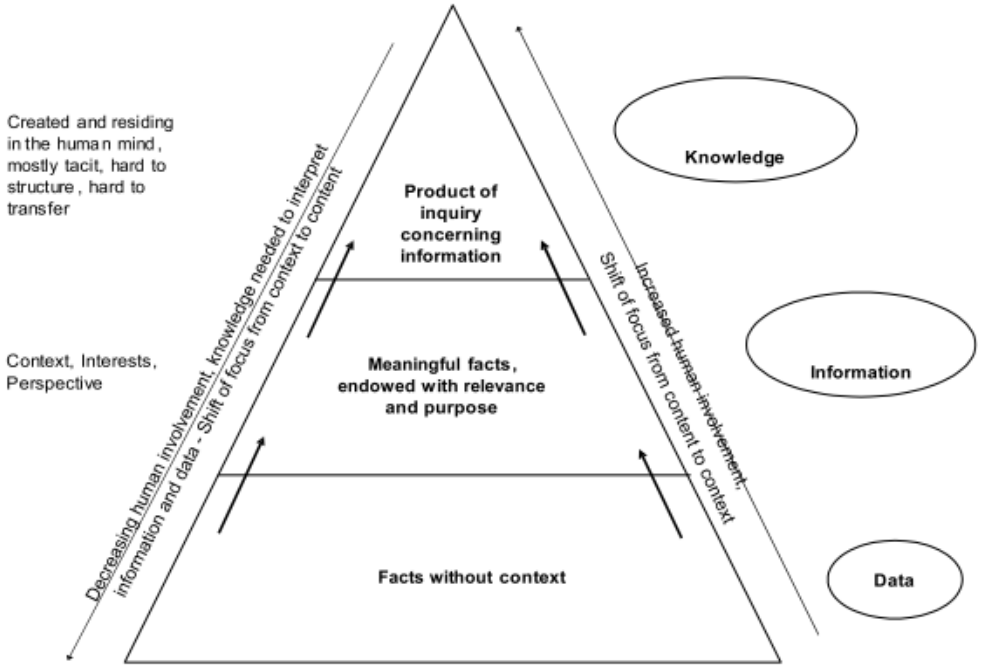


Figure 3.1: The information pyramid [4].

3.1.2. Classification of Engineering Knowledge

Knowledge itself can be categorised to help understand it better [29]: procedural vs conceptual knowledge, and explicit vs tacit knowledge. The differences between procedural and conceptual is that the former is knowledge that is about processes, tasks, and activities, while the latter is about how things are related to each other and about their properties. Shadbolt [35] explains that it is a difference between 'knowing that' and 'knowing how'. The differences between explicit and tacit is that the former concerns basic tasks, relations, and properties of concepts which are easy to explain; while tacit knowledge is rooted deep in the brain and is forged through experience, which can be expressed as gut feelings and hunches which are hard to explain. Figure 3.2 summarises in a simple way the differences in types of knowledge.

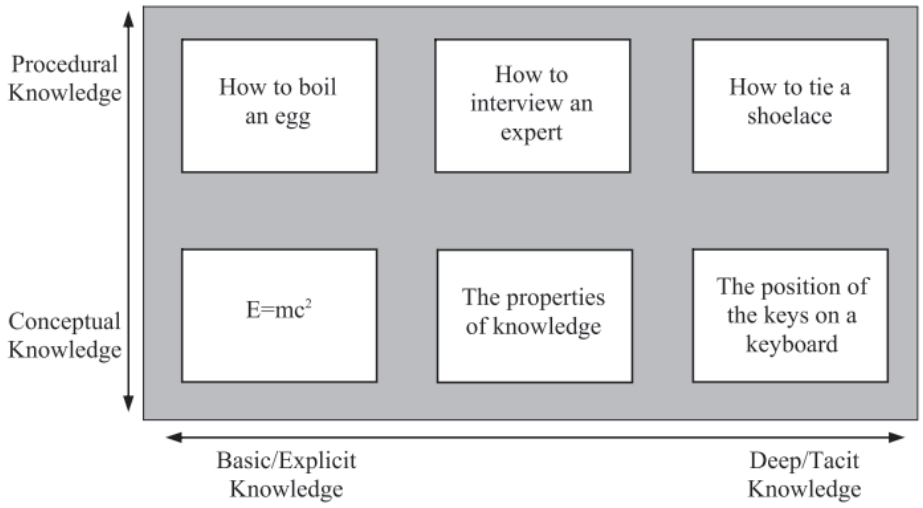


Figure 3.2: Examples of different types of knowledge using a 2-dimensional graph [29]

Understanding the different types of knowledge is important as it helps categorise it and relate it with one another. It also helps to create understanding of the various types of knowledge existing in the domain of KBE applications. Furthermore, it helps to develop methods to capture every type of knowledge. For example, structuring a rule which is a formula, e.g. $E = mc^2$, can be done easily using the variables of the formula as it is explicit knowledge. However, to structure the rules of how to tie shoelaces needs a more complex capture procedure, as it is tacit knowledge. Boateng [4] explains that explicit knowledge can be codified and may exist in forms such as rules, procedures, and theories. In contrast, tacit knowledge may not be transcribed into a rule, possibly limiting the accuracy of engineering rules. However, explicit and tacit knowledge are not inseparable, as Ibrahim mentions [19]: ”They are not two ends of a continuum, but two sides of the same coin”. This means that every piece of knowledge has a tacit part to it. This tacit dimension cannot be codified, however, an attempt can be made to articulate tacit knowledge into explicit knowledge.

It is important to distinguish tacit knowledge from explicit knowledge to be able to model it correctly. Understanding the difference between procedural and conceptual knowledge should also be taken into account when modelling knowledge, as one deals with processes while the other deals with relations and properties. These distinctions should be represented in the ontology that is being developed.

Knowing the four extremes of knowledge, it is possible to take a look at how this applies for KBE applications and what type of knowledge can be found in KBE applications in the form of engineering rules. Boley et al. [5] explains that the classification of engineering rules could be done by its form or its purpose. They also explain that the term rule is an umbrella for a number of related, but quite different concepts. Rules may specify ”constraints, (implicit) construction of new data, data transformations, updates on data or, more general, event-driven actions”. Hu et al. [18] show that engineering rules can be classified as three types: deductive rules (or derivation rules), normative rules (or integrity rules), and reactive rules (or active rules).

Deductive rules can be used to trigger a forward or backwards reasoning engine to derive implicit facts, or as Boley et al. [5] describes: they are rules that derive knowledge from other knowledge with the use of logical inference. Normative rules can be seen as constraints, which pose restrictions on objects. Deductive and normative rules are both conceptual knowledge, and can both either be explicit or tacit. Reactive rules are needed to be able to update the KBE application during runtime, as the others cannot take action. Reactive rules are therefore procedural, as they contain knowledge about processes and activities.

Lemmens [26] discuss five rule classification schemes found in the business rule management discipline, and compare them to find similarities and differences. In this discipline, a business rule is similar to an engineering rule from the KBE discipline. The five different classification schemes all had different types of business rules; however, they could all fit into three larger categories: constraints, derivation rules, and process rules. This is similar to what was explained above but with a different naming, so it seems there are three categories of engineering rules:

- Constraints
- Derivation Rules
- Process Rules

3.2. Introduction to Knowledge Based Engineering

There are a lot of definitions for Knowledge Based Engineering that can be found in literature. La Rocca [23] mentions that the different definitions found can be explained by the fact that there are different views on KBE due to different ”KBE customers”, this makes it difficult to fit KBE into a single description. However, La Rocca describes KBE as:

> ”Knowledge based engineering (KBE) is a *technology* based on *dedicated software tools* called KBE systems, that are able to *capture* and *reuse product and process* engineering knowledge” [23]

According to Stjepandic [38], KBE is one of the most promising research and applications fields in the context of knowledge capture, structure, and re-use. The main objective of KBE is ”to reduce the time and cost of product development”, mainly by automating repetitive, non creative design tasks [9]. This is illustrated in Figure 3.3.

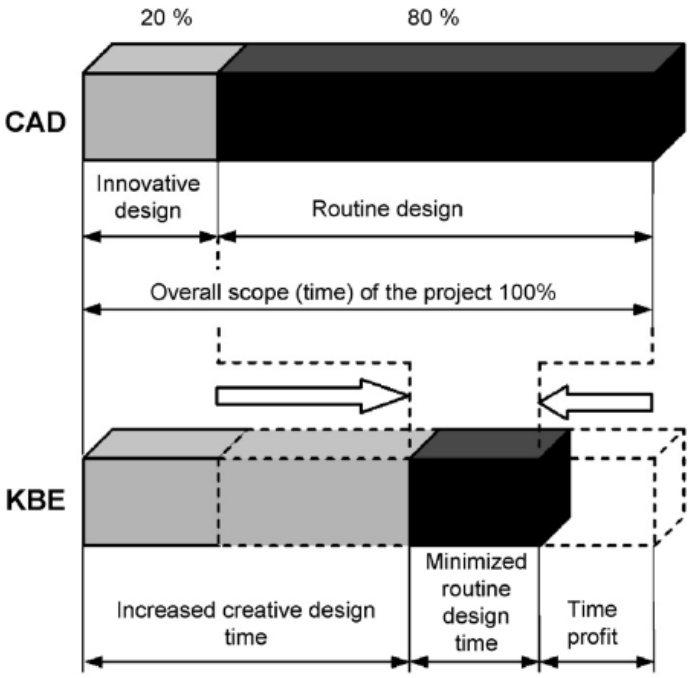


Figure 3.3: Influence of KBE usage on time of main design tasks [36].

Table 3.1 shows three examples of KBE projects where there was a significant reduce in lead time, adapted from [33].

Table 3.1: Examples of projects where the lead-time was reduced using KBE technologies, adapted from Reddy [33].

Reference	Focus Area	Achievement
Emberey et al. [15]	Engineering Design Application Development in Aerospace	With respect to traditional design process lead-time reduction of 75%
Jodin et al. [21]	British Aerospace Wingbox design	8000 hours to 10 hours
Jodin et al. [21]	Jaguar car inner bonnet design	8 weeks to 20 minutes

3.2.1. Advantages of adopting KBE technology

As the lead time of a project is reduced, there is more time to explore a larger part of the design envelope [41] [10] [28]. Moreover, it gives the companies using KBE systems the chance of mass customisation needed as the market may shift rapidly [33]. After the design development process, the knowledge may be re-used for the next project [41] [10], this is partly due to the fact that the code is built modularly which allows the programmers to re-use modules of codes for the new application. Figure 3.4 shows how KBE can improve on the design development process, affecting the design freedom, committed costs, available knowledge, and incurred costs.

The benefits of KBE is achieved by merging of Object Oriented Programming (OOP), Artificial Intelligence (AI), and Computer Aided Design (CAD) technologies [7]. KBE is an extension of Knowledge Based Systems (KBS) into the design engineering domain /citeMa2008, where a KBS is ”any system which performs a task by applying rules of thumb to a symbolic representation of knowledge, instead of employing more algorithmic or statistical methods” [20]. KBE mainly adds facilities for geometry manipulation [38] to the existing KBSs [38]. An often brought up misunderstanding is that KBE is an alternative for CAD; however, it does not remove the need for CAD but reduces the number of CAD stations needed for a particular task [10].

3.2.2. KBE Platforms

There have been several KBE platforms throughout the years, and Figure 3.5 shows which platforms appeared, merged, and disappeared. Each platform uses different programming languages, but what

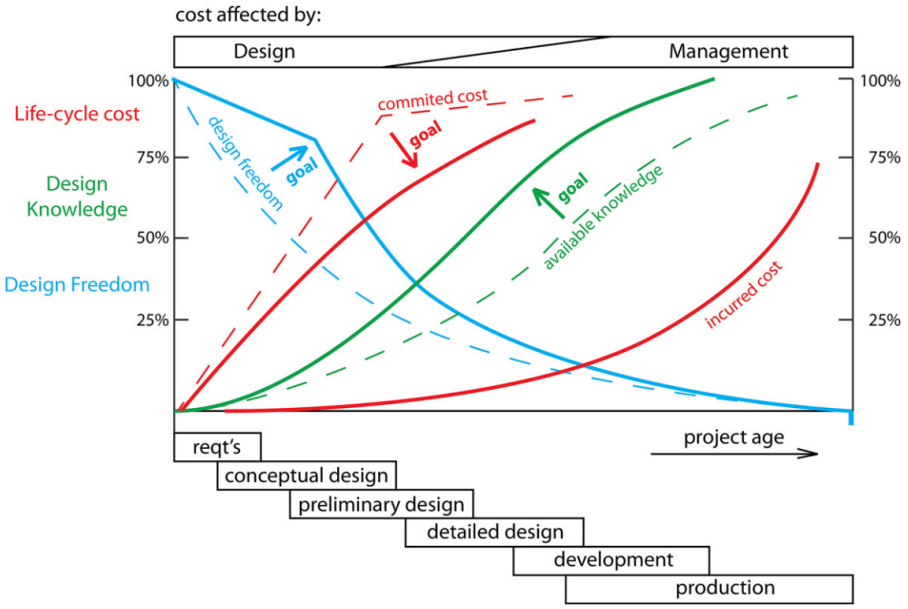


Figure 3.4: How KBE can improve on the design development process [41]

they have in common is that they are built using an Object Oriented Programming (OOP) language, which allows the application to be built modularly [10]. However, the language used is either a difficult language to learn or unique for that platform; this is one of the drawbacks of KBE. A breakthrough for KBE applications is brought by ParaPy which uses Python as the programming language which makes the platform more attractive [1].

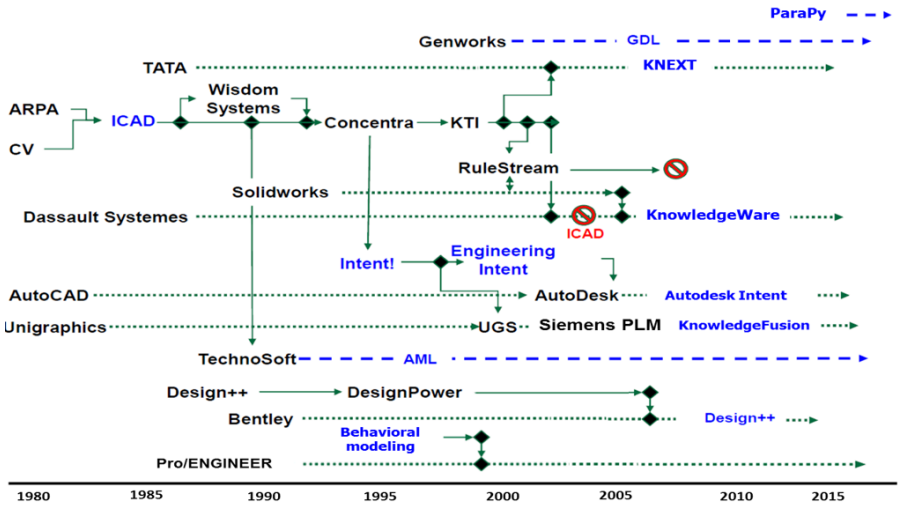


Figure 3.5: A genealogy of KBE systems [37]

A KBE platform uses the code that has been programmed to automatically generate geometry, which is different than generally done with CAD systems, where the geometry is generated by hand. This is done by the use of engineering rules [10] which helps to reduce the amount of human involvement, one of the objectives of KBE systems [28]. As KBE applications use rules for design and reasoning, it can be classified as a rule-based system. The following figure shows how a column is generated using the KBE platform ParaPy by defining the radius, height, and width in the code.

Another advantage of KBE platforms is that it works with lazy evaluation. By utilising backward chaining, only the necessary rules are fired to get to the solution that is desired. Most coding languages

[1] www.parapy.nl

```
73          @Part
74          def inner_column(self):
75              return Cylinder(radius=self.radius,
76                              height=self.h_step * self.n_step,
77                              color="black")
```

Figure 3.6: An engineering rule which generates a cylindrical geometry when fired, using four inputs

make use of forward chaining, which fires the rules depending on which one is first. For a complex KBE application this would take a long time and it is inefficient, especially if only one functionality of the application is wanted. This can be, for example, if a user would like to do a FEM analysis of a model. Instead of running the whole model and all other functionalities (e.g. CFD), lazy evaluation only fires the rules necessary for a FEM analysis. Lazy evaluation is illustrated in Figure 3.7.

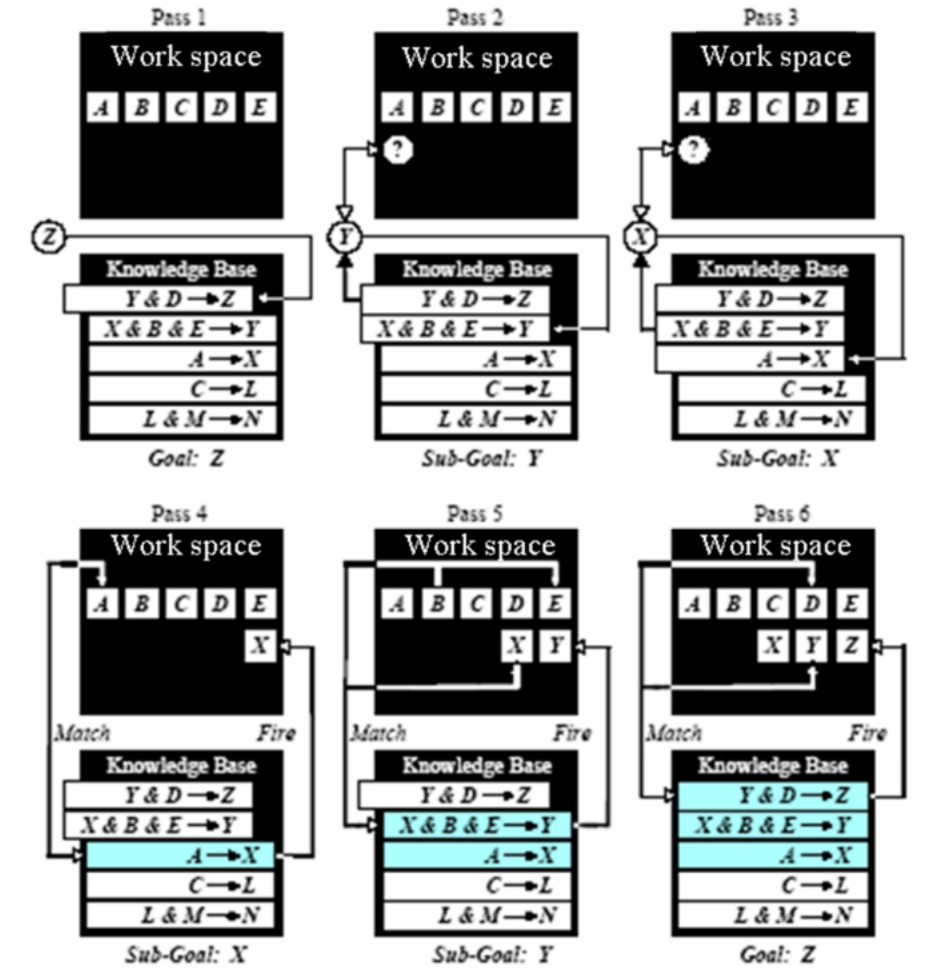


Figure 3.7: Example of the lazy evaluation inference mechanism [31]

3.2.3. Stakeholders in the Development of a KBE Application

There are several stakeholders in the development of a KBE application, these can be stakeholders that: help to develop an application, use the application itself, or oversee the development of the application. The stakeholders in a KBE development process can be seen in Figure 3.8.

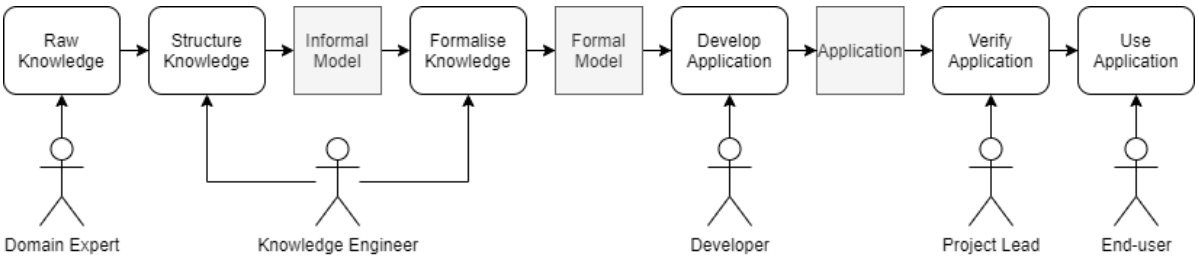


Figure 3.8: The steps to get from raw knowledge to a working application, and which stakeholders are performing that step.

The differences between raw knowledge, informal models, formal models, and the application can be seen in Figure 3.9. In this figure, raw knowledge is shown as plain text which could be a report given by a domain expert. The knowledge engineer first structures this into several objects that capture essential details about the application domain. An example is an ICARE form from MOKA as seen in Figure 3.21, which is discussed in section 3.5. By relating these forms with each other, an informal model can be created as seen in Figure 3.9. The informal model must still be understandable by a domain expert, so that the knowledge engineer can review the informal model together with the domain expert. Afterwards, the informal model is used as a starting point to create a formal model that better structures knowledge to be used in a KBE application. In this example, this is represented by a UML Class diagram, which is explained in subsection 3.3.1. This is understandable by developers, who then are able to develop application code, which is the final step that knowledge takes to be implemented in the application code.

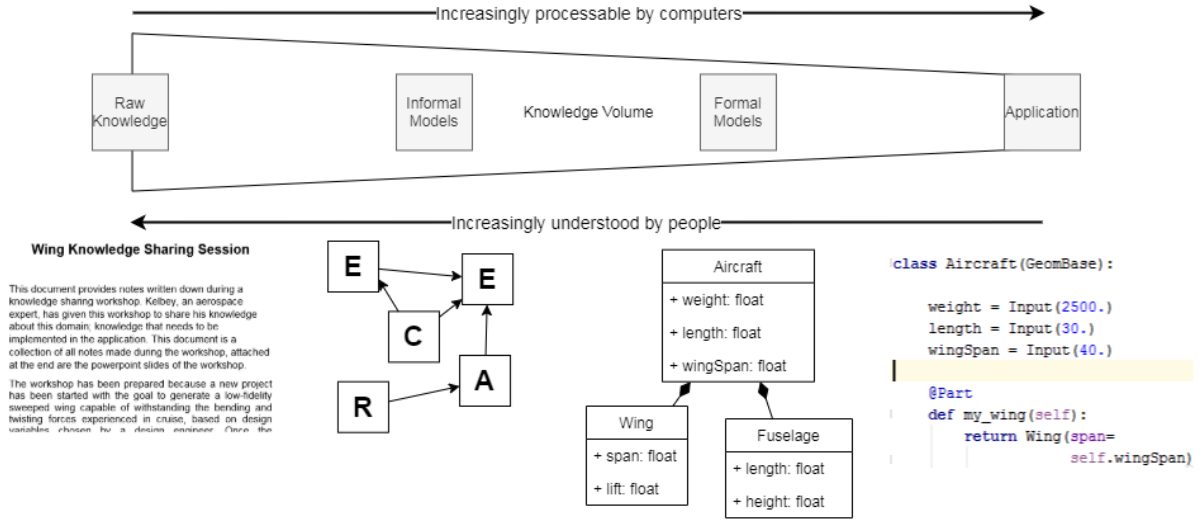


Figure 3.9: From raw knowledge to application code, the volume of the knowledge decreases and it becomes more processable by a computer, adapted from MOKA [8]. However, it becomes less understandable by people.

Domain expert is the role of an engineer specialised in a certain domain. These are typically engineers from an engineering background that are involved in design. Their input is vital for KBE applications, as they possess the knowledge required to develop a KBE application. Often, the time of a domain expert is valuable, so it is important that a domain expert wants to participate in the development of a KBE application by giving their knowledge. A domain expert does not understand how a KBE application works as they have no experience in the development of a KBE application.

Knowledge engineer is the role of an engineer that acquires knowledge and structures it so that it can be used for a KBE application. These are usually engineers with a computer science or engineering background. Often, the knowledge engineer works together with a domain expert to transfer their knowledge in a consistent and unambiguous format. This is done in two steps. The first is to process the knowledge informally using forms as seen in Figure 3.21. This form is an ICARE form from the MOKA methodology which is explained in section 3.5. By filling these forms and relating them

with each other, an informal model can be created. This informal model is understandable by domain experts, so that it can be reviewed by them.

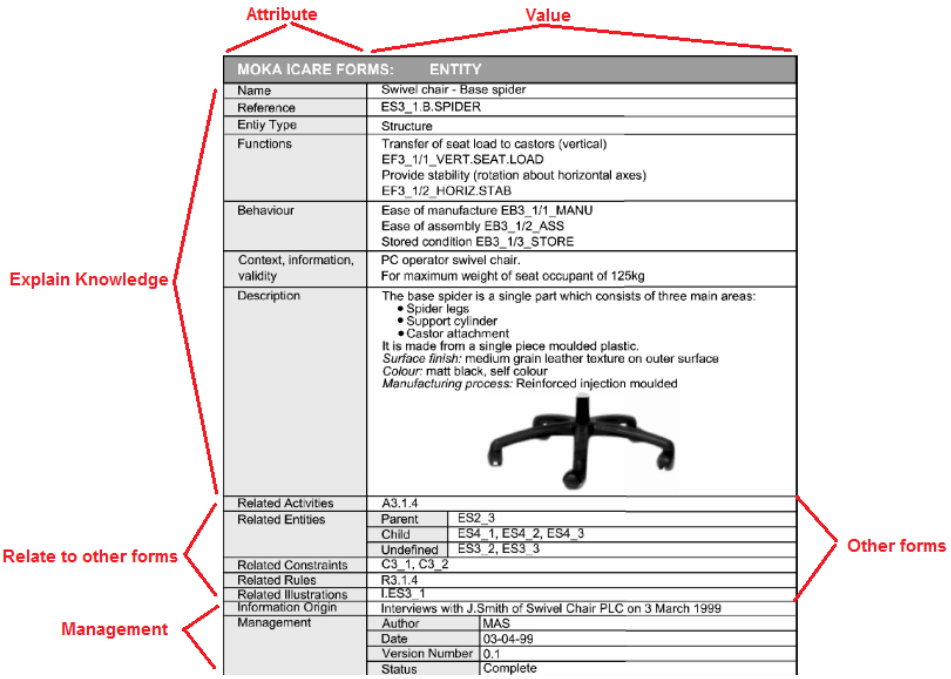


Figure 3.10: An example of an ICARE form, from [6]. A form has several attributes that have to be filled in to describe the knowledge being captured. Then, this form can be related to other forms which allow an informal model to be created showing the relations between all the forms.

Once the informal model is complete, the knowledge engineer starts working on more formal models. These models are representations of the captured knowledge that is understandable by developers, to allow them to develop the application code directly. Examples of formal models are presented in section 3.5.

Developer is the role of an engineer that can read formal representations of knowledge and use that to develop a KBE application. UML diagrams are powerful tools that can be read by a developer to help him understand the structure of a KBE application, but also the connection between elements, the order of execution, and how different objects relate to one another.

Project lead is the role of an engineer that oversees the development of the KBE application. The project lead is responsible for the correct development of the KBE application and makes sure that it complies to requirements.

End-user is the engineer that uses the KBE application once it has been developed. These are often engineers with a technical background that work on engineering designs. The KBE application will automate (part of) their design tasks.

3.2.4. Roles in a SME environment

Large corporations that have been using KBE for some time have engineers play a single role. From the field study (discussed in section 2.1), it was found that at one of the leading aerospace companies, there are large departments for a single role. For the role of domain experts, there are four departments with each a different 'level' of domain experts where the highest level has the most experience. However, Small and Medium-sized Enterprises (SME) are starting to adopt KBE technologies and they do not have the resources of dedicating whole departments for a single role. In fact, a single engineer is assigned to several roles. In this case, an engineer could be the one structuring the knowledge and also developing the KBE application code. This new development is also made possible due to the fact that programming is now a basis skill taught in most curricula. This means that a modern engineer that

has a technical background has some IT experience, and will be able to understand the OOP approach used in KBE applications. What is seen, is that a domain expert is also the developer in these SMEs.

3.3. Modelling Knowledge

Knowledge plays a big role in KBE applications, and KBE has a tight interaction with Knowledge Engineering and Knowledge Management as it can be said they intersect, complement, and specialise each other [25], as can be seen in Figure 3.11.

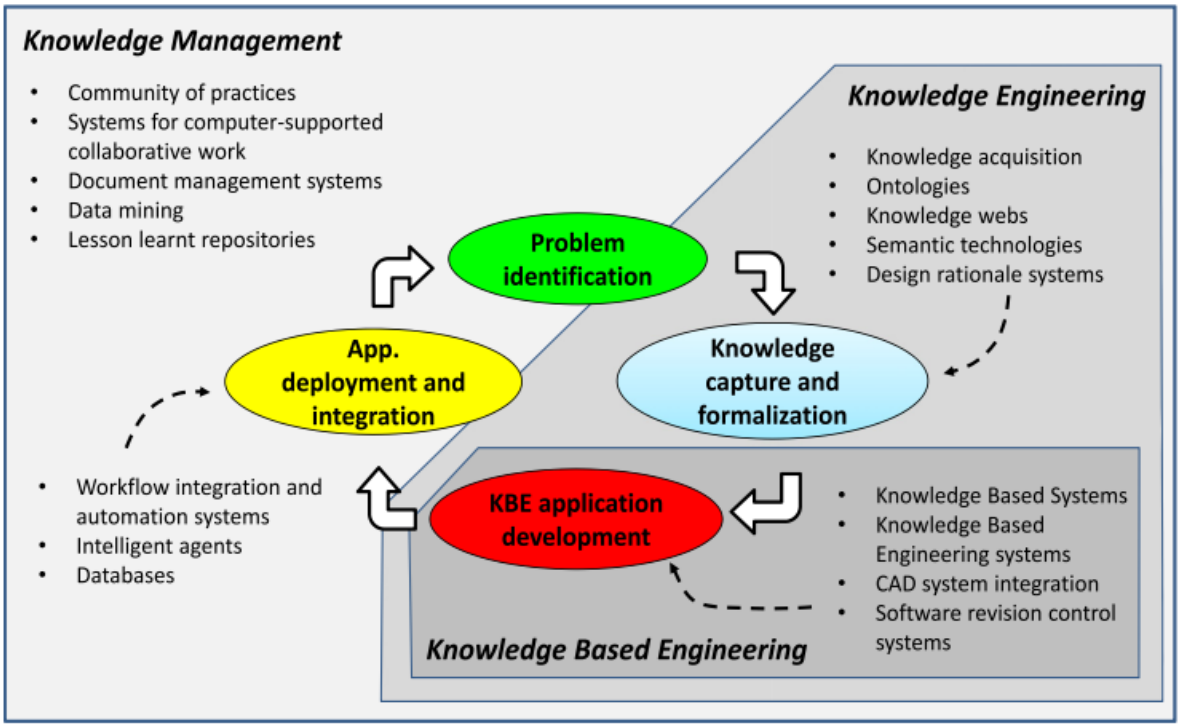


Figure 3.11: Relative positioning of KBE, Knowledge Engineering and Knowledge Management. Lists of knowledge technologies involved in the various phases of the development process of a KBE application for engineering design [25]

It shows that being able to manage knowledge is important for KBE applications, and being able to model it correctly plays a just as important role. Modelling knowledge is an aspect that is tied to the transparency of applications and also to the traceability of knowledge into application code. This is because the knowledge first has to be modelled before it can be codified into a KBE application. This is a part of Knowledge Engineering where knowledge is acquired using ontologies, just as MOKA uses the MOKA Modelling Language (MML). MML which is an extension of the Unified Modelling Language (UML) and enables to model formal representations of knowledge.

3.3.1. Unified Modelling Language

One of the well known modelling languages is the Unified Modelling Language (UML) [2]. This language aims at helping stakeholders raise the level of abstraction of their application. This can be done as UML has thirteen types of diagrams, see Figure 3.12, which can either represent the static application structure, the general behaviour, or different aspects of interactions.

[2] www.uml.org

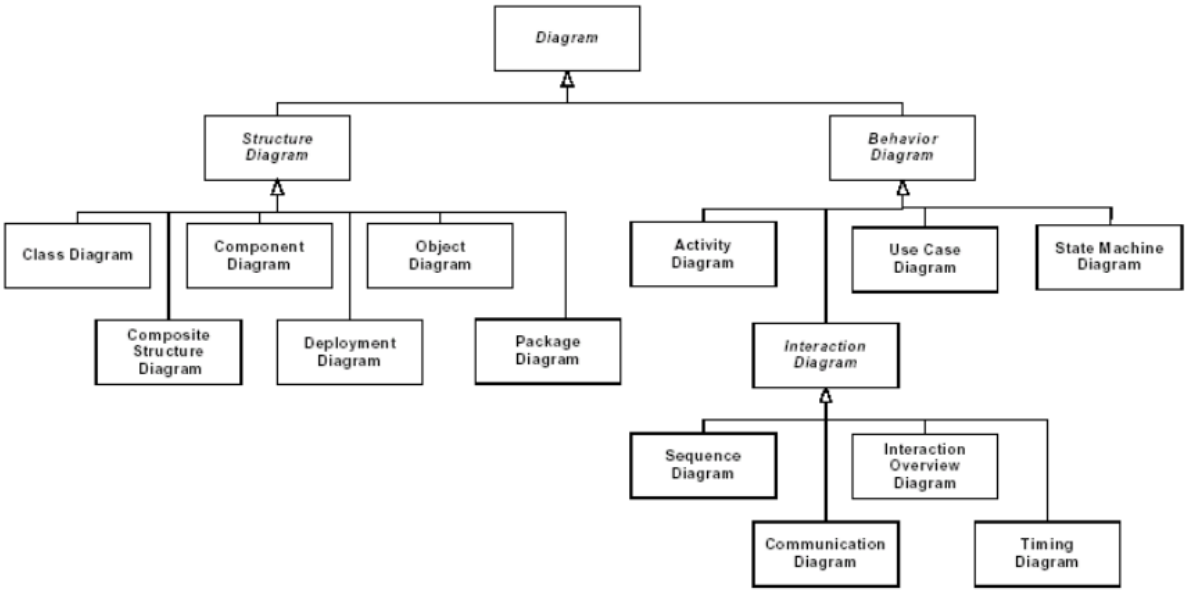


Figure 3.12: Hierarchy of UML diagrams [39]

The power of UML is that it can provide different perspectives of the domain in a human-readable way, while allowing to store the perspectives in a machine-readable way. It is also possible to automate the generation of diagrams from the source code. DeWitte [13] has developed a tool where a user can create a UML Class diagram and check for inconsistencies between the diagram and the application code. From this UML Class diagram it was also possible to generate code skeleton.

The Class diagram of UML can be used to capture how different classes (or types) of objects relate to one another. The Class diagram can be used to provide a clear overview of different types of knowledge, and how they relate to each other. To fully appreciate the Class diagram, it is important to understand what each arrow means, Figure 3.13 shows the basic relations of the UML Class diagram.

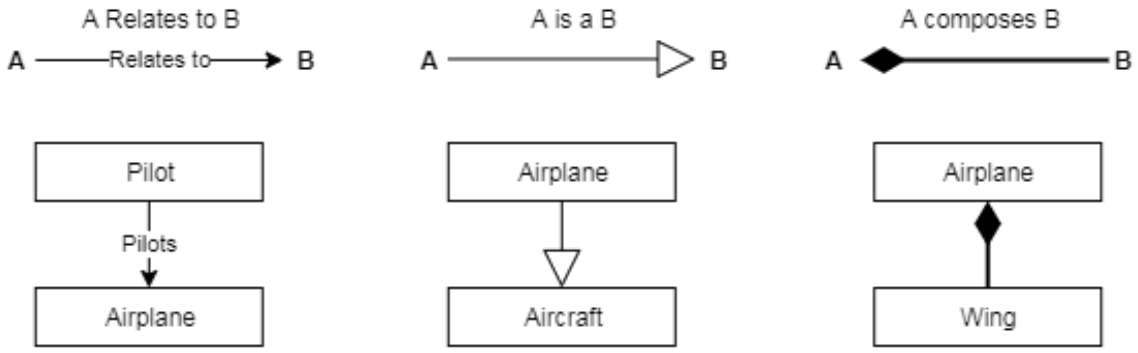


Figure 3.13: Basic relations of the UML Class diagram

Another important diagram is the Object diagram, which is an instantiation of a Class diagram. The Class diagram gives a blueprint to generate an Object diagram as seen in Figure 3.14. Both the Class diagram and Object diagram can be used to model knowledge for a KBE application. The Class diagram defines the possible type of objects and the relations between these types. The Object diagram is an instantiation of that Class diagram and explains how knowledge has been modelled.

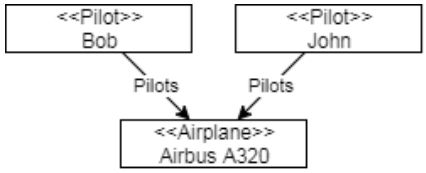


Figure 3.14: Example of an Object diagram, which is an instance of a Class diagram

Extending UML

UML is embedded within the four-layer modelling framework, as can be seen in Figure 3.15. UML itself (the definition of the UML language) can be said to be an instance of the Meta Object Family (MOF), and is referred to as the meta-model [22]. Knowing the meta-model, it is possible to create instances which become UML models. The models themselves are a structured representation of user data. Knowing these four layers, it is possible to add new graphical elements and terms to create domain specific profiles [39]. There are two mechanisms to expanding UML: lightweight (profile extensions) and heavyweight (meta-model extensions) [1]. The profile extension makes it easier to introduce new concepts and allows more to be created using UML for a specific domain. The meta-model extension can add new meta-classes and meta-constructors to the meta-model through the MOF.

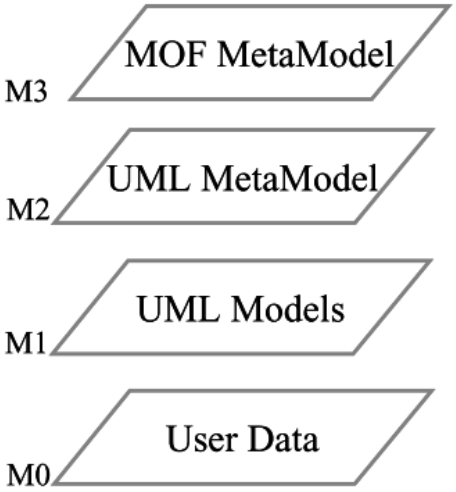


Figure 3.15: Hierarchy of UML models [22]

MOKA has expanded on UML on the profile level, meaning it was a lightweight extension. MOKA added new classes to be able to capture domain specific Product Model and the Design Process Models, by adding stereotypes such as 'Parts', 'Assemblies' for the Product Model, or 'Compound Activity' and 'Elementary Activity' for the Design Process Model. It shows that UML is a versatile language to be able to create a custom profile suitable for ones needs.

3.3.2. Ontologies

An ontology is an explicit specification of a conceptualization [17]. Ontologies are often used to represent hierarchy and dependencies within a context space [30], this is done because ontologies support a set of modelling primitives which define individuals, classes, and their relations. According to Dillon et al. [14], ontologies form a more complete representation of concepts and relationships than models, and are state in the art in capturing semantics and formal knowledge [27]. It has also terminology and conceptualisations that is agreed to by the community and has to be used consistently through-out the community. The flagship language for ontologies is OWL (Web Ontology Language) [22].

The standardised language OWL is embedded within the four-layer modelling framework, as shown in Figure 3.16. The figure shows that OWL is built on top of the RDF (Resource Description Framework) Schema which in turn is built on top of RDF. The relationship between the languages is a usage and a

specialisation relationship, it can be called a ”functional hierarchy” [22] as each language has a certain function.

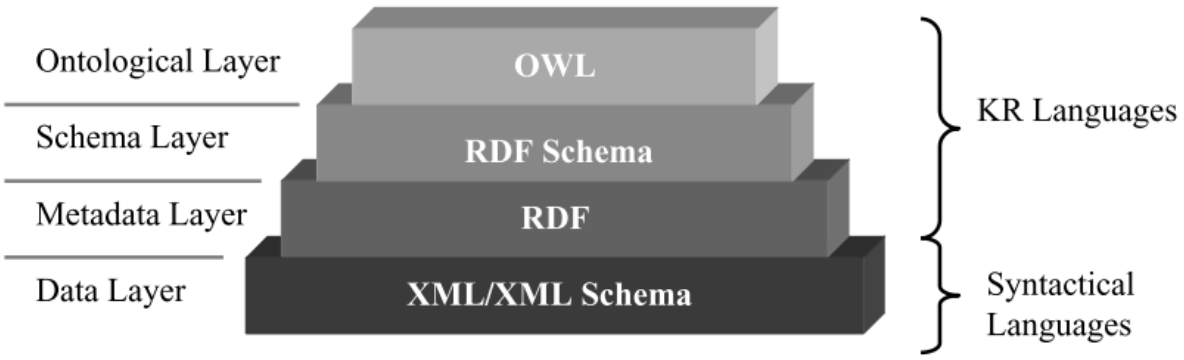


Figure 3.16: Four level modelling framework [22]

The RDF Schema can be seen as the meta-model of the RDF, defining the classes and relations that are available. RDF itself is the collections of triples in the dataset, the two are visualised in Figure 3.17.

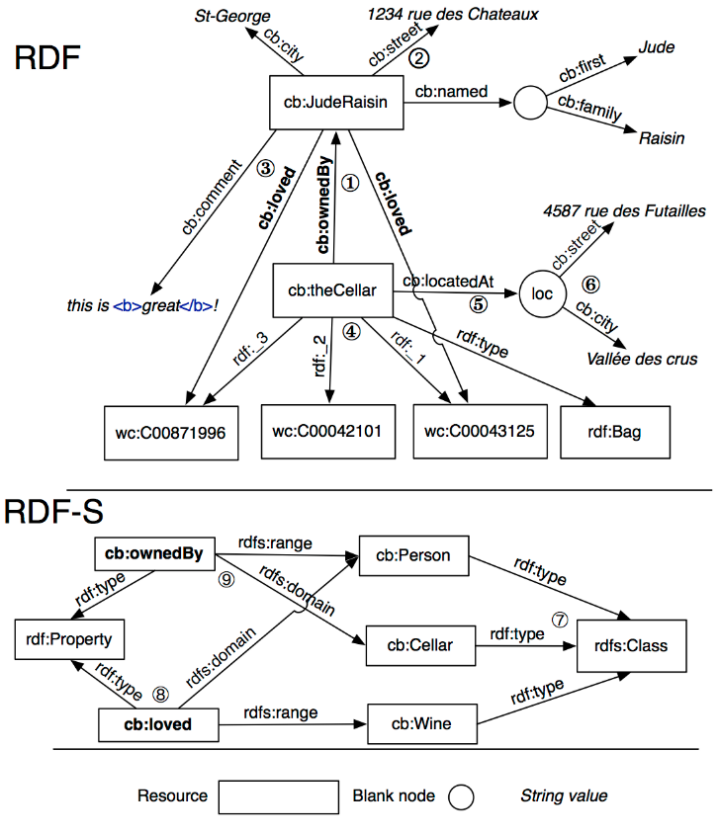


Figure 3.17: The RDF is an instance of the RDF-S (RDF Schema) [24]. RDF-S defines the context as the different objects and the possible relations are set, e.g. that a Cellar can be owned by a Person. RDF uses the RDF-S to determine possible relations, e.g. that theCellar is ownedBy JudeRasin.

In RDF, triples are used to model the knowledge. It defines a binary relation between two objects as can be seen in the Figure 3.18 [24]. Each element in the RDF is identified by an URI (Universal Resource Identifier) which allows to link triples with each other.

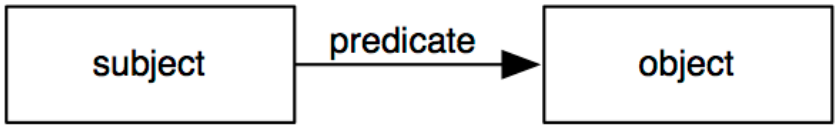


Figure 3.18: Triples in RDF [24]. Used to define how two objects relate to each other.

Having triples makes it easier than models to understand how knowledge is interlinked, while being readable by the computer. OWL is built on top of the RDF Schema as it can use its vocabulary and defined triples to infer connections and relationships that have not been defined manually. This makes it possible to infer new information automatically based on the relations set manually.

With the combination of standardised languages such as OWL it has become rather easy to re-use ontologies, there are also frameworks that provide ready-to-use ontological models. Ontologies are successful due to the formalisation and hierarchisation of the knowledge they provide, however ontologies are far more difficult and time consuming to design and to implement compared to models [30].

3.4. Black Box Perception

KBE is a promising technology to automate engineering design; however, there is still no convincing breakthrough to date apart from major aerospace and automotive companies, as methodologies and technological considerations are still evolving [41]. Verhagen [41] lists several reasons why companies are not adopting KBE technologies, and this thesis focuses on one of the main problems concerning KBE applications: the fact that a KBE application is non-transparent and is perceived as a black box. This is due to the fact that most applications are represented by context-less data and formulas [40]. The black box perception can thus be described as the feeling of unawareness of what is happening within the application. As Reddy [33] describes it: ”it [a KBE application] produces some output with some input, but nobody knows what happens in between”. The main reason for this is that knowledge is not modelled correctly, or not even modelled at all.

3.4.1. Consequences of the Black Box Perception

The lack of transparency has an effect on the whole company, as illustrated in Figure 3.19 [16]. The black box perception decreases the trust in the application [16] as there are groups that have limited to no access inside the KBE application and cannot see for themselves how knowledge is implemented. By not being able to look inside the KBE application, the participation and engagement of domain experts is limited [11], making it difficult to convince these experts to adopt KBE technologies.

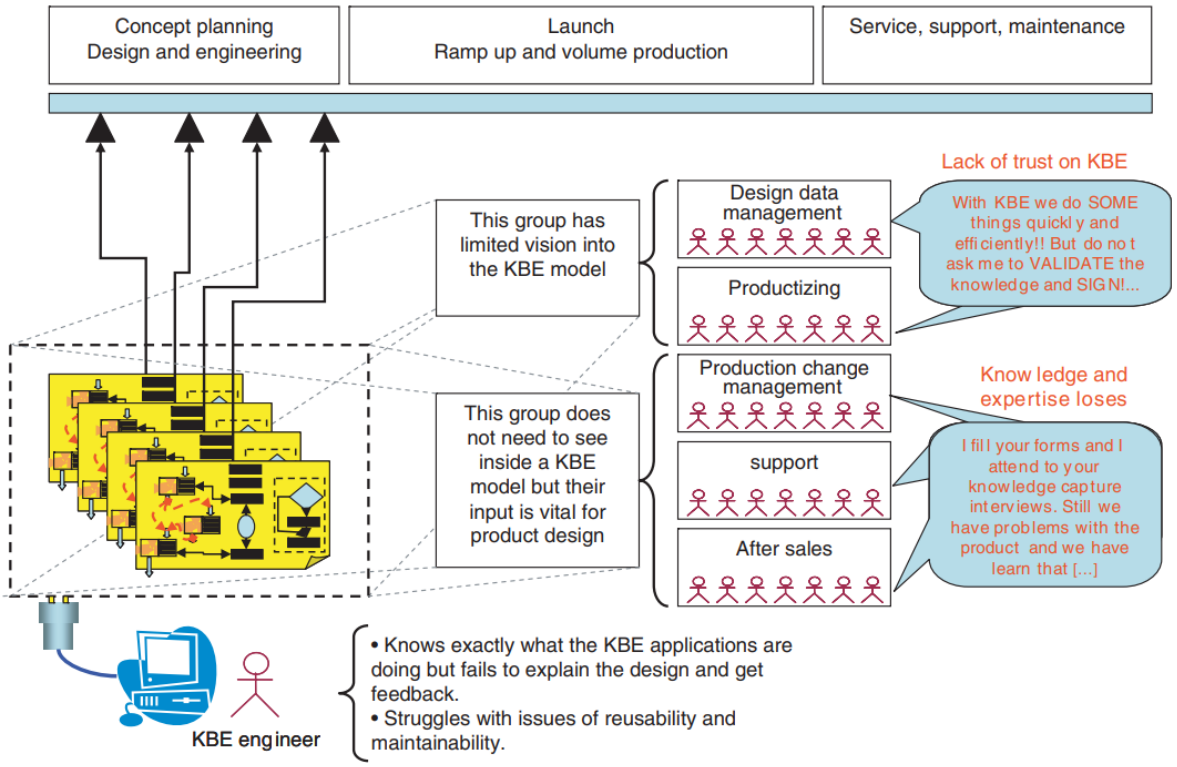


Figure 3.19: The effect of a non-transparent KBE application [16]

Bermell-Garcia recognises that there is a lack of transparency in KBE applications [2], and mentions that it is necessary to improve the traceability of knowledge into application code [3]. With traceability, it is meant the direct connection between captured knowledge and the application code. One of the problems hindering the traceability of knowledge into application is that there is no simple knowledge modelling technique to capture knowledge and structure it accordingly [25]. Verhagen also mentions that there is no appropriate methodology or design process which can be seen as a standard, which means that a KBE application development life-cycle is case based and ad-hoc [41]. This was concluded based on the fact that it was found that 87% of 37 papers did not adhere to a specific methodology. A major reason for not adopting KBE methodologies, is that it requires an engineer to be specialised as a knowledge engineer.

A lack of traceability has more negative effects for KBE applications, one of these is the fact that it hinders systematic re-use of the knowledge embedded in the KBE application as it is not known where and how the knowledge is used in the application [40]. By enabling traceability of knowledge into application code it also solves the problem that users fear that they are not in control [23], as experts and users can understand themselves how knowledge is implemented.

3.4.2. Effect on different Stakeholders

The **domain expert** is expected to populate a knowledge base to develop an application; however, the domain expert is not able to see or understand how knowledge has been implemented in the KBE application. This means that there is a lack of trust in the KBE application [16]. Moreover, the non-transparency of a KBE application gives the domain expert a feel of exclusion, causing them to lack participation in the development of a KBE application which could hinder adoption of KBE technology. This lack of engagement makes it difficult to capture knowledge from domain experts [11].

Moreover, when a domain expert leaves the company the knowledge contained in the expert is lost [4]. If the knowledge that the domain expert had was used in a KBE application, it is of little use if there is a black box perception. This is due to the fact that other domain experts are not able to understand the knowledge that is structured as application code. It is also hard to find out where the knowledge is in the vast amount of application source code.

The **developer** is not able to work efficiently as they will not be able to understand the application code of other developers. This is especially the case in a collaborative environment that is becoming ever more common. A developer will have difficulties understanding the application code of a colleague as most knowledge embedded in application code is represented by context-less data and formulas [40], making it difficult to process. Moreover, the black box perception is often the effect of not modelling knowledge. Without any models, the developer will have difficulties developing the application code. This makes the developer more prone to making errors.

The **project lead** has to verify that the developed application complies to the requirements. Due to the black box perception, the project lead is not able to understand whether knowledge has been implemented correctly or whether the application is computing an output as required. The only way a project lead could verify if the application is doing what is required is by running the application and trying out some values. However, the fact that the application came to the correct conclusion does not mean it did so by deriving the correct facts. It is very hard to validate the application, and that makes it difficult to convince the customer that it does what it should.

The **end-user** is the stakeholder that uses the KBE application to automate (part of) the design process. Due to the black box perception, an end-user might have a lack of trust in the application as they are not able to understand what the application is doing. For the end-user, the application just 'magically' computes an output from some inputs.

3.5. MOKA

There are several KBE methodologies attempting to increase the effectiveness of the application development process, such as CommonKADS [34], KNOMAD [12], and MOKA [8]. KNOMAD extends on MOKA to account for life cycle management issues, and focusses on the multi-disciplinary aspect of KBE applications, thus is not adopted for this research and is not discussed further. CommonKADS focusses on the organisational aspect of KBS application development and how various agents communicate and relate, and its results are not directly used but its influence can be found back in MOKA. MOKA is one of the well-known KBE methodologies which define a life-cycle consisting of six steps as seen in Figure 3.20: identify, justify, capture, formalise, package, and activate.

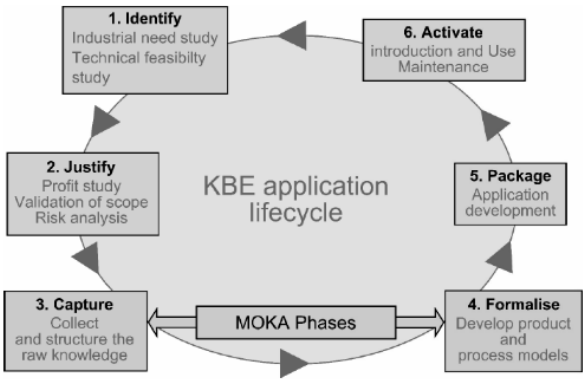


Figure 3.20: The MOKA life-cycle, from [6].

Even though MOKA has defined these six steps, it focuses its efforts on the capture and formalise steps. This is what makes MOKA an interesting methodology to adopt during this thesis. For the capture step, MOKA developed the ICARE model (Illustration, Constraint, Activity, Rule, Entity). With the ICARE model, a knowledge engineer can develop an informal model by producing a set of linked ICARE forms. Illustration forms are used to describe any case studies or relevant examples, Constraint forms are used to describe limitations on Entities, Activity forms are used to describe the elements in the design process, Rule forms are used to regulate Activities and provide the 'know-how', and Entity forms are used to capture the objects that describe the product. The ICARE model has been developed to be the first of two steps to structure knowledge for a KBE application. It was meant to be understood by domain experts, so that they could review the knowledge that has been structured.

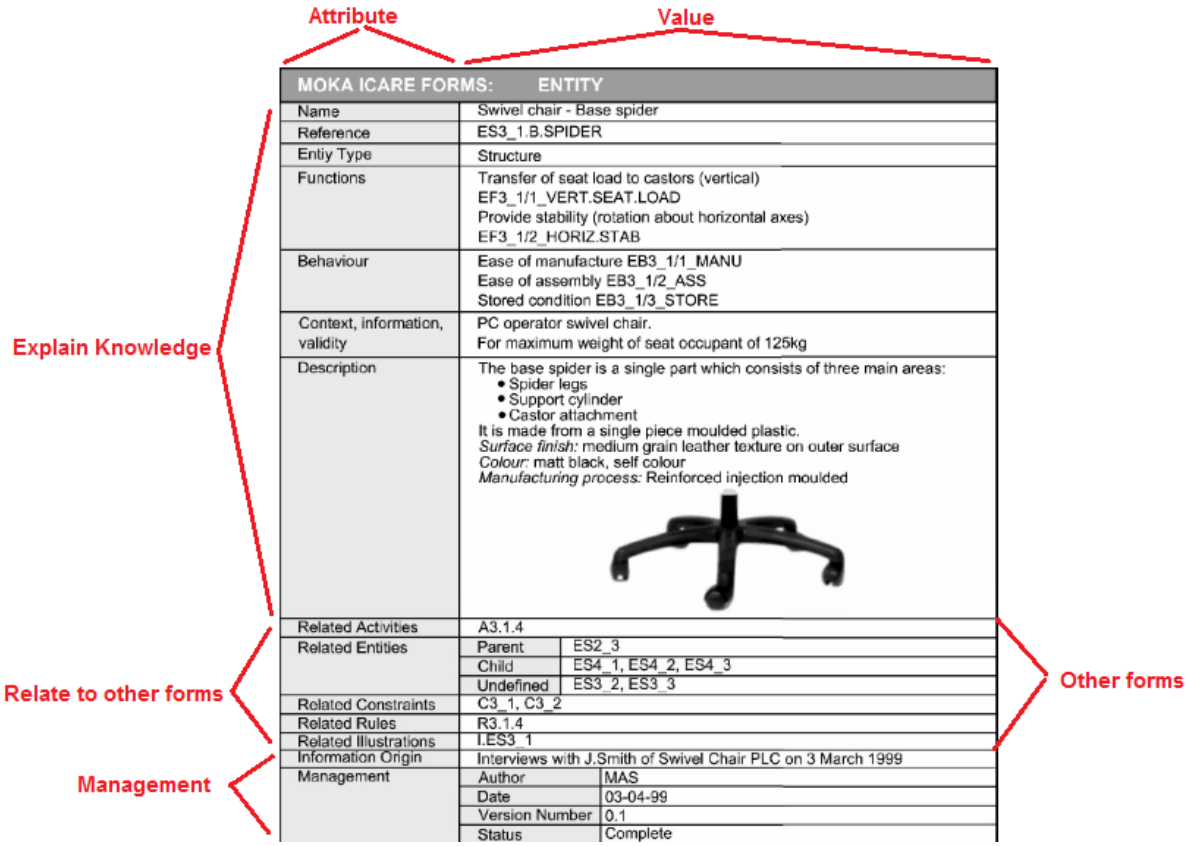


Figure 3.21: An example of an Entity form, adapted from [6].

An example of an Entity form can be seen in Figure 3.21. It can be seen that knowledge can be structured textually to describe its purpose, function, and composition. It can also relate to other ICARE forms, that help describe where it fits in the design flow, how it is affected by rules, and which constraints are acting on it.

The knowledge captured with the ICARE model is not detailed enough to be used in a KBE application, that is why the structure step is followed by a formalise step. Using the ICARE forms, a knowledge engineer can develop two formal models: a Product Model (PM) and a Design Process Model (DPM). These formal models use the informal model which consists of ICARE forms to capture more detail about the knowledge, they can be seen in Figure 3.22. The product model describes how the engineering product is built up from assemblies and parts, the materials used, the way it must behave, and the function is fulfils. It is also possible to capture how it is manufactured and its shape and size. The design process model captures how a design process is taking place, it covers the design flow: the order of activities, iterations, and paths to take. With both these formal models, the knowledge should be detailed enough for the next step called package, where a KBE developer codifies knowledge.

While MOKA has developed three models which can capture knowledge in enough detail to be used in a KBE applications, it still has it drawbacks. One of the drawbacks is that it requires an engineer to be specialised as a knowledge engineer, something not many companies have available. It also takes too much time to first model knowledge informally with the ICARE model, and then further modelling it into the two formal models. As detailed as it is, MOKA does not solve the problem mentioned before of not being a simple knowledge modelling technique [25]. Verhagen [41] further mentions that MOKA does not focus its methodology on the end user. This means that the end-user does not have a clear benefit of using KBE applications, which decreases acceptance, use, and maintenance of a KBE application.

What seems to further lack when looking at MOKA, is that there is no connection between the models and the application code. In other words, there is no traceability of knowledge into application code. As mentioned before, this is an aspect that must be tackled to improve the transparency of KBE application [3].

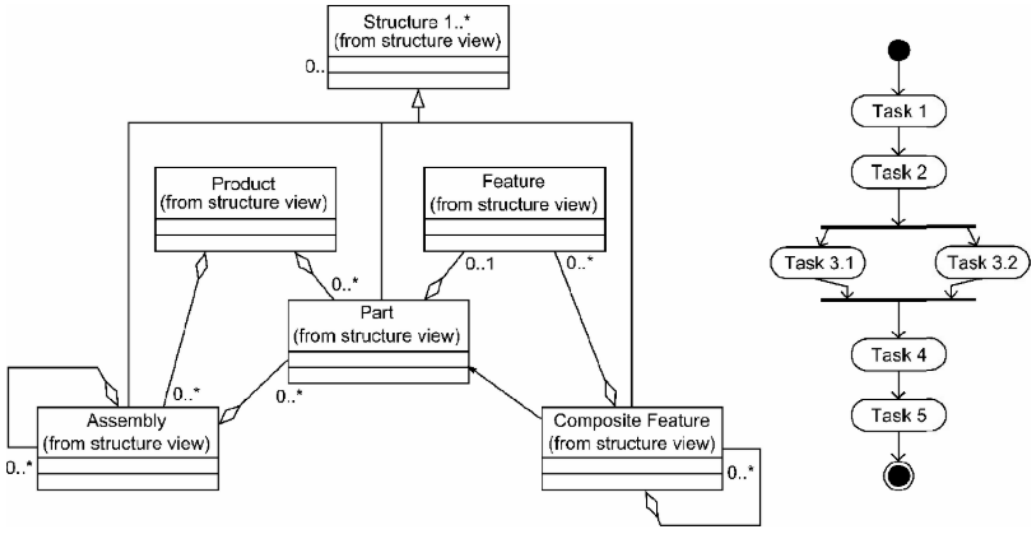


Figure 3.22: Left: Structure view of the Product Model explaining the hierarchy between objects found in the KBE application domain. Right: The Design Process Model, which explains the steps in the application design process. from [6].

4

Methodology

This chapter discusses the steps taken to answer the research question and to meet the goals. First, the Traceability Model (TM) was developed, discussed in section 4.1. Then, the Knowledge Portal (KP) was developed which provided a platform where domain experts can use the TM to structure knowledge, as well as more functionalities as discussed in section 4.2. Afterwards, the reasoning for the experiments are discussed in section 4.3.

4.1. The Traceability Model

The TM is an ontology that provides a middle ground between MOKA's ICARE model and the two formal models (Product Model & Design Process Model), as seen in Figure 4.1. It aims to be simple enough to enable the structure of knowledge without the necessity of knowledge engineers, while being formal enough to be processed by a computer to automate part of creating models.

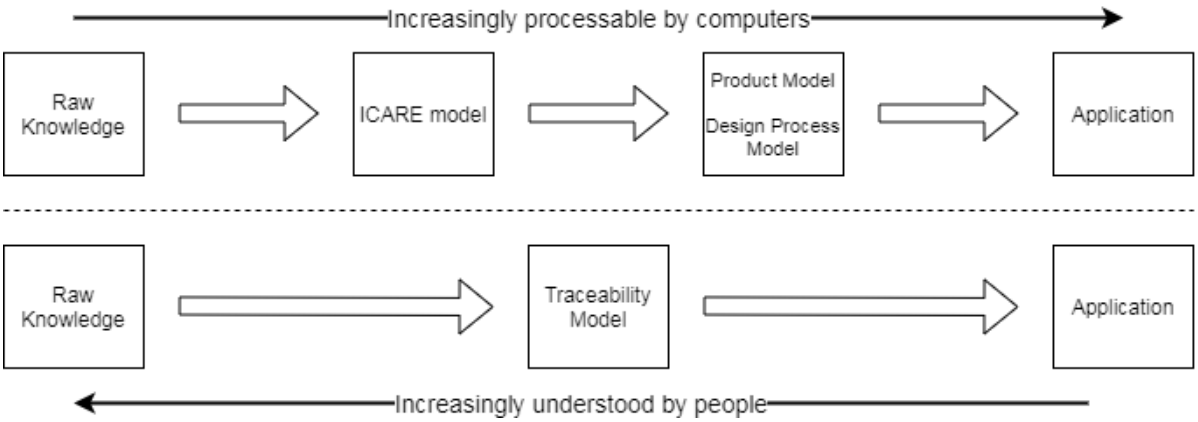


Figure 4.1: The top shows how MOKA defined how to get from raw knowledge to a working KBE application. It requires knowledge to be structured twice: the first with the ICARE model which can be understood by experts, while the second is used to structure knowledge to be developed in a KBE application. The bottom shows that the TM lies in between the two steps defined by MOKA. It increase the formality of the ICARE model whilst still being processable by a computer.

When looking at MOKA, the ICARE model does not structure knowledge detailed enough to allow a connection between knowledge and application code, while the formal models are too detailed and require a specialist. Therefore it has been chosen to increase the formality and expressiveness of the ICARE model by developing the TM. MOKA has defined two steps to structure knowledge as it was assumed that domain experts do not understand how a KBE application should be structured at all. In a SME environment, where engineers fulfil multiple roles, it can be assumed that domain experts do understand the Object Oriented Programming (OOP) approach used for KBE applications.

The TM defines different classes of Knowledge Forms (KF) to be able to capture different aspects of knowledge required for a KBE application: e.g. product hierarchy, processes. An example of a KF

can be seen inFigure 4.2. It adopts the ICARE model from MOKA, as the ICARE model has already classified knowledge into five useful classes: Illustrations, Constraints, Activities, Rules, and Entities. However, the ICARE model has two main deficiencies: it does not capture knowledge detailed enough, and it does not explain how knowledge is related to application code. This was chosen as MOKA has decided to keep the ICARE model informal as it would allow domain experts to understand what has been captured. However, there is a new generation of engineers who can develop some application code and are most likely to fulfil the role of a domain expert as well as a developer. Therefore, the TM added a new class of knowledge called *Property* to the ICARE model and requires the domain expert to capture knowledge about object properties, which is typical for an OO modelling approach. Moreover, the TM revises how the other classes are used to model knowledge and adds the necessary formality.

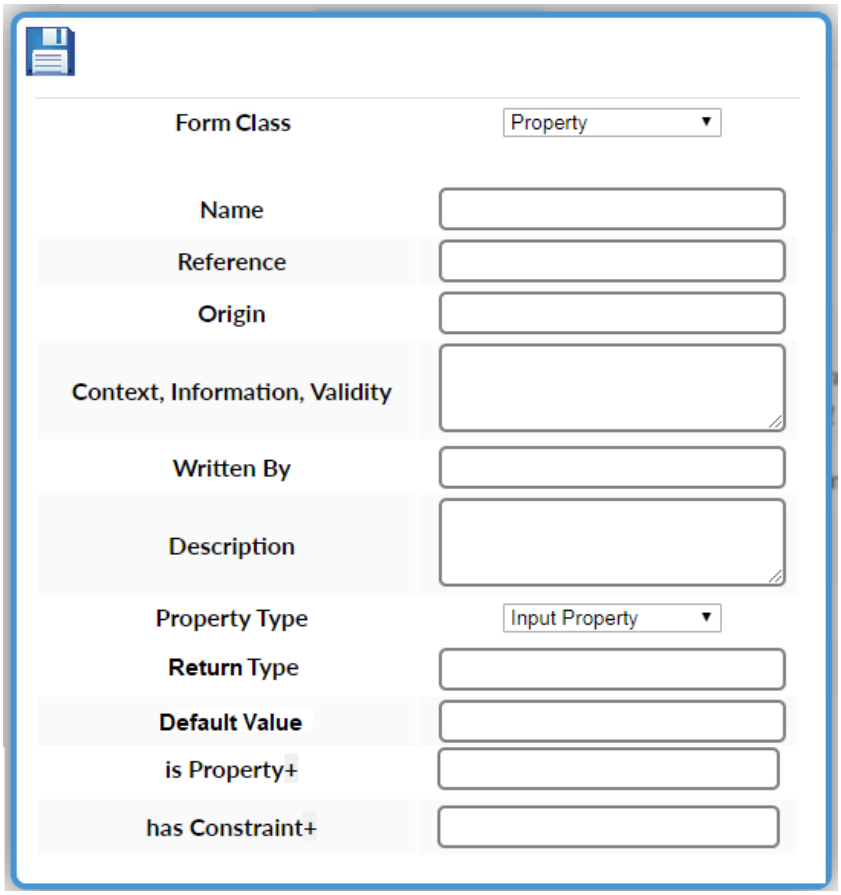


Figure 4.2: An example of a KF, which helps guide a domain expert to capture what is essential for a given class of KF. In this example, this KF is a Property, which is the new class added to what the ICARE model proposed. In the KF, it is also possible to set relations with other KFs, such as which Entity this KF belongs to, or whether there is a Constraint acting on this KF.

Another issue is that there might be a lack of motivation to model knowledge, as the engineer who develops the KBE application is also the one that possesses the knowledge. Therefore, it might seem easier to skip the knowledge modelling and directly implement the knowledge in application code. However, modelling knowledge has more benefits than merely helping the developer develop application code:

- Knowledge that is modelled is not lost when an employee leaves
- Knowledge is more accessible and easier to re-use
- Knowledge becomes a shared resource among the employees
- It enables the ability to directly connect the structured knowledge to application code
- It improves the transparency of the application

And as discussed previously, improving the transparency of the application has a positive effect for all the stakeholders. Furthermore, the TM enables the KP to generate representations of knowledge automatically. These representations focus on a different perspective of the structured knowledge to help understand the knowledge better: e.g. an activity diagram to help understand the design process steps of the application. By having these representations generated automatically, an engineer does not spend time creating them manually. This should decrease the feeling that the extra step required is too time-consuming.

The TM provides the basis to remove the necessity of specialised knowledge engineers and provides domain experts with a knowledge modelling technique. It has been developed with three goals in mind:

- Increase the formality of the ICARE model
- Define how knowledge relates to KBE application code
- Remain simple enough such that specialised knowledge engineers are not required

4.2. The Knowledge Portal

The KP provides the stakeholders with a platform to perform tasks, and it solves the problem experienced that there is a lack of a platform. It is possible to do the following with the KP:

- Capture and structure knowledge by creating Knowledge Forms (KFs)
- Relate KFs with each other
- Automatically generate models based on the created KFs
- Parse newly developed KBE application code automatically into separate code bits, called Code Objects (CO)
- Connect KFs to COs

The KP is developed to help the stakeholders during the development of a KBE application. Each role uses the KP differently, and they have different advantages by using the KP. It is important to note that an engineer may have several of these roles, meaning that they may be assisted in several ways. During the development of the KP, it was assumed that an engineer is both a domain experts as well as a developer. How each role is assisted is described shortly.

Domain Experts are able to use the KP to structure knowledge into KFs. The KP makes use of the TM to help domain experts perform this task. The KP saves all the KFs on the database and it allows the domain expert to easily browse through KFs. This ensures that knowledge is not lost any more, as it is saved in the database. This helps to counter the fact that knowledge is lost when an employee leaves the company. Knowledge is also more accessible, and it is possible to find out where the knowledge comes from, who wrote the KF, and how actual the knowledge is.

Moreover, the expert can find out how the knowledge has been implemented in a KBE application. This makes it possible for an expert to understand how his knowledge is being put to use, and whether the application is doing so correctly. This should help increase the trust in the KBE application and increase the participation of domain experts in the KBE application development process.

Developers that are developing a KBE application can browse all the KFs. Moreover, as the KP generates knowledge representations to help understand the vast amount of KFs, the developer is better prepared to develop a new KBE application as they would have a better understanding of the domain.

A developer might also join a project that is already in progress, or as is seen often an application development process is collaborative. This means that that the developers will have to understand the code developed by their colleagues. The KP helps understand how application code from colleagues was developed as the KP helps users find out what KFs were implemented into which parts of the code. Especially the Activity Diagram should give the developer an idea of how the application works and how it is implemented in the application code. This better prepares the developer to join the development team, and helps developers to collaborate.

Another scenario is when the developer is re-using code for a new project. Now it is possible to understand application code from a previous project better, as it is better documented. Moreover, knowing that certain KFs have been codified already, the developer can directly see how those KFs translate to application code, and it should help to better prepare the developer to develop the new application.

Project Leads has a better overview of the developed application. The project lead verifies whether the application complies to the requirements. Besides being able to run the application to see if it is working, the project lead can now use the KP to figure out what knowledge has been implemented where and whether it has been implemented correctly.

End Users are the customers who use the developed KBE application to automate (part of) their design process. With the KP, the end user is able to understand how the application is generating an output. This increases the trust in the KBE application and helps customers adopt KBE technology. The end user uses the various knowledge representations to get an understanding of what the KBE application is doing.

4.3. The Experiments

Experiments need to be performed to test whether the stakeholders do in fact see a benefit to using the KP. Due to time constraints, a test could be performed for two of the four stakeholders. It was chosen to test developers and project leads during the development of a KBE application.

4.3.1. Developer Experiment

The developer experiment tested whether a developer is better prepared to join a collaborative development process, as the participants were tested whether they could continue on the development of a KBE application which is missing a final module. This is a common scenario when developing a KBE application, hence this case was chosen to be tested. This experiment tested whether the KP helps a developer understand the already developed application code of colleagues better, and whether it helped a developer better prepare for the development of the KBE application. The following question should be answered by the experiment:

”To what extent does the KP better prepare developers to work in a collaborative development process by helping them understand what has been implemented in application code already, and how?”

4.3.2. Project Lead Experiment

The project lead experiment tested whether a project lead is better able to find where knowledge has been implemented in application code, and whether it has been implemented correctly. If it was not implemented correctly, the participant had to mention what needs to be done to ensure that the application code is fixed correctly. This experiment directly tested whether the traceability of knowledge into application code has been established, and whether the transparency of a KBE application has been improved. The following question should be answered by the experiment:

”To what extent does the KP help a project lead verify whether the KBE application complies to requirements, by having an improved transparency due to the fact that knowledge has a direct connection to the KBE application code?”

5

The Traceability Model

The Traceability Model (TM) enables the traceability of knowledge into application code, as it defines how different types of Knowledge Forms (KF) can connect to different types of Code Objects (CO). The TM consists of two parts: the ParaPy Ontology and the Knowledge Ontology. The ParaPy Ontology defines different types of COs, such that the Knowledge Portal (KP) is able to parse them out of application code. Figure 5.1 shows how application code can be parsed into COs. It is important to differentiate between the types of COs as they all have a distinct purpose in a KBE application. Furthermore, the KP allows the domain experts to create KFs and it also allows the developers to connect KFs to COs. How the KP helps the stakeholders is further explained in chapter 6.

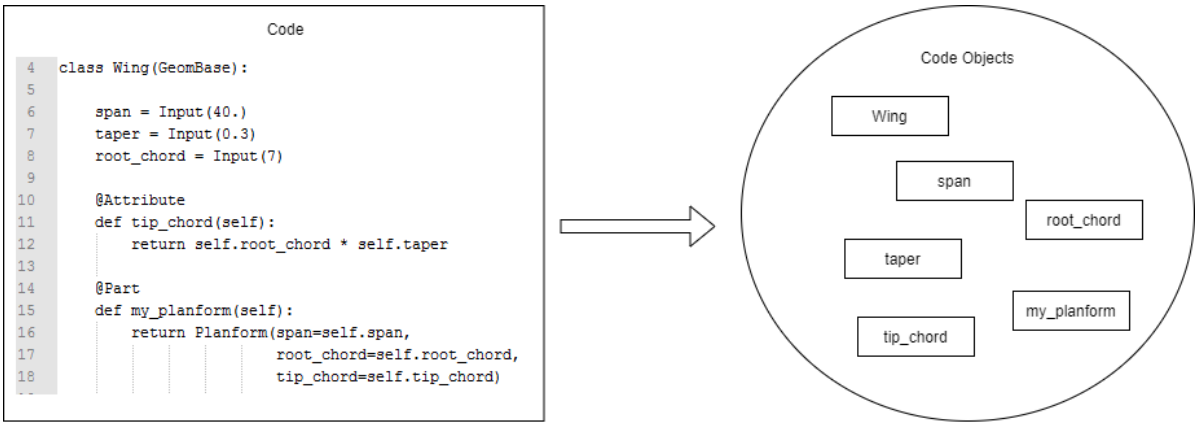


Figure 5.1: From KBE application code, different COs can be retrieved.

The Knowledge Ontology defines types of KFs to assist the domain expert to structure knowledge correctly such that it can be used to develop a KBE application. The TM defines connections between these two ontologies such that it becomes clear how classes of KFs connect to types of COs. This allows a developer to use the KP to make direct connections between the created KFs and the parsed COs. Once KFs and COs are connected, the KP can generate an overview of how the KFs are implemented in application code, as seen in Figure 5.2. How the stakeholders can use the KP to perform their tasks and how they can use it to their benefit is explained in chapter 6.

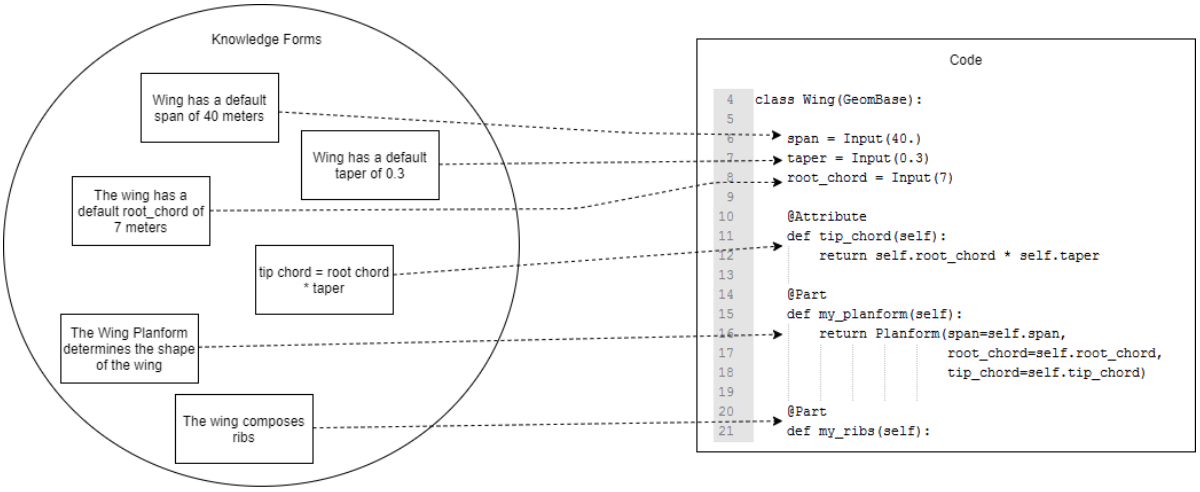


Figure 5.2: Once a KF is connected to a CO, the KP is able to show how KFs are implemented in the KBE application code.

5.1. The ParaPy Ontology

Figure 5.3 shows the ParaPy Ontology. A ParaPy application consists of Code Objects (CO), bits of code with distinct purposes. The highest level CO is the Application itself. An Application comprises several Packages and / or Modules, where a Package is a collection of Modules. These COs explain how a KBE application is built up. A Module comprises `Classes`, these are ParaPy objects (ParaPy objects are referred in `teletypefont`). A `Class` comprises `Slots`. A `Slot` in the ParaPy language is a special type of slot that can be found in Python that enables the KBE technology: e.g. lazy evaluation. A `Slot` can specialise into an `Input`, an `Attribute`, or a `Part`. These give the slot extra functionalities.

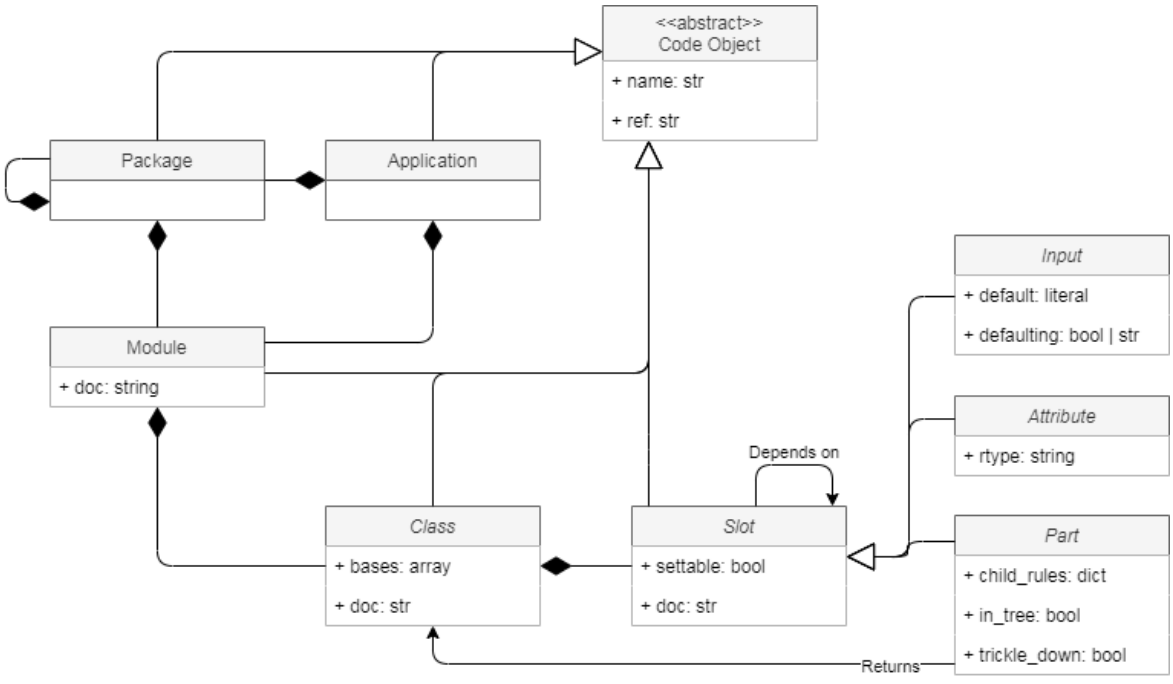


Figure 5.3: The ParaPy Ontology

`Attributes` have the default setting of having their value derived and they are not settable, this means that it cannot be overwritten manually by a user. This type of CO is mainly used to capture engineering rules, and it depends on other COs.

`Inputs` have a unique mechanism to retrieve its value as it might come from a parent object. They are also able to capture a default value, which can be overwritten by either a parent object or manually by a user.

`Parts` returns an instance of a `Class`, and they act as a container for **child rules**, which are used to overwrite `Inputs` from the `Class` being instantiated. Child rules are used to transform a `Class` into a special version of itself when being instantiated by a `Part`. This allows for a generic `Class` to be re-used often in different contexts. This is depicted in Figure 5.4. In this figure, it can be seen that a generic `Class` called 'Lifting Surface' is specialised three times. When one relation is selected (the blue line between the `Part` 'horizontal tail' and the `Class` 'Lifting Surface') it becomes clear that there is a child rule specialising the Lifting Surface: the span is defined by the parent `Class` Empennage. In the ParaPy language this is done by adding a keyword argument to the `Part`, but this report does not dive too deep into how to develop an application with ParaPy.

In a complex KBE application, there are more child rules than shown in this simple case. Moreover, the child rule might be more complex than a simple equality, e.g. an equation. However, this figure shows the advantage as three *Parts* ('wing', 'horizontal tail', 'vertical tail') all instantiate a single `Class` 'Lifting Surface' and put it into different contexts with the help of these child rules.

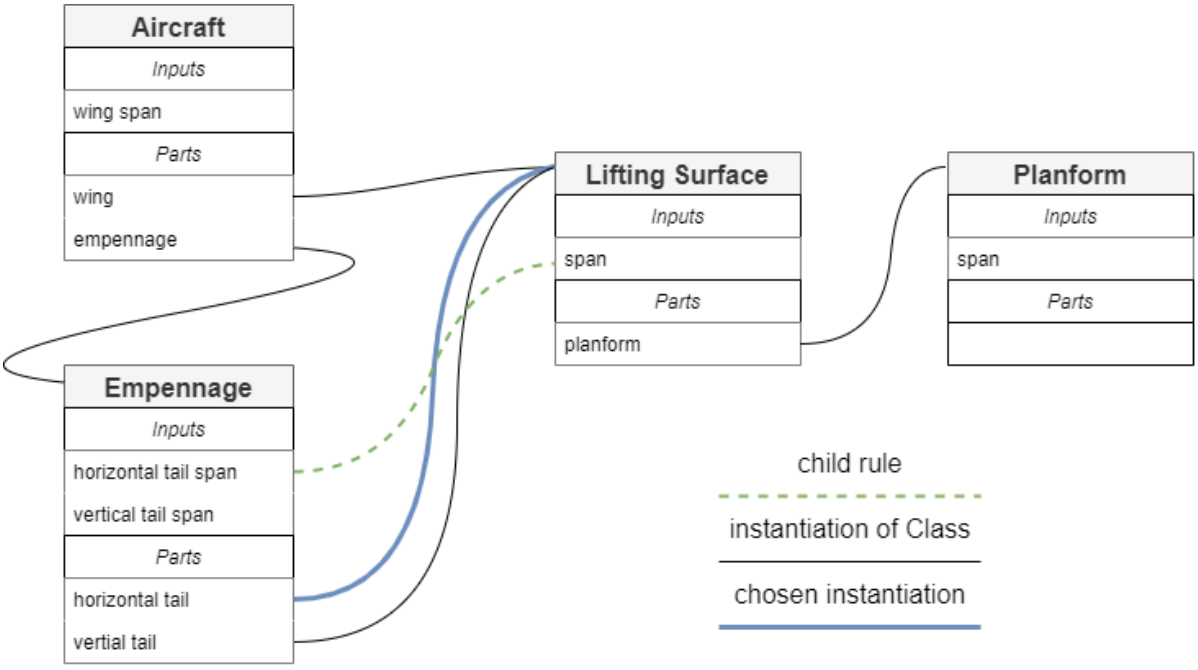


Figure 5.4: How different *Classes* relate to one another. The `Classes` are depicted as the grey boxes, they have `Inputs` and `Parts`; the `Attributes` have been left out in this example. The `Parts` instantiates a `Class`, this enables the parent-child relation. There are three `Parts` that all instantiate the same `Class` 'Lifting Surface'. To visualise the effect of child rules, one relation has been selected (blue line), and the child rule attached to that relation is shown in the dotted green line. Each relation has its own set of child rules to specialise the generic `Class`.

5.2. The Knowledge Ontology

The Knowledge Ontology defines different classes of Knowledge Forms (KF). The idea is that each class of KF enables a domain expert to capture a different perspective of knowledge: e.g. the design process, the hierarchy of the product, the engineers rules. The Knowledge Ontology builds upon the MOKA ICARE (Illustration Constraint Activity Rule Entity) model, which has defined five different classes of KF as can be seen in Figure 5.5. This figure also shows the superclass Knowledge Form which is inherited by all other classes.

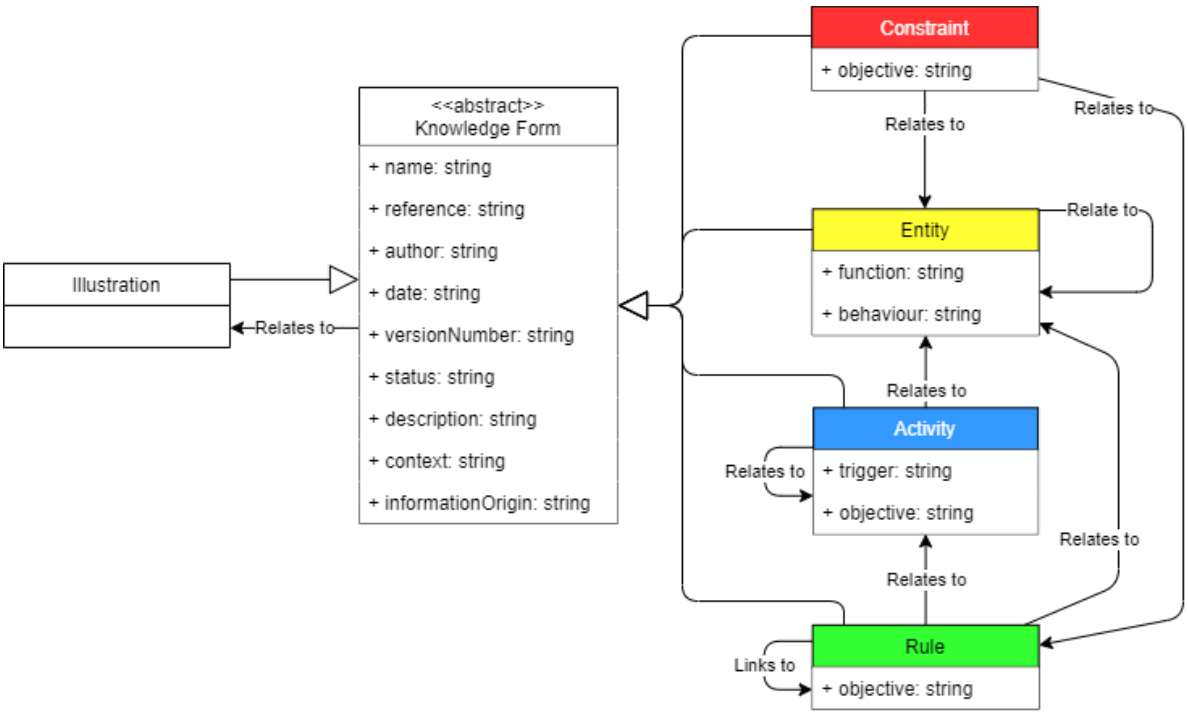


Figure 5.5: MOKA's ICARE model defines 5 classes of KFs, each is a specialisation of the superclass Knowledge Form.

The Knowledge Ontology revises the different classes defined by the ICARE model and the relations between them. The aim of the Knowledge Ontology is to increase the formality of the ICARE model such that a KBE application can directly be codified from it. Currently, the ICARE model is one of two steps to formalise knowledge correctly for use in KBE applications. How the Knowledge Ontology relates to the ICARE model and the formal models of MOKA was shown in Figure 4.1.

The next few sections discuss three things: how MOKA defines each class, the differences made to it for the Knowledge Ontology, and then a practical example of how it can be used to model knowledge in a KBE application. For clarity, a KF is referred to capitalised and in italic: e.g. *Entity*.

5.2.1. Entity

MOKA

MOKA explains that an *Entity* is used to describe the 'product objects'. With the *Entity* it is possible to describe a hierarchy of objects, as can be seen in Figure 5.6. The *Entity* describes the object in a generic fashion, and it is possible to capture the relation with another *Entity*.

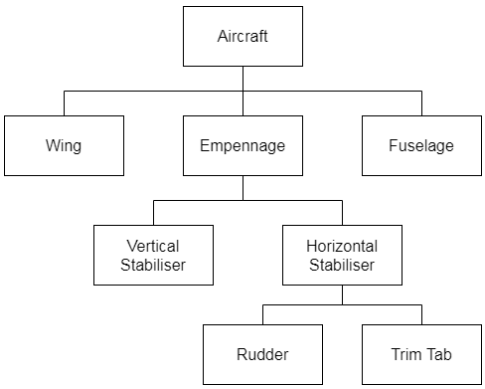


Figure 5.6: An example of a hierarchy of product objects.

With the *Entity*, ICARE allows users to capture the properties of an *Entity* and other related infor-

mation in a textual description field.

Knowledge Ontology

In the Knowledge Ontology, the *Entity* is still used to describe product objects. However, the properties have been taken out of the *Entity* and given its own class: *Property*. The reason for this is the fact that the ICARE model did not allow a detailed enough capture of the *Properties* of an *Entity*. One of the reasons why MOKA did not add this to the ICARE model, is because domain experts usually did not have any application coding experience. However, as mentioned throughout the report, engineers today do have application coding experience. Moreover, in SMEs, a single engineer has multiple roles e.g. developer and domain expert. It is therefore assumed that a domain expert structuring knowledge with the Knowledge Ontology understands how to structure *Properties* detailed enough. That means that it is possible to capture the structure of a KBE application in more detail with the Knowledge Ontology than compared to the ICARE model.

An *Entity* can now comprise two types of *Property*: an *Input* and an *Output* as can be seen in Figure 5.7. An *Input* is a type of *Property* that may have a default value, and it can have its value overwritten. An *Output* has its value determined by an expression, which is evaluated during run-time of the KBE application. An expression can be complex or literal. Complex refers to the fact that it is an expression that depend on other *Properties*. A literal expression refers to the fact that the expression is of a single data type (e.g. numeric).

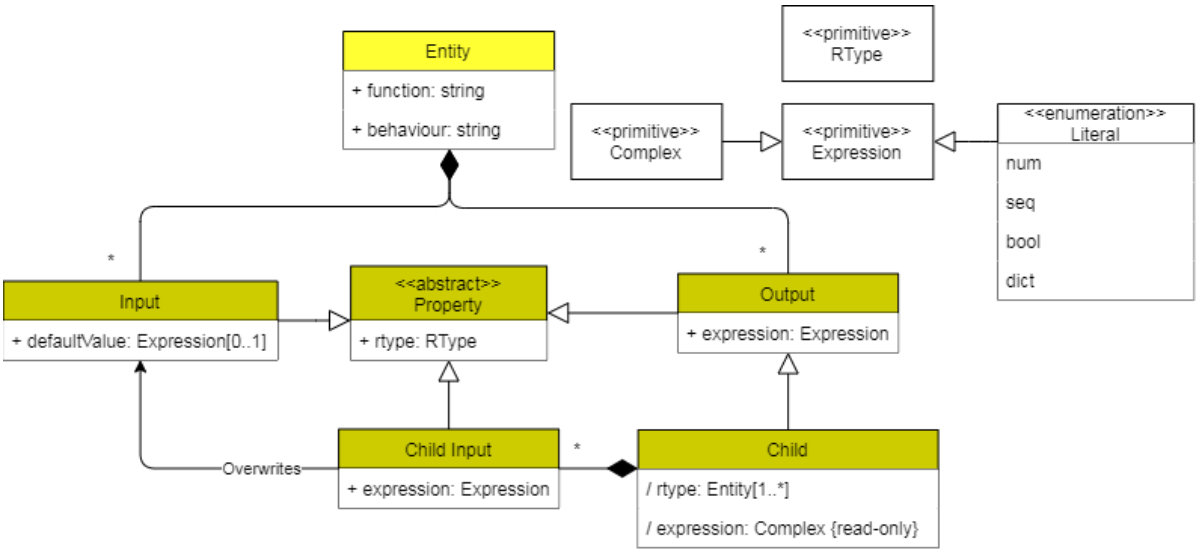


Figure 5.7: An *Entity* comprises *Inputs* and *Outputs*. An expression can be complex or literal. Complex refers to the fact that it is an expression that depend on other *Properties*. The expression of a *Child* is read-only, meaning it cannot be modified manually. The *Child* returns one-to-many *Entities*, allowing to capture a collection. Finally, the back-slash means that the attribute is computed automatically from other information by using values of other attributes.

Furthermore, the relation between two different *Entities* is more complex then what can be captured by the ICARE model. When an *Entity* becomes a child of another *Entity*, it does not remain the same but becomes a specialised version of itself. This is because a parent *Entity* can overwrite *Properties* of the child *Entity*, by using ***child rules***. To capture this complex relation between two *Entities*, there are two more types of *Properties*: *Child* and *Child Input*. A *Child* is a special type of Output that always returns another Entity, this enables the relation between two *Entities*. This is different than the ICARE model where it was possible to relate *Entities* directly. The reason to add a layer in between *Entities* is thus to enable the capture of child rules. This is possible as a *Child* comprises *Child Inputs*. *Child Inputs* overwrite *Inputs* from a child *Entity*, specialising the child *Entity* to put it into context of the parent *Entity*. This means that the *Child* acts as a container to gather all the *Child Inputs* containing the child rules for the child *Entity*.

By having the *Property* separated from the *Entity* it is possible to define clearly how different types of *Properties* are connected to different types of Code Objects (COs). Figure 5.8 shows how the discussed KFs connect to types of COs. An *Entity* always connect to a `Class`, which can be explained as they

have a similar role: *Entities* are a collection of *Properties*, and a `Class` is a collection of `Slots`. An *Output* connects to an `Attribute`, as both have their value evaluated during run-time. An *Input* connects to an `Input`, as both can have their values overwritten and may have a default value. A *Child* connects to a `Part`, as they both return another `Class` / *Entity* and they both fulfil the function of being a container for child rules. The *Child Input* connects indirectly twice: onto the `Input` that is overwritten, and onto the `Part` that is overwriting the `Input`.

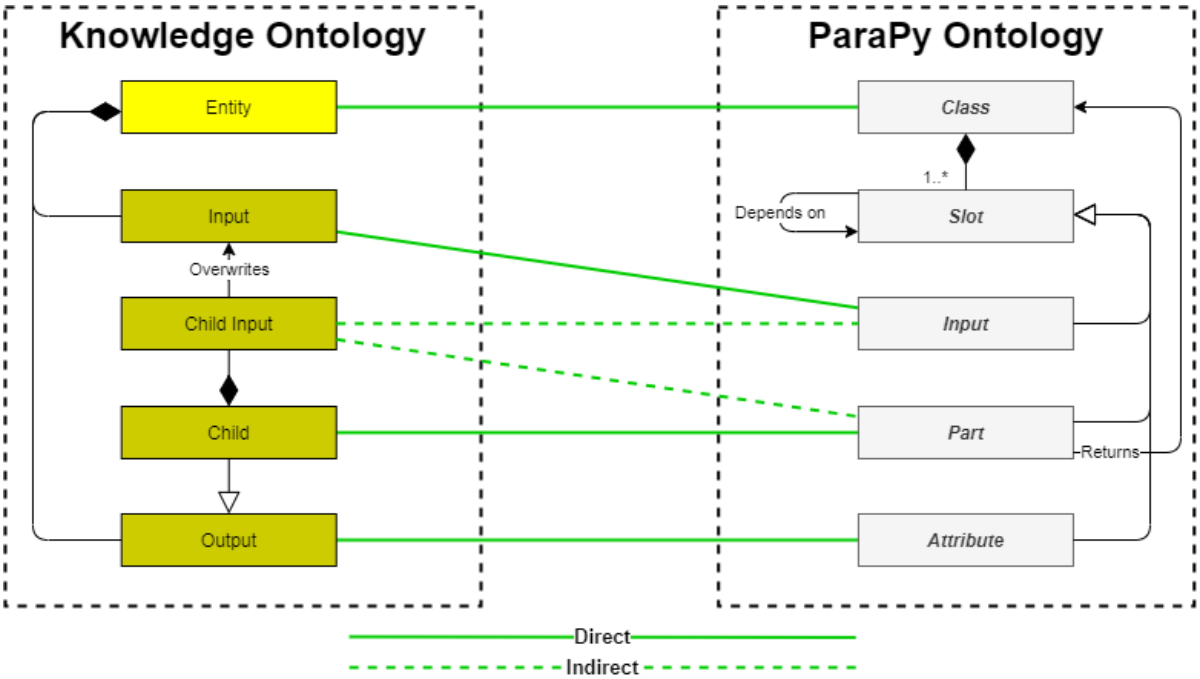


Figure 5.8: Overview of how *Entities* and *Properties* relate to types of COs. The green line represents the relation. The dotted green line is a relation that can be inferred, and represents an indirect relation.

By defining a direct connection between the different types of *Properties* and COs, it is no longer necessary to directly connect other types of KFs to COs as these relations can be inferred. This is because each other type of KF is (in)directly related to a *Property*.

Practical Example

An example instance of an *Entity* and the different type of *Properties* can be seen in Figure 5.9. In the right half of this figure, the application code counterpart of the model can be found to help understand what knowledge the model is capturing. For example, *Properties* cannot explain what the expression is when it is complex, this is explained by *Activities* and *Rules*; therefore the `Attribute` returns a blank as it cannot be captured with just these types of KFs. It also shows to which pieces of code the different KFs are related to. With the currently discussed forms, it is possible to capture the fact that an *Entity* 'Aircraft' comprises multiple *Properties*. The two *Inputs* can capture the fact that there are default values. It is known that the expression of the *Output* is complex; however, it is not possible to capture the exact expression with a *Property*. This is possible with other forms as will become clear later on.

Finally, the *Entity* 'Aircraft' comprises a *Child* 'my_wing' which returns an *Entity* 'Lifting Surface'. The *Child* 'my_wing' enables the part-assembly relation between the Aircraft and the Lifting Surface. As explained by the name of the *Child* 'my_wing', the *Entity* 'Lifting Surface' is returned by it and the corresponding *Child Input* 'lift' will specialise it into a wing. This is because *Child Inputs* capture child rules that overwrite *Inputs* of another *Entity*, as explained in section 5.1. Therefore, the *Child* 'my_wing' comprises a *Child Input* 'lift' which overwrites the *Input* 'lift' of the *Entity* 'Lifting Surface'.

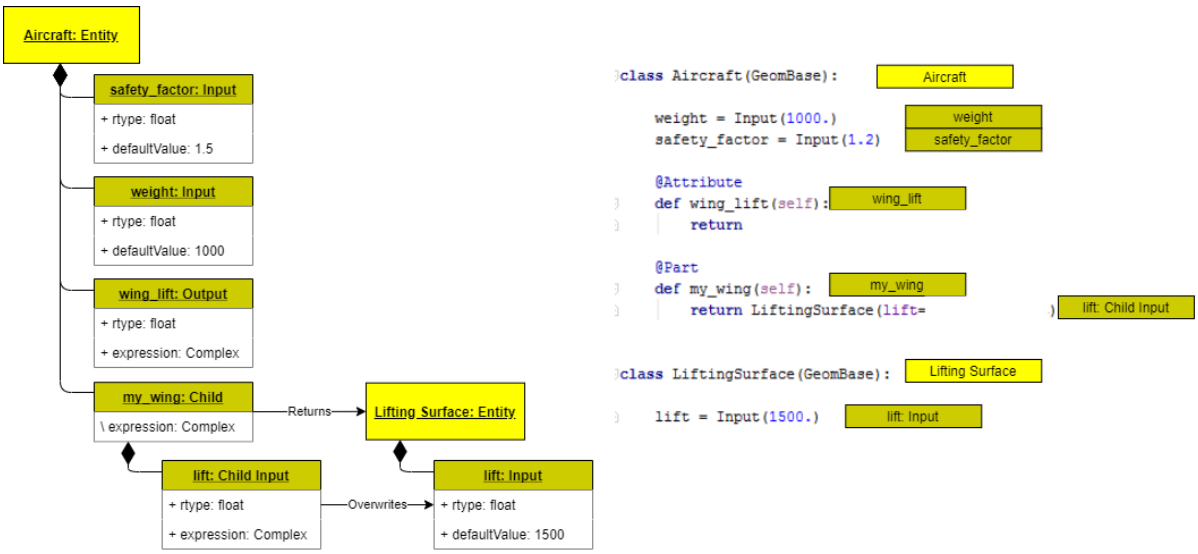


Figure 5.9: An *Entity* comprises two *Inputs*, an *Outputs*, and a *Child*. The *Child* 'my_wing' returns the *Entity* 'Lifting Surface'. The *Child* 'my_wing' comprises a *Child Input* that overwrites an *Input*. The code counterpart shows what can be captured with these KFs. It is not possible to capture what the expression is of the 'wing_lift' as this knowledge is captured in *Activities* and *Rules*, which will be explained later on.

As can be seen from the example, the *Entity* and the different types of *Properties* allow to capture hierarchy just as in the ICARE model. It also allows a more detailed capture of the knowledge behind the part-assembly relation between *Entities*. Moreover, it allows the capture of child rules which specialise an *Entity* when being a child *Entity*.

5.2.2. Constraint

MOKA

MOKA explains that a *Constraint* can be used to describe limitations on the product, such as: the number of struts must be 2, 3, 4; pay load must lie between limits; or there must be 1 or 2 nose gear struts. Therefore, it allows a *Constraint* to be related to an *Entity*. The attribute 'objective' is used to describe the *Constraint* that is affecting the *Entity*.

Knowledge Ontology

The problem with the *Constraint* from the ICARE model is that it does not relate to the *Property* that is being constrained but only to the *Entity*. This makes it difficult for a computer to process the knowledge and find out which *Property* is actually being constrained. Therefore, a change has been made: the *Constraint* now constrains a *Property* directly as seen in Figure 5.10. There is also a new attribute called 'consequence' which can be used to explain what happen when the *Constraint* is violated, e.g. a lethal error or a warning.

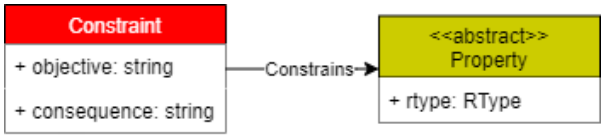


Figure 5.10: A *Constraint* now directly constrains a *Property*.

Practical Example

Two examples of a *Constraint* can be seen in Figure 5.11. The first one constrains the possible value of an *Input*, making sure that the number of blades is uneven. The second one constrains two values, making sure that the root chord is always larger that the tip chord.

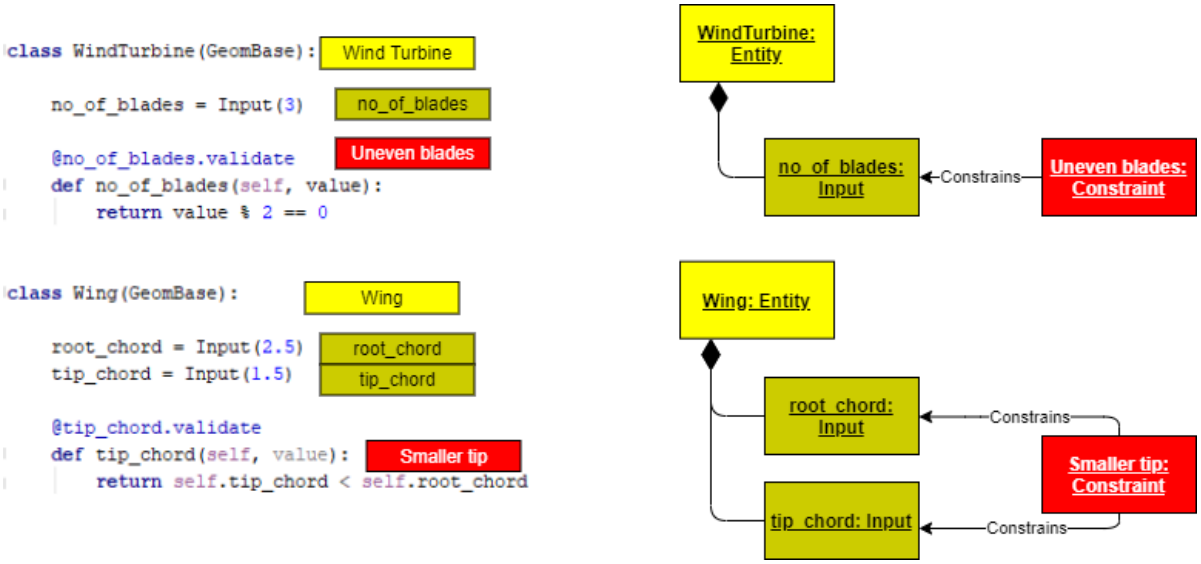


Figure 5.11: The first *Constraint* restricts the possible outcome to an uneven number. The second *Constraint* ensures that a *Property* cannot exceed the value of another *Property*.

5.2.3. Activity

MOKA

In the ICARE model, an *Activity* is used to describe a step in the design process. It can be used to explain what needs to be done to come to an output, and it is possible to model several layers of *Activities*. Figure 5.12 shows how different layers of the design steps can be captured with *Activities*. An *Activity* relates to an *Entity*, this is the *Entity* that is involved in the design step.

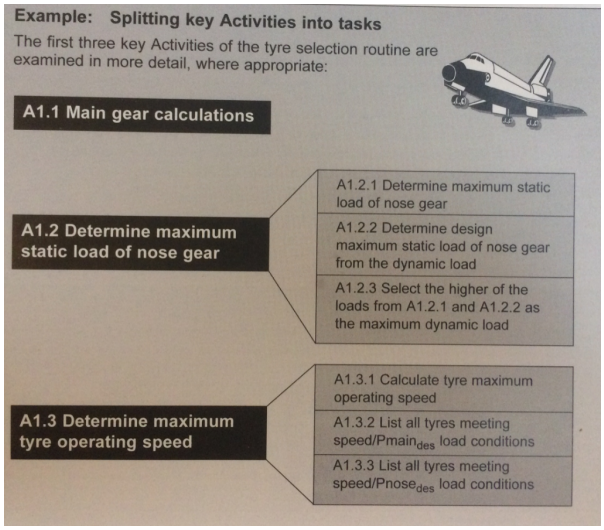


Figure 5.12: The ICARE models explains that an *Activity* is used to describe a step in the design process.

Knowledge Ontology

The problem of how the ICARE model describes an *Activity* is mainly that it is unclear how an *Entity* is affected. This is because the ICARE model only allows an *Activity* to relate to an *Entity* and not directly with the properties of an *Entity*. As the Knowledge Ontology has the new *Property* class, the *Activity* can now directly relate to a *Property*.

There are two relations possible between an *Activity* and a *Property*: *Properties* can be provided to the *Activity*, and the *Activity* can set the expression of a *Property*. This means that an *Activity* can capture knowledge about what *Properties* are used in the expression of another *Property*. Exactly how

these *Properties* are used in the expression cannot be captured in an *Activity* but in a *Rule*, as explained later in subsection 5.2.4

To help a domain expert to structure knowledge correctly, there are two types of *Activities*: *Assignment* and *Child Definition*. The *Assignment* can be seen in Figure 5.13. It can have *Inputs* and *Outputs* provided, and it sets the expression of an *Output*. In the special case that an *Input* has an expression for its default value, the *Assignment* can also set its expression.

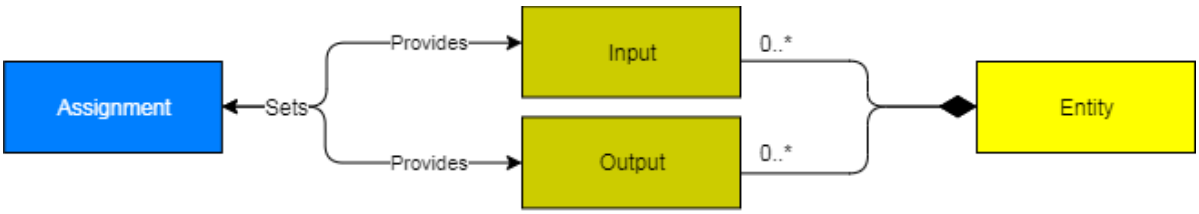


Figure 5.13: An *Assignment* has *Inputs* and *Outputs* provided, and can set the expression of an *Input* or an *Output*.

The *Child Definition* is used to set the expression of a *Child*. In other words, it is used to explain how a child *Entity* is brought into context of a parent *Entity*. As explained in subsection 5.2.1, there are *Child Inputs* which overwrite the *Inputs* of the child *Entity*. Just as the *Child* is a container for all the *Child Inputs*, the *Child Definition* is a container for all the *Assignments* that set the value of the *Child Inputs* as shown in Figure 5.14. As can be seen, when a *Child Definition* comprises an *Assignment*, that *Assignment* can only set the value of a *Child Input* instead of either an *Input* or an *Output*.

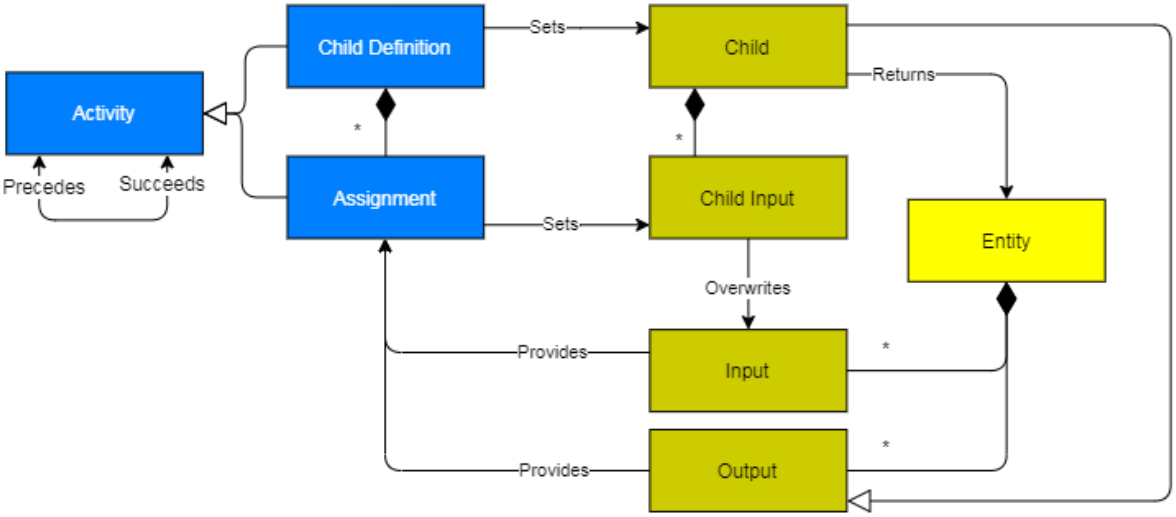


Figure 5.14: A *Child Definition* sets the expression of a *Child*. It comprises *Assignments*, in which case the *Assignments* set the expression of *Child Inputs*.

Practical Example

To help understand how an *Assignment* and a *Child Definition* can be used, two examples are presented. These examples continue on the example shown in Figure 5.2.1 for the *Entity* and *Property*.

Figure 5.15 shows the *Entity* 'Aircraft' which comprises the same *Properties* as before: two *Inputs*, one *Output*, and a *Child*. The *Output* 'wing_lift' has a complex expression which is set by an *Assignment*, as seen in the figure. To do this, two *Inputs* are provided to the *Assignment*. Now it is possible to capture the fact that the *Output* 'wing_lift' is a function of the *Input* 'safety_factor' and the *Input* 'weight'. This is shown in the code as $f(weight, safety\_factor)$. However, it is still not possible to capture exactly how these two *Inputs* are used to form an expression. This knowledge is captured later on in a *Rule*, as explained later in subsection 5.2.4. What can be captured by the *Assignment* is the order of execution (in this case it is the first), and the fact that it is using two *Inputs* to set the value of a single *Output*.

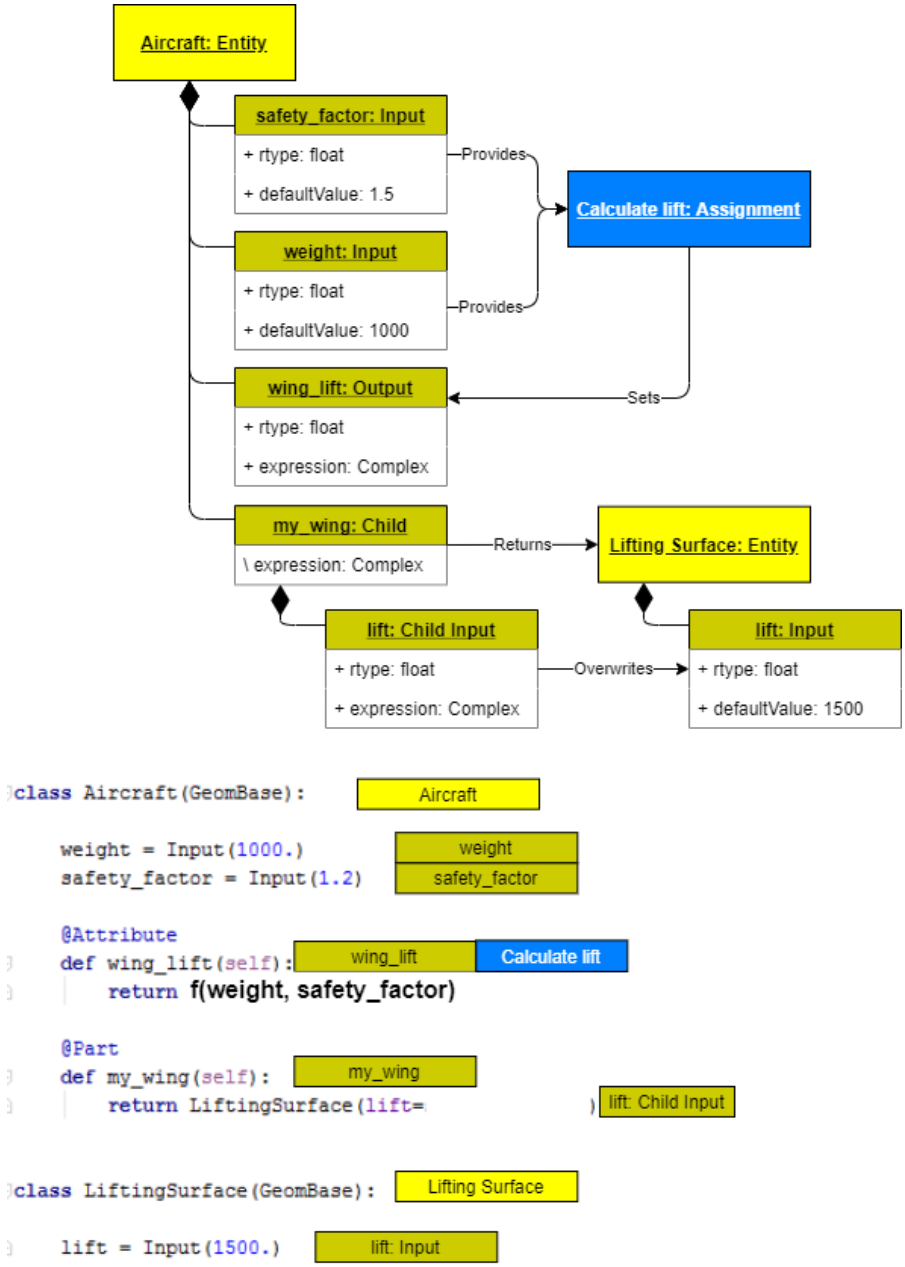


Figure 5.15: An *Entity* 'Wing' comprises two *Inputs* and one *Output*. The two *Inputs* are provided to the *Activity*, which knows that it can use them to set the expression of the *Output*. It can now be said that the 'wing_lift' is a function of the 'safety_factor' and the 'weight'.

Figure 5.16 shows the same *Entity* 'Aircraft'. The *Child* 'my_wing' returns the *Entity* 'Lifting Surface' and it comprises the *Child Input* 'lift' which overwrites the *Input* 'lift' from the *Entity* 'Lifting Surface'. For every *Child Input*, there must be an *Assignment* that sets its expression. This *Assignment* is part of the *Child Definition*, as can be seen in the figure where a *Child Definition* comprises an *Assignment*. The *Child Definition* can be seen as a container that collects all the *Assignments* that set the expression of the *Child Inputs*, similarly to how a *Child* is a container for all the *Child Inputs*. In this case, there is only one *Child Input*. What can be noted is that in the application code, the expression is not a function of several properties; but it is a direct assignment. This is a possibility of an *Assignment* when setting the expression of a *Child Input*. The *Child Definition* has now captured how a child *Entity* is defined to be the child of the parent *Entity*, and it can also capture the order of the design steps like any other *Activity*. In this case, the *Child Definition* 'Generate wing' succeeds the *Assignment* 'Calculate lift'.

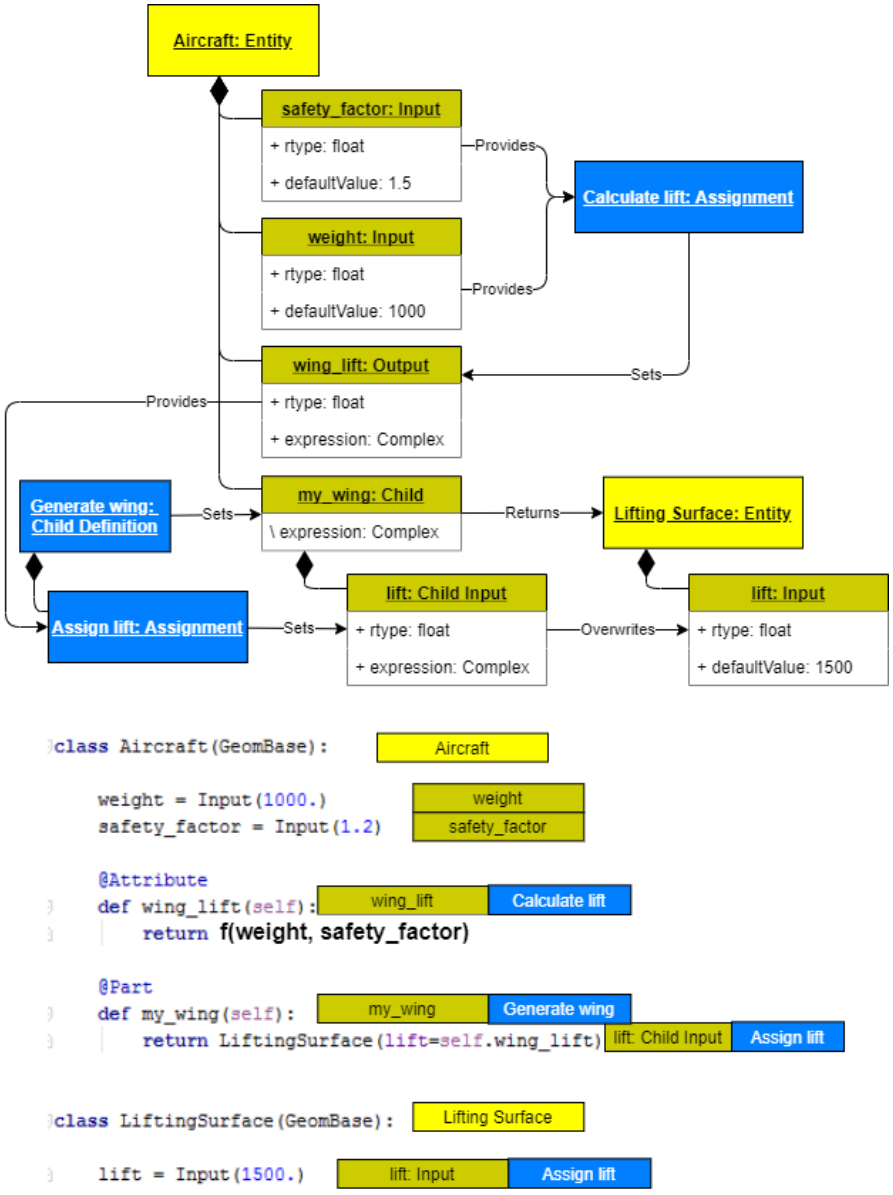


Figure 5.16: An *Entity* 'Aircraft' comprises a *Child* 'my_wing' which comprises a *Child Input* 'lift' which overwrites the value of the *Input* 'lift' of the *Entity* 'Lifting Surface'. A *Child Definition* 'Generate wing' comprises an *Assignment* 'Assign lift' which sets the expression of the *Child Input* 'lift' which is a composition of the *Child* 'my_wing'.

As can be seen from the examples, there is one aspect of knowledge that still cannot be captured with the other classes of KFs: how *Properties* are used in the expression of another *Property*. This aspect can be captured in a *Rule*, as is explained in the next section.

5.2.4. Rule

MOKA

According to the ICARE model, a *Rule* explains how to move from an input to an output: it provides the know-how of an *Activity*. *Rules* and *Activities* often share the same name when modelling with the ICARE model, as they describe the same process step. MOKA acknowledges that there is an overlap between *Activities* and *Rules*. However, the difference is that the *Activity* describes the 'WHAT' as to position it in the overall design process, identify objectives, and link it to *Entities*; while the *Rules* describe the 'HOW'.

Knowledge Ontology

A *Rule* governs *Activities*. As in MOKA, it provides the know-how and explains how to move from an input to an output. However, in the Knowledge Ontology the *Rule* is captured a bit more formally. Moreover, there is a clear distinction set between *Activities* and *Rules*, as an ontology should remove ambiguities. In the Knowledge Ontologies, *Activities* capture the position in the overall design process and which *Properties* are provided and set. *Rules* explain how the provided *Properties* are used to set the expression of an output *Property*.

The *Rule* has two attributes: inputs and outputs. They help understand how *Properties* given as an input are used to define the expression of the output. These can be seen in Figure 5.17. The *Rule* has a third attribute 'body' which explains how an output is defined using the inputs. The *Rule* is in a generic context while the *Activity* it is governing is in a specific context. This means that the *Rule* can capture engineering knowledge regardless of the *Activities* that it governs.

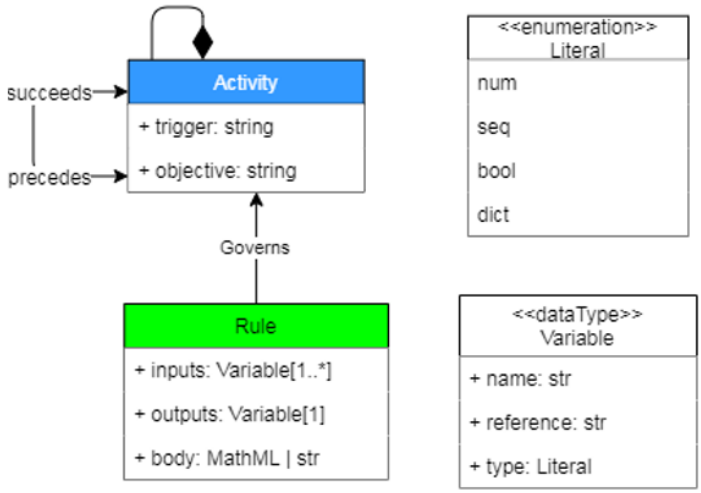


Figure 5.17: A *Rule* governs an *Activity*. A *Rule* has inputs and outputs, these are used in the body to form an expression. When an *Activity* is being created, the domain expert connects *Properties* provided to the inputs of the *Rule*. The domain expert also connects the output of the *Rule* to the *Property* being set. This allows the KP to infer that the expression of the output *Property* is the expression given by the *Rule*.

The domain expert needs to define variables in the inputs and outputs which are used in the body. A variable contains a name, a reference that is used in the body, and a data type. Once variables are defined as inputs and outputs, and the body is finished, the *Rule* is ready to govern an *Activity*. This is done by having the *Activity* connect the *Properties* provided to the correct variables in the inputs of the *Rule*. The output of the *Rule* is connected to the *Property* that is being set by the *Activity*. From here on, the KP can infer that the expression of the output *Property* is equal to the body of the *Rule*.

Practical Example

The example for the *Rule* also continues upon the previous examples of the other classes of KFs. The last piece of knowledge missing from those example was how *Properties* are used in the expression of another *Property*. As can be seen in Figure 5.18, there is now a *Rule* governing the *Assignment* 'Calculate lift'. The *Rule* has two inputs: weight and safety_factor, and a single out lift. The body of the *Rule* explains that it is a simple equation that multiplies the two inputs.

When creating the *Activity*, it is possible to connect the *Properties* to the correct inputs of the *Rule*. The same applies for the output of the *Rule*. By doing this, it is possible to infer what the expression is of the output *Property*. As can be seen in the figure, all knowledge about this piece of application code was able to be captured. No *Rule* was necessary for the *Assignment* of the *Child Input*, as it was directly assigning a *Property* to the *Child Input* ($lift = wing\_lift$).

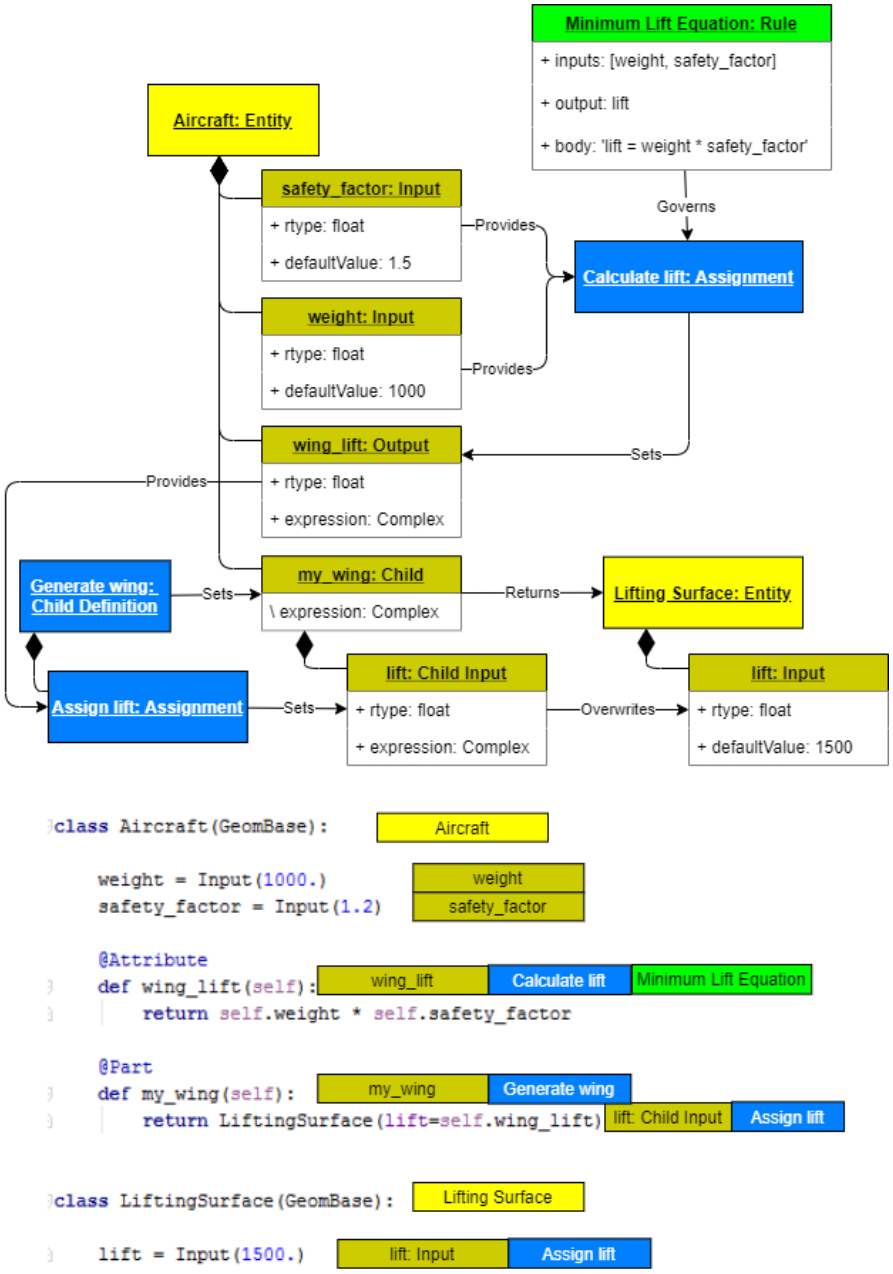


Figure 5.18: The *Rule* 'Minimum Lift Equation' is governing the *Assignment* 'Calculate lift'. It has two inputs, weight and safety_factor, and a single out lift. The body explains what the expression is. The *Assignment* connects the two *Properties* provided to the inputs, and also connect the *Property* being set to the output of the *Rule*. Now it is clear how the *Property* 'wing_lift' is going to be evaluated.

5.3. Connecting KFs to COs

In subsection 5.2.1 it was shortly discussed how *Entities* and the different types of *Properties* are connected to different types of COs. From these connections, all the other types of KFs can connect to COs indirectly. This can be inferred by the KP automatically, so that the domain expert does not need to define each relation separately. The novelty is in the fact that the different types of *Properties* serve different purposes which are similar to those found in the types of COs found in ParaPy application code. Figure 5.19 shows how all the types of KFs connect to code (in)directly.

There are a few notable connections. It can be seen that a *Constraint* can connect to any type of `Slot` indirectly, as it can affect any type of *Property*. Also notable is that a *Rule* is always connected to an *Activity*, and based on what type of *Property* that *Activity* is connected to determines how it connects

to application code. Furthermore, a *Child Input* connects indirectly to a `Part` and an `Input`. Finally, only an *Assignment* that is in a composition relation with a *Child Definition* can connect to a *Child Input*.

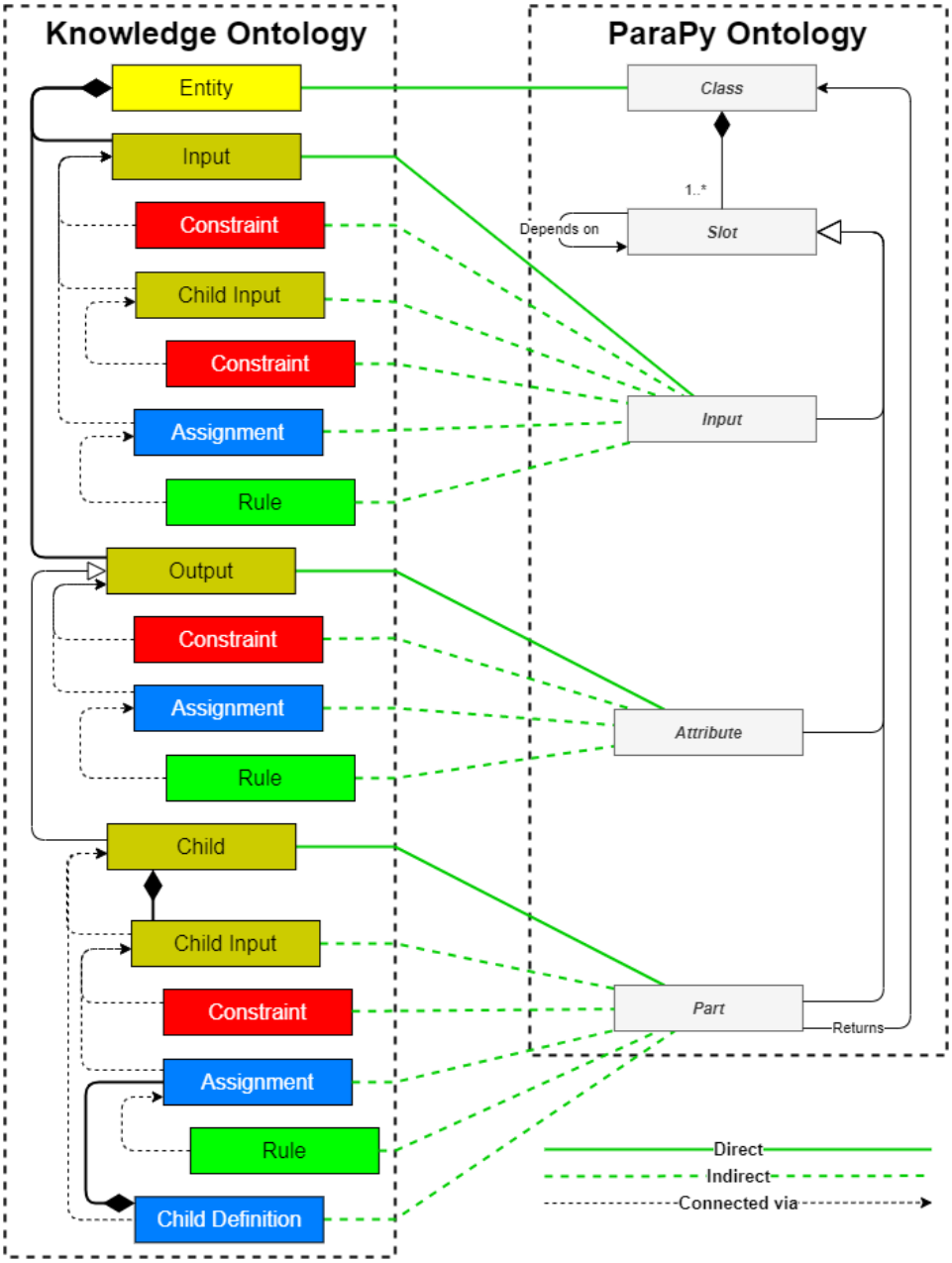


Figure 5.19: An overview of the (in)direct connections between KF classes and COs.

6

The Knowledge Portal

The Knowledge Portal (KP) has been developed to provide a software platform that assists the stakeholders to perform their tasks. There are three scenarios that are described for the type of KBE application development processes, the steps taken for each scenario is shown in Figure 6.1. The three scenarios are described as follows:

1. **From Scratch**: A new KBE application is being developed while there is no knowledge structured yet and there is no application code to re-use. This scenario is the only scenario to incorporate all the steps to develop a KBE application.
2. **Application Re-use**: There is already a KBE application, it will be parsed by the KP into COs. However, knowledge has not been modelled yet. Therefore, knowledge will be structured by creating KFs and connecting them to the already existing COs.
3. **Knowledge Re-use**: Knowledge has already been structured for another application, hence there are already KFs. However, the new application still has to be developed. Once developed, it will be parsed by the KP into COs, and then the developers can connect the KFs to the corresponding COs.

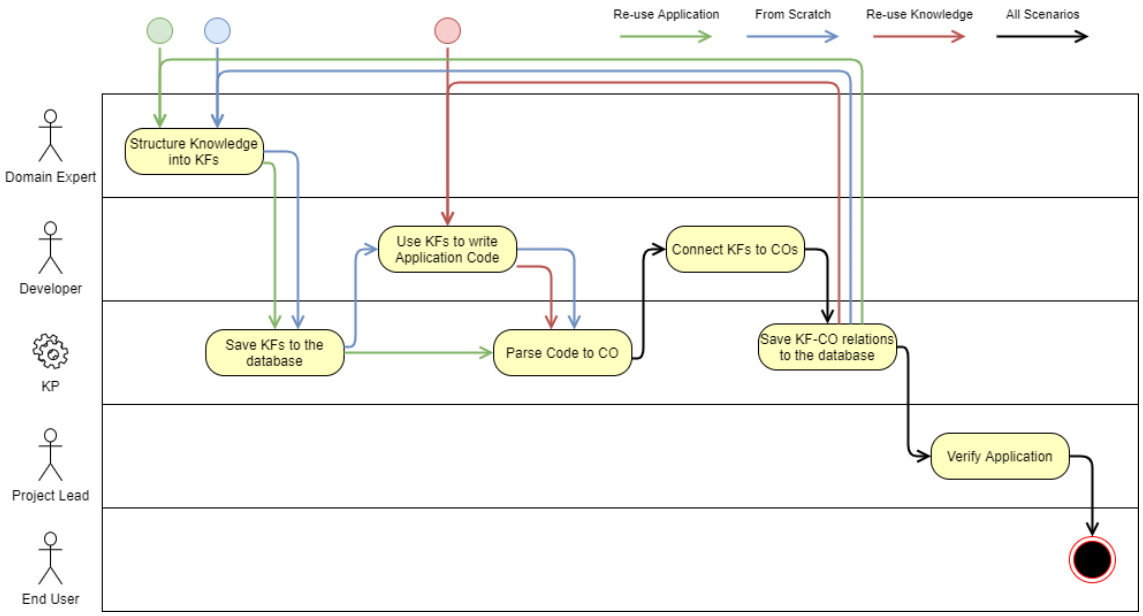


Figure 6.1: The steps in a KBE application development process with the three different scenarios described, as defined for this methodology. After KFs are connected to COs, the process can repeat itself as the development of a KBE application is iterative. Once all iterations are complete, a project lead can verify that the application fulfils all requirements and deliver it to the customer so that the end-user can use the application.

These three scenarios are all extremes, as it is possible that a single project follows all three scenarios, as some parts of application code may be re-used, some knowledge that has been captured already might be re-used, and some parts might need to be developed from scratch. However, the scenarios should help the reader have a clear picture of how the KP can be used.

As can be seen in Figure 6.1 there are seven steps in total, from which four are performed by the stakeholders: structure knowledge into KFs, codify knowledge that can be parsed into COs, connect KFs to COs, and verifying the application to deliver it to the customer. Once delivered, the end-user can make use of the application. These steps can all be performed using the KP. As can also be seen in the figure, the KP ensures that KFs and relations are saved to the database.

The KP uses the Traceability Model to restrict possible relations between different KF classes, this is how domain experts are guided to structure knowledge correctly. The TM can be imported into the KP as a JSON file, an open-standard file format, which allows an updated model to be swapped in easily when changes are made, or to use the model in various applications. To help understand the KFs and how they relate to each other and COs, there are several knowledge perspectives that are generated automatically by the KP. These knowledge perspectives are explained in section 6.2. The user interface of the KP can be seen in Figure 6.2.

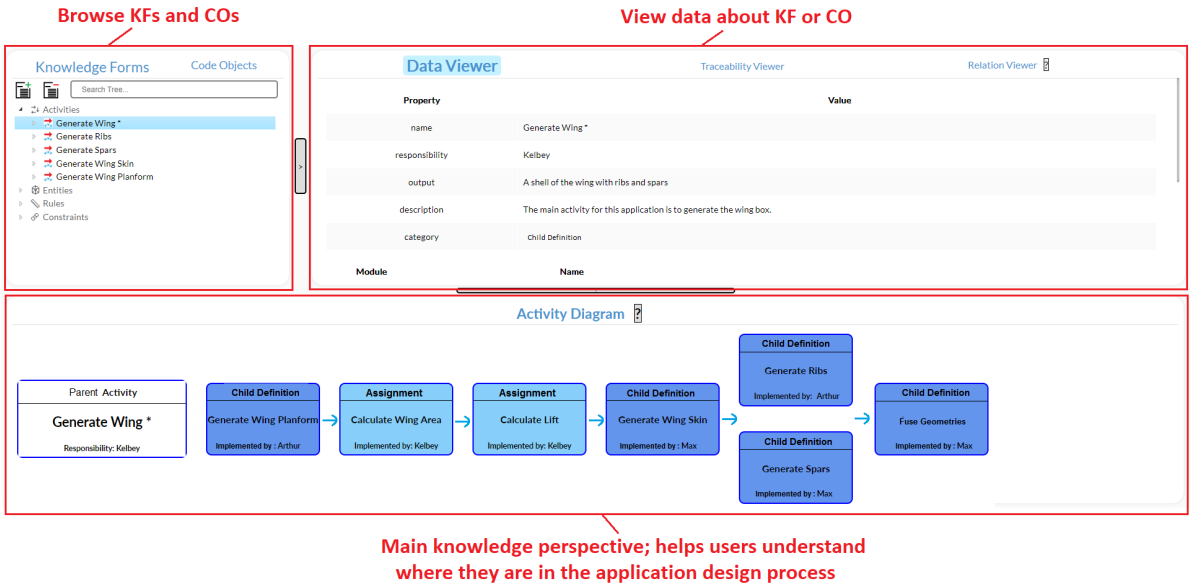


Figure 6.2: User interface of the Knowledge Portal. There are three main windows: one to browse KFs and COs and to load them into the different windows, one to view the data and relations a KF or CO has in different perspectives, and one to view the Activity Diagram. The Activity Diagram helps users understand where in the design steps they are, in this example they are in the step to generate a wing which consists of 7 sub-steps.

There are three main windows in the user interface of the KP: the browser, the viewer, and the activity diagram. The browser loads all the KFs into a KF tree and also all the COs that have been parsed into a CO tree. The user can load a KF into the various knowledge perspectives from here. It is also possible to navigate through the code using the CO tree. The other two windows contain the four different knowledge perspectives. The bottom window contains the knowledge perspective which is thought to be the most important for all users. Currently this is the Activity Diagram as that gives the user an idea where in the application design process they currently are. This can be changed to comply to the wishes of the customer.

This chapter discusses how a user will be using the KP, and what can be done with it. How the KP is developed, and how it generates knowledge perspectives automatically, can be read in Appendix B.

6.1. Structuring Knowledge

A domain expert can structure knowledge by creating new Knowledge Forms (KF), Figure 6.3 step 1. This opens up a window asking for the class of the new KF, step 2 in the figure. Once a class has been selected, the attributes of that class are listed. The KP does this by looking up the attributes that

correspond to the chosen class in the TM. The domain expert can now fill in every attribute that is listed. It is possible that the type has to be chosen as well, in the figure that would correspond to the Input, a type of Property. Finally, the domain expert can capture how the new KF relates to other KFs. This can be done by clicking on the + sign, step 3 in the figure. When this button is clicked, the KP scans the TM and only allows the domain expert to relate the KF with another viable KF. For example, a Rule is only allowed to relate to an Activity. This is done by disabling the other types of KFs ensuring no wrong relation can be made.

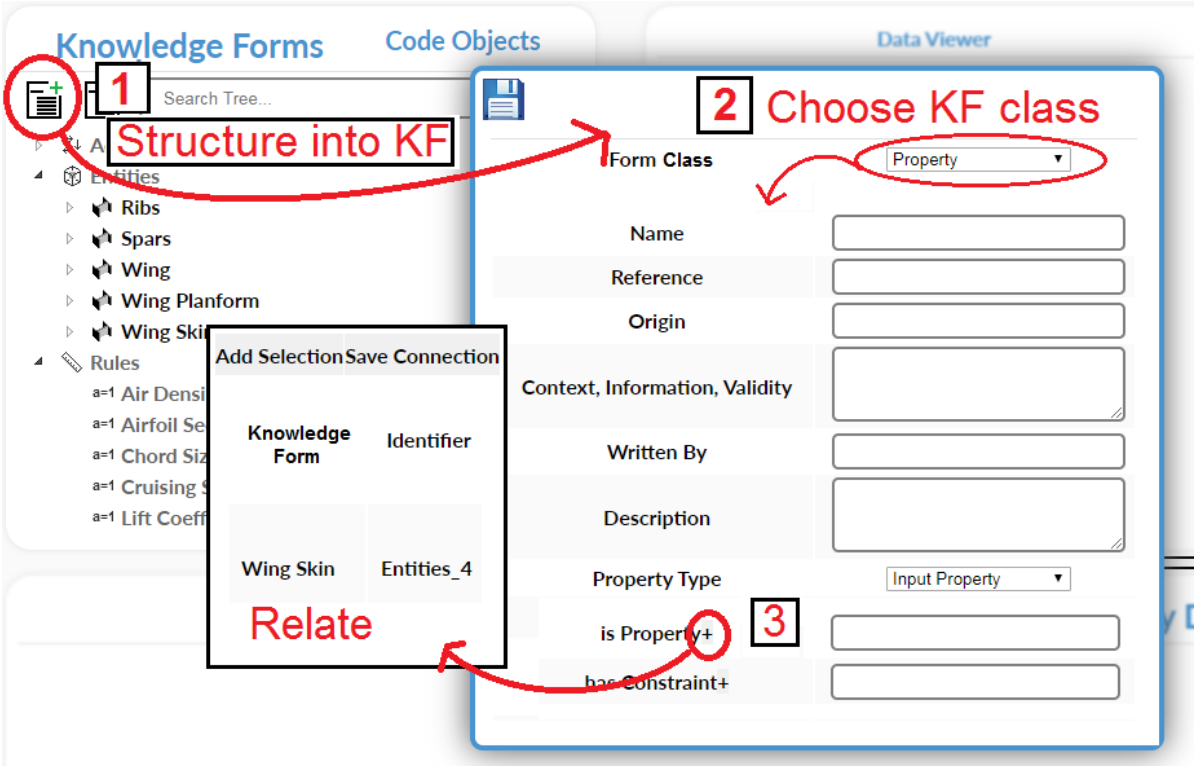


Figure 6.3: Creating a new KF can be done in three steps. First a new KF should be opened. Secondly, the domain expert can choose the KF class and fill in the information required for that class of KF. Finally, the domain expert can relate the created KF with other KFs by clicking on the '+'.

6.2. Knowledge Perspectives

When the domain expert has populated the KP with KFs, some knowledge perspectives can be generated. Knowledge perspectives are different representations of the capture knowledge which each give a different perspective. This is done by the KP automatically, using the relations that were defined when creating KFs. Moreover, the KP infers relations that were not set manually by the domain expert: e.g. a *Rule* can explain the expression of *Property* directly without the *Activity* in between. Having to restrict the domain expert to only allow a Rule to relate to an *Activity* is useful to ensure that the domain expert is guided to structure knowledge correclty. However, in some cases it is better to show the directly how a *Rule* relates to a *Property* without the steps in between.

The developed knowledge perspectives all serve a different purpose and are targeted to a specific stakeholder: e.g. the Traceability Viewer is designed for a project lead to understand whether knowledge is correctly implemented in application code. However, knowledge perspectives are accessible and usable for every stakeholder! Using the relations defined in the TM, it is possible to develop new knowledge perspectives to comply to the wishes of the customer. The knowledge perspectives can be accessed via the KF tree, as seen in Figure 6.4. When left-clicking a KF, the KP loads the chosen KF in the Data Viewer. Right-clicking a KF opens up a context-menu, giving the user the choice to load the other knowledge perspectives.

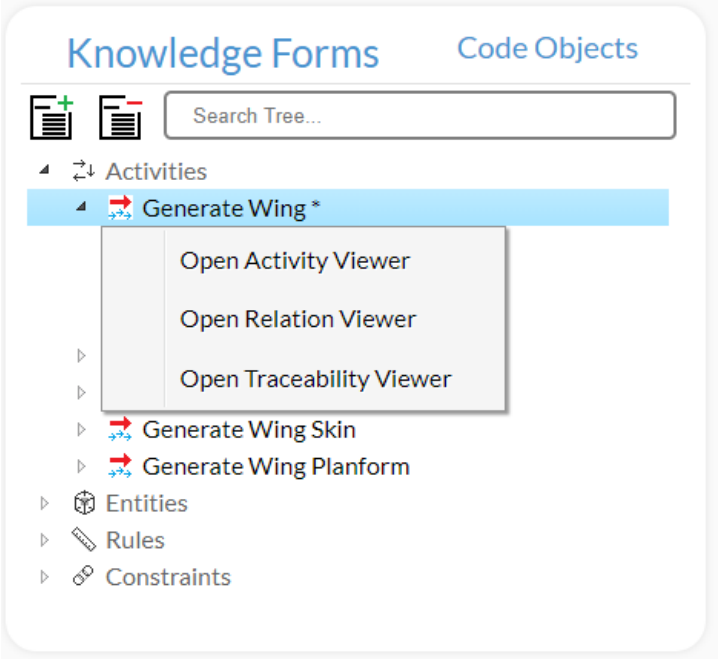


Figure 6.4: An overview of the KF tree which contains all the KFs that were created. One may note that the *Properties* are not seen in the KF tree, this is because all the *Properties* are grouped under the correct *Entity*. By right-clicking a KF a context-menu opens up to allow the user to load the chosen KF in one of the knowledge perspectives. By left-clicking a KF, the chosen KF opens up in the Data Viewer.

The next sections discusses each developed knowledge perspective and explains a few aspects:the navigability, the targeted stakeholders, and the benefit of using it. This should help understand how the KP can be used in a way that helps the stakeholders to develop a KBE application.

6.2.1. Data Viewer

The Data Viewer is the only knowledge perspective that can be accessed by left clicking on a KF. This is the default knowledge perspective and shows what has been filled in by a domain expert: a user can view the data. From this viewer, it is also possible to modify entries. When a KF that has already been connected to application code is being modified, a warning is issued. This is because a change in the KF might require a change in the application code. This is required to ensure that both the KF and the CO contain the same knowledge. The Data Viewer can be seen in Figure 6.5.

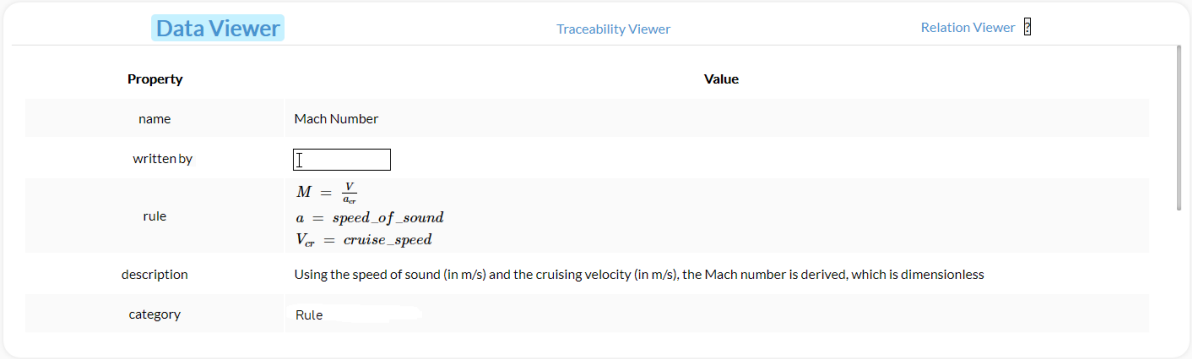


Figure 6.5: UI of the Data Viewer. It gives an overview of the information captured about the chosen KF. It is possible to modify fields by clicking on it, as can be seen for the attribute 'written by' which is being modified.

Navigability

By left-clicking on a KF in the KF-tree, the KP loads the chosen KF in the Data Viewer. This is the easiest way to get to the Data Viewer. Another way to get here is to click on a KF when looking at the

Traceability Viewer or Relation Viewer. This is possible to allow the user to dive deeper into what the KF exactly is when investigating the other knowledge perspectives.

Targeted Stakeholder

The domain expert uses the Data Viewer to understand KFs and to make modifications to it. It provides the domain expert with a compact view of the KF and the knowledge that has been captured. Accessing this viewer is as simple as clicking on the KF in the KF tree. The domain expert also uses the Data Viewer to find out where the knowledge comes from, whether it is up-to-date, and to verify that it is correct. Any mistakes found can be redirected to the author, which can be found from the 'written by' field.

The developer also uses this viewer to understand the KF when they have to codify it. It also gives them the possibility to find which domain expert wrote it in case it is not clear. In the case that the KF is a Property, the developer can connect it to COs to enable the traceability of knowledge into application code. This can be done by scrolling down to 'Related Code Object' and clicking on the '+'. This opens up a similar window as seen in step 3 of Figure 6.3.

Benefit

From this view it is possible to understand the KF in detail, but also find out information about where the knowledge comes from, who has created it, whether it is up to date, and other managerial aspects. It is also possible to modify the KF if required, or relate it to COs. The main goal is to be a simple and accessible way to understand and modify KFs.

6.2.2. Relation Viewer

The relation viewer shows the relations of a chosen knowledge form to other knowledge forms (in colour) and code objects (in gray). It provides a birds eye overview of how it is used by other elements. It is possible to navigate through the web of relations, and also allows to navigate to the data viewer or to the traceability viewer. This knowledge perspective is used by users that want to understand how knowledge is related. The relation viewer can be seen in Figure 6.6. Take note that no *Activities* are shown in this viewer; this is because it was thought unnecessary to show the *Activities* here as they are already shown in the Activity Diagram. The relation between the Rule and the Property is inferred by the KP, as it is not define manually by the domain expert.

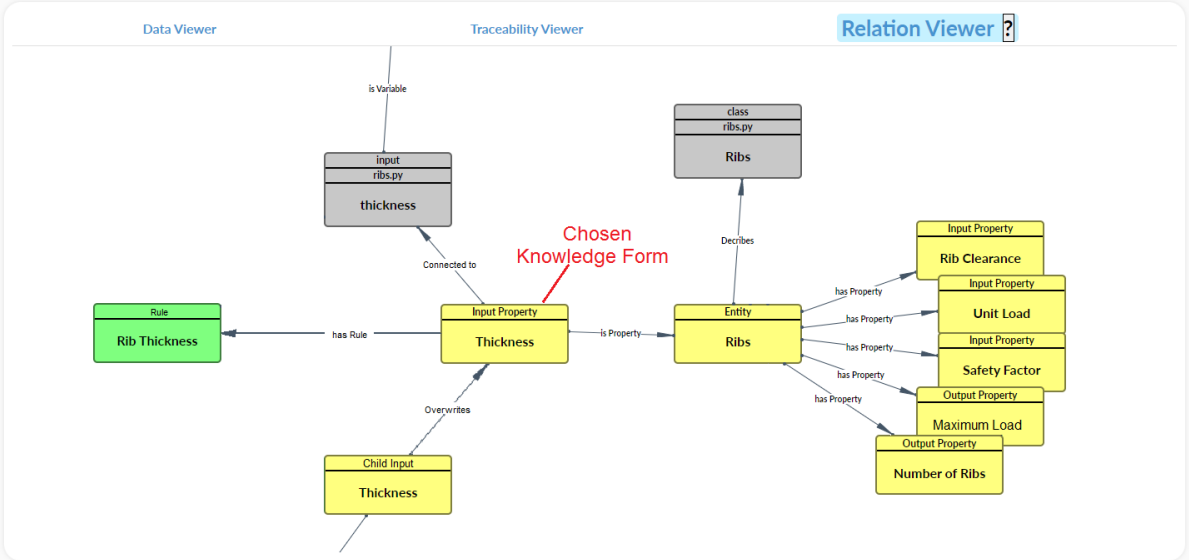


Figure 6.6: UI of the Relation Viewer showing KFs and COs as blocks, where the grey blocks are COs. It shows the relation a chosen KF or CO has with other KFs and COs. It is used to get a clear overview of how KFs and COs relate to each other. Left-clicking a block focusses on its relations, while right-clicking a block opens the Data Viewer of the chosen KF.

Navigability
The Relation Viewer can be accessed by opening the context menu of a KF in the KF tree and clicking on 'Open Relation Viewer'. The user is also able to navigate through the KFs by double clicking one of the KFs or COs shown. This makes it possible to navigate through the vast amount of KFs. By single clicking on a KF or CO, it opens and focusses in the Traceability Viewer (focused means it scrolls automatically to the correct CO, centres it, and highlights it).

Targeted Stakeholder
The Relation Viewer is targeted to any stakeholder. All the stakeholders can use this knowledge perspective to understand how a certain KF is related to other KFs. It is also possible to find out to which COs it is related, giving insights into where it has been implemented.

Benefit
The benefit of the Relation Viewer is that it gives a vast overview of the relations a KF has. An example is when developing a Property in application code this viewer helps understand whether there is a constraint affecting it and which rule provides the expression. It is also possible to see in one overview which Properties an Entity has.

6.2.3. Activity Diagram

The activity diagram is a visualisation of the Activity forms put into order of execution. The order is determined by the domain expert when creating a KF. Figure 6.7 shows the activity diagram.

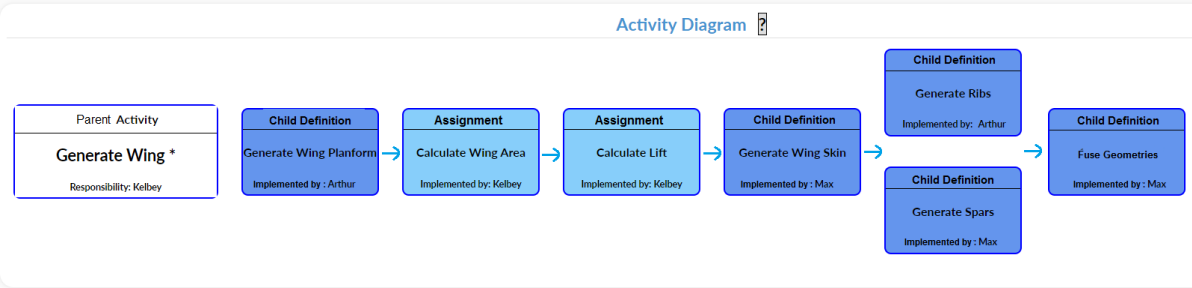


Figure 6.7: UI of the Activity diagram, showing the Activities as blocks. The dark blue blocks are Child Definitions while the light blue blocks are Assignments. Double-clicking a Child Definition loads the Activities of the Entity it is returning indirectly (*Child Definition* sets *Child* returns *Entity*). Left-clicking a block loads and focusses it in the Traceability Viewer, showing where the Activity is taking place.

Navigability
The user can load the Activity Diagram by opening up the context menu of a KF in the KF tree and selecting 'view activity diagram'. It loads the Activity tied to that KF. If the chosen KF is not an Activity, the correct Activity is inferred: e.g. if a Property is chosen, the KP will determine with which Activity it is related.

Double-clicking a Child Definition navigates the user to a deeper level, as this Activity relates to another Entity with its own activity flow. This is useful for users that want to understand the design flow.

Once application code has been developed, parsed, and connected to KFs, it is possible to click on the Activity in the Activity Diagram to show where it is in the Traceability Viewer. This is useful for users that want to see how the Activity is implemented in code.

Targeted Stakeholders
The developer uses the Activity Diagram when developing new code or trying to understand how a previous project has been developed. The Activity Diagram helps understand what the steps are in the application design process, and it helps the developer understand where in the design process the chosen KFs are located.

A project lead uses the Activity Diagram to verify whether the steps in the design process has been implemented correctly. Using the Activity Diagram, the project lead can browse through the different layers in the design process. This is possible as a *Child Definition* returns an Entity; so when a *Child Definition* is double clicked, all the Activities that are related to the *Properties* of that *Entity* are loaded

and put into order. This gives an idea how the *Activities* are taking place in that *Entity*. Furthermore, they are able to locate where the steps have been implemented in the application code by left-clicking.

The end-user uses the Activity Diagram to better understand the steps the finished KBE application is taking to derive an output. This should help an end-user gain trust in the application.

Benefit

In the scenario that the KFs still need to be codified, the developer uses the Activity Diagram to understand the steps in the design process. This helps better prepare the developer to develop a module as they have a greater understanding of the steps necessary to get to the output. In the scenario that there is already code developed, the developer can use the Activity Diagram to understand which Activities relate to which COs. This helps create a better understanding of how the developed code fits with the knowledge.

The project lead uses the Activity Diagram to take a look at each step in the design process. By clicking on an Activity it load in the Traceability Viewer, allowing the project lead to directly see how the design step is implemented in the code. This direct connection should help the project lead navigate the vast amount of code and help him verify the correctness of the KBE application.

6.2.4. Traceability Viewer

The traceability viewer shows each CO separated and groups the KFs that are connected to it. This separating and grouping is done by the KP. It provides the best overview to show what knowledge is implemented in a CO. Moreover, it allows the user to print a document summarising all the KFs connected to the COs, essentially generating documentation. The Traceability Viewer can be seen in Figure 6.8.

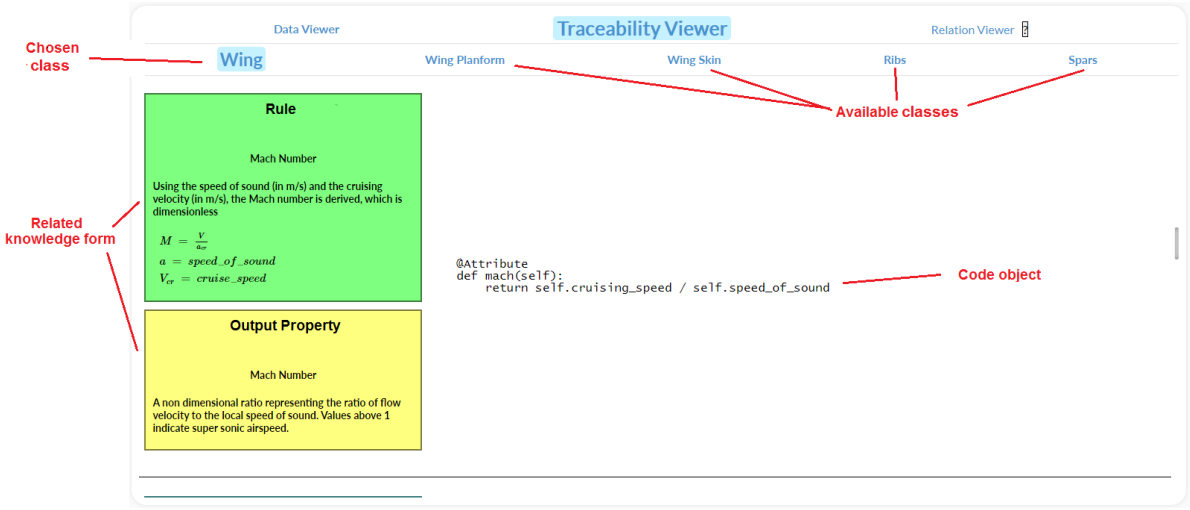


Figure 6.8: UI of the Traceability Viewer. The KP separates each CO from a `Class` and groups the KFs that are connected to it. This provides the user with an overview of which KFs are connected to the CO. As can be seen, no Activity is connect to the CO. This choice has been made as the Activity already has its own Activity Diagram in a separate window. By clicking on an Activity in the Activity Diagram, the corresponding CO is loaded, focused, and highlighted.

Navigability

The Traceability Viewer can be accessed from almost all other knowledge perspectives. This was chosen as it provides the user with the best overview to understand how and where knowledge is implemented in application code. From the KF tree or the CO tree the user can open a context menu and click on 'Open Traceability viewer'. From the Activity Diagram the user can click on an Activity to load and focus it in the Traceability Viewer. From the Relation Viewer, the user can click on any KF or CO to load and focus it in the Traceability Viewer.

From the Traceability Viewer, users can also click on a KF or CO to open it up in the other knowledge perspectives.

Targeted Stakeholder
In the scenario that application code is being re-used for a new application, the developer uses the Traceability Viewer to understand how the application has implemented the knowledge.

The project lead uses the Traceability Viewer to determine whether knowledge has been implemented correctly in the application code. An example of a scenario is when a project lead would like to find out whether a constraint is implemented correctly. This can be done by searching the constraint in the KF tree, opening its context menu and clicking on 'Open Traceability viewer'. The project lead directly sees whether it is implemented, and if so, whether it is correctly done.

Benefit
This knowledge perspective takes advantage of the fact that knowledge is now connected to code. It presents a great overview of how all the knowledge captured is implemented in the code. Moreover, the Traceability Viewer is navigable from all the other knowledge perspectives. This ensures that users are able to pinpoint exactly what knowledge is implemented where, and how it is implemented.

Where the Activity Diagram gives the user an overview of the design steps, the Traceability Viewer gives the user an overview of how KFs are implemented. Together, they provide the user with a clear understanding of the knowledge and the application code.

7

Experimental Results

To test the effect the KP and the TM has on the development process of a KBE application, two experiments have been set-up. Ideally, each stakeholder would be tested, but this was not possible due to time constraints. The two stakeholders being tested are the developer and the project lead. The developer was chosen to be tested as it was deemed important test whether the KP would better prepare developers to develop application code, and to see whether the KP helps in a collaborative environment. The project lead was chosen as it gave a perfect opportunity to test the transparency of KBE application when using the KP compared to without. The experiment tests whether a project lead is more able to find how knowledge is implemented and whether it is implemented correctly.

7.1. Experiment Set-up: Develop Application Code

Purpose

This experiment tested whether the KP helps to better prepare a developer to develop a KBE application. It also tested whether the KP helps a developer in a collaborative environment, as they would need to understand application code that has been developed by another developer.

Participants

A good fit for this experiment is an engineer who is experienced in developing KBE applications with ParaPy. This narrowed down the search to customers of ParaPy, ParaPy developers themselves, and MSc Aerospace students from the TU Delft who have experience with the KBE course (which uses ParaPy). 10 participants were found.

Scenario

The following scenario has been given to the participants:

"You have joined the development team of the Wing Design project. There are four existing modules of code already, the main module is Wing.py. There are three other modules already implemented, one to generate ribs (Ribs.py), one to generate spars (Spars.py), and one to generate Wing Skin (WingSkin.py). The following image shows what the application is capable of already:

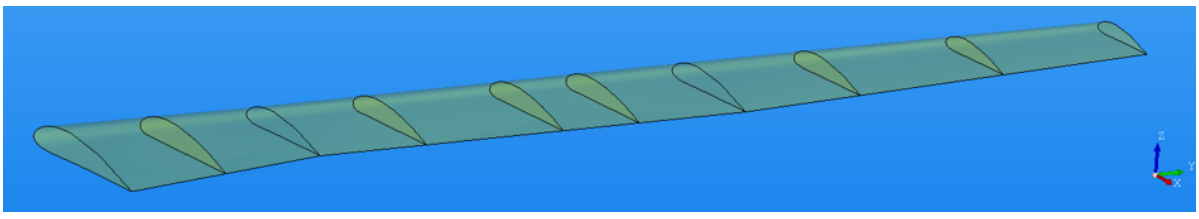


Figure 7.1: An isometric view of what the application can currently generate.

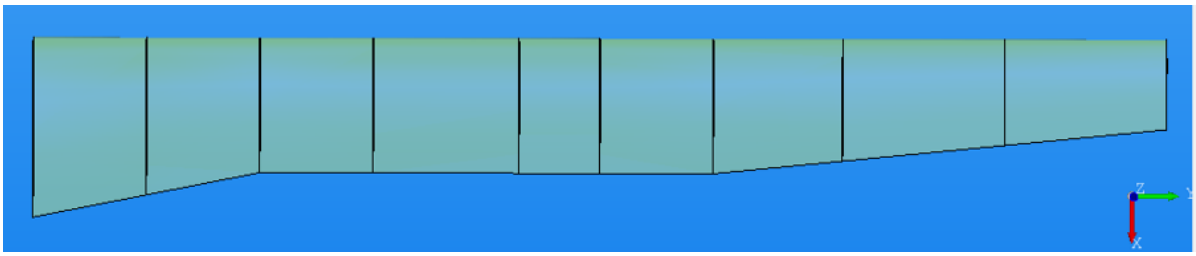


Figure 7.2: A top view of what the application can currently generate.

Currently, a basic wing planform can be generated from two Inputs: section boundaries (e.g. the root and the tip, if the list is more than two there is a kink), and the chord sizes that belong on each section boundary.

A wing planform is a 2D outline top-view of a wing; however, in the application no wing planform outline is generated, merely the locations of the section boundaries are calculated; which, together with chord sizes, is enough to infer what the shape of the wing will be. This makes it possible to generate a wing geometry.

Your task would be to replace these two inputs with a new feature that will generate a (more complex) wing planform that complies to the rules and constraints given to you. After you have implemented these new features, you need to investigate whether other classes are affected by the changes and which action steps are necessary to make the correct changes."

Task

The participants were tasked to develop a new module. Moreover, the new module would be replacing existing COs, so the participants were asked to figure out whether the existing code would be influenced by the new module, and if so, how.

It was not possible to ask the participants to develop the whole application code as this would take too much time. Therefore, the participants were merely asked to write down which steps are necessary to develop the new module as actions steps. Examples of action steps are:

- Create a new class 'Wing Planform'
- Add a @Part called 'wing_planform' into class Wing and pass the three inputs 'span', 'chord_sizes', and 'boundary_locations '
- Add an @Attribute called 'kink_location' to calculate the kink location, minimum location from the root

It was important that the participants to write down the following about a code object: its type (numeric, a list, a dict), in which class it should be located, which other code objects it is using, whether it uses a rule (e.g. equation), and whether it is constrained. By writing down all these aspects, it could be determined whether the participants understood correctly how the knowledge should be implemented.

Groups

There were two groups during this experiment. The test group made use of the KP, where all knowledge had already been structured into KFs. All the relations between KFs had been defined as well, this means that the participants could use the Data Viewer, Relation Viewer, and Activity Diagram. The test group participants were also asked to take time to understand the TM, as this would help them understand how classes of KFs should be developed into certain types of COs: e.g. an *Output* should be coded into an `Attribute` as defined by the relation between the Knowledge Ontology and the ParaPy Ontology.

The control group would not be allowed to use the KP. Instead, all the knowledge had been structured into a single Word document. This Word document was made to resemble a summary of an interview with a domain expert. It is important to note, that both the Word document and the KP contained *all* the same knowledge and images, as to make it fair.

Assessment

The experiment tested how well prepared the participants were to develop a piece of application code. The hypothesis was that the KP better prepared the participants. The idea is that if a participant scores high, they know the knowledge of the application domain better, they understand what needs to be implemented better, and they make less mistakes. To correctly assess the performance of a participant, a scoring sheet was prepared. This was done so that after the experiment it was possible to tick off the correct steps decided by the participants and award points. The total number of points for this experiment was 50 points. The scoring sheet can be seen in Figure 7.3.

#	Answer	Pts	Score
	Code Management		
1	Create a new class called Wing Planform, add a Part to the class Wing to instantiate the class	2	
2	Move the section boundaries, chord_sizes, and span to the new class	3	
3	Add a validator for the section_boundaries and chord_sizes ensuring it is the same length as the number of sections input	3	
4	Pass down cruising_speed to the class Wing Planform	1	
	Developing New Module		
5	Add an input number_of_sections to class Wing Planform; add a constraint to keep value larger than 1	4	
6	Add input speed of sound in class Wing and pass down to class Wing Planform	3	
7	Add attribute Mach in class WingPlanform with correct rule	3	
8	Add attribute leading edge sweep in class WingPlanform with correct rule	3	
9	Add a constraint to leading edge sweep of a minimum of 10 degrees	2	
10	Add a constraint preventing subsequent chord sizes to be larger than previous one (from root > tip)	4	
11	Add an attribute calculating new leading-edge positions of section boundaries	4	
12	Add an attribute that recalculates position of boundary locations due to inner section having no trailing edge sweep	2	
13	Explain that the main output of this class is a list containing x, y coordinates of section boundaries; or something similar	4	
	Effect on Existing Classes		
14	Explain that class Spars and class WingSkin must be modified	2	
15	Replace section_boundaries with the output of class WingPlanform.	2	
16	Modify part placed_airfoils to take output of class WingPlanform and uses it for both the x and y location (as it is a 2D array now)	4	
17	Pass down output of class Wing Planform to class Spars as an Input called boundary_locations	2	
18	Either: Add an attribute to help with part spar_planes Or: Modify spar_planes normal vector to fit swept wing	2	

Figure 7.3: The scoring sheet used to assess the performance of the participants. The total number of points is 50. The points were given based on each aspect of a code object: e.g. an attribute put into a class that has a rule and also a constraint is worth 4 points as it considers 4 aspects.

A participant scores good if they are able to understand what the structure of the new module needs to be. They also write down clearly what the name of the code objects are, what their type is, whether there is a rule defining what the expression is, and whether there is a constraint. An example of a good

action step that has been seen:

"Create a new attribute called 'lift' in the class 'WingPlanform' that will return a numeric value, it takes 'weight and 'safety factor' as inputs and the expression is equal to lift = weight x safety factor. There is a constraint posing a minimum value for the lift of 1000N."

A participant scores low if there are holes and gaps in the code: when not everything is implemented. A participant also scores bad if the action plan is missing details and if it missing constraints or rules explain the rule. An example of an incomplete action step that was seen is:

"The lift is calculated in the Wing Planform class"

7.2. Experiment Set-up: Verify Application Code

Purpose

The experiment tested whether the participants are better able to understand how the knowledge is implemented in the application code using the KP compared to without. It also tested whether the participants using the KP are able to find the pieces of code necessary more efficiently compared to without the KP. These two aspects should show whether the KP helps increase the transparency of KBE application by enabling a direct connection between knowledge and application code. This is due to the fact that a transparent application is an application that is easy to understand, and one that makes it clear what it is doing, and how.

Participants

A good fit for this experiment is an engineer who is experienced in overseeing projects using the ParaPy software. The only available participants were the project leads of customers of ParaPy. This made the search difficult to find suitable participants. Moreover, many of the participants had limited availability as they were very busy. Nonetheless, it was possible to perform the experiment with 6 participants, all from the industry.

Scenario

The following scenario has been given to the participants:

"You are the project lead of the Wing Design project. The project has been through all the development phases: knowledge acquisition, knowledge structuring, and application development. The project is now nearing its delivery, but before that can happen you have to ensure that the application has been developed correctly: you will have to verify it. It will not be possible to test the whole application, so you will only be asked to check whether constrains have been implemented correctly and whether some functional requirements are met.

The application generates a wing with ribs and spars. The wing planform is an important part of this project, as the wing planform is a 2D top view which determines the shape of the wing. The following image shows an example of a wing planform.

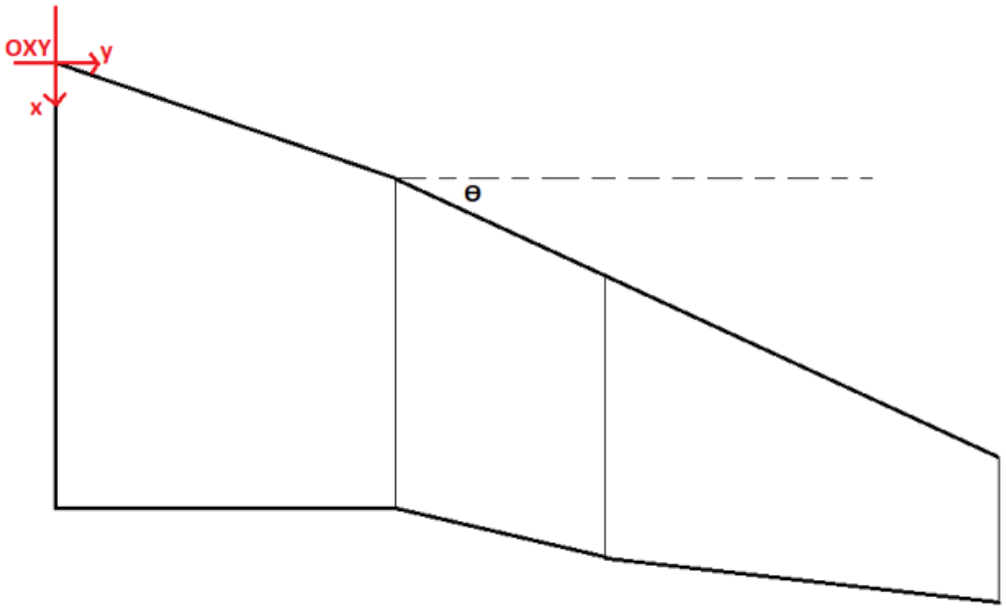


Figure 7.4: A top view of a wing planform. It captures the shape of the wing, it also shows a sweep angle.

You will receive an excel file containing the functional requirements and the constraints. You have to find out whether they have been implemented correctly in the application code. You must also note which code object the requirement or constraint is applied to. If something is implemented incorrectly, you need to come up with a plan to ensure that it will be fixed as soon as possible, as the delivery deadline is coming up soon! ”

Task

The participants browse through the application code and test whether the functional requirements and the constraints have been implemented correctly. Three fields have to be answered per requirement or constraint: whether it has been implemented correctly, what the name is of the CO that implements it, and what has to be done to ensure it can be fixed as soon as possible if it is not implemented correctly.

Groups

Similarly to the developer experiment, there are two groups. The test group made use of the KP, where all knowledge has already been structured into KFs, all the application code is already parsed into COs, and all the KFs are connected to the correct COs. The participants can use all the functionalities of the KP to perform the experiment, but especially the Traceability Viewer in combination with the KF tree isthe main point of interest. This is because the user can search for *Constraints* in the KF tree, and click on ’Open Traceability Viewer’ to directly see how the *Constraint* has been implemented.

The control group are handed the raw application code and the same Word document as the previous experiment which contains all the knowledge necessary.

Assessment

The experiment tested how well a project lead can navigate the code and understand how constraints and functional requirements are implemented. To test this, the participants received an Excel file with constraints and functional requirements that should be implemented in the application code. They can be seen in Figure 7.5. In total, 63 points can be scored, one for each correct answer.

		Fulfilled?			
Req	*Wing*	Yes	No	Code Object	Plan of action
1	The end-user must be able to choose the cruising speed				
2	The wing must consist of at least two sections				
3	The end-user must be able to modify airfoil sections on each section boundary				
4	In case of two or more invalid ribs, the safety factor must be increased from 1.2 to 1.4				
5	There must be a minimum of 2 spars				
6	The minimum lift should be 300,000 N (to be able to lift a 30 ton aircraft)				
7	The end-user must be able to choose to span of the wing				
8	Minimum Airfoil Detail is required, meaning the airfoil .dat files must consist of at least 80				
9	There must be a minimum of two ribs, one at the root and one at the tip				
10	The end-user must be able to choose the number of sections the wing will have				
11	The trailing edge sweep of the first section must be 0				
12	A safety factor of at least 1.2 should be used for ribs and spars				
13	The spar placement must be within a certain area away from the leading edge and trailing edge				
15	The end-user must be able to choose the chord sizes of each section boundary				
16	The leading edge sweep of the outer wing sections must be larger than 10				
17	The ribs must have a clearance around section boundaries				
19	The wing skin is assumed to have no thickness				
20	The end-user must be able to choose the locations of each section boundary				
21	During Rib Placement, invalid ribs must be repositioned to the nearest available location outside of the clearance				

Figure 7.5: The scoring sheet used to assess the performance of the participants. The total number of points is 63. The points were given based on each correct answer whether a code object was fulfilled, which code object it was, and what the action plan was.

A participant scores well if they are able to locate the code object that is implemented the constraint or requirement. If it is implemented incorrectly, they are able to assess that and come up with a plan to fix it as soon as possible. An example of a good answer for a constraint is:

"Constraint not fulfilled. Code object name: section_boundary. Code developed by Arthur while the knowledge comes from Kelbey: plan of attack is to contact Arthur why section boundary is not at least a list of 3 entries, and contact Kelbey whether this is a mistake in the knowledge form."

A participant scores bad if they are not able to find the code object.They also score worse if they wrongly identify whether the knowledge is implemented correctly. Moreover, participants often lacked to give a plan of action to fix the mistake. An example of an incomplete answer that was found is:

"Constraint not fulfilled. Code object name: section_boundary. There is no minimum."

7.3. Results

The results of the experiments can be seen in Figure 7.6, as percentage scored of the total points possible. As there were not many participants available for the experiments, there is a large margin of error: for the 'develop application code' experiment 27%, while for the 'verify application code' experiment 31%. These margin of errors are both for a confidence of 90%. This means that it can be concluded that there is a 90% certainty that the average results are in the range of the errors.

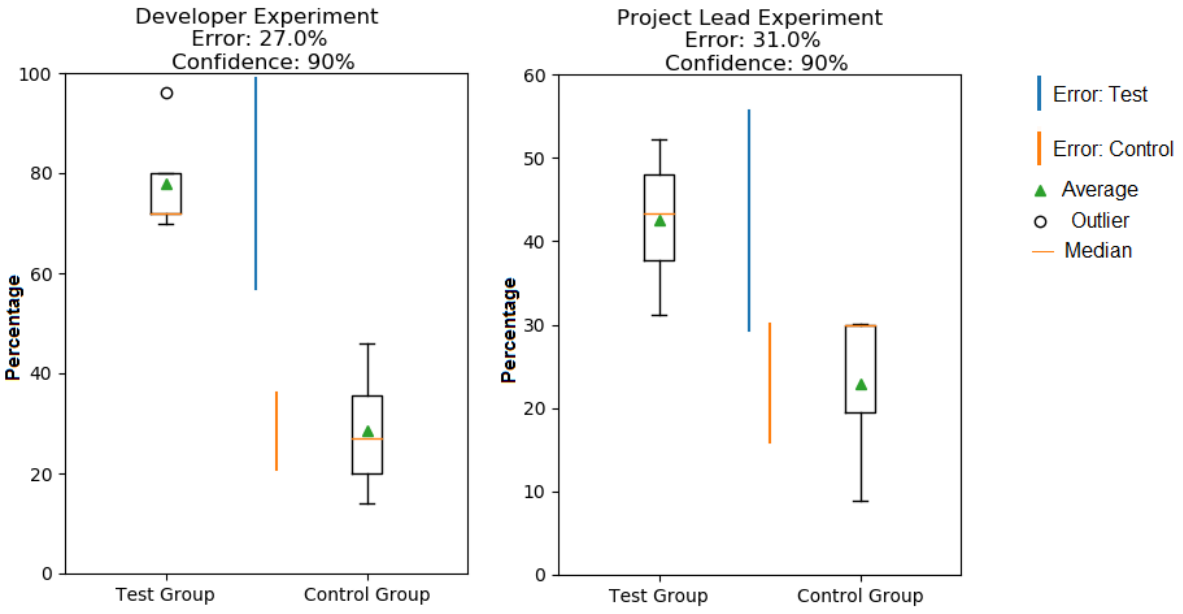


Figure 7.6: Both results show the points scored as percentage of the maximum points possible. Both experiments show that the test group scored higher than the control group. For both cases there is an error as there were not many participants. However, even with the error the results show an increase in score.

7.4. Discussion

This section will discuss the results of the experiments, starting with the developer use case. Then, the project lead use case will be discussed. Finally, some general observations that was noticed during both experiments will be discussed.

7.4.1. Developer Use Case

The left diagram of Figure 7.6 shows that the test group scored on average 2.8 times better than the control group. In reality this value is somewhere between 1.6 and 4.8 due to the margin of error. Interesting to note, is that a participant in the test group scored 48/50 points, which was almost perfect.

The results show that participants using the KP were better able to understand how the existing application code worked, and how the knowledge was implemented. To understand the existing code they mainly used the Activity Diagram to understand the steps in the design process to generate a wing. Clicking on an Activity brought them to the Traceability Viewer where they could directly see what knowledge was implemented into which COs, and how it was implemented. With a good understanding of how the application was already built up, they proceeded to think about how the new module had to be implemented.

As the test group was using the TM, they knew that the new module, which was captured in an *Entity*, had to be implemented in a `Class`. The Relation Viewer was then used as it showed exactly which *Properties* the *Entity* 'WingPlanform' had. This meant that the participants knew which `Inputs` and `Attributes` were necessary. Furthermore, the Relation Viewer helped the participants understand whether there were *Constraints* acting on the *Properties*, as this could directly be seen in the Relation Viewer. *Constraints* played a large role in this experiment, so the Relation Viewer proved to be very helpful. Finally, they were able to understand how *Properties* were used for the expression of other

Properties, as the Relation Viewer also shows which *Activities* and *Rules* applied on the *Property*.

The control group had to use the Word document to develop a new module. The first difference was already seen when they tried to understand the knowledge, which was more difficult as it was not structured yet. It was also difficult to understand exactly how the wing planform had to be developed, as it was not clear exactly how everything related to each other. In some cases, probably also due to the limited time available, they provided the completely wrong expression. It also depended on the participant whether they were able to effectively distil the important parts of the story for the module. A few participants had a clear tactic and noted down each possible CO they encountered, while others had difficulties getting through the text and had to read it over and over. It was also difficult for the control group to understand how the existing application code was developed, as there was limited documentation in the application code. This made it especially difficult to correctly come up with action steps that were targeted to pre existing application code.

Moreover, participants from the control group mentioned that the Word document that was provided was more structured than what they are usually handed when they have to develop a KBE application. They mentioned that sometimes they were not even handed a Word document, but given folders of documents and they had to find their own way through all that knowledge. This means that the quality of the Word document was high, and this further enforces the meaning of the results.

7.4.2. Project Lead Use Case

The right diagram of Figure 7.6 shows that the test group scored on average 1.9 times more than the control group. Due to the margin of error, the actual improvement is between 1.0 and 3.5 times.

The participants that were using the KP had some troubles at the start of the experiment with understanding exactly how the KP works and what the functionalities are. Therefore, it seemed that a large portion of the time was needed to get used to the KP. It took a while for the participants to find the search button, which would allow them to search the required KF in the KF tree. Some participants knew of the search button, but still preferred to scroll through the KF tree rather than searching for it. Once the participants understood that you can directly load a KF in the Traceability Viewer, the requirements and constraints were verified very efficiently.

When coming up with an action plan for the requirements that were not met, the participants were able to determine easily what that piece of code should have done and what is still lacking. They were also able to find who wrote the piece of code, and who created the KFs concerning that piece of code. This made it possible to come up with a concrete plan to fix the problem.

The results of the control group was interesting as the participants all showed different behaviours. One participant was pretty comfortable in the coding language and could find the required code pretty easily. However, this participant still did not manage to score higher than any participant of the test group. It was also mentioned that they would probably appreciate the KP more if the experiment was performed for a larger application. As the application only consisted of five modules, it was rather easy to find what you were looking for if you know how to navigate through application code. However, there was one participant who was less skilled in programming who was not able to navigate rather easily through the code. This participant only managed to find their way to 3/21 requirements, and when a requirement was found they would wrongly assume that it was implemented correctly.

7.4.3. Observations

The different knowledge perspectives helped the participants understand the knowledge embedded in the KBE application better, and it helps understand how knowledge is implemented due to the direct connection with application code. However, the participants did mention that there was one important knowledge perspective missing: a UML Class diagram. Such a diagram would show the hierarchy of the various objects and would give them a sense of how the product was built up.

It took the participants 10 minutes of getting used to the KP before they could use the KP to their advantage. Moreover, the User Interface (UI) of the KP was still very primitive and did not help to give the best overview possible. The User eXperience (UX) of the KP was also not the best. This was the case as the KP was designed to allow a user to move fast from one view to another using short cuts and clicking with either the left, middle, or right mouse button. However, all these short cuts and different types of clicks made it too difficult to navigate the KP effectively. It seems that having simple buttons and a single type of click to navigate is better than having too many different possibilities.

The set-up of both experiments could have been better to gather better results. A single participants

should have both participated in the test group as well as the control group. This would eliminate bias of some participants being more skilled than others. For example the participant that scored 48/50 points in the first experiment might have scored high as well if he was in the control group. This was also seen in the second experiment, where a participant who knew how to develop code could easily navigate the application code while others could not. However, having a more elaborate experiment set-up was not possible due to time-constraints of the thesis, and due to the fact that the participants were available for a limited amount of time.

8

Conclusion and Recommendations

The aim of this research was to develop a methodology that enables the traceability of knowledge onto Knowledge Based Engineering (KBE) application code without the need of specialized knowledge engineers. The necessity became clear as KBE applications are perceived as a black box, which decreases the trust in KBE applications, hinders systematic knowledge re-use, and lowers the participation of domain experts. Having a direct connection between knowledge structured in models and the application code would improve the transparency. However, there was also the problem that knowledge is not modelled correctly, hindering the ability to connect knowledge to application code.

8.1. Conclusion

MOKA is one of the well-known KBE methodologies, and it proposes to structure knowledge into several models that are proven to be very effective to model knowledge. However, they require specialist knowledge engineers which are not available at many companies. MOKA has also been developed with the assumption that the engineer who owns the knowledge, the domain expert, is a different person than the engineer who develops the application, the developer. These knowledge engineers structure knowledge in two steps. The results of the first step is an informal model that can be reviewed by the domain expert. Only when the first step is reviewed, the knowledge engineer proceeds to the second step to structure knowledge into a formal model that is understood by a developer.

However, a field study performed during this research showed that a single engineer fulfils multiple roles. These engineers are often engineers with a technical background, but also have basic IT skills that were taught during their studies. This meant that the person developing a KBE application is often the domain expert as well as the developer. However, this contradicts MOKA's assumption, where a knowledge engineer was required to help bridge the gap in understanding knowledge between the domain expert and the developer. Now that the domain expert is also the developer, the knowledge engineer is not required any more.

What is seen, is that this gives developers the feeling that knowledge modelling is not necessary as they feel they might as well directly implement their knowledge into application code. This is possible, but without correctly modelling knowledge the black box perception still remains a problem which causes many side effects that affect the overall quality of the developed KBE application. Therefore, it is important that knowledge modelling is still performed. Therefore, a new ontology was developed which would motivate knowledge modelling and define a connection between knowledge and application code.

The new ontology is called the Traceability Model (TM). The hypothesis that domain experts are able to structure knowledge themselves in a formal matter made it possible to develop the TM more formally than the informal model proposed by MOKA. The formal model is removed as to remove the necessity of knowledge engineers. The main change made to the informal model is the addition of *Properties* which enables a domain expert to capture the essence of an Object Oriented (OO) modelling approach. This allowed the knowledge to be directly connected to application code. This is possible as the TM defines two ontologies: a Knowledge Ontology and a ParaPy Ontology. The Knowledge Ontology defines different classes of Knowledge Forms (KF) and are related to each other. A KF is a

form that must be filled in by a domain expert, which guides them to structure knowledge correctly. The ParaPy Ontology defines different types of Code Objects (CO) that can be found in the ParaPy coding language. The TM defines a connection between the Knowledge Ontology and the ParaPy Ontology, which defines how knowledge can be connected to application code.

Once the TM was developed, the Knowledge Portal (KP) had been developed to provide a platform to structure knowledge and directly connect knowledge to code. Moreover, the KP automatically generates knowledge representations, to help understand the knowledge that is captured in KFs. The goal of the KP was to assist the various stakeholders with their tasks, the following should be possible: domain experts can create KFs to structure knowledge, developers can directly understand how knowledge has to be develop into code, project leads can directly see how knowledge is implemented in the code to see whether it complies to requirements, and the end-user can use the KP to understand how the developed KBE application generates an output from some input.

Three goals and a research question were formulated at the start of this thesis. To test whether the goals are met, and to answer the research question, two experiments had been performed. The experiments focussed on the development of a KBE application and checking whether it complies to requirements. The experiments tested two groups, where one would be able to use the KP. The participants were assessed based on a scoring sheet, which allowed them to score points based on their performance.

The first experiment focussed on whether developers using the KP are better prepared to develop application code in a collaborative environment. The results show that using the KP increases the average points scored by 2.8 times. It can be concluded that the participants understood the knowledge of the application domain better, and they are more able to understand how existing application code is written and how it implements knowledge.

The second experiment focusses on whether project leads using the KP can better verify whether the developed KBE application complies to requirements. The results show that using the KP increases the average points scored by 1.9 times. It can be concluded that the KP improves the traceability of knowledge into application code, as the participants using the KP were able to pinpoint precisely where the requirements were location. They were also better able to understand the application. It can be concluded that improving the traceability of knowledge into application code has improved the transparency of the KBE application.

From this research, it was concluded that KBE applications are perceived as a black box. Adoption of KBE methodologies is hindered by the fact that it requires a knowledge engineer. The hypothesis is that the TM removes the necessity for knowledge engineers; however, this still has to be tested with an experiment. Once knowledge is modelled, it is possible to directly connect it to application code. This was achieved by the TM and made possible with the KP. The KP further helped to improve the transparency of the application by generating knowledge representations that helped understand the knowledge embedded in the application. The two experiments validated that both the developers and the projects leads scored higher using the KP compared to without, proving that it increased the transparency of the KBE application.

8.2. Recommendations

It still remains unclear whether the TM remained simple enough to allow the structuring of knowledge without the necessity of knowledge engineers. The hypothesis is that it does remove the necessity for two reasons: it is assumed that domain experts also fulfil the role of the developer so they understand the Object Oriented modelling approach and are capable to structure knowledge correctly, and the KP guides the domain expert to capture what is necessary and restricts any impossible relations ensuring that the structured knowledge is correct. However, this hypothesis should still be tested by having domain experts from the industry model knowledge using the KP.

Furthermore, the TM currently only allows to capture rather generic types of knowledge, with the exception of *Properties*. When looking at *Rules*, there are many types of *Rules* that could be define. The same is true for Constraints and Activities. By having a more complete typology of these classes it will be possible to better guide domain experts to structure knowledge properly.

Moreover, the ParaPy Ontology assumes that a Module only comprises `Classes`. In reality, a Module can consist of other types of Code Objects such as functions. However, these have been neglected as the choice is made to focus the ParaPy Ontology on the `Class` as it contains most

knowledge in a ParaPy application.

Moreover, attempts can be made to broaden the functionalities of the KP to further aid in the development of KBE applications. An example is to automatically generate skeleton code, which will save a lot of time of the developers as it is pieces of code that is recurring a lot. Another functionality would be to automatically check for inconsistencies between the application code and the created KFs. This should further increase the trust in the developed application, as there still is a problem of knowledge being out-of-sync with what is implemented in the application code.

Bibliography

[1] M. S. Abdullah, A. Evans, I. Benest, R. Paige, and C. Kimble. Modelling knowledge based systems using the eXecutable Modelling Framework (XMF). *2004 Ieee Conference on Cybernetics and Intelligent Systems, Vols 1 and 2*, pages 1055–1060, 2004. doi: 10.1109/ICCIS.2004.1460735.

[2] P. Bermell-García. *A Metamodel to annotate Knowledge Based Engineering codes as enterprise knowledge resources*. PhD thesis, Cranfield University, 2007. URL http://medcontent.metapress.com/index/A65RM03P4874243N.pdf.

[3] P. Bermell-Garcia, W.J.C. Verhagen, S. Astwood, K. Krishnamurthy, J.L. Johnson, D. Ruiz, G. Scott, and R. Curran. A framework for management of Knowledge-Based Engineering applications as software services: Enabling personalization and codification. *Advanced Engineering Informatics*, 26(2):219–230, 2012. ISSN 14740346. doi: 10.1016/j.aei.2012.01.006. URL http://dx.doi.org/10.1016/j.aei.2012.01.006.

[4] R. Boateng. KnowledgeManagement and HR Learning. *Knowledge Management and HR*, pages 1–29, 2011.

[5] H. Boley, M. Kifer, P. Patranjan, and A. Polleres. Rule Interchange on the Web. *Reasoning Web - RWEB*, pages 269–309, 2007. ISSN 03029743. doi: 10.1007/978-3-540-74615-7_5.

[6] C. Chapman and S. Preston. Utilising enterprise knowledge with knowledge-based engineering. *International journal of computer appliactions in technology*, 28(2/3):169, 2007. ISSN 0952-8091. doi: 10.1504/IJCAT.2007.013354. URL http://www.inderscience.com/link.php?id=13354{%}5Cnhttp://inderscience.metapress.com/index/Q683254N2L653431.pdf.

[7] C.B. Chapman and M. Pinfold. The application of a knowledge based engineering approach to the rapid design and analysis of an automotive structure. *Advances in Engineering Software*, 32 (12):903–912, 2001. ISSN 09659978. doi: 10.1016/S0965-9978(01)00041-2.

[8] The MOKA Consortium. *Managing Engineering Knowledge*. Professional Engineering Publishing, 2001.

[9] D. Cooper and G. LaRocca. Knowledge-based techniques for developing engineering applications in the 21st century. *Collection of Technical Papers - 7th AIAA Aviation Technology, Integration, and Operations Conference*, 1:146–167, 2007. doi: 10.2514/6.2007-7711.

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

[11] R. Curran, W. Verhagen, and M. van Tooren. The KNOMAD Methodology for Integration of Multidisciplinary Engineering Knowledge Within Aerospace Production. *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, 48(January):1–16, 2010. doi: 10.2514/6.2010-1315. URL http://arc.aiaa.org/doi/10.2514/6.2010-1315.

[12] R. Curran, W.J.C. Verhagen, M.J.L. Van Tooren, and T.H. Van Der Laan. A multidisciplinary implementation methodology for knowledge based engineering: KNOMAD. *Expert Systems with Applications*, 37(11):7336–7350, 2010. ISSN 09574174. doi: 10.1016/j.eswa.2010.04.027. URL http://dx.doi.org/10.1016/j.eswa.2010.04.027.

[13] P.J. DeWitte. *Development and Reuse of Engineering Automation - Incremental Code and Design Documentation Generation*. PhD thesis, TU Delft, 2014.

[14] T. Dillon, E. Chang, M. Hadzic, and P. Wongthongtham. Differentiating conceptual modelling from data modelling, knowledge modelling and ontology modelling and a notation for ontology modelling. *Conferences in Research and Practice in Information Technology Series*, 79, 2008. ISSN 14451336.

[15] C. L. Emberey, N. R. Milton, J. P.T.J. Berends, M. J.L. Van Tooren, S. W.G. Van Der Elst, and B. Vermeulen. Application of Knowledge Engineering Methodologies to Support Engineering Design Application Development in Aerospace. *Collection of Technical Papers - 7th AIAA Aviation Technology, Integration, and Operations Conference*, 1(September):83–95, 2007. doi: 10.2514/6.2007-7708.

[16] I. Fan and P. Bermell-Garcia. International Standard Development for Knowledge Based Engineering Services for Product Lifecycle Management. *Concurrent Engineering*, 16(4):271–277, 2008. ISSN 1063-293X. doi: 10.1177/1063293X08100027. URL http://journals.sagepub.com/doi/10.1177/1063293X08100027.

[17] T.R. Gruber. Towards Principles for the Design of Ontologies Used for Knowledge Sharing. *Formal Ontology in Conceptual Analysis and Knowledge Representation*, 1993.

[18] Y.J. Hu. Challenges for Rule Systems on the Web Rule Interchange and Applications. *Challenges for Rule Systems on the Web Rule Interchange and Applications*, pages 1–12, 2009.

[19] F. Ibrahim. Knowledge Management Methodology: Developing a Phenomenalogical Middle-range Thinking Approach for Knowledge Management Research. *Journal of Management Research*, 9(4):110, 2017. ISSN 1941-899X. doi: 10.5296/jmr.v9i4.10790. URL http://www.macrothink.org/journal/index.php/jmr/article/view/10790.

[20] P. Jackson. *Introduction to expert systems*. Addison-Wesley, Wokingham, England Reading, Mass, 1990. ISBN 978-0-201-17578-3.

[21] D. Jodin and C. Landschützer. Knowledge-Based Methods for Efficient Material Handling Equipment Development. *Progress in Material Handling Research*, 12th IMHRC, 2012. URL http://www.mhi.org/downloads/learning/cicmhe/colloquium/2012/jodin.pdf.

[22] K. Kiko and C. Atkinson. A detailed comparison of UML and OWL. *Technical Report 4, Department of Mathematics and Computer Science, University of Mannheim*, pages 1–58, 2005. URL https://ub-madoc.bib.uni-mannheim.de/1898/1/TR2008{_}004.pdf.

[23] G. La Rocca. *Knowledge Based Engineering Techniques to Support Aircraft Design and Optimization*. PhD thesis, Delft University of Technology, Delft, 2011. URL https://repository.tudelft.nl/islandora/object/uuid:45ed17b3-4743-4adc-bd65-65dd203e4a09.

[24] G. Lapalme. Xml: Looking at the forest instead of the trees, Apr 2019. URL https://www.iro.umontreal.ca/~lapalme/ForestInsteadOfTheTrees/HTML/ch07.html.

[25] G. LaRocca. Knowledge based engineering: Between AI and CAD. Review of a language based technology to support engineering design. *Advanced Engineering Informatics*, 26(2):159–179, 2012. ISSN 14740346. doi: 10.1016/j.aei.2012.02.002. URL http://dx.doi.org/10.1016/j.aei.2012.02.002.

[26] I. Lemmens, J. Bulles, and P. R. Munniksma. Business Rules Management – an introduction. *CogNIAM Finance*, pages 1–32, 2013.

[27] J. Lutzenberger, I. Marthinusen, K. Kristensen, G. Iversen, P. Klein, O.I. Sievertsen, and G. Rutkowska. Methods for KBE related knowledge acquisition and codification. *LinkedDesign-Consortium 2011-2015*, pages 1–73, 2012. URL http://www.linkeddesign.eu/files/LinkedDesign{_}Deliverable{_}6.1.pdf.

[28] Q.C. Ma and X.W. Liu. Review of Knowledge Based Engineering with PLM. *Applied Mechanics and Materials*, 10-12:127–131, 2008. ISSN 1662-7482. doi: 10.4028/www.scientific.net/AMM.10-12.127. URL http://www.scientific.net/AMM.10-12.127.

[29] N.R. Milton. *Knowledge Acquisition in Practice: A Step-by-step Guide (Decision Engineering)*. Springer, 2007. ISBN 1846288606. URL http://www.amazon.com/Knowledge-Acquisition-Practice-Step-step/dp/1846288606.

[30] G.J. Nalepa and S. Bobek. Rule-based solution for context-aware reasoning on mobile devices. *Computer Science and Information Systems*, 11(1):171–193, 2014. ISSN 18200214. doi: 10.2298/CSIS130209002N.

[31] Negnevitsky. *Artificial Intelligence*. Pearson Education, 2005. ISBN 9788131720493. URL https://books.google.nl/books?id=8TD8RN-WXFAC.

[32] I. Nonaka. A Dynamic Theory of Organizational Knowledge Creation. *Organization Science*, 5 (1):14–37, 1994. ISSN 1047-7039. doi: 10.1287/orsc.5.1.14. URL http://pubsonline.informs.org/doi/abs/10.1287/orsc.5.1.14.

[33] E.J. Reddy, C.N.V. Sridhar, and V.P. Rangadu. Knowledge Based Engineering: Notion, Approaches and Future Trends. *American Journal of Intelligent Systems*, 5(1):1–17, 2015. ISSN 2165-8994. doi: 10.5923/j.ajis.20150501.01.

[34] G. Schreiber, B. Welinga, and R. de Hoog. CommonKADS: A Comprehensive Methodology br KBS Development. *IEEE*, 1994.

[35] N. Shadbolt and N. Milton. From Knowledge Engineering to Knowledge Management. *British Journal of Management*, 10:309–322, 1999.

[36] W. Skarka. Application of MOKA methodology in generative model creation using CATIA. *Engineering Applications of Artificial Intelligence*, 20(5):677–690, 2007. ISSN 09521976. doi: 10.1016/j.engappai.2006.11.019.

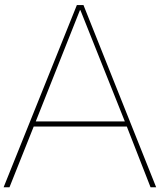
[37] J. Sobieszczanski-Sboieski, A. Morris, M. van Tooren, G. La Rocca, and W. Yao. *Multidisciplinary Design Optimization*. Wiley, 2015.

[38] J. Stjepandic, S. Rulhoff, W. J. C. Verhagen, H. Liese, and P. Bermell-Garcia. Design Process Acceleration By Knowledge Based Engineering in Automotive and Aerospace Industry. *Proceedings of the 13th International Design Conference DESIGN 2014*, pages 1915–1924, 2014.

[39] A. Teilans, A. Kleins, U. Sukovskis, Y. Merkuryev, and I. Meirans. A meta-model based approach to UML modelling. *Proceedings - UKSim 10th International Conference on Computer Modelling and Simulation, EUROSIM/UKSim2008*, pages 667–672, 2008. doi: 10.1109/UKSIM.2008.60.

[40] W.J.C. Verhagen and R. Curran. Knowledge-Based Engineering review: Conceptual foundations and research issues. *Advanced Concurrent Engineering*, 31(0):239–248, 2010. ISSN 0717-6163. doi: 10.1007/978-0-85729-024-3. URL http://link.springer.com/10.1007/978-0-85729-024-3.

[41] W.J.C. Verhagen, P Bermell-Garcia, R.E.C. Van Dijk, and R. Curran. A critical review of Knowledge-Based Engineering: An identification of research challenges. *Advanced Engineering Informatics*, 26(1):5–15, 2012. ISSN 14740346. doi: 10.1016/j.aei.2011.06.004. URL http://dx.doi.org/10.1016/j.aei.2011.06.004.



Interview Questions

- Knowledge Capture
  - Where does the knowledge come from?
  - What techniques are used to capture knowledge?
  - Where is the captured knowledge stored?
  - What is the role of the person capturing knowledge?
  - Does he need special training, such as in knowledge engineering?
  - Do you use a methodology? If yes, which? If no, why not?
  - Are there any tools used to help capture knowledge?
  - How many iterations are necessary to capture knowledge in a sufficient amount of detail?
  - How do you verify that the captured knowledge is correct?
  - Can anyone capture knowledge?
  - What could be improved in this step?
- Knowledge Re-use
  - Is knowledge re-used?
  - How do you approach re-using knowledge from previous projects?
  - How do you approach re-using knowledge embedded in a legacy application?
  - Is it possible to re-use complete KBE modules for a new project?
- Knowledge Structure
  - Are there guidelines how to structure knowledge? If yes, which guidelines? If no, why not?
  - Are there any tools that support the structuring of knowledge?
  - What is the role of the person structuring knowledge?
  - Does he need special training, such as knowledge engineering?
- Codification
  - How do you approach codifying knowledge?
  - How is the quality of the knowledge used for codification?
  - Is there anything that could be improved to help you codify knowledge?
- Directly Connecting Knowledge to Code
  - Is there any connection between the application code, the run-time application, and / or the knowledge?

  - If yes, how do you establish this connection?
  - Is it possible to trace how knowledge is embedded in application code?
  - If yes, how do you approach this? If no, why not?
  - Is it possible to find the rationale of a design choice? How would you do this?
  - Is it possible to view relations knowledge has with other knowledge?
  - What would you like to see improved on the traceability of knowledge in application code?
  - What is limiting the projects from having this traceability?

- Application & Knowledge Quality
  - Are the developed applications transparent? Please explain.
  - Are applications and / or knowledge maintained? Please explain.
  - Is there a lack of trust in the application? Please explain.
  - Is knowledge often lost, duplicated, vague, or ambiguous? Please explain.
  - Is the knowledge stored the same version as the knowledge in the application? Please explain.

B

Knowledge Portal Development

This chapter will shortly discuss how the KP has been developed. It will start off by discussing the programming languages used, and the set-up of the KP. Then, the part of the KP that parses application code into separate COs is discussed. Afterwards, it will be explained how the developed ontology, the TM, has been embedded in the KP. This is followed by an explanation of how new KFs are created, modified, and saved to the database. Finally, a short explanation of the steps taken to generate the knowledge perspectives is given. This chapter should give the reader an idea of how the KP works.

B.1. Programming Language & Set-Up

Before the development of the KP could start, a programming language needed to be chosen and the different parts of the KP determined. The KP consists of five parts, as can be seen in Figure B.1. The Python source code is not regarded as part of the KP, it is only passed to the KP to parse the KBE application into COs.

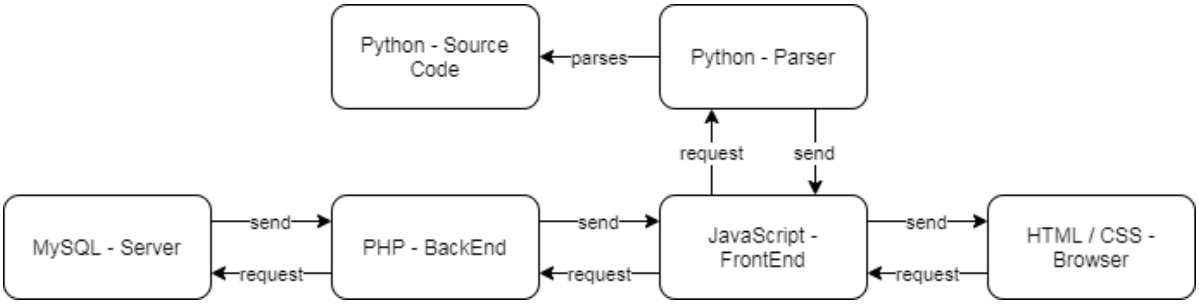


Figure B.1: The various parts of the Knowledge Portal, the languages, and the communication. The source code is not regarded as part of the KP.

It was important to consider what the KP should be able to do beforehand and which components were important. First, there was a choice whether it would be a web-based platform or an application; the former was chosen as it would be easier accessible for stakeholders and requires less up-front effort to get it up and running for the stakeholder during experiments. The downside was that I had no prior experience developing web-based applications. To develop a web-based application, it would require a front-end, a back-end, and a server.

For the front-end, the three standard languages were used: HTML, CSS, and JavaScript. For the back-end, it was possible to use Python, which is a language I am experienced in. However, PHP was chosen as it is one of the default programming languages for back-end and it did not seem hard to learn. Moreover, there is a lot of documentation on-line and a large community which could help with the development. For the server, MySQL has been chosen for similar reasons as PHP. Furthermore, MySQL is a language that is often used by server providers, so no extra effort would be necessary when migrating the Knowledge Portal to a service provider.

The last thing to consider was how to load the KBE application code in the Knowledge Portal. This could be done manually by entering forms, but it seemed tedious and unnecessary. Therefore, a parser was written that could automatically parse a python file into COs. As the KBE applications are written in Python, this language is also used to parse the application code.

B.2. Application Source Code Parser

The KP should be able to parse the KBE application into COs, to allow the user to connect the created KFs to the COs. A first attempt was done trying to develop a parser from scratch; however, the method used to parse seemed illogical and inefficient. Some research was done as to how Python compiles the code, and whether that could be used to parse it. When compiling code, Python will first parse it into a parse tree. Afterwards, this will be transformed into an Abstract Syntax Tree (AST) which is a high-level abstraction of the source code. This AST is transformed into a Control Flow Graph (CFG), and eventually converted into bytecode which can be read by the Central Processing Unit (CPU) to execute commands.

The AST that is generated turns out to be useful to parse code objects. Figure B.2 shows how the AST is used to extract information about a ParaPy Input. More complex COs such as ParaPy Attributes and ParaPy Parts will have a larger tree with more layers and information.

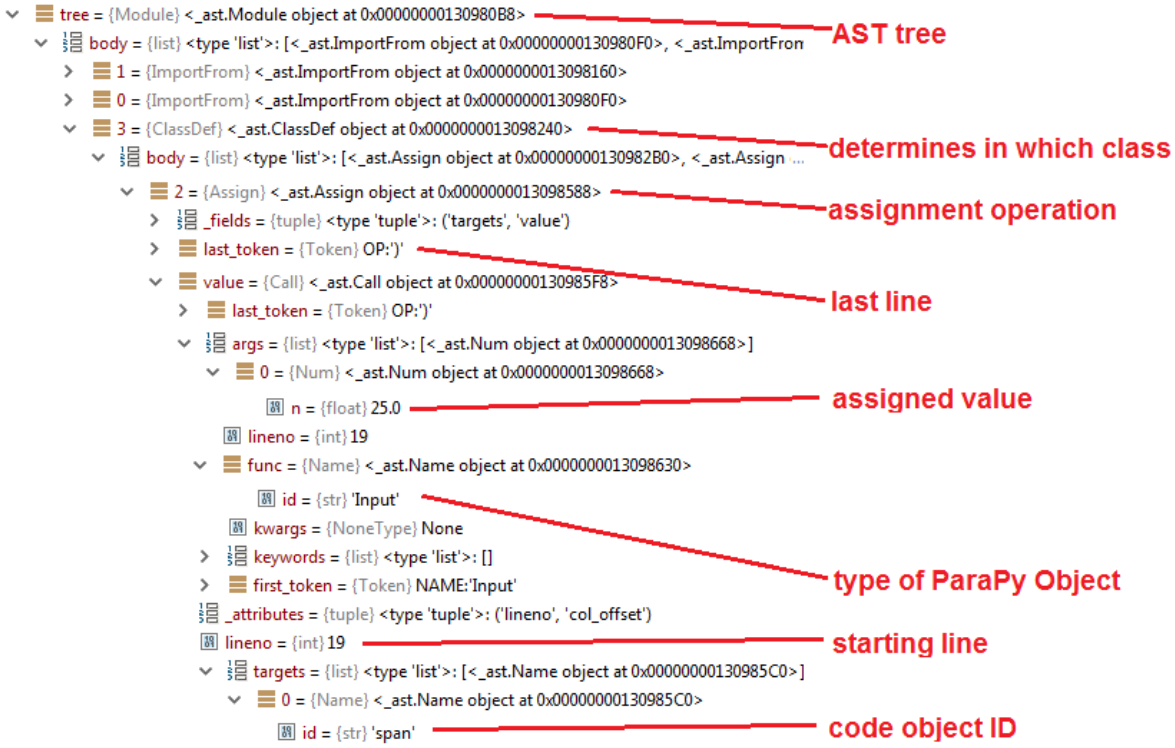


Figure B.2: Example of how to use the AST to extract information of a simple code object. Note: a lot of nodes were taken out of the tree to fit it into a figure.

Knowing which nodes an AST has made it possible to parse COs from application code. To determine the classes in the application code, the parser looks for the 'ClassDef'. Once found, it will iterate over its 'body', which contains all the COs that are being looked for. For example, in Figure B.2, there is an assignment operation object. This object consists of several attributes that can be iterated over to find out information about its line in the code, the value that is being assigned, and the type of CO (in this case an `Input`). It is also possible to find the ID of the CO: 'span'. Each bit of information is then saved in a dictionary format. This process is repeated for every object that is found in the class definition (ClassDef). Once this process is finished, all the COs have been extracted from the class, and it is possible to repeat the process for the next class in the application code.

After all classes have been iterated, there will be a final dictionary containing all the parsed COs and

their information. Figure B.3 shows the final dictionary with all the parsed information, the dictionary is built up in a way that it can be used to create a tree in JavaScript later on. There are several pieces of information captured for each code object: its id (text), its type, the icon corresponding to the type, its state which is used later on in JavaScript, and the children it has for the tree. This dictionary can easily be converted to a JSON string, which is an open data standard.

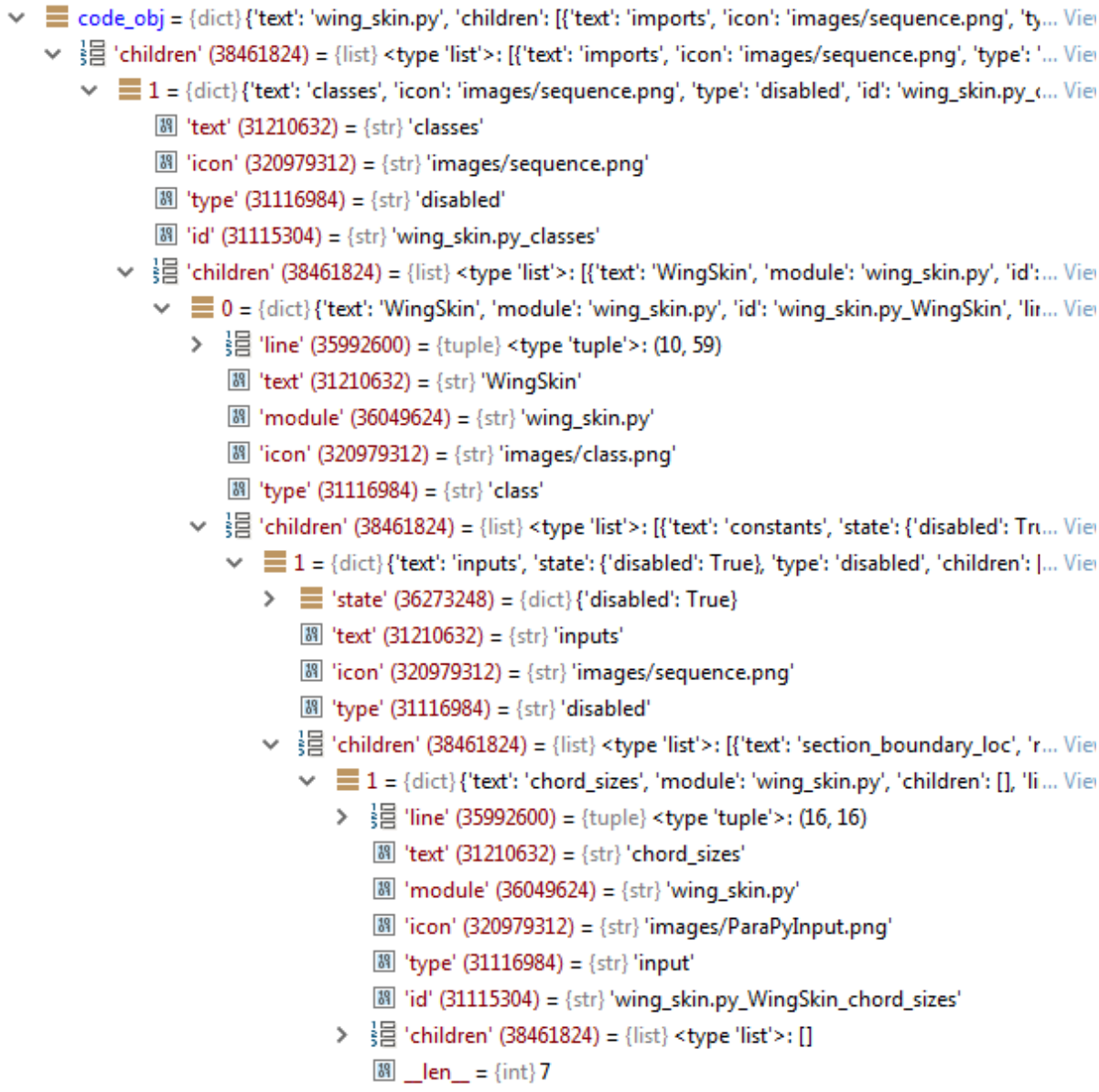


Figure B.3: The final dictionary containing all the COs that could be found in the application source code. The top object is a container for `Classes`, and it can be seen in the 'children' that there is a single `Class` called 'WingSkin'. This `Class` comprises several `Inputs`, `Attributes`, and `Parts`. However, the dictionary is too long to show, so only the first `Input` and its information is shown.

B.3. Importing the Traceability Model

The TM defines the classes of KFs, what knowledge needs to be captured for each class of KF, and the possible relations that can be defined with other KFs or COs. To make the TM usable in the KP, it has been captured in a JSON string. Figure B.4 shows part of the model used in the example, how it looks like in the JSON string, and how it looks like when creating a KF in the KP. Each type of CO has a certain type of KF it can be related. Another example is a `Part`, which can only relate to a *Child*. Each class of KF has certain relations it can set with only certain classes of KFs.

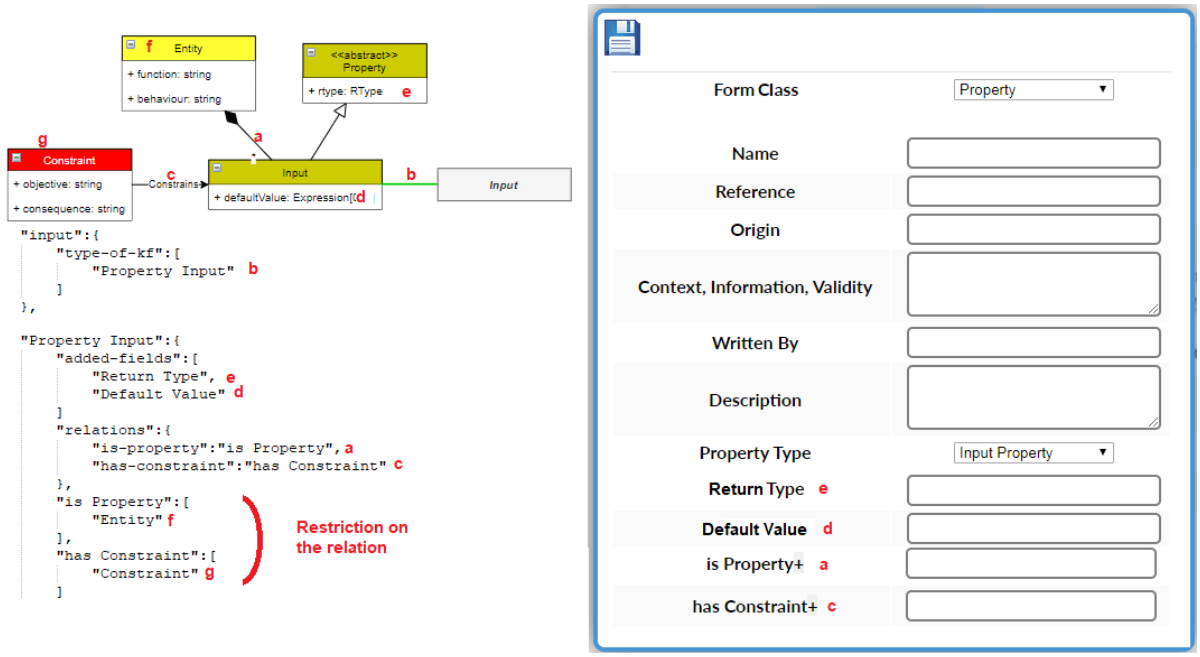


Figure B.4: In this example there are seven aspects that should be captured in the JSON string: a) f) *Input* composes an *Entity*, b) *Input* relates to an `Input`, c) g) *Input* can relate to a *Constraint*, d) *Input* has an added attribute Default Value, e) Every type of *Property*, so also *Input*, has an added attribute Return Type. This is captured in JSON as can be seen in the figure, and the corresponding KF that is created by the KP can also be seen in the figure.

B.4. Communication with the database

Whenever a user interacts with the KP, there is a great chance that their command is being sent to the back-end, and then to the database to retrieve information about a certain KF or the relations that are saved. This is done using the JQuery AJAX (Asynchronous JavaScript and XML) technology. The fact that AJAX is asynchronous means that data can be loaded into the KP without having to refresh the KP, this makes the application more intuitive and easier to use. Figure B.5 shows a simple example of an AJAX statement that sends a command to a 'test.html' with a context called 'document.body'. This command is carried out in parallel to other commands on the application, and once it is done it will carry out the commands that are given in the .done() body.

```
$.ajax({
  url: "test.html",
  context: document.body
}).done(function() {
  $( this ).addClass( "done" );
});
```

Figure B.5: An example of an AJAX statement, that sends a command to a file called "test.html" with a certain context. When 'done', it will carry out the statement in the .done() body. It will allow for parallel processes and for the KP to work without needing to be refreshed.

AJAX is used throughout the KP: to save a KF to the database, to retrieve a KF from the database, and to retrieve relations from the database. This is done by communication with PHP files to execute commands in the PHP file. The PHP file will then communicate with the database by send MySQL statements: e.g. "INSERT INTO Rule_table('kf_json') VALUE json". The next sections discuss how KFs are saved and retrieved in the database, whenever it is said that something is loaded or saved to the database, it will go through the process as explained in this section.

B.5. Managing Knowledge Forms

B.5.1. Loading Knowledge Forms

Once the KP is opened in the browser, one of the first things done is that existing KFs are loaded into the KF tree. The KF tree is built using an existing JavaScript module called JSTree. There are five nodes in this tree, one for each KF class, except for *Properties* which are grouped under the corresponding *Entity*. This was done as it seemed logical to group the *Properties* with their corresponding *Entity*. When the browser is loading, a function runs that retrieves each KF from the database and translates it into a format usable for the JSTree.

B.5.2. Creating Knowledge Forms

Creating KFs can be done by clicking on the Create Form button. This runs a function that opens a new window asking for a form type. Once a type has been selected, it will retrieve the attributes of the chosen type automatically from the JSON string containing the TM. When the attributes are loaded, a user can fill in information as required; if a sub-type is required it will be shown. The relations that can be set from the chosen type will appear on the bottom of the form. Each relation will have a '+' button to add a knowledge form to that specific relation. When the button is activated, a new window opens to select which KFs or COs are to be related. Meanwhile, each KF and CO has been disabled except for the type that may be related to. This is how a user is guided to make correct relations. When the user clicks on the 'Save Connection' button, the related KF or CO is shown in the correct attribute field. Once all information is captured, and all relations are set, the user can press on the save button. This will do 2 things:

Saving the KF as JSON string

The first is that it will translate the attributes (with field values) into a JSON string and be sent to the server to be saved, along with a new ID. The ID is chosen by concatenating the type of form and the ID on the database (e.g. Rule_7). An example JSON string can be seen in Figure B.6, this JSON string is sent to the database.

```
{
    "name":"Equispaced",
    "origin":"Domain Expert",
    "written":"Arthur",
    "variables":"[list, L, s, i]",
    "rule":"list = L / s * i",
    "context":"Any geometric placement",
    "description":"Geometries that need to be placed equispaced can be done so by using this rule",
    "category":"Rule"
}
```

Figure B.6: A Rule and its attributes, with values, captured in a JSON string.

Saving Relations as Triples

The second thing is that the chosen relations are to be saved. The first thing to do is to retrieve the KF's own ID, which has just been saved to the database. This can be done in MySQL with the command "SELECT LAST_INSERT_ID();". The ID of the KF or CO that is to be related is passed when clicked on the save button, as it was chosen by the user which two KFs or COs relate to one another. Now there are two options, either the KF relates to another KF, or it relates to a CO. The KP knows that when a KF and an CO is related, that it will have the same relation: 'Mapped onto'. This refers to the fact that the KF is connected to a CO. The relation will be saved in the relation table in the database as a triple, by passing a subject - predicate - and an object. An example of a triple can be seen in Figure B.7

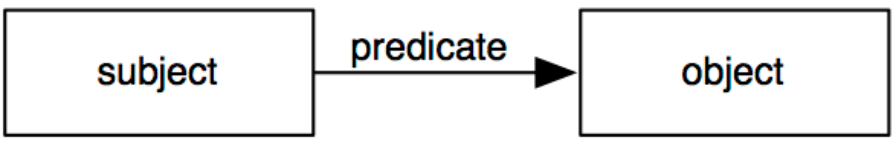


Figure B.7: How a triple is built up.

In the case of a KF and a CO: KF - Maps onto - CO. To ensure that all the relations from a CO can be achieved, the KP automatically knows that when KF - Maps onto - CO, a CO - Maps onto - KF. For 2 KFs this is more complex. Before saving it into the relation database, a predicate is necessary. This will be retrieved from the JSON string containing the TM. An example could be Entity - Composes - Property. The relation the other way around is known by the KP as every relation, and its counter-part, is also saved in the JSON string containing the TM. Therefore the KP can automatically deduce that Property - is Property - Entity.

B.5.3. Modifying Knowledge Forms

A KF can be modified by selecting it in the KF tree. When selected, it will be opened in the Data Viewer. From the Data Viewer, the attributes of the KF can be modified simply by overwriting the attribute. Relations can be modified by overwriting the existing relation. When an attribute is overwritten, a whole new JSON string is generated and it will replace the current JSON string in the database.

B.6. Loading Knowledge Perspectives

B.6.1. Data Viewer

The Data Viewer can be loaded from the KF tree, or by choosing a KF in the Relation Viewer. When a KF is clicked, the Data Viewer is loaded and the KF ID is passed to the database to retrieve the JSON string of the corresponding KF. The JSON is then looped through, retrieving the attribute and the corresponding value, which are added to a table one by one. After all attributes are added to the table, the whole KF with its information is shown in the Data Viewer. The Data Viewer allows the user to modify attributes or relations., how this is done is explained in section B.5.

B.6.2. Traceability Viewer

The KP uses the parsed application code to find out which `Classes` are available. Each `Class` will have its own tab in the Traceability Viewer. When a `Class` is selected, the corresponding code from that `Class` is loaded into the right-hand pane of the Traceability Viewer. After the code is loaded, each CO is run through the relation database table to find out which KFs are connected to it. Each connected KF is loaded next to the CO in the left-hand pane.

B.6.3. Relation Viewer

To get to the Relation Viewer, the user will have to right click a KF and click on 'Relation Viewer'. When that button is clicked, the ID of the KF is sent to the relation database, and each related KF or CO is put into a dictionary. The KP will loop through each entry in the dictionary to find how each entry is related to other KFs and COs. This is repeated twice to retrieve all the KFs and COs that need to be shown in the Relation Viewer. Then, the KP will determine the position of each element. This is done using the polar coordinate system, where the position of an element is determined from a pre-determined radius and calculated angle of the parent element. Knowing that a circle has 360 degrees, each element is spaced out $\frac{360}{n}$ degrees from one another, with 'n' being the number of KFs and COs that is related to the chosen KF. This process of positioning is repeated for each level of depth; however, the available degrees is limited to the boundaries of its location. For example, if an element is placed every 120 degrees, that element will have its sub-elements equispaced within a cone of 60 to 180 degrees.

B.6.4. Activity Viewer

The Activity Viewer is loaded by right-clicking a KF and choosing 'Open Activity Viewer'. When an Activity is chosen, the KP will retrieve the Activities that precede or succeed it from the database and put it into correct order. When another class of KF is chosen, the KP will infer which Activity is related to it (if there is one, which there should be if structured correctly). For example, if a Property is chosen, the correct Activity will be found by search in the relation table which Activity is related to the chosen Property. Once the corresponding Activity is found, the preceding and succeeding Activities will be retrieved from the database and put into order.