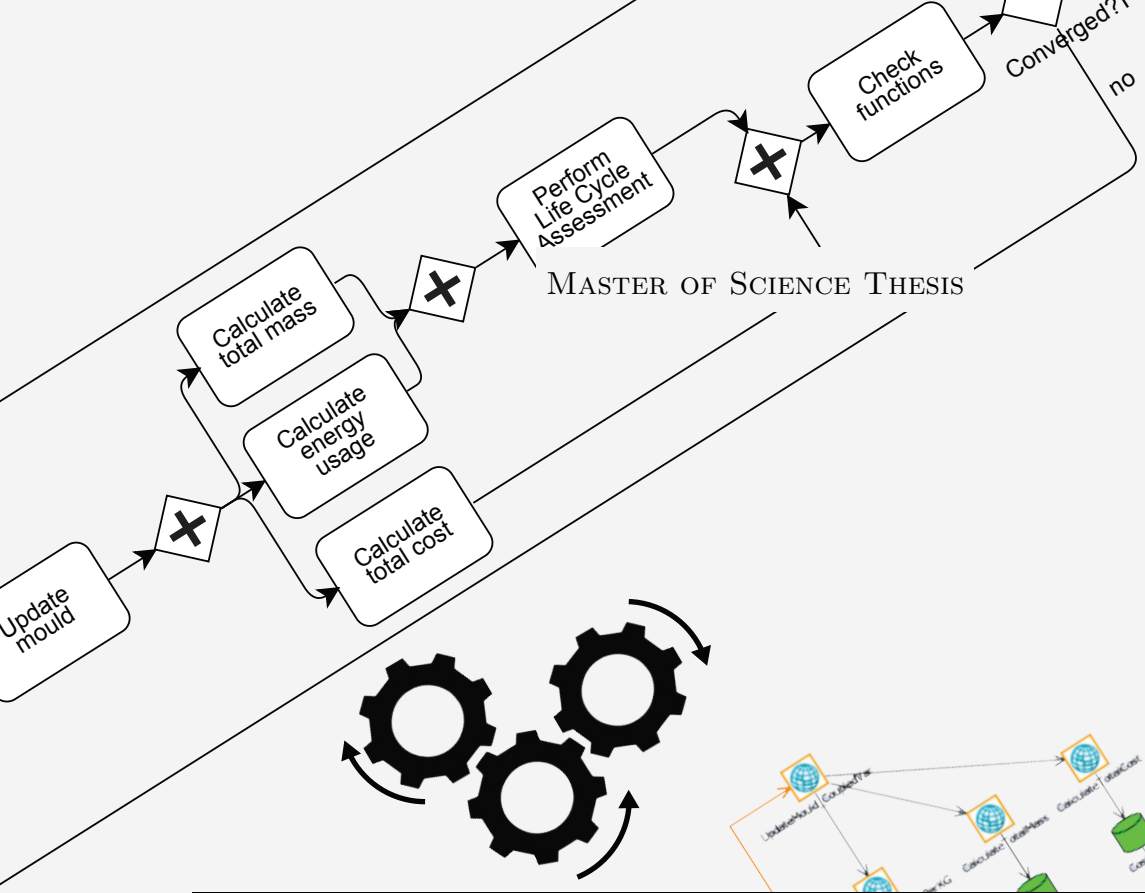
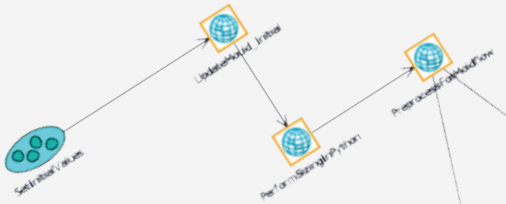


Master of Science Thesis

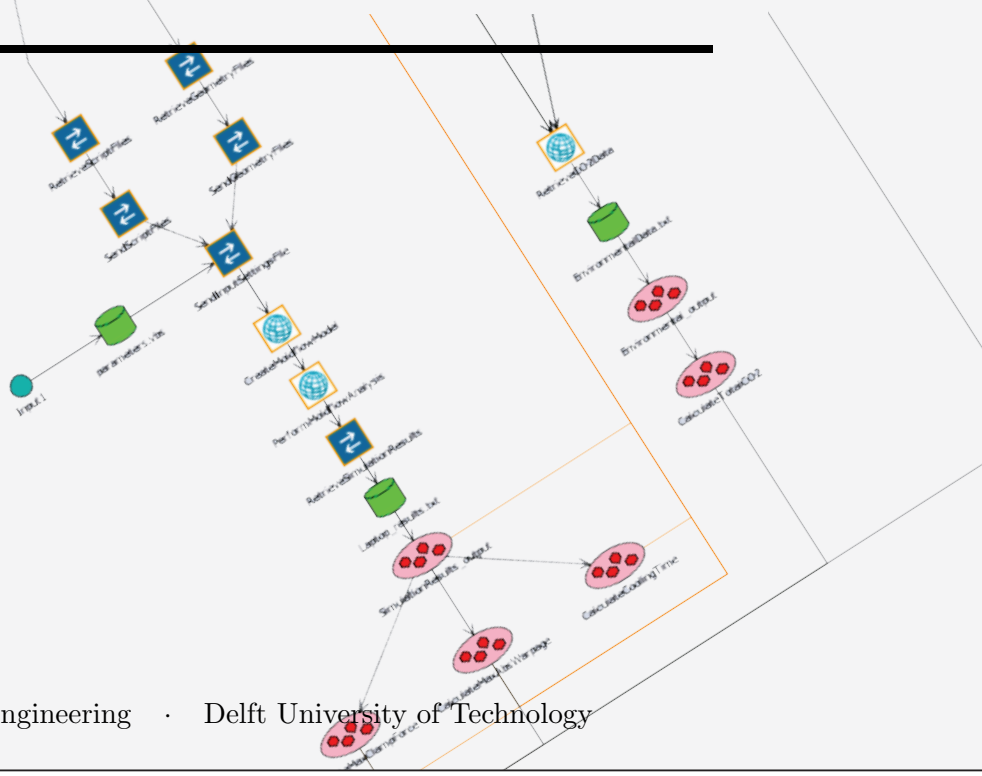
A New Methodology for the Development of Simulation Workflows

Moving Beyond MOKA

P.K.M. Chan B.Sc.



15th March 2013



Faculty of Aerospace Engineering · Delft University of Technology

TU Delft

A New Methodology for the Development of Simulation Workflows

Moving Beyond MOKA

Master of Science Thesis

For obtaining the degree of Master of Science in Aerospace Engineering at Delft University of Technology

P.K.M. Chan B.Sc.

15th March 2013

Faculty of Aerospace Engineering · Delft University of Technology

Delft University of Technology

Delft University Of Technology
Department Of
Flight Performance and Propulsion

The undersigned hereby certify that they have read and recommend to the Faculty of Aerospace Engineering for acceptance a thesis entitled **"A New Methodology for the Development of Simulation Workflows"** by **P.K.M. Chan B.Sc.** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: 15th March 2013

Head of department: _____
Prof.dr.ir. M.J.L. van Tooren

Supervisor: _____
ir. R.E.C. van Dijk

Reader: _____
R. d'Ippolito M.Sc.

Reader: _____
dr.ir. A.J.H. Hidders

Reader: _____
dr.ir. A.H. van der Laan

Summary

One of the main challenges in Multi-disciplinary Design Optimisation (MDO) is the interoperability of heterogeneous simulation tools. Some researches have reported that, due to these interoperability issues, only around 20% of the product development time is spent on analyses and creative design tasks. Clearly, there is a lot to gain, when it comes to improving this figure.
Key to the success of MDO are Knowledge Based Engineering (KBE) and Simulation Workflow Management (SWFM) technologies. However, developing KBE and SWFM applications requires a substantial amount of (programming) knowledge and expertise. Due to these constraints, the technologies are less accessible to non-programmers. Additionally, there is an increased risk that applications may become black boxes when it is not clear what knowledge went into the application. This complicates sharing and reusing knowledge in future projects.
A methodology is needed to avoid these complications. Of all methodologies, MOKA is the most well-known methodology for developing KBE applications. It focuses on capturing and structuring knowledge to increase transparency. However, MOKA is rather product-oriented than process-oriented, and thus lacks the methods and tools for developing simulation workflows.

Based on these findings, there are two goals in this research:

1. *Develop a new methodology for SWFM.*
2. *Reduce the amount of required expertise for modelling simulation workflows.*

A new methodology is developed starting from the foundations laid by MOKA. It presents new step-by-step instructions to guide engineers in the modelling process. Furthermore, the methodology introduces new forms, the Business Process Model and Notation (BPMN), and an N^2 notation to capture and structure process knowledge. This knowledge is then formalised (i.e. translated to a format which is closer to computer language) before Model Driven Software Engineering (MDSE) techniques are used to automatically generate the workflow.
For this purpose, a new integration framework has been developed, based on the Integrated Design and Engineering Architecture (IDEA) which evolved from the Design and Engineering Engine (DEE). The new framework couples a Knowledge Base (KB), product (KBE) and process (SWFM) tools (see Figure 1).

Reducing the required expertise is achieved by introducing High-Level Activities (HLA). HLAs are activities that consist of five primitives: *input*, *preprocessor*, *analysis*, *postproces-*

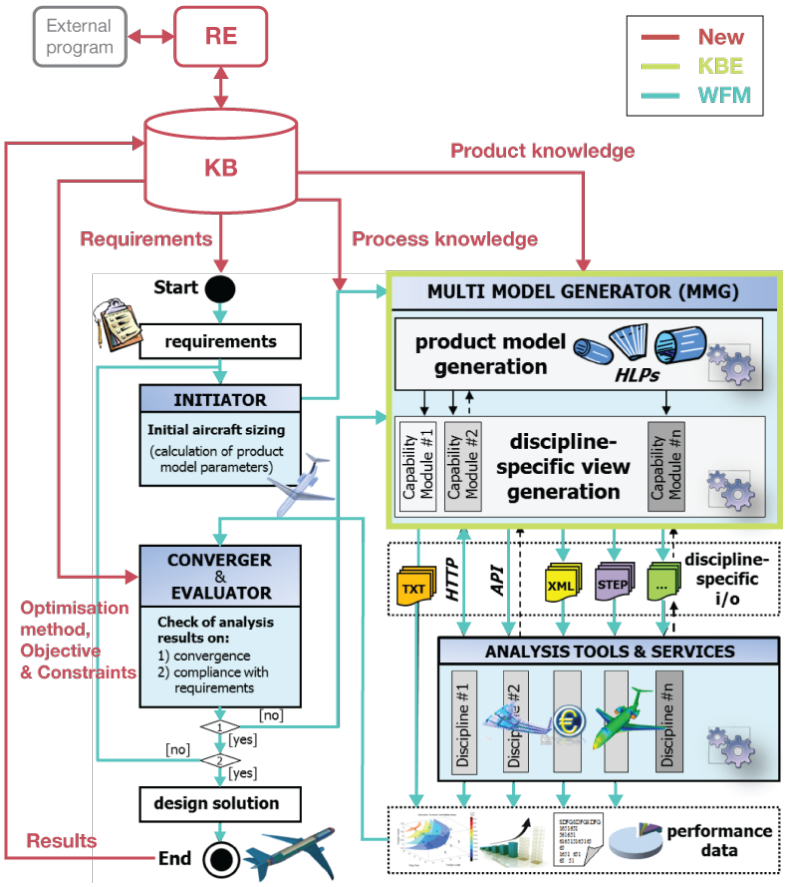


Figure 1: Diagram of the IDEA framework showing which components are new (in red) and where KBE (in green) and SWFM (in blue) technologies are applied.

sor, and *output*. Capturing lower-level knowledge in these HLAs allows for inexperienced engineers to model workflows at a higher abstraction level. Meanwhile, a new parametric high-level workflow has been designed, that enables engineers to optimise KBE product models without actually modelling a workflow. Both the HLAs and the parametric workflow are used in several use cases involving a packaging design optimisation and an MDO workflow for thermoplastic injection moulding.

In the end, this work has delivered tools, methods, and a framework that:

- increases transparency of SWFM applications
- saves development time
- reduces required expertise to model simulation workflows

Preface

This thesis work forms another milestone in the mission of the Flight Performance and Propulsion (FPP) department to increase the productivity of modern-day engineers in complex product design. It provides a new methodology and supporting tools to develop simulation workflows, starting from a very high level down to the implementation. The iProd project objectives proved to be in line with the specific objectives of this thesis, therefore part of this work has been performed in the scope of this project (see Figure 2). To this end, this thesis has contributed to iProd in the form of the development of several engineering workflow ontologies and the implementation of an example MDO problem.

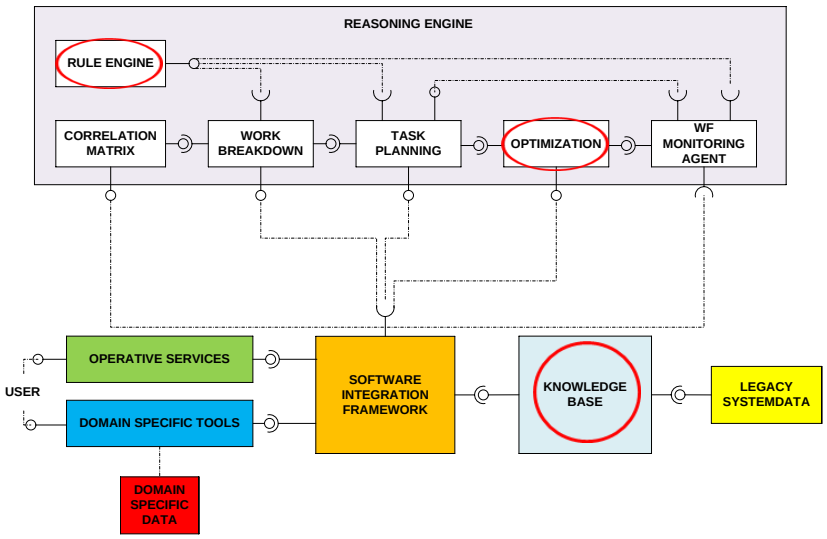


Figure 2: Mapping of thesis work onto the iProd software architecture

Acknowledgements

Here I am, at the end of my time as an aerospace engineering student. It has been a long journey, that sadly enough will come to an end. But without doubt, it has been a very interesting and challenging period. I realise that I could not have accomplished all this without the support from a number of people.

First of all, I would like to thank Reinier van Dijk for his continuous support, invaluable expertise, and the numerous interesting discussions we had on the topic. His enormous drive inspired me to push myself to come with great solutions. Furthermore, I would like to thank Michiel van Haanschoten for his great efforts in providing IT support. I would also like to thank the members of my committee for investing their time and effort to evaluate my work: Michel van Tooren, Roberto d'Ippolito, Jan Hidders, and Ton van der Laan.

I would like to thank the iProd - Integrated Management of product heterogeneous data - FP7 project of the European Community (Grant agreement no. 257657), and consortium for the provision of background material, use cases and feedback during my thesis.

Many thanks to all my friends for their support and encouragements whenever I needed it the most. Much love goes out to my dear family who have always supported me in my life. Finally, I would like to express my enormous gratitude to a special person, my girlfriend Siran, who has always been there for me.

Delft, The Netherlands — P.K.M. Chan B.Sc.
15th March 2013

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations

AI	Artificial Intelligence
AM	Analysis Module
ANSI	American National Standards Institute
ATM	Air Traffic Management
BPMN	Business Process Model and Notation
CAD	Computer Aided Design
CAE	Computer Aided Engineering
CAx	Computer Aided Technologies
CFD	Computational Fluid Dynamics
CL	Common Lisp
CM	Capability Module
CRM	Customer Relationship Management
CWA	Closed-World Assumption
DEE	Design and Engineering Engine
DL	Description Logic
DSL	Domain-Specific Language
EPC	Event-driven Process Chain
FEA	Finite Element Analysis
GenDL	General-purpose Declarative Language
GGG	General Geometry Generator
GUI	Graphical User Interface

HES	High-level Engineering Service
HLA	High-Level Activity
HLP	High Level Primitive
HWFM	Human Workflow Management
IaaS	Infrastructure as a Service
ISO	International Organization for Standardization
KBE	Knowledge Based Engineering
KBS	Knowledge Based System
KR	Knowledge Representation
LCA	Life Cycle Assessment
MDO	Multi-disciplinary Design Optimisation
MDSE	Model Driven Software Engineering
MMG	Multi-Model Generator
MML	MOKA Modelling Language
MOKA	Methodology and tools Oriented to Knowledge-based engineering Applications
NASA	National Aeronautics and Space Administration
OMG	Object Management Group
OPMD	Optimization Model
OWA	Open-World Assumption
OWL	Web Ontology Language
PaaS	Platform as a Service
PDSL	Problem Domain-Specific Language
PIDO	Process Integration and Design Optimisation
RDB	Relational Database
RDF	Resource Description Framework
REST	Representational State Transfer
RE	Reasoning Engine
RIF	Rule Interchange Format
SaaS	Software as a Service
SFTP	SSH File Transfer Protocol
SOAP	Simple Object Access Protocol
SOA	Service Oriented Architecture
SPARQL	SPARQL Protocol and RDF Query Language
STEP	Standard for the Exchange of Product model data
SWFM	Simulation Workflow Management
UCA	User Customisable Action
UDDI	Universal Description Discovery and Integration
UML	Unified Modeling Language

UPI	Unique Parameter Identifier
WADL	Web Application Description Language
WFMS	Workflow Management Systems
WFM	Workflow Management
WSDL	Web Service Definition Language
XML	Extensible Markup Language
YAWL	Yet Another Workflow Language

Chapter 1

Introduction

1.1 Future of Aerospace Engineering

The aerospace industry is on the verge of entering a new era. In the next decades, people can expect to see novel aeroplanes built with the latest technologies in the most original configurations (see Figure 1.1). This new generation of aircraft delivers an ever more comfortable and convenient experience, attracting even more travellers than today. It is expected that the number of flights within Europe will grow from *9.4 million* in 2011 to *25 million* in 2050 (Kallas et al., 2011). It will be a real challenge for the aerospace engineering industry to live up to these expectations.

Figure 1.1: Future aircraft concepts presented by NASA and Airbus (sources: NASA, 2012; Airbus, n.d.).

The main concern from a technological viewpoint is to maintain safety and sustainability when air traffic grows. Safety is guaranteed by developing new intelligent Air Traffic Management (ATM) systems, while innovative materials and engine technologies provide sustainability. Especially sustainability has become a main priority for the aerospace

community. Europe's vision is to reduce the *perceived noise* by 65%, CO_2 *emissions* by 75%, and NO_x *emissions* by 90% by the year 2050 (relative to the amounts measured in 2000). Reducing emissions by these amounts drives the aerospace industry to further innovate technologies. However, focusing solely on new product technologies will not be sufficient. Changing market conditions forces aerospace companies to redefine their design process as well. Or as La Rocca (2011) concluded: *"In order to make a step change in aviation, a paradigm shift in the design methodology will be required."*

1.2 Challenges in Aircraft Design

One of the most urgent challenges for the European aerospace industry is the heavy competition from upcoming countries, such as Brazil, China, India, and Russia. These countries, with the exception of Russia, have already settled among the top five competitors (see Figure 1.2). Russia lost its dominance after the cold war, but the tide is changing as it is the fastest growing country on the index, moving from the 20th to 14th place (Roth et al., 2010). Moreover, it aims to become the worlds third-largest aircraft manufacturer by 2015 (Platzer, 2009). More competition allows customers to select products that perform best in terms of quality, performance, cost, and time to market. Even though there are many other factors that determine a countries' competitiveness, it is clear that the new generation of manufacturing powers outclass the established countries, including West-European countries.

Rank	Country	Index score 10=High 1=Low
1	China	10.00
2	India	8.15
3	Republic of Korea	6.79
4	United States of America	5.84
5	Brazil	5.41

Rank	Country	Index score 10=High 1=Low
1	China	10.00
2	India	9.01
3	Republic of Korea	6.53
4	Brazil	6.32
5	Untied States of America	5.38

Source: Deloitte and US Council on Competitiveness - 2010 Global Manufacturing Competitiveness Index; ©Deloitte Touche Tohmatsu, 2010.

Figure 1.2: Tables showing the competitiveness index for the top 5 countries in 2010 (left table) and the expected top 5 in 2015 (right table) (source: Roth et al., 2010).

The same report by Deloitte (Roth et al., 2010) has identified talent-driven innovation as the most important competitive driver. All manufacturers around the globe agree that attracting talented and skilled people is a key requirement for innovation and for increasing production efficiency. Unfortunately, manufacturers in developed countries are struggling to find new talents that are able to replace their retiring workforce. Many of these countries suffer from an overall decline in birth rate. And even though more people go through tertiary education nowadays, the net result is a shortage of qualified engineers. Gertler attributes this problem to a reduced interest in aerospace (PE Magazine, 2008). After the cold war the number of research programmes declined, which had a negative effect on promoting the aerospace industry. The loss of excitement in engineering has pushed young talents to other industries.
Moreover, the young engineers who have been attracted, are exposed to less development programmes (see Figure 1.3). As a result, these engineers have less experience, but are still expected to design increasingly complex products. Altogether, future engineers need

to be much more productive due to these complications (van Tooren, 2003).
Another direct consequence is that companies have to find ways to retain knowledge when their experts retire. But without solutions in knowledge management, retiring experts will not be able to transfer their expertise effectively to the new generation. This may lead to a loss of knowledge, which will have a large impact on future product development. Work needs to be redone, while the competition continues to innovate.

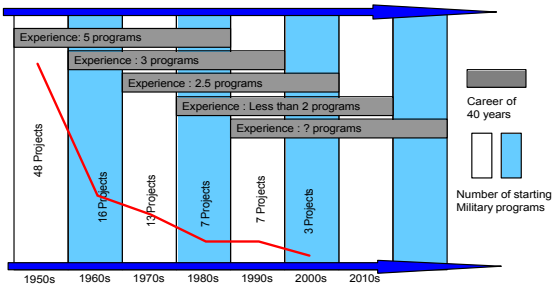


Figure 1.3: Future engineers have less experience (source: van Tooren, 2003).

Besides organisational challenges, companies are facing technological challenges as well. Many are related to the increasing usage of information technologies. Nowadays design engineers have access to a growing amount of simulation tools, also known as Computer Aided Engineering (CAE) tools. Consequently, designers are exposed to more information, which improves the decisions made during the design process. However, the increased usage of simulation leads to certain complications as well.

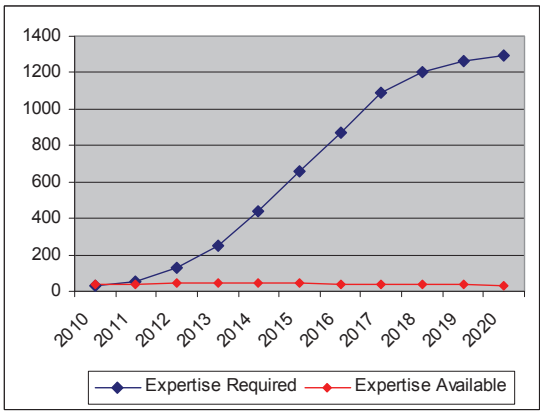


Figure 1.4: Graph showing expected problems in simulation software usage (source: Walsh, 2011)

The first complication relates to simulation expertise, the skill of defining an engineering problem into a simulation problem, which is rapidly becoming an obstacle. Walsh (2011) has found evidence in much research that the amount of required simulation expertise will increase drastically while the availability of expertise will remain nearly constant (see Figure 1.4). The only way to confront these problems is by reducing the amount of required expertise, and thus making software *"smarter"* and not only *"easier to use"*. The latter has already been tried for over twenty years and has not succeeded. Therefore,

the former option of making software *"smarter"* has to be the choice of future solutions. Another growing issue is the interoperability of heterogeneous tools (i.e. tools that have not been designed to cooperate nicely). As a result, design engineers spend the majority of their time managing dataflow between software tools, including manually translating data to tool-specific formats. Interoperability studies in the U.S. show that it costs the automotive supply chain sector $1 billion per year (Brunnermeier & Martin, 1999) and the capital facilities industry $15.8 billion per year (Gallaher et al., 2004). Figures from the aerospace industry have not been reported, but NACFAM (2001) expects similar amounts.

Without new solutions, design engineers continue to waste time on non-value adding activities. Bazilevs et al. (2009) refers to the results of a study where only 23% of the overall simulation time is spent on analysis. Similar figures are found by Stokes (2001), who discovered that merely 20% of the design process is dedicated to creative processes. These numbers will further decline if companies decide to apply Multi-disciplinary Design Optimisation (MDO) without rethinking their design process. MDO is a methodology for the design of complex systems where strong interaction between disciplines motivates designers to control variables from several disciplines. It involves heavy simulation usage in various disciplines. Hence, one of the main challenges of building an MDO framework is the coupling of heterogeneous simulation tools. Yet, companies should consider investing in developing a framework. Boeing demonstrated with their design project of a hypersonic vehicle that MDO can be a solution to the inefficient traditional design methods (Bowcutt et al., 2008). Flager & Haymaker (2007) surveyed Boeing's design team that worked with both traditional design processes and the newly designed MDO framework, and discovered that a drastic reduction of time spent on managing information is achievable (see Figure 1.5). The saved effort gave the designers more freedom to explore the design space and evaluate the results. The success of this project provides the necessary incentive for the aerospace industry to adopt MDO. The next section explains two essential technologies for MDO frameworks.

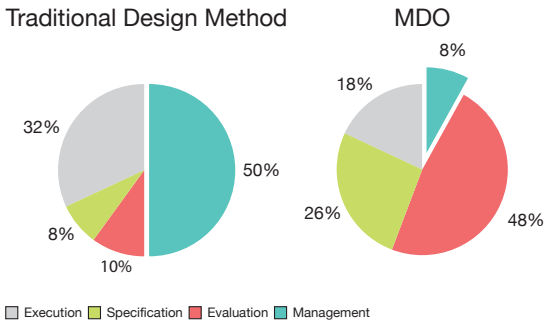


Figure 1.5: Comparison of time spent on design activities (source: Flager & Haymaker, 2007)

1.3 Current and Future Trend

MDO is growing in popularity as companies aim to increase their competitiveness and productivity. But to be successful, MDO depends heavily on automation technologies.

An automated MDO framework enables companies to design products with higher quality standards and performance, while reducing lead time and the required number of engineers. Two essential technologies supporting MDO are Knowledge Based Engineering (KBE) and Simulation Workflow Management (SWFM).

For complex products, MDO relies on parametric, generative modelling techniques to manipulate the geometry and to (re)configure the product (van Dijk et al., 2012; La Rocca, 2012). Computer Aided Design (CAD) allows for parameterisation of the product geometry, but is limited to a single topology only. KBE on the other hand, is capable of capturing engineering knowledge that is necessary for automatically generating various configurations (see Figure 1.6). Moreover, this knowledge extends to disciplines besides geometry (such as cost and manufacturing) which is used to produce different 'views' of a product model. This supports automation of multi-disciplinary analysis and optimisation.

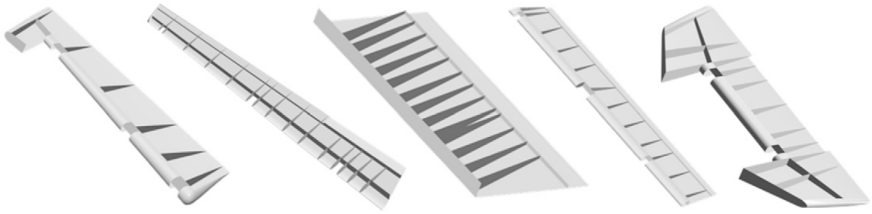


Figure 1.6: Example of different aircraft movable configurations, all generated from the same product model with KBE technologies (La Rocca, 2012).

MDO requires another technology for coupling simulation tools and to automate the execution of these tools. SWFM is a discipline that covers simulation workflows, where heterogeneous simulation tools are brought together in a workflow. Well-equipped (Simulation) Workflow Management Systems (WFMS) provide a rich Graphical User Interface (GUI) for modelling and executing simulation workflows. In the engineering domain these systems are also capable of performing optimisation, hence are often referred to as Process Integration and Design Optimisation (PIDO) solutions. PIDO software excel in integrating tools into a workflow and then automate design optimisation. It functions as a central platform that controls the flow of processes and the dataflow between tools. This reduces the number of interfaces between tools, which increases exponentially once more tools become available (see Figure 1.7). Therefore, PIDO solutions can greatly simplify MDO.

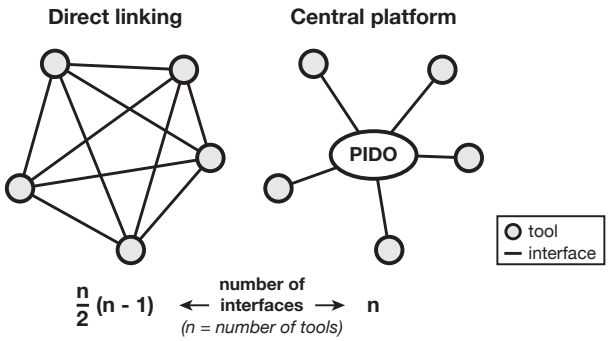


Figure 1.7: The number of interfaces increases exponentially if tools are linked directly.

1.4 Scope of this research

The development of MDO frameworks involves many methodologies and technologies. Altogether, it is a very wide domain that is impossible to research during a Master's thesis. Therefore, it is important to narrow down the scope and set realistic goals.

The focus in this research is on SWFM. The keyword in SWFM is Workflow Management (WFM). WFM is the discipline that defines, creates, and manages the execution of workflows. In WFM the process structure and interactions with participants and other processes is more important than describing the technical details related to implementation. In other words, the knowledge behind a workflow is more valued than the actual implementation details. By capturing this knowledge explicitly, organisations obtain more insight into their processes. This knowledge can then be shared and reused in other workflows.

Human Workflow Management (HWFM) is the other domain in WFM, where workflows consist of human activities. Normally, all processes are defined in a top-down manner starting with human activities. Therefore, HWFM should not be neglected. But because MDO is a simulation intensive process, it has been decided to focus on SWFM.
Now that the scope has been narrowed down to SWFM, the next section discusses the shortcomings of current solutions.

1.5 Shortcomings of Current SWFM Solutions

Current SWFM solutions excel in integrating and automating processes, which contributes to increasing competitiveness and productivity. However, this section explains that these solutions need improvements to overcome the remaining challenges. The following shortcomings have been identified:

- Modelling workflows requires expertise that does not belong to the design engineer's skill set. SWFM software do not possess intelligence and are therefore only *"easier to use"*.
- Workflows tend to behave as black boxes, because SWFM systems do not capture the rationale of a workflow.
- Knowledge management features are limited in current solutions. Without these features, it is more difficult to retain knowledge when experts are leaving the company.
- Workflows are not modelled in a standardised, neutral process language, which promotes sharing and reuse of process knowledge.

These shortcomings are explained in more detail below.

SWFM systems are designed for modelling and executing workflows, and do not include tools or methods that facilitate designing a workflow. The system relies on the user's knowledge for designing well-defined and efficient processes. In current practices, it is common that design engineers collaborate with IT engineers for this task. This is an inefficient process where individuals rely on each other's expertise. A better solution is to simplify usage intelligently. An intelligent system reuses process knowledge to support on

the technical details of workflow creation and configuration. Then the design engineer can focus on workflow design, while the IT engineer remains responsible for the operational aspects.
Furthermore, without any guidelines SWFM practitioners will most likely derive ad hoc solutions that will be inconsistent and incompatible with other solutions. Knowledge will thus be unexchangeable and reuse of tools will be limited, with the result that work is being redone and application development will take much longer.

Workflows are perceived as black boxes when the rationale behind the workflow is unclear. This occurs when decisions that are made during the design of the workflow have not been captured. Often, this knowledge is difficult to retrieve, or in the worst case, lost. The consequence is that no one understands what the workflow does or how it works.
Besides understanding the workflow, there are four other reasons why capturing knowledge is valuable for SFWM systems.

1. For reuse of knowledge in other projects.
2. For reuse of knowledge to implement intelligent features.
3. To redefine SWFM systems as platforms for sharing knowledge, similar to how the internet has become a global medium for sharing knowledge.
4. To provide a reference or handbook to newcomers.

Unfortunately, current solutions focus rather on execution aspects of SWFM than on knowledge management. However, this is unjustified, because knowledge may be the most valuable asset of a company.

Lastly, PIDO software vendors do not stimulate the use of a standardised and neutral process language because vendors want their customers to be dependent on their products. This behaviour is known as *vendor lock-in* and it prevents users from using another product because of high switching costs. If a standardised language existed, companies could easily transfer old workflows to the new PIDO solution, thus avoiding any loss of knowledge. And as long as there is wide support, it will be easy to find qualified experts or to train new people.

To improve on these shortcomings, a methodology is needed that facilitates capturing, sharing, and reusing knowledge. Of all methodologies, MOKA is the most promising. MOKA is the Methodology and tools Oriented to Knowledge-based engineering Applications (Stokes, 2001). It focuses on capturing and structuring relevant engineering knowledge. Even though MOKA is rather product-oriented than process-oriented (see Chapter 3), and thus lacks the methods and tools for developing simulation workflows, it provides a good starting point for the development of a new methodology. This is one of the research goals that are presented in the next section.

1.6 Research Goals

The aerospace industry in Europe needs to invest in new solutions in order to regain and maintain a position at the top of this competitive industry. One of these solution is a new methodology for SWFM. This methodology contains methods and tools for modelling the workflow at a higher abstraction level than current SWFM systems. The following set of research goals defines the path to this solution.

- ◇ *Investigate how current Workflow Management Systems can be complemented with knowledge management technologies and find solutions for reducing the complexity of Workflow Management.*
- ◇ *Extend the MOKA methodology with an ontology for storing process knowledge (on an informal and formal level) in a platform-independent and transparent model, as to maximise the potential for sharing and reusing this knowledge.*
- ◇ *Build and demonstrate an advanced Workflow Management System in a number of use cases varying in scope and complexity, that captures and reuses process knowledge for automatic generation of simulation workflows.*
- ◇ *Integrate the Workflow Management System into an engineering design framework that supports Knowledge Based Engineering applications and Multi-disciplinary Design Optimisation.*

1.7 Contributions of this Research

This research presents new step-by-step instructions for modelling simulation workflows in a top-down approach, starting with defining the design problem. For understanding the rationale of the workflow, it is important to know what the overall objective is (which is defined in the problem statement) and what decisions have been made in the modelling process. These decisions are captured while going through these steps. As a result, a logical link is maintained from the problem statement to the final simulation workflow.

A second contribution is new Problem-, Discipline-, and Tool-forms to complement MOKA's ICARE-forms. The original set of forms are insufficient to capture specific knowledge in the simulation workflow and design problem domains.

Another contribution is a coupling between a Knowledge Base (KB) and a PIDO system. The coupling uses Model Driven Software Engineering (MDSE) techniques to automatically generate simulation workflows. This saves development time and simultaneously shifts the focus from modelling on a PIDO platform to modelling knowledge, thereby lowering the threshold for non-experts.

Accessibility is further enhanced by the final contribution, which is the High-Level Activity (HLA). HLAs are activities that consist of five primitives: *input*, *preprocessing*, *analysis*, *postprocessing*, and *output*. With this definition, a new framework has been created that forms the basis for every activity. It captures process knowledge to enable parametric process modelling, so that engineers can easily build workflows with reusable building blocks. The idea is similar to KBE's capability to perform parametric product modelling.

1.8 Outline of the Report

This research involves an array of technologies. For a better understanding of the results, readers are advised to read Chapter 2, which explains the methodologies and technologies that are applied in this research. Topics include Knowledge-Based Engineering (KBE),

Workflow Management (WFM), various knowledge technologies (e.g. semantic web technologies and rule-based reasoning), and two engineering frameworks (the DEE and IDEA). Then, Chapter 3 begins with a short introduction into MOKA, followed by an analysis to point out the strengths and weaknesses of MOKA. This is the starting point for the new methodology, named MOKA 2, which has been tailored to support both KBE and WFM. Chapter 4 describes this improved methodology that has been developed during this research.
The thesis proceeds with the technical implementation of the methodology in an integration framework in Chapter 5. It shows the system's architecture and explains the various ways of implementing the framework.
Chapters 6–8 describe the several use cases that demonstrate the capabilities of the new methodology and framework through some examples. The objective is to clarify the steps that the end user has to take when applying the new methodology, and to show the end results that are achieved. This is done for two levels of difficulty, starting with a relatively simple product packaging optimisation and ending with a thermoplastic injection mould design.

Chapter 2

Enabling Methodologies and Technologies

Chapter 1 explained that an automation framework for solving MDO problems is needed to overcome a variety of challenges. This framework should use KBE and SWFM technologies to automate design optimisation and provide tools for capturing engineering knowledge. The Design and Engineering Engine (DEE) is an engineering design framework that uses KBE technologies to automate design optimisation. However, it lacks the tools for knowledge management. Therefore, a new framework is developed as a successor, which is called the Integrated Design and Engineering Architecture (IDEA).

This chapter describes the technologies that are applied in the IDEA. But because *knowledge* is a recurring term in this research, Section 2.1 explains first what knowledge exactly is. Then, Section 2.2 describes KBE, one of the main technologies in the framework, before the DEE is discussed in Section 2.3. The latter section ends with the shortcomings of the DEE. The sections that follow describe what new technologies are needed to improve on these shortcomings. This begins with a set of technologies for storing knowledge in a Knowledge Base (KB) (Section 2.4). It explains technologies such as RDF, triple stores, querying (SPARQL), ontology modelling (OWL), and reasoning. This is followed by Section 2.5, which explains how Workflow Management and web services can be solutions for managing processes in distributed environments. Finally, Section 2.6 ends the chapter by describing the IDEA itself.

2.1 Knowledge in Context

Knowledge management literature seems to be undecided about the definitions of *data*, *information*, and *knowledge*. For a better understanding of this research it is important to clarify their meaning in this context. Rowley (2007) analysed the different formal definitions for each concept, and found some coherence. Summarised, it can be stated that:

- **Data** is usually defined by its lack of meaning or value. It is unorganised and unprocessed, and stands at the bottom of the hierarchy.
- **Information** is organised and structured data. It is data that has been processed for a purpose, which makes it meaningful, valuable, useful and relevant.
- Discussions around **knowledge** have not led to a clear definition of the concept. However, most definitions are variations that include one or more of these terms: information, understanding, skills, experience, capability and expertise. In that regard, a suitable definition is: *"Knowledge is data and/or information that have been organized and processed to convey understanding, experience, accumulated learning, and expertise as they apply to a current problem or activity."* (Turban et al., 2005).
- **Wisdom** in the knowledge management context is the most ambiguous concept. In fact, while not always justified, it is often omitted in literature. Perhaps because it is more involved with human intuition than with systems. In general, wisdom is described as applying obtained knowledge from one domain to a new situation, occasionally infused with ethical judgement.

Milton (2007) categorises knowledge further into two different views: it can be *procedural* or *conceptual*, and either *explicit* or *tacit*. But Milton emphasises that in reality there are no clear boundaries between two extremes. The transition is continuous.
Shortly stated, **procedural knowledge** answers to *"I know how to ..."*, and is therefore related to tasks. **Conceptual knowledge** completes sentences starting with *"I know that ..."*, and relates to concepts, their properties and their relationships. On the other side, **explicit knowledge** describes knowledge that can be easily explained and documented, and is transferable without much effort, such as facts. **Tacit knowledge** is associated with knowledge gained through experience and skills. It is hard to explain with just words and is therefore most difficult to capture. A great example of tacit knowledge is the ability to ride a bicycle.
KBE is a discipline where knowledge is captured and reused in engineering applications. The technology is described in the next section.

2.2 Knowledge Based Engineering

Knowledge Based Engineering (KBE) is a technology that uses intelligent software systems for the design of engineering products with minimal effort from the design engineer. These systems are capable of capturing and reusing product and process knowledge in a specific domain, hence reduce development time and cost by automating repetitive and non-creative design tasks.
There are numerous definitions for KBE, which demonstrates that the technology is still evolving. Perhaps the most notable statement is made by Ammar-Khodja et al. (2008): "*In reality, there is no unambiguous definition of KBE. However, most of them are similar.*" Ammar-Khodja et al. found in most definitions that KBE systems are used for solving design problems and are capable of storing design knowledge for automating (parts of) the design process. But as KBE advanced, the technology moved its focus away from geometry manipulation. More recent definitions take a wider approach and define KBE by its ability to use design knowledge to automate repetitive tasks. So instead of describing what it is, it may be more appropriate to define KBE by what it is supposed to do.

Verhagen et al. (2011) have expressed this as follows:

> "*The objective of KBE is to reduce time and cost of product development, which is primarily achieved through automation of repetitive design tasks while capturing, retaining and re-using design knowledge.*"

The Roots of KBE
KBE merged from CAD technologies (Computer Aided Design), AI techniques (Artificial Intelligence), and object oriented programming. Its roots go back to the 1970s when Knowledge Based Systems (KBS) entered the market. KBSs are computer systems that store knowledge in a specific domain, and solve problems through reasoning over this knowledge. Many successful KBS have been developed in a variety of fields, such as healthcare and chemistry, but it never became a real success in engineering applications. Mainly because KBSs miss two crucial qualities for engineering: the ability to manipulate geometry and process engineering data (La Rocca, 2012). Since most of the engineering design work involves both these activities, KBSs were deemed unsuitable for the job. Only years later, when ICAD was first introduced, KBE made its first appearance.
Now, nearly thirty years later, it can be concluded that KBE has not made the same impact on the engineering design community as CAD has. The next section discusses the experiences with KBE thus far.

2.2.1 An Evaluation of KBE Technologies

Since its introduction, mostly the aerospace and automotive industry have benefited from KBE technologies (van der Laan, 2008; Corallo et al., 2009; Chapman & Pinfold, 2001). More than two decades of KBE development has passed, and users gained experience with the technology. Verhagen et al. (2011) reviewed fifty of the most influencing research papers on KBE and found that using KBE systems has these advantages:

- Development costs are reduced by automating design processes.
- Automation of repetitive and non-creative tasks has freed up time for more creative processes (see Figure 2.1).
- Knowledge has become more accessible via a shared knowledge base.
- The knowledge management phase is a great opportunity to review available knowledge within the organisation.

Generally, KBE is conceived as a promising technology with potential to replace traditional CAD systems. It is far more capable than CAD mainly because knowledge is inherently tied to the product model in KBE. This proved to be valuable for performing MDO in an automated design framework. A key requirement for applying MDO successfully, is a parametric geometry generation system (Bowcutt, 2003). Bowcutt analyses essential characteristics of a parametric geometry generation system and explains why CAD is not ideal for MDO. Some key points:

- A CAD model does not contain knowledge that is necessary for creating different 'views' of the model for each analysis discipline.
- Integrating sub-system components involves rules and relationships between different components. Some CAD systems provide this functionality, but most do not.

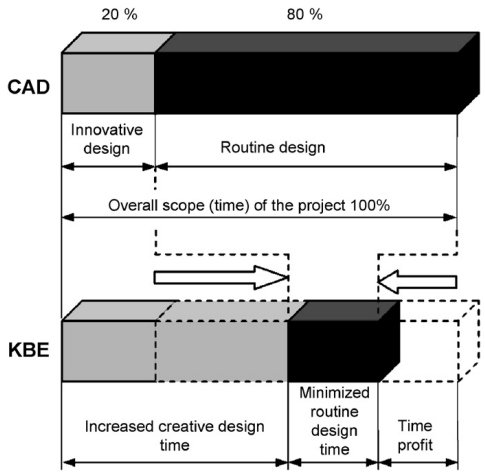


Figure 2.1: Time allocation in CAD- and KBE-oriented design processes (source: Skarka, 2007)

- In some cases the physics determine how curves and shapes should be constructed. However, CAD is incapable of storing this knowledge in the model.

Taken into account that automation is a must, it was concluded that CAD is not suitable for MDO. Boeing then created the General Geometry Generator (GGG), which is a general purpose geometry generation tool. It is very similar to the Multi-Model Generator (MMG) of the Design and Engineering Engine (DEE) that uses KBE technologies to realise a parametric geometry generation system (see Section 2.3). KBE is capable of storing knowledge and performing computations that are necessary in an automated MDO framework.

Despite the advantages, KBE never launched itself to a commercial success. There are several serious shortcomings that complicate the implementation of KBE systems. Verhagen et al. summarised these shortcomings and proposed future challenges for KBE research (see Table 2.1). The interested reader is encouraged to read the review, where these shortcomings and future challenges are thoroughly discussed.

Table 2.1: An evaluation of current KBE researches (source: Verhagen et al., 2011).

	Current shortcoming	Future challenge
1	Case-based, ad-hoc development of KBE applications	Improve methodological support for KBE
2	A tendency toward development of 'black-box' applications	Moving beyond black-box KBE applications
3	A lack of knowledge re-use	Effectively sourcing and re-using knowledge
4	A failure to include a quantitative assessment of KBE costs and benefits	KBE success metrics
5	A lack of a (quantitative) framework to identify and justify KBE development	Assessment framework for KBE opportunities

2.3 Design and Engineering Engine

The Design and Engineering Engine (DEE) is one of the few methodologies for KBE development (see shortcoming 1 in Table 2.1). It is an aircraft design framework, developed by La Rocca (2011), consisting of four key components: the *Initiator*, *Multi-Model Generator (MMG)*, *Analysis tools*, and *Converger and Evaluator* (see Figure 2.2).

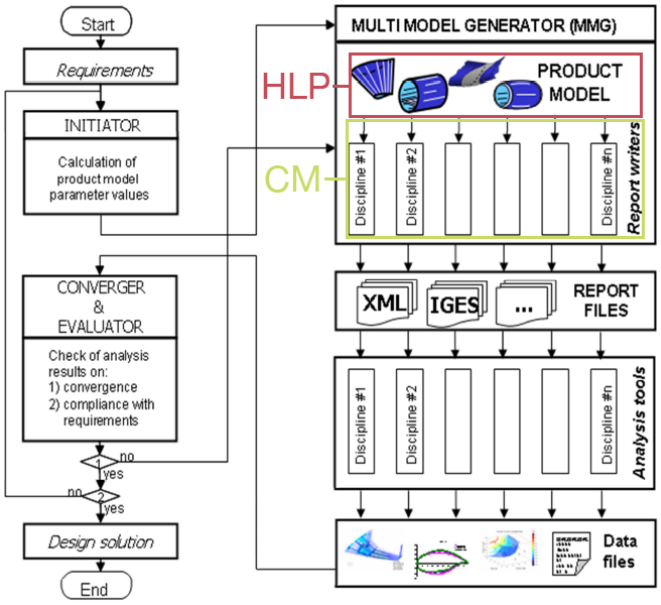


Figure 2.2: The Design and Engineering Engine (DEE) (source: La Rocca, 2011). The High Level Primitives (HLP) and Capability Modules (CM) are highlighted in the diagram.

The goal of the **Initiator** is to find a feasible solution within the design space. In more complicated problems the Initiator can perform a simplified optimisation to find an initial solution before it is provided to the MMG. The **MMG** is the core of the DEE, formalised into a KBE application. It is a parametric modelling engine that can quickly output any configuration using High Level Primitives (HLP). Unlike CAD-primitives (points, lines, surfaces, etc.), HLPs contain knowledge. For example, one of the HLPs is a *wing-part*. Instead of drawing the wing geometry using lines and surfaces, the design engineer provides values for parameters such as the sweep, chord length, twist, and airfoil data. The MMG automatically generates the geometry based on the input. Using this approach, the design engineer can rapidly create an aircraft's main wing, winglets, vertical and horizontal tail surfaces based on a single wing-part HLP (see Figure 2.3).
Besides the wing-part there are three other HLPs in the aircraft domain: a *fuselage-part*, *engine* and *connection-element*. La Rocca (2011) has shown that a wide range of aircraft configurations can be generated with a combination of only these four HLPs.

The HLPs do not only contain geometric knowledge, but also knowledge from other disciplines, such as simulations, manufacturing or cost. Therefore, it is relatively easy to create different *"views"* of the product model. This is another strength of the MMG, which uses Capability Modules (CM) to automatically generate this view. This can be

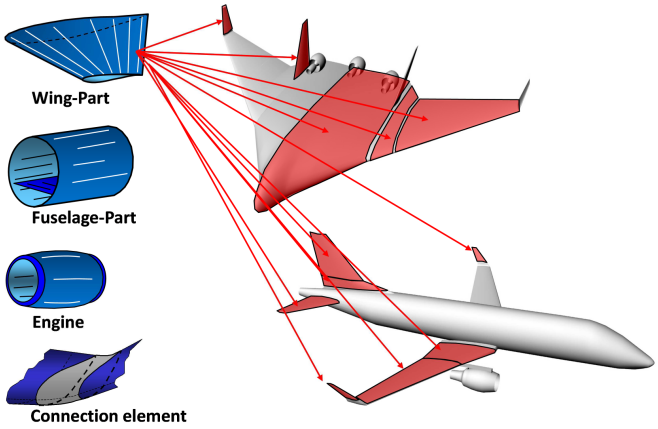


Figure 2.3: The HLPs are adaptable for modelling different aircraft configurations (source: La Rocca, 2011)

a mesh for FEA or CFD, or product information for cost analysis. Because these views are derived from the same product model, the engineer does not have to spend time on creating alternate models and can instead focus on further improving the "master" model. The modules automatically generate files that serve as the input to the **Analysis tools**. Finally, the **Converger and Evaluator** analyses the results and determines whether the optimal and feasible solution is found, or that the optimisation continues with the next iteration.

The use of KBE technologies in a design framework is truly a step forward in automated design optimisation. The capability modules greatly reduce time spent on repetitive and mundane work that is normally done by the engineer. Despite these efforts, the DEE has not solved all problems. There are, for instance, no tools or methods for capturing and storing knowledge in a KB. Thus, the knowledge is hidden in the code, making it more difficult to share and reuse, and also more problematic for non-programmers to understand. Furthermore, the DEE leans towards KBE rather than WFM with its focus on product knowledge in the MMG and CMs. It does not include a methodology or framework for using process knowledge.
For these reasons, a new framework has been developed that improves on the DEE. This framework, the IDEA, requires new technologies that are discussed in the following sections, starting with technologies for a KB.

2.4 Knowledge Technologies

Next generation design systems should store engineering knowledge explicitly in a KB. The Extensible Markup Language (XML) is a popular format for data storage and data transfer, designed to be human-readable and machine-readable. It is a standardised syntax that uses (customisable) tags for describing data. XML files can be accompanied by XML Schema files that express a set of rules for validating the XML document, thus guaranteeing that the XML document can be parsed correctly by machines. Even though this may sound like a perfect solution for storing knowledge, it is in fact not an ideal

technology.

XML is indeed a standardised syntax that can be read by both humans and machines. However, only humans are able to **understand** what is written in the XML document. Consider the phrase *"The author of the page is Ora"*. In XML this can be written as (source: Berners-Lee, 1998):

```
<author>
    <uri>page</uri>
    <name>Ora</name>
</author>
```

Or as:

```
<document href="page">
    <author>Ora</author>
</document>
```

Or maybe as:

```
<document>
    <details>
        <uri>href="page"</uri>
        <author>
            <name>Ora</name>
        </author>
    </details>
</document>
```

And even within a single tag:

```
<document href="http://www.w3.org/test/page" author="Ora" />
```

These examples are all valid XML documents. Even though it is clear to humans that Ora is the author of the page in all four examples, a computer does not understand this so easily. Computers are able to parse the XML statements, but the various document structures and different tags are confusing for a computer. In other words, the computer is able to read XML, but not able to understand it. Automation of design optimisation, where various heterogeneous tools are involved, is much easier when computers are able to understand what is stored. Therefore, this research proposes a new set of knowledge technologies for storing engineering knowledge.

2.4.1 Storing Knowledge

The most popular database to date is the relational database. But there is another database technology that is preferred for storing knowledge: the triple store. The following sections explains why triple stores are used in this research, beginning with explaining how the meaning of data can be stored.

Resource Description Framework

One of the biggest advancements in web technologies is the development of **the semantic web**. The main driver for this development is that data should be stored with their semantics. Whereas **syntax** is related to how to describe data, **semantics** is about the meaning of data. For example, in Figure 2.4 the sentence *"I love the web"* has been

written in two different ways. Although clear to humans, a computer does not understand that these two sentences are related.
When computers know the meaning of the data, information can be found, shared and linked more easily. On top of that, machines will be able to communicate with other machines. Therefore, semantics is considered as *"the next big thing"* in web technology development.

Figure 2.4: The sentence "I love the web" is written in two different ways. The syntax is different, but the semantics are the same.

The internet known today is a collection of data, presented in a human-readable form. A computer knows how to display this data as web pages, but does not understand its contents. The semantic web changes the internet environment from a web of documents to a *web of entities*. Entities are no longer plain text, but are unique objects or concepts that are related to other entities.
For example, while searching for Paris on the semantic web, the computer understands it is a city. It may then provide suggestions that are useful to the user, such as available hotels, options for travel, events in the city, etc. This would be a great enhancement of current knowledge management technologies. Hane (2010) has already written how searching with semantic technologies has improved results in online healthcare databases. And most certainly, engineering design can benefit from this technology as well.

The semantic web is built on a standardised data structure, called the Resource Description Framework (RDF) (W3C, 2004c). In RDF, entities and their relations are described in the triple format. Triples are simple expressions formed by a subject, predicate and object. For instance

subject	*predicate*	*object*
I	**love**	**the web**

Triples are easily extendible with new triples that form a relationship with other triples (e.g. *"I am a person"* or *"the web is huge"*). Eventually, all these interconnected 'sentences' will form a **web of entities**.

Triple Stores

Databases dedicated to storing triples are simply called **triple stores**. Triple stores are graph databases, where data is stored as *"things (or nodes) and relationships between things."* (Eifrem, 2012). Hence triple stores have no hierarchy (like XML) or tables with columns and rows (like relational databases). It is said to be *unstructured*, which is in fact an intuitive way of representing real-world objects.

Triple stores are recommended to follow the standardised RDF data structure. That aside, since RDF is only a framework, triple stores can be stored in a variety of formats, such as XML, Notation 3 (N3), Turtle, or N-Triples. Switching between database management systems is therefore relatively straightforward, as long as both systems support RDF. Some examples of available systems today are Franz Inc. Allegrograph, Apache Jena, OpenLink Virtuoso, Ontotext BigOWLIM, and Garlik 4store.

Relational Databases

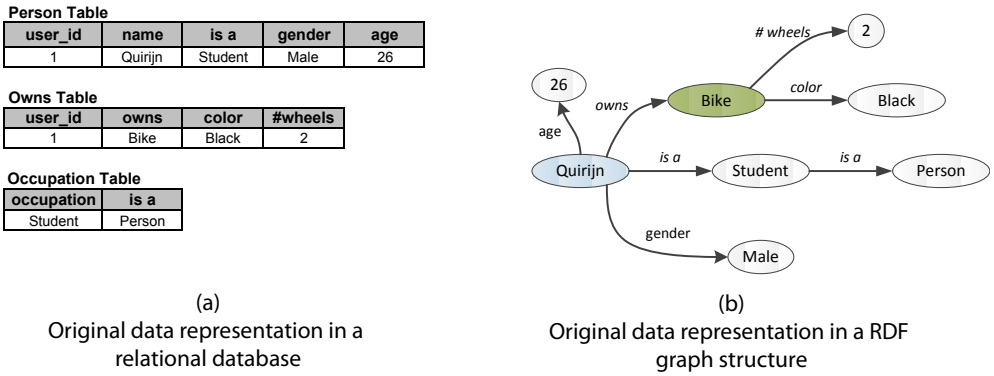
By far the most popular and most widely used type of database to date is the **Relational Database** (RDB). RDBs consist of well-structured tables that each represent real-world objects or events, such as a customer, an inventory item, client's orders, or telephone calls. And similar to how real-world entities have relationships, tables can be linked to each other using unique keys or id's.
Nearly all RDBs use SQL (which stands for Structured Query Language) for managing data. SQL-databases have become the norm mostly because of standardisation of SQL by both ANSI and ISO in the 1980s. This allows companies to easily train new people or find qualified experts. At the same time, companies can easily switch between database companies without losing their valuable data. This drives competition between database companies in terms of features and performance. Well-known RDB systems are IBM's DB2, Microsoft's SQL Server, MySQL, and Oracle databases.

The relational model, introduced by Codd (1970) in the 1970s, is facing heavy competition from modern database models. These models are ironically termed NoSQL ("Not only SQL") solutions and can be placed into four categories: *key-value databases*, *document databases*, *wide-column databases*, and *graph databases* (Bendiken, 2010). Of all the NoSQL solutions only RDF-based graph databases (or triple stores) are standardised, which proved to be essential for a high adoption rate when SQL was introduced. Therefore, only triple stores are considered as a contender in the comparison with RDBs.

Comparison Between Relational Databases and Triple Stores

In terms of flexibility, RDBs are no match for triple stores. The structured character of RDBs requires a schema that needs to be defined before the database can be populated with data. A schema defines all the structural elements of an RDB, such as the tables, fields, and relationships (see Figure 2.5 (a)). Think of it as how the layout of spreadsheets in Excel need to be predefined before it can be filled.
Triple stores are graph databases and consist only of nodes and relationships between nodes (see Figure 2.5 (b)). Because triple stores are unstructured, there is no schema that defines how data should be stored in the database. Unlike RDBs, where it is clearly defined in advance what data each table can store, nodes are added to a triple store without any particular order or layout. This may seem unnatural at first, but it has its advantages when changes are made to the database.

Once the schema of an RDB has been defined, making changes to it is rather problematic. For instance, Figure 2.6 shows that Quirijn now owns a cat and a new car. In the triple store this data is simply added by creating new nodes and adding the relationships to



Person Table

user_id	name	is a	gender	age
1	Quirijn	Student	Male	26

Owns Table

user_id	owns	color	#wheels
1	Bike	Black	2

Occupation Table

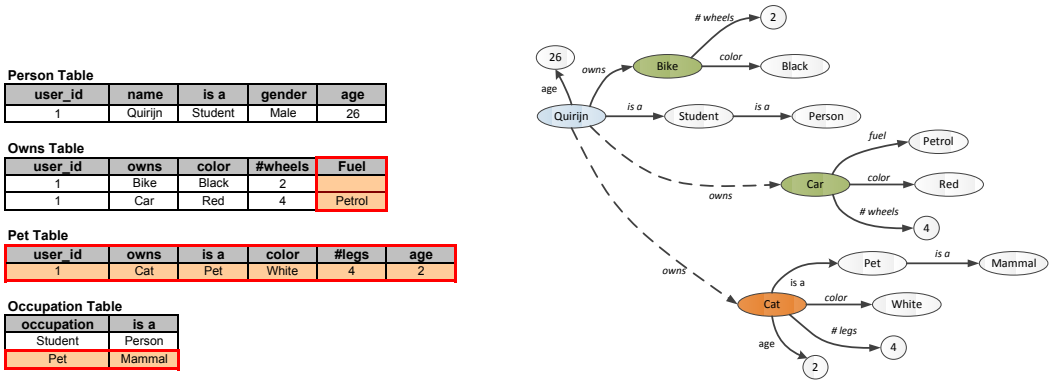
occupation	is a
Student	Person

(a) Original data representation in a relational database

(b) Original data representation in a RDF graph structure

Figure 2.5: Data and relations represented in (a) a Relational Database and (b) an RDF graph (source: Landman, 2011).

the original graph. However, in the RDB this change is not so straightforward. First, the original schema did not take into account that 'persons' can own 'pets'. Therefore, the schema needs to be modified by adding a new *Pet table*, defining what that table contains, and then the data can be added to the table. Another change is made in the *Owns table*. Previously, this table has been designed to contain data for a bike. Now that a car has been added, the table needs to be extended with a column that contains the information about fuel. This example demonstrates that RDBs require significantly more effort in managing the database structure than triple stores.



Person Table

user_id	name	is a	gender	age
1	Quirijn	Student	Male	26

Owns Table

user_id	owns	color	#wheels	Fuel
1	Bike	Black	2	
1	Car	Red	4	Petrol

Pet Table

user_id	owns	is a	color	#legs	age
1	Cat	Pet	White	4	2

Occupation Table

occupation	is a
Student	Person
Pet	Mammal

(a) Altered data in a Relational Database

(b) Altered data in a RDF graph structure

Figure 2.6: Extended data and relations represented in (a) a Relational Database and (b) a RDF graph (source: Landman, 2011).

Another advantage of using triple stores is the ability to perform complex queries (i.e. retrieving data from the database). Aasman (2012) used this example to demonstrate the capabilities of triple stores: *"Find all meetings that happened in November within 5 miles of Berkeley that was attended by the most important person in Jans friends and friends of friends."* In RDBs it is much more difficult to execute similar queries over several degrees of separation, where connections between two things can only be found through multiple relations. This is because triple stores are capable of doing inferencing and rule processing, thus finding new connections that are not explicitly stated in the database.

The following sections on querying and reasoning go into more detail on these topics.

In practice however, RDBs still have an advantage over triple stores. RDBs have been on the market much longer, hence are more mature and offer more features. In terms of performance and robustness, RDBs are still ahead, but triple stores are catching up quickly. In 2011, Franz Inc. achieved to load a triple store with over *one trillion triples* using Allegrograph (Franz Inc., 2011). This was a major achievement and it is a great step towards making triple stores suitable for enterprise applications.

Querying

Querying in computing means retrieving data from a database, spreadsheet, document, or any other digital source containing data. In the context of this research, querying means retrieving data from a triple store. Instead of using keywords for search, like in Google's search engine, databases provide a specific query language that allows users to retrieve data more effectively. For RDF data there is a query language called **SPARQL**, which stands for *SPARQL Protocol and RDF Query Language*. It complements the set of semantic web technologies and is therefore standardised by the W3C as well (W3C, 2008). SPARQL is for triple stores what SQL is for RDBs. In fact, even the syntax shows similarities (see Listings 2.1 & 2.2, source: Prud'hommeaux (n.d.)).

Listing 2.1: SQL example answering the question "What is the address of every person living in Massachusetts (MA)?"

```
SELECT Person.fname, Address.city
FROM Person, Address
WHERE Person.addr=Address.ID
AND Address.state="MA"
```

Listing 2.2: SPARQL example answering the question "What is the address of every person living in Massachusetts (MA)?"

```
SELECT ?fname ?city
WHERE {
      ?who <Person#fname> ?fname ;
           <Person#addr> ?addr .
      ?adr <Address#city> ?city ;
           <Address#state> "MA"
}
```

The SQL query in Listing 2.1 retrieves the first name (*fname*) from the *Person table* and *city* from the *Address table*. The keywords can be explained as follows:

SELECT: specifies which column of which table.
FROM: indicates in which tables to search.
WHERE: is a condition that *'joins'* the two tables through *'keys'*.
AND: is another condition that the state is Massachusetts.

Figure 2.7 shows how the two tables are *'joined'* together by their *'keys'*. The *Person table* has a foreign key (*addr*) that matches the primary key of the *Address table* (*ID*).

SPARQL uses similar keywords as SQL, but here the functionality is slightly different.

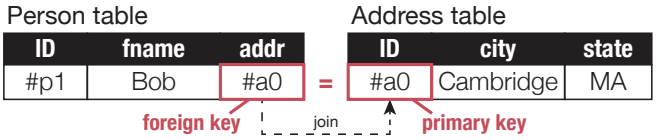


Figure 2.7: Relational database tables that are *'joined'* together through *'keys'*.

SELECT: specifies which variables are returned.
WHERE: specifies a graph pattern and may contain variables (terms starting with '?') for the subject, predicate, or object.

SPARQL tries to match the graph pattern specified in the WHERE statement with the graph in the triple store. The variables (starting with '?') can be any entity that matches the pattern. Once a pattern is found that fully matches the query, the entities that map onto *?fname* and *?city* are returned.

Since SQL has been designed for traversing tables and SPARQL for graphs, there are key differences between the two. With SPARQL it is much easier to retrieve linked data. There is no need to understand the underlying structure of the tables in order to write the query. In SQL it is necessary to specify from which table what information is retrieved, thus making it more difficult to achieve the same result.
SPARQL has an impressive feature that allows users to query from different sources, so-called SPARQL endpoints. Currently, the user can use the SERVICE keyword to redirect parts of the SPARQL query to the SPARQL endpoints that contain the data. This is in fact similar to traversing the structured data of RDBs, where the query is directed at different tables to retrieve the desired data. However, with SPARQL the user can write the query without knowing the foreign database's schema in advance. Future solutions may include intelligence (e.g. semantic metadata for each SPARQL endpoint) that enable systems to find the correct sources automatically without specifying the SERVICE keyword. These solutions have not been fully developed yet. Therefore, the SERVICE keyword has been included to improve query performance at the cost of ease of use.
Additional features as FILTER, ORDER, and OPTIONAL replicate functionality available in SQL.

2.4.2 Ontology Modelling

Now that the semantic web has adopted ontology modelling, it is reaching a much larger audience. The term ontology was first used in philosophy, where an ontology *"is the philosophical study of the nature of being, existence, or reality, as well as the basic categories of being and their relations."* (Wikipedia, 2012b). In computing, ontologies first appeared in the AI domain, albeit with a slightly altered definition. In short, *"An ontology is an explicit specification of a conceptualization."* (Gruber, 1993). But over the years, ontologies have become a medium for sharing domain knowledge among experts. And thus, the definition changed accordingly: *"An ontology is a formal explicit description of concepts in a domain of discourse (classes), properties of each concept describing various features and attributes of the concept (slots), and restrictions on slots."* (Noy & McGuinness, 2001).

An ontology defines a **common vocabulary** for expressing, sharing, and reusing knowledge in a certain domain. It describes for a domain the terms, concepts, and their relationships, and provides formal definitions that constrain the interpretation of these terms. Having explicit formal definitions removes any ambiguities of terms and allows new users to the domain to understand their meaning. The simplest way to illustrate the importance of ontologies is to compare it with the English language. Because English has been accepted as the world language, many people from different countries have learned English as their second language. Now people can easily communicate with each other and share information and knowledge using this common language. This is similar for sharing knowledge within a domain and also for the communication between computers.
Consider two websites that provide information in the same domain, for example wines. If these websites share the same underlying ontology, it will be simple for a user or software agent to extract the needed information. Whether it is a webshop selling wines or an encyclopedic website describing wine history, both will have Cabernet Sauvignon classified as a red wine. Furthermore, the ontology may include certain properties of wines, such as body, flavour, or even its origin, which are all retrievable from either website using the common vocabulary.
Ontologies can also be shared among different domains. For example, the origin of the wines may be connected to another ontology specialised in geographic locations. If this is a standardised ontology, used widely on the internet, it may even describe geographic locations for totally unrelated subjects, such as travel websites. It is much more efficient to have one geographic data ontology and reuse it for all websites, than to have each website develop their own geographic ontology.

The W3C has standardised RDF Schema and the Web Ontology Language (OWL) as general-purpose languages for modelling ontologies (W3C, 2004b,a). The ontologies developed in this research use both RDFS and OWL to model the domain. Some purposes the languages can be used for are:

- categorisation using classes and subclasses.
- defining property and cardinality restrictions, e.g. cars have four wheels.
- annotation of concepts.
- explicitly express equivalence, e.g. class *'Cars'* is equivalent to *'Automobiles'* class (useful for integrating ontologies that share a domain but use different terms).

Another standardised general-purpose modelling language that has similar features is the Unified Modeling Language (UML). Designed for modelling object-oriented programming code, UML also has classes, relationships between classes (called associations), and multiplicity (similar to cardinality restrictions). However, there is one significant difference between UML and OWL. UML is designed for programming, whereas OWL is designed for modelling the 'world' (or at least a particular domain). Their intention is different, and therefore their functionality. UML defines classes and creates objects based on those classes. For example, *Supercar* can be a class of cars with a top speed higher than 250 km/h. Then, an instance of the class can be created, *car A*, with a top speed of 300 km/h. In OWL, classes are not templates for creating objects, but rather categories that classify objects. It is possible to create objects first that are not based on a class. Thus, in OWL *car A* can be created first with a top speed of 300 km/h. Afterwards, a new category is introduced, which is the class *Supercar*, that specifies that cars belonging to that class have a top speed higher than 250 km/h. Then, through reasoning it can be determined

that *car A* belongs to the class *Supercar*.
To conclude, the advantage of using semantic web technologies is the ability to reason over knowledge to infer new facts. For this purpose, a language is needed that is based on formal logics. OWL is based on logics, whereas UML is not. Therefore, OWL is a more suitable language for modelling ontologies.

2.4.3 Reasoning

KBS use reasoning techniques from AI for inferring new facts that are not explicitly mentioned in data. Formally, reasoning can best be described as: *"Reasoning allows one to infer implicitly represented knowledge from the knowledge that is explicitly contained in the knowledge base."* (Baader et al., 2003). Two types of reasoning have been applied in this research: *Description Logic (DL) reasoning* and *Rule-based reasoning*.

Description Logic Reasoning

Knowledge Representation (KR) is an area that researches how knowledge should be represented as to facilitate reasoning on it. Description Logic (DL) is a family of KR languages, that is a subset of first-order logic. DLs represent the knowledge in a particular domain by modelling the concepts and their relationships. DL was first used in AI for formal reasoning over knowledge stored in a knowledge base. Now, it gained popularity because DLs are part of the semantic web.

DLs follow an Open-World Assumption (OWA) rather than a Closed-World Assumption (CWA), meaning that any statement that is not known is unknown instead of false. For example, if the knowledge base has a fact stating that `hasChild(PETER; HARRY)`, then in a CWA this is understood as Peter having only one child, Harry. On the other hand, in an OWA the facts say that Harry is a child of Peter. The difference is that in the OWA it is not conclusive whether Harry is the only child of Peter or that Harry has siblings. Consequently, it is not possible to state in an OWA that all children of Peter are men, whereas in a CWA that is true. The only way to do this in an OWA is to say explicitly that Peter has only one child, $(\leq$ `1 hasChild)(PETER)`. Because an OWA may have many interpretations, answering a query is more complex, and may require further analysis of the answers (Baader et al., 2003).

Normally, Description Logic reasoning is used for decision problems, which are related to one of these questions (Wikipedia, 2012a):

- **Instance checking:** is a particular instance a member of a certain class?
- **Relation checking:** does the relation exist between two instances?
- **Subsumption:** does a concept belong to the subset of another concept?
- **Concept consistency:** is there no contradiction between any of the definitions?

The example in Listing 2.3 demonstrates how DL can be used to detect inconsistencies (source: Bechhofer, 2003).

Listing 2.3: Mad cows are inconsistent; A DL example.

```
Class(a:cow partial a:vegetarian)
Class(a:vegetarian complete intersectionOf(
  restriction(a:eats allValuesFrom (complementOf(restriction(a:part_of
    someValuesFrom (a:animal)))))
  restriction(a:eats allValuesFrom (complementOf(a:animal))) a:animal))
Class(a:mad_cow complete
  intersectionOf(a:cow restriction(a:eats
    someValuesFrom(intersectionOf(restriction
        (a:part_of someValuesFrom (a:sheep)) a:brain)))))
Class(a:sheep partial a:animal
  restriction(a:eats allValuesFrom (a:grass)))
```

Simply explained, these facts state that:

- *Cows are naturally vegetarians*
- *Vegetarians do not eat animals or parts of animals*
- *A mad cow is one that has been eating sheeps brains*
- *Sheep are animals*

DL is able to derive that a mad cow has been eating part of an animal, which is inconsistent with the definition of a vegetarian (Bechhofer, 2003).

OWL, one of the ontology languages for the semantic web, has three variants with an increasing level of expressivity: OWL-Lite, OWL-DL, and OWL-Full. OWL-Full is the most expressive variant and is in fact not based on DL. It is undecidable, but it has been added to maintain some compatibility with RDFS. Both OWL-Lite and OWL-DL are based on DLs, for which sound and complete reasoning is possible.

OWL has additional features to make properties more expressive with the following characteristics: *functional*, *inverse functional*, *transitive*, *symmetric*, *asymmetric*, *reflexive*, and *irreflexive*.

A functional property relates at most one individual to another individual. In Figure 2.8 this means that Peggy and Margaret must be the same individual. Otherwise the statement would be inconsistent.

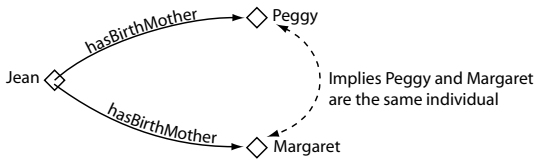


Figure 2.8: An example of a functional property (source: Horridge, 2011)

A transitive property can relate individual A to C via property P, if A is related to B and B is related to C through the same property P. Then through reasoning, it can be concluded that William is the ancestor of Matthew without explicitly stating that relationship (see Figure 2.9).

Finally, a symmetric property simply mirrors the property. So, if Matthew has Gemma as a sibling, then Gemma has Matthew as a sibling too (see Figure 2.10).

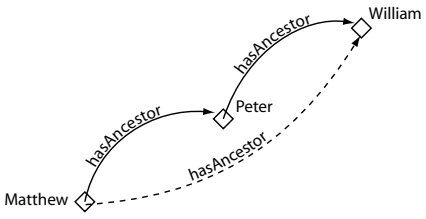


Figure 2.9: An example of a transitive property (source: Horridge, 2011)

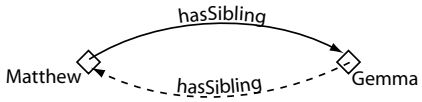


Figure 2.10: An example of a symmetric property (source: Horridge, 2011)

This is a short introduction into property characteristics. The interested reader is recommended to take a look at the Protege manual (Horridge, 2011), where examples are given for the remaining characteristics.

Rule-based Reasoning

DL reasoning has only a limited range of purposes. For instance, it is not possible to automate workflow modelling tasks, such as instantiating lower-level activities (see Section 5.3). Therefore, rule-based reasoning has been applied in addition to DL reasoning.

Rule-based expert systems use reasoning to infer new facts or perform an action based on a set of rules. Rules are written in the **if-then** form, where the *if-part* (antecedent) contains a set of events or facts and the *then-part* (consequent) actions or facts. When the antecedent is satisfied (i.e. all the if-statements are fulfilled), the consequent will be executed. The main component of an expert system is a Reasoning Engine (RE), which uses the rules to reason over a set of facts and deliver a conclusion. Rule-based REs apply either *forward reasoning* or *backward reasoning*.
Forward reasoning engines start with a fact and try to match this fact with the if-part of a rule. If a rule matches, the RE executes the statement in the then-part (the rule has fired). This process repeats until none of the rules matches and a conclusion has been derived. Forward REs are therefore great for executing actions based on facts.
Backward reasoning engines start with a goal, which may answer a query, and will instead try to match it with the then-part of a rule. The RE works its way backward to discover which facts or events need to be satisfied for the goal to be true. Hence, backward REs do not infer new facts or trigger actions, but try to prove a fact (the goal) based on what is known.
The conclusion is that the two methods of reasoning exist for different purposes. In this research, a forward RE has been implemented because the RE is used for deriving facts rather than answering queries.

Although the basics of a rule are simple, there are many classifications of rules. In terms of KBE rules, La Rocca (2011) has identified five types:

1. Logic rules
2. Math rules
3. Geometry handling rules: (a) Parametric rules and (b) Geometric entities
4. Configuration selection rules
5. Communication rules

Examples of these rules are given in Table 2.2 (middle column).

Reijnders (2012), who researched how engineering rules can be captured and stored in the KB, discovered that these rules are not really different types of rules on a formal level. Therefore, Reijnders rewrote the rules in a general form (see Table 2.2, last column).

Table 2.2: KBE rule types identified by La Rocca (2011) and formalised by Reijnders (2012).

Rule type	Example	General form
1. Logic	If $width \leq 1$ then $length = 1$ else $length = 2$	**If** $width \leq 1$ **Then** $length = 1$ **If** not$(width \leq 1)$ **Then** $length = 2$
2. Math	For aeroplanes the lift equals $\frac{1}{2}\rho V^2 S C_L$	**If** object.type = aeroplane **Then** object.lift = $\frac{1}{2} air.\rho object.V^2 object.S object.C_L$
3a. Parametric	$A.length = B.length + C.length$	**If** true **Then** $A.length = B.length + C.length$
3b. Geometric Entity	Define a container as a box with $length = 10$, $width = 20$, and $height = 30$	**If** true **Then** $container.type = box$ $container.length = 10$ $container.width = 20$ $container.height = 30$
4. Configuration Selection	If the wing is longer than $10m$, use 5 ribs	**If** $wing.length > 10m$ **Then** $wing.rib\text{-}sequence.size = 5$
5. Communication	If the loads are not present in memory, calculate and load them	**If** *not(load.status=loaded)* **Then** *execute(retrieve(loads))*

For this research, rules are used for automatic workflow generation. The rules capture workflow modelling knowledge, so that the RE can automate tasks that are normally performed by a human user. The entire list of rules that have been applied is included in Appendix D.
On a formal level, two types of rules have been identified for workflow modelling (see Table 2.3). The first rule type determines the sequence by matching inputs to outputs of activities. The second type is math rules, for modelling the objective function, formulas, and constraints.

Table 2.3: Process rule types for workflow modelling.

Rule type	Example	General form
1. Process sequence	If *Activity2* requires output from *Activity1* as input, then *Activity1* is followed by *Activity2*	**If** $Activity1.output = Activity2.input$ **Then** *Activity1 Activity2*
2. Math	Constraint: $volume < 2$	**If** true **Then** $V < 2$

Rule language

Even though rules are all written in the same form, capturing rules is not as straightforward as it seems. Rules are formal statements expressed in a logic. The logic gives a

meaning to the rule. Multiple logics exist, each being a compromise between *expressivity* and *decidability* (ability to prove a statement). Very expressive logics may have difficulties with proving statements and may not come to an answer. Therefore, the difficulty is to select a logic that is as expressive as possible, yet still decidable, and is able to capture the rules in the domain of interest. Reijnders (2012) has performed an extensive research on engineering rules and concluded that the Rule Interchange Format (RIF) is the most suitable rule language for the IDEA.

2.5 Managing Processes in Distributed Environments

The design of complex engineering systems is increasingly becoming a collaborative task between geographically distributed design teams. The consequence is that design knowledge and simulation tools are spread out over different locations. Managing the exchange of knowledge and simulation results is a real challenge, even with today's computational systems. Two technologies that support the distributed design environment are Workflow Management and web services.

2.5.1 Workflow Management

Workflow Management (WFM) has been introduced briefly in Chapter 1, which did not cover the entire topic. This section explains WFM in more detail, with a particular focus on *human workflows* and *simulation workflows*.

Human Workflows

One of the shortcomings of current SWFM systems is the lack of a standardised, neutral process language, which promotes sharing and reuse of process knowledge. This should be a simple language that is easy to use by anyone without IT and/or workflow expertise. This language is found in the discipline Human Workflow Management (HWFM).

In human workflows, human participants perform activities either manually or with support from information systems. Examples of activities are sending emails, generate or send documents, providing input to systems, etc. The workflow basically forms a blueprint, assisting people in the work that needs to be done (see Figure 2.11). This guarantees that processes are executed efficiently and that results are consistent.

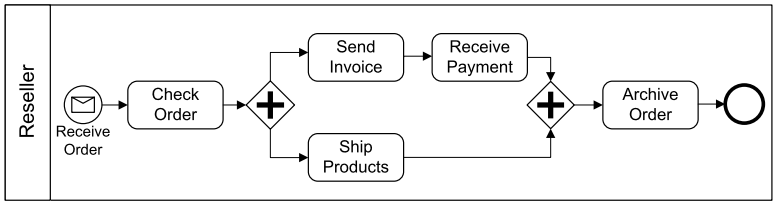


Figure 2.11: A simple example of a human workflow in the BPMN notation. The example shows an ordering process (source: Weske, 2012).

A closely related topic is Business Process Management (BPM), which is not to be confused with WFM or human workflows. The focus is indeed also on human activities, but according to van der Aalst et al. (2003): *"BPM extends the traditional WFM approach by support for the diagnosis phase and allowing for new ways to support operational processes."* (see Figure 2.12).

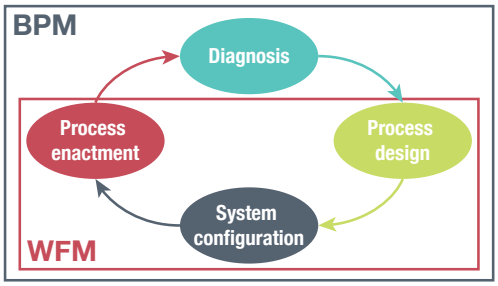


Figure 2.12: BPM extends WFM with the diagnosis phase (source: van der Aalst et al., 2003).

BPM emerged when business users started to experience difficulties in aligning IT systems with the business needs. The solution was rather simple: abstract the business process from the IT layer into a separate, higher level layer. The result is that organisational processes are no longer coupled to IT processes, thus allowing business managers to design processes, analyse process, and improve process efficiency without the need to communicate with IT first. This separates responsibilities in a functional way. A similar approach is necessary in engineering, whether it involves human or simulation workflows, where the responsibility of designing workflows is separated from the implementation of workflows.

BPM is more popular than WFM, and hence better supported. This has resulted in several efforts in standardising notations for modelling business processes and a set of best practices. One of these standards is the Business Process Model and Notation (BPMN), which is a graphical representation for business processes. BPMN gained its popularity because it is a standardised graphical notation, adopted by the Object Management Group (OMG), and it is easy to understand also for non-IT experts. Since version 2.0, BPMN also includes execution semantics and a standardised XML-based syntax. BPMN has become the preferred process language after a trade-off between various notations and languages (see AppendixA and Table 2.4).

Table 2.4: Trade-off of the various workflow modelling languages.

Criteria	Weight	UML	BPMN	EPC	BPEL	YAWL
Graphical notation	1	++	++	++	-	++
Designed for execution	1	-	+	- -	++	++
Standardised	1	++	++	-	++	-
Widely adopted	1	++	++	+	++	-
Easy to use	1	+	+	+	-	0
Score	**max: 10**	**6**	**8**	**1**	**4**	**2**

Simulation Workflows

It is well understood that replacing expensive physical tests with simulations cuts cost of product development. This has led to a massive increase of simulation usage and hence the selection of simulation tools, ranging from full-featured commercial software to in-house developed applications and spreadsheets. As pointed out earlier, the transition to a simulation heavy design process is not effortless. Problems have been identified regarding the shortage of simulation expertise in the near future. Other studies have reported about the high cost associated with the interoperability between heterogeneous tools. And lastly, multiple sources have found that roughly only 20% of the design process is actually spent on creative design tasks (Bazilevs et al., 2009; Stokes, 2001).

Manually executing and chaining simulation tools is a very time consuming and error-sensitive task. The design engineer has to set up simulation, manage data (including translation), and prepare tool-specific views of the product model. This requires exceptional IT skills, which is often not part of the designer's skill set. In these cases the design engineer would greatly benefit from using workflow software.

Workflow software are designed for integrating and automating execution of heterogeneous simulation tools. In the engineering design domain these systems are often referred to as Process Integration and Design Optimization (PIDO) solutions. These software contain all the tools that design engineers need for analysing data and performing optimisation. For example, tools for interfacing with CAE tools, post-processing tools for statistical analysis and data visualisation, and a wide selection of optimisation algorithms are standard features of most PIDO solutions. The rich graphical environment keeps the learning curve low, making it accessible to non-programmers.

Some examples of well-known PIDO solutions are *Dassault Systèmes Simulia iSight*, *Esteco modeFRONTIER*, *Noesis Optimus*, *Phoenix Integration ModelCenter*, and *PIDOTECH PIAnO* (see Figure 2.13, in clockwise direction). There are slight variations among these systems, but the core functionality is the same for all.

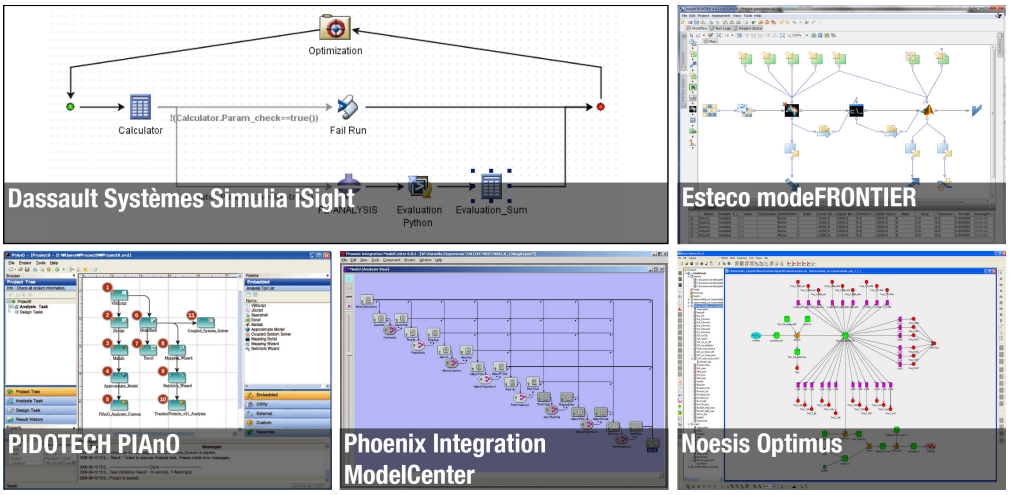


Figure 2.13: Screenshots of various PIDO solutions (source: Dassault Systèmes, n.d.; PIDOTECH, n.d.; Noesis, n.d.; Esteco, n.d.; Phoenix Integration, n.d.; Bowcutt et al., 2008).

2.5.2 Web Services

According to the W3C *"A Web service is a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML based messages exchanged via internet-based protocols."* (W3C, 2002). The less technical explanation is that web services are software applications, which no longer need to be installed on the user's computer, but can instead be accessed over the internet. Or, when drawing the comparison with electricity supply: *"It might help to think of Web services in terms of your electricity supply. You don't generate your own electricity, you just plug in there. The electricity is delivered in agreed standard units and you have a meter telling you how much is being consumed."* (ComputerWeekly.com, 2002).

As organisations grow, and offices are spread all around the world, software applications are developed at different locations. Instead of copying the applications to every other office, an application may be deployed as a web service. This means that the application will be accessible over the internet from any location in the world. Additionally, by using the standard interfaces of the internet (e.g. HTTP) the web service becomes platform-independent, meaning that applications can be used on any operating system and any device that has access to the internet. This is a great solution for solving interoperability problems that are often encountered in WFM.

There are two main technologies that play an important role in the implementation of web services: *Service Oriented Architecture (SOA)* and *Software as a Service (SaaS)*.

Service Oriented Architecture

First of all, SOA is sometimes being confused with SaaS. Most likely because both technologies are often related to web services, but there is a fundamental difference (Laplante et al., 2008). SaaS is a software-delivery model, whereas SOA is a software-construction model. **SOA** is a conceptual architectural framework that describes how a service should be provided and how a service can be accessed through standard interfaces and communication protocols.
In a SOA, there is a service provider that offers a service for others to use (see Figure 2.14). The provider has the task to detail its interface, thus describing the operations of the service and its input and output messages for each operation. Furthermore, a binding description describes how to send messages to where the service is located. The language often used for describing services is the Web Service Definition Language (WSDL). This WSDL description is then published to a service directory, such as the Universal Description Discovery and Integration (UDDI), so that service requesters can look for available services at a central location. This directory often includes additional information about the service provider and the service so that requesters can discover services more easily using search criteria. Once the requester has found a suitable service, it can analyse its WSDL description, and communicate with the service provider directly using the Simple Object Access Protocol (SOAP). SOAP has been standardised by the W3C (2007) as the protocol for exchanging information with web services.

A recent trend is to use Representational State Transfer (REST) instead of SOAP, and the accompanying Web Application Description Language (WADL) in place of WSDL. There

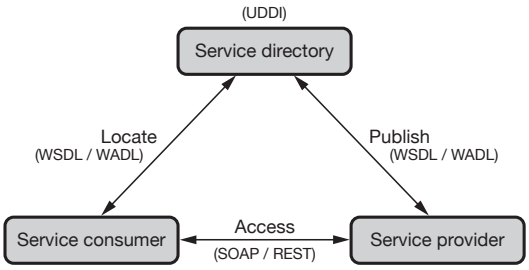


Figure 2.14: The SOA architecture.

are countless discussions on which one is better for web services (Spies, 2008; Francia, 2010; Singh, 2009). Summarising, the main advantages of using REST are:

- *Lightweight:* with REST messages are shorter and require less bandwidth usage.
- *Easy to understand:* REST uses standard HTTP and is much simpler than SOAP.
- *Easy to build:* no tools are required for development.

And the main advantages of using SOAP are:

- *More mature:* SOAP enjoys a better support from other standards (WS-*) and development tools.
- *Rigid:* SOAP offers built-in error handling.

In the end, the decision for either one really depends on the application. In general, REST is favoured by the majority of web service providers, including some of the largest services available today (DuVander, 2011). Throughout this research REST is used because of its simplicity. However, if for instance additional security features are desired, then SOAP is the only option.

Software as a Service

SaaS is thus a software-delivery model, where the software and data are hosted at a central location (often referred to as **the cloud**) and users access it via the internet or intranet. With this model, software no longer needs to be installed on the user's computer and instead users can interact with the software through a thin client, usually a web browser. It is comparable with utility services, where software is delivered as a utility and is charged based on the amount of usage. Next to SaaS, there are many other types of services, which are collectively placed under the term cloud computing.

Salesforce.com was one of the first companies that became successful by offering SaaS for Customer Relationship Management (CRM). While CRM is still the most used type of service, SaaS solutions now exist for human resource management, email, and even for supply chain and inventory control (Weier & Smith, 2007). There are several reasons for using SaaS products. Ease of deployment and management is one of the main advantages. Companies using SaaS are no longer troubled with deploying, maintaining, and updating their software systems. Secondly, SaaS products are often considered to be more flexible

and better support changing business needs. And lastly, companies switch to SaaS solutions because of lower costs. Savings on investments in hardware and on IT personnel can be substantial.

Although SaaS brings many benefits to businesses, there are also companies who have their doubts about SaaS. The main concern of businesses is the security, as business information is generally sensitive. Meanwhile, most businesses have shown their concern regarding reliability and uptime of hosted software, since it is no longer in their control. However, if staying in control is the only problem, then companies might consider Platform as a Service (PaaS) or Infrastructure as a Service (IaaS) as a solution. Both offer more control but still offer some of the benefits of SaaS. Other criticism are related to functionality and interoperability with legacy systems or with other software. But in the end, for most companies these concerns do not outweigh the implementation and cost benefits.

2.6 Integrated Design and Engineering Architecture

The Integrated Design and Engineering Architecture (IDEA) is a new engineering framework that evolved from the DEE. Compared to the DEE, the IDEA has changed the way how knowledge is stored. Because of transparency issues, knowledge is no longer hidden in the application code, but is instead stored explicitly in a KB (see Figure 2.15). The KB is a critical component of the IDEA, as it is a single source for all product and process knowledge, including engineering rules. For the reasons mentioned in Section 2.4, the IDEA uses semantic web technologies for the KB. Then, KBE and WFM applications are built automatically from this knowledge according to the Model Driven Software Engineering (MDSE) methodology.
In MDSE, software applications are developed from abstract models that can be translated into code. This has several benefits. First, the application code is stored in a neutral format, thus allowing applications to be generated on different platforms from the same models. The second reason is that MDSE simplifies the development of new applications because knowledge can be reused more easily. Furthermore, this approach improves collaboration between developers through sharing models and best practices. And lastly, the higher abstraction level makes knowledge more accessible to non-programmers.

Another addition to the IDEA is a Reasoning Engine (RE), that can infer implicit knowledge from the explicit knowledge that is stored in the KB. A rule-based RE was the most logical option, since engineering rules are stored in the KB in a formal rule language (RIF). Other technologies that were used in the DEE remain existent in the IDEA. For example, both KBE and WFM still have a large role in the IDEA (see Figure 2.15). However, WFM is worked out in much more detail than in the original DEE. The original DEE suggested the use of web services, but did not give much implementation details. The IDEA on the other hand relies heavily on web services and maintains a Service Oriented Architecture (SOA) wherever possible. Moreover, the DEE did not provide guidelines on how to implement the design framework in a WFMS, whereas the IDEA presents a clear solution using PIDO software.

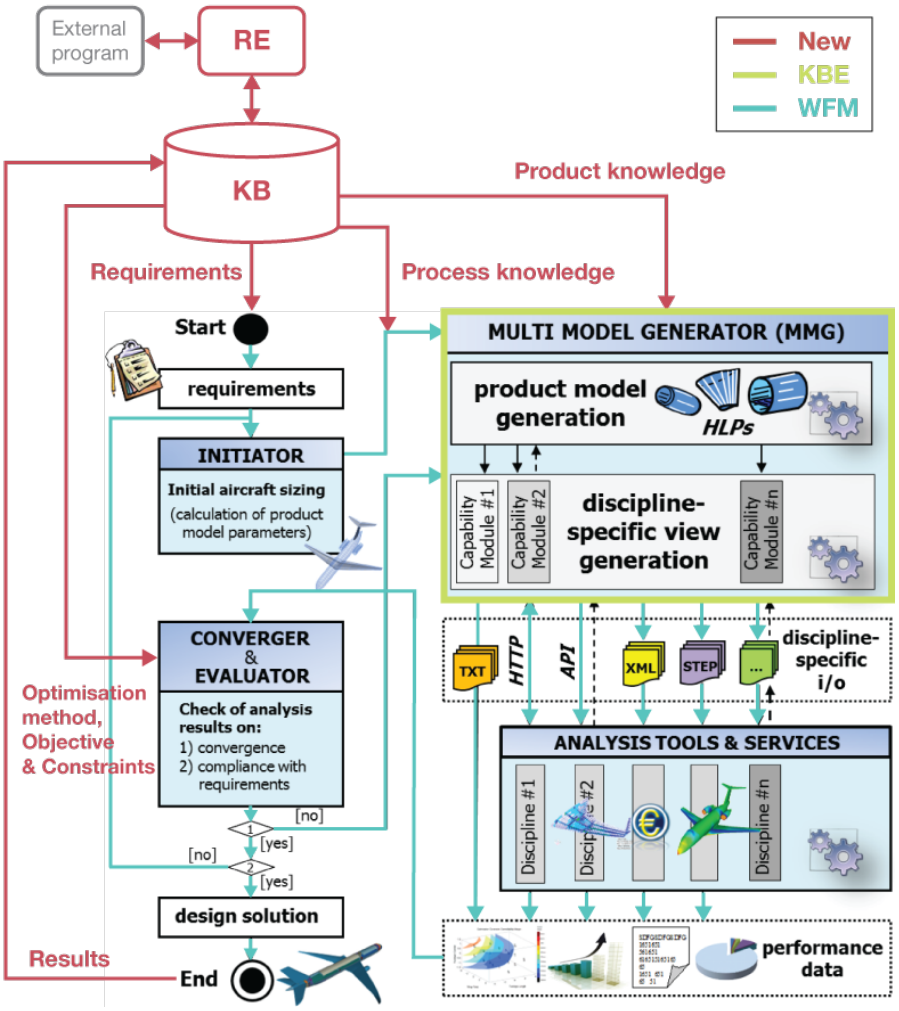


Figure 2.15: Diagram of the IDEA framework showing which components are new (in red) and where KBE (in green) and WFM (in blue) technologies are applied (modified from source: van Dijk et al., 2012)

Chapter 3

MOKA: The Old Methodology

MOKA's attempt to standardise KBE development has not been unnoticed in the KBE community. Today, more than a decade after its introduction, it is clear how the methodology has performed.
MOKA introduced good methods for developing knowledge systems, but there is room for improvement, especially in the domain of WFM. Therefore, the goal here is to find out how MOKA can be improved.

The chapter begins with a short introduction into the original MOKA methodology (Section 3.1), followed by an analysis of the methodology's strengths and weaknesses (Sections 3.2 and 3.3). At the end of this chapter, it is clear what can be reused and what is missing in MOKA. This determines the future direction for a new methodology.

3.1 A Methodology for KBE Development

The development of MOKA started in 1998 as a European initiative with partners from the aerospace, automotive, IT and academic sectors (Stokes, 2001). The aim was to promote the use of KBE in Europe, since Europe was falling behind the competition from America and East Asia. It took thirty months to formalise the MOKA methodology.

The low adoption rate of KBE in Europe was most likely due to the lack of well-defined standardised methods. Thus, MOKA decided to work on a new methodology for developing knowledge systems in engineering design. The main objectives were to (Oldham et al., 1998):

- Reduce the lead times and costs of developing KBE applications by 20-25%
- Provide a consistent way of developing and maintaining KBE applications.
- Develop a methodology which will form the basis of an international standard.

Each of these objectives are described in more detail below.

The bottleneck for KBE application development is the elicitation and formalisation of knowledge. MOKA's objective was to reduce development time and cost by reducing the

required skill level for capturing and formalising knowledge.
The outcome of the project is a formalised methodology for analysing and modelling product and process knowledge in a more consistent form. This allows for KBE applications to be developed and maintained more efficiently, because knowledge is represented more clearly.
Finally, MOKA aimed to become an industry standard. It was expected that a methodology, that is generic and independent of any KBE platform, would lead to wide adoption by the industry.

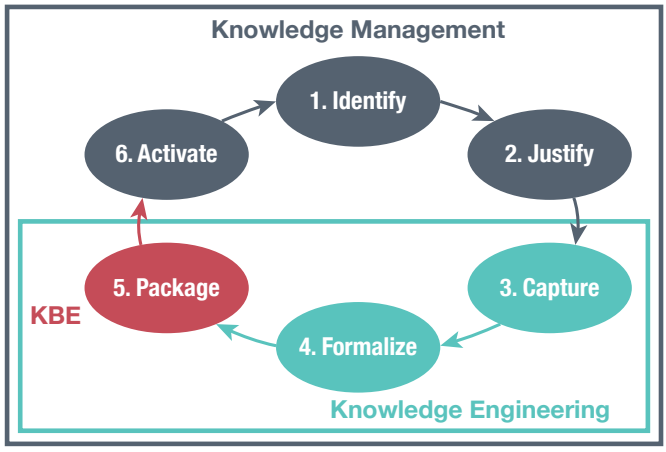


Figure 3.1: MOKA's KBE lifecycle overlaid with the boundaries of *knowledge management*, *knowledge engineering*, and *KBE*.

Although MOKA covers the entire knowledge management lifecycle, termed *KBE lifecycle*, the focus is on the knowledge engineering subset (see Figure 3.1). The main portion of the development costs occurs during the phase where knowledge needs to be captured and structured before it can be applied in an application. Therefore, MOKA has attempted to standardise methods by introducing a stepwise approach to KBE application development, consisting of an **Informal Model** and a **Formal Model**. The process starts with capturing the raw knowledge from the domain experts (*step 3*). This knowledge is filtered, structured and organised in knowledge models as a first step towards KBE applications. For this purpose MOKA designed the so-called **ICARE-forms**, which enable knowledge engineers to structure knowledge in five categories: *Illustration*, *Constraint*, *Activity*, *Rule*, and *Entity*. The set of ICARE-forms describe product and process knowledge at an informal level that is understandable by both the knowledge engineer and the domain expert. This is an important aspect, because the expert has to validate the knowledge before it can be processed in the next step.
This next step formalises the knowledge (*step 4*), meaning that the knowledge is translated into a format that is closely related to computer language. It is the second and last step from raw knowledge to KBE application. The readability of the Formal Model is compromised for experienced engineers (with IT knowledge) and computers. The idea behind this is to decrease the gap between engineering knowledge and computer code, so that code generators can translate the knowledge into a KBE application, while the engineer remains in control of the knowledge.
To maintain transparency, it is important that the knowledge inside the KBE application

can be linked through the knowledge models (informal and formal) all the way to the raw knowledge.

The MOKA methodology is a guideline for developing and maintaining KBE applications, and includes useful tips gained from industrial experience. A decade after the introduction of MOKA it is clear what the impact has been on the KBE community. Below follows an evaluation of its strengths and weaknesses.

3.2 MOKA's Strengths

MOKA has become the most well-known and most used methodology because it addresses many issues encountered in KBE development. Even though many alternative methodologies have been proposed, such as KOMPRESSA (Lovett et al., 2000), KNOMAD (Curran et al., 2010), and a knowledge acquisition methodology by Milton (2007), MOKA prevailed because it sincerely aimed to become the industry standard. In the end, MOKA achieved its status due to the following strengths.

S-1. Methodological approach: Until now, the majority of KBE applications have been developed in a case-based manner (Chapman & Pinfold, 2001; Yang et al., 2012; Bermell-Garcia & Fan, 2002). This complicates the reuse and sharing of applications and the captured knowledge in future projects, which ironically is supposed to be a unique selling point of KBE. This result is not unexpected though, because most KBE platforms focus only on implementation. But as Figure 3.1 illustrated, implementation is only part of the KBE lifecycle. Developing and maintaining KBE applications is a complicated process that starts well before building the application. Therefore, MOKA made an attempt to standardise KBE application development by introducing a methodological approach that covers the entire KBE lifecycle.

S-2. Transparency: Domain experts who provide their knowledge to the knowledge systems are normally also the end users. However, in current practices, where KBE applications are developed ad hoc, end users perceive these systems as black boxes because the knowledge is hidden in incomprehensible code. This complicates validation of the software and ultimately wide acceptance. Therefore, MOKA emphasises to store knowledge explicitly in a Knowledge Base (KB).
But simply storing the raw knowledge (i.e. the expert's knowledge) in a KB is insufficient though, because the transition from raw knowledge to the final KBE application is huge. This formalisation step transforms the raw knowledge to such an extent that it becomes unrecognisable to the expert once it is applied in the application. It is then not so obvious how the raw knowledge is linked to the knowledge in the application. Hence, the problem still exists, where it is impossible to validate the software because it is too difficult to retrace where the knowledge came from. To solve this, MOKA introduced the Informal and Formal Model as intermediate steps, where the link between each step is maintained all the way from the raw knowledge to the final KBE application.

S-3. Integrated product and process knowledge: Generally, KBE applications have a strong focus on product knowledge, while (mostly) ignoring process knowledge. But in reality, these two are interconnected. If this is not taken into account during the development of a design framework, then the result will be a disconnected solution which

is far from optimal. MOKA recognised the importance and developed the methodology as a complete solution where product and process knowledge are integrated. However, as the next section on weaknesses explains, MOKA could improve on modelling process knowledge.

S-4. Tools and methods: A new methodology is able to accelerate adoption if it provides tools and methods for its users. MOKA focused mainly on the methods and does not provide the tools to apply these methods.
For the *Capture* phase, MOKA explains how the knowledge engineer can prepare for capturing knowledge. It gives a brief introduction into the various methods that are effective for capturing knowledge from different sources, such as (paper) documents, human experts, and computer files. MOKA only provides the basics though, but does provide references to better sources dedicated to knowledge elicitation.
The ICARE-forms are MOKA's solution to structuring the knowledge. The set of forms enable the knowledge engineer to organise engineering knowledge effectively. Additional diagrams help creating and maintaining the meaningful relationships between the ICARE-forms.
Finally, MOKA assists in the formalisation process by providing formal knowledge models and extensive guidelines for the process. Various generic templates are included for recording the breakdown of the product into different views: *behaviour*, *function*, *representation*, *structure*, and *technology* views. This is an important step as it will facilitate KBE application development. For this purpose, MOKA also developed the MOKA Modelling Language (MML), an extension of UML for modelling formal product and process knowledge (Brimble & Sellini, 2000). Templates for breaking down the process are missing though, as becomes clear in Section 3.3.

S-5. MDSE approach: The knowledge models and (digital) KB together are perfect ingredients for Model Driven Software Engineering (MDSE). MOKA promotes the use of code generators in the *Packaging* phase, where formal knowledge is translated into an application. This reduces application development time and simultaneously the number of human errors. As a result, programmers spend less time on coding tasks and can focus on modelling instead.

S-6. Neutrality and standards: One of the main technological strengths of MOKA is its appeal for a *neutral* knowledge representation. Neutrality is a condition for platform-independence, which guarantees preservation of knowledge when an organisation decides to switch to another KBE platform. For MOKA, neutrality is a necessity as MOKA aspires to become an industry standard. For the same reasons, MOKA supports technology *standards* as much as possible. For example, the industry standards STEP (Standard for the Exchange of Product model data) and XML (Extensible Markup Language) are recommended for storing geometric data and formal knowledge models respectively.

3.3 MOKA's Weaknesses

Although MOKA is a great methodology for KBE development, there are areas that can be improved. Multiple sources reviewed MOKA and concluded that MOKA is not flawless (Curran et al., 2010; Verhagen et al., 2011; van Dijk, 2013; Skarka, 2007) Combining their findings with the author's own analysis of MOKA have led to these weaknesses.

W-1. Bias towards product rather than process knowledge: The most prominent shortcoming in relation to this research is that MOKA is more product-oriented than process-oriented. MOKA attributes this to the great diversity in design processes. Design processes are complex for many reasons and may differ per industry, company, and may also depend on how KBE is implemented. Therefore, MOKA labelled the design process model as *"the biggest challenge"*, as no simple, straightforward model would fit. This is mainly a limitation of MOKA's determination to remain neutral. It may indeed be too challenging to define a generic process model that would fit all processes. However, it is viable to define more domain-specific process models as an extension of the methodology. Moreover, whereas the methodology discusses the transition from product knowledge to KBE applications, it fails to present succeeding steps for the process model. It remains unclear how process knowledge can be implemented in WFM solutions and support KBE applications.
Solution: MOKA's process model becomes more effective when the scope is narrowed down to a certain domain, e.g. simulation workflows and design optimisation in this case. This clarifies ambiguities about what type of processes are treated, and thus the purpose of capturing and formalising process knowledge.

W-2. Shortage of use cases: MOKA fails to provide use cases that demonstrate how the methodology is used to eventually build KBE or SWFM applications. Full case studies that demonstrate how the methodology supports KBE development would certainly make MOKA more accessible to newcomers. Currently, MOKA shows segments of various engineering projects that explain the various models and forms. However, these projects are not related in any sense and without a coherent example it is not clear how a user would go through the entire process. The methodology misses step-by-step instructions supplied with examples in the formalisation and implementation phases.
Solution: Showing how it is used in practice contributes to the success of a methodology, especially in a field as complex as knowledge management. The industry is only interested in the benefits of applying the methodology. If this does not become clear immediately, companies will hesitate to apply the methodology throughout their organisation.

W-3. Lack of process templates: MOKA lacks domain-specific models or templates for building knowledge systems immediately without prior research into developing these models. Currently, it takes too much effort for commercially-minded companies to invest in implementing MOKA, especially for processes. MOKA remains at a higher level, describing the general engineering design process, and does not even specify the types of processes it supports, whether these are in manufacturing, testing, or simulations.
Solution: MOKA would benefit from having domain-specific templates, similar to how CommonKADS provides templates for various tasks, such as classification, diagnosis, design, and planning (Schreiber et al., 2000). These templates can be derived by applying the methodology to various case studies in different industries for a wide variety of applications. Of course, this will be a lengthy process that requires a considerable amount of resources. Nevertheless, this thesis contributes by developing knowledge models for simulation workflows and MDO.

W-4. Limited availability of software tools: MOKA suggests standardised technologies for storing knowledge and data transfer, but does not advise on suitable software tools. Currently, hardly any tools are available for modelling MOKA's Informal and Formal Models. PCPACK seems to be an exception, with its many knowledge acquisition

and modelling tools as well as support for various methodologies including MOKA (Epistemics, 2008). As mentioned before, the industry prefers ready solutions over technologies that require a fair amount of research and development.
SOLUTION: Offering software tools and technologies will accelerate the adoption of the methodology. Otherwise, users have to individually develop their own tools, thus returning to the problem of creating ad hoc solutions.

W-5. Focus on "Capture" and "Formalize" phases: The KBE lifecycle starts with identifying opportunities and justifying whether KBE applications will improve design efficiency. MOKA acknowledges that these steps are important, but the documentation on these steps is rather thin. MOKA focuses more on "Capture" and "Formalize" phases and admits that it relies on other knowledge management methodologies for the assessment steps.
SOLUTION: A business is only interested in whether the development of a KBE application is worth investing in. But MOKA is rather limited in this respect. As Verhagen et al. (2011) have pointed out, having a quantitative assessment framework additional to qualitative criteria will improve reliability of the analysis.

3.4 Future Direction

The new methodology should continue to build on MOKA's strengths, while improving on its weaknesses. Shortly summarised, this methodology should:

1. Narrow down the domain for processes, e.g. simulation workflows and design optimisation in this thesis.
2. Demonstrate how to use the methodology in a variety of use cases.
3. Provide domain-specific templates for describing various types of processes.
4. Provide software tools for capturing, structuring, and storing knowledge.
5. Have a quantitative assessment framework for identifying opportunities.

These opportunities to improve the methodology are addressed in the next chapter.

Chapter 4

MOKA 2: The New Methodology

MOKA has been a great initiative in standardising KBE development. The amount of research papers referring to MOKA show that it has been successful. However, as more experience was gained with the methodology, several weaknesses started to surface (see Chapter 3). The main problem, in regard to this research, is that MOKA does not cover design problems, MDO, and simulation workflows. These are important topics in the future of engineering design. And as a significant amount of knowledge is involved, it is becoming increasingly important to understand how knowledge can be applied in the setup of design problems and simulation workflows.
One of the objectives of this research is to develop new methods for capturing and structuring SWFM and MDO knowledge in relation to a KBE application. The new methodology has simply been named MOKA 2, as it maintains familiarity with the original MOKA methodology.

This chapter begins with stating the objectives in Section 4.1, followed by a step-by-step guide describing the modelling process. This process consists of four steps in the Informal Model (Section 4.2.1) and one in the Formal Model (Section 4.2.2). The chapter presents new knowledge forms that extend the ICARE-forms, two notations to visualise relationship between the forms (N^2 and BPMN), and a formal process ontology.

4.1 Objectives

The intention is to improve methodological support for SWFM, starting from the foundations laid by MOKA. The objectives stated below are formulated according to MOKA's weaknesses. Meanwhile, these objectives for MOKA 2 align with the thesis' main goals stated in Section 1.6.

O-1. Extend MOKA's process models with domain-specific elements for WFM and MDO.
While leaving MOKA's structure intact (i.e. the Informal and Formal Model), the goal is

to provide a detailed procedure for capturing and storing process knowledge and demonstrate how this knowledge can be reused in SWFM applications.

O-2. Develop process models at a higher level of abstraction to reduce the complexity of WFM. One of the thesis' research goals is to investigate new WFM solutions that will reduce the complexity of WFM. This can be achieved by developing a solution that allows WFM users to define processes at a higher level of abstraction than current WFM solutions.

O-3. Provide detailed use cases that demonstrate how the methodology is applied in the development of WFM applications.
To improve understanding of the various tools, methods, and technologies, it is essential to include examples and case studies that demonstrate how the methodology is applied. This will lower the learning curve and hence acceptance of the methodology. In order to fulfil this objective, an advanced WFMS will be built for testing and validating the methodology.

O-4. Demonstrate how existing software tools can be used in the methodology and develop new software tools wherever necessary.
Filling in the "technological blanks" is essential for gaining acceptance in the industry. Therefore, MOKA 2 should provide details about the KB implementation, provide suggestions for KBE and WFM platforms, and deliver translators and code generators that can automate the transition from formal model to knowledge application. The answers that fill in the blanks are provided in the system that is built during this research.

The remaining weakness, the lack of a quantitative assessment framework, has not been addressed. Based on the available resources it has been decided not to include it within the scope of this research. Also because it is considered a business-related topic, as it involves a thorough analysis of the organisation.

Requirements
MOKA's strengths are highly valued in MOKA 2 as well. Therefore, it is important to maintain MOKA's principles and guidelines in MOKA 2. These principles are:

- Neutrality and the preference for standardised technologies
- Transparency of knowledge systems
- Central role for the KB, which enables MDSE
- Link between product and process knowledge

4.2 A Methodology for the Development of Next Generation MDO Frameworks

To ensure that processes are designed efficiently, it is important to take a top-down approach in WFM. Thus, the methodology starts at the top-level, where design problems are formulated, and continues to describe lower levels until, eventually, a simulation workflow has been modelled that can solve the problem. This process has a total of five steps (see Figure 4.1).

Actors
To understand this procedure clearly, it is helpful to first describe the actors in this process (see Figure 4.2).

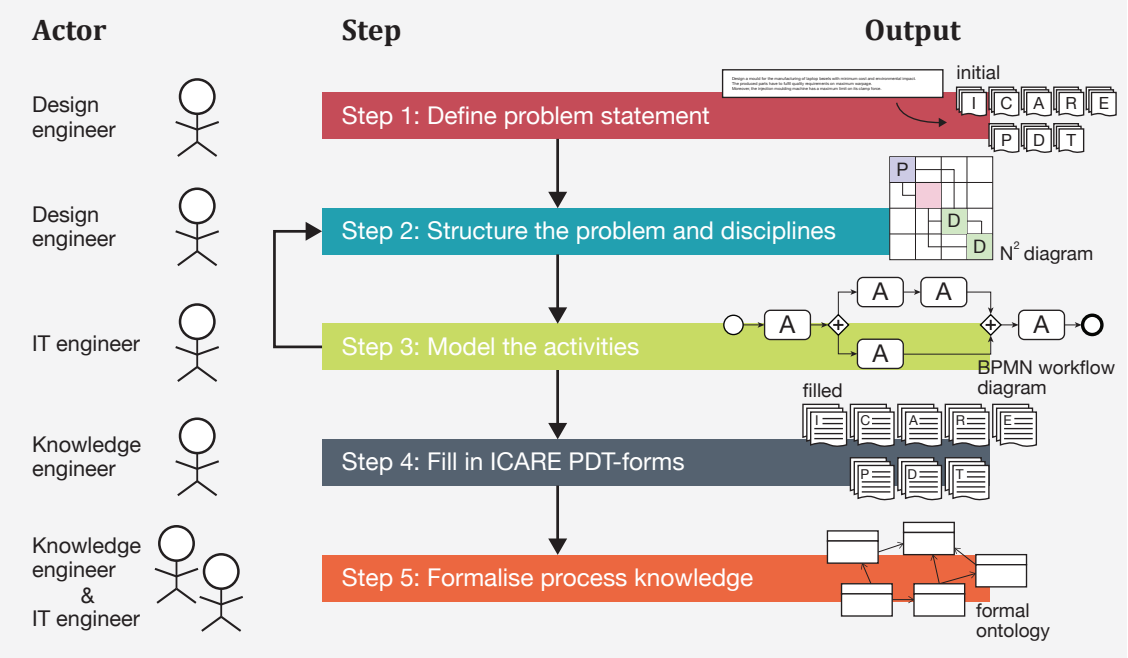


Figure 4.1: In this new methodology simulation workflows are modelled in five steps.

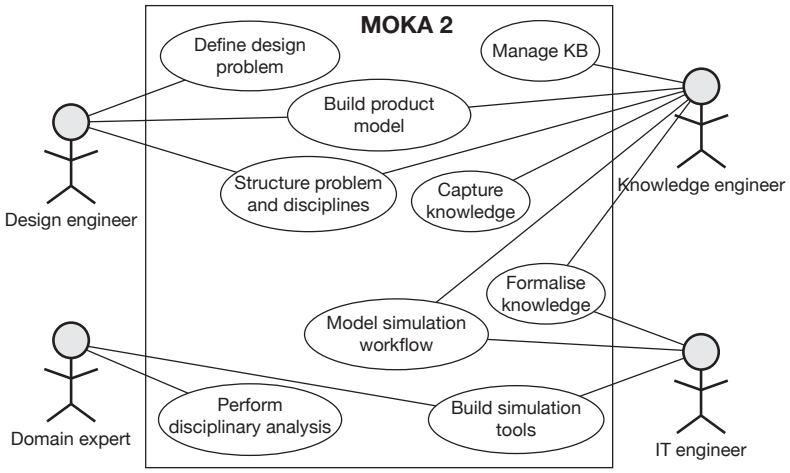


Figure 4.2: UML use case diagram showing the actors and their roles in the development of next generation design systems.

Besides knowing about products, the **design engineer** understands the problem to be solved. The design engineer determines what analyses are needed to solve the problem, but relies on **domain experts** to deliver the results. The expert knows how to perform disciplinary analyses and possesses the knowledge that goes into simulation tools. These can be in-house developed or commercially available software tools. In-house developed tools are built by the **IT engineer**, who can be a programmer or an engineer with IT knowledge. Furthermore, IT engineers know how to build KBE applications and model executable workflows. Their expertise is also useful for formalising knowledge. The **knowledge engineer** has a supporting role in this development process, to ensure

that knowledge is captured and structured properly. Ideally, support from this actor would not be necessary. However, knowledge management is a complex discipline that requires the expertise of a knowledge engineer.

4.2.1 Informal Model

The informal level is about modelling, and not about programming or execution. This section describes steps 1 to 4 of the modelling process, who are involved, what their role is, and how the methodology supports their work.

In this process, knowledge is captured in Problem-, Discipline-, Activity-, and Tool-forms. The relationships between the forms are visualised with N^2 and BPMN diagrams (see Figure 4.3). These are introduced in step 2 and 3 respectively. Furthermore, the figure shows that there is in fact a hierarchical relationship that connects all forms. It shows the (modelling) path from the problem to the simulation tools that will solve the problem. This path is as follows:

1. The main problem stands at the top.
2. This problem is decomposed into subproblems (optional).
3. Each (sub)problem is linked to disciplines.
4. Disciplines are mapped onto activities.
5. Activities are executed by tools.

When the bottom is reached, the path returns to the top to go through the hierarchy again in an iterative process. The process begins with defining the problem statement, which is explained in the next section.

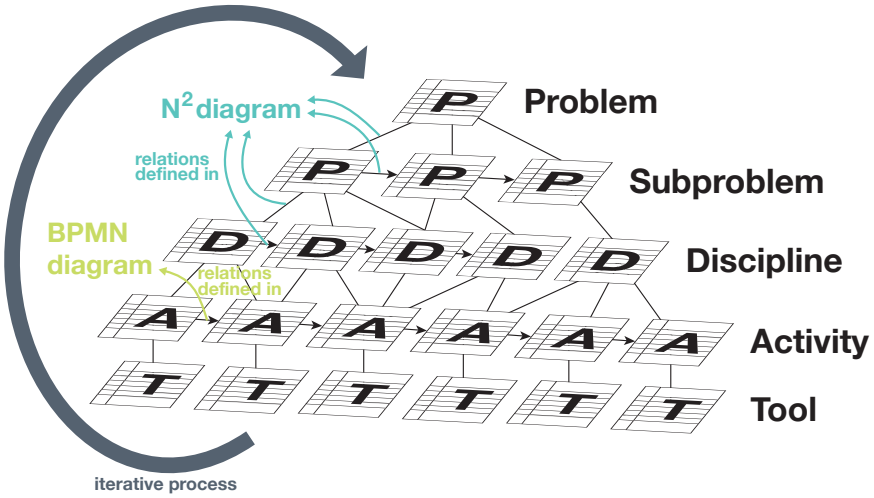


Figure 4.3: Hierarchy of the P-, D-, A-, and T-forms. The figure also shows which relationships are visualised with the N^2 and BPMN diagrams.

Step 1: Define problem statement
Every project begins with defining the design problem.

- What kind of product is designed?

- What is the objective?
- Under what conditions?

The design engineer answers these questions. For example, an answer could be:

> *"Design a **mould** for the manufacturing of laptop bezels with **minimum cost and environmental impact**. The produced parts have to fulfil quality requirements on **maximum warpage**. Moreover, the injection moulding machine has a **maximum** limit on its **clamp force**."*

This example is taken from the use case in Chapter 8. The answer to the three questions contains all the relevant information to set up a problem. When abstracting the highlighted words, it becomes clear that:

- the product is a mould (E-form).
- it is an optimisation problem (P-form).
- the objective is to minimise cost and environmental impact (R-form).
- this involves a cost analysis (D-form).
- an environmental analysis is needed (D-form).
- there are constraints to this problem on maximum warpage and maximum clamp force (C-form).
- it involves a flow analysis to determine these quantities (D-form).

At the end of each line is a reference to MOKA's ICARE-forms, which are used for capturing the knowledge of each of these statements. However, a limitation of the ICARE-forms is that it does not model problems or disciplinary analyses. Therefore, new forms are introduced to fill this gap, named *Problem* (P-form) and *Discipline* (D-form) (explained in step 4). Note that optimisation is a specialisation of problem, and that there are other methods available for solving problems, such as "feasilisation" which aims to find only a feasible solution rather than the optimum. A discipline is defined as any operation that transforms an input to an output. Hence it is not bound to either product or process.
Although the forms have been created here, filling in the details is done in step 4, after elaborating on the problem and workflow. This begins with structuring the problem and disciplines in step 2.

Step 2: Structure the problem and disciplines
The objective in this step is to structure the information from step 1 and draw relationships between the various elements of the problem. The design engineer has to do this to determine the strategy for solving the problem. This is done by defining:

- the hierarchy of problems (for multi-level problems).
- responses (results) of each problem.
- involved disciplines in each problem.
- dependencies between disciplines.

Visualising the problem is a strong solution to get a complete overview. Several notations are available for describing MDO problems (Alexandrov & Lewis, 2004; de Wit & van Keulen, 2010), however Lambe & Martins (2011) managed to design a notation that represents the entire problem in a single diagram (see Figure 4.4).

This notation is used to visualise relationships between P- and D-forms. It is based on standard N^2 diagrams, where the Problem and Disciplines are placed on the diagonal.

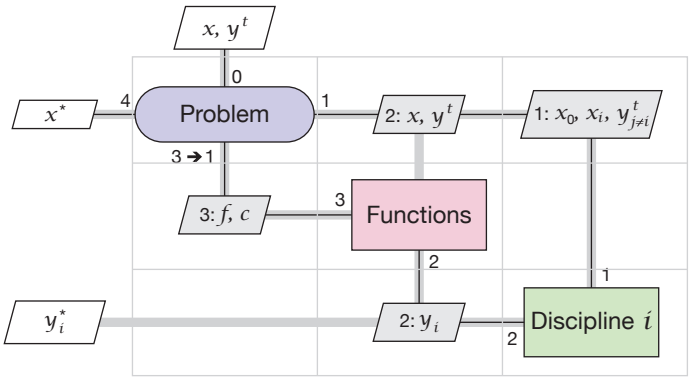


Figure 4.4: A notation for visualising MDO problems based on N^2 diagrams (modified from source: Lambe & Martins, 2011).

Inputs to a component are placed in the same column and outputs in the same row. External inputs and outputs are modelled on the edge of the N^2 diagram. The notation is fairly sophisticated and contains lots of details. Shortly stated:

- Numbers depict the flow through the diagram going from low to high. It can show parallel flow and even looping behaviour (indicated by the arrow below problem).
- x is a vector of design variables
- y^t is a vector of coupling variable targets (inputs to a disciplinary analysis)
- y is a vector of coupling variable responses (outputs from a disciplinary analysis)
- f is the objective function
- c is a vector of constraints
- Starred variables are optimised

Readers who are interested in a detailed explanation of the notation are advised to read the paper by Lambe & Martins (2011). After defining the relationships between P- and D-forms in this N^2 notation, the methodology proceeds to step 3.

Step 3: Model the activities

Disciplines transform inputs to outputs. The only missing information is how the input is transformed into the output. The IT engineer has the knowledge about simulation tools, and is capable of modelling a sequence of activities that delivers the output. This sequence is derived as follows.

The problem statement only provides information about the responses of a problem. In other words, the outputs of disciplines are known, but not the inputs. This creates a situation where backwards reasoning is needed to model the sequence of activities and to determine the inputs of the discipline.
For instance, for a cost analysis, it is known that the output is **total cost** (see Figure 4.5). The IT engineer then selects the activity that *calculates the total cost*. This activity requires its own inputs that may be provided by other activities. By repeating these steps, the IT engineer may deduce a sequence of activities until at some point the activities require input from outside this discipline. This is when the IT engineer has finished modelling activities for cost analysis.

The sequence of activities is visualised with BPMN, which in fact captures the relationships between A-forms. Figure 4.6 shows a mockup of a web interface for modelling

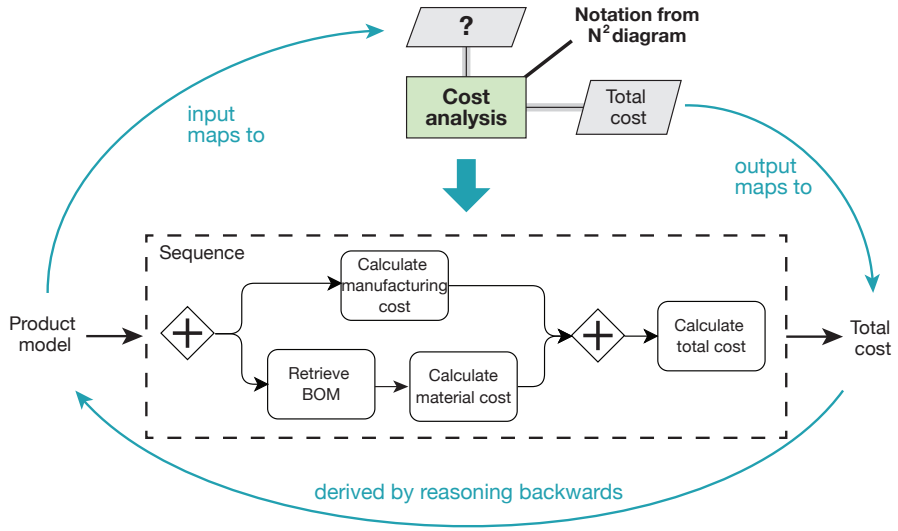


Figure 4.5: Example of a sequence of activities that performs a disciplinary analysis. Activities are modelled in BPMN.

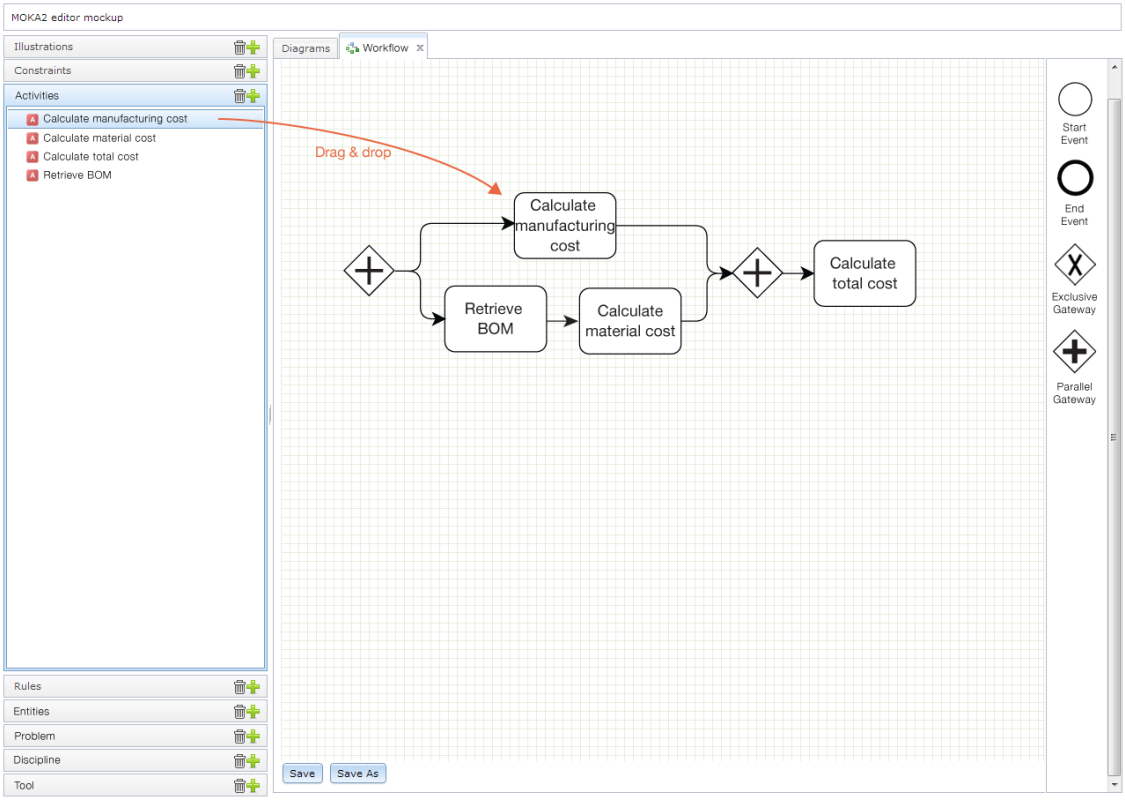


Figure 4.6: User interface mockup of the diagram editor. Activity-forms listed in the left pane are dragged onto the workspace to create a workflow.

workflow diagrams. The A-forms listed on the left are dragged onto the workspace to create this sequence of activities. More mockups are included in Appendix C.

Iterating over steps 2 and 3

In practice, the N^2 diagram can only be completed after iterating several times over steps 2 and 3. For instance, the newly acquired input from cost analysis needs to be updated in the N^2 diagram (step 2). Then, there are two scenarios:

1. This input is provided by the user as external input.
2. The input is an output of another discipline.

When it is an external input, the design engineer only has to place the input at the edge of the diagram. In the second scenario, the new input changes the definition of the other discipline. Or when this discipline did not exist, it introduces a new discipline to the diagram. In either case, the process returns to step 3, where the IT engineer has to model new activities for changed disciplines and new disciplines. This continues until the N^2 diagram is completed. How this works is demonstrated in the two use cases in Chapters 7 and 8.

Constructing the workflow

When activities are modelled for every discipline, there will be multiple sequences that are still unconnected. The links between sequences are determined by the dependencies between disciplines, laid out in the N^2 diagram. With this information, the sequences can be combined into the final simulation workflow (see Figure 4.7). And with this result, the loop over steps 2 and 3 ends.

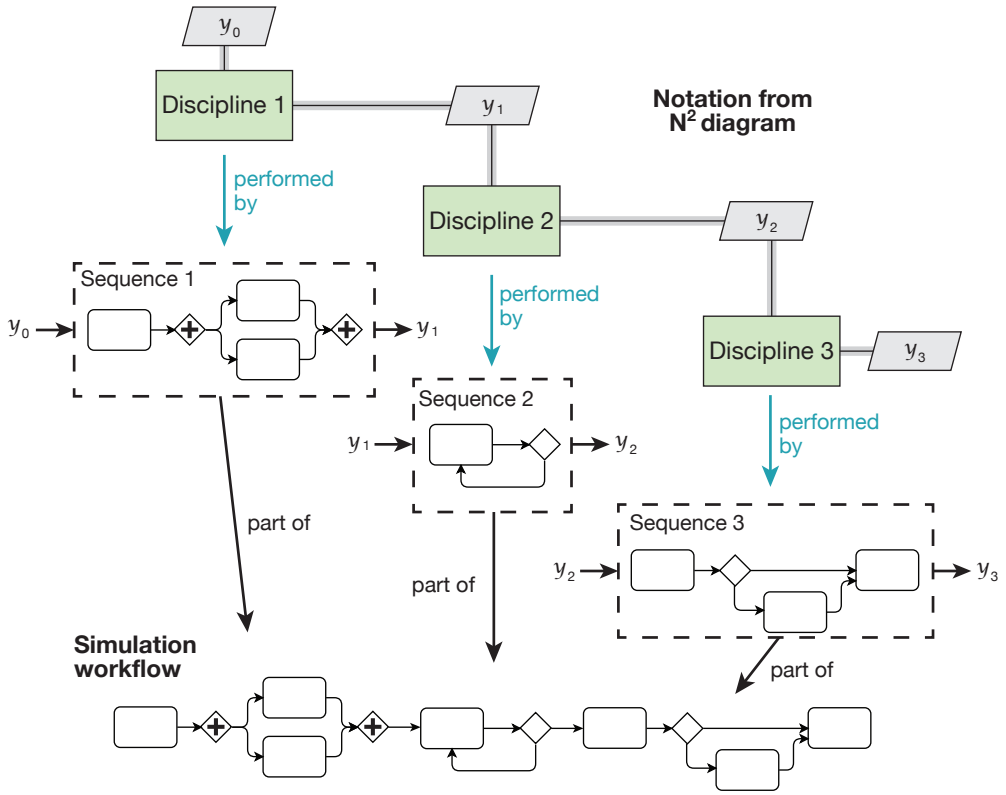


Figure 4.7: Sequences performing various disciplinary analyses can be linked into a full-size workflow based on the dependencies between disciplines.

Step 4: Fill in ICARE PDT-forms
In this final step in the Informal Model, the knowledge engineer records detailed knowledge in a set of ICARE-forms. Previously, new P- and D-forms have been introduced to complement ICARE. Additionally, the A-form has been modified to model simulation activities. Then, there is one more form needed to capture the knowledge of simulation tools: the T-form. With this final form, the set of forms is complete. In MOKA 2, these forms are named *ICARE PDT-forms*.

Figure 4.8 shows the content of the P-form. It captures problem-specific knowledge, such as the algorithm for solving the problem, the objective function, constraints, and design variables. The field for the objective function and constraints may contain a mathematical equation or references to R- and C-forms respectively, if a more elaborate description is desired. References to other ICARE PDT-forms are to:

- P-forms for (sub)problems
- D-forms for disciplines
- E-forms for entities (the product to be optimised)
- A-forms for activities (the workflow that solves the problem)

Templates for the new D-, T-, and A-forms can be found in Appendix C, as well as an interface mockup for filling in these forms. The new forms are also linked to the original ICARE-forms according to the diagram shown in Figure 4.9.

MOKA2 form:	Problem
Name	Name of the problem
Reference	Reference to the problem (e.g. an ID)
Objective	Short text to explain the general objectives
Description	Description of the problem explaining the details. This is for understanding what it does.
Problems involved	Reference to MOKA Problem forms that describe (sub)problems
Disciplines involved	Reference to MOKA Discipline forms that are involved in the problem
Entities involved	Reference to MOKA Entity forms that are involved in the problem
Activities involved	Reference to MOKA Activity form that describes the workflow solving the problem
Algorithm	Specify which algorithm is used
Description	Description of the algorithm. Explain why this method is chosen for this problem and also why these particular values are used.
Setting	Name of the setting (i.e. Max iterations, convergence criteria, target value, etc.)
Value	Value of the setting
Objective function	Reference to MOKA Rule form that describes the objective function
Constraints	Reference to MOKA Constraint forms
Design Variables	
Name	Name of the design variable
Description	Description of what the design variable is and why it has been selected as a design variable. Also explain why these particular values are used.
Initial value	Initial value of the design variable
Lowerbound	Lowerbound of the design variable's range
Upperbound	Upperbound of the design variable's range
Stepsize	(Uniform) Stepsize between the values
Discrete values	Values of discrete design variables that do not have a uniform stepsize. Values can be non-numerical.
Management	
Author	Author of the form
Date	Date of last modification
Version number	Version number of the form
Status	In Progress/Complete/Verified (pick list)
Modification	List any modifications to the form
Information origin	References to the associated raw knowledge

Figure 4.8: The new Problem form (P-form) captures problem-specific knowledge.

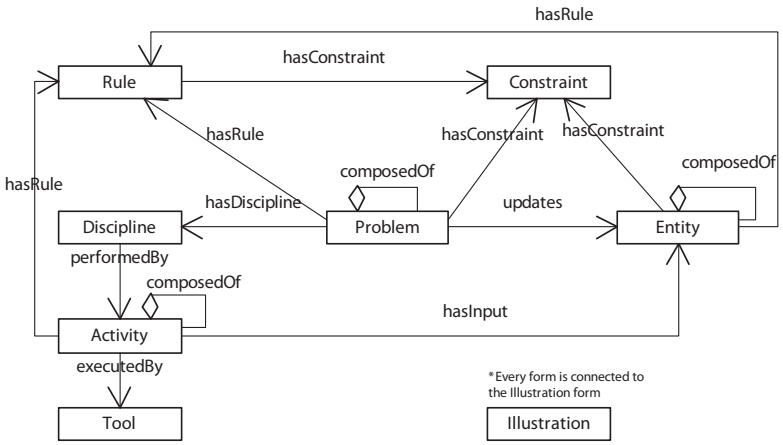


Figure 4.9: The diagram shows relationships between the new P-, D-, and T-forms and the original ICARE forms.

Filling in the forms concludes modelling at the informal level. The next step is to map this informal knowledge onto the formal process ontology.

4.2.2 Formal Model

The Formal Model is often overlooked by companies as it is not always clear what the benefits are of formalising knowledge. As a result, applications are often developed directly from the Informal Model. This is undesired though, because without the intermediate Formal Model it is more likely that the link between knowledge in the Informal Model and the final application will be lost. Figure 4.10 shows the Formal Model as an intermediate layer between the Informal Model and simulation workflow.

In the previous steps, (informal) knowledge is captured in a variety of diagrams and forms. In the Formal Model, this knowledge is translated to a computer-understandable form before a simulation workflow is generated. There are two domains in the Formal (process) Model: *problem* and *workflow*. Knowledge for modelling the problem is provided through a P-form (algorithm, design variables, constraints, etc). The BPMN diagram, A-, and T-forms provide the required knowledge for modelling the workflow (activities, flow, inputs and outputs, etc.). The next section explains how this is modelled in a new process ontology.

Step 5: Formalise process knowledge
In this final step, the knowledge engineer maps the knowledge from the Informal Model onto the formal process ontology. The IT engineer, who possesses formal knowledge about activities and tools, may provide support in this process. The end result is a formal representation of the simulation workflow in the KB that will be used to generate the workflow. Before the ontology is shown, this section begins by explaining how process knowledge can be captured.

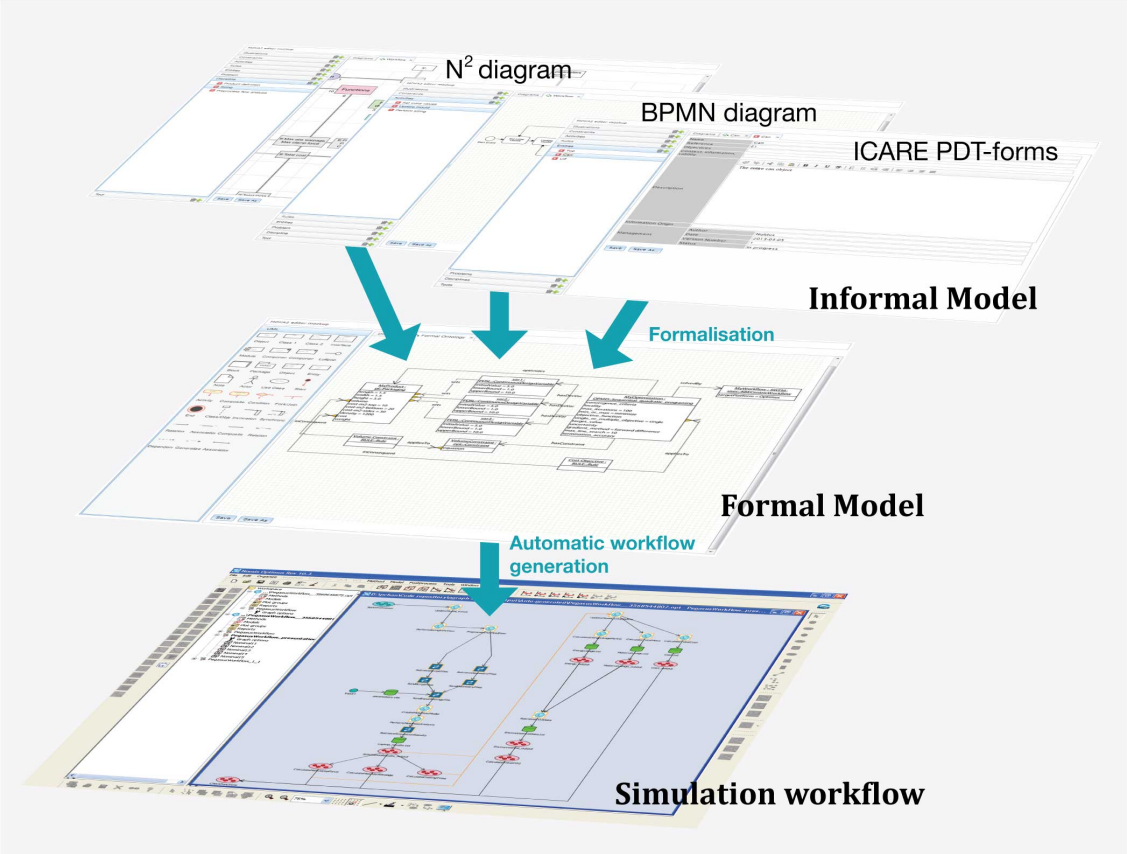


Figure 4.10: The top-down modelling process and the respective tools and methods for each layer.

Defining Processes at a Higher Abstraction Level

One of the research goals is to reduce the complexity of WFM so that it becomes more accessible to design engineers. One way to achieve this is to facilitate linking of CAx tools (Computer Aided Technologies). Although PIDO tools have been designed for this purpose, it requires a significant amount of IT knowledge to accomplish this. To make it more accessible for non-experts, a new solution is developed for modelling simulation workflows at a higher abstraction level.

The key to this solution is the **High-Level Activity (HLA)**, which consists of five elementary steps:

1. **Input:** the input to the activity.
2. **Preprocessing:** this (optional) transformation step modifies the input into a format that the CAx tool requires. Preprocessing may be executed by the PIDO tool or by an external tool.
3. **Analysis:** this is the main transformation step. It transforms input into an output through an analysis.
4. **Postprocessing:** the output of a CAx tool is either stored in a file or returned as a message (e.g. an HTTP response). If the file can be processed immediately by other tools, then no postprocessing step is necessary. Otherwise, the file needs to be

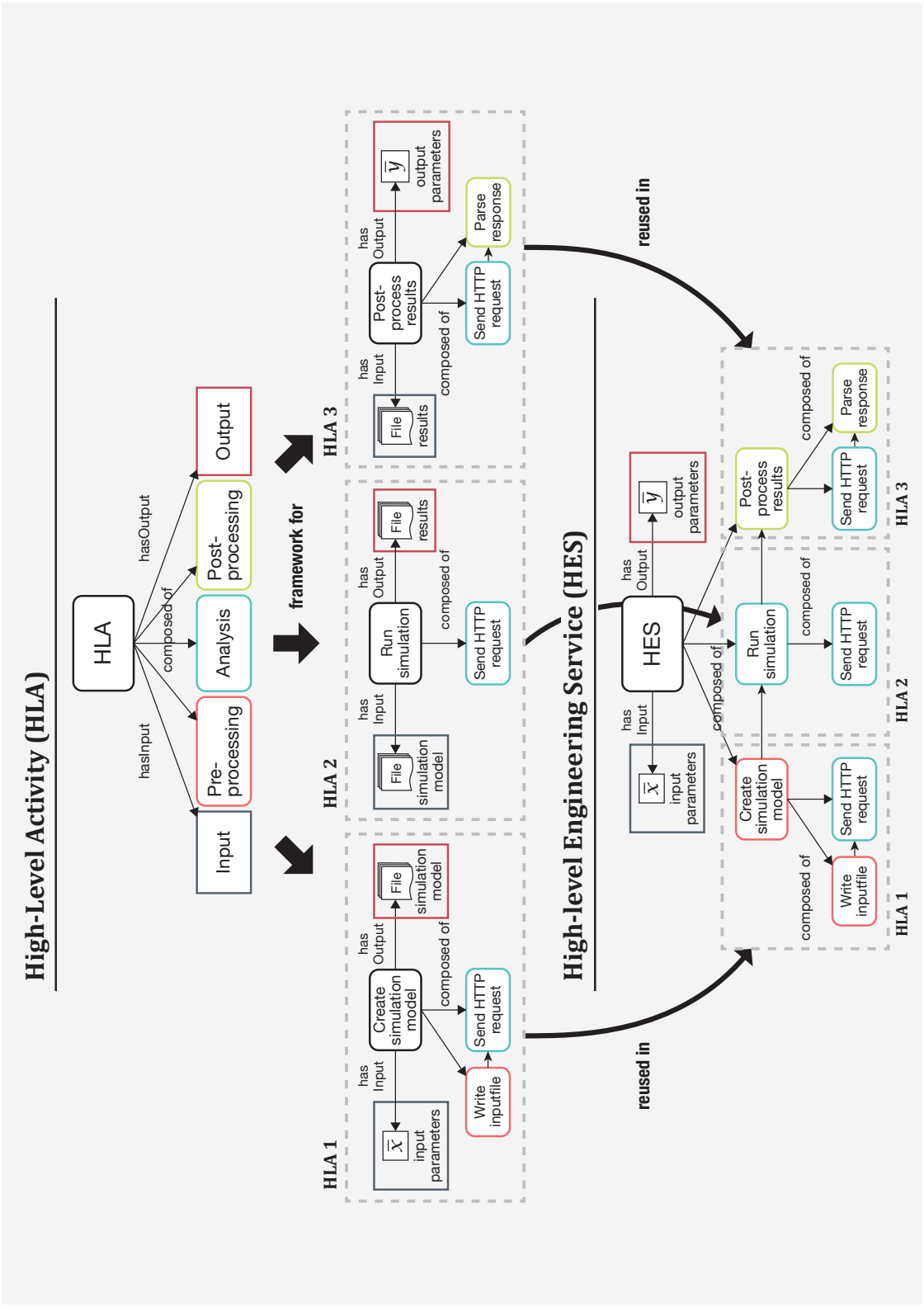


Figure 4.11: A new framework is designed for modelling activities at a higher abstraction level.

parsed to extract the values. Messages may need postprocessing as well, depending on how the PIDO tool handles incoming messages.

5. **Output:** the output of the activity.

With this definition, a new framework has been created that forms the basis for every activity. Figure 4.11 shows how different (high-level) activities are modelled based on this framework.

HLAs add value by capturing lower level knowledge that is not of interest to the user to have control over. By taking the underlying knowledge out of the user's sight (not hidden though), the user can focus on modelling what is important. For example, the HLA *"Create simulation model"* is defined as an activity that always requires a vector of input parameters, a *"Write inputfile"* and a *"Send HTTP request"* activity, and produces files. When the user decides to use the HLA *"Create simulation model"*, the system returns what the required inputs and what the available outputs are. There is no need for the user to model the remaining two activities, because that knowledge is captured in the HLA. The user only needs to decide on what input to provide and what output is desired.

It is arguable that a user is not even interested in modelling the individual HLAs (1-3) if this simulation always involves these three activities (see Figure 4.11). In that case, the same basic framework can be used to model activities at an even higher abstraction level, where HLAs are reused as transformation steps of this higher level HLA. This new level of activities has been given the name **High-level Engineering Service (HES)**.

Note that pre- and postprocessors have a different definition on different levels. For HLAs, a preprocessor is defined as a transformation step that can only write parameters and values to an inputfile. Similarly, a postprocessor can only parse files to extract values. For HESs on the other hand, a preprocessor can be a HLA that preprocesses data for a simulation tool, such as creating a mesh. Postprocessors may be another HLA that checks for problems occurred during the simulation before results are returned. Being clear on this distinction facilitates the mapping of higher level activities to the eventual simulation workflow.

Modelling Execution Details of Activities

To enable automatic workflow generation, it is necessary to capture execution details in the ontology as well. An activity requires details regarding:

- the type of inputs and outputs on a formal level.
- the tool that will execute the activity.
- the communication with the tool.
- the protocol that is used.

An inspiration for modelling these details is IDEF0 (see Figure 4.12). IDEF0 is an industry standard for functional modelling that has defined activities by its basic components.

Inputs and *outputs* are common to every activity and require little explanation. IDEF0 adds *mechanism* to model who or what system performs the activity. *Controls* act as constraints to an activity or define how the activity is executed, such as a schedule or blueprint. Although IDEF0 itself does not model the flow through activities, it provides a useful definition that separates the functions of a single activity.

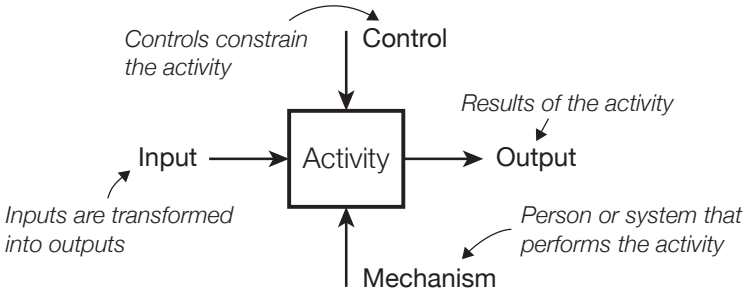


Figure 4.12: IDEF0 describes activities by its basic components.

This definition can be applied to HLAs as well, to model the communication with the tools. For that purpose, the original IDEF0 model has been adapted to take web requests and operating system commands instead (see Figure 4.13). Input and output have been further specified into two kinds: *parameters* and *files*. Then, depending on the combination of input type and control type, a suitable protocol is selected to trigger the activity (right side of Figure 4.13).

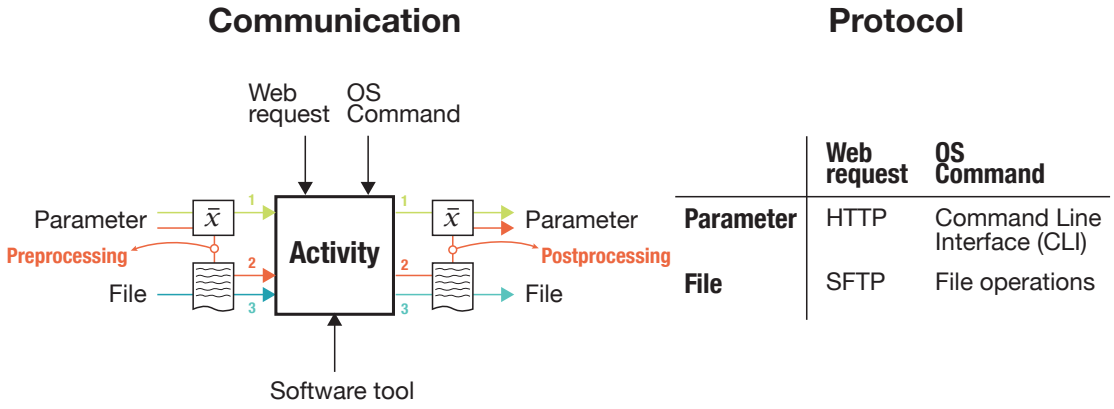


Figure 4.13: Adaptation of IDEF0 to model High-Level Activities (HLA).

From a workflow perspective, there are three ways to provide input:

1. A parameter is supplied directly to the simulation tool.
2. A parameter is written to a file by a preprocessor, and the file is supplied to the simulation tool.
3. A file is supplied directly to the simulation tool.

The reversed can be stated for the output.

Defining Inputs and Outputs

Modelling simulation workflows requires a more refined definition of inputs and outputs as well. Figure 4.14 shows in four diagrams how inputs and outputs are modelled.

1. Modelling relationships with the KBE Product Model
Some activities interact with the KBE product model to update or retrieve product at-

tributes. These activities need to know exactly how to address it in the tool. This knowledge is captured in the product ontology, therefore these activities need a link to the product model and a binding between process parameters and product attributes.

2. Attributes of inputs
Inputs can be categorised according to the table in Figure 4.14 (2). This determines how the input is processed in the workflow.
There are six possible combinations. First, an input can either be **variable** or **fixed**. When an input is variable, the value changes per iteration and the parameter is modelled as a variable in the workflow using a symbol. Fixed inputs are provided once at the start of the execution, and remain fixed throughout the run. Second, an input can be **required** or **optional**. The user must provide a value for required inputs, whereas optional inputs have default values. If it is a **process default**, the value is defined in the process domain. If it is a **tool default**, the value is stored in the tool. For the latter, the default value will not be processed in the workflow and is only included for annotation purposes.

3. Dependencies between inputs and outputs
Outputs have a dependency on inputs, meaning that an activity requires certain input to deliver a specific output. This property avoids unnecessary work. For example, if an activity has a hundred inputs, and output y depends only on one input, then based on the dependency, the user knows exactly which input to provide (instead of providing all hundred). Moreover, modelling dependencies is a way to determine the order of activities, especially when the same activity occurs more than once in a workflow. This is explained through an example in Chapter 8.

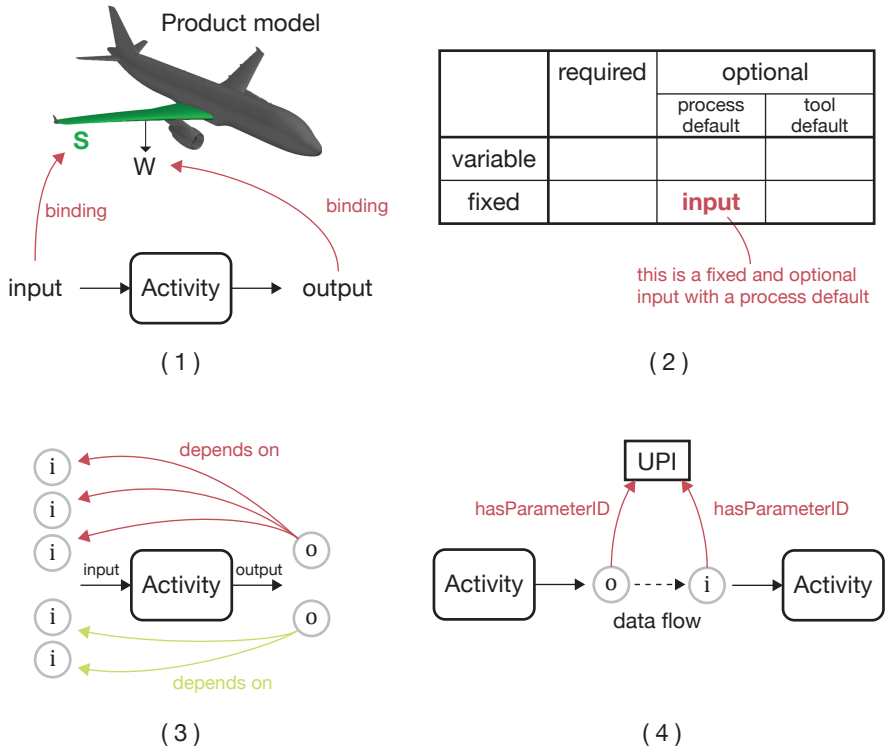


Figure 4.14: Inputs and outputs are modelled according to these four diagrams.

4. Unique Parameter Identifier
In an ontology, every entity is unique and identified by its URI. This introduces a problem in the reuse of HLAs, which may already have inputs and outputs defined in the context of the activity. It may occur that two different URIs are used for the same engineering parameter. For example, activity A has an output with URI `http://www.lr.tudelft.nl/activityA#Nr_ribs` and activity B requires an input with URI `http://www.lr.tudelft.nl/activityB#NumberOfRibs`. Modelling the data flow from activity A to B requires an additional relationship due to the mismatch of URIs. A simple solution is to link parameters to a Unique Parameter Identifier (UPI), that only exists in the domain of a workflow (or perhaps a project). Then, parameters with the same UPI are by definition the same.

Automatically Generated File Transfer Activities

Some CAx tools produce files that are provided as input to other CAx tools. In those cases, files may need to be transferred between locations. These are, in fact, simple rules that can be captured and reused to let the computer reason when file transfers are necessary. With this solution, there is no need for the human user to model file transfers explicitly.

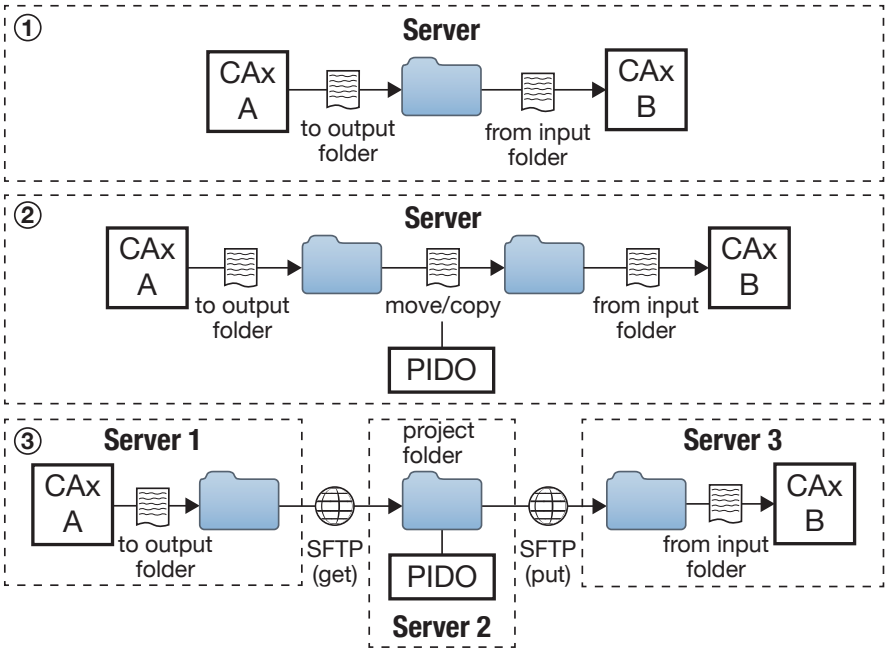


Figure 4.15: There are three methods for file transfers, depending on the location of the input and output folders of CAx tools.

To achieve this, every CAx tool must have a predefined input folder and output folder. Then, three rules can be defined for file transfers, based on the location of the input and output folders (see Figure 4.15):

1. **IF** it is the same folder, **THEN** nothing needs to be done.

2. **IF** these are different folders, but on the same server as the PIDO tool, **THEN** move or copy files to the input folder.
3. **IF** the folders are on different servers than the PIDO tool, **THEN** use SFTP (which stands for SSH File Transfer Protocol) to transfer files between servers.

The PIDO tool gives the command to transfer a file. Therefore, in the second case, it can only move or copy a file if the folders are on the same server as the PIDO tool. In the third case, where SFTP is used to transfer files, the PIDO tool always needs to transfer files to a local folder first before sending it to the input folder of CAx B. It is not possible to issue SFTP commands that transfers files between two remote servers (there is only *get* and *put*).

Modelling Activities in the ontology

Finally, activities can be modelled in the ontology according to the definitions given above. Figure 4.16 shows the main classes of the ontology.

Since workflows are modelled with BPMN, it would be reasonable to map workflows on a BPMN ontology. Because then, no translation will be necessary between notations with the risk of losing knowledge in the process. Fortunately, the standard has been made available as an OWL ontology by Natschläger (n.d.). This facilitates importing the BPMN ontology into the process ontology.

The subset of BPMN that is used to describe simulation workflows is shown in Figure 4.17. It is extended with *human task* from the HWFM domain (Human WFM). Although the focus in this research is on the SWFM domain (Simulation WFM), it shows that the ontology can be extended for modelling human workflows as well. The SWFM domain extends the BPMN ontology with *workflow* and *computer task*.

Figure 4.18 shows that the HLA is implemented in the process ontology as a BPMN subprocess. A subprocess can have flow elements that describe a lower level workflow. In this case, the lower level workflow is a set of computer tasks that map on the preprocessor, analysis, and postprocessor steps of the HLA. Eventually, these tasks are translated to PIDO activities.
The figure also shows the inputFolder and outputFolder attributes that will be used for file transfer reasoning. This will be demonstrated in the use case in Chapter 8.

The modified IDEF0 model (see Figure 4.13) has been mapped onto the ontology as shown by Figure 4.19. It is inspired by OWL-S, an ontology for describing the interaction with semantic web services (OWL-S Coalition, n.d.). In this relatively straightforward ontology, activities (Process in OWL-S) have inputs and outputs, and are performed by participants. These classes have been further specified to describe concepts in the WFM domain. Participants can be a human or computer, where a computer can be a desktop computer or server. When a task is performed by a desktop computer, it will have an attribute for `osCommand`. In the other case, when a server performs the task, the attribute will be `webRequest`. Inputs and outputs are modelled according to the definition given in Figure 4.14. All these additions are necessary for generating simulation workflows.

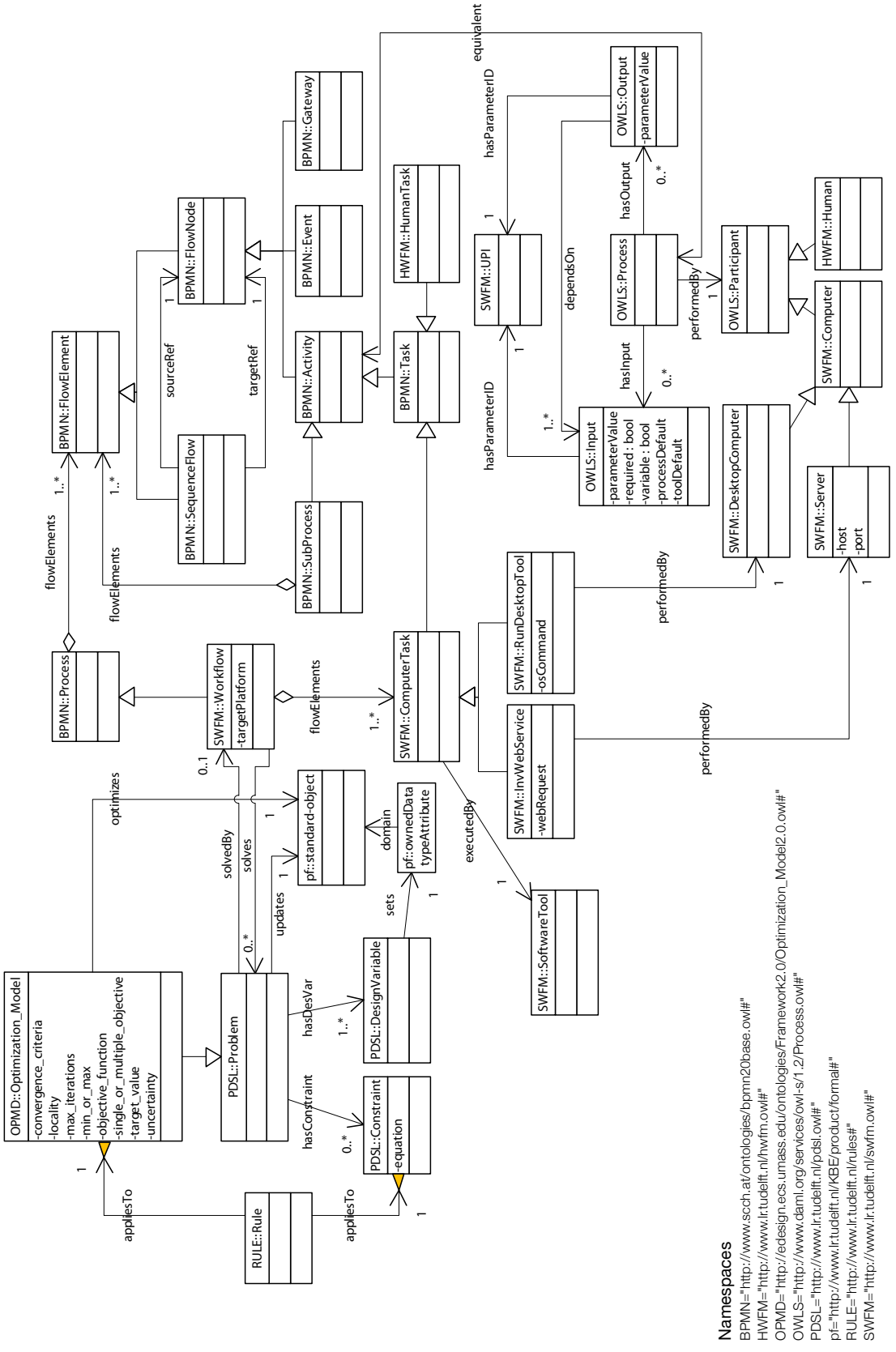


Figure 4.16: Overview of the main classes in the process ontology.

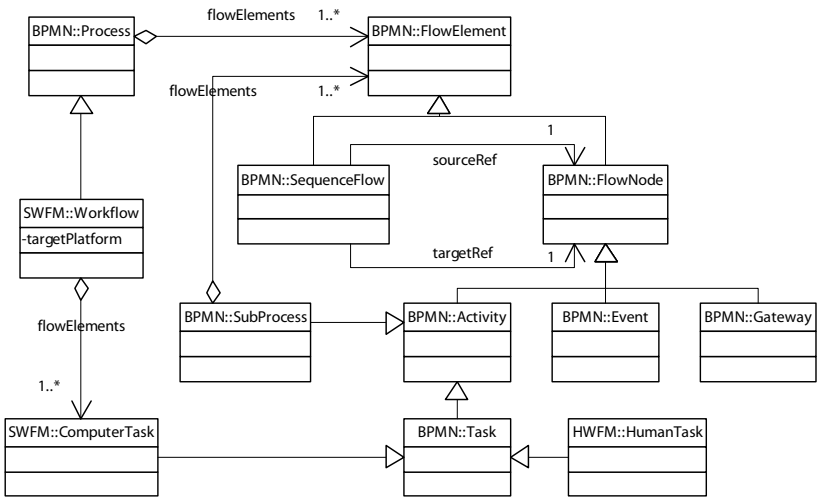


Figure 4.17: A subset of BPMN is included in the process ontology.

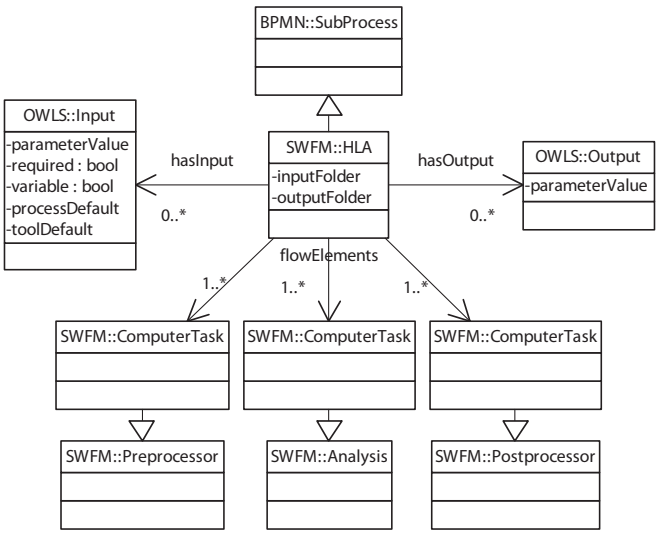


Figure 4.18: Implementation of the HLA in the process ontology.

Describing problems in the Process Ontology

The classes in Figure 4.20 are derived from common concepts in the problem domain. This domain, called Problem Domain-Specific Language (PDSL), describes the general problem class, design variables and constraints. It is extended with the Optimization Model ontology (OPMD) created by the Center for eDesign (n.d.). The OPMD ontology consists of an extensive hierarchy of optimisation algorithms and various optimisation settings, including the objective function.
The remaining classes in the figure, `RULE::Rule` and `pf:standard-object` (i.e. the product), come from ontologies developed by collaborators in this research. The product ontology is covered in detail by van Dijk (2013), whereas the rule ontology is explained by Reijnders (2012). The `pf:standard-object` is the product model to be optimised or

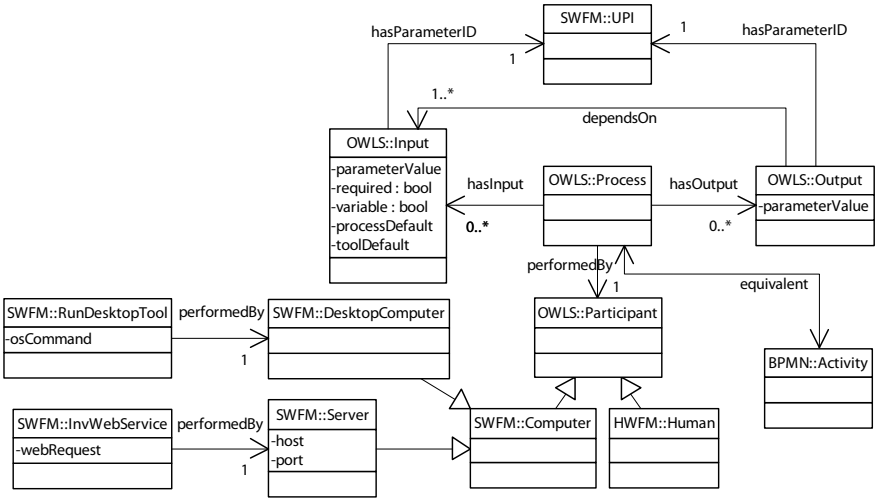


Figure 4.19: Execution details in the process ontology.

updated. Here, `RULE::Rule` contains the mathematical expression of the objective function and constraints, written in MathML. How this is modelled is explained in Chapter 7 through an example. Both ontologies are included in Appendix C.

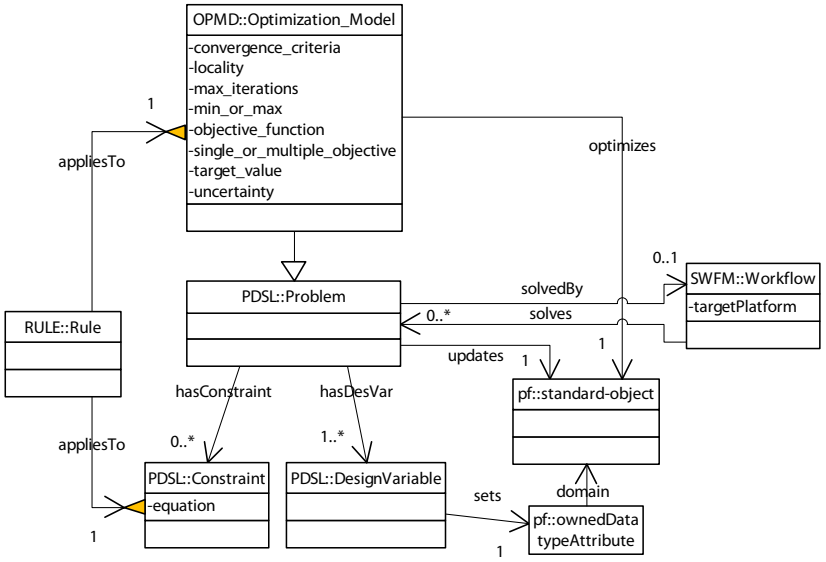


Figure 4.20: Problems in the process ontology are described according to this class diagram.

This chapter described the methodology for modelling process knowledge on an informal and formal level. The next chapter discusses the technical implementation of the methodology in an integration framework. Eventually, the methodology and the integration framework will be demonstrated in several use cases.

Chapter 5

Integration Framework for KBE and WFM

Many students and researchers at the faculty of Aerospace Engineering have built product models using KBE, ranging from morphing leading edges and flaps to entire wings and fuselages. In most cases, these applications have been built to perform (multi-disciplinary) design optimisation. However, there is currently no software framework that couples a KBE tool with analysis tools and can automate design optimisation. There have only been ad hoc solutions for design optimisation. It is a waste of time and effort if everyone creates its own solution. Moreover, it is very likely that these ad hoc solutions are not reusable in other projects. Therefore, the goal is to provide a design framework so that these (future) engineers do not have to spend time on developing a coupling between product and process tools, but can instead focus on optimising their design.

The chapter begins with an introduction to the integration framework and the software technologies that have been used in Section 5.1. Then, Section 5.1.2 explains that there are two ways to implement the framework: a KBE–PIDO coupling (Section 5.2) and a KB–PIDO coupling (Section 5.3). KBE application developers benefit from the first solution, as it provides new problem and workflow Domain-Specific Languages (DSL) (Section 5.2.2) for performing design optimisation from within the KBE environment. The second coupling is for engineers without much programming experience, and focuses on modelling. For each coupling, the section describes the software architecture and characteristics of the coupling, and explains how workflows are automatically generated.

5.1 The Integration Framework

There are many ways to implement a software framework for design optimisation. Traditionally, the design process is dominated by decoupled systems that have not been optimised for interoperability. The first is the CAD-oriented framework, illustrated on

the far left of Figure 5.1, which does not perform well on available optimisers, multi-disciplinary analysis, and parametrisation. Flager & Haymaker (2007) have discovered that it takes roughly one month to generate and analyse one design option in the CAD-oriented design process. Design optimisation would be impossible, as it would take years before an optimum is reached. The second solution is PIDO-oriented. PIDO software are designed for integrating CAE tools and performing optimisation. However, these lack essential geometric functionality. An obvious solution is to combine CAD and PIDO, as demonstrated by the third configuration. PIDO software often provide interfaces to CAD tools so that the geometry can be altered during the optimisation process. Both platforms complement each other and most weaknesses are covered. However, optimising more complex products requires parametrisation that goes beyond geometric operations. CAD is limited to geometry-oriented modifications, whereas KBE offers more sophisticated methods, such as topology changes. Moreover, creating parametric models with KBE technologies allows engineers to automatically generate different views of the product model for multi-disciplinary analyses.

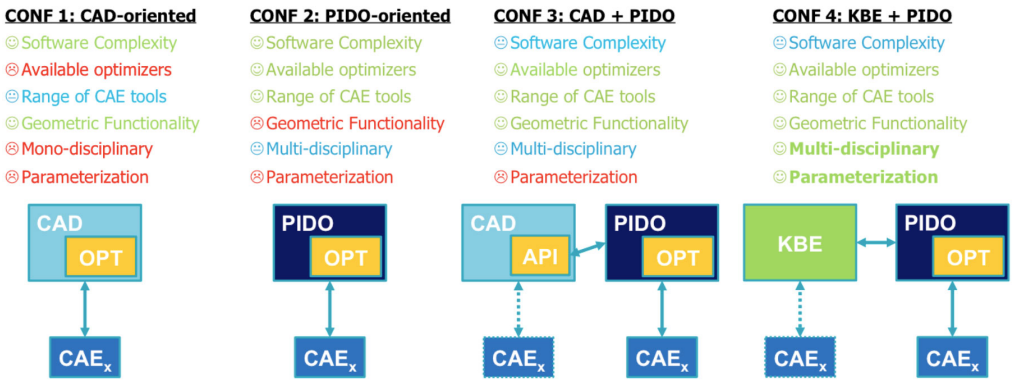


Figure 5.1: Characteristics of various frameworks for MDO (source: van Dijk, 2012).

The IDEA is built on the KBE–PIDO configuration, but extends it with a Knowledge Base (KB) for storing knowledge explicitly and independent of the platform. Moreover, modelling knowledge in a KB shifts the focus away from programming. The end result is the integration framework as shown in Figure 5.2.

The highly automated framework integrates the KB with product tools (KBE) and process tools (PIDO). It is the realisation of MOKA 2 in an actual design system, where advanced computing technologies reduce the amount of manual tasks, and hence development time and human errors. The framework uses code generation to automatically build KBE and PIDO applications. Round-tripping the results keeps the KB updated with the latest optimisation results and product configurations. Additionally, it is possible to round-trip application code. This is useful for quickly adding knowledge to the KB from previous projects (legacy applications). Meanwhile, it allows experienced users to continue working in the KBE or PIDO platforms, since their work can be committed to the KB. Finally, product and process tools are closely coupled in the integration framework. Yet, flexibility is maintained as the tools are deployed as web services and are accessible over HTTP.

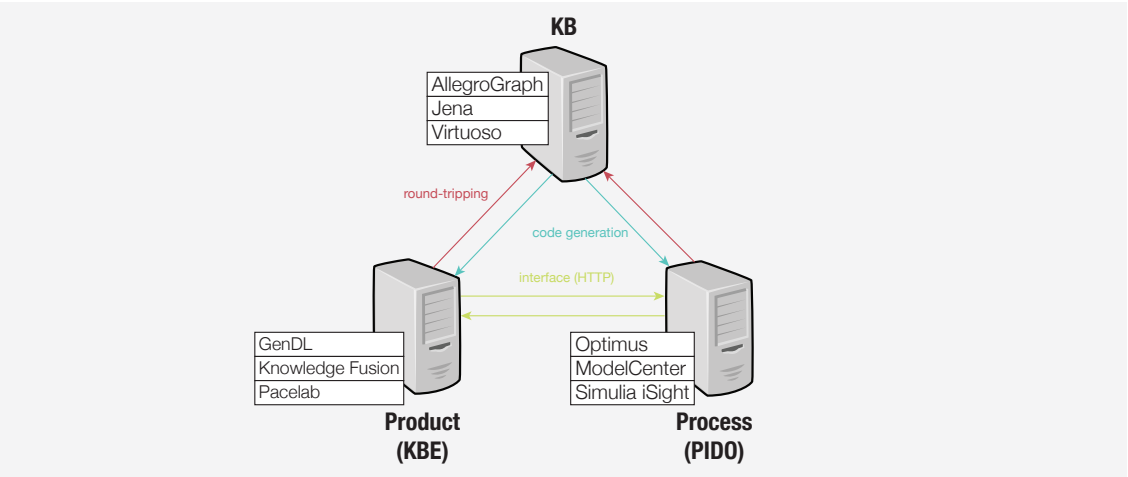


Figure 5.2: A simplified representation of the integration framework. Examples of available software tools are shown next to each component.

5.1.1 Software Technologies of the Framework

Figure 5.2 presents a list of software technologies for each component. The integration framework itself is independent of software technologies, and each of the technologies in the list could be replaced by its alternatives. This section discusses briefly the software technologies used in this research: *AllegroGraph*, *GenDL*, and *Optimus*.

AllegroGraph is a commercial triple store designed according to the RDF standard (Franz Inc., 2012). It is developed by Franz Inc. , who also offers Allegro Common Lisp (Allegro CL), and has proven to have great performance (Franz Inc., 2011). AllegroGraph follows the W3C standards by offering SPARQL for querying, as well as Prolog and its own query API. Furthermore, it contains a built-in reasoner that applies RDFS++ reasoning, which supports all of RDFS and parts of OWL. AllegroGraph itself runs on a server and allows users to access the triple store through clients, which are offered in a range of programming languages, such as CL, Java, Python, Ruby, Perl, and C#. AllegroGraph is the triple store used in this research, also because GenDL is implemented in Allegro CL which facilitates the coupling to AllegroGraph using a Lisp client.

GenDL, or the **General-purpose Declarative Language**, is an open-source programming language designed by Genworks International (n.d.). It is a superset of ANSI Common Lisp and hence offers the features and robustness of the CL language. Furthermore, it is an object-oriented and declarative language, meaning that it is able to describe objects, their properties, and the hierarchy of objects in an "engineer-friendly" way. GenDL is capable of performing geometry manipulation and visualisation through an integrated geometry kernel based on SMLib from Solid Modeling Solutions (2012). It is the KBE platform that is used throughout this research.

Optimus is the PIDO solution from Noesis (Noesis, n.d.). It is powered by a rich GUI, which makes simulation process integration and automated design optimisation accessible to design engineers. The addition of various data analysis and post-processing tools turn Optimus into a complete design optimisation system. A Python API has been included since version 10.2, allowing users to interact with Optimus using the popular

Python scripting language. Optimus has been selected as the PIDO platform for this research, where its API has been used extensively in creating the coupling with GenDL and AllegroGraph.

5.1.2 Implementation of the Framework

The integration framework has been implemented in two ways. In the first implementation, product and process knowledge are stored explicitly in a central KB. Central is the key here, as the product can then be linked to the process. Code generation techniques ensure that no programming skills are required for generating the product model and simulation workflow. This solution allows engineers without programming experience to work with KBE and PIDO software.
The second option is for experienced users of KBE and PIDO software who want to perform optimisation quickly from within their familiar environment, and do not want to spend much time on modelling in the KB. These users may be programmers or design engineers with programming skills who can get things done more quickly in the KBE or PIDO software. Round-tripping techniques can be applied to commit their work to the KB.

Thus, there are two couplings. The remainder of this chapter discusses the differences between these two.

1. **KBE–PIDO coupling:** direct coupling between product and process tools without a connection to the KB (for experienced users).
2. **KB–PIDO coupling:** workflows are generated from process knowledge in the KB.

5.2 KBE–PIDO coupling

The KBE–PIDO coupling focuses on the link between product and process tools (see Figure 5.3). With this configuration there are again two ways to implement the coupling.

1. Optimisation within a **PIDO environment**
2. Optimisation within a **KBE environment**

5.2.1 Optimisation within a PIDO environment

The PIDO-oriented solution is a more common approach, where a KBE application is built first and the PIDO tool is used to model a workflow and perform optimisation. Current PIDO platforms already provide standard interfaces to CAD systems, but not to KBE platforms. This is partly because KBE is less established in the engineering industry. If an integrated solution exists for design optimisation, it might attract PIDO users to KBE.

The work consists of developing a standard interface to GenDL, that allows users to control product parameters, model the workflow, set up the optimisation problem, and analyse results from within a single environment. However, from a research point of view this is not as interesting nor as innovative as the KBE-oriented solution. Therefore, the decision was made to focus on the latter option.

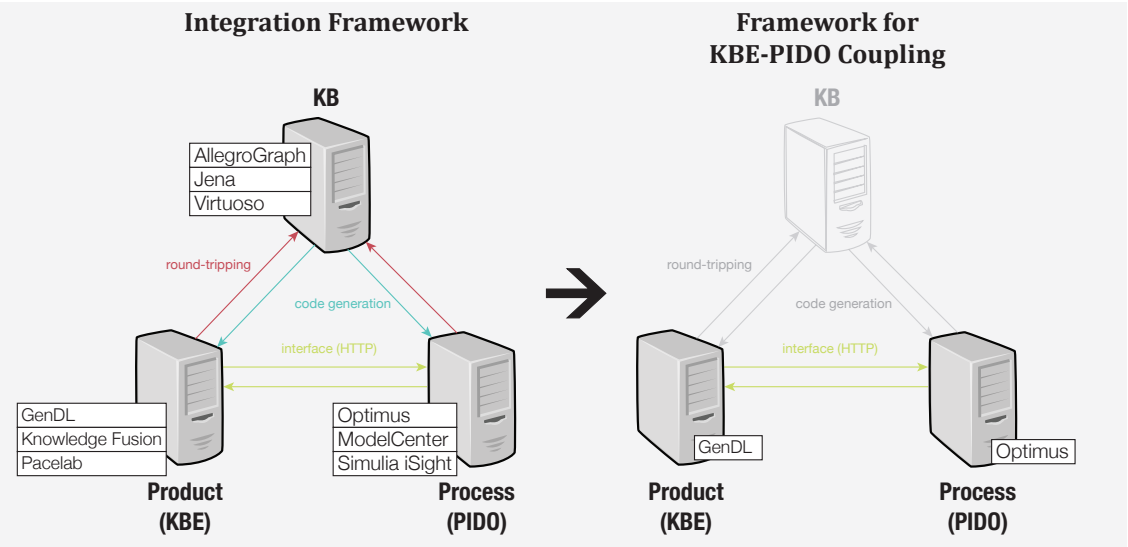


Figure 5.3: The KBE–PIDO coupling focuses on the bottom half of the integration framework.

5.2.2 Optimisation within a KBE environment

The goal in the KBE-oriented approach is to add optimisation capabilities to the KBE platform. This section describes how the coupling has been realised by showing the software architecture and two new domain-specific languages for describing problems and workflows in GenDL. The end result is that workflows, design problems, and product models are modelled in the same programming environment, while a PIDO system solves the problem in the background, out of sight of the user.

The Software Architecture

For this research a coupling has been created between GenDL (KBE tool) and Optimus (PIDO tool). In this Service Oriented Architecture (SOA) there are two servers: one for GenDL and one for Optimus (see Figure 5.4).

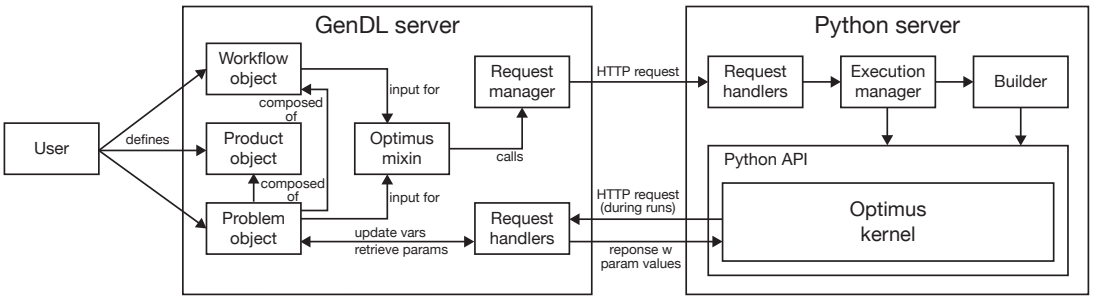


Figure 5.4: Software architecture of the GenDL-Optimus coupling.

The design process starts on the left, where the user defines a product, problem, and workflow object in GenDL. In reality, it may be a programmer who writes the code for

the product model and workflow, and a design engineer who creates the problem object (using the GUI shown in Figure 5.6). After defining these objects, the user gives the command to run the optimisation from within GenDL. From that moment, GenDL makes a series of calls over HTTP to generate the entire workflow, all automatically. These calls are sent to the Python server, which acts as a wrapper around the Optimus API. The wrapper has been implemented to simplify interaction with the Optimus API. Although it is running on a local computer now, it can easily be deployed on a remote server. With this feature, Optimus has become available as an optimisation service within the IDEA framework.

Once the workflow is generated, a final request is sent to execute the workflow. During execution, Optimus will update the product and retrieve values from the product model in every iteration. Since GenDL is running on a server as well, Optimus can send HTTP requests to interact with the product model. This solution is more flexible, as GenDL can be running on another server than Optimus. Thus, GenDL has become available as a service to other applications as well. Once Optimus has found the optimum, the product model is updated for a final time and the result is an optimised product.

Automatic Workflow Generation

Optimus mixin is the component that pulls information from the problem and workflow object and makes calls to the request manager to generate the workflow. From the problem object it needs the design variables, objective, constraints, and algorithm, and from the workflow object the activities and connections that need to be generated. The mixin component is also coupled to Graphviz (Graphviz, n.d.), a graph visualisation software for automatically drawing the layout of various diagrams. Graphviz determines the x, y-positions of the activities in Optimus. At the end, the mixin component sets up the problem in Optimus and makes the call to execute the workflow.

The **request manager** processes the calls from Optimus mixin and sends HTTP requests to the Python server. Figure 5.4 shows that the Python code consists of a *request handler*, *execution manager*, and *builder*. The **request handler** contains the server functions, and 'listens' to incoming HTTP requests and relays requests to the appropriate wrapper functions. These requests are processed by the **execution manager**. This module is connected to the Optimus API and to the builder, which creates the various workflow elements. The **builder** is in fact divided into two modules: an *item builder* and *execution configurator*. The **item builder** creates all the workflow elements, whereas the **execution configurator** is responsible for setting up the execution method. Appendix B describes the wrapper code in more detail.

This concludes the section on the architecture and workflow generation process. The above-mentioned problem and workflow objects are actually new to the GenDL language. The section below describes these new objects.

New Domain-Specific Languages

Two new Domain-Specific Languages (DSL) have been developed to complement the coupling: a *problem DSL* and *workflow DSL*. Both languages have been used in the use case in Chapter 6.

Problem DSL

Product objects are native to GenDL, and are modelled with the `define-object` statement. However, design problems are new to GenDL, and thus a new **problem object** has been introduced using a comparable `define-problem` statement (van Dijk, 2013). It follows the same structure of product objects, but redefines the object with problem-specific terms, such as algorithm, design variables, constraints, and objective (see Figure 5.5).

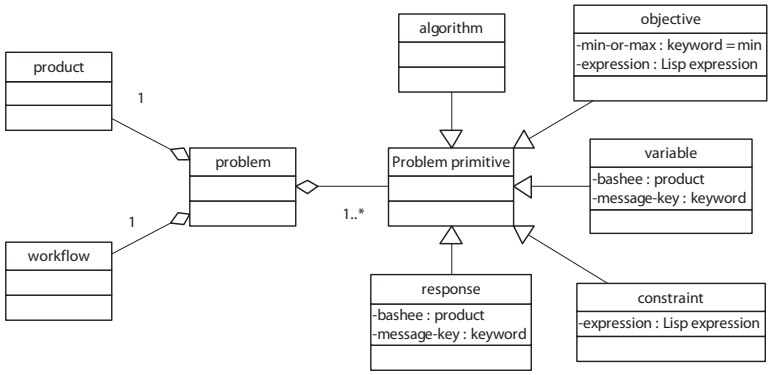


Figure 5.5: UML class diagram of the new problem DSL.

Since it is written in a human-oriented engineering language, most objects require little explanation. However, there are a few terms that should be clarified.
First, the attributes of *algorithm* depend on its type. An optimisation algorithm requires different settings than, for instance, a DOE.
Responses are the results of the design problem. Here, a response retrieves a value from the product model, so that the expressions of the objective and constraints can be evaluated. This is useful for checking the constraints or even perform optimisation within GenDL. Without the responses, the variables in the expressions would be unknown and the evaluation would raise an error.
Lastly, the two attributes *bashee* and *message-key* relate design variables and responses to the actual parameters of the product. The bashee is related to the product object, and points by default to the root object of the product. However, if a parameter belongs to a subcomponent of the product (i.e. a child of the product), then it is necessary to provide the bashee that will point the variable or response to the right object. For example, if only a single rib is varied in the design of an aircraft movable (e.g. rudder or elevator), then this rib instance is provided to the bashee attribute. The message-key attribute is used to point the variable or response to the right parameter, e.g. *"length"* of the rib. By default, this is the same name as the variable or response.

To assist the engineer in modelling problems, there is a web interface for adding and removing objects (i.e. design variables, responses, constraints, etc.) and for providing values to these objects (see Figure 5.6). It is a fairly straightforward design with input boxes and buttons for modelling these objects. Additional tabs are offered for algorithms and results. It is an alternative to writing GenDL code, which lowers the threshold for new users.

Workflow DSL

The second DSL has been introduced to describe workflows. It consists of classes de-

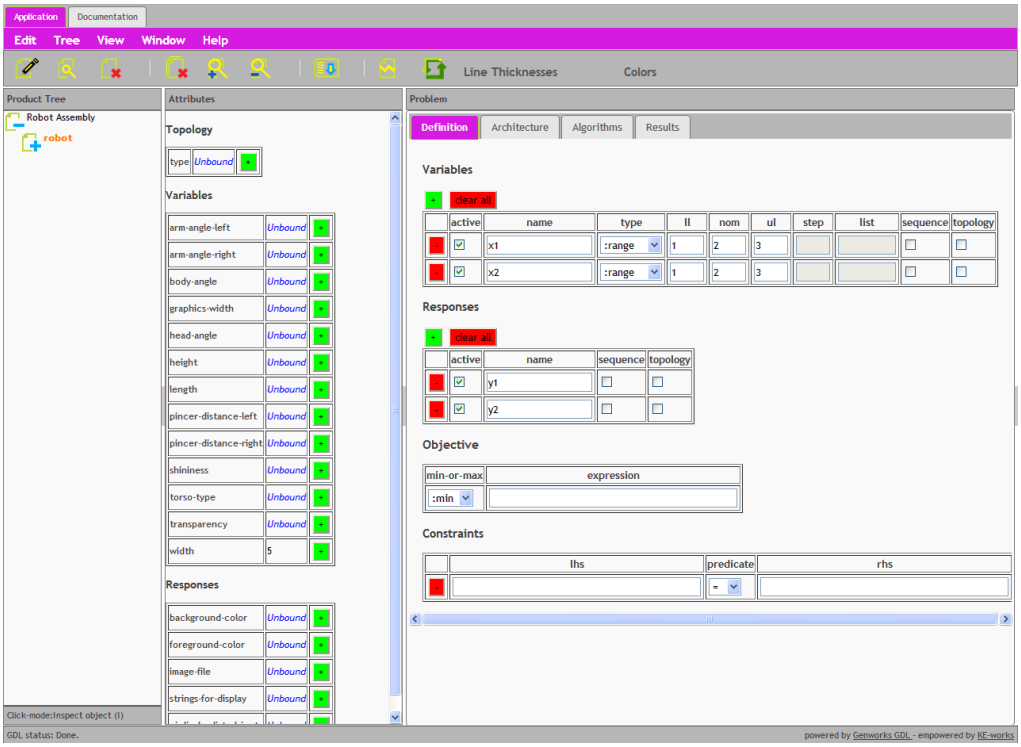


Figure 5.6: First prototype of the problem modelling web interface (source: van Dijk, 2013).

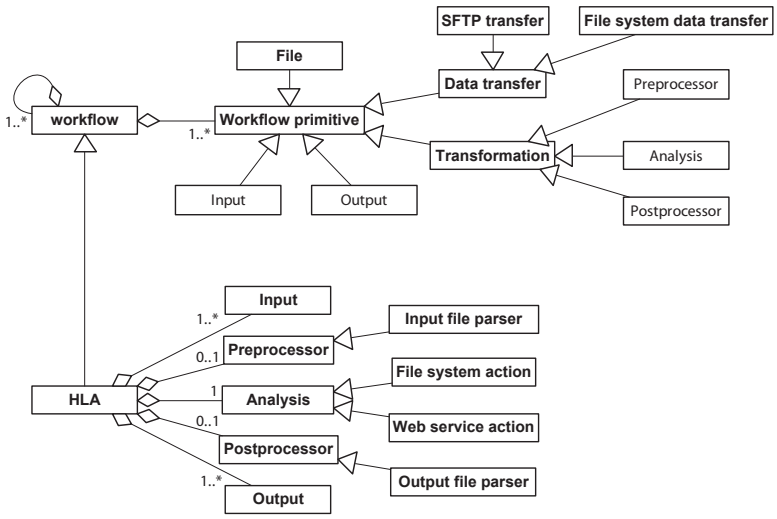


Figure 5.7: UML class diagram of the new workflow DSL.

scribing activities, the flow, and data objects that are necessary for generating simulation workflows (see Figure 5.7).

A workflow is composed of workflow primitives, which are basic elements of a workflow. Examples of primitives are file, input and output, transformation steps, and data transfer. These can be further specialised into types, e.g. file system data transfer is a type of data transfer. The main object, the HLA with its five elementary steps, is also applied here.

Every activity in the workflow can be built on this basic structure. With this DSL, the KBE user is able to model workflows in GenDL using object-oriented programming, that can be translated into a simulation workflow.

The workflow DSL is still under development, but in its current state it is sufficient to model relatively simple workflows. This is demonstrated in the use case in Chapter 6. Moreover, the chapter will also introduce a parametric high-level workflow, built with this workflow DSL, that allows design engineers to optimise a KBE product model without modelling the workflow. It is a solution for engineers without SWFM experience.

5.3 KB–PIDO Coupling

The KB–PIDO coupling is designed to be more accessible to non-programmers. The target is the design engineer. Eventually, the goal is to have design engineers model the product and process without support from others.

In this implementation, code generators take over the programmer's task of writing code. This saves development time, reduces number of human errors, and reduces the required IT expertise. KBE and PIDO applications are automatically built from the KB models. This section explains the role of reasoning in this coupling and describes the software architecture. A schematic of the framework is presented in Figure 5.8.

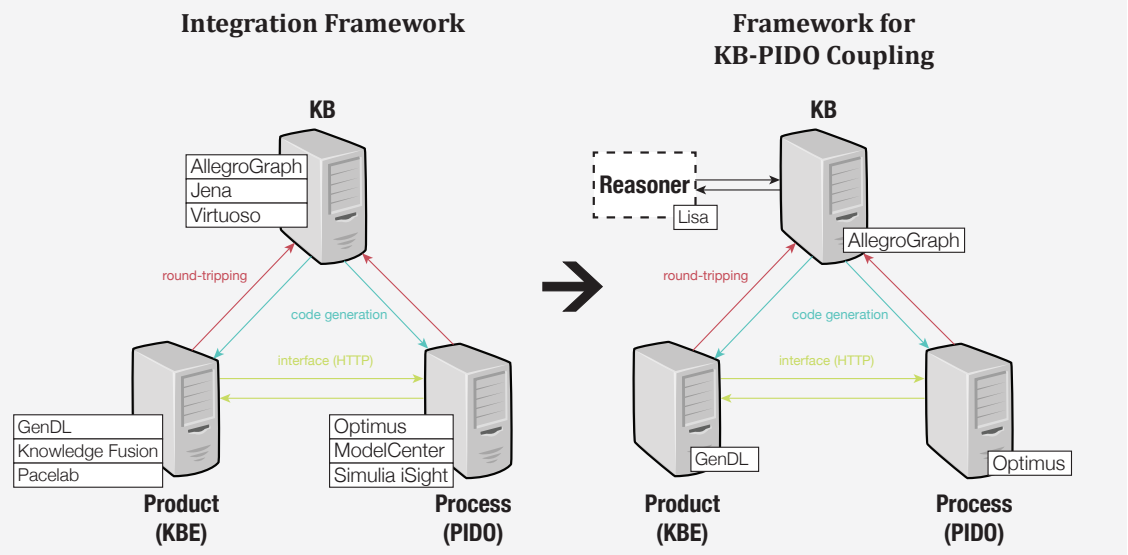


Figure 5.8: The KB–PIDO coupling uses all components of the integration framework, including a KB and reasoner.

Reasoning over Knowledge

Ultimately, the goal is not only to shift the focus from programming to modelling, but also to bring modelling to a higher abstraction level. Knowledge can be used to automatically fill in lower level details. For instance, modelling the transformation steps of a HLA

requires expertise about sending HTTP requests and parsing files, and is typically a task that a design engineer should not be troubled with. This knowledge can instead be captured in classes, properties, and restrictions in the ontology.

This level of automation is achieved by interpreting restrictions in OWL in an unconventional way. Normally in OWL, reasoning is used to derive facts that are not stated explicitly. In this research, reasoning is applied to instantiate classes.
For example, Figure 5.9 shows the class *Bike* with restrictions that it has exactly one *Engine* and two *Wheels*. Then, if there is an instance of an object with exactly one engine and two wheels, it is classified as a bike. Thus, in this example, the Yamaha R1 can be classified as a bike through reasoning.
The same restrictions are interpreted differently for the Ducati Monster. In this second example, an instance is created that is known to be a bike. However, its engine and wheels have not been instantiated. Then, through reasoning, a computer can instantiate an engine and two wheels based on the restrictions.
This is how a higher level of abstraction is achieved for HLAs. The aim is to let users instantiate the HLAs, but not the transformation steps of the HLAs. How this works is demonstrated in the use cases in Chapters 7 and 8.

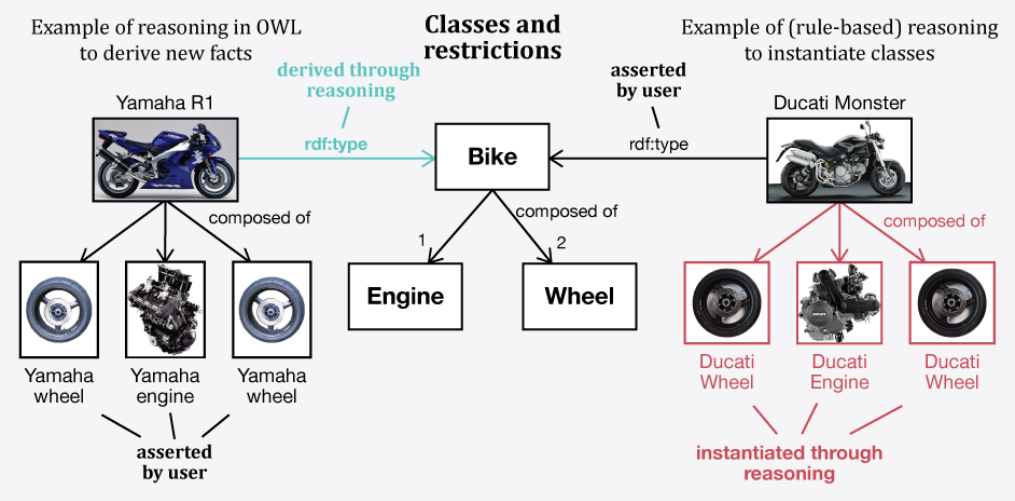


Figure 5.9: Classes and restrictions are used to instantiate classes through reasoning.

The Software Architecture

The architecture has been extended with a triple store (AllegroGraph), a Reasoning Engine (RE), and several new components in the GenDL server (see Figure 5.10). In this new configuration, the user defines the engineering rules, product model, and process model according to their respective ontologies. Reijnders (2012) has already built a web interface for modelling rules. GUIs for modelling product and process models are being developed by collaborators in this research.

The workflow generation process begins with defining the process model, after which the facts and rules are imported into the **Reasoning Engine (RE)** (see Figure 5.11). The RE developed for this framework is a modified version of Lisa (Young, 2010). Reijnders

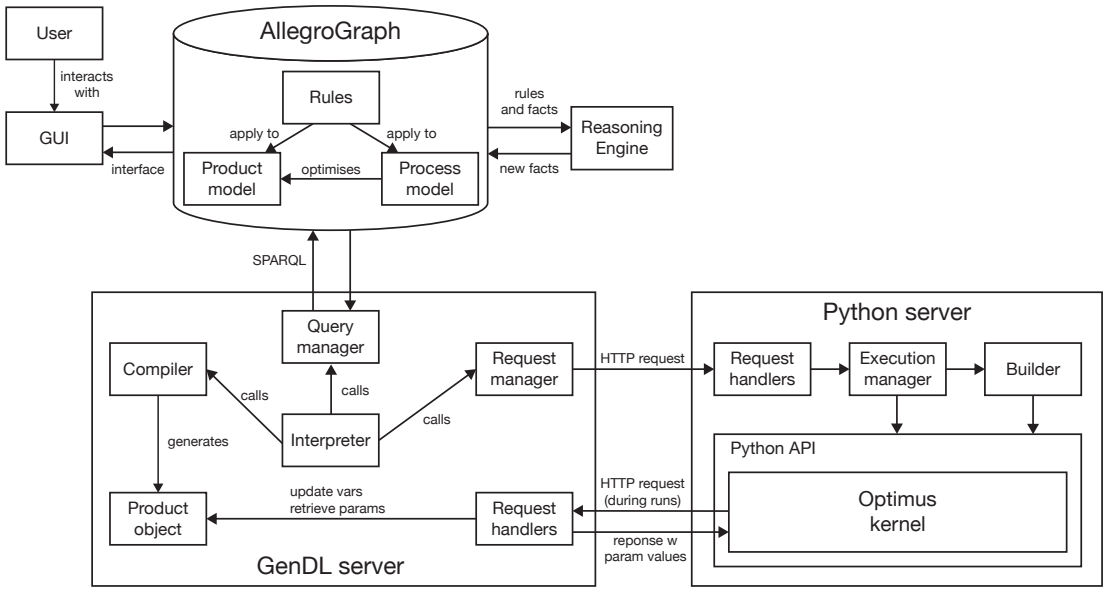


Figure 5.10: Software architecture of the integration framework.

(2012) has modified Lisa to be compatible with RIF and to have an interface with the triple store. Lisa is a forward RE implemented in Common Lisp (CL) and has therefore access to the full CL language from within the RE. It is an expert system shell, meaning that it offers the full functionality of an expert system but does not provide the knowledge. The user feeds the knowledge to the system. This knowledge ends up in the *working memory*, which normally contains the facts, or triples in this case. Then, the *rule memory* (or knowledge base in the traditional terminology) contains the rules, which are translated from RIF. The *inference engine* does the actual work, and can infer new facts or trigger

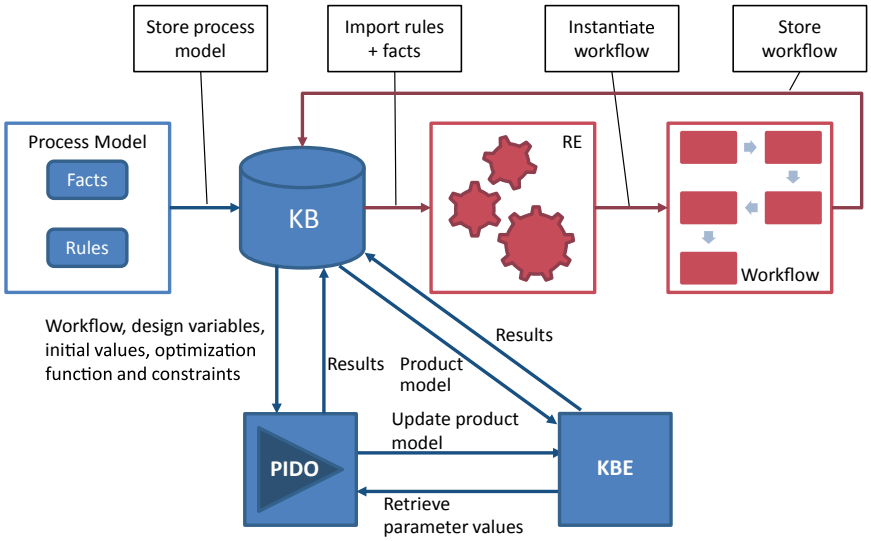


Figure 5.11: The workflow generation process (source: Reijnders, 2012).

actions (functions in CL). Chapter 7 explains how the RE is used to instantiate the workflow. The workflow is stored back into the KB as new triples. From this point, the interpreter can automatically generate and execute the simulation workflow.

Automatic Workflow Generation

The *query manager* is in control of interacting with the triple store and sends queries to retrieve relevant knowledge. The *code generator* uses this knowledge to build KBE applications of product models, which is explained in detail by van Dijk (2013). The *interpreter* on the other hand, focuses on the process knowledge to automatically generate an executable simulation workflow. Simply stated, the interpretation process consists of four steps:

1. The interpreter sorts out which activity needs to be created in Optimus, and translates properties of the activity to the right input for Optimus.
2. It then creates the connections that determine the flow through the workflow.
3. When the entire workflow is generated, it sets up the optimisation problem.
4. At the final stage, it executes the Optimus workflow (optional).

A more elaborate explanation of the process can be found in Appendix B. The remaining components, including the Python server, remained the same.

The entire methodology, including a framework, has been explained in the past two chapters. The next chapters are use cases that demonstrate the new methods and tools that have been introduced.

Chapter 6

Use Case 1: KBE–PIDO Coupling

The previous chapters have described a new methodology for engineering design, including a framework for design systems. The remainder of this thesis describes use cases that demonstrate the capabilities of the solutions, with the purpose to verify the methods and tools that have been developed.
The use case in this chapter is an implementation of the KBE–PIDO coupling. It has been designed to demonstrate optimisation within a KBE environment, and more specifically, to show how the new problem object is used in GenDL to perform optimisation. The goal is to provide a solution for design engineers to do design optimisation with minimal effort. This can be achieved by capturing process knowledge in a KBE application that describes a parametric workflow for optimising KBE product models.

The subject of the use case is the optimisation of a product packaging. The first section describes the mathematical problem and presents the analytical solution that is used later for verification (Section 6.1). It is a relatively simple problem so that the answer can be verified with the analytical solution. The main part of this chapter is the implementation, which runs through the code and shows the results (Section 6.2). The chapter is concluded with an evaluation of the use case in Section 6.3.

6.1 Problem Description

A fictional company has sent a request to their design team to design a new packaging for their product. The company has stressed that the cost of the packaging should be minimised. The cost for the packaging material is given as:

$$\begin{aligned} &€\,20/\mathrm{m}^2 \text{ for the bottom} \\ &€\,30/\mathrm{m}^2 \text{ for the sides} \\ &€\,10/\mathrm{m}^2 \text{ for the top} \end{aligned}$$

Furthermore, the design team is told that the packaging must have a volume of exactly $4\mathrm{m}^3$. With this information the design team starts working on the challenge of optimising the product packaging.

Figure 6.1: Dimensions of the packaging object.

In fact, it is a simple object that can easily be solved analytically (see Figure 6.1). First, the areas of the surfaces are calculated as:

$$A_{bottom} = xy$$
$$A_{sides} = 2xz + 2yz$$
$$A_{top} = xy$$

Putting this together with the material cost gives the objective function (*minimise cost*):

$$f(\bar{x}) = 30xy + 60(xz + yz)$$

The optimisation problem has one equality constraint for the volume:

$$h(\bar{x}) = xyz = 4$$

Solving the optimisation problem gives:

$$x = y = 2,\ z = 1$$
$$f_{min} = 360$$

Thus, the optimised packaging has a length and width of *2m* and a height of *1m*. For these dimensions the cost has been calculated at €360.

The next step is to solve the optimisation problem using the integration framework. It is a simple object, but it is a good example for demonstrating the capabilities of the framework.

6.2 Implementation

The primary actor in this process is the KBE user who writes code to build KBE applications. Building the packaging object is not the challenge, however optimising the packaging is. In this framework, the KBE user benefits from having the KBE–PIDO

coupling to perform optimisation. It is a relatively simple problem that does not involve any CAE tools other than the KBE and PIDO platforms. Yet, it is a problem that occurs often. Therefore, it is worthwhile to capture the knowledge about the workflows that solve these kind of problems. This knowledge is then used to create a high-level parametric workflow, tailored to optimise KBE product models. With this solution, the user can simply define a problem object and execute the optimisation process without defining a workflow. This section demonstrates how it is done.

6.2.1 KBE code

The KBE code for the packaging is fairly straightforward, because a packaging is nothing more than a box, which is a native object (class) in GenDL. This code is written first, before the user writes the code for the problem. Figure 6.2 shows the code for the packaging object.

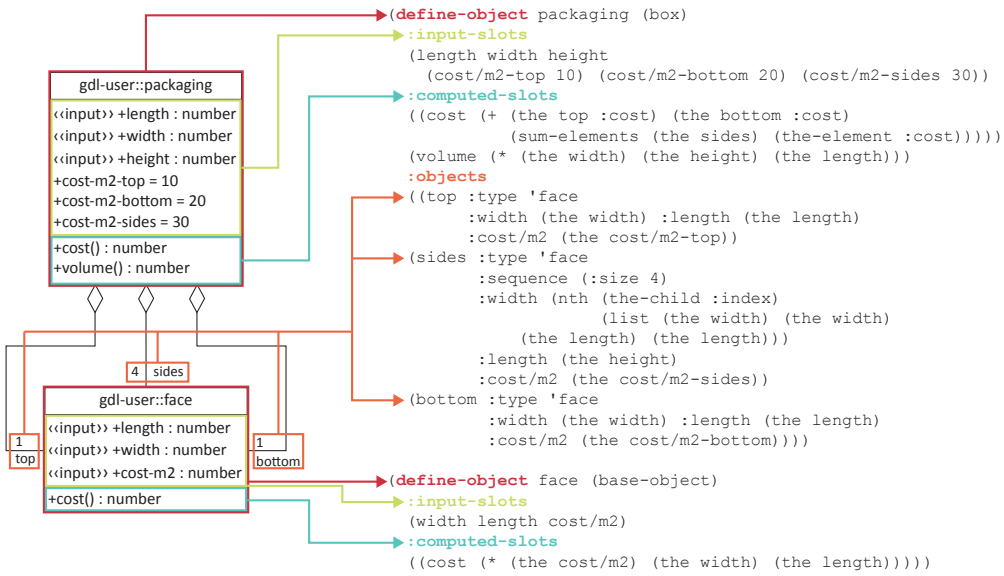


Figure 6.2: GenDL source code for the packaging.

The `define-object` statement in GenDL defines a class, which can be instantiated using the `make-object` function. The `input-slots` of the class show the inputs of the packaging object, which are the dimensional parameters and the material cost. The costs have default values as specified in the problem description. Furthermore, the second block of code, `computed-slots`, can be used for attaching additional properties to the object. In this example, the additional properties are cost and volume, both computed from a formula. The total cost of the packaging is a sum of the cost of its child objects: the top, sides, and bottom. These objects are of the type *"face"* that is defined below the packaging object. The face object is used for modelling the cost of the top, sides, and bottom. It takes three inputs, as can be seen in the code, and it calculates its own cost based on these inputs.

For this new KBE–PIDO coupling, a new problem object is introduced to the GenDL platform. It follows the same syntax as `define-object`, except that it has been tailored

to define optimisation problems. The code for the problem is shown in Figure 6.3.

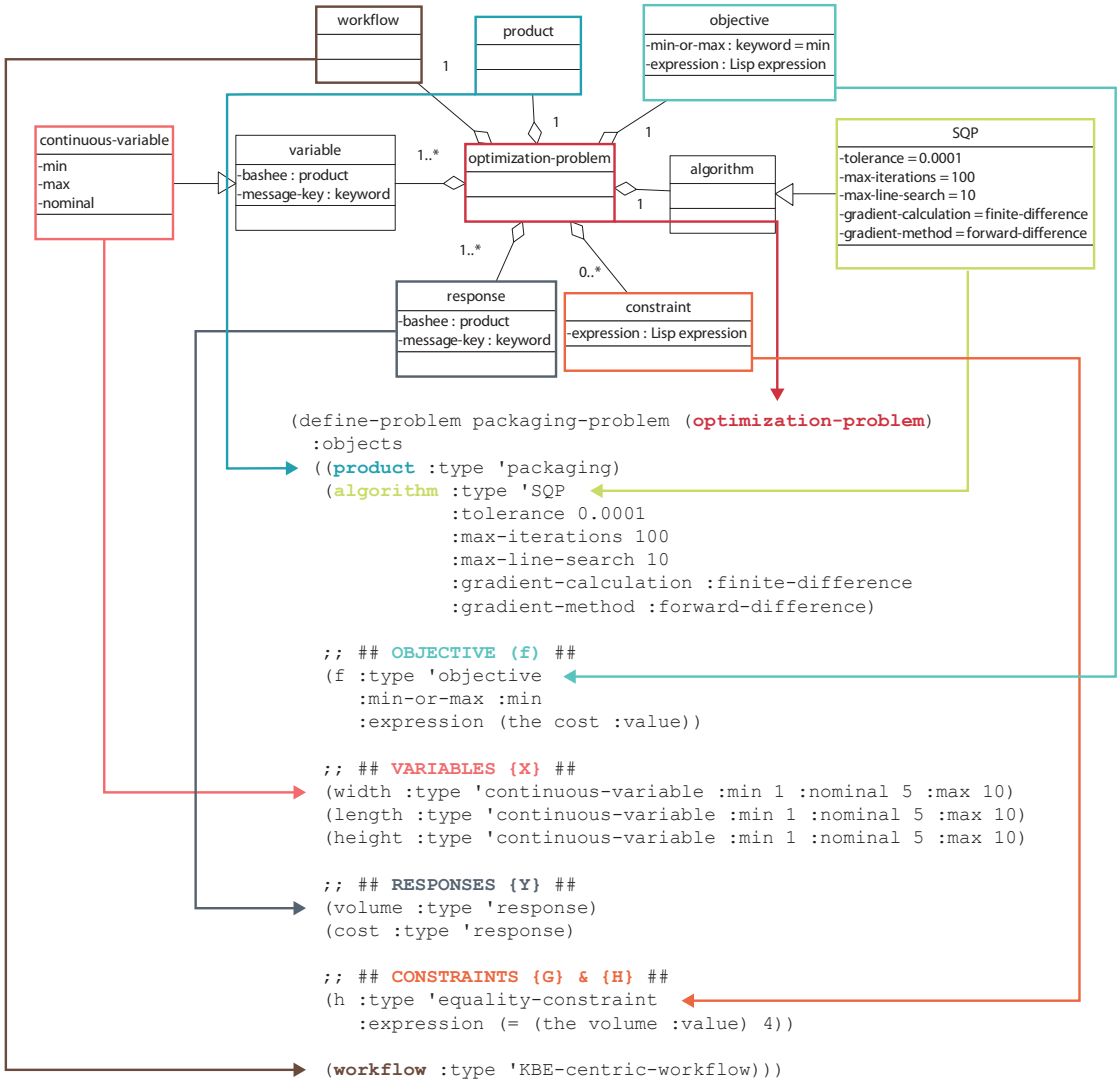


Figure 6.3: GenDL source code for the new problem object.

Looking closely at the code reveals that the problem is only composed of objects. These objects are instantiated once the problem object is instantiated. The objects are common to design optimisation, but have also been explained in Chapter 5. The only new class is *SQP*, which is a subclass of algorithm and introduces new attributes that are native to the SQP algorithm.
The first object, *product*, refers to the packaging code in Figure 6.2. Further down the code are the *responses*. Responses are necessary for evaluating the expressions in the objective function and constraints in GenDL (hence the keyword `:value`, which retrieves the value of the parameter). The versatility of the language allows users to create additional checks or even perform optimisation within GenDL. But in this use case, the problem is solved by a workflow, which is placed at the bottom of the code.

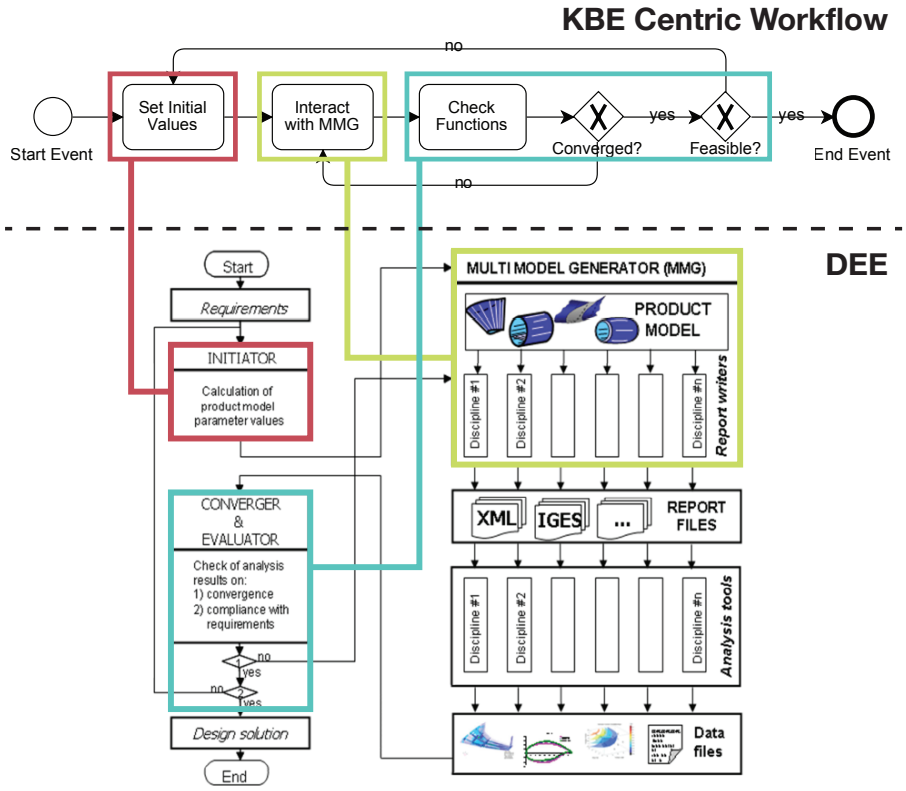


Figure 6.4: The KBE Centric Workflow follows the DEE framework.

KBE Centric Workflow

Normally, the programmer would build a workflow with the new domain-specific language in GenDL (see Section 5.2). But for this use case, a new high-level workflow (*KBE Centric Workflow*) has been developed for optimising KBE product models. It follows the DEE framework, although it does not involve any analysis tools (see Figure 6.4).
The KBE Centric Workflow captures the knowledge about the activities, such as parameters for sending HTTP requests and knowledge about modelling design variables and constraints in Optimus. Then, with only the information from the problem object, a workflow is generated in Optimus that will solve the problem. Thus, the user is not troubled with modelling the workflow. This knowledge is stored in the code so that optimisation becomes more accessible.

The KBE Centric Workflow consists of three activities. The main activity is the interaction with the MMG. During each iteration, this activity updates the product model and retrieves responses that appear in the objective function and constraints.
The activity is a **HLA**, where the inputs are the design variables and the outputs the responses. There are two transformation steps: **analysis** and **postprocessing**. The analysis step sends a HTTP request to GenDL, and thus contains knowledge about server location and how to build the query string. The postprocessor reads the HTTP response that is returned, which is done with *extraction rules* in Optimus. This object contains knowledge about how to create these extraction rules for the outputs of the HLA.

The remaining two activities are default activities that appear in every workflow. *"Set initial values"* maps onto an input array in Optimus, which is how design variables are declared in Optimus. *"Check functions"* maps onto an output array, and contains the formulas for the objective function and constraints and sets the constraints (i.e. equality type and the value).

Because the entire workflow has been parameterised, it is not bound to only this packaging product. With this solution, it will be significantly easier for KBE users without SWFM experience, to optimise KBE product models. The results section shows how this code is mapped onto an Optimus workflow.

Execution
After compilation, the whole optimisation process is initiated either from the command line or from GenDL's GUI, called Tasty. From the command line, this can be done with only two commands (see Listing 6.1). The first line instantiates the problem and all its child objects, including the product. Then, a simple command will trigger the whole process. This is all the input the user has to provide.

Listing 6.1: GenDL commands for instantiating the problem object, and to initiate the optimisation process.

```
>(setf MyProblem (make-problem 'packaging))

>(the-object MyProblem :optimize!)
```

The same process can also be triggered from within the GUI. One of the computed-slots is linked to the same function (or method) `:optimize!`. Thus, clicking the attribute in Tasty triggers the same action (attributes of an object are presented in the inspector of Tasty, i.e. the table in the middle in Figure 6.5).

Behind the scenes, GenDL sends HTTP requests to the Optimus Python server to generate the workflow elements and set up the optimisation problem. When this is finished, a final request is sent to execute the workflow. During each iteration, Optimus sends an HTTP request to the GenDL server to update the packaging object with new values for the design variables and to retrieve updated values for cost (the objective) and volume (the constraint). When Optimus has reached an optimum, it saves these values to a file. This file is then parsed to update the packaging with the optimal values.

6.2.2 Results

The results section shows the product geometry, generated workflow, and optimisation results. It starts with inspecting the geometry in Tasty. Figure 6.5 shows the instantiated packaging in its initial state.

Optimus has generated the workflow that will optimise the packaging (see Figure 6.6). The figure shows how the KBE Centric Workflow has been translated into Optimus workflow elements. The series of screenshots in Figures 6.7, 6.8, 6.9, and 6.10, show how the KBE code is translated into the workflow. The values have been magnified for better readability.

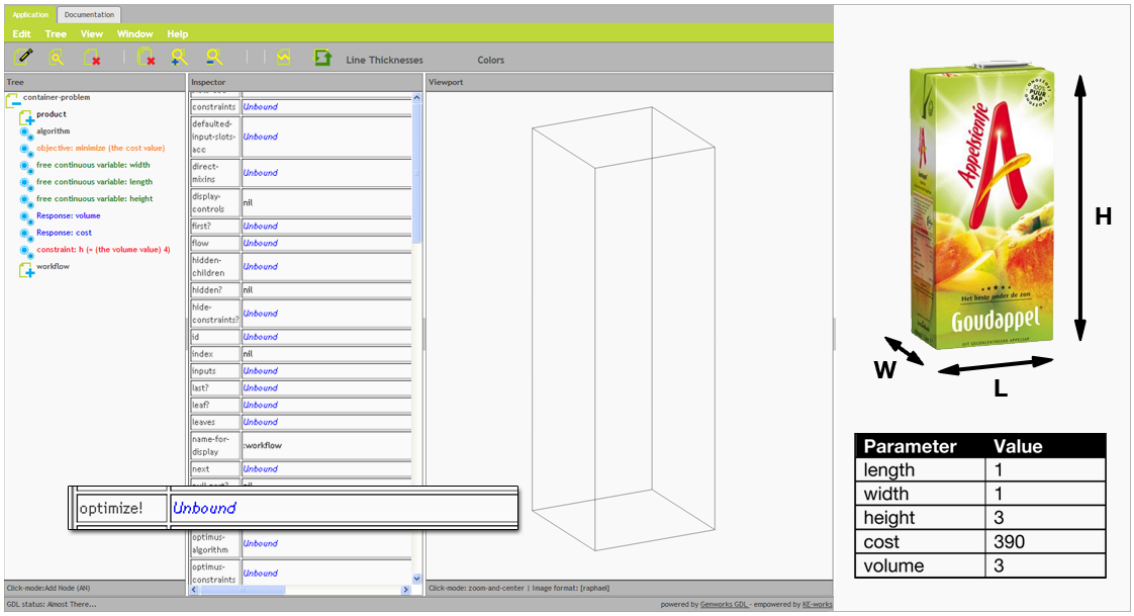


Figure 6.5: Screenshot of Tasty showing the packaging before optimisation. The inspector shows the computed-slot `:optimize!` which triggers the optimisation process.

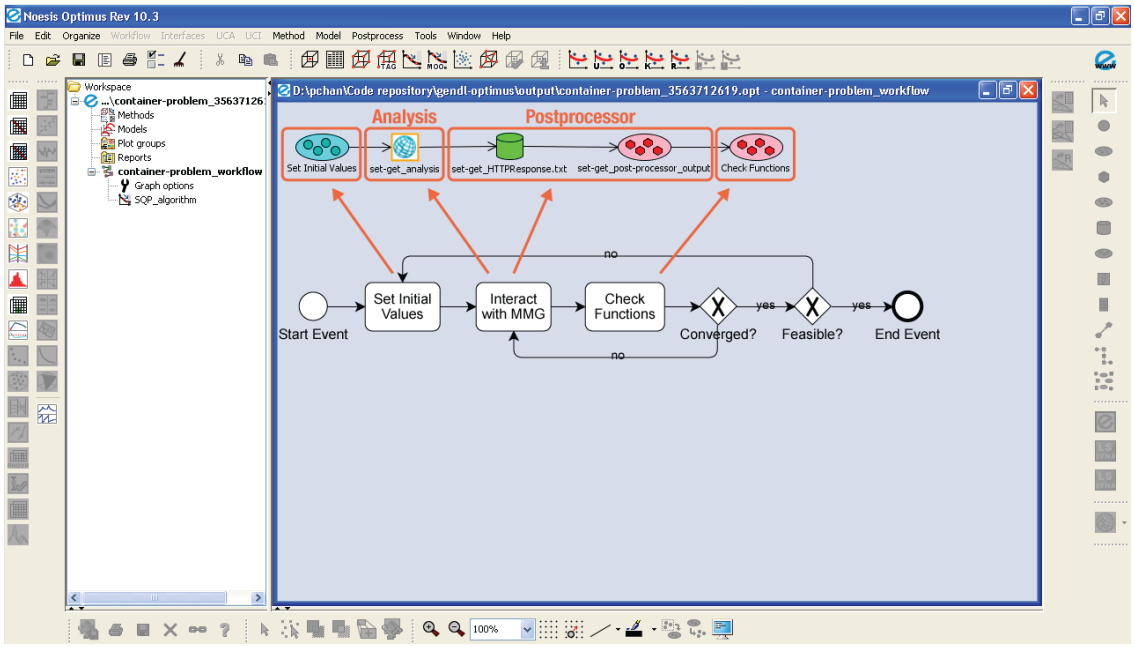


Figure 6.6: Optimus screenshot of the generated workflow. The figure illustrates how each activity of the KBE Centric Workflow is mapped onto the Optimus workflow.

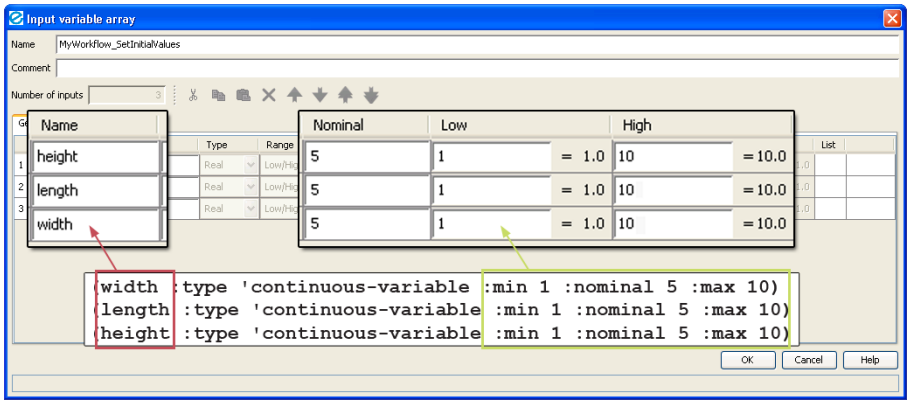


Figure 6.7: Optimus screenshot of the design variables. It shows how the KBE code is translated to values in the Optimus workflow.

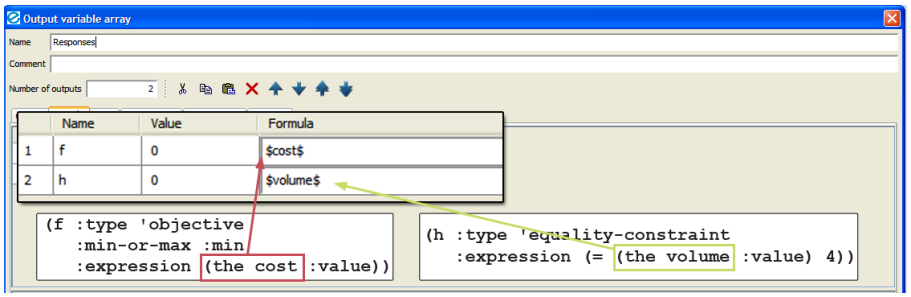


Figure 6.8: Optimus screenshot of the cost objective function and volume constraint formulas.

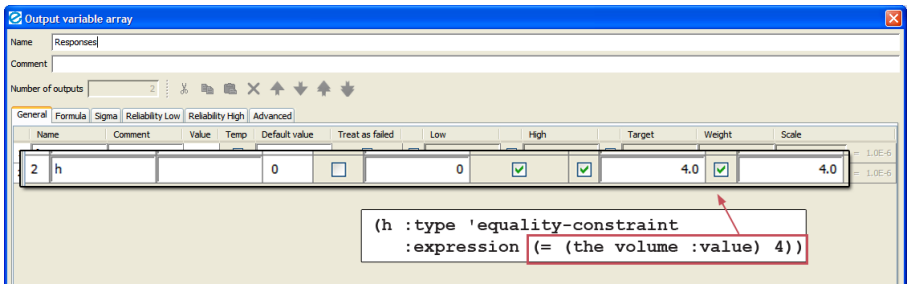


Figure 6.9: Optimus screenshot of the volume constraint equation (equality type and value).

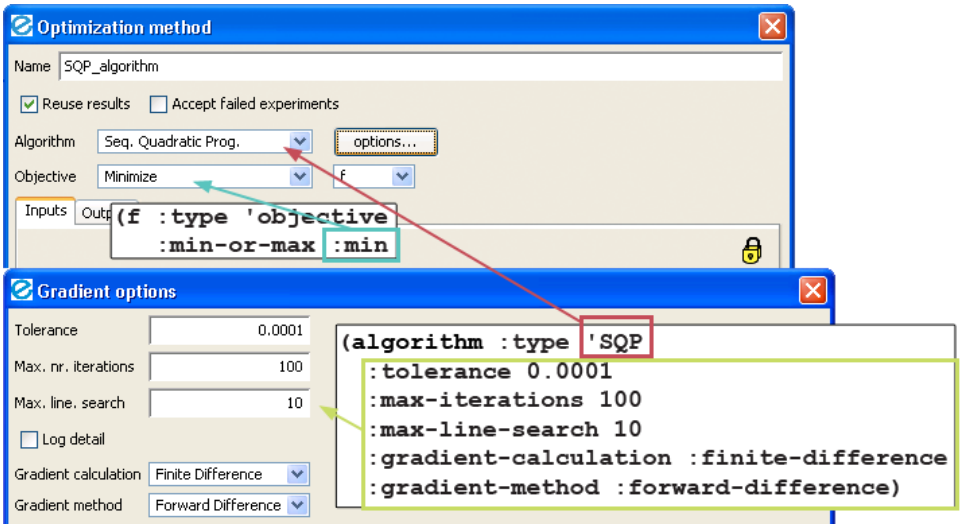


Figure 6.10: Optimus screenshot of the optimisation settings.

From the moment that the user gives the command, the whole process is executed automatically. Generating the workflow and setting up the optimisation problem takes less than a second for such a small workflow. When Optimus has found an optimum, the packaging is automatically updated with the optimal values. The optimised geometry is again inspected using the Tasty web interface (see Figure 6.11).

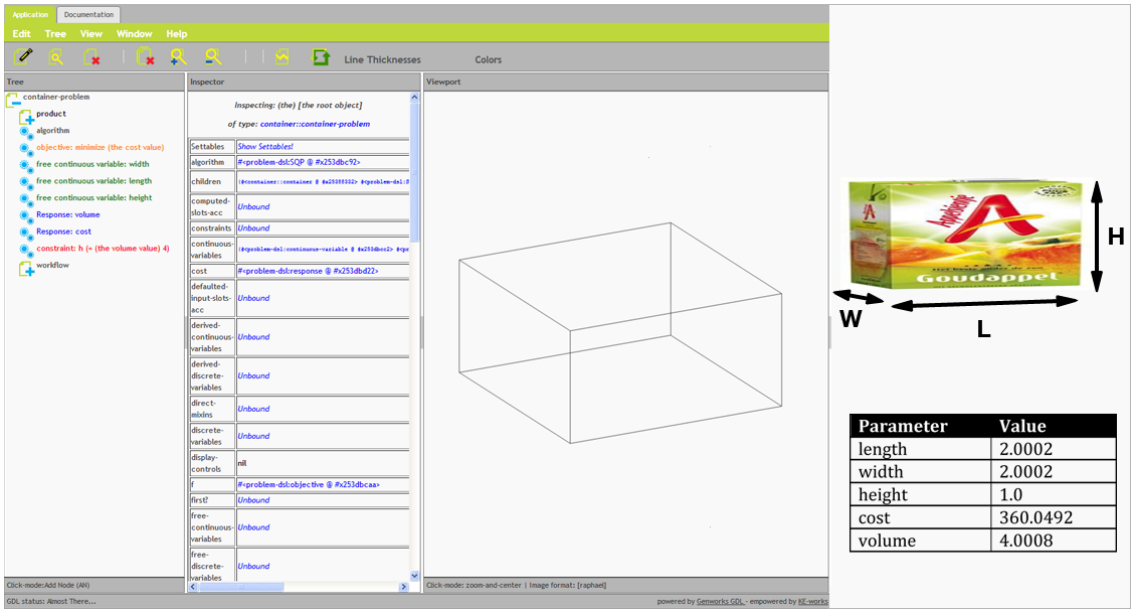


Figure 6.11: Screenshot of Tasty showing the optimised packaging.

There is a slight deviation from the analytical results because of the tolerance (see Table 6.1). Aside from that, the results show the optimal values.

Table 6.1: Comparison of the analytical solution with the simulation results. The differences are caused by the tolerance value.

Parameter	Analytical solution	Simulation results
length	2	2.0002
width	2	2.0002
height	1	1.0
cost	360	360.0492
volume	4	4.0008

Reconfigurability

The real strength of parametric, object-oriented modelling emerges once changes are made to the design problem. For this second test, two changes are made:

1. The *height* parameter is removed as a design variable. It is now fixed to its initial value of 3.
2. A new *weight constraint* is added to the problem: $\rho V < 5000$ (where $\rho = 1200$)

To incorporate these changes, the user has to alter the problem code from Figure 6.3. For the first change, the user can simply remove the line for *height* under variables. The second change can be implemented in two ways:

1. Add *weight* to the computed-slots of the packaging object and let GenDL calculate the weight.
2. Model the constraint as a mathematical expression and let Optimus evaluate the constraint instead.

The first is exactly the same as the volume constraint. GenDL evaluates the expression and returns the value to Optimus, which checks the constraint. It requires two changes to the product code, highlighted in Figure 6.12.

The second option allows users to use formulas for constraints in the problem object. The following lines are added to the problem object to include the new weight constraint.

```
(g :type 'opt:inequality-constraint
   :expression (< (* (the density :value) (the volume :value)) 5000))
```

Note that also here the density should be added to the input-slots of the packaging object and as an additional response to the problem.

All other objects in the problem code remain the same, including the workflow. Because it is parametric, it will change automatically according to the new problem definition. The user can simply execute the optimisation process without worrying about how these changes affect the workflow.

The presented results emphasise on the parametric characteristics of the workflow. First, the *height* now no longer appears in the list of design variables in the Optimus workflow. Additionally, the HTTP request sent by *"Interact with MMG"* has been adjusted automatically to exclude *height* as well. A comparison between the two requests is shown here, where *height* is omitted in the new request.

```
(define-object packaging (box)
  :input-slots
  (length width height
    (cost/m2-top 10) (cost/m2-bottom 20) (cost/m2-sides 30)
    (density 1200))
  :computed-slots
  ((cost (+ (the top :cost) (the bottom :cost)
            (sum-elements (the sides) (the-element :cost))))
  (volume (* (the width) (the height) (the length)))
  (weight (* (the density) (the volume))))
  :objects
  ((top :type 'face
        :width (the width) :length (the length)
        :cost/m2 (the cost/m2-top))
  (sides :type 'face
         :sequence (:size 4)
         :width (nth (the-child :index)
                     (list (the width) (the width)
                 (the length) (the length)))
         :length (the height)
         :cost/m2 (the cost/m2-sides))
  (bottom :type 'face
          :width (the width) :length (the length)
          :cost/m2 (the cost/m2-bottom))))
```

Figure 6.12: The *density* is added to the input-slots (in blue) of the packaging source code, and *weight* to the computed-slots (in red).

Previous HTTP request: http://127.0.0.1:9000/Optimus-GenDL/set-get?id=g30289&variables(0)=width&values(0)=$width$&variables(1)=length&values(1)=$length$&**variables(2)=height&values(2)=$height$**&responses(0)=volume&responses(1)=cost

New HTTP request: http://127.0.0.1:9000/Optimus-GenDL/set-get?id=g30954&variables(0)=width&values(0)=$width$&variables(1)=length&values(1)=$length$&responses(0)=volume&**responses(1)=density**&responses(2)=cost

Name	Value	Formula
f	0	$cost$
g	0	($density$ * $volume$) New constraint
h	0	$volume$

Figure 6.13: Constraints (and objective functions) can be full mathematical expressions as is shown here.

The new weight constraint is an expression that is evaluated by Optimus (see Figure 6.13). Note that the new equation requires the values for ρ and V to calculate the weight. The *KBE Centric Workflow* understands this and automatically updates the HTTP request to retrieve the density in the new request. Additionally, the *output file parser* is also updated to parse the new values.
The new workflow is generated and executed automatically in the background. This example is a good demonstration of how everything that is connected changes according to modifications in the definition. The optimised values of the reconfigured optimisation problem are listed in Table 6.2

Table 6.2: The results of the new design problem with a modified design variable and constraint.

Parameter	Initial value	Optimised value (Old problem)	Optimised value (New problem)
length	1	2.0002	1.1558
width	1	2.0002	1.1547
height	3	1.0	3
cost	390	360.0492	455.9405
volume	3	4.0008	4.0040
weight	3600	4800.96	4804.8098

6.3 Discussion

This chapter demonstrated a solution that extends KBE platforms with design optimisation capabilities. It showed how the problem and workflow DSL are used in practice. Moreover, the workflow DSL has been used to build a parametric high-level workflow (*KBE Centric Workflow*). With this solution, the user only needs to model the product and problem to perform design optimisation. Execution is an automated process, as the workflow is generated and executed automatically in the background by Optimus. Furthermore, the parametric capabilities of the KBE Centric Workflow enables users to quickly reconfigure problems without modifying the workflow. For optimisation problems, the product object is updated with its optimum values at the end.

At the end of this use case, it can be concluded that:

- The common engineering language in the problem object is lightweight, domain-specific, and thus also accessible to newcomers. Moreover, these newcomers can use the GUI to model the problem instead of writing programming code (see Figure 5.6).
- The KBE Centric Workflow introduced parametric process modelling for design engineers without SWFM experience to solve their design problem.
- Automatically generating the workflow has been successful, as the problem is solved in the background and the product model is automatically updated with the optimum values. These values have been validated with the analytical results.
- Since both GenDL and Optimus are deployed as web services, it is possible to implement this coupling across physical boundaries.

Limitations and Future Direction

Although the coupling provides powerful features for automated design optimisation, a few limitations are worth mentioning. Until now, it has only been tested with this relatively simple object. Although deemed possible, it is unknown how the system will handle more complex products composed of multiple sub-components. Similarly, it is expected that more complex workflows can be generated that involve external CAE tools, but this has also not been tested in this use case. It is recommended to investigate these possibilities in the future.
The current solution is limited to single-level optimisation problems only. Work is being

done on extending functionality to multi-level optimisation problems. This requires a top-level optimiser that acts as a controller, varying problem definitions of sub-problems. With a parametric workflow, the top-level optimiser could rapidly generate workflows, each solving one problem, and select the best solution.

Normally, Optimus provides extensive postprocessing tools through the GUI. However, direct access to the data through the API is not available yet, thus limiting analysis capabilities in this system. It is expected that future updates of Optimus will provide this functionality. Meanwhile, it would be possible to provide postprocessing capabilities in GenDL. This has not been implemented yet as it was not within the scope of this research.
Furthermore, only a limited number of optimisation methods have been implemented in the Python wrapper. Optimus provides a whole range of methods, but for this prototype it was not necessary to include all.

Even though the coupling is a great addition to KBE platforms, it does not solve the issues associated with KBE itself. Some transparency issues remain, as the knowledge is still mainly hidden in the code. Eventually, this may lead to the loss of knowledge or limit reuse due to misunderstanding of the KBE application. Therefore, another solution has been developed that applies MOKA 2's methodological approach using a KB.

Chapter 7

Use Case 2: KB–PIDO Coupling

The use case in this chapter demonstrates the methodological approach for the same packaging optimisation problem as in the previous use case. Applying the methodology changes the focus from programming to modelling, which allows for design engineers to be more involved in the modelling process. The simulation workflow is generated directly from the KB and, unlike the previous use case, there is no coding involved. Moreover, when knowledge is captured and structured according to the methodology, it will be clear how the end result (generated workflow) is obtained. Since the problem is identical, it is expected that the generated workflow will be identical as well. An additional objective is to recreate the *KBE Centric Workflow* in this approach.

The step-by-step instructions of the methodology are central to this use case, starting at the informal level (Section 7.1.1) and continuing to the formal level (Section 7.1.2). An additional step (step 6*) describes how the workflow can be parameterised to replicate the KBE Centric Workflow. The chapter is completed with a discussion on the results and experiences (Section 7.2).
Note that this thesis focuses on the process side of knowledge modelling, even though there are links to the product knowledge. The chapter begins with the assumption that:

- the packaging product has already been modelled in the KB.
- that the KBE code for packaging object is either written manually or automatically generated.
- and that the packaging object is instantiated before the workflow is executed.

7.1 Implementation

Several actors are involved in the modelling process. Identified in Chapter 4, these are:

- Design engineer - responsible for steps 1 and 2.
- IT engineer - responsible for step 3, provide support in step 5.
- Knowledge engineer - responsible for steps 4 and 5.

7.1.1 Informal Model

This section shows the standard methodological approach to model simulation workflows. As a result, the process will iterate over steps 2 and 3. However, for such a simple problem, it is likely that the N^2 and BPMN diagrams can be modelled immediately. There are no restrictions in the methodology that prohibit users from doing this. But for demonstration purposes, the modelling process is explained step-by-step. This begins with formulating the problem statement.

Step 1: Define problem statement
The (mathematical) problem has not changed from the description in Section 6.1. Reformulating the design problem in a problem statement gives:

> *"Design a **packaging** with **minimum cost** and a **fixed volume** of $4m^3$."*

From this statement it can be concluded that:

- The product is a packaging.
- It is an optimisation problem.
- The objective is to minimise cost.
- There is one equality constraint: the volume is fixed at $4m^3$.

Step 2: Structure the problem and disciplines
The design engineer works through step 2 with the information from step 1. Here, the N^2 notation helps the design engineer to visualise the relationships between the problem and disciplines.

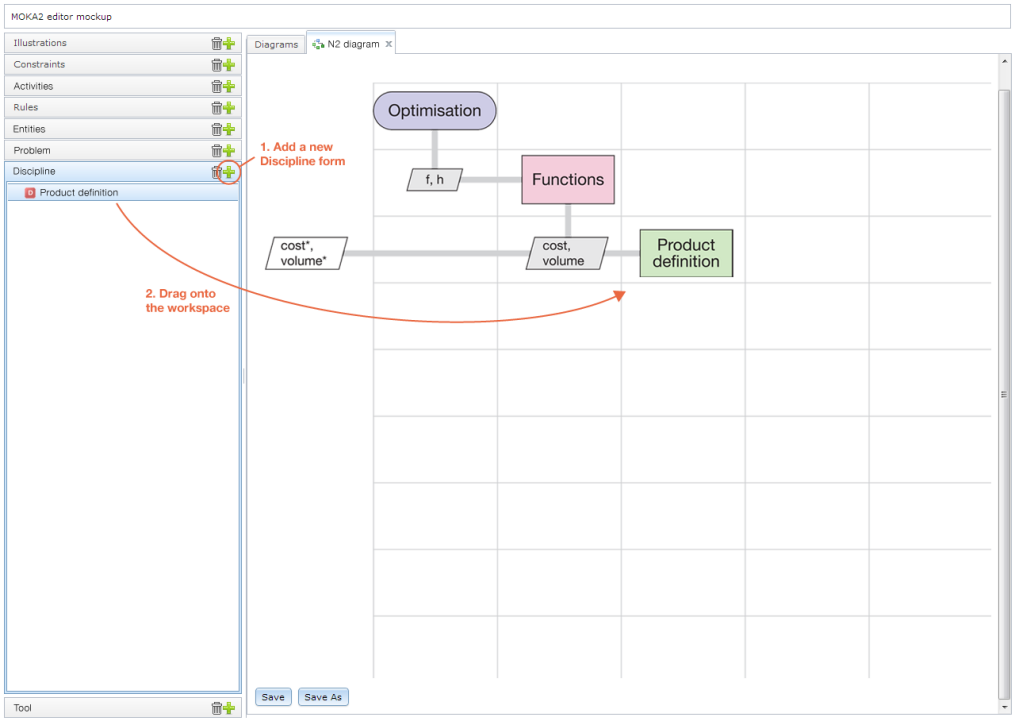


Figure 7.1: N^2 diagram describing the packaging optimisation problem. At this stage, the diagram can only be filled partially.

At this stage, there is only one discipline: *Product definition*. From the problem statement it is known that the two responses *cost* and *volume* are needed to evaluate the objective function and constraints. These can be placed in the diagram as shown in Figure 7.1. It is also known that it is an optimisation problem, hence *optimisation* is placed in the top left corner. Once this is finished, the methodology proceeds to step 3.

Step 3: Model the activities
In step 3, the IT engineer models the activities for the discipline *Product definition*. From the output of the discipline that is known (*cost* and *volume*), the IT engineer can reason which sequence of activities can produce this output. In this case, there is only one HLA (see Figure 7.2). This activity requires input from outside this discipline. This new information can be used to update the N^2 diagram.

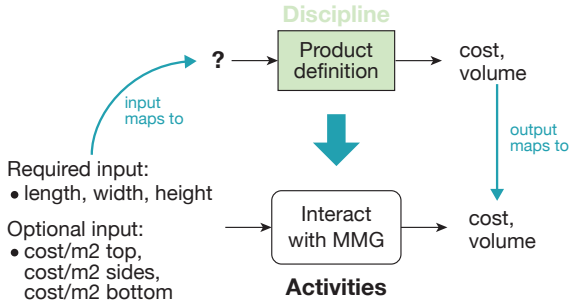


Figure 7.2: The *Product definition* discipline has only one HLA.

Step 2, iteration 2: Update the N^2 diagram
Modelling the problem and modelling activities is an iterative process. In this second iteration, the design engineer updates the N^2 diagram with the inputs found in step 3.

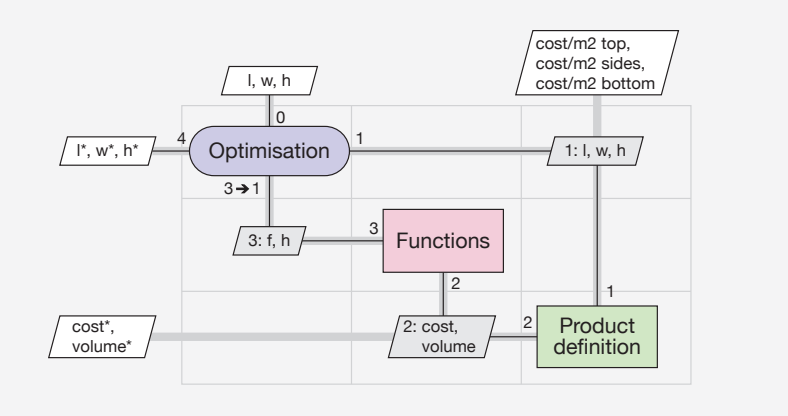


Figure 7.3: A completed N^2 diagram describing the packaging optimisation problem.

The inputs for product definition are placed in the same column as the discipline (see Figure 7.3). The three *cost/m²* parameters are external inputs, and therefore positioned along the top edge of the diagram. In this use case, these parameters are optional and have tool defaults (i.e. values are provided in the KBE application). The *length*, *width*, and *height* of the packaging are the design variables, and are placed in the N^2 diagram

above the optimisation block. Now, the N^2 diagram has been completed and the process continues to step 3 to finish the workflow diagram.

Step 3, iteration 2: Completing the workflow diagram
The IT engineer completes the workflow with two standard activities: one for setting initial values (for declaring design variables) and one for checking objective and constraint functions in the workflow. The final workflow is modelled as shown in Figure 7.4.

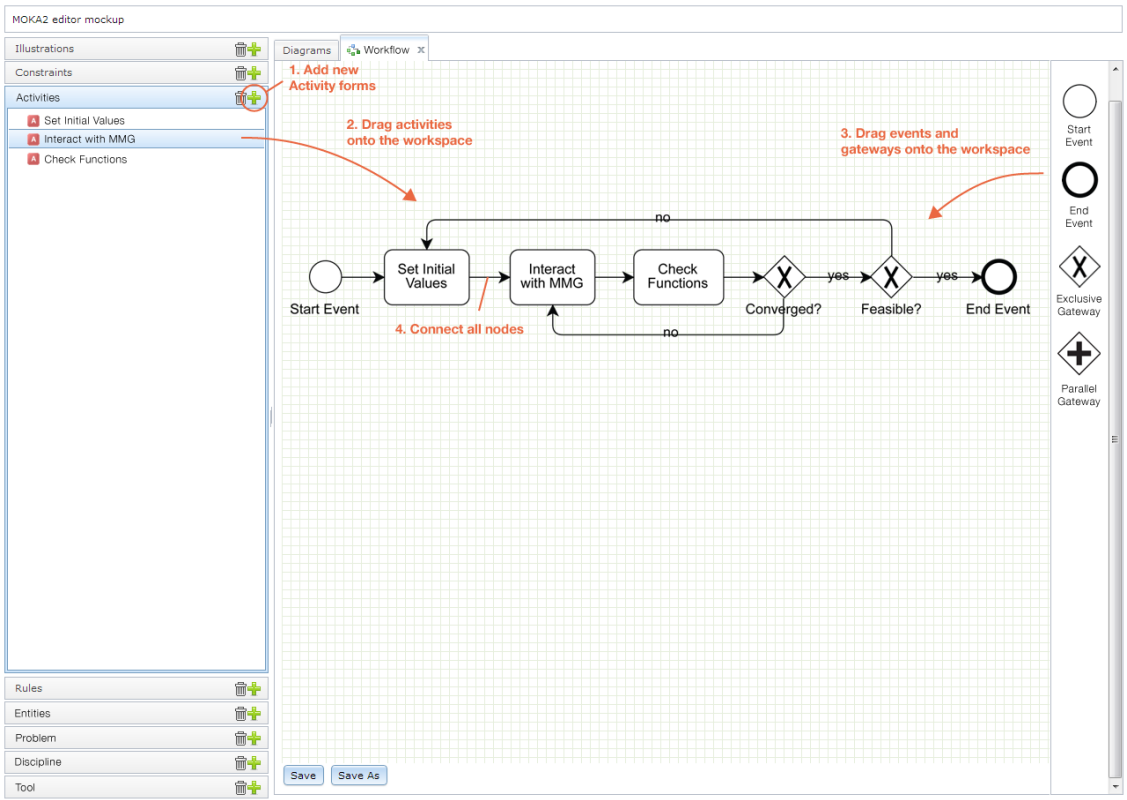


Figure 7.4: Mockup showing how the final workflow is modelled in a web interface.

Once both diagrams have been completed, the process proceeds to step 4.

Step 4: Fill in ICARE PDT-forms
In this step, the knowledge engineer captures knowledge of the problem, disciplines, activities, and software tools in *ICARE PDT-forms*. Figure 7.5 shows an example of a P-form for the packaging design problem. The P-form provides fields to describe the problem in more detail. Among these fields are a description of the problem, the optimisation algorithm and settings, and values of the design variables. Examples of the remaining forms are included in Appendix E.

When the diagrams are completed and the ICARE PDT-forms filled, the modelling process continues with formalising this knowledge.

MOKA DPM form:	Problem				
Project name	Packaging design				
Project phase	Preliminary design				
Name	MyOptimization				
Reference	O1				
Objective	Minimise the cost of the packaging.				
Description	Minimise the material cost of the packaging under the condition that the volume is fixed.				
Problems involved	-				
Disciplines involved	Product Definition (D1)				
Entities involved	MyProduct (E1)				
Activities involved	MyWorkflow (A1)				
Algorithm	SQP				
Description	Sequential quadratic programming (SQP) is an iterative method for nonlinear optimization.				
Setting	Max iterations	Max line search	Tolerance	Gradient calculation	Gradient method
Value	100	10	0.0001	Finite difference	Forward difference
Objective function	Cost objective (R1)				
Constraints	Volume constraint (C1)	Weight constraint (C2)			
Design Variables					
Name	length	width	height		
Description	Length of the packaging	Width of the packaging	Height of the packaging		
Initial value	5	5	5		
Lowerbound	1	1	1		
Upperbound	10	10	10		
Stepsize	-	-	-		
Discrete values	-	-	-		
Management					
Author	P. Chan				
Date	29-12-2012				
Version number	1				
Status	In Progress				
Modification	-				
Information origin	P. Chan				

Figure 7.5: The P-form captures the packaging problem details.

7.1.2 Formal Model

The objective is to map the knowledge that is stored in diagrams and forms onto the formal process ontology. This formalisation process is a task for the knowledge engineer, who understands the ontology and knows what input is required for automatic workflow generation. This section shows how the problem and workflow are modelled in the ontology.

In future updates, the ontology is modelled in a web interface (see Appendix C). The interfaces presented in this thesis are only mockups and not yet functional (except for the interface for filling in ICARE PDT forms; see Figure C.6). Therefore, the ontology is modelled in Protégé (Protégé, 2013), an editor for modelling OWL ontologies (see Figure 7.6).

Step 5: Formalise process knowledge
Starting with the workflow, the knowledge engineer has to model all the activities (incl. all their properties) and connections. This part of the explanation follows a bottom-up approach. Thus, the end result is shown first for every activity.

Modelling the activities
There is one HLA in the workflow: *"Interact with MMG"*. Figure 7.7 shows how this activity is modelled in Optimus.
The first activity sends an HTTP request to GenDL and receives a response that is saved in a file by Optimus. This is then parsed by a file parser. Next to the activities, the figure also shows which HTTP request is sent and the two extraction rules for parsing the HTTP response exactly as it appears in Optimus.

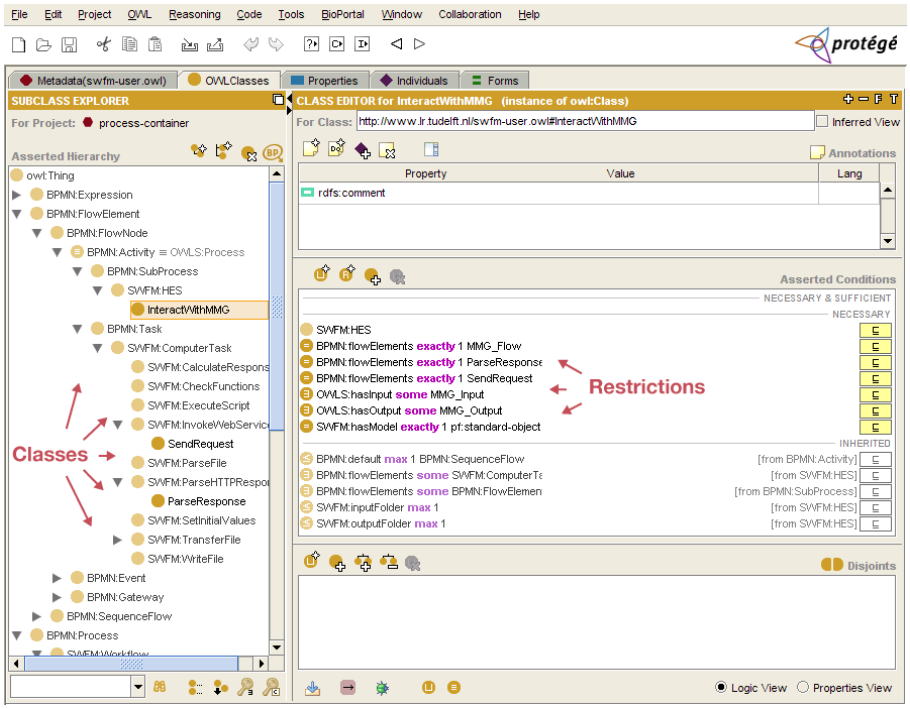


Figure 7.6: Screenshot of OWL classes and restrictions modelled in Protégé.

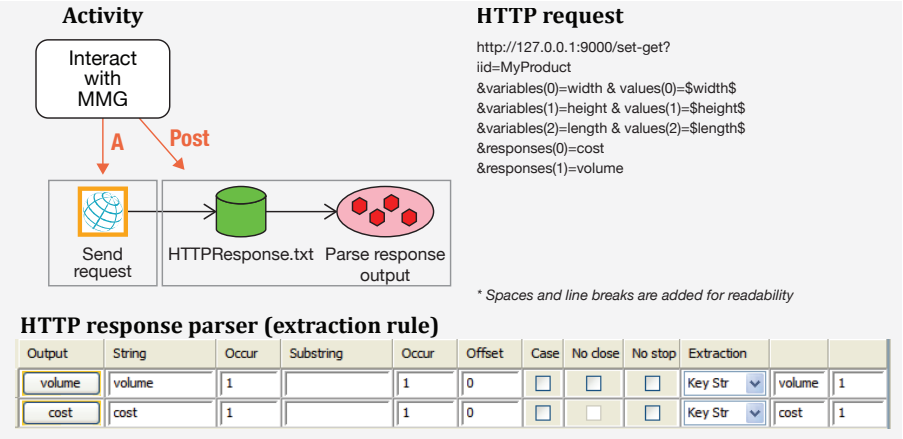


Figure 7.7: This figure shows how the HLA *"Interact with MMG"* is modelled in Optimus. It has one *Analysis* and one *Postprocessing* step.

This is then modelled in the ontology as shown in Figure 7.8). The knowledge engineer models all the classes, properties, and restrictions, and ensures it follows the informal process model. Then, the figure highlights which classes are instantiated by the user, the knowledge engineer, and the computer. Starting from the left, the user connects the product model to the HLA with the `hasModel` property. That would be the packaging instance. The inputs to this activity are provided in Table 7.1. Below, in Table 7.2, are the outputs of the activity.

The binding with the product attribute (last column in Table 7.1) is necessary for getting

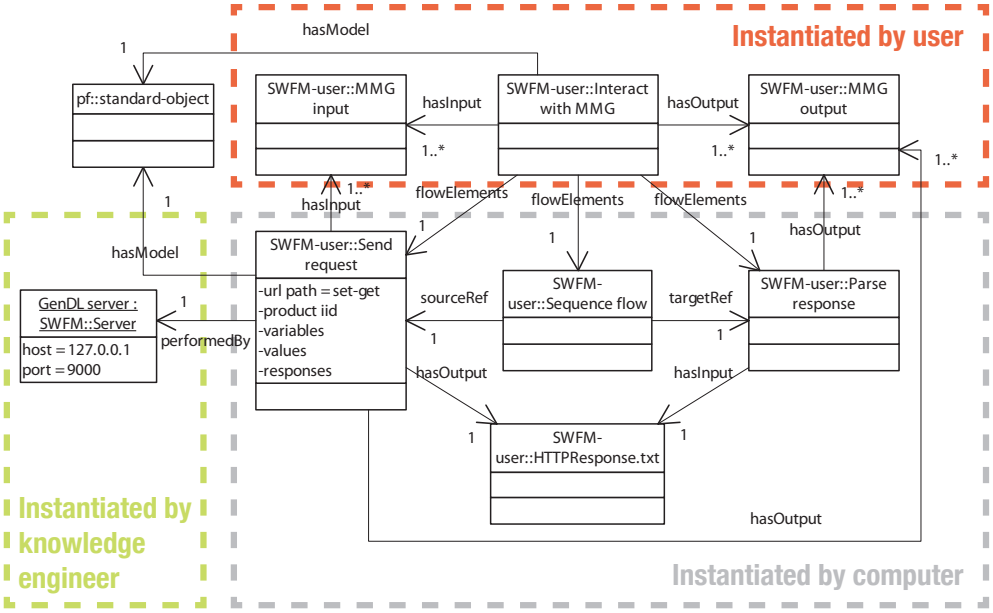


Figure 7.8: UML class diagram of the HLA *"Interact with MMG"*.

Table 7.1: Inputs for *"Interact with MMG"*

Input	Variable/ Fixed	Required/ Optional	Parameter Value	Default Value	Binding
length	Variable	Required	-	-	packaging.length
width	Variable	Required	-	-	packaging.width
height	Variable	Required	-	-	packaging.height

Table 7.2: Outputs for *"Interact with MMG"*

Output	File parameter settings						Binding
	label	word nr	occurrence	row offset	start pos	end pos	
cost	cost	-	-	-	-	-	packaging. cost
volume	volume	-	-	-	-	-	packaging. volume

the right name, which is then used to update the product model. This is an example where the relationship between product and process is unmissable. Eventually, the inputs and outputs are mapped onto the query string.

- Inputs to *variables(i)* and *values(i)*.
- Outputs to *responses(i)*

Outputs are also translated to extraction rules for file parsing (according to the file parameter settings in Table 7.2). This is shown in Rule R19 in Appendix D.
These are trivial tasks in the modelling process, and are therefore done by the RE. The knowledge of how to do it, is captured in a rule (Rule R13 in Appendix D). Rules are

also used to have the computer instantiate classes (see Figure 7.8). Figure 7.9 shows a screenshot of a rule that instantiates flow elements.

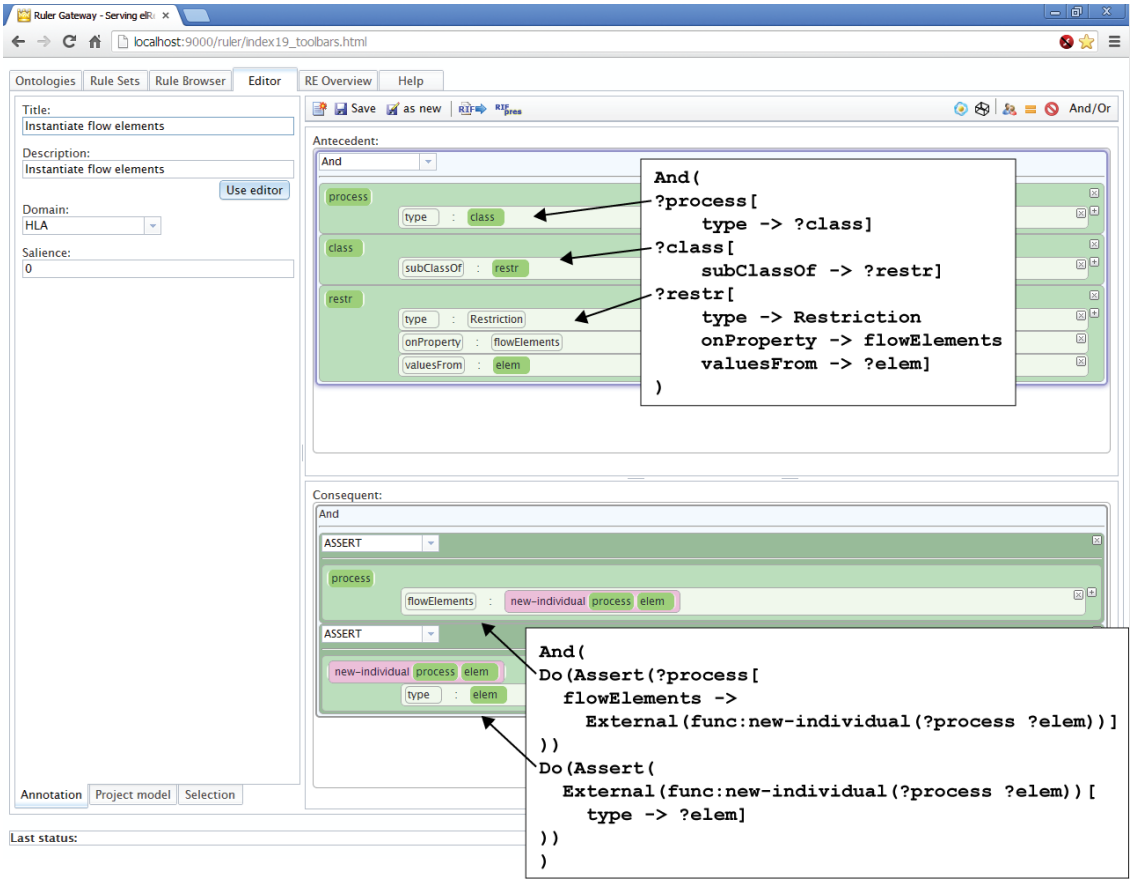


Figure 7.9: Screenshot of the Ruler interface developed by Reijnders (2012). The boxes display the rule in RIF, written in frame logic.

This rule instantiates the flow elements of the HLA, thus the two activities and one sequence flow. Rule modelling works with variables, which are similar to the variables in SPARQL (terms with '?'). If a pattern in the triple store matches the statement in the antecedent, then the RE will execute what is specified in the consequent. Simply stated, this rule looks for elements (`?elem`) that need to be instantiated. This works as follows:

1. The RE matches objects to `?process`. Its class has a restriction on the property `flowElements`. In this example, the HLA instance would be a match.
2. The restriction has the information about the element that needs to be instantiated, which is stored in the variable `?elem`. For example, *Send request* is an `?elem`.
3. Then, according to the consequent, two new triples are asserted:
   (a) The first assigns the new instance to the `?process` found in step 1. The function *new-individual* aggregates two URIs to create a new URI for the new instance.
   (b) The second states that the new instance is of type `?elem`.

This example showed how restrictions in the ontology cooperate with rules to instantiate flow elements. The entire list of rules applied to this use case can be found at the end

of this section. The remaining two activities, *"Set initial values"* and *"Check functions"*, are explained after showing how the problem is modelled.

Modelling the problem

Before the workflow can be generated, the knowledge engineer has to set up the problem in the ontology (see Figure 7.10). Modelling the problem object and its attributes is fairly straightforward. This problem has three continuous design variables and is subjected to one constraint. The product instance is placed on the left, and has a direct link with the problem instance.

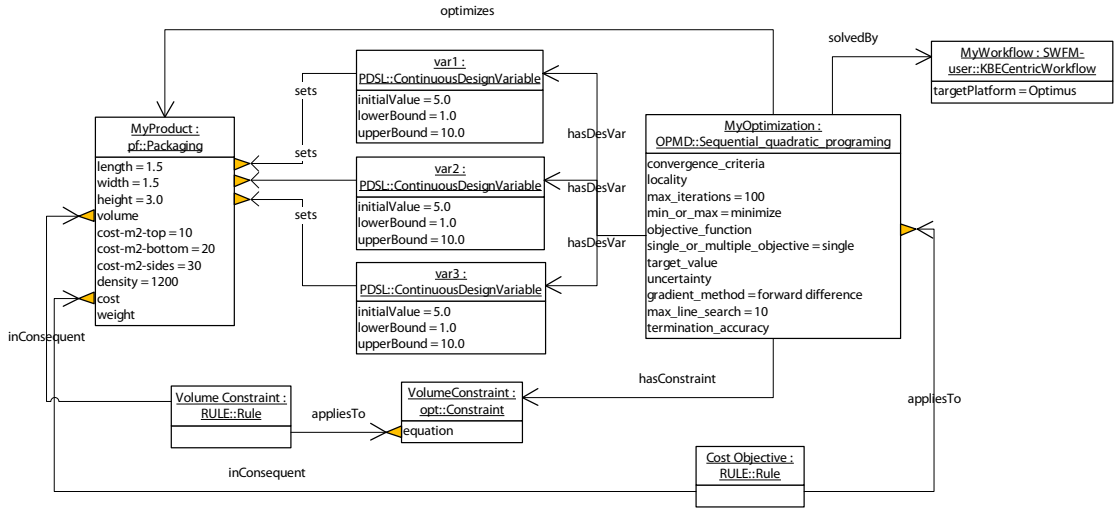


Figure 7.10: UML diagram of the packaging optimisation problem in the formal process ontology.

Figure 7.10 shows the relationships between the product, rule, and process domains. Besides the obvious relation between the problem and the product to be optimised, there are relationships between design variables and product attributes, and between rules and functions (constraint and objective). These have been modelled as follows.

Design Variables

Although in engineering terms it is natural to talk about the *length* as a design variable, in the ontology the *length* as a design variable is not the same as *length* as a product attribute. This is an intrinsic property of ontology modelling in OWL, where each entity is unique. Thus, instead of designating the product's *length* attribute as the actual design variable, a separate entity is created (*var1*) that `sets` the *length* attribute. The two entities are related, but a design variable is its own object with its own attributes. The values of design variables are determined per case, and can therefore not be fixed to the product attribute. This would affect all optimisation problems where the same product is the subject.

Design variables are inputs to the activity *"Set initial values"* (see Table 7.3). The activity maps onto an input array in Optimus, which declares the design variables in the workflow. The end result is shown in Figure 7.11.

Additionally, the activity has an output for every design variable. This output parameter has a UPI, which is the name that appears in the input array in Figure 7.11. Outputs have

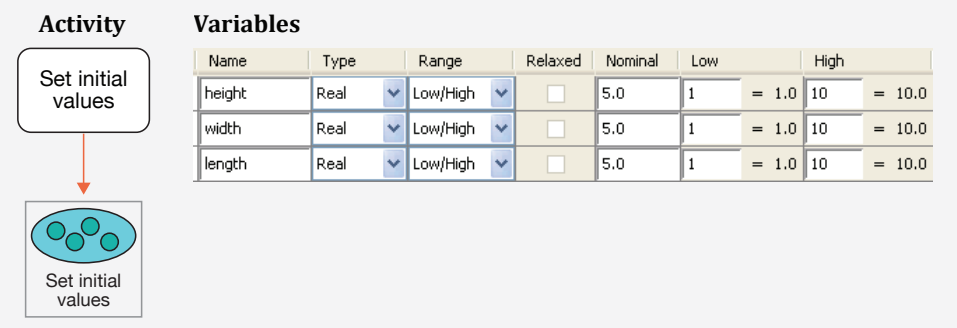


Figure 7.11: Figure showing how the activity *"Set initial values"* is modelled in Optimus.

Table 7.3: Inputs for *"Set initial values"*

Design Variable	Type	Initial value	Lower bound	Upper bound	Sets
var1	Continuous	5.0	1.0	10.0	packaging.length
var2	Continuous	5.0	1.0	10.0	packaging.width
var3	Continuous	5.0	1.0	10.0	packaging.height

been added to model the data flow in the Optimus workflow (see rule R3 in Appendix D).

Objective function and constraints

Mathematical formulas in the optimisation problem are entities defined in the rules domain. As Figure 7.10 shows, formulas (formally a Rule in the ontology) are linked to both the product parameters that appear in the formula and the entity that it applies to (constraints and objective function). Formulas are modelled in Ruler (see Figure 7.12), which includes a mathematics editor that stores expressions in MathML (an XML-based syntax for mathematical expressions). There are two formulas in this use case.

Objective function: $cost$
Volume constraint: $V = 4$

Every parameter in the formula has a *definitionURL* attribute in the MathML string, which adds semantics to MathML by linking the parameter to an entity in the KB. In this example, the *definitionURL* of parameter V is the URI of the packaging's volume.

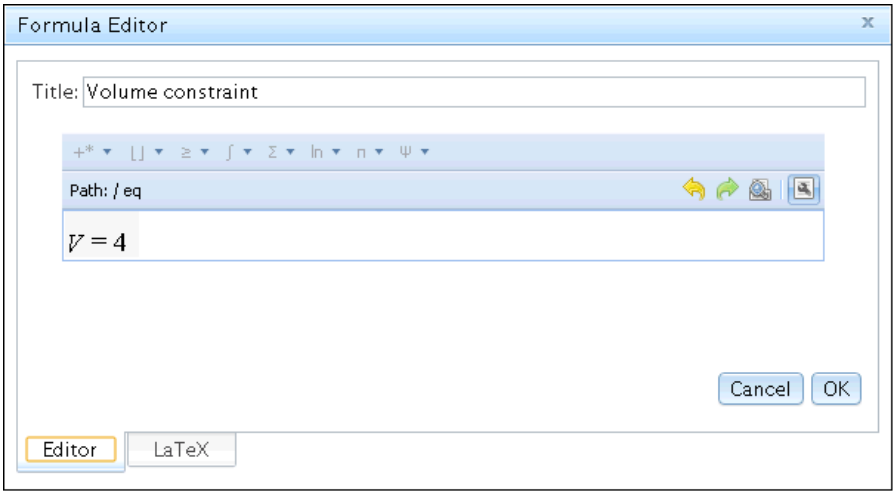
The objective function and constraints appear in the activity *"Check functions"* in Optimus. The MathML string is translated to Optimus' mathematics syntax and inserted in the formula field. The end result is shown in Figure 8.18.

Now that all three activities have been formalised, the workflow has been completed. Once the problem is linked to the workflow, it can be automatically generated in Optimus. However, one of the objectives for this use case is to recreate the *KBE Centric Workflow*. How this is done is explained in step 6*.

Step 6*: Parameterise the Workflow

Step 6 is starred because it is an additional step that is not part of the (core) methodology. It is included in this use case to show high-level modelling in practice.

In a parametric workflow, it is impossible to define certain facts in advance because it



MathML string

```
<math xmlns="http://www.w3.org/1998/Math/MathML" title="Volume constraint">
<apply><eq/>
<ci definitionURL="http://www.lr.tudelft.nl/KBE/product/formal/packaging#packaging.volume">V</ci>
<cn>4</cn></apply></math>
```

Figure 7.12: Screenshot of the volume constraint in the MathML editor. The string below the editor shows (in red) how the equation is stored. The *definitionURL* attribute adds semantics to MathML.

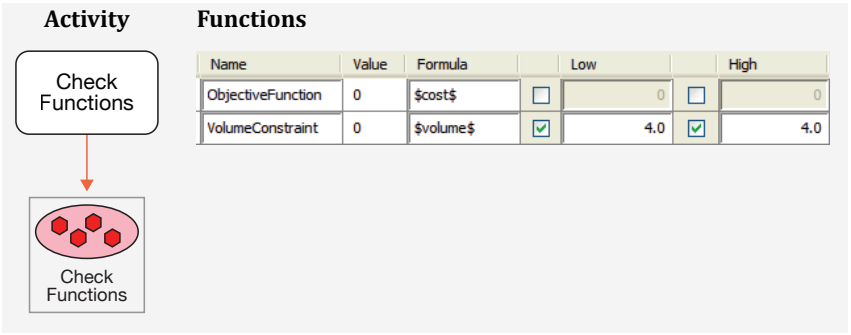


Figure 7.13: Figure showing how the activity *"Check functions"* is modelled in Optimus.

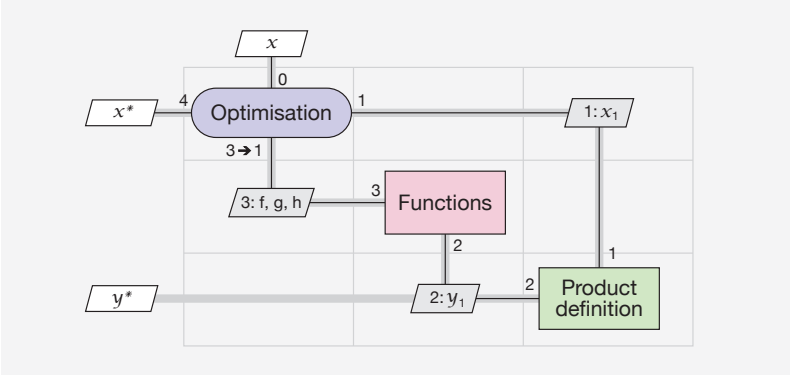


Figure 7.14: N^2 diagram of the parametric workflow. x_1 and y_1 vary per case.

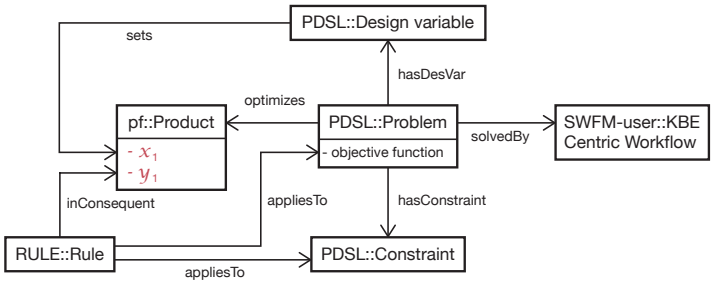


Figure 7.15: This figure shows what the origin is of x_1 and y_1.

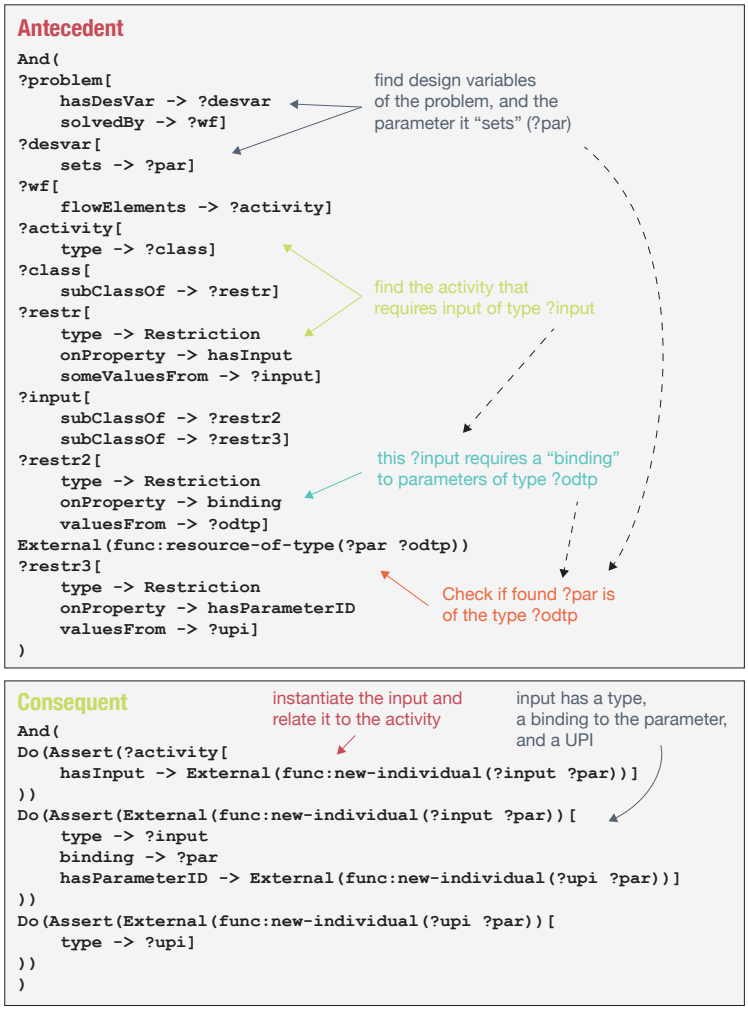


Figure 7.16: This rule instantiates inputs for the activity *"Interact with MMG"* for every design variable in a problem.

varies per case. It is the input to the parametric workflow. This is visualised in the N^2 diagram in Figure 7.14, where x_1 and y_1 are the parameters of the workflow.

Although it is impossible to define in advance what these parameters are, it is possible to describe what **kind** of parameter it should be. For the KBE Centric Workflow it is known that x_1 and y_1 are related to product attributes (see Figure 7.15). To be more specific:

- x_1 is a vector of attributes that are *set* by design variables.
- y_1 is a vector of attributes that are responses of the problem, i.e. parameters in objective function and constraint formulas.

Thus, instead of providing x_1 and y_1, it is possible to capture in rules what it should be. For example, the rule shown in Figure 7.16 instantiates inputs for the activity *"Interact with MMG"* for every design variable in a design problem. The remaining set of rules, used to instantiate the KBE Centric Workflow, are listed below. The code for these rules, written in RIF, is included in Appendix D.

- **R1**
- **R3-R9**
- **R11-R13**
- **R19-R26**

The KBE Centric Workflow class is modelled by the knowledge engineer in a similar way as the HLAs (see Figure 7.17). Since the activities of the workflow show similarities to the lower level activities of the HLA, the same techniques can be used to instantiate these activities.

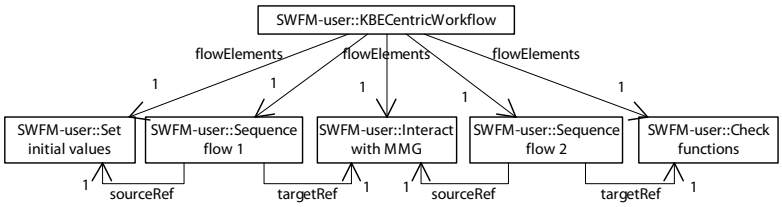


Figure 7.17: UML class diagram of the KBE Centric Workflow.

Then, in the modelling process, the design engineer defines the design problem (instantiates problem, design variables, and constraints), adds relations to the product attributes, and models formulas in Ruler (basically what is shown in Figure 7.10). The design engineer does not have to model the workflow. Instead, the RE will instantiate the entire workflow and stores it in the KB. From this moment, the workflow can be generated automatically in Optimus.

Reconfigurability

The KBE Centric Workflow allows for the design engineer to adjust the problem without modifying the workflow. Following the same example as in the previous use case:

1. The *height* parameter is removed as a design variable. It is now fixed to its initial value of 3.

2. A new *weight constraint* is added to the problem: $\rho V < 5000$ (where $\rho = 1200$)

In the ontology this is modelled as follows (see Figure 7.18):

- Remove the design variable for height from the problem instance.
- Add a weight constraint instance to the problem.
- Provide the weight constraint equation in Ruler (see Figure 7.19).

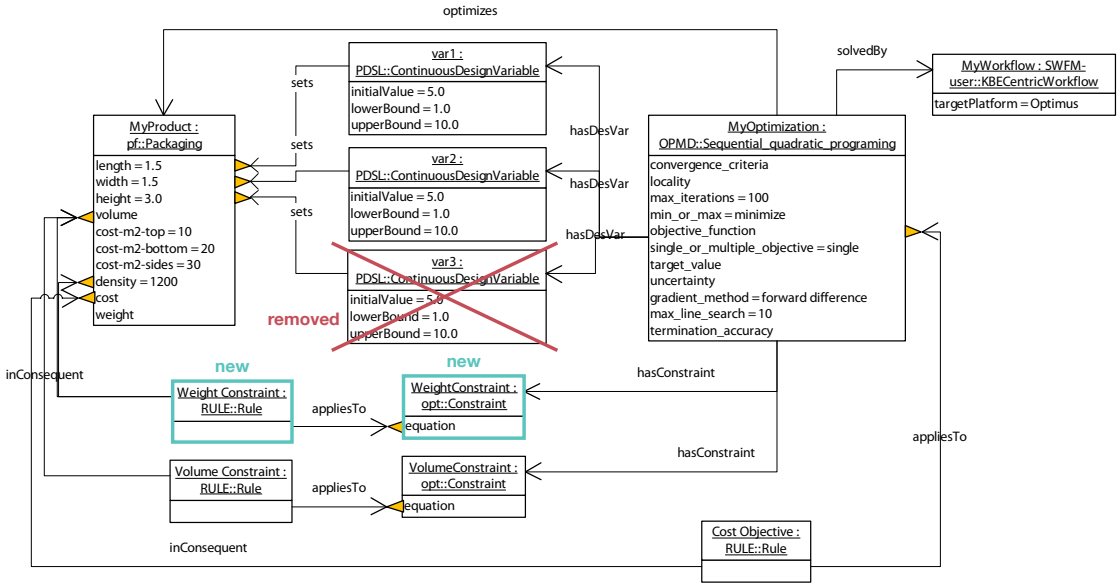
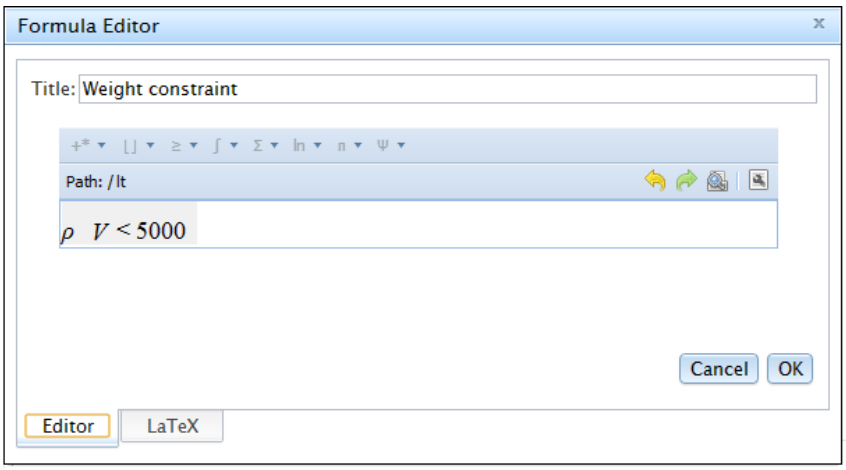


Figure 7.18: Class diagram of the new optimisation problem. The figure shows which objects are new and which are removed.



MathML string

```
<math xmlns="http://www.w3.org/1998/Math/MathML" title="Weight constraint">
<apply><lt/><apply><times/>
<ci definitionURL="http://www.lr.tudelft.nl/KBE/product/formal/packaging#packaging.density">ρ</ci>
<ci definitionURL="http://www.lr.tudelft.nl/KBE/product/formal/packaging#packaging.volume">V</ci>
</apply><cn>5000</cn></apply></math>
```

Figure 7.19: Screenshot of the new weight constraint in the MathML editor.

Name	Value	Formula		Low		High
ObjectiveFunction	0	$cost$	☐	0	☐	0
VolumeConstraint	0	$volume$	☑	4.0	☑	4.0
WeightConstraint	0	($density$ * $volume$)	☐	0	☑	5000.0

Figure 7.20: The new weight constraint in Optimus.

The interpreter translates the MathML string into a formula in Optimus. The result is the new weight constraint in Figure 7.20.

The RE instantiates a new workflow according to the changes made to the problem definition. This workflow is then generated in Optimus. The results are discussed in the next section.

7.1.3 Results

The interpreter translates the knowledge from the KB into an Optimus workflow. This workflow generation process is fully automated, hence no human activities are involved. The result is the workflow in Figure 7.21, which is identical to the generated workflow from the previous use case. Note that the different naming scheme of the activities in this workflow is the result of instantiating these activities by the RE. The function *new-individual* from Figure 7.9 aggregates two URIs to create a new URI.

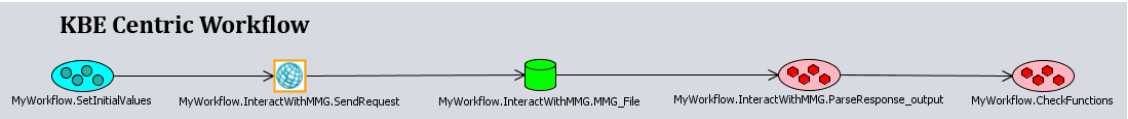


Figure 7.21: The generated workflow is identical to the workflow created with the KBE–PIDO coupling.

Because the optimisation is executed in Optimus, and thus independent of the performance of the KB–PIDO coupling, the optimisation results are exactly the same as in the previous use case. Table 7.4 shows the optimisation results for the standard problem and the new problem (with fixed height and additional weight constraint).

Table 7.4: The results of the optimisation problem are exactly the same as in the previous use case (under the same conditions).

Parameter	Initial value	Optimised value (Old problem)	Optimised value (New problem)
length	1	2.0002	1.1558
width	1	2.0002	1.1547
height	3	1.0	3
cost	390	360.0492	455.9405
volume	3	4.0008	4.0040
weight	3600	4800.96	4804.8098

7.2 Discussion

In the end, both implementations (KBE–PIDO and KB–PIDO coupling) have a shared objective, which is to reduce complexity of WFM and design optimisation. This use case demonstrated a methodological approach to WFM for a relatively simple optimisation problem. Working out a simple problem is easier for understanding the methodology. It has been a good example for demonstrating:

- how to apply the methodology step-by-step.
- how process knowledge is linked with product knowledge and rules.
- how the KBE Centric Workflow is implemented in a KB.
- that the interpreter can successfully translate process knowledge into a simulation workflow.

The goal to generate the workflow from knowledge in the KB has been achieved. There is no coding required, and the system is no longer dependent on the KBE platform for generating and executing the workflow. An additional step in the methodology explained how the workflow is parameterised to replicate the KBE Centric Workflow in the KB. With this solution, the design engineer can be more involved in the modelling process. Finally, the step-by-step approach clarifies where the knowledge in the workflow comes from.

Limitations and Future Direction

Despite being a success, there are already some thoughts for improvement. The first, and most important improvement, is to build a unified web interface for modelling diagrams and the ontology. In order to raise acceptance among engineers, it is crucial to provide intuitive interfaces to the system. And once Optimus extends its API, it will be possible to provide postprocessing capabilities in the web interface as well.
Ultimately, it should be integrated with the product and rule modelling interfaces. Then, together with the centralised KB, the integration framework will turn into a shared platform that enhances collaboration between design teams, domain experts, and IT engineers.
A second improvement, is to implement round-tripping. This feature has not been developed yet because of other priorities, however it would be valuable if the system could automatically return results and process knowledge to the KB.
The KBE Centric Workflow proved to be a powerful tool for optimising KBE models, however it is limited to performing analyses in the KBE platform only. Future research should focus on developing more high level parametric processes for common analyses, such as FEA or CFD. This will make design optimisation more accessible to non-SWFM experts.

Incorporating these changes would truly be a leap forward towards the next generation design system, featuring an intelligent, web-based engineering environment. But before this research is concluded, the same methodological approach is applied to an MDO problem.

Chapter 8

Use Case 3: MDO Workflows

The methodology has been demonstrated for a relatively simple engineering problem in the previous use case. The challenge now is to take a similar approach for an MDO problem.
In realistic engineering problems, where MDO is involved, there is an increased risk that a workflow becomes a black box, where no one understands how it works and what it does except for the person who built it. An example where this has occurred, is the workflow that has been used for the Pegasus project. Without a methodological approach, it is very difficult to reuse the knowledge in this workflow. And reusing knowledge is a key element in reducing development time and cost.
In this use case the workflow is rebuilt with the methodology to make clear what it does. Meanwhile, with the knowledge that is captured, it is possible to reach a higher level of abstraction and to implement intelligent features that will simplify SWFM. The main goal though, is to demonstrate that the methodology is effective not only for simple cases but also for more complex workflows.

The chapter begins with background information about the Pegasus project (Section 8.1). This section explains what issues may occur in ad hoc solutions and presents an improved solution. Section 8.2 goes through the methodology, showing the steps for this design problem. Eventually, the simulation workflow is automatically generated in Optimus (Section 8.2.3). The chapter ends with a discussion in Section 8.3.
Also here, there is the assumption that:

- the product has already been modelled in the KB or round-tripped from KBE code (if code is already written).
- the KBE code for the mould has been written or automatically generated (if product model exists in the KB).
- and that the mould is instantiated before the workflow is executed.

8.1 Problem Description

The Pegasus project is a European collaboration under the FP6 framework between the automotive industry and various research institutes (Pegasus, 2010). The objective was to develop an automated design framework for optimising the design of thermoplastic injection moulds. The final result of the project is a framework that changed the mould optimisation process using KBE and PIDO technologies. The main interest in this research is the workflow that has been created in Optimus (see Figure 8.1). More details about the project, including design challenges in thermoplastic injection moulding, are described by Janse (2013).

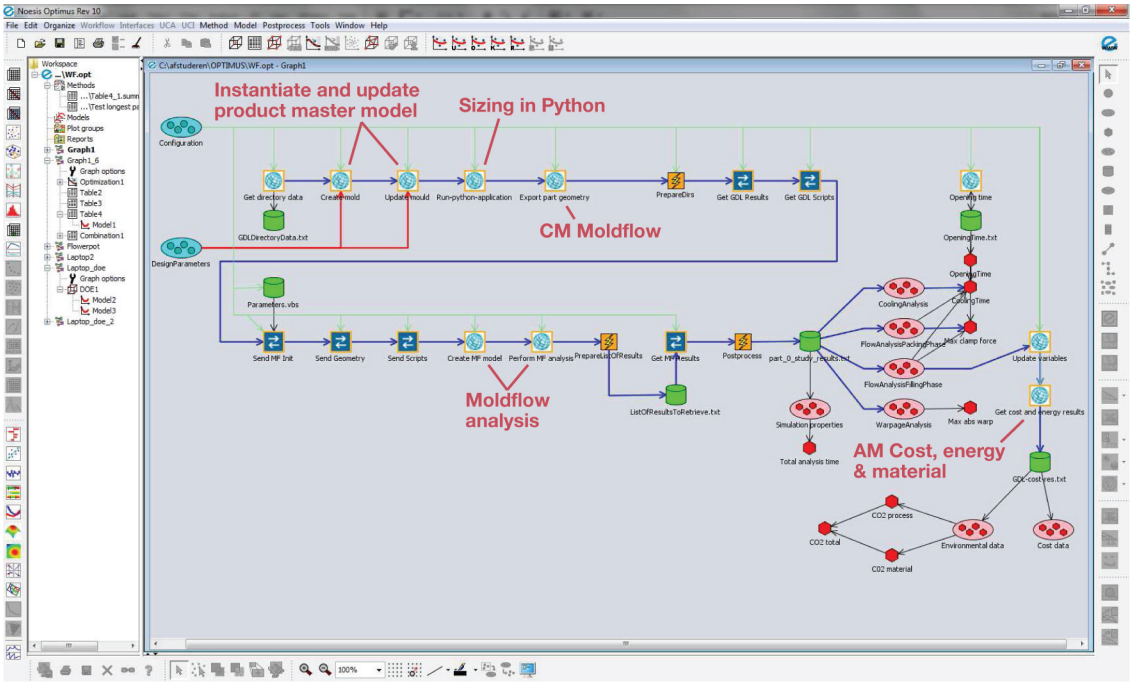


Figure 8.1: The original Pegasus workflow, created in the Optimus GUI (source: Janse, 2013).

The workflow covers multiple disciplines, using the following set of simulation tools. All these tools are set up as web services.

- A KBE application for mould design has been created in GenDL (product master model).
- Sizing the mould is done by a tool written in Python.
- The commercial tool Moldflow (Autodesk, 2012) is used for flow analysis.
- Various Analysis Modules (AM) are created in GenDL for analysing cost, energy usage, and material usage. AMs are similar to CMs, with the difference that AMs produce results through analysis.
- A Life Cycle Assessment (LCA) tool is used to calculate the environmental impact of the moulding process.

The Pegasus workflow has been designed and modelled in the Optimus GUI as an ad hoc solution. It exemplifies what problems may occur when a workflow is designed without

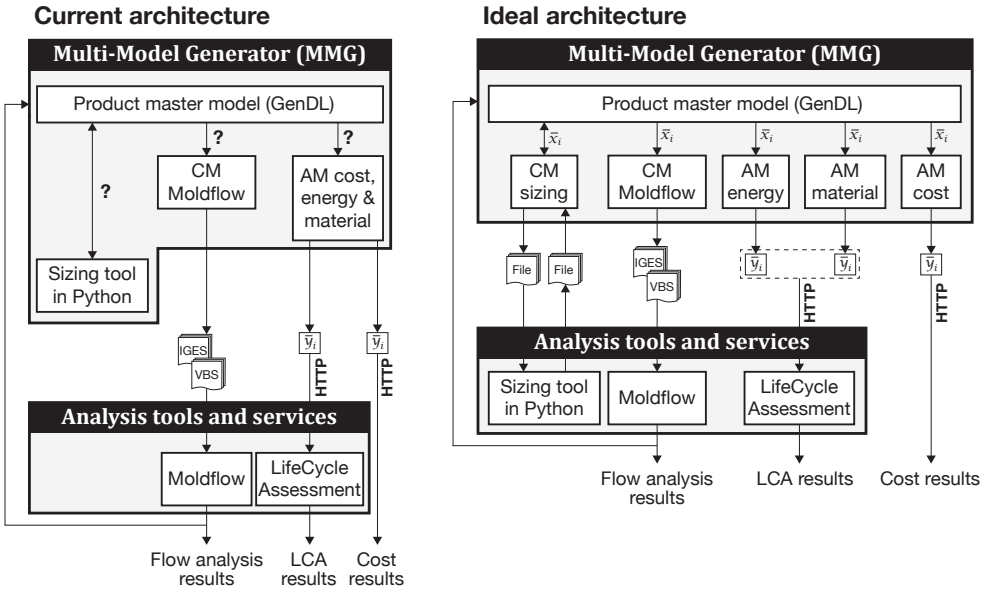


Figure 8.2: The diagram on the left shows the current architecture of the framework and the diagram on the right what it should have been in the ideal case.

following a methodology. Below are four main points that highlight undesired characteristics of this workflow.

1. It is unclear what the exact disciplines are in this workflow. The main problem in this workflow is that the product definition has become one big black box where multiple disciplines are integrated into one tool. In terms of modularity, the framework scores very low. Figure 8.2 shows the software architecture of the current framework. There are three issues regarding modularity and transparency:

1. The sizing tool is fully integrated into the MMG.
2. It is unknown what information flows from the product master model to the various modules (see question marks in the left diagram).
3. The workflow gives the impression that the AMs for cost, energy usage, and material usage are built into one module. There is only one activity that sends a request to the MMG to retrieve values from these modules.

2. The inputs and outputs of activities, and dependencies between disciplines are not made explicit. While looking at the workflow in Figure 8.1, it is not clear how the data flows between activities. When this information is lacking, then:

- How is the correct order of activities determined?
- What are dependencies between inputs and outputs?
- Which inputs are required/optional, and which are fixed/variable?

Even when inputs and outputs are handled internally by the tool itself, it is still worthwhile to make this explicit. It may not be necessary for modelling the workflow, but it increases transparency and eventually the understanding of the tools.

3. The sizing tool is tightly coupled to the product model. In the current framework, the sizing tool is completely integrated into the MMG. To understand what

the tool does and to maintain modularity it should be separated instead. Figure 8.2 shows a better solution on the right, where a CM for sizing is added and the sizing tool has become an analysis tool outside the MMG.

4. The flow analysis tool should be decoupled from the product model. Even as a service, Moldflow is depending on GenDL to provide the paths to the input files for running the analysis. Also, the HTTP request that is sent to run Moldflow requires the product name. These couplings to the product model are not necessary, and complicate the reuse of Moldflow as an engineering service.

In an ideal architecture (see right diagram in Figure 8.2), all tools are decoupled and also the internal data flow of the MMG (presented as x) is specified. An even better solution is to break down the product definition into several disciplines:

- Mould configuration
- Cooling system definition
- Feeding system definition
- Demolding mechanism definition

This would be the ideal case, if the Pegasus framework was built from the start with a methodological approach. Since these tools are already built, it would take too much time to redo it all afterwards. Therefore, this use case is executed under the assumption that the sizing tool remains integrated into the MMG.

High-Level Engineering Workflow
The ideal, high-level engineering workflow reduces the amount of work for the workflow modeller and is modelled with engineering knowledge instead of IT knowledge. The result is shown in Figure 8.3. The number of activities has been reduced to 11, compared to the 33 activities in the generated workflow (see end result in Figure 8.19). The required amount of work and knowledge for modelling the workflow has decreased significantly. This is the goal. However, this is currently not achieved yet. It is a topic for future research where modelling high-level workflows with HESs is the focus. In this use case, the MDO workflow is modelled with HLAs, which is described in the remainder of this chapter.

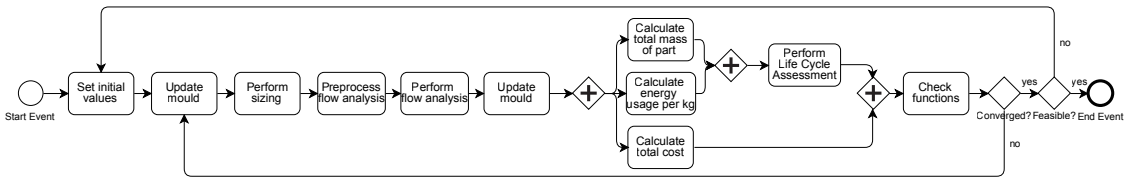


Figure 8.3: This high-level engineering workflow would be the ideal workflow for this MDO problem.

8.2 Implementation

The primary function of a methodology is to provide methods and guidelines to support people in their work. Thus far, the methodology has been applied to a relatively simple packaging design problem. This section demonstrates the five steps of the methodology for a more complex problem.

8.2.1 Informal Model

The Optimus workflow, which is the end result, has already been shown in the problem description. An existing workflow was selected because for development it is easier to follow a bottom-up approach. But since the methodology follows a top-down approach, it is assumed that the modelling process starts with a clean sheet. Nothing is known about the workflow, thus it begins with defining the problem statement.

Step 1: Define problem statement
The problem in the Pegasus project can be stated as follows:

> *"Design a **mould** for the manufacturing of laptop bezels with **minimum cost and environmental impact**. The produced parts have to fulfil quality requirements on **maximum warpage**. Moreover, the injection moulding machine has a **maximum** limit on its **clamp force**."*

From this statement it can be concluded that:

- The product is a mould.
- It is a minimisation problem.
- There are two objectives: total cost and total CO2 emission.
- There are two inequality constraints: maximum on warpage and a maximum on clamp force that is allowed.

With this much information, the design engineer can proceed to the next step.

Step 2: Structure the problem and disciplines
In step 2, the design engineer takes the information from the problem statement and structures it. The design engineer can visualise the relationships between problem and disciplines using the N^2 notation.

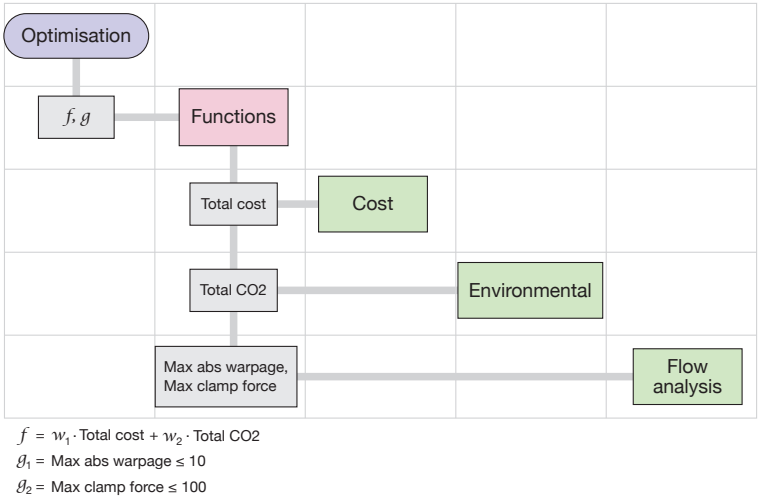


Figure 8.4: First snapshot of the problem and disciplines visualised in an N^2 diagram.

Although there is no fixed order in an N^2 diagram, it is convenient to put *optimisation* at the top-left corner, with *functions* directly below it (see Figure 8.4). The disciplines

are aligned further down the diagonal in a random order. Finally, the responses can also be added to the diagram, with the right connections to their respective discipline and to functions.

Creating this diagram is an iterative process and cannot be completed right away. It is handed to the IT engineer, who will model the activities in step 3.

Step 3: Model the activities

For every discipline that delivers results, the IT engineer needs to model the activities that will produce these results. For example, the discipline environmental analysis delivers the output *total* CO_2 emission. Then, the IT engineer needs to reason backwards what sequence of activities will produce the total CO_2 emission (see Figure 8.5 (a)). At some point, the IT engineer ends up at the beginning of the sequence where inputs are required from another discipline than environmental analysis. This is when this step is completed for this single discipline.

The same procedure can be repeated for the other disciplines (see Figure 8.5 (b) and (c)). When this is done, the process goes back to step 2 to update the N^2 diagram with the latest findings.

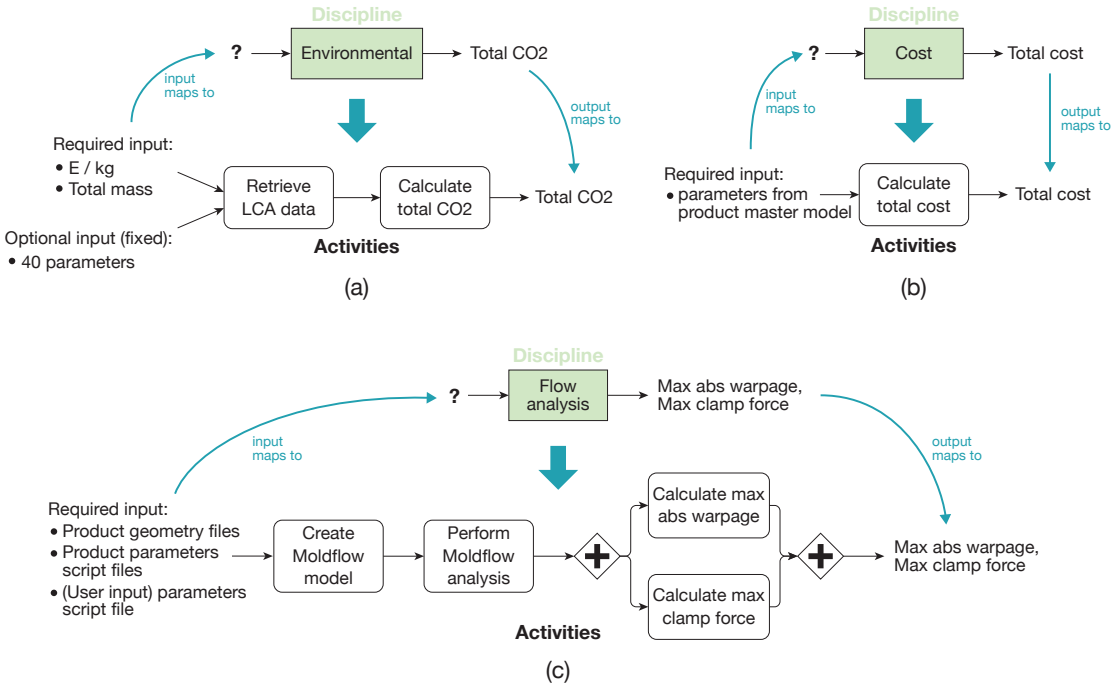


Figure 8.5: Activities are modelled according to the output that is required from the disciplinary analysis. This is done for environmental analysis (a), cost analysis (b), and flow analysis (c).

Step 2, iteration 2: Update the N^2 diagram

In this second iteration, the N^2 diagram is updated with the inputs found in the previous step. Several of these inputs have been identified as outputs from new disciplines: energy usage, material usage, and preprocessing flow analysis. These are then added to the diagram, as shown in Figure 8.6.

Completing the diagrams

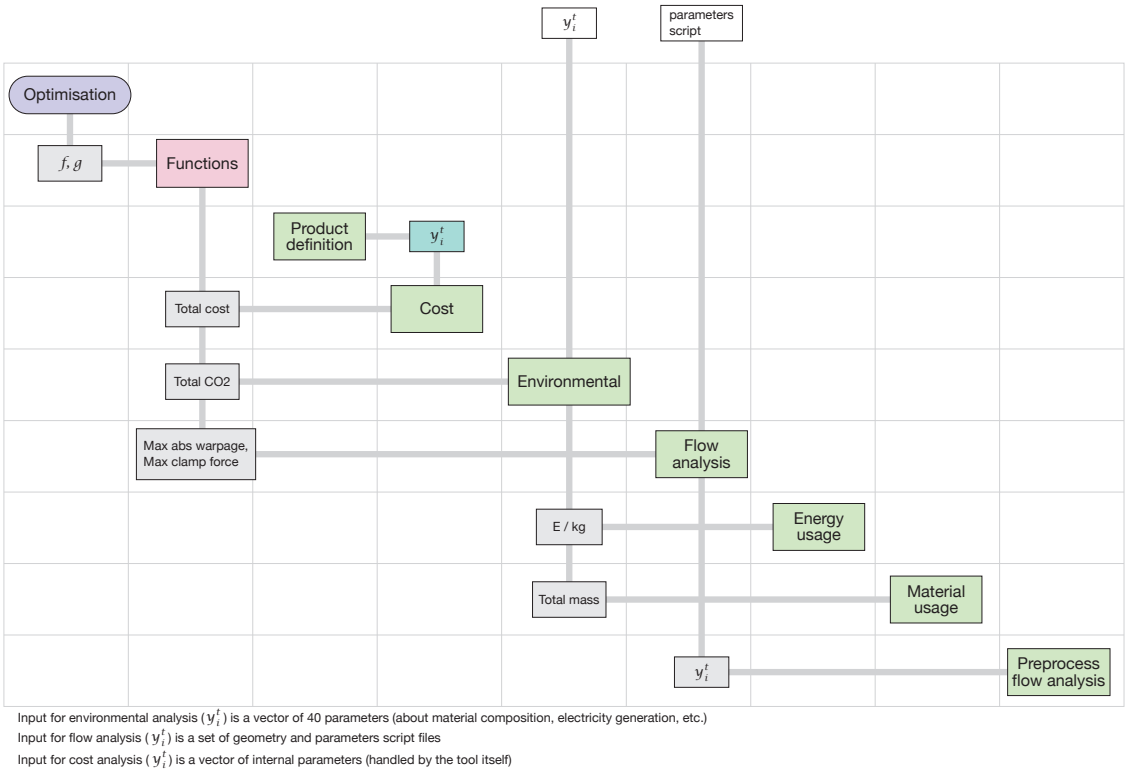


Figure 8.6: Completing the N^2 diagram is an iterative process. This figure shows the diagram at the second iteration.

From here on, the modelling process continues to step 3, where the same method is used to model the activities. Step 2 and 3 are repeated until the N^2 diagram and workflow are completed. The final diagrams are shown in Figure 8.7.

Explaining the N^2 diagram: In this final diagram, the disciplines are reordered for convenience. External inputs and outputs are placed at the edge of the diagram. Other noticeable elements are the coupling variables with a blue colour. These are new to the notation and have been introduced to indicate that these inputs and outputs are managed internally by the MMG. The data flow is not visible in the workflow, but it is still useful to make it explicit. The vectors in the diagram are briefly explained below.

- x_0 is a vector of the design variables.
- y_1 is a vector of fixed input parameters to product definition.
- y_1^t is a vector of output parameters from the product master model.
- y_2^t is a vector containing the results of the sizing step.
- y_3^t is a set of product geometry and parameter script files for flow analysis.
- y_8 is a vector of fixed input parameters for the Life Cycle Analysis.

The numbers in the diagram indicate the flow through the disciplines, going from low to high. These are determined by the flow of inputs and outputs between disciplines. A discipline cannot deliver outputs until it receives the required inputs from another discipline (or the user).
There was one difficulty in determining the right order of disciplines. As the diagram shows, there are many disciplines dependent on the product definition. The problem is

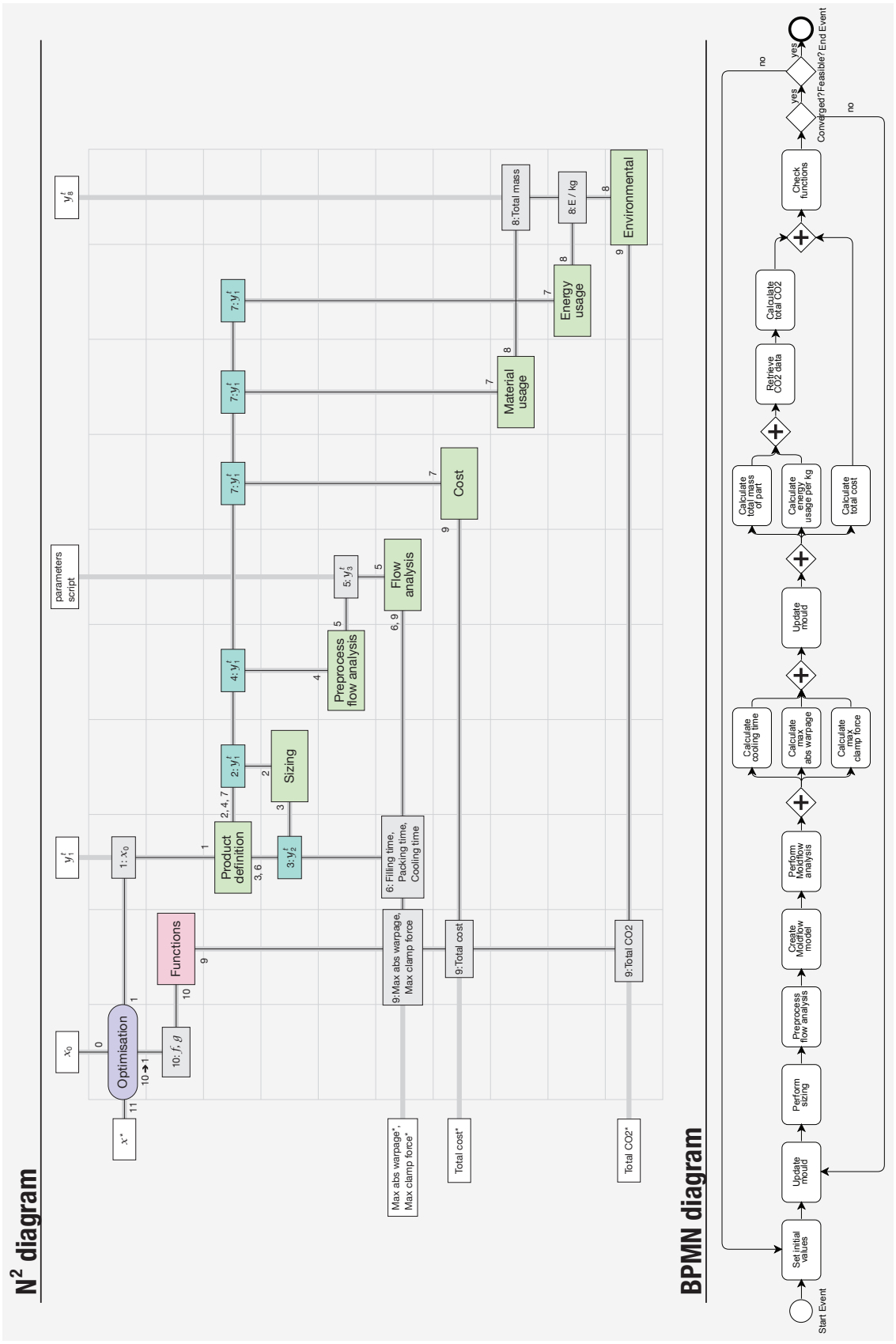


Figure 8.7: Completed N² and BPMN diagram for the Pegasus project.

that the product model exists at any time in the workflow, but with different values. Technically, it is possible to retrieve values from the product model at any time, because most parameters have default values. However, this will not give the correct results. Take for example the cost, which can be asked ahead of the flow analysis, but this will not be the desired answer. There is a link missing, namely that cost responses can only be requested after the mould has been updated with results from the flow analysis (updated with cooling, packing, and filling times). There are two ways to include this crucial bit of information in the definition: (1) using states, and (2) define dependencies between inputs and outputs.

A product model can be in a certain **state**. Its state changes when the product model is modified. By using states, it is possible to define when a discipline may request values from the product model. In a workflow this can be modelled as pre- and postconditions of an activity.

The other method is to define dependencies between inputs and outputs. This dependency is a relationship that states that an output can only be given on the condition that the required inputs have been provided. For instance, the inputs for cost analysis are outputs of product definition (see Figure 8.7). By stating that these outputs depend on cooling, packing, and filling times, it can be reasoned that a cost analysis can only be done after flow analysis.

Both methods work equally well, although the solution using dependencies may be leaner (in the ontology). For this use case, the first method using states has been applied, which are shown in Table 8.1.

Table 8.1: Pre- and postconditions that have been used in this workflow.

Activity	Precondition	Postcondition
Update mould (1)	-	Mould with initial values
Perform sizing	Mould with initial values	Sized mould
Preprocess flow analysis	Sized mould	-
Update mould (2)	-	Mould updated with flow analysis results
Calculate total mass of part	Mould updated with flow analysis results	-
Calculate energy usage per kg	Mould updated with flow analysis results	-
Calculate total cost	Mould updated with flow analysis results	-

Explaining the BPMN diagram: The final workflow is built by chaining activities that have been modelled in step 3. The order is defined by the numbers in the N^2 diagram. There is one flaw in the workflow right after the "Perform sizing" activity. According to the flow in the N^2 diagram, another "Update mould" activity is expected between the sizing and preprocess activities. But because the original application has been designed as a tightly coupled system, updating the mould is done internally by the tool and not from the workflow. Once both diagrams have been completed, the process proceeds to step 4.

Step 4: Fill in ICARE PDT-forms
Step 4 is the knowledge acquisition phase, where the knowledge engineer captures knowledge regarding the optimisation problem, disciplines, activities, and tools in *ICARE PDT-forms*. An example of a filled D-form is given in Figure 8.8.

MOKA2 form:	Discipline				
Name	Flow analysis				
Reference	D4				
Objective	Perform a simulation of the plastic injection moulding process.				
Description	A simulation of the plastic injection moulding process provides accurate results for analysing the mould design, plastic part design, and optimising the injection moulding process.				
Input requirements/preconditions	-				
Context, information, validity	Thermoplastic injection moulding				
Problem involved	P1				
Activities involved	A4.1, A4.2, A4.3, A4.4, A4.5				
Entities involved	E1				
Parameters					
Name	Product geometry files	Product parameter script files	User input parameter script file	Cooling time	Pac
Description	A set of IGES files describing the mould and part geometry. The number of files varies, depending on the mould configuration.	A set of VBS files containing additional information about the product model	An input script file containing parameters, such as project name, filenames, and simulation settings.	The cooling time is the time that is required for the molten plastic to cool down and solidify	
Input/Output	Input	Input	Input	Output	Out
Variable/Fixed	Variable	Variable	Variable		
Parameter Value	-	-	-	-	
Default value (process)	-	-	-	-	
Default value (tool)	-	-	-	-	
Source	Preprocess flow analysis discipline (D3)	Preprocess flow analysis discipline (D3)	User input		
Management					
Author	P. Chan				
Date	06-02-2013				
Version number	1				
Status	In Progress				
Modification					
Information origin	F.J. Janse: workflow and report				

Figure 8.8: Example of a filled D-form capturing details of the *Flow analysis* discipline.

8.2.2 Formal Model

This section contains only one step, which is to formalise process knowledge. It explains how process knowledge can be modelled, so that an interpreter can generate the workflow automatically. The focus is on modelling High-Level Activities (HLA).

Step 5: Formalise process knowledge
Instead of showing how to model all activities, similar activities are grouped and only one of each group is explained. The other activities in the same group are modelled in a similar way. There is a total of seven groups.

Group 1
The first group of activities updates the product master model, thus setting new values to variables. There are two tasks in this group:

- Update mould (1)
- Update mould (2)

Figure 8.9 shows the target for this activity. It is mapped onto a single activity in Optimus, an HTTP request UCA. The activity sends the request that is displayed next to it.

The UML diagram shows how it is modelled in the ontology. The product model, related to the activity through the `hasModel` property, provides its name to *iid* (instance ID) in the query string. Input parameters are mapped onto *variables* and *values*. The keyword

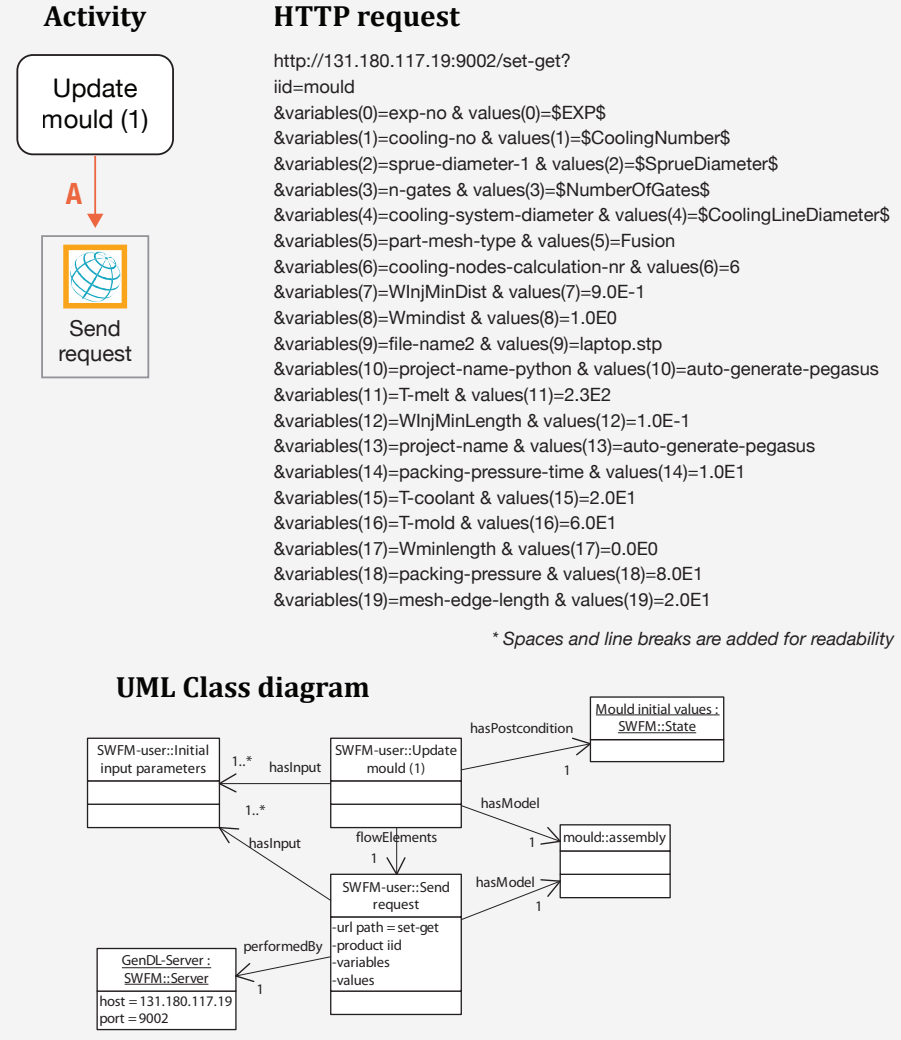


Figure 8.9: Figure showing how the HLA *"Update mould (1)"* is modelled in Optimus (Group 1). The UML class diagram shows how it is modelled in the ontology.

variables contains the names of the product attributes. Then, if the input is variable, its value will appear with dollar signs in the query string. For example, the sprue diameter is a variable input parameter (see *values(2)* in the HTTP request). Otherwise, a fixed value is inserted in the query string.
All inputs of the activity *"Update mould (1)"* are provided in Table 8.2. Rules R11 and R13-R15 perform the mapping for this group of activities (see Appendix D).

Group 2

The second group consists of activities that trigger a CM in the MMG.

- Perform sizing
- Preprocess flow analysis

"Perform sizing" is a relatively simple activity in the workflow because most actions are handled internally by the tool. This is an example where tools are tightly coupled. As a result, the sizing tool is executed by asking a *response* from the product model. The

Table 8.2: Inputs for *"Update mould (1)"* (Group 1)

Input	Variable/ Fixed	Required/ Optional	Parameter Value	Default Value	Binding
Cooling number	Variable	Required	-	-	assembly.cooling-no
Number of gates	Variable	Required	-	-	assembly.n-gates
Sprue diameter	Variable	Required	-	-	assembly.sprue-diameter-1
Cooling line diameter	Variable	Required	-	-	assembly.cooling-system-diameter
Experiment number	Variable	Required	-	-	assembly.exp-no
Mould temperature	Fixed	Required	60	-	assembly.T-mold
Melt temperature	Fixed	Required	230	-	assembly.T-melt
Coolant temperature	Fixed	Required	20	-	assembly.T-coolant
Packing pressure	Fixed	Required	80	-	assembly.packing-pressure
Packing pressure time	Fixed	Required	10	-	assembly.packing-pressure-time
Part file name	Fixed	Required	laptop.stp	-	assembly.file-name2
Project name	Fixed	Required	auto-generate-pegasus	-	assembly.project-name
Part mesh type	Fixed	Required	Fusion	-	assembly.part-mesh-type
Mesh edge length	Fixed	Optional	20	4.5 (tool default)	assembly.mesh-edge-length
Project name sizing	Fixed	Optional	auto-generate-pegasus	same as project-name (tool default)	assembly.project-name-python
Weighting min distance	Fixed	Required	1.0	-	assembly.Wmindist
Weighting min length	Fixed	Required	0	-	assembly.Wminlength
Weighting injection min distance	Fixed	Required	0.9	-	assembly.WInjMinDist
Weighting injection min length	Fixed	Required	0.1	-	assembly.WInjMinLength
Cooling nodes calculation nr	Fixed	Required	6	-	assembly.cooling-nodes-calculation-nr
Include feed system in cooling analysis	Fixed	Optional	-	0 (tool default)	assembly.include-feed-sys-in-cooling-analysis

request, *"responses(0)=size-sub-systems"*, is fixed and modelled as a restriction to the class (see Figure 8.10).

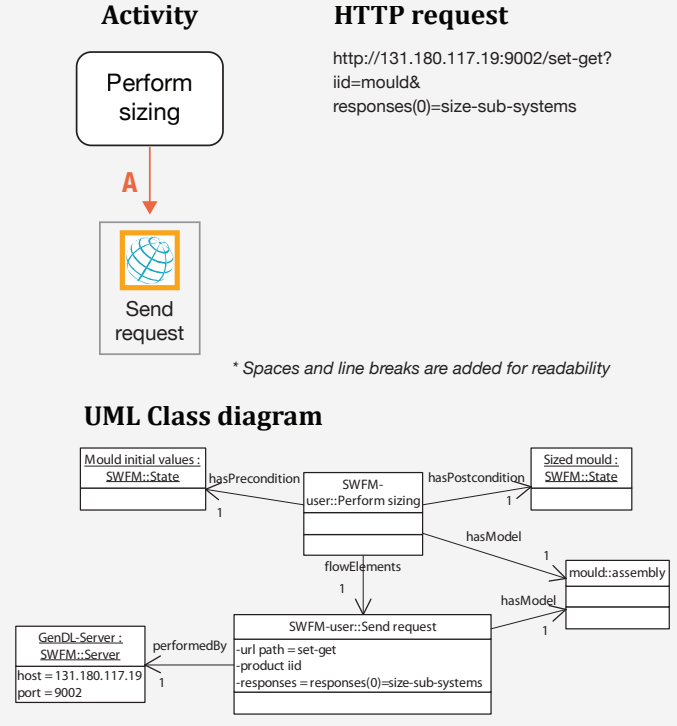


Figure 8.10: Figure showing how the HLA *"Perform sizing"* is modelled in Optimus (Group 2). The UML class diagram shows how it is modelled in the ontology.

Group 3

Activities in group three trigger AMs in the MMG and return one or more responses. All three activities have a precondition to indicate that these responses can only be asked after the flow analysis has been performed.

- Calculate energy usage per kg
- Calculate total mass of part
- Calculate total cost

AMs only have internal input, which are not visible in the workflow. Therefore, these three activities only have outputs (see Figure 8.11). Table 8.4 shows all available outputs for *"Calculate energy usage"*. The outputs for the other two HLAs are included in Appendix F.
Selecting the outputs from the list determines which values are retrieved. Thus, the selected outputs appear in the HTTP request (under *responses*) and as file extraction rules in the file parser (done by rules R10 and R12). In this example, the activity *"Calculate energy usage per kg"* has only one output, which is *Total energy usage per kg*.

Group 4

Group four consists of two Moldflow activities.

- Create Moldflow model
- Perform Moldflow analysis

Table 8.3: Outputs for *"Calculate energy usage"* (Group 3)

Output	File parameter settings					
	label	word nr	occurrence	row offset	start pos	end pos
Total energy usage per kg of material used	E-total-molding-per-kg	-	-	-	-	-
Total energy usage for manufacturing cooling system	E-total-cooling-system-manufacturing	-	-	-	-	-
Total energy usage	E-total	-	-	-	-	-
Total energy usage of moulding process	E-total-molding	-	-	-	-	-

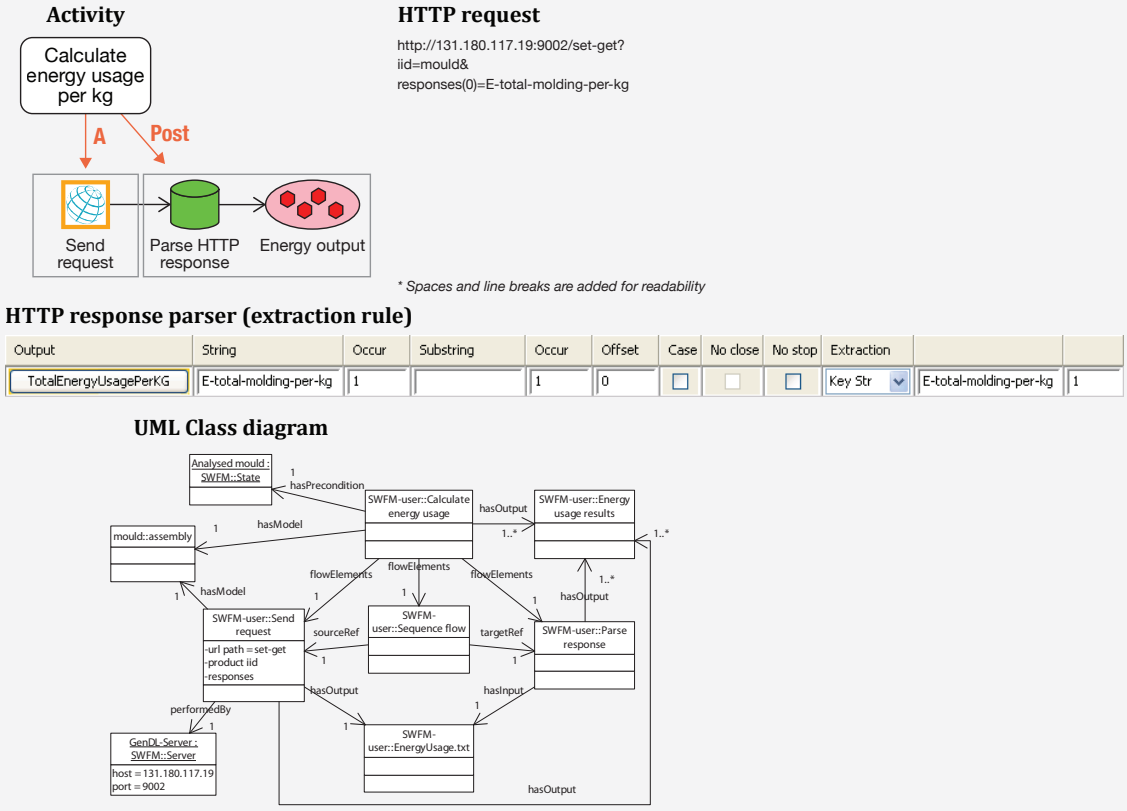


Figure 8.11: Figure showing how the HLA *"Calculate energy usage"* is modelled in Optimus (Group 3). The UML class diagram shows how it is modelled in the ontology.

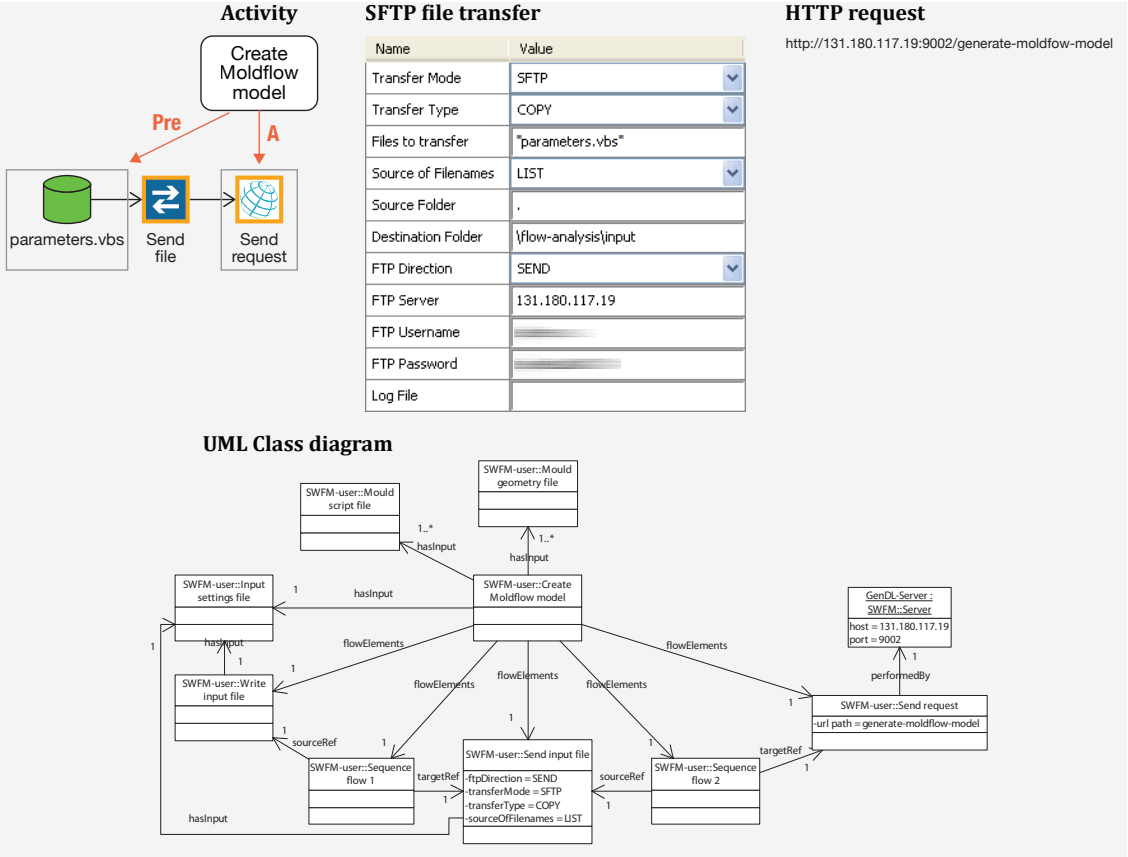


Figure 8.12: Figure showing how the HLA *"Create Moldflow model"* is modelled in Optimus (Group 4). The UML class diagram shows how it is modelled in the ontology.

The activity *"Create Moldflow model"* has a preprocessing and analysis step (see Figure 8.12). The preprocessor writes an input file based on a template. Here, it is an input settings file (parameters.vbs) containing parameters that are required for creating the Moldflow model. Optimus inserts values in the template that change during every iteration, such as the experiment number. Input files are then saved in the Optimus project folder of the current run. Therefore, it is always required to transfer the file to the input folder of the engineering service.

Figure 8.12 shows the required settings for this file transfer. Since Moldflow is set up as a web service, the file is transferred over SFTP. Other settings that have fixed values (COPY, LIST, and SEND) are modelled as restrictions to the class. The values for the server and destination folder depends on the server that hosts the service and the input folder of the service. These values are not fixed, therefore a rule is used to provide these values (rule R28).

The analysis step sends an HTTP request to create the Moldflow model. Previously, this request required the product name to trigger this action. But because the required inputs to create the model are only files (geometry and script files), it was not necessary to have a coupling to the product model. Therefore, it is now a fixed request without a query string.

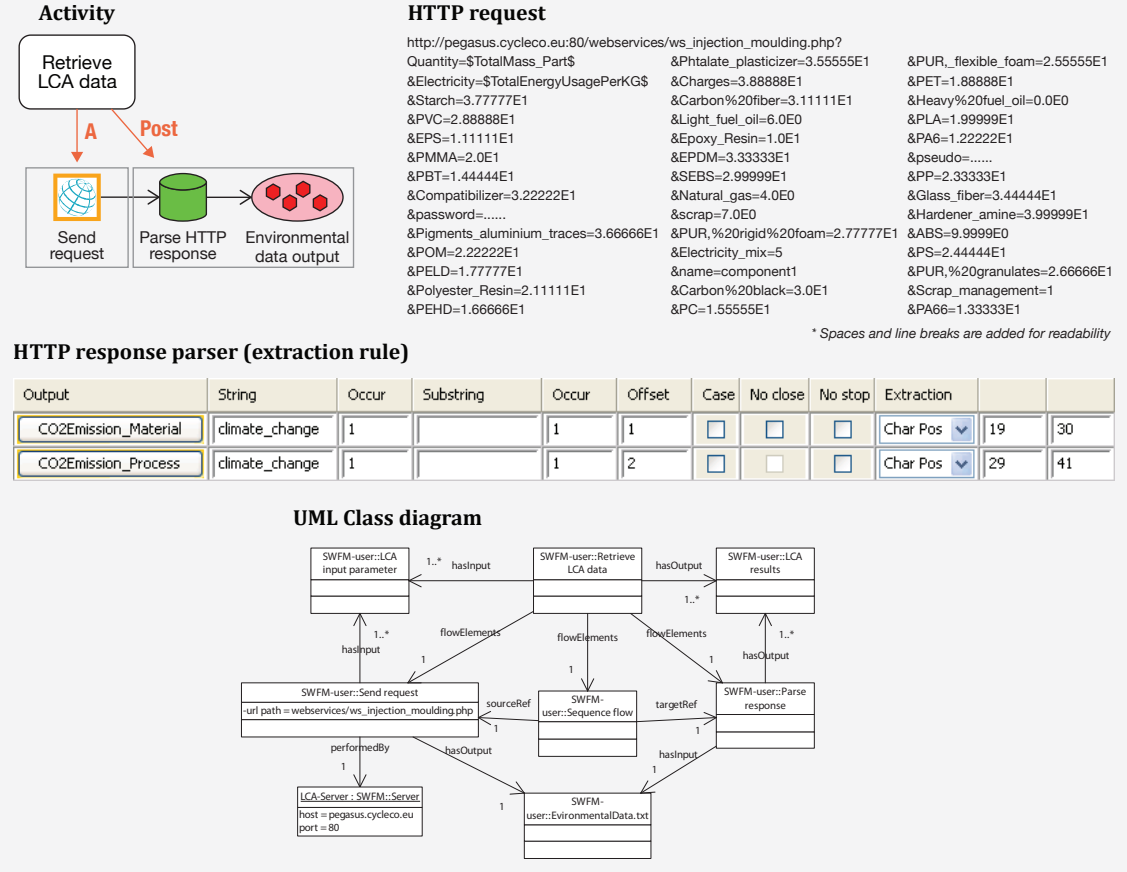


Figure 8.13: Figure showing how the HLA *"Retrieve LCA data"* is modelled in Optimus (Group 5). The UML class diagram shows how it is modelled in the ontology.

Group 5

The fifth group consists of one activity that performs an environmental analysis (LCA web service).

- Retrieve CO2 data

This activity sends a request (with 42 input parameters in total) to retrieve environmental data (see Figure 8.13). Most parameters are fixed, except for two.

- *Quantity*, which is the total mass of the part (from the discipline material usage).
- *Electricity*, which is the total energy usage per kg (from the discipline energy usage).

The meaning of all input parameters is provided in a table in Appendix F. The fixed parameters is a mix of required and optional inputs.
Regardless of the input, the service always returns four values. These are listed in Table 8.4.

The query string for the service, is modelled differently from the other activities. Where the inputs of the other activities always map onto *variables* (and *values*), here the query string is fixed. Thus, every parameter has a fixed keyword.

Creating a solution, without hardcoding the values in the query string, requires a link between the query parameter and the corresponding input parameter. The solution is

Table 8.4: Outputs for *"Retrieve LCA data"* (Group 5)

Output	File parameter settings					
	label	word nr	occurrence	row offset	start pos	end pos
CO2 emission material	climate_change	-	-	-	19	30
CO2 emission process	climate_change	-	-	-	29	41
Resources material	resources	-	-	-	19	30
Resources process	resources	-	-	-	29	41

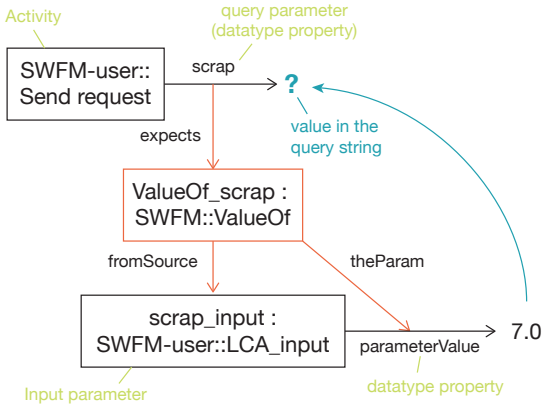


Figure 8.14: Query parameters are linked to input parameters according to this construct (Group 5).

shown in Figure 8.14. It is inspired by bindings in OWL-S, where in this case a resource (*ValueOf*) 'tells' the query parameter which value to take from which source. A rule is used to actually do the mapping (rules R16-R18).

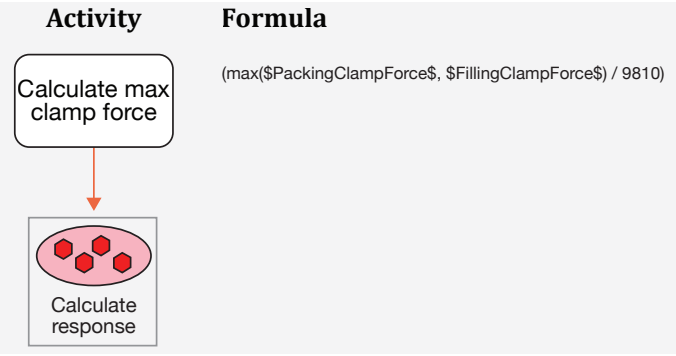


Figure 8.15: Figure showing how the HLA *"Calculate max clamp force"* is modelled in Optimus (Group 6). The UML class diagram shows how it is modelled in the ontology.

Group 6

The activities in the sixth group calculate a response in Optimus by evaluating a formula.

- Calculate cooling time
- Calculate max abs warpage
- Calculate max clamp force
- Calculate total CO2

These activities are modelled as instances of *Calculate Response* (see Figure 8.15). Inputs are parameters that appear in the formula (*Packing* and *Filling clamp force*) and the only output is the parameter that is calculated (*Max clamp force*). Therefore, the inputs and output are always the same. The formula is modelled in the MathML editor in Ruler and translated to Optimus' mathematics syntax by the interpreter (see Figure 8.16).

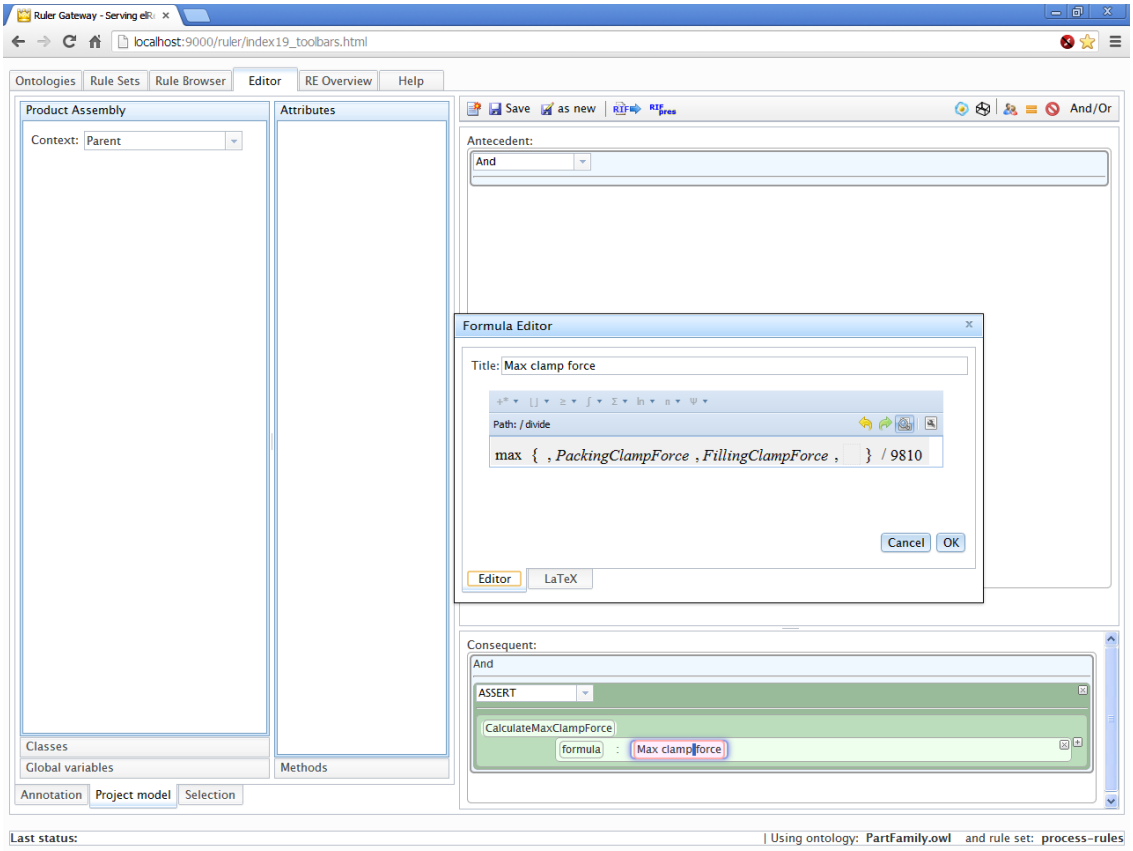


Figure 8.16: Screenshot of Ruler where the formula for calculating the max clamp force is modelled (Group 6).

Group 7

The final group of activities are always required in a workflow.

- Set initial values
- Check functions

Set initial values defines the design variables according to the values provided in Table 8.5. The end result, as it appears in Optimus, is shown in Figure 8.17.

Table 8.5: Inputs for *"Set initial values"*

Design Variable	Type	Initial value	Lower bound	Upper bound	Step size	Sets
var1	Continuous	1.5	0.5	3.0	-	assembly.sprue-diameter-1
var2	Discrete numeric uniform	1	1	3	1	assembly.cooling-no
var3	Discrete numeric uniform	1	1	4	1	assembly.n-gates
var4	Continuous	11.5	8	15	-	assembly.cooling-system-diameter

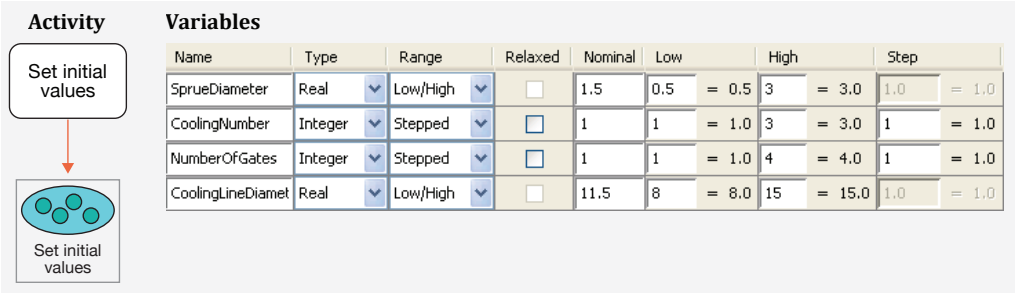


Figure 8.17: Figure showing the end result for the activity *"Set initial values"*.

For the activity *"Check functions"* there are two constraints:

Max abs warpage constraint: $MaxAbsWarpage \leq 10[mm]$
Max clamp force constraint: $MaxClampForce \leq 100000[kg]$

Note that the objective was not to have accurate constraints. A sufficiently large number is chosen to ensure that results would be valid. Furthermore, the objective function is lacking here. This design problem is a multi-objective optimisation, which has not been implemented in the Python wrapper yet. Setting up the problem is done manually in the Optimus GUI. Then, the end result for *"Check functions"* is shown in Figure 8.18.

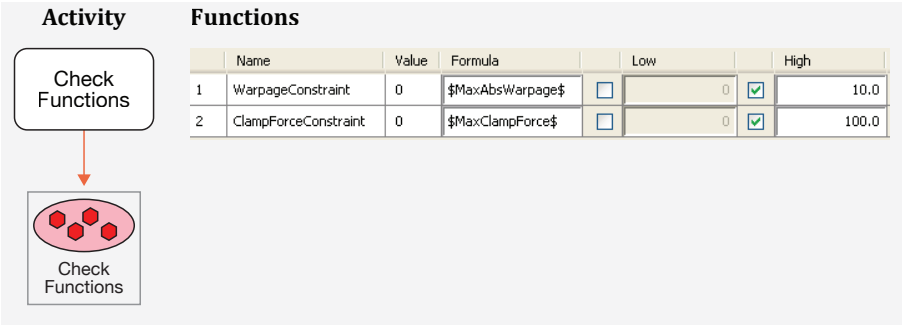


Figure 8.18: Figure showing the end result for the activity *"Check functions"*.

A total of 21 rules have been applied in this use case (see list below) The code for these rules, written in RIF, is included in Appendix D.

- **R1-R18**
- **R27-R29**

The workflow generation process is exactly the same as in the previous use case. After the RE has inferred new facts and stored it in the triple store, the interpreter will initiate workflow generation. The final result is discussed in the next section.

8.2.3 Results

The generated workflow is presented in two formats (see Figure 8.19). The original format (at the top) shows the layout as it has been determined by Graphviz. It is a long sequence of activities and impossible to present on paper. Therefore, the activities are reordered in the second format to present the workflow in a more readable layout. The activities are laid out conforming to the DEE framework.

The outlines with rounded corners shows the domain of a HLA. HLAs group activities, that are always needed in this order, under one higher level activity. As a result, the user instantiates each HLA as one activity, and the computer reuses knowledge to generate the actual activities in Optimus.
Another nice feature is demonstrated in this use case. The four SFTP transfer activities, highlighted in the middle of Figure 8.19, are created entirely by the computer. In the original workflow, there is a direct flow from *"Preprocess flow analysis"* to *"Create Moldflow model"*. A rule (R27) has recognised that *"Preprocess flow analysis"* produces files that are required by *"Create Moldflow model"*. Since both activities are performed by a remote server, the rule has inserted these SFTP activities to manage the file transfers. Meanwhile, Optimus will create a backup of every file that is transferred.

Before the workflow can be executed, the user must perform three activities:

1. Copy-paste the input settings file template for Moldflow to *parameters.vbs*.
2. Add an input variable that connects to *parameters.vbs*. Without the variable, Optimus will not recognise variables in the template.
3. Set up the problem (execution method) in the Optimus GUI.

At the time of development, it was not exactly clear how an input file template could be provided through the API. Since there is only one input file, it has been decided to do this manually.

The workflow has been executed successfully. The inputs for one of the experiments are given in Table 8.6, which lead to the responses presented in Table 8.7.

Table 8.6: Values for the design variables in one of the experiments.

Design Variable	Unit	Value
Sprue diameter	*mm*	3
Cooling number	–	3
Number of gates	–	4
Cooling line diameter	*mm*	8

During this experiment, Moldflow performed a flow analysis. The results are shown in Figures 8.20-8.22.
Figure 8.20 displays the mould model created by Moldflow, with the laptop bezel and three cooling systems (design variable *Cooling number*) per mould half. Then, Figure 8.21 shows

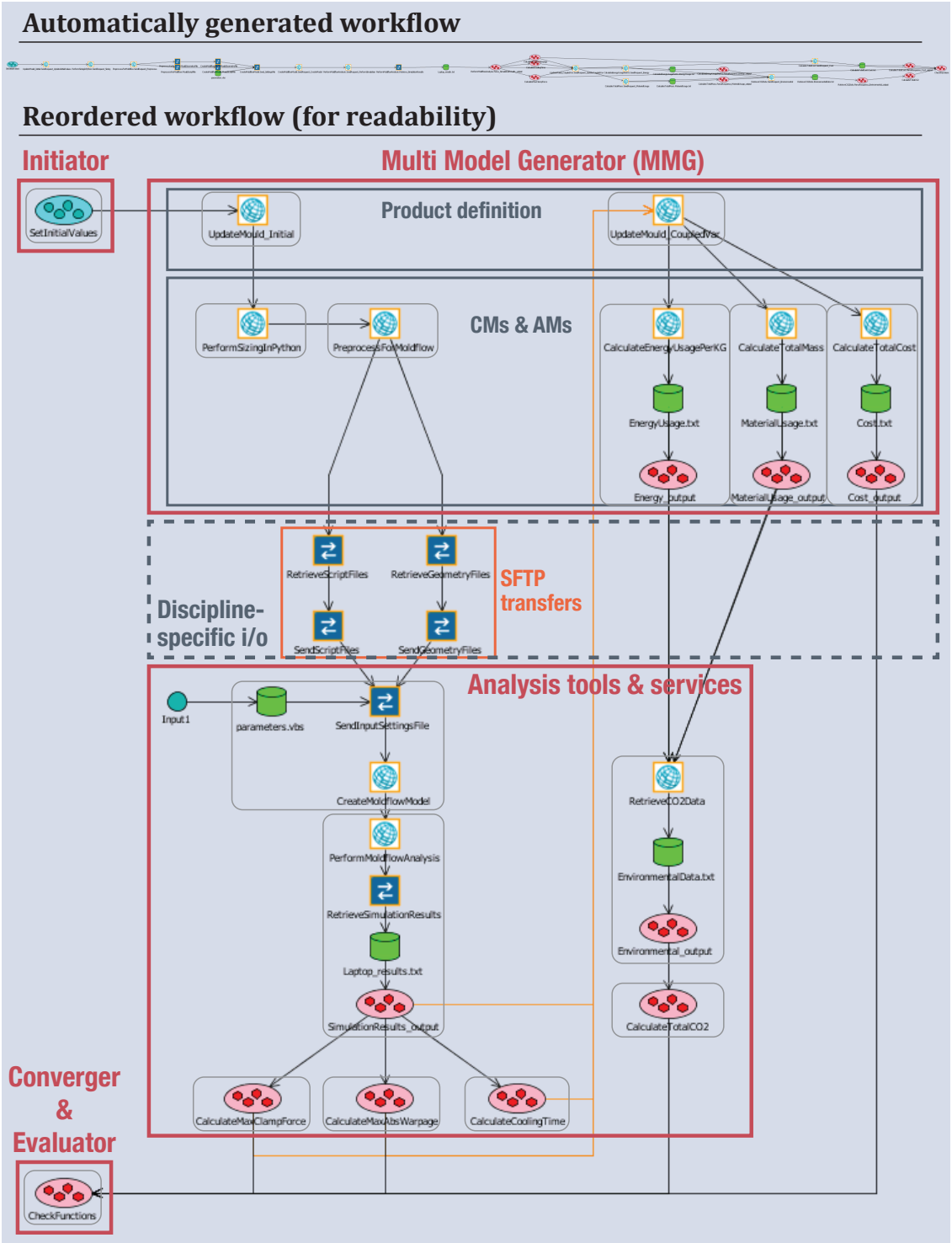


Figure 8.19: The generated workflow is reordered into a more readable layout. The original workflow is shown at the top.

Table 8.7: Responses of one of the experiments.

Response	Unit	Value
Total cost	$	441158.99191905
Total energy usage per kg	kWh/kg	1.7326908329483
Total mass	kg	35163.823145303
CO_2 emission material	kg	936160.03926
CO_2 emission process	kg	80500.2194054
Total CO_2 emission	kg	1016660.2586654
Max displacement (X)	mm	0.10709
Max displacement (Y)	mm	0.41241
Max displacement (Z)	mm	0.3743
Min displacement (X)	mm	-0.27176
Min displacement (Y)	mm	-0.12624
Min displacement (Z)	mm	-0.70699
Cooling time	s	7.5761
Cycle time	s	9.5372
Filling time	s	1.9611
Packing time	s	11.9511
Filling clamp force	N	236380
Packing clamp force	N	766640
Max abs warpage	mm	0.70699
Max clamp force	kg	78148.827726809

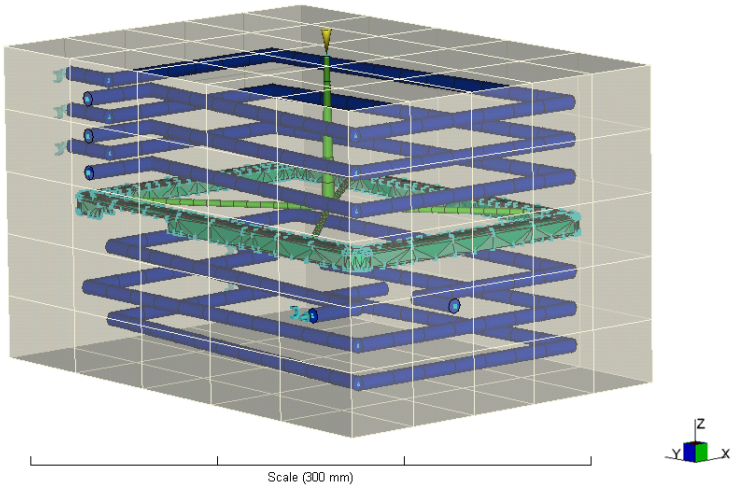


Figure 8.20: Screenshot taken from Moldflow. It shows the mould model, laptop bezel, and cooling systems.

the time that is needed to fill the mould cavity with plastic. The four gates (design variable *Number of gates*) are visible in this screenshot. The last result shows the temperature of the coolant inside the cooling system (see Figure 8.22).

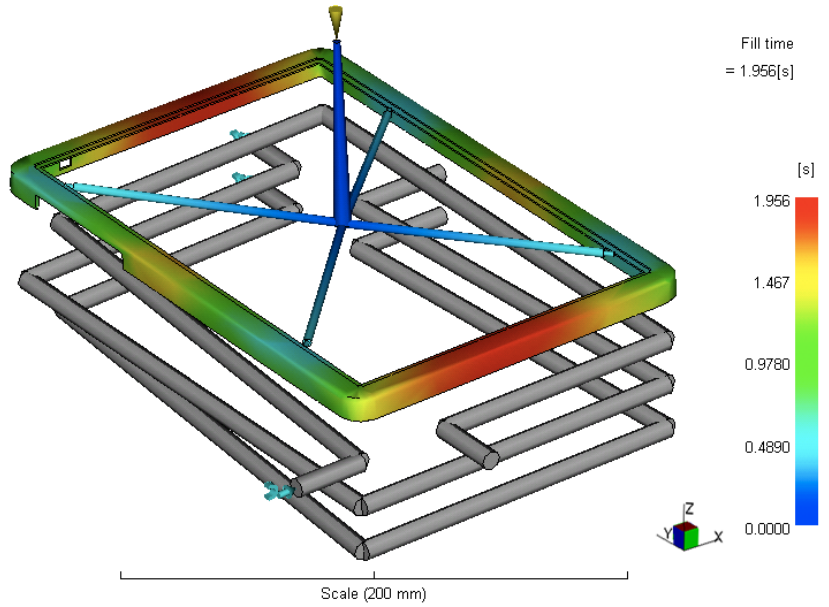


Figure 8.21: Screenshot taken from Moldflow. It shows the filling analysis results. Parts in red take the longest time to reach.

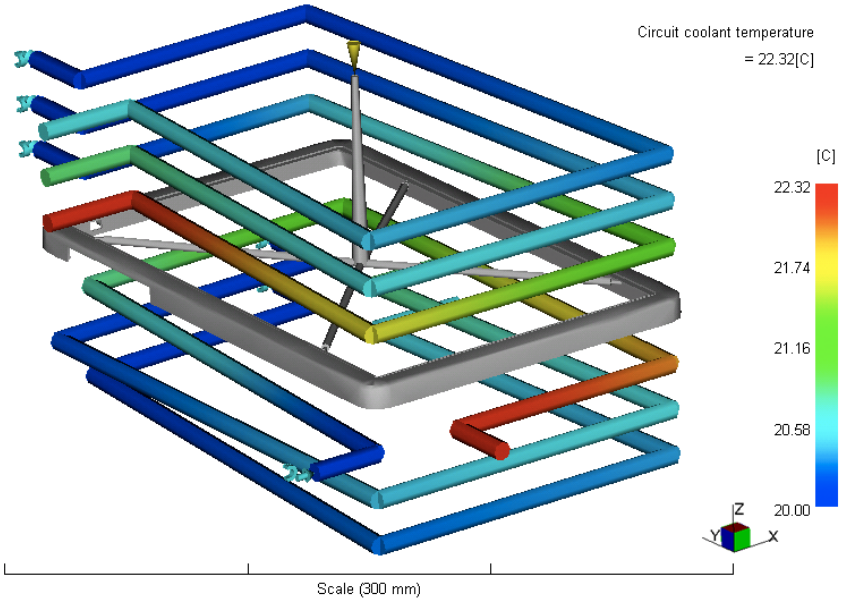


Figure 8.22: Screenshot taken from Moldflow. It shows the cooling analysis results.

8.3 Discussion

The past use case demonstrated the five steps of the methodology for modelling a simulation workflow for an MDO problem. It began with an analysis of an ad hoc solution, which illustrated that a methodology is in fact necessary for problems that involve this many disciplines. Without, there is an increased risk that the end result becomes incomprehensible. The consequence is that knowledge cannot be shared or reused.

Modelling the workflow step-by-step, in a top-down approach, ensures that knowledge in the workflow can be retraced to its owner. Moreover, the informal representation of the workflow makes it easier for non-SWFM experts to understand what the workflow does. Process knowledge is captured in the HLAs. This enables novice users, for instance a design engineer, to model the workflow at a higher abstraction level than the PIDO level. Furthermore, with the captured knowledge, it is possible to create intelligent solutions, such as the automatically derived SFTP file transfers. This reduces the amount of work for the user, and more importantly, the required expertise.

Altogether, the Pegasus workflow has been a good test that shows the potential of the methodology. It is a step closer to practical use, although more testing and development is required to make the system fully functional for the industry. The current WFMS is still a prototype, and there are many things to improve.

Limitations and Future Direction

The first limitation is that the functionality of the wrapper needs to be expanded. Some actions required manual input from the user, which would not be acceptable in an automated system. A future version of the Python wrapper should at least include the functions to generate an input file in Optimus, and expand the number of execution methods.

The end of Section 8.1 introduced a high-level engineering workflow for this MDO problem. That level of abstraction is achieved by modelling High-level Engineering Services (HES). In the Pegasus workflow, the Moldflow analysis could be modelled as a HES. The HES groups the Moldflow activities under one activity. Thus, the **Moldflow HES** consists of the HLAs defined in this use case (see Figure 8.23). The inputs are geometry and script files, and the outputs of the service are *cooling time*, *max abs warpage*, and *max clamp force*. The HES simplifies adding a Moldflow analysis to the workflow even further.

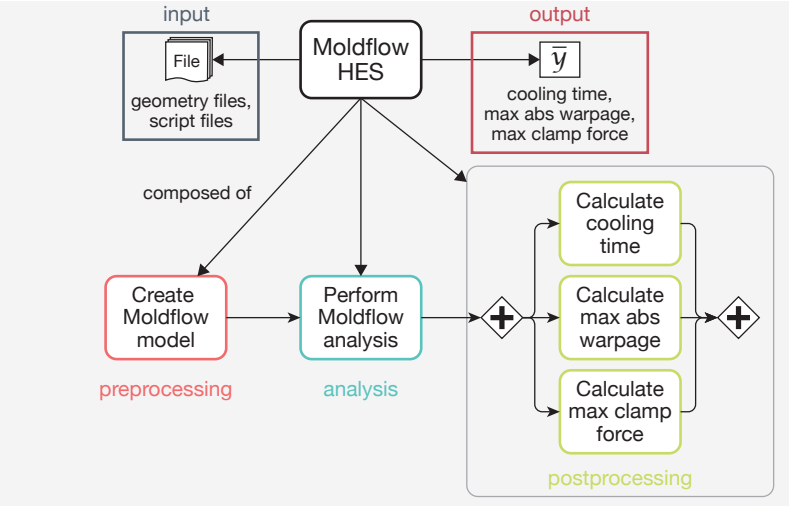


Figure 8.23: The transformation steps of the Moldflow HES are HLAs defined in this use case.

Chapter 9

Conclusions and Recommendations

This thesis presented a new methodology for WFM and an integration framework that couples a KB, KBE tool, and PIDO tool. Two use cases have demonstrated how the methodology is applied to model a simulation workflow in a top-down approach. Based on those experiences, several conclusions can be drawn. Additionally, recommendations are given for future developments that will further improve these solutions.

9.1 Conclusions

The conclusions are based on the research goals presented in Chapter 1.

- **Goal 1:** *Investigate how current Workflow Management Systems can be complemented with knowledge management technologies and find solutions for reducing the complexity of Workflow Management.*

Setting up simulation workflows in a WFMS requires a fair amount of (low-level) IT knowledge. Therefore, the first goal was to research and develop a solution for capturing and storing this knowledge in a way that it can be shared and reused.
This is where MOKA, a methodology for developing KBE applications, came into the picture. MOKA focuses on capturing and structuring knowledge to increase transparency of KBE applications. It introduced a methodological approach, with an Informal and Formal Model, to clarify what knowledge went into the applications. This would improve the ability to share and reuse knowledge in future projects.

However, MOKA could not be applied straight away to WFM, as the methodology fell short on modelling process knowledge. MOKA attributes this to the complexity and variety of design processes. Therefore, this research narrowed down the domain to simulation workflows and design optimisation.

- **Goal 2:** *Extend the MOKA methodology with an ontology for storing process knowledge (on an informal and formal level) in a platform-independent and transparent model, as to maximise the potential for sharing and reusing this knowledge.*

A new methodology has been developed, named **MOKA 2**. For this thesis, the focus was on the process side. The main contribution is the step-by-step instructions for modelling simulation workflows following a top-down approach. As a result, transparency of simulation workflows has increased because of the link to the source of the knowledge. Furthermore, the standardised approach facilitates sharing and reuse of knowledge in other projects.

At the informal level, new **Problem-**, **Discipline-**, and **Tool-forms** have been developed to extend ICARE. The original ICARE-forms do not capture design problem and simulation workflow knowledge. Moreover, **BPMN** and an **N^2 notation** have been introduced to visualise the relationships between these forms. At the formal level, a **formal process ontology** has been developed, with relations to the product and rule ontologies. The objective was to reduce complexity of SWFM, so that less expertise is required to model simulation workflows and perform design optimisation. This is achieved by bringing SWFM to a higher abstraction level, with **High-Level Activities (HLA)** and **High-level Engineering Services (HES)** as reusable building blocks. The path to a higher level led to the design of a parametric high-level workflow, the **KBE Centric Workflow**, which enables users to optimise KBE product models without actually modelling a workflow. The KBE Centric Workflow contains the knowledge to build the workflow based on information from the problem definition.

- ◇ **Goal 3:** *Build and demonstrate an advanced Workflow Management System in a number of use cases varying in scope and complexity, that captures and reuses process knowledge for automatic generation of simulation workflows.*
- ◇ **Goal 4:** *Integrate the Workflow Management System into an engineering design framework that supports Knowledge Based Engineering applications and Multi-disciplinary Design Optimisation.*

Workflows are automatically generated and executed by Optimus, which runs as a web service in the background. The Optimus web service is a cornerstone of the **integration framework**: a triangle between a *Knowledge Base* (KB), *product* (KBE), and *process* (PIDO) tools. Within this framework, two couplings have been developed: a *KB–PIDO* and a *KBE–PIDO coupling*.
The **KB–PIDO coupling** uses Model Driven Software Engineering (MDSE) techniques to generate simulation workflows automatically from the knowledge in the Formal Model. This avoids the step of manually translating knowledge to an executable workflow, to save time and to make it more accessible to non-experts.
The **KBE–PIDO coupling** on the other hand, is a solution for programmers who want to quickly perform design optimisation. The new **problem** and **workflow Domain-Specific Languages (DSL)** provide the tools for programmers to model design problems and simulation workflows inside the KBE environment.

At the end, it can be concluded that the research goals have been fulfilled for the most part. This research demonstrated in a number of use cases how:

- knowledge management technologies are applied to WFMS.
- process knowledge is captured and structured on an informal and formal level.
- simulation workflows are automatically generated.
- a KB, product, and process tools are integrated into a framework.

The methodology also presented new solutions to reduce complexity of SWFM. However, more extensive research is required to measure the extent to which it has been reduced. This has not been quantified in this research. Future work can improve the current accomplishment by following the recommendations in the next section.

9.2 Recommendations

Based on the work performed, several recommendations are given for further improvements that can be made.

Web interface. The first, and likely the most important improvement, is to build a working web interface for modelling diagrams and the ontology. To make it more accessible and to raise acceptance among engineers, it is essential to provide intuitive interfaces to the system. Ideally, it should be integrated with the product and rule modelling interfaces to create a unified interface to the design system.

Round-tripping. Round-tripping is another valuable feature that would enrich the system. If the system can automatically return simulation results and process knowledge from workflows (i.e. reverse automatic workflow generation) to the KB, then users will always have access to the latest data.

More use cases. Currently, the methodology has been applied to a relatively simple optimisation problem and a more complex MDO problem. However, a few use cases are not sufficient to verify the tools and methods. The methodology and framework should be tested in more use cases, and preferably in an actual engineering environment. This will most likely generate feedback from engineers, which can be used to further improve the methodology.

Postprocessing. With the current prototype, postprocessing is only possible from within the Optimus GUI. But because it is an essential feature for design systems, it is recommended to include postprocessing capabilities in the web interface.

Extend optimisation algorithms. As the system has been built as a prototype, it does not include all available algorithms that Optimus provides. The Python wrapper needs to be extended with the remaining algorithms.

More high-level parametric processes. High-level parametric processes, such as the KBE Centric Workflow, proved to be a valuable contribution to making design optimisation more accessible. Future research should focus on developing more high-level processes, as these processes simplify WFM with intelligent solutions.

Automate the formalisation process. The next step for the formalisation process is to use rules or an algorithm that formalises knowledge automatically.

References

Aasman, J. (2012, May 24). *Recorded Webcast: How to Use Graph Databases to Analyze Relationships, Risks and Business Opportunities.* Retrieved from http://www.franz.com/ps/services/conferences_seminars/semantic_technologies_v30.lhtml (accessed 23 October 2012)

Airbus. (n.d.). *The Airbus concept plane.* Retrieved from http://www.airbus.com/innovation/future-by-airbus/concept-planes/the-airbus-concept-plane/ (accessed 19 October 2012)

Alexandrov, N. M., & Lewis, R. M. (2004). Reconfigurability in MDO problem synthesis, part 1. *10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*.

Ammar-Khodja, S., Perry, N., & Bernard, A. (2008, March). Processing Knowledge to Support Knowledge-based Engineering System Specification. *Concurrent Engineering, Vol. 16 No. 1*, pp. 89–101.

Autodesk. (2012). *Autodesk Simulation Moldflow.* Retrieved from http://usa.autodesk.com/moldflow/ (accessed 11 November 2012)

Baader, F., Calvanese, D., McGuinness, D., Nardi, D., & Patel-Schneider, P. (2003). *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press.

Bārzdiņš, J., Bārzdiņš, G., Čerāns, K., Liepiņš, R., & Sproģis, A. (2013). *OWLGrEd.* Retrieved from http://owlgred.lumii.lv/ (accessed 8 March 2013)

Bazilevs, Y., Calo, V. M., Cottrell, J. A., Evans, J. A., Hughes, T. J. R., Lipton, S., ... Sederberg, T. W. (2009, March 14). Isogeometric analysis using T-splines. *Computer methods in applied mechanics and engineering*.

Bechhofer, S. (2003, December 03). *OWL Reasoning Examples.* Retrieved from http://owl.man.ac.uk/2003/why/latest/ (accessed 24 October 2012)

Bendiken, A. (2010, April 22). *How RDF Databases Differ from Other NoSQL Solutions.* Retrieved from http://blog.datagraph.org/2010/04/rdf-nosql-diff (accessed 23 October 2012)

Bermell-Garcia, P., & Fan, I. S. (2002, October). A KBE System for the Design of Wind Tunnel Models Using Reusable Knowledge Components. *International Congress on Project Engineering*.

Berners-Lee, T. (1998, October 14). *Why RDF model is different from the XML model.* Retrieved from http://www.w3.org/DesignIssues/RDF-XML (accessed 2 December 2012)

Bowcutt, K. G. (2003, December 15–19). A Perspective on the Future of Aerospace Vehicle Design. *12th AIAA International Space Planes and Hypersonic Systems and Technologies*.

Bowcutt, K. G., Kuruvila, G., Grandine, T. A., Hogan, T. A., & Cramer, E. J. (2008, April). Advancements in Multidisciplinary Design Optimisation Applied to Hypersonic Vehicles to Achieve Closure. *15th AIAA International Space Planes and Hypersonic Systems and Technologies Conference*.

Brimble, R., & Sellini, F. (2000). *The MOKA Modelling Language.* Retrieved from http://web1-eng.coventry.ac.uk/moka/Documents/Papers/ekaw2000.pdf (accessed 31 October 2012)

Brunnermeier, S. B., & Martin, S. A. (1999, March). *Interoperability Cost Analysis of the U.S. Automotive Supply Chain* (Tech. Rep.). Gaithersburg, MD, USA: National Institute of Standards and Technology (NIST).

Center for eDesign. (n.d.). *e-Design Framework2.0.* Retrieved from http://edesign.ecs.umass.edu/?page_id=52 (accessed 29 October 2012)

Chapman, C. B., & Pinfold, M. (2001). The application of a knowledge based engineering approach to the rapid design and analysis of an automotive structure. *Advances in Engineering Software, Vol. 32 No. 12*, pp. 903–912.

Codd, E. F. (1970, June). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM, Vol. 13 No. 6*, pp. 377–387.

ComputerWeekly.com. (2002, March). *A layman's guide to Web services.* Retrieved from http://www.computerweekly.com/feature/A-laymans-guide-to-Web-services (accessed 24 October 2012)

Corallo, A., Laubacher, R., Margherita, A., & Turrisi, G. (2009). Enhancing product development through knowledge-based engineering (KBE): A case study in the aerospace industry. *Journal of Manufacturing Technology Management, Vol. 20 No. 8*, pp. 1070–1083.

Curran, R., Verhagen, W. J. C., van Tooren, M. J. L., & van der Laan, T. H. (2010). A multidisciplinary implementation methodology for knowledge based engineering: KNOMAD. *Expert Systems with Applications, Vol. 37*, pp. 7336–7350.

Dassault Systèmes. (n.d.). *Isight & the SIMULIA Execution Engine.* Retrieved from http://www.3ds.com/products/simulia/portfolio/isight-simulia-execution-engine/overview/ (accessed 26 October 2012)

de Wit, A. J., & van Keulen, F. (2010, April 12–15). Overview of Methods for Multi-Level and/or Multi-Disciplinary Optimization. *51st AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*.

Dumas, M., & ter Hofstede, A. H. M. (2001). UML Activity Diagrams as a Work ow Specification Language. *UML'2001*.

DuVander, A. (2011, March 8). *3,000 Web APIs: Trends From a Quickly Growing Directory.* Retrieved from http://blog.programmableweb.com/2011/03/08/3000-web-apis/ (accessed 24 October 2012)

Eifrem, E. (2012, September 26). *Graph Databases: The New Way to Access Super Fast Social Data.* Retrieved from http://mashable.com/2012/09/26/graph-databases/ (accessed 23 October 2012)

Epistemics. (2008, December 9). *PCPACK.* Retrieved from http://www.epistemics.co.uk/Notes/55-0-0.htm (accessed 28 October 2012)

Esteco. (n.d.). *modeFRONTIER: the multi-objective optimization and design environment.* Retrieved from http://www.esteco.com/home/mode_frontier/mode_frontier.html (accessed 26 October 2012)

Flager, F., & Haymaker, J. (2007). A Comparison of Multidisciplinary Design, Analysis and Optimization Processes in the Building Construction and Aerospace Industries. *24th International Conference on Information Technology in Construction*, pp. 625-630.

Francia, S. (2010, January). *SOAP vs. REST.* Retrieved from http://spf13.com/post/soap-vs-rest (Accessed: July 2011)

Franz Inc. (2011, August 16). *Franzs AllegroGraph Sets New Record - 1 Trillion RDF Triples.* Retrieved from http://www.franz.com/about/press_room/trillion-triples.lhtml (accessed 23 October 2012)

Franz Inc. (2012). *AllegroGraph®.* Retrieved from http://www.franz.com/agraph/allegrograph/ (accessed 3 November 2012)

Gallaher, M. P., O'Connor, A. C., Dettbarn, J. L., & Gilday, L. T. (2004, August). *Cost Analysis of Inadequate Interoperability in the U.S. Capital Facilities Industry* (Tech. Rep.). Gaithersburg, MD, USA: National Institute of Standards and Technology (NIST).

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* USA: Addison-Wesley.

Genworks International. (n.d.). *GenDL.* Retrieved from http://genworks.com/ (accessed 3 November 2012)

Graphviz. (n.d.). *Graphviz - Graph Visualization Software.* Retrieved from http://www.graphviz.org/ (accessed 16 November 2012)

Gruber, T. R. (1993). A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, *Vol. 5*(No. 2), pp. 199–220.

Hane, P. J. (2010, July 1). *Health Sites Use Semantic Technologies to Provide Better Results.* Retrieved from http://newsbreaks.infotoday.com/Spotlight/Health-Sites-Use-Semantic-Technologies-to-Provide-Better-Results-68135.asp (Accessed: August 2011)

Hellkamp, M. (2012). *Bottle: Python Web Framework.* Retrieved from http://bottlepy.org/docs/dev/ (accessed 6 March 2013)

Horridge, M. (2011, March 24). *A Practical Guide To Building OWL Ontologies Using Protégé 4 and CO-ODE Tools, Edition 1.3* (Guide). Manchester, UK: The University Of Manchester.

Janse, F. J. (2013). *The Development of a KBE System for Thermoplastic Injection Mold Design Automated Analysis and Optimization of a Thermoplastic Injection Mold.* Masters thesis, Delft University of Technology, Delft, The Netherlands. (to be published)

Kallas, S., Geoghegan-Quinn, M., Darecki, M., Edelstenne, C., Enders, T., Fernandez, E., & Hartman, P. (2011). *Flightpath 2050: Europe's Vision for Aviation.* Luxembourg: European Commission. (High Level Group on Aviation Research)

Ko, R. K. L., Lee, S. S. G., & Lee, E. W. (2009). Business process management (BPM) standards: a survey. *Business Process Management Journal, Vol. 15 No. 5*, pp. 744–791.

La Rocca, G. (2011). *Knowledge Based Engineering Techniques to Support Aircraft Design and Optimization.* Unpublished doctoral dissertation, Delft University of Technology, Delft, The Netherlands.

La Rocca, G. (2012, March 16). Knowledge based engineering: Between AI and CAD. Review of a language based technology to support engineering design. *Advanced Engineering Informatics*, *Vol. 26*, pp. 159–179.

Lambe, A. B., & Martins, J. R. R. A. (2011, June 13–17). A Unified Description of MDO Architectures. *9th World Congress on Structural and Multidisciplinary Optimization*.

Landman, Q. E. (2011). *Investigation of an Improved Safety Assessment Method in Preliminary Aircraft Design: Using Knowledge-Based Engineering and Bayesian Networks.* Masters thesis, Delft University of Technology, Delft, The Netherlands.

Laplante, P. A., Zhang, J., & Voas, J. (2008). Whats in a Name? Distinguishing between SaaS and SOA. *IT Professional*, *May/June 2008*, pp. 46–50.

Lovett, P. J., Ingram, A., & Bancroft, C. N. (2000). Knowledge-based engineering for SMEs - a methodology. *Journal of Materials Processing Technology, Vol. 107*, pp. 384–389.

Milton, N. R. (2007). *Knowledge Acquisition in Practice: A Step-by-step Guide*. London, UK: Springer.

NACFAM. (2001, February). *Exploiting E-Manufacturing: Interoperability of Software Systems Used by U.S. Manufacturers* (Tech. Rep.). Washington D.C., USA: National Coalition for Advanced Manufacturing (NACFAM).

NASA. (2012, February 3). *Down to Earth Future Aircraft.* Retrieved from http://www.nasa.gov/topics/aeronautics/features/future_airplanes_index.html (accessed 19 October 2012)

Natschläger, C. (n.d.). *Simplify the BPMN specification.* Retrieved from http://www.scch.at/en/Page56-8330.aspx (accessed 29 October 2012)

Noesis. (n.d.). *Design for real: Optimus.* Retrieved from http://www.noesissolutions.com/Noesis/ (accessed 26 October 2012)

Noy, N. F., & McGuinness, D. L. (2001, March). *Ontology Development 101: A Guide to Creating Your First Ontology* (Technical Report). Stanford, CA, 94305: Stanford Knowledge Systems Laboratory and Stanford Medical Informatics.

Oldham, K., Kneebone, S., Callot, M., Murton, A., & Brimble, R. (1998). MOKA - A Methodology and tools Oriented to Knowledge-based engineering Applications. *Proceedings of the Conference on Integration in Manufacturing*, pp. 198–207.

OWL-S Coalition. (n.d.). *OWL-S 1.2 Release.* Retrieved from http://www.ai.sri.com/daml/services/owl-s/1.2/ (accessed 29 October 2012)

PE Magazine. (2008, April). *Aerospace Industry Faces Workforce Shortage.* Retrieved from http://www.nspe.org/PEmagazine/pe_0408_Communities_Industry_aerospace.html (accessed 19 October 2012)

Pegasus. (2010). *Pegasus Project: Integrated engineering processing & materials technologies for the European sector.* Retrieved from http://www.pegasus-eu.net/ (accessed 11 November 2012)

Phoenix Integration. (n.d.). *PHX ModelCenter: Desktop Trade Studies.* Retrieved from http://www.phoenix-int.com/software/phx-modelcenter.php (accessed 26 October 2012)

PIDOTECH. (n.d.). *PIAnO.* Retrieved from http://pidotech.com/en/product/piano.aspx (accessed 26 October 2012)

Platzer, M. D. (2009, December 3). *U.S. Aerospace Manufacturing: Industry Overview and Prospects* (Tech. Rep.). Washington, D.C., USA: Congressional Research Service.

Protégé. (2013). *The Protégé project.* Retrieved from http://protege.stanford.edu/ (accessed 8 March 2013)

Prud'hommeaux, E. (n.d.). *SPARQL vs. SQL - Intro.* Retrieved from http://www.cambridgesemantics.com/semantic-university/sparql-vs-sql-intro (accessed 2 November 2012)

Reijnders, A. W. (2012). *Integrating Knowledge Management and Knowledge-Based Engineering: Formal and Platform Independent Representation of Engineering Rules.* Masters thesis, Delft University of Technology, Delft, The Netherlands.

Roth, A. V., Giffi, C. A., Chaudhuri, A., & Roehm, H. (2010, June). *2010 Global Manufacturing Competitiveness Index.* The U.S. Council on Competitiveness and Deloitte Touche Tohmatsu.

Rowley, J. (2007). The wisdom hierarchy: representations of the DIKW hierarchy. *Journal of Information Science, Vol. 33*, pp. 163–180.

Russell, N., van der Aalst, W. M. P., ter Hofstede, A. H. M., & Wohed, P. (2006). On the Suitability of UML 2.0 Activity Diagrams for Business Process Modelling. *Third Asia-Pacic Conference on Conceptual Modelling*.

Schreiber, A. T., Akkermans, J. M., Anjewierden, A., De Hoog, R., Shadbolt, N., Van De Velde, W., & Wielinga, B. J. (2000). *Knowledge Engineering and Management: The CommonKADS Methodology.* Cambridge, Massachusetts: The MIT Press.

Singh, T. (2009, August 24). *REST vs. SOAP The Right WebService.* Retrieved from http://geeknizer.com/rest-vs-soap-using-http-choosing-the-right-webservice-protocol/ (accessed 24 October 2012)

Skarka, W. (2007). Application of MOKA methodology in generative model creation using CATIA. *Engineering Applications of Artificial Intelligence, Vol. 20*, pp. 677–690.

Solid Modeling Solutions. (2012). *SMLib^{TM}.* Retrieved from http://www.smlib.com/ (accessed 3 November 2012)

Spies, B. (2008, May). *Web Services, Part 1: SOAP vs. REST.* Retrieved from http://ajaxonomy.com/2008/xml/web-services-part-1-soap-vs-rest (Accessed: July 2011)

Stokes, M. (2001). *Managing Engineering Knowledge - MOKA: Methodology for Knowledge Based Engineering Applications.* Suffolk, UK: Professional Engineering Publishing Limited.

Turban, E., Rainer, R. K., & Potter, R. E. (2005). *Introduction to Information Technology.* New York: Wiley.

van der Aalst, W. M. P., Ter Hofstede, A. H. M., & Weske, M. (2003). Business process management: A survey. In *Proceedings of the 1st international conference on business process management, volume 2678 of lncs* (pp. 1–12). Springer-Verlag.

van der Laan, A. H. (2008). *Knowledge based engineering support for aircraft component design.* Unpublished doctoral dissertation, Delft University of Technology, Delft, The Netherlands.

van Dijk, R. E. C. (2012, October 16). *Next Generation Design Systems: Knowledge-Based Multi-disciplinary Design Optimization.* Munich, Germany. (Optimus World Conference)

van Dijk, R. E. C. (2013). *Managing the Knowledge-Based Engineering Life-Cycle.* Unpublished doctoral dissertation, Delft University of Technology, Delft, The Netherlands. (to be published)

van Dijk, R. E. C., Zhao, X., Wang, H., & van Dalen, F. (2012, October 9). Multidisciplinary Design and Optimization Framework for Aircraft Box Structures. *3rd Aircraft Structural Design Conference*.

van Tooren, M. J. L. (2003, March 5). *Sustainable Knowledge Growth* (Inaugural Speech). Delft, The Netherlands: Delft University of Technology.

Verhagen, W. J. C., Bermell-Garcia, P., van Dijk, R. E. C., & Curran, R. (2011). A Critical Review of Knowledge-Based Engineering: An Identification of Research Challenges. *Advanced Engineering Informatics*.

W3C. (2002, August 19). *Web Services Architecture Requirements.* Retrieved from http://www.w3.org/TR/2002/WD-wsa-reqs-20020819 (accessed 24 October 2012)

W3C. (2004a, February 10). *OWL Web Ontology Language.* Retrieved from http://www.w3.org/TR/owl-features/ (accessed 24 October 2012)

W3C. (2004b, February 10). *RDF Vocabulary Description Language 1.0: RDF Schema.* Retrieved from http://www.w3.org/TR/rdf-schema/ (accessed 24 October 2012)

W3C. (2004c, February 10). *Resource Description Framework (RDF).* Retrieved from http://www.w3.org/RDF/ (accessed 23 October 2012)

W3C. (2007, April 27). *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition).* Retrieved from http://www.w3.org/TR/soap12-part1/ (accessed 24 October 2012)

W3C. (2008, January 15). *SPARQL Query Language for RDF.* Retrieved from http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/ (accessed 23 October 2012)

Walsh, J. (2011, May 23–26). What's Stopping Widespread Deployment of Simulation Tools? *NAFEMS World Congress 2011*.

Weier, M. H., & Smith, L. (2007, April 14). *Businesses Get Serious About Software As A Service.* Retrieved from http://www.informationweek.com/businesses-get-serious-about-software-as/199000824 (accessed 24 October 2012)

Weske, M. (2012). *Business Process Management: Concepts, Languages, Architectures* (2nd ed. ed.). Springer Berlin Heidelberg. doi: 10.1007/978-3-642-28616-2

Wikipedia. (2012a). *Description logic — wikipedia, the free encyclopedia.* Retrieved from http://en.wikipedia.org/w/index.php?title=Description_logic&oldid=508802095 ([Online; accessed 24-October-2012])

Wikipedia. (2012b). *Ontology — wikipedia, the free encyclopedia.* Retrieved from http://en.wikipedia.org/w/index.php?title=Ontology&oldid=519253132 ([Online; accessed 24-October-2012])

Yang, H. Z., Chen, J. F., Ma, N., & Wang, D. Y. (2012). Implementation of knowledge-based engineering methodology in ship structural design. *Computer-Aided Design, Vol. 44 No. 3*, 196 - 202. Retrieved from http://www.sciencedirect.com/science/article/pii/S0010448511001540

Young, D. E. (2010, February 19). *The Lisa Project.* Retrieved from http://lisa.sourceforge.net/ (accessed 9 December 2012)

Appendix A

Workflow Modelling Languages

This chapter describes the search for a new workflow modelling language that will be used in the newly developed Workflow Management System (WFMS). This is desired because well-known PIDO solutions all have their own modelling language designed specifically for their platform and not based on industry standards. As a result, users are restricted to using the native language that is provided. This has several disadvantages.

When the language is not an industry standard, and thus only used in one PIDO system, then people need to be trained to use the notation. Their gained expertise is not transferable to another system though. Thus, the amount of qualified experts remains small. If the PIDO solution is not widely adopted, it will be difficult to find experienced people. Another consequence is vendor lock-in. This occurs when all the workflows have been modelled in the vendor's language, and have become dependent on one platform. Switching to another PIDO solution will be costly, since saved workflows are not easily transferable and need to be translated to the new language. Even then, it is not certain whether that is possible. When it is not entirely clear what the workflows are supposed to do, it will be very difficult, if not impossible, to recreate the workflow in the new language.
Finally, the platform-specific languages do not align with higher level (business) processes. These processes define what the simulation workflow is supposed to do, and how it contributes to the overall design process. The higher level process needs to be mapped onto the simulation workflow. This extra step is a non-value adding and time consuming task.

From these reasons emerged the goal to find a neutral language, that is platform-independent and easy to learn and use also for non-IT experts. Ultimately, it has to fulfil these requirements:

- The language has to provide a graphical notation. Modelling workflows is more intuitive using diagrams than code.
- Preferably, the language has been designed for execution, as it may facilitate automatic workflow generation.
- The language has to be standardised to ensure that workflows are easily portable between systems.

- Besides standardisation, the language should be widely supported. This guarantees that a sufficient amount of qualified experts is available, that it is well documented, and that there are enough tools available supporting the language.
- The language must be easy to learn and use. Companies are hesitant to adopt new technological solutions if it is not clearly understood. Therefore, it is important to lower the threshold for new users.

The Business Process Management (BPM) community has put much effort in developing standardised notations for modelling processes. Moreover, as these notations are intended for business users, simplicity has been highly regarded. Therefore, BPM would be a good source for finding a suitable language. The most popular languages are briefly analysed in the section below.

Unified Modeling Language – Activity Diagram

The Unified Modeling Language (UML) is a family of notations, originally designed for object oriented programming. Since then, it has grown to become the number one standard in the programming field. Although UML diagrams have been designed for software modelling, the UML Activity Diagrams (UML AD) are occasionally used for modelling business processes as well. The characteristics of UML AD are listed in Table A.1.

Table A.1: Strengths and weaknesses of UML Activity Diagrams.

Strengths	Weaknesses
• Provides a standardised graphical notation • Comprehensive support for control flow and data flow (Russell et al., 2006) • The UML standard dominates the software modelling domain	• Some of their constructs lack a precise syntax and semantics (Dumas & ter Hofstede, 2001). • Limited in modelling organisational aspects of business processes (Russell et al., 2006).

Business Process Modeling Notation

The Business Process Modeling Notation (BPMN) is by far getting the most attention from the BPM community in the last few years. Partly because it has been adopted by the Object Management Group (OMG) as a BPM standard. It became popular as a graphical standard, but since version 2.0 the BPMN specification has included execution semantics. Formerly it relied on other technologies for execution (often BPEL). Furthermore, because BPMN has been designed for the business user, it is easy to learn, even for non-IT specialists. Table A.2 summarises the characteristics of BPMN.

Event-driven Process Chain

Another language that is often encountered in BPM is Event-driven Process Chain (EPC). Similar to BPMN, EPC was not designed for modelling executable processes, but rather

Table A.2: Strengths and weaknesses of BPMN.

Strengths	Weaknesses
• Provides a standardised graphical notation • Easy to understand for non-IT experts • BPMN 2.0 includes execution semantics • A subset of BPMN can be naturally transformed to BPEL for execution	• The BPMN 2.0 execution language is still very new and not thoroughly tested and supported • BPMN was initially not designed for process execution, but for communication between people

for visualising business processes. It provides the expressiveness to include business-oriented elements directly in the process. Although EPC is quite well supported, it has not become a standard. The characteristics of EPC are described in Table A.3.

Table A.3: Strengths and weaknesses of EPC.

Strengths	Weaknesses
• EPC is more expressive in linking with business-oriented elements than most languages • Is therefore most often used for high-level business processes • It is easier to learn than BPMN	• Less expressive in control flow structures than BPMN • Although regularly used, it is not a standard

Business Process Execution Language

BPEL is the preferred execution language for business processes. It has been designed from the start as an orchestration language for web services. Due to this different approach, BPEL does not have a standard graphical notation, which was considered as out of scope. Furthermore, it is part of the WS-$*$ (read "WS-Star") standards for web services. BPEL's strengths and weaknesses are listed in Table A.4.

Table A.4: Strengths and weaknesses of BPEL.

Strengths	Weaknesses
• BPEL is one of the most frequently used and widely accepted industry business process execution language • Is one of web service standards (WS-$*$) • Focuses on operational semantics for executing processes, but provides both a programmatic and graph-oriented approach	• Not great in dealing with data structures and complex control flows • No standardised graphical notation • Limited expressibility compared to BPMN, which is not ideal for BPM • No native support for human tasks, but extensions are available with BPEL4People and WS-HumanTask

Yet Another Workflow Language

YAWL stands for Yet Another Workflow Language, and seems to be designed as a replacement for BPEL. It is an extension of petri nets, which is a mathematical modelling lan-

guage for processes. Having this mathematical foundation has the benefit that processes can be analysed more easily. YAWL is thus built on a strong basis, but its acceptance as a new standard has not occurred yet. See Table A.5 for the characteristics of YAWL.

Table A.5: Strengths and weaknesses of YAWL.

Strengths	Weaknesses
• Integrated support for human tasks • Formal semantics based on mathematical foundation • Includes (formal) graphical notation	• It is not a standard and neither supported by industry; currently it is a single system mainly developed by academia • No support for meta-modeling, to describe what the workflow does

Trade-off

Based on the analysis of all languages, a decision can be made in a direct comparison. The trade-off is done qualitatively using equal weight factors. Each language is evaluated on the criteria mentioned in the introduction. The results are shown in Table A.6.

Table A.6: Trade-off of the various workflow modelling languages.

Criteria	Weight	UML	BPMN	EPC	BPEL	YAWL
Graphical notation	1	++	++	++	-	++
Designed for execution	1	-	+	- -	++	++
Standardised	1	++	++	-	++	-
Widely adopted	1	++	++	+	++	-
Easy to use	1	+	+	+	-	0
Score	**max: 10**	**6**	**8**	**1**	**4**	**2**

BPMN has the highest score after the trade-off, because it is the most complete language of these five. That may also be the reason that it is quickly becoming the industry standard in BPM. Its direct competitor, UML AD, is in fact very similar to BPMN. Even the graphical notation shows some similarities. However, it lost points on execution, because UML AD cannot be mapped to execution code in contrast to BPMN (Ko et al., 2009). Moreover, UML AD is losing followers who seem to favour BPMN for process modelling.

EPC and YAWL have mainly lost points because both languages have not been standardised. The last one, BPEL, has proven itself as a formal execution language, but failed as a graphical notation. Because of its focus on execution, many implementations of BPEL only provide a graphical interface that reflects the code underneath (Ko et al., 2009). This is far from optimal when it comes to ease of use.

Appendix B

Code Documentation

This chapter provides more extensive documentation for the code that has been written during this research. The schematic architecture shown in Chapter 5 has been repeated here (see Figure B.1).

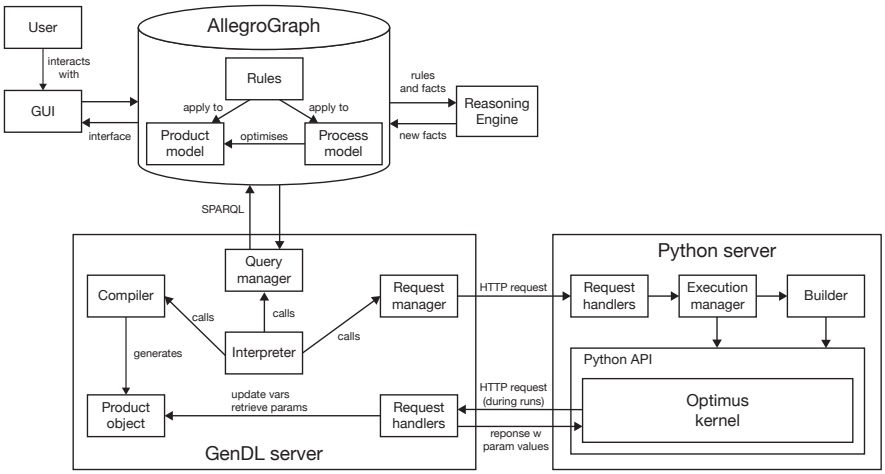


Figure B.1: Architecture of the integration framework. The GenDL server code is written in Common Lisp.

The programming can be divided into two sections: **Common Lisp (CL) code** and **Python code**. The CL code inside the GenDL server manages all the interactions with the triple store (AllegroGraph), interprets the knowledge, and sends requests to the Python server to generate the workflow. This process is fully automated and does not require human intervention (except for triggering the whole process). It provides the connection between the KB and Optimus.
The Python code is written to wrap around the Optimus Python API. The wrapper simplifies interaction with the API. Additionally, it has a built-in server that allows users to send HTTP requests to generate and execute simulation workflows. Through this

functionality, Optimus has not only become available to the GenDL server, but to everyone who has access to the server.
The following sections explains how the code is built up.

B.1 Common Lisp Code

The CL code consists of an *interpreter*, *query manager*, and *request manager*. Together, these modules ensure a smooth transition from formal knowledge to simulation workflow. The **interpreter** is the centre of this subsystem. It controls the main generation process and is supported by the other modules in this process (see Figure B.2). The **query manager** manages all the interactions with the triple store. It only queries the triple store and returns the results untouched. To keep functions separated, only the interpreter is able to "interpret" the query results. The remaining component is the **request manager**. The request manager is in control of all outgoing HTTP requests to the Python server. Data is sent in the JSON format. JSON is an alternative to XML that is more concise and easier to parse by computers. It is a very popular format on the web, because it is natively supported in JavaScript. The main elements of JSON are object (curly braces {}) and arrays (square brackets [ ]). Objects contain key-value pairs, similar to dictionaries or hash-tables in programming languages, and arrays are lists of values. The example below shows how both elements are used to store data.

```
{"inputarray" : {"name" : "Design variables", "pos" : [100, 200], "vars" : [
    {"name" : "length", "nominal" : 5, "lowerbound" : 1, "upperbound" : 10},
    {"name" : "width", "nominal" : 5, "lowerbound" : 1, "upperbound" : 10},
    {"name" : "height", "nominal" : 5, "lowerbound" : 1, "upperbound" : 10}]}}
```

This example will create an input array in Optimus with the name *"Design variables"* at the position (100, 200) (x,y-coordinates). Next to the key *"vars"* begins an array that contains the design variables *length*, *width*, and *height*.

Figure B.2 shows an UML Activity Diagram of the CL code. At the beginning of the process, before tasks can be generated, Graphviz is called to determine the x,y-coordinates of every task in the workflow. Thereafter, the interpreter will loop through all the tasks and generate these one after another. In this loop, the interpreter determines the type of the task, retrieves values for necessary input parameters, and sends requests to the Python server to generate the task. When all tasks are created, the interpreter sends requests to generate all connections (sequence and data flow). At this point, the entire workflow has been generated. The final step is to set up the execution method. If the command is given to execute the workflow immediately, Optimus will execute it and save the project once it finishes the run. Otherwise, the project is saved immediately, so that the user can open the workflow in Optimus for inspection.

B.2 Python Code

The Python code is inspired by the facade pattern (see Figure B.3), one of the infamous design patterns for object oriented programming (Gamma et al., 1994). The facade pattern is, analogous to an architectural facade, a wall that hides everything behind the

Figure B.2: UML Activity Diagram of the automatic workflow generation process.

facade. Thus, in programming a facade is a higher level interface that simplifies interaction with the (sub)system behind the facade. Users interact only with the facade, hence are not troubled with the code behind it. The abstraction works very well for creating the wrapper code around the Optimus API.

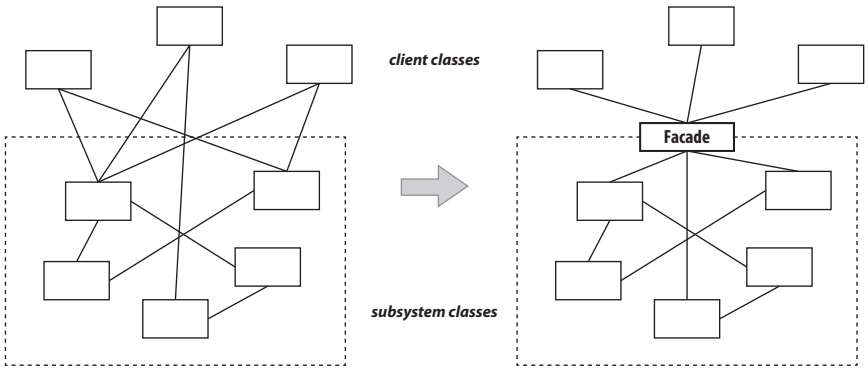


Figure B.3: The facade pattern; one of the many design patterns for object oriented programming (source: Gamma et al., 1994)

Figure B.1 shows that the Python code consists of a *request handler*, *execution manager*, and *builder*. The **request handler** contains the server functions, and 'listens' to incoming HTTP requests and relays requests to the appropriate wrapper functions. These requests are processed by the **execution manager**. This module is connected to the Optimus API (e.g. for managing Optimus projects), but also directs the builder, which creates the various workflow elements. The **builder** in Figure B.1 is divided into two modules: an *item builder* and *execution configurator*. The **item builder** creates all the workflow elements, whereas the **execution configurator** is responsible for setting up the execution method.

Table B.1: All the functions of the Python server. The function is described by its path in a URL. The full URL would become: *http://host:port/optimus/start/project*

Server functions	Description
1. /optimus/start/project	Starts a new Optimus project
2. /optimus/saveandclose/project	Saves and closes the Optimus project
3. /optimus/create/input	Creates an input array containing input variables
4. /optimus/create/output	Creates an output array containing output variables
5. /optimus/create/action	Creates an "action" element that can execute a script
6. /optimus/create/uca	Creates a User Customisable Action (UCA), which is defined by the user to perform a custom task (e.g. send HTTP request to a web service)
7. /optimus/create/file	Creates a file object
8. /optimus/create/connection	Creates a connection between two workflow nodes
9. /optimus/configure/execution	Configures the execution method
10. /optimus/execute/workflow	Triggers execution of the workflow

Table B.1 presents the URLs of all the server functions. The server is built with the

Bottle web framework (v0.10.2) (Hellkamp, 2012), which is, besides the Optimus Python API module, the only external module (i.e. not in the standard Python library) used in the Python code. The server simply redirects the requests to the appropriate wrapper function. The code for the wrapper is described in the UML diagram in Figure B.4.

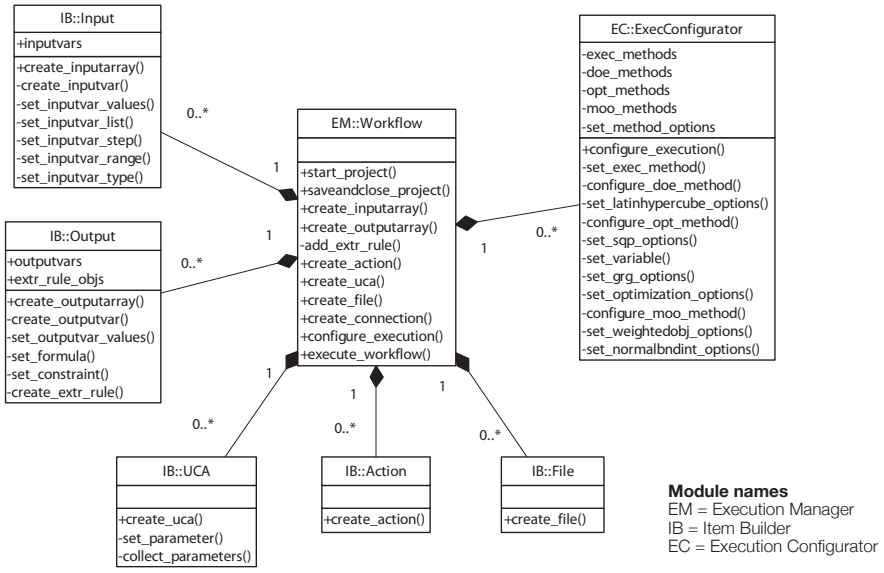


Figure B.4: UML class diagram of the wrapper around the Optimus Python API.

The main component is the *Workflow* class, which contains the functions that are accessed by the server. These functions either interact directly with the API or instantiate one of the other classes. Every class is in itself a facade, as many methods of the classes are private (denoted by the minus sign in front of the method name). The Workflow class can simply call the top-level function, provide the necessary input, and the responsible class will take care of the request.

Appendix C

MOKA 2: Tools and Methods

This chapter is an addition to Chapter 4, which describes MOKA 2. It contains the new Discipline-, Activity-, and Tool-forms that are part of the methodology, but have not been presented in Chapter 4. Since the Problem-form is already included in Chapter 4, it is not repeated here.
Furthermore, Figures C.4 and C.5 show the ontologies for product and rule respectively. The ontologies have been included for completeness and are not further explained here. The product ontology is covered in detail by van Dijk (2013), while the rule ontology is explained by Reijnders (2012).

The first form, the D-form, captures relevant knowledge for understanding disciplines (see Figure C.1). It describes the discipline's inputs and outputs, and the involved P-, A-, and E-forms.
The A-form is modified from the original A-form in MOKA's ICARE to describe simulation workflow activities. The main addition is the more detailed description of input and output parameters (see Figure C.2).
Finally, the T-form captures knowledge for interacting with tools (see Figure C.3). Additionally, it includes the necessary information for selecting the right tool for a certain design task, such as idealisations that have been applied or the context of the tool (e.g. buckling analysis). The second table at the bottom is designed to describe web services, with fields for the HTTP request and HTTP response.

Figure C.6-C.8 show additional user interface mockups for modelling N^2 diagrams, filling in ICARE PDT-forms, and modelling the ontology in the Formal Model. The notation used in Figure C.8 is based on UML class diagrams, although an extension is needed to cover OWL. An option is to use OWLGrEd by Bārzdiņš et al. (2013).

MOKA2 form:	Discipline
Name	Name of the discipline
Reference	Reference to the discipline (e.g. an ID)
Objective	Short text to explain the general objectives
Description	Description of the discipline explaining the details. This is for understanding what it does.
Input requirements/preconditions	Input requirements or preconditions before the discipline can be executed
Context, information, validity	Short text to explain in which case this knowledge can be applied
Problem involved	Reference to MOKA Problem form that this discipline is part of
Activities involved	Reference to MOKA Activity forms that perform the disciplinary analysis
Entities involved	Reference to MOKA Entity forms that are required in the discipline
Parameters	
Name	Name of the parameter (Name of the symbol when applicable)
Description	From the description of the parameter it must be clear what the activity exactly needs as input and what it produces as output
Input/Output	Input or output
Variable/Fixed	Variable or fixed
Parameter Value	Value of the parameter
Default value (process)	Default value of the parameter defined in the process
Default value (tool)	Default value of the parameter defined in the tool
Source	A reference to the source of the parameter makes it easier to understand where it comes from (dataflow), e.g. User input, output of another discipline, file, etc.
Management	
Author	Author of the form
Date	Date of last modification
Version number	Version number of the form
Status	In Progress/Complete/Verified (pick list)
Modification	List any modifications to the form
Information origin	References to the associated raw knowledge

Figure C.1: The new Discipline form (D-form) captures discipline-specific knowledge.

MOKA2 form:	Activity
Name	Name of the activity
Reference	Reference to the activity (e.g. an ID)
Objective	Short text to explain the general objectives
Description	Description of the activity explaining the details. This is for understanding what it does.
Trigger	Describe the event that triggers this activity
Input requirements/preconditions	Input requirements or preconditions before the activity can be executed
Context, information, validity	Short text to explain in which case this knowledge can be applied
Potential failure modes	Completion criteria should enable one to assess whether the activity has been completed satisfactorily and, if not, what additional acitvity is required
Executed by	Name of the person, team, or software tool that executes this activity. When it is a person, name the role instead, e.g. Design engineer, knowledge engineer, etc.
Related activities	
Parent activity	Name the parent activity if the current activity belongs to a sub-process.
Sub activities	List the sub activities if the current activity itself is a sub-process.
Preceding activities	The preceding activities in the sequence
Following activities	The following activities in the sequence
Rules involved	Reference to MOKA Rule forms that are required in the activity, e.g. business rules, process rules, formulas, etc.
Entities involved	Reference to MOKA Entity forms that are required in the activity.
Parameters	
Name	Name of the parameter (Name of the symbol when applicable)
Description	From the description of the parameter it must be clear what the activity exactly needs as input and what it produces as output
Input/Output	Input or output
Variable/Fixed	Variable or fixed
Parameter Value	Value of the parameter
Default value (process)	Default value of the parameter defined in the process
Default value (tool)	Default value of the parameter defined in the tool
Source	A reference to the source of the parameter makes it easier to understand where it comes from (dataflow), e.g. User input, output of another activity, file, etc.
Management	
Author	Author of the form
Date	Date of last modification
Version number	Version number of the form
Status	In Progress/Complete/Verified (pick list)
Modification	List any modifications to the form
Information origin	References to the associated raw knowledge

Figure C.2: The modified Activity form (A-form) captures knowledge about activities.

MOKA2 form:	Tool
Name	Name of the tool
Reference	Reference to the tool (e.g. an ID)
Software application	The software application that the tool uses (include version number)
Context	Describe for which case this tool can be used. This is useful, for instance, when searching for tools that can do "buckling" analysis
Objective	Short text to explain the general objectives
Description	Description of the tool explaining the details. This is for understanding what it does.
Idealisations and limitations	Describe clearly which idealisations have been applied in the tool and what the limitations are of the tool
Input requirements/preconditions	Input requirements or preconditions before the tool can be executed
Parameters	
Name	Name of the parameter (Name of the symbol or name in the query string when applicable)
Description	From the description of the parameter it must be clear what the tool exactly needs as input and what it produces as output
Input/Output	Input or output
Required/Optional	Required or optional parameter
Default value	Default value of the parameter
Data type	Data type of the parameter (i.e. String, double, etc.)
Management	
Author	Author of the form
Date	Date of last modification
Version number	Version number of the form
Status	In Progress/Complete/Verified
Modification	List any modifications to the form
Information origin	References to the associated raw knowledge
MOKA DPM form:	**Service**
URL	URL of the service (without query string), e.g. http://127.0.0.1:9000/example/service
Message format	Specify whether data is sent as a query string, XML, JSON, etc.
Example request (query string format)	Show an example of the http request for clarification (i.e. The URL including query string)
Example response	Describe the response that is returned by the server, also for clarification purposes

Figure C.3: The new Tool form (T-form) captures knowledge about software tools.

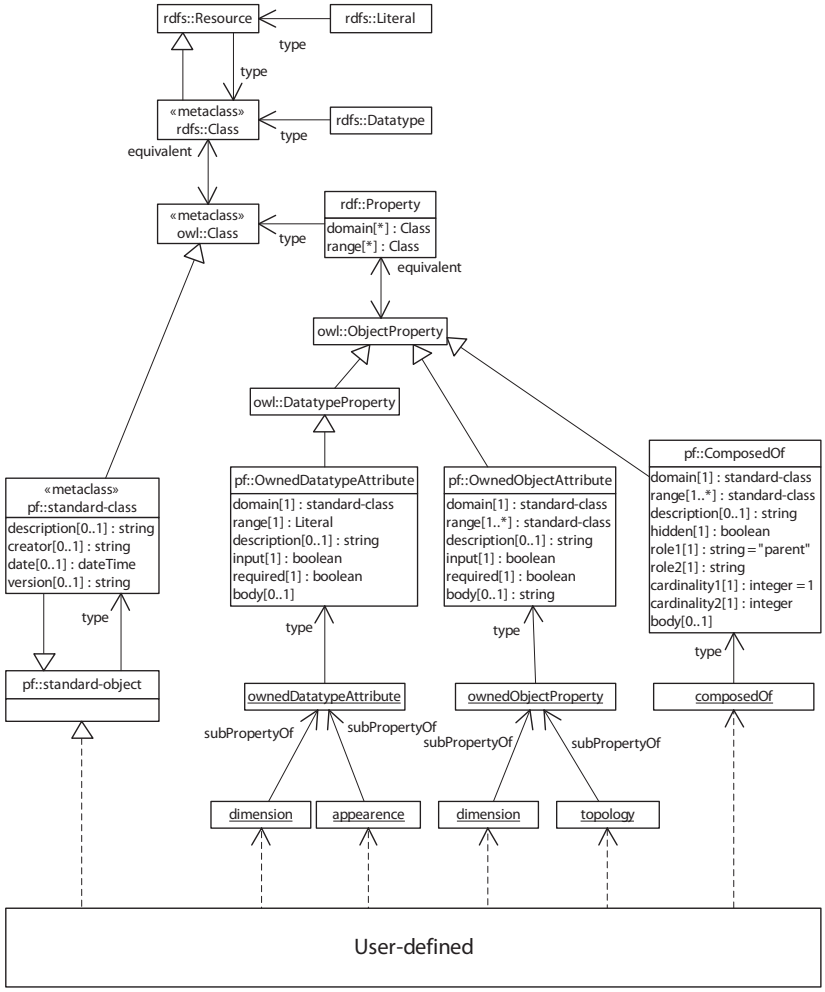


Figure C.4: UML diagram of the product ontology (source: van Dijk, 2013).

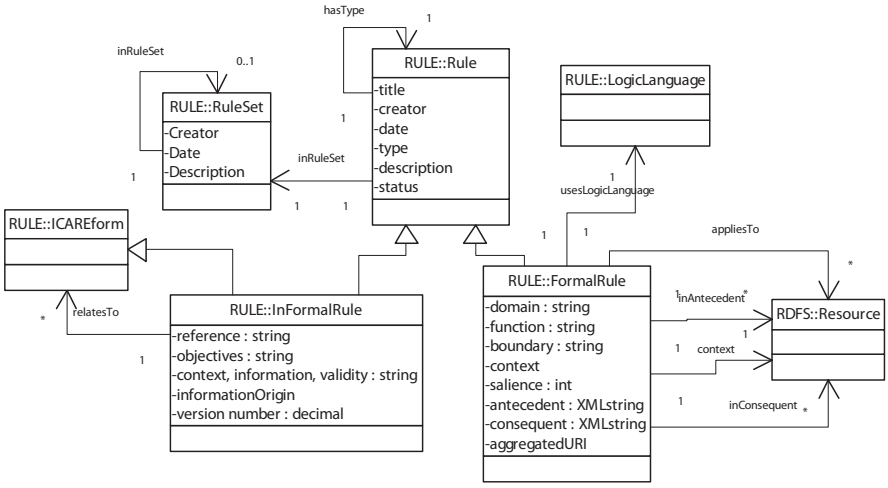


Figure C.5: UML diagram of the rule ontology (source: Reijnders, 2012).

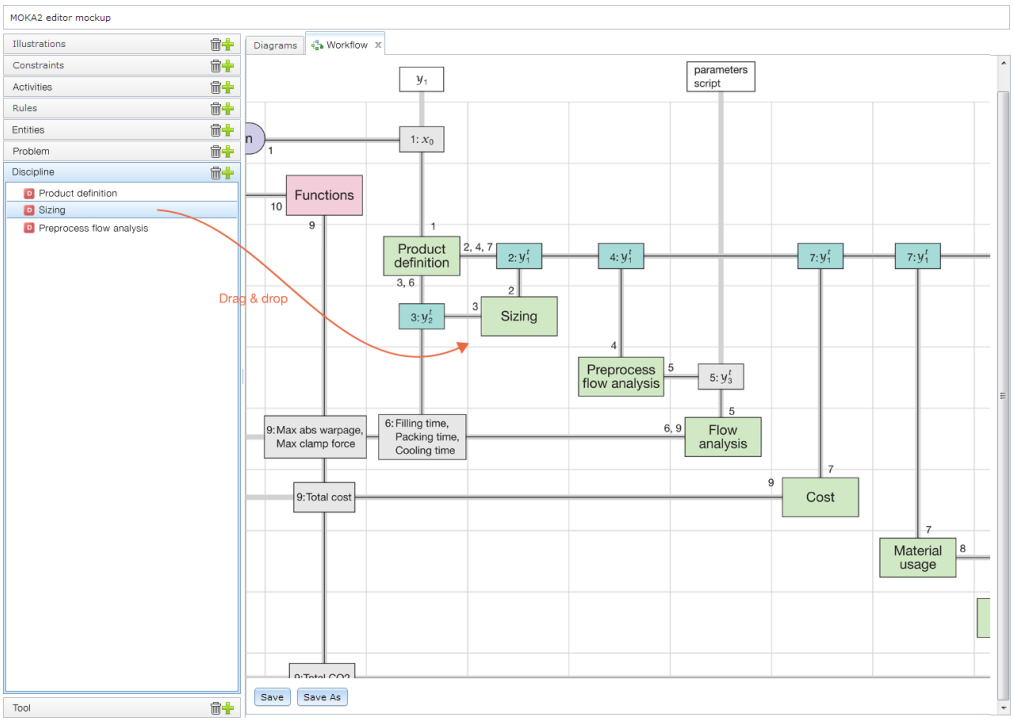


Figure C.6: User interface mockup for modelling N^2 diagrams.

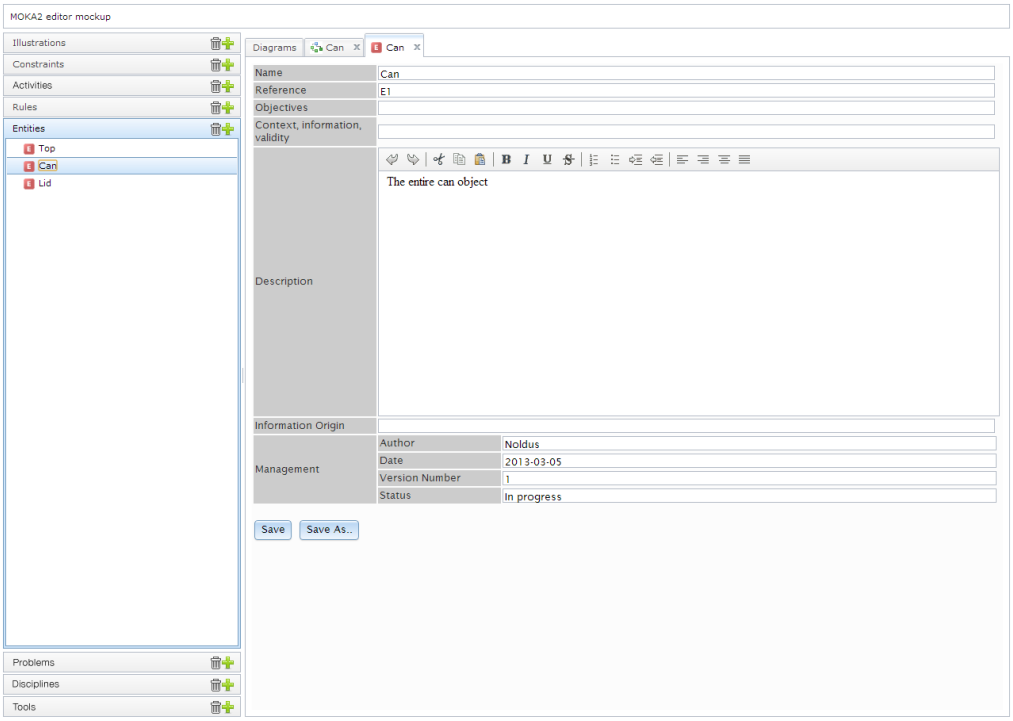


Figure C.7: User interface mockup for filling in ICARE PDT-forms.

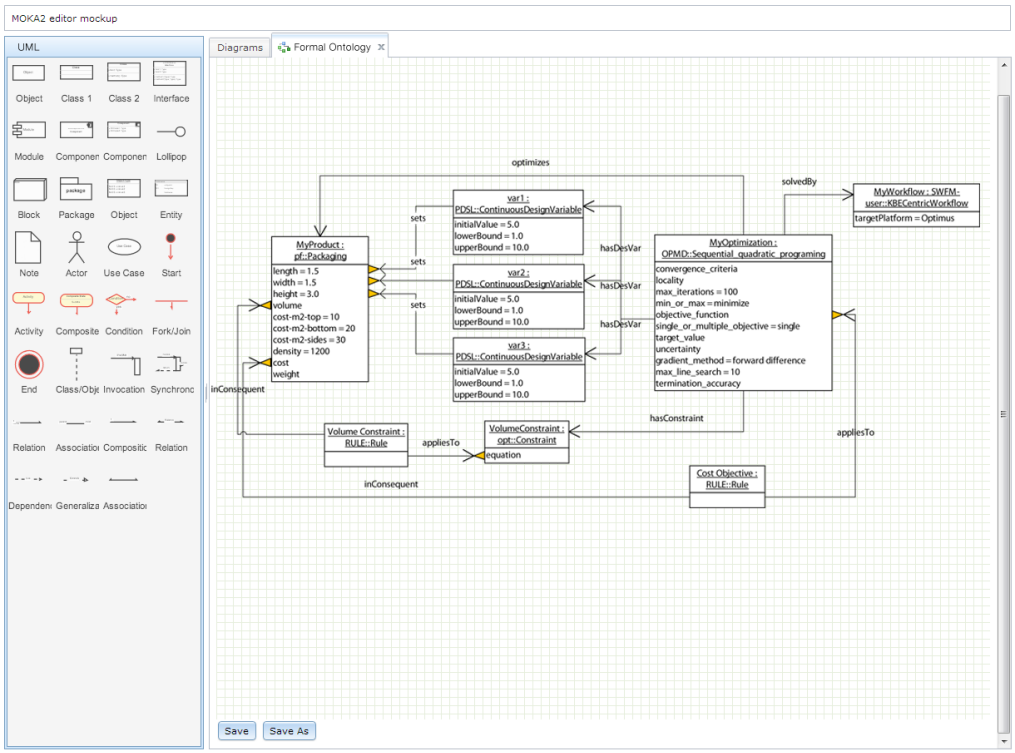


Figure C.8: User interface mockup for modelling the ontology in the Formal Model.

Appendix D

Rules for Automatic Workflow Generation

This chapter contains all the rules that have been modelled for automatic workflow generation. These rules have been applied to both the packaging use case (Chapter 7) and MDO use case (Chapter 8). Some rules are applicable to both use cases, while others are tailored specifically to that use case. Rules that are applied to both, may eventually become part of the methodology. But with only two use cases, it is not certain whether these rules are truly generic. Below, in Table D.1, is an index of all rules, including their domain, description, and the use case it has been applied to.

Two rules have already been explained in the packaging use case (R4 and R21). There are a total of 29 rules that have been used. Not all rules are explained in this chapter, as all the rules together take up more than a 1000 lines of code. Instead, six rules have been selected and are explained in Figures D.1–D.7. The rest of the rules are included, but not explained in detail.

Table D.1: Rules index

ID	Domain	Title	In use case
R1	General	hasValue restrictions	packaging, MDO
R2	General	Inherit hasValue restrictions	MDO
R3	General	Dataflow	packaging, MDO
R4	HLA	Instantiate flow elements	packaging, MDO
R5	HLA	Connect sequenceflows	packaging, MDO
R6	HLA	Map input from sub-process to tasks	packaging, MDO
R7	HLA	Map output from sub-process to tasks	packaging, MDO
R8	HLA	Map product model from sub-process to tasks	packaging, MDO
R9	HLA	Instantiate data objects in sub-processes	packaging, MDO
R10	HLA	Attach file parameters to files in high level processes	MDO
R11	HLA	Map product name to query parameter	packaging, MDO
R12	HLA	Map product attribute to query parameter (get-parameter)	packaging, MDO
R13	HLA	Map product attributes to query parameter (set parameter; variable)	packaging, MDO
R14	HLA	Map product attribute to query parameter (set parameter; fixed; required)	MDO
R15	HLA	Map product attribute to query parameter (set parameter; optional; process default)	MDO
R16	HLA	Map required input parameters to query parameter (variable)	MDO
R17	HLA	Map required input parameters to query parameter (required; fixed)	MDO
R18	HLA	Map required input parameters to query parameter (optional; process default)	MDO
R19	KBECWF	Instantiate file parameters	packaging
R20	KBECWF	Bind product to activities	packaging
R21	KBECWF	Design variable input (part 1)	packaging
R22	KBECWF	Design variable input (part 2)	packaging
R23	KBECWF	Objective function input (part 1)	packaging
R24	KBECWF	Objective function input (part 2)	packaging
R25	KBECWF	Constraint input (part 1)	packaging
R26	KBECWF	Constraint input (part 2)	packaging
R27	Intelligence	Automatic file transfer (SFTP)	MDO
R28	Moldflow	Configure Send_SettingsFile	MDO
R29	Moldflow	Configure Retrieve_SimulationResults	MDO

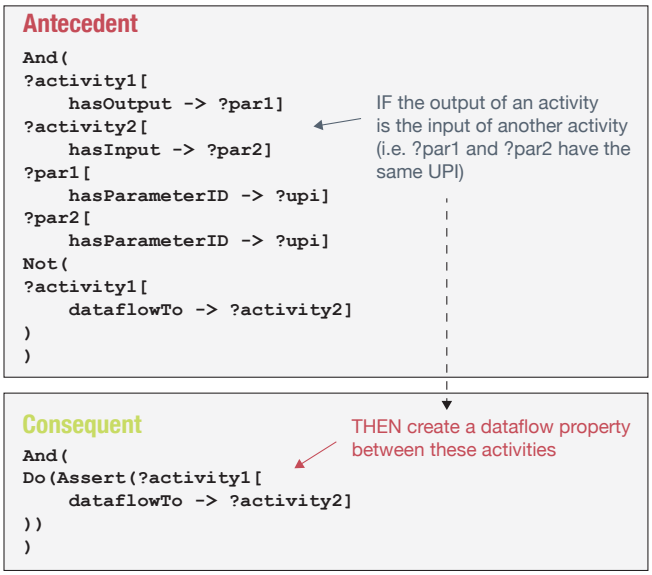


Figure D.1: R3 - Dataflow

Listing D.1: R1 - hasValue restrictions

```
If
    And(
    ?subj[type -> ?class]
    ?class[subClassOf -> ?restr]
    ?restr[type -> Restriction
           onProperty -> ?pred
           hasValue -> ?obj])
Then
    And(
    Do(Assert(?subj[?pred -> ?obj])))
```

Listing D.2: R2 - Inherit hasValue restrictions

```
If
    And(
    ?subclass[subClassOf -> ?class]
    ?class[subClassOf -> ?restr]
    ?restr[type -> Restriction
           hasValue -> ?obj
           onProperty -> ?pred])
Then
    And(
    Do(Assert(?subclass[subClassOf -> External(func:new-individual(?class ?restr))]))
    Do(Assert(External(func:new-individual(?class ?restr))[
        type -> Restriction
        onProperty -> ?pred
        hasValue -> ?obj])))
```

Listing D.3: R5 - Connect sequenceflows

```
If
    And(
    ?process[flowElements -> ?flow
             flowElements -> ?node]
    ?flow[type -> ?class]
    ?class[subClassOf -> ?restr]
    ?restr[type -> Restriction
           onProperty -> ?ref
           valuesFrom -> ?obj]
    ?node[type -> ?obj]
    External(func:resource-of-type(?flow SequenceFlow)))
Then
    And(
    Do(Assert(?flow[?ref -> ?node])))
```

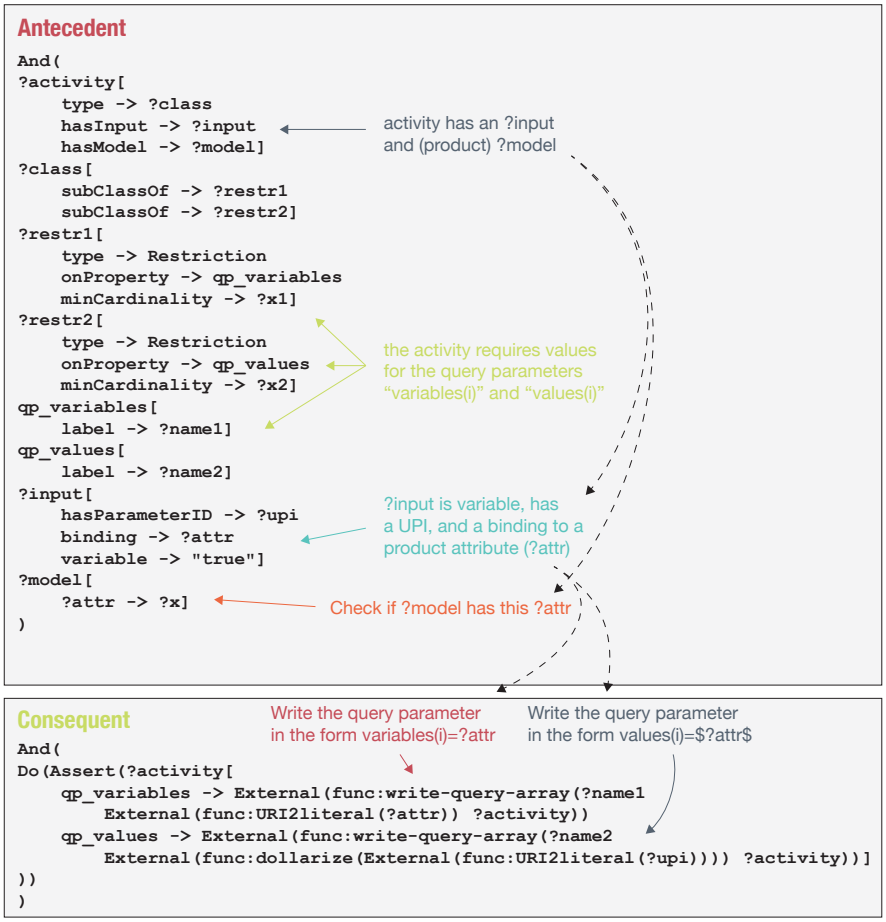


Figure D.2: R13 - Map product attributes to query parameter (set parameter; variable)

Listing D.4: R6 - Map input from sub-process to tasks

```
If
    And(
    ?sp[flowElements -> ?activity
        hasInput -> ?input]
    ?activity[type -> ?class]
    ?class[subClassOf -> ?restr]
    ?restr[type -> Restriction
           onProperty -> hasInput]
    Or(
    ?restr[valuesFrom -> ?inputtype
           someValuesFrom -> ?inputtype
           allValuesFrom -> ?inputtype])
    External(func:resource-of-type(?input ?inputtype)))
Then
    And(
    Do(Assert(?activity[hasInput -> ?input])))
```

Listing D.5: R7 - Map output from sub-process to tasks

```
If
    And(
    ?sp[flowElements -> ?activity
        hasOutput -> ?output]
    ?activity[type -> ?class]
    ?class[subClassOf -> ?restr]
    ?restr[type -> Restriction
           onProperty -> hasOutput]
    Or(
    ?restr[valuesFrom -> ?outputtype
           someValuesFrom -> ?outputtype
           allValuesFrom -> ?outputtype])
```

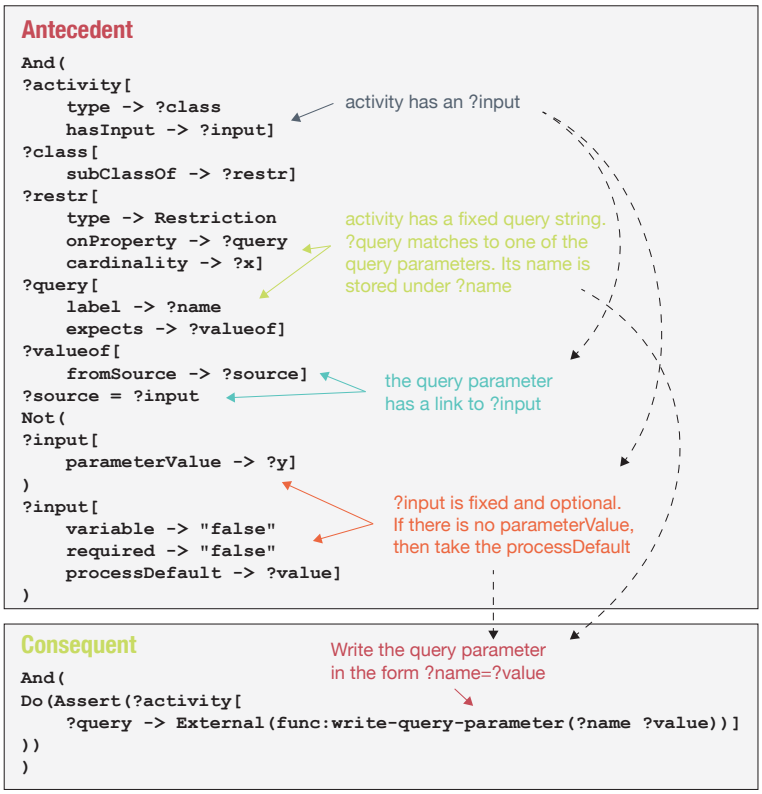


Figure D.3: R18 - Map required input parameters to query parameter (optional; process default)

```
    External(func:resource-of-type(?output ?outputtype )))
Then
    And(
    Do(Assert(?activity[hasOutput -> ?output])))
```

Listing D.6: R8 - Map product model from sub-process to tasks

```
If
    And(
    ?sp[flowElements -> ?activity
        hasModel -> ?model]
    ?activity[type -> ?class]
    ?class[subClassOf -> ?restr]
    ?restr[type -> Restriction
           onProperty -> hasModel])
Then
    And(
    Do(Assert(?activity[hasModel -> ?model])))
```

Listing D.7: R9 - Instantiate data objects in sub-processes

```
If
    And(
    ?sp[flowElements -> ?activity
        flowElements -> ?activity2]
    ?activity[type -> ?class]
    ?class[subClassOf -> ?restr]
    ?restr[type -> Restriction
           onProperty -> hasOutput
           valuesFrom -> ?outputtype]
    ?activity2[type -> ?class2]
    ?class2[subClassOf -> ?restr2]
    ?restr2[type -> Restriction
            onProperty -> hasInput
            valuesFrom -> ?inputtype]
```

Figure D.4: R19 - Instantiate file parameters

```
    ?outputtype = ?inputtype)
Then
    And(
    Do(Assert(?activity[hasOutput -> External(func:new-individual(?sp ?outputtype))]))
    Do(Assert(?activity2[hasInput -> External(func:new-individual(?sp ?inputtype))]))
    Do(Assert(External(func:new-individual(?sp ?outputtype))[type -> ?outputtype])))
```

Listing D.8: R10 - Attach file parameters to files in high level processes

```
If
    And(
    ?act[hasInput -> ?file
         hasOutput -> ?output]
    ?output[binding -> ?filepar]
    External(func:resource-of-type(?filepar FileParameter))
    External(func:resource-of-type(?file File)))
Then
    And(
    Do(Assert(?file[hasParameter -> ?filepar])))
```

Listing D.9: R11 - Map product name to query parameter

```
If
    And(
    ?activity[type -> ?class
              hasModel -> ?model]
    ?class[subClassOf -> ?restr]
    ?restr[type -> Restriction
           onProperty -> ?query]
    ?query = qp_productIID
    ?query[label -> ?name])
Then
    And(
```

Figure D.5: R27 (Antecedent) - Automatic file transfer (SFTP)

```
    Do(Assert(?activity[?query -> External(func:write-query-parameter(?name
        External(func:URI2literal(?model))))])))
```

Listing D.10: R12 - Map product attribute to query parameter (get-parameter)

```
If
    And(
    ?activity[type -> ?class
              hasModel -> ?model
              hasOutput -> ?output]
    ?class[subClassOf -> ?restr]
    ?restr[type -> Restriction
           onProperty -> ?query]
    ?query = qp_responses
    ?query[label -> ?name]
    ?output[binding -> ?attr]
    ?model[?attr -> ?x])
Then
    And(
    Do(Assert(?activity[?query -> External(func:write-query-array(?name
        External(func:URI2literal(?attr ))?activity))])))
```

Figure D.6: R27 (Consequent) - Automatic file transfer (SFTP)

Listing D.11: R14 - Map product attribute to query parameter (set parameter; fixed; required)

```
If
    And(
    ?activity[type -> ?class
              hasInput -> ?input
              hasModel -> ?model]
    ?class[subClassOf -> ?restr1
           subClassOf -> ?restr2]
    ?restr1[type -> Restriction
            onProperty -> qp_variables
            minCardinality -> ?x1]
    ?restr2[type -> Restriction
            onProperty -> qp_values
            minCardinality -> ?x2]
    qp_variables[label -> ?name1]
    qp_values[label -> ?name2]
    ?input[binding -> ?attr
           variable -> "false"
           parameterValue -> ?value]
    ?model[?attr -> ?x])
Then
    And(
    Do(Assert(?activity[
        qp_variables -> External(func:write-query-array(?name1
            External(func:URI2literal(?attr)) ?activity))
        qp_values -> External(func:write-query-array(?name2 ?value ?activity))])))
```

Listing D.12: R15 - Map product attribute to query parameter (set parameter; optional; process default)

```
If
    And(
    ?activity[type -> ?class
              hasInput -> ?input
              hasModel -> ?model]
    ?class[subClassOf -> ?restr1
           subClassOf -> ?restr2]
    ?restr1[type -> Restriction
            onProperty -> qp_variables
            minCardinality -> ?x1]
    ?restr2[type -> Restriction
```

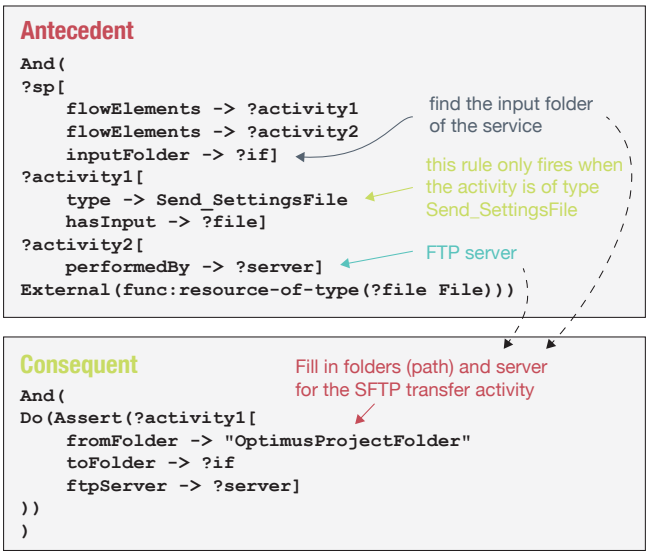


Figure D.7: R28 - Configure Send_SettingsFile

```
            onProperty -> qp_values
            minCardinality -> ?x2]
    qp_variables[label -> ?name1]
    qp_values[label -> ?name2]
    ?input[binding -> ?attr
           variable -> "false"
           required -> "false"
           processDefault -> ?value]
    ?model[?attr -> ?x]
    Not(
    ?input[parameterValue -> ?y]))
Then
    And(
    Do(Assert(?activity[
        qp_variables -> External(func:write-query-array(?name1
            External(func:URI2literal(?attr)) ?activity))
        qp_values -> External(func:write-query-array(?name2 ?value ?activity))])))
```

Listing D.13: R16 - Map required input parameters to query parameter (variable)

```
If
    And(
    ?activity[type -> ?class
              hasInput -> ?input]
    ?class[subClassOf -> ?restr]
    ?restr[type -> Restriction
           onProperty -> ?query
           cardinality -> ?x]
    ?query[label -> ?name
           expects -> ?valueof]
    ?valueof[fromSource -> ?source]
    ?input[hasParameterID -> ?upi
           variable -> "true"]
    ?source = ?input)
Then
    And(
    Do(Assert(?activity[?query -> External(func:write-query-parameter(?name
        External(func:dollarize(External(func:URI2literal(?upi))))))])))
```

Listing D.14: R17 - Map required input parameters to query parameter (required; fixed)

```
If
    And(
    ?activity[type -> ?class
              hasInput -> ?input]
    ?class[subClassOf -> ?restr]
    ?restr[type -> Restriction
           onProperty -> ?query
           cardinality -> ?x]
    ?query[label -> ?name
```

```
            expects -> ?valueof]
    ?valueof[fromSource -> ?source]
    ?source = ?input
    ?input[variable -> "false"
            parameterValue -> ?value])
Then
    And(
    Do(Assert(?activity[?query -> External(func:write-query-parameter(?name ?value))])))
```

Listing D.15: R20 - Bind product to activities

```
If
    And(
    ?problem[optimizes -> ?product
             solvedBy -> ?wf]
    ?wf[flowElements -> ?activity]
    ?activity[type -> ?class]
    ?class[subClassOf -> ?restr]
    ?restr[type -> Restriction
            onProperty -> hasModel
            valuesFrom -> ?source]
    External(func:resource-of-type(?product ?source)))
Then
    And(
    Do(Assert(?activity[hasModel -> ?product])))
```

Listing D.16: R22 - Design variable input (part 2)

```
If
    And(
    ?problem[hasDesVar -> ?desvar
             solvedBy -> ?wf]
    ?wf[flowElements -> ?activity]
    ?activity[type -> ?class]
    ?class[subClassOf -> ?restr
            subClassOf -> ?restr2]
    ?restr[type -> Restriction
            onProperty -> hasInput
            someValuesFrom -> ?input]
    ?desvar[sets -> ?par]
    External(func:resource-of-type(?desvar ?input))
    ?restr2[type -> Restriction
             onProperty -> hasOutput
             someValuesFrom -> ?output]
    ?output[subClassOf -> ?restr3
             subClassOf -> ?restr4]
    ?restr3[type -> Restriction
             onProperty -> binding
             valuesFrom -> ?odtp]
    External(func:resource-of-type(?par ?odtp))
    ?restr4[type -> Restriction
             onProperty -> hasParameterID
             valuesFrom -> ?upi])
Then
    And(
    Do(Assert(?activity[hasInput -> ?desvar
                        hasOutput -> External(func:new-individual(?output ?par))]))
    Do(Assert(External(func:new-individual(?output ?par))[
        type -> ?output
        binding -> ?par
        hasParameterID -> External(func:new-individual(?upi ?par))]))
    Do(Assert(External(func:new-individual(?upi ?par))[type -> ?upi])))
```

Listing D.17: R23 - Objective function input (part 1)

```
If
    And(
    ?rule[inConsequent -> ?problem
          inConsequent -> ?par]
    ?problem[objective_function -> ?formula]
    Not(
    ?par = ?problem)
    Not(
    ?par = objective_function)
    ?wf[solves -> ?problem
        flowElements -> ?activity]
    ?activity[type -> ?class]
    ?class = CheckFunctions
    ?class[subClassOf -> ?restr]
    ?restr[type -> Restriction
            onProperty -> hasInput
            someValuesFrom -> ?input]
    ?input[subClassOf -> ?restr2
            subClassOf -> ?restr3]
```

```
    ?restr2[type -> Restriction
            onProperty -> binding
            valuesFrom -> ?odtp]
    External(func:resource-of-type(?par ?odtp))
    ?restr3[type -> Restriction
            onProperty -> hasParameterID
            valuesFrom -> ?upi])
Then
    And(
    Do(Assert(?activity[hasInput -> External(func:new-individual(?input ?par))]))
    Do(Assert(External(func:new-individual(?input ?par))[
        type -> ?input
        binding -> ?par
        hasParameterID -> External(func:new-individual(?upi ?par))]))
    Do(Assert(External(func:new-individual(?upi ?par))[type -> ?upi])))
```

Listing D.18: R24 - Objective function input (part 2)

```
If
    And(
    ?rule[inConsequent -> ?problem
          inConsequent -> ?par]
    ?problem[objective_function -> ?formula]
    Not(
    ?par = ?problem)
    Not(
    ?par = objective_function)
    ?wf[solves -> ?problem
        flowElements -> ?activity]
    ?activity[type -> ?class]
    ?class[subClassOf -> ?restr]
    ?restr[type -> Restriction
           onProperty -> hasOutput
           someValuesFrom -> ?output]
    ?output[subClassOf -> ?restr2
            subClassOf -> ?restr3]
    ?restr2[type -> Restriction
            onProperty -> binding
            valuesFrom -> ?odtp]
    External(func:resource-of-type(?par ?odtp))
    ?restr3[type -> Restriction
            onProperty -> hasParameterID
            valuesFrom -> ?upi])
Then
    And(
    Do(Assert(?activity[hasOutput -> External(func:new-individual(?output ?par))]))
    Do(Assert(External(func:new-individual(?output ?par))[
        type -> ?output
        binding -> ?par
        hasParameterID -> External(func:new-individual(?upi ?par))]))
    Do(Assert(External(func:new-individual(?upi ?par))[type -> ?upi])))
```

Listing D.19: R25 - Constraint input (part 1)

```
If
    And(
    ?rule[inConsequent -> ?constr
          inConsequent -> ?par]
    ?constr[equation -> ?formula]
    Not(
    ?par = ?constr)
    Not(
    ?par = equation)
    ?problem[hasConstraint -> ?constr]
    ?wf[solves -> ?problem
        flowElements -> ?activity]
    ?activity[type -> ?class]
    ?class = CheckFunctions
    ?class[subClassOf -> ?restr]
    ?restr[type -> Restriction
           onProperty -> hasInput
           someValuesFrom -> ?input]
    ?input[subClassOf -> ?restr2
           subClassOf -> ?restr3]
    ?restr2[type -> Restriction
            onProperty -> binding
            valuesFrom -> ?odtp]
    External(func:resource-of-type(?par ?odtp))
    ?restr3[type -> Restriction
            onProperty -> hasParameterID
            valuesFrom -> ?upi])
Then
    And(
    Do(Assert(?activity[hasInput -> External(func:new-individual(?input ?par ))]))
    Do(Assert(External(func:new-individual(?input ?par))[
        type -> ?input
```

```
        binding -> ?par
        hasParameterID -> External(func:new-individual(?upi ?par ))]))
    Do(Assert(External(func:new-individual(?upi ?par))[type -> ?upi])))
```

Listing D.20: R26 - Constraint input (part 2)

```
If
    And(
    ?rule[inConsequent -> ?constr
          inConsequent -> ?par]
    ?constr[equation -> ?formula]
    Not(
    ?par = ?constr)
    Not(
    ?par = equation)
    ?problem[hasConstraint -> ?constr]
    ?wf[solves -> ?problem
        flowElements -> ?activity]
    ?activity[type -> ?class]
    ?class[subClassOf -> ?restr]
    ?restr[type -> Restriction
           onProperty -> hasOutput
           someValuesFrom -> ?output]
    ?output[subClassOf -> ?restr2
            subClassOf -> ?restr3]
    ?restr2[type -> Restriction
            onProperty -> binding
            valuesFrom -> ?odtp]
    External(func:resource-of-type(?par ?odtp))
    ?restr3[type -> Restriction
            onProperty -> hasParameterID
            valuesFrom -> ?upi])
Then
    And(
    Do(Assert(?activity[hasOutput -> External(func:new-individual(?output ?par))]))
    Do(Assert(External(func:new-individual(?output ?par))[
        type -> ?output
        binding -> ?par
        hasParameterID -> External(func:new-individual(?upi ?par))]))
    Do(Assert(External(func:new-individual(?upi ?par))[type -> ?upi])))
```

Listing D.21: R29 - Configure Retrieve_SimulationResults

```
If
    And(
    ?sp[flowElements -> ?activity1
        flowElements -> ?activity2
        outputFolder -> ?of]
    ?activity1[type -> Retrieve_SimulationResults
               hasInput -> ?file]
    ?activity2[performedBy -> ?server]
    ?server[rootFolder -> ?root]
    External(func:resource-of-type(?file File)))
Then
    And(
    Do(Assert(?activity1[fromFolder -> ?of
                         toFolder -> "OptimusProjectFolder"
                         ftpServer -> ?server])))
```

Appendix E

Extension to Use Case 2: KB-PIDO Coupling

This appendix contains a Discipline-, Activity-, and Tool-form for the packaging optimisation use case (Chapter 7). It serves as an example for how to fill in the new forms.

- The D-form captures information about the discipline *Product definition* (see Figure E.1).
- The A-form describes the HLA *Interact with MMG* (see Figure E.2).
- And the T-form describes the tool/service that is used to build the product model (see Figure E.3).

Furthermore, three formulas have been modelled for this use case (see index in Table E.1). These are included in Listings E.1-E.3.

MOKA2 form:	Discipline					
Name	Product definition					
Reference	D1					
Objective	Create a KBE product model of the packaging					
Description	The KBE application is built for packaging design optimisation					
Input requirements/preconditions	-					
Context, information, validity	Packaging design					
Problem involved	P1					
Activities involved	A1.2					
Entities involved	E1					
Parameters						
Name	length	cost/m2 top	cost	width	height	cos
Description	The length of the packaging	The cost per m2 for the top surface	The cost of the packaging	The width of the packaging	The height of the packaging	The bot
Input/Output	Input	Input	Output	Input	Input	Inp
Variable/Fixed	Variable	Fixed		Variable	Variable	Fix
Parameter Value	-	-	-	-	-	-
Default value (process)	-	-	-	-	-	-
Defautl value (tool)	-	10	-	-	-	
Source	Design variable	user input	-	Design variable	Design variable	use
Management						
Author	P. Chan					
Date	24-02-2013					
Version number	1					
Status	In Progress					
Modification						
Information origin	P. Chan					

Figure E.1: Example of a D-form for the packaging design problem.

MOKA2 form:	Activity					
Name	Interact with MMG					
Reference	A1.2					
Objective	Update the KBE product model and retrieve responses.					
Description	Update the KBE product model and retrieve responses.					
Trigger	-					
Input requirements/preconditions	-					
Context, information, validity	Packaging design					
Potential failure modes	-					
Executed by	GenDL Packaging Model (T1)					
Related activities						
Parent activity	MyWorkflow (A1)					
Sub activities	Send request (A1.2.1); Parse response (A1.2.2)					
Preceding activities	Set initial values (A1.1)					
Following activities	Check functions (A1.3)					
Rules involved	-					
Entities involved	Packaging (E1)					
Parameters						
Name	length	width	volume	cost/m2 bottom	height	cost/m2 top
Description	The length of the packaging	The width of the packaging	The volume of the packaging	The cost per m2 for the bottom surface	The height of the packaging	The cost per m2 for the top surface
Input/Output	Input	Input	Output	Input	Input	Input
Variable/Fixed	Variable	Variable		Fixed	Variable	Fixed
Parameter Value	-	-	-	-	-	-
Default value (process)	-	-	-	-	-	-
Default value (tool)	-	-	-	20	-	10
Source	Design variable	Design variable	-	user input	Design variable	user input
Management						
Author	P. Chan					
Date	24-02-2013					
Version number	1					
Status	In Progress					
Modification						
Information origin	P. Chan					

Figure E.2: Example of an A-form for the packaging design problem.

Listing E.1: F1 - Objective function: MyOptimization

```
If
Then
    And(
    Do(Assert(MyOptimization[objective_function -> "<math xmlns=\"http://www.w3.org/1998/Math/
        MathML\" title=\"Cost objective\"><ci definitionURL=\"http://www.lr.tudelft.nl/KBE/
        product/formal/container#container.cost\">cost</ci></math>"])))
```

Table E.1: Formulas index - packaging use case

ID	Domain	Title
F1	Formula	Objective function: MyOptimization
F2	Formula	Constraint: Volume
F3	Formula	Constraint: Weight

Listing E.2: F2 - Constraint: Volume

```
If
Then
    And(
    Do(Assert(VolumeConstraint[
        equation -> "<math xmlns=\"http://www.w3.org/1998/Math/MathML\" title=\"Volume
        constraint\"><apply><eq/><ci definitionURL=\"http://www.lr.tudelft.nl/KBE/product/
        formal/container#container.volume\">V</ci><cn>4</cn></apply></math>"])))
```

Listing E.3: F3 - Constraint: Weight

```
If
Then
    And(
    Do(Assert(WeightConstraint[
        equation -> "<math xmlns=\"http://www.w3.org/1998/Math/MathML\" title=\"Weight
        constraint\"><apply><lt/><apply><times/><ci definitionURL=\"http://www.lr.tudelft.nl/
        KBE/product/formal/container#container.density\">$RHO$</ci><ci definitionURL=
        \"http://www.lr.tudelft.nl/KBE/product/formal/container#container.volume\">V</ci>
        </apply><cn>5000</cn></apply></math>"])))
```

MOKA2 form:	Tool					
Name	GenDL Packaging Model					
Reference	T1					
Software application	GenDL 1582					
Context	KBE					
Objective	Create a KBE product model of the packaging					
Description	The KBE application is built for packaging design optimisation					
Idealisations and limitations	-					
Input requirements/preconditions	-					
Parameters						
Name	length	width	height	cost/m2 top	cost/m2 bottom	cost/m2 sides
Description	The length of the packaging	The width of the packaging	The height of the packaging	The cost per m2 for the top surface	The cost per m2 for the bottom surface	The cost per m2 for the side surfaces
Input/Output	Input	Input	Input	Input	Input	Input
Required/Optional	Required	Required	Required	Optional	Optional	Optional
Default value	-	-	-	10	20	30
Data type	double	double	double	double	double	double
Management						
Author	P. Chan					
Date	24-02-2013					
Version number	1					
Status	In Progress					
Modification						
Information origin	P. Chan					
MOKA DPM form:	Service					
URL	http://127.0.0.1:9000/set-get					
Message format	REST query					
Example request (query string format)	http://127.0.0.1:9000/set-get?iid=packaging&variables(0)=length&variables(1)=width&variables(2)=height&values(0)=1&values(1)=1&values(2)=3&responses(0)=cost&responses(1)=volume					
Example response	cost 390 volume 3					

Figure E.3: Example of a T-form for the packaging design problem.

Appendix F

Extension to Use Case 3: MDO Workflows

This appendix is an extension of Chapter 8. It contains additional information about inputs and outputs of the following HLAs:

- Calculate material usage (Table F.1)
- Calculate cost (Table F.2)
- Retrieve LCA data (Figures F.1 and F.2)

Furthermore, six formulas have been modelled for this use case (see index in Table F.3). These are included in Listings F.1-F.6.

Table F.1: Outputs for *"Calculate material usage"*

Output	File parameter settings					
	label	word nr	occurrence	row offset	start pos	end pos
Total mass of used material per cycle	material-mass-per-cycle	-	-	-	-	-
Total mass of used material (all parts)	material-mass-total	-	-	-	-	-

Table F.2: Outputs for *"Calculate cost"*

Output	File parameter settings					
	label	word nr	occurrence	row offset	start pos	end pos
Total cost	cost-total	-	-	-	-	-
Cost without cooling system	cost-without-cooling-system	-	-	-	-	-
Cost of cooling system	cost-cooling-system	-	-	-	-	-
Cost of mould without cooling system	cost-mold-without-cooling-system	-	-	-	-	-
Cost of part	cost-part	-	-	-	-	-
Cost of mould maintenance	cost-mold-maintenance	-	-	-	-	-
Cost of mass production	cost-mass-production	-	-	-	-	-
Cost of material	cost-material	-	-	-	-	-
Cost of defects	cost-defects	-	-	-	-	-
Cost of manufacturing cooling system	cost-cooling-system-manufacturing	-	-	-	-	-
Cost of energy usage in moulding process	cost-energy-molding	-	-	-	-	-

Parameter	Input values	Unit
Quantity	Any amount	kg
Composition of plastic/ blend / compound	ABS	%
	Epoxy Resin	%
	EPS	%
	PA6	%
	PA66	%
	PBT	%
	PC	%
	PEHD	%
	PELD	%
	PET	%
	PLA	%
	PMMA	%
	Polyester Resin	%
	POM	%
	PP	%
	PS	%
	PUR, flexible foam	%
	PUR, granulates	%
	PUR, rigid foam	%
	PVC	%
	SEBS	%
	Carbon black	%
	Carbon fiber	%
	Compatibilizer (maleic anhydride)	%
	EPDM	%
	Glass fiber	%
	Phtalate plasticizer	%
	Pigments (aluminium traces)	%
	Starch	%
	Talcum	%

Figure F.1: List of inputs for the LCA web service. This first table contains parameters about material composition.

Parameter	Input values	Unit
Injection moulding process (without energy consumption and waste management)[a]	Any amount	kg[b]
Electricity mix	0 (Europe)	/
	1(France)	/
	2(Germany)	/
	3(Great Britain)	/
	4(Italy)	/
	5(Netherlands)	/
	6(Portugal)	/
	7(Spain)	/
	8(Switzerland)	/
	9(USA)	/
	10(China)	/
Electricity	Any amount	kWh/kg[b]
Natural gas	Any amount	MJ/kg[b]
Heavy fuel oil	Any amount	MJ/kg[b]
Light fuel oil	Any amount	MJ/kg[b]
Scrap	Any amount	%
Scrap management	1 (incineration)	/
	2 (landfilling)	/
	3 (recycling)	/
	4 (reuse in production)	/

Figure F.2: List of the remaining inputs of the LCA web service. These inputs are related to energy consumption and waste management.

Table F.3: Formulas index - MDO use case

ID	Domain	Title
F4	Formula	Formula: Cooling time
F5	Formula	Formula: Max clamp force
F6	Formula	Formula: Max abs warpage
F7	Formula	Formula: Total CO2
F8	Formula	Constraint: Max abs warpage
F9	Formula	Constraint: Max clamp force

Listing F.1: F4 - Formula: Cooling time

```
If
Then
    And(
    Do(Assert(CalculateCoolingTime[formula -> "<math title=\"Cooling time\" xmlns=\"http://www.
        w3.org/1998/Math/MathML\"><apply><minus/><ci definitionURL=\"http://www.lr.tudelft.nl/
        swfm-user.owl#CycleTime\">CycleTime</ci><ci definitionURL=\"http://www.lr.tudelft.nl/
        swfm-user.owl#FillingTime\">FillingTime</ci></apply></math>"])))
```

Listing F.2: F5 - Formula: Max clamp force

```
If
Then
    And(
    Do(Assert(CalculateMaxClampForce[formula -> "<math xmlns=\"http://www.w3.org/1998/Math/
        MathML\" title=\"Max clamp force\"><apply><divide/><apply><max/><ci definitionURL=
        \"http://www.lr.tudelft.nl/swfm-user.owl#PackingClampForce\">PackingClampForce</ci>
        <ci definitionURL=\"http://www.lr.tudelft.nl/swfm-user.owl#FillingClampForce\">
        FillingClampForce</ci></apply><cn>9810</cn></apply></math>"])))
```

Listing F.3: F6 - Formula: Max abs warpage

```
If
Then
    And(
    Do(Assert(CalculateMaxAbsWarpage[formula -> "<math title=\"Max abs warpage\" xmlns=\"http://
        www.w3.org/1998/Math/MathML\"><apply><max/><apply><abs/><ci definitionURL=\"http://www.
        lr.tudelft.nl/swfm-user.owl#MinDispX\">MinDispX</ci></apply><apply><abs/><ci
        definitionURL=\"http://www.lr.tudelft.nl/swfm-user.owl#MaxDispX\">MaxDispX</ci></apply>
        <apply><abs/><ci definitionURL=\"http://www.lr.tudelft.nl/swfm-user.owl#MinDispY\">
        MinDispY</ci></apply><apply><abs/><ci definitionURL=\"http://www.lr.tudelft.nl/
        swfm-user.owl#MaxDispY\">MaxDispY</ci></apply><apply><abs/><ci definitionURL=\"http://
        www.lr.tudelft.nl/swfm-user.owl#MinDispZ\">MinDispZ</ci></apply><apply><abs/><ci
        definitionURL=\"http://www.lr.tudelft.nl/swfm-user.owl#MaxDispZ\">MaxDispZ</ci></apply>
        </apply></math>"])))
```

Listing F.4: F7 - Formula: Total CO2

```
If
Then
    And(
    Do(Assert(CalculateTotalCO2[formula -> "<math title=\"Total CO2\" xmlns=\"http://www.w3.org/
        1998/Math/MathML\"><apply><plus/><ci definitionURL=\"http://www.lr.tudelft.nl/
        swfm-user.owl#CO2Emission_Material\">CO2Material</ci><ci definitionURL=\"http://www.lr.
        tudelft.nl/swfm-user.owl#CO2Emission_Process\">CO2Process</ci></apply></math>"])))
```

Listing F.5: F8 - Constraint: Max abs warpage

```
If
Then
    And(
    Do(Assert(WarpageConstraint[equation -> "<math title=\"Max abs warpage constraint\" xmlns=
        \"http://www.w3.org/1998/Math/MathML\"><apply><leq/><ci definitionURL=\"http://www.lr.
        tudelft.nl/swfm-user.owl#MaxAbsWarpage\">MaxAbsWarpage</ci><cn>10</cn></apply></math>"])
    ))
```

Listing F.6: F9 - Constraint: Max clamp force

```
If
Then
    And(
    Do(Assert(ClampForceConstraint[equation -> "<math title=\"Max clamp force constraint\"
        xmlns=\"http://www.w3.org/1998/Math/MathML\"><apply><leq/><ci definitionURL=\"http://
        www.lr.tudelft.nl/swfm-user.owl#MaxClampForce\">MaxClampForce</ci><cn>100</cn></apply>
        </math>"])))
```