

# Collaborative Knowledge Based Engineering

San Kilis

November 23<sup>rd</sup>, 2022



# Collaborative Knowledge Based Engineering

**San Kilkis**

to obtain the degree of Master of Science  
at the Delft University of Technology,  
to be defended publicly on November 29, 2022 at 2:00 PM.

Project duration:	October 1, 2020 – November 29, 2022	
Thesis committee:	Dr. ir. Gianfranco La Rocca,	TU Delft, supervisor
	Ir. Reinier van Dijk,	ParaPy B.V., supervisor
	Dr. ir. Maurice F. M. Hoogreef,	TU Delft
	Dr. Christoph Lofi	TU Delft

*This thesis is confidential and cannot be made public until November 29, 2024.*

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



# Preface

This journey started with my desire to combine my dream of software development with my graduate studies. I am eternally thankful for the opportunity to have done this thesis project at ParaPy. I will cherish the times that I was able to spar my ideas with Dr. ir. Gianfranco La Roca and ir. Reinier van Dijk. Even though the thesis itself has been extremely challenging, I am excited to be in a position where I can use the skills that I have learned during this period to contribute to the ParaPy platform.

I would like to thank both Dr. ir Maurice FM Hoogreef and Dr. Christoph Lofi for their valuable time as part of my jury for the defense. Furthermore, I would like to personally thank all my colleagues at ParaPy, Max, Colin, Anton, Jelle, Brendan and Luc. During the stressful moments of my thesis work, it was always a joy to discuss what is going on over coffee. Finally, I thank all of my friends who have been so supportive during the thesis and throughout my academic life, Bryan, Victor, Amit, Kostas, Kilian, Michael, Miha, Yvonne, Frans, Femke, Floris, Daniel, Rudy, Tejas, and Julien.

Special thanks to my dear family, Siir, Umit, Birol whose loving support has made me the person I am today. I am also very grateful for the support and love that I have received over the past 2 years from my lovely girlfriend, Rosalie, as well as her family. I cannot wait to celebrate this result with all of you!

*San Kilkis  
November 23, 2022*

# Abstract

Present Knowledge Based Engineering (KBE) applications do not facilitate collaborative processes due to lack of process formalization and orchestration capabilities, leading to ad-hoc solutions. This increases application development time, reduces re-usability, and increases the cognitive burden of users leading to decreased design efficiency. Generic solutions from Workflow Management (WfM) can capture process knowledge in models outside of the KBE application and enable better collaboration. However, a challenge is to maintain flexibility when using WfM, which is rigid compared to other groupware. The research work of this thesis contributes technologies for saving and accessing the information of a KBE application to enable a methodology that flexibly facilitates collaboration by taking advantage of runtime caching and dependency tracking to dynamically populate tasks based on runtime context. These contributions are highlighted through case studies that demonstrate how present limitations of KBE and WfM can be overcome to open a new frontier for the next-generation of engineering software. Based on a verified software prototype, the key research contributions of this thesis are four-fold: (a) development of an information modeling approach making use of GraphQL to flexibly query KBE models to support collaborative activities, (b) development of a generic persistence capability for ParaPy to support transactional usage and retain full design history, (c) implementation of the worklet concept in KBE to increase flexibility of workflows and provide new form of process automation, and (d) creation of correlated dependency technique which has potential for use on emergent workflows. These contributions are paramount to undertaking multiple opportunities for collaborative KBE applications. It is expected that these research contributions provide a theoretical basis for achieving improved collaborative usage of KBE applications.

# Table of Contents

<b>Preface</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>List of Symbols</b>	<b>v</b>
<b>List of Figures</b>	<b>vi</b>
<b>List of Tables</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Problem Statement . . . . .	2
1.2 Research Objective & Questions . . . . .	4
1.3 Thesis Structure . . . . .	4
<b>2 Theoretical Content</b>	<b>5</b>
2.1 Knowledge Based Engineering . . . . .	5
2.1.1 Core Technologies . . . . .	6
2.1.2 Limitations . . . . .	7
2.1.3 Relevant Literature . . . . .	7
2.2 Workflow Management . . . . .	9
2.2.1 Task Definition . . . . .	9
2.2.2 Fault Tolerance . . . . .	10
2.2.3 Control Flow vs. Data Flow . . . . .	10
2.2.4 Historical Perspective . . . . .	11
2.2.5 Specialized Systems to Automate Workflows . . . . .	12
2.2.6 Modeling Techniques . . . . .	13
2.2.7 Limitations & Research Trends . . . . .	15
<b>3 Methodology</b>	<b>16</b>
3.1 Experimental Set-up . . . . .	16
3.2 Information Modeling . . . . .	17
3.2.1 Schema Autogeneration . . . . .	19
3.2.2 Modeling Approach . . . . .	21
3.3 Model Persistence . . . . .	22
3.3.1 Persistence Architecture . . . . .	23
3.3.2 Handling Geometry . . . . .	23
3.3.3 Graph Contraction . . . . .	24
3.3.4 Consistency Maintenance . . . . .	25
3.4 Process Orchestration . . . . .	26
3.4.1 Process Modelling . . . . .	27
3.4.2 Service Architecture . . . . .	28
3.4.3 Correlated Dependencies . . . . .	29
3.4.4 Data Management . . . . .	30
3.4.5 Handling User Tasks . . . . .	31
3.4.6 Dynamic Workflow . . . . .	33
3.4.7 Emergent Workflow . . . . .	34

<b>4 Results</b>	<b>38</b>
4.1 Information Modeling . . . . .	38
4.2 Model Persistence . . . . .	40
4.3 Process Orchestration . . . . .	42
4.3.1 Case Study: Earthquake Analysis . . . . .	42
4.3.2 Case Study: Hot Air Balloon . . . . .	43
4.4 Verification . . . . .	45
<b>5 Discussion</b>	<b>46</b>
5.1 Information Modeling . . . . .	46
5.2 Model Persistence . . . . .	46
5.3 Process Orchestration . . . . .	47
5.4 Answers to Research Questions . . . . .	49
5.5 Research Contributions . . . . .	50
5.6 Limitations & Future Work . . . . .	50
<b>6 Conclusion</b>	<b>51</b>
<b>References</b>	<b>52</b>
<b>Appendix A Market Studies</b>	<b>58</b>
A.1 Knowledge Based Engineering . . . . .	58
A.2 Workflow Management . . . . .	59
A.3 Web Application Programming Interfaces . . . . .	60
<b>Appendix B Information Modeling</b>	<b>61</b>
<b>Appendix C Example Queries</b>	<b>65</b>
<b>Appendix D Examples of Dynamism</b>	<b>72</b>
<b>Appendix E Algorithm Formalization</b>	<b>74</b>

# List of Symbols

## Abbreviations

<b>BPM</b>	Business Process Management
<b>BPMN</b>	Business Process Model and Notation
<b>BPMS</b>	Business Management Systems
<b>CAD</b>	Computer Aided Design
<b>CE</b>	Collaborative Engineering
<b>CS</b>	Computer Science
<b>DAG</b>	Directed Acyclic Graph
<b>DBD</b>	Decision Based Design
<b>DIKW</b>	Data, Information, Knowledge, Wisdom
<b>DoE</b>	Design of Experiment
<b>ECM</b>	Engineering Change Management
<b>EPCs</b>	Event-Process Chains
<b>ETL</b>	Extract, Transform, Load
<b>FILO</b>	First In Last Out
<b>HDOT</b>	Hinge-System Design and Optimization Tool
<b>HESs</b>	High-Level Engineering Services
<b>HLAs</b>	High-Level Activities
<b>HPC</b>	High Performance Computing
<b>iProd</b>	Integrated Management of Product Heterogenous Data
<b>KBE</b>	Knowledge Based Engineering
<b>KBS</b>	Knowledge Based Systems
<b>KM</b>	Knowledge Management
<b>MDO</b>	Multi-Disciplinary Optimization
<b>MMG</b>	Multi-Model Generator
<b>MTO</b>	Made-to-Order
<b>OCP</b>	Open Closed Principle
<b>OMG</b>	Object Management Group
<b>OOP</b>	Object-Oriented Programming
<b>PDP</b>	Product Development Process
<b>PIDO</b>	Process Integration and Design Automation
<b>RPC</b>	Remote Procedure Call
<b>SOA</b>	Service Oriented Architecture
<b>SSOT</b>	Single Source of Truth
<b>SWfMSs</b>	Scientific Workflow Management Systems
<b>SWT</b>	Semantic Web Technologies
<b>TDD</b>	Test-Driven Development
<b>WfM</b>	Workflow Management

# List of Figures

1.1	Current vs. Next Generation Process Modelling Approaches for KBE . . . . .	2
1.2	Current vs. Next Generation Approach for Earthquake Analysis Process . . . . .	3
1.3	The Collaborative Work Spectrum, Adapted from . . . . .	3
2.1	Data, Information, Knowledge, Wisdom (DIKW) Taxonomy . . . . .	5
2.2	ACID Transaction Properties Adapted to Reflect Task Boundary . . . . .	10
2.3	Workflow Execution Models Categorized by Control and Data Flow . . . . .	11
2.4	BPMN Diagram of a Recruiting Process at Strategic Level . . . . .	14
3.1	Example of a Selection Set in a Query to Obtain Aircraft Data . . . . .	18
3.2	UML Class/Object Diagram Depicting Relationship Between ParaPy & GraphQL . . . . .	18
3.3	Visualization of Origin Type Discovery Algorithm . . . . .	19
3.4	Example of When Type Inferencing is Required . . . . .	20
3.5	Example of the GraphQL Schema Modeling Approach with Overrides . . . . .	21
3.6	UML Diagram of Object and Dependency Graph for an Airfoil . . . . .	22
3.7	Depiction of Untraversed Sub-Graph Requiring Contraction . . . . .	24
3.8	Wing Construction Methodology Used in the MMG . . . . .	25
3.9	Visualization of Consistency Maintenance Algorithm . . . . .	26
3.10	Types of Dynamism in Workflows . . . . .	26
3.11	Tipping Point Between KBE Assisted and KBE Controlled Workflows . . . . .	27
3.12	Screenshot of Process Modeling / Formalization in Camunda Modeler . . . . .	28
3.13	Screenshots of Task Template Implementation to Assist Process Modeling . . . . .	28
3.14	Service Architecture of KBE-WfM Synthesized Software . . . . .	29
3.15	Depiction of How a Correlated Dependency Transaction is Executed . . . . .	30
3.16	Example Process Variables from a Process Instance . . . . .	31
3.17	Comparison of Generic KBE User Interface vs. Task Specific Interface . . . . .	31
3.18	UML Sequence Diagram Depicting User Task Handling . . . . .	32
3.19	Depiction of How a User Task Subscription is Launched . . . . .	32
3.20	Worklet Decorator Developed for the ParaPy SDK . . . . .	33
3.21	BPMN Pattern Created to Enable the Worklet Concept . . . . .	34
3.22	Correlated Dependency Concept for Emergent Workflow Orchestration . . . . .	35
3.23	UML Sequence Diagram Depicting Worklet Execution . . . . .	36
3.24	Process Model of a Generic Tool Transformation . . . . .	37
4.1	Schema Transpilation Performance Measured on Example KBE Applications . . . . .	39
4.2	KBE Applications Used to Test Model Persistence . . . . .	40
4.3	Synthetic KBE Model Serialization Benchmark Results . . . . .	41
4.4	Earthquake Analysis Workflow BPMN Diagram . . . . .	42
4.5	User Interface of the Monolithic Hot Air Balloon Application . . . . .	43
4.6	Hot Air Balloon Case Study BPMN Diagram . . . . .	43
4.7	Segmentation of Hot Air Balloon Design Process into Worklets . . . . .	44
4.8	KBE Controlled Hot Air Balloon Case Study BPMN Diagram . . . . .	44
5.1	Differences in Iterative Usage on Static vs. Dynamic Workflows . . . . .	47
5.2	Hot Air Balloon Workflow Status, Audit Log, & Process Variables . . . . .	48
5.3	Goals, Enablers, Results of the Conducted Research . . . . .	50
A.1	Timeline of Popular API Description Languages . . . . .	60
B.1	Comparison of Source Code and Transpiled GraphQL Schema . . . . .	61



B.2	Transpiled Hot Air Balloon Schema Visualized in GraphQL Voyager . . . . .	63
B.3	Profiling Result of Warehouse Application (Without Inferencing) . . . . .	64
B.4	Profiling Result of Warehouse Application (With Inferencing) . . . . .	64
C.1	Query Formulation for Requesting a Boolean Value Slot . . . . .	65
C.2	Mutation Formulation for Updating a Boolean Value Slot . . . . .	65
C.3	Query Formulation for Accessing Dependency Information . . . . .	66
C.4	Query Formulation for a Requesting a Dynamic Type . . . . .	67
C.5	Query Formulation for Accessing Items of a Sequence . . . . .	68
C.6	Mutation Formulation for Updating an Item in a Sequence . . . . .	69
C.7	Query Formulation for Accessing Items of a Dynamic Sequence . . . . .	70
C.8	Query Formulation for Accessing Geometry . . . . .	71
D.1	Example of a Dynamic Selection of Assignee (Who) . . . . .	72
D.2	Example of a Dynamic Selection of Worklet (What) . . . . .	72
D.3	Example of Dynamic Task Skipping (What) . . . . .	73
D.4	Example of Dynamic Timing (When) . . . . .	73
D.5	Example of a Dynamic Task Selection (How) . . . . .	73
E.1	Graph Contraction Algorithm . . . . .	74
E.2	Graph Contraction Algorithm . . . . .	75
E.3	Visualization of Graph Contraction Algorithm . . . . .	76
E.4	Wing Model Dependencies Before and After Graph Contraction . . . . .	77

# List of Tables

2.1	Categorization of Specialized WfMSs Based on Role and Relevance . . . . .	12
2.2	Modeling Techniques for Each Dominant Business Process Perspective . . . . .	14
4.1	KBE Serialization Performance on Real Use Cases . . . . .	40
A.1	Market Study of Commercial KBE Systems with Coupled Integration . . . . .	58
A.2	Survey of Open-Source Workflow Management Systems . . . . .	59

# 1 Introduction

Modern engineering projects are challenging because of the need to coordinate geographically distributed resources. Globalization and market liberalization have led to more direct competition, which has driven the need for designs of higher performance and quality [1]. However, these designs require more development effort to achieve; thus, new paradigms are required to support them. Collaborative Engineering (CE) is the application of collaboration sciences to engineering, allowing multiple stakeholders to work together across geographic, disciplinary, temporal, and cultural boundaries [2, p.176]. However, a higher coordination overhead is required to achieve collaboration [3], placing more burden on the Information Communication Technology (ICT) infrastructure to support the design process.

In aerospace, a paradigm shift is underway to improve aircraft design by making use of advances in computing technology to meet the challenges of increasingly stringent safety, health, environmental, economic, and operational requirements [4, p.2]. Software needed for this change must cope with unique challenges posed by aircraft design due to increasing complexity, long product lifecycles, and high volumes of data [5, p.1055]. At present, a single tool that can automate the design process, its knowledge, and data, is not yet available, which results in a patchwork of specialized tools that are understood by few and supported by even fewer; leading to misinformed decisions due to inconsistent data [6]. Consequently, a demand emerges for software that enables effective collaboration by providing an information backbone while simultaneously helping to coordinate tasks. In response, the sponsor of this research, ParaPy, has set out to improve their novel Knowledge-Based Engineering (KBE) platform to meet this demand.

KBE facilitates automation of highly repetitive tasks efficiently—through the core technologies of lazy evaluation, runtime caching, and dependency tracking—enabling the definition of adaptable parametric products that elegantly respond to design changes. These core technologies maintain consistency across multiple disciplinary views of a product [7, p.8], reduce human errors, and enable the exploration of more what-if scenarios by accelerating the design process [8, p.336]. Thus, this thesis focuses on exploring if it can be used to assist collaboration by maintaining consistency and making quicker product iterations. However, the initial vision of using KBE software to automate repetitive tasks, often produces monolithic applications that do not sufficiently facilitate collaborative processes, confining their usage to small teams [9].

This thesis stipulates that the reason is the lack of a “top-down” overview of tasks that represents when interactions between the application, tools, and humans should take place. This is highlighted by Figure 1.1a, where a knowledge engineer develops a KBE application according to an envisioned process model without formalizing it. While the declarative product modeling of KBE systems allows the process to be encoded implicitly through dependency relationships—managing product complexity—it causes the process to emerge at runtime through demand-driven reasoning. Combined with the lack of process formalization, it leaves users in the dark as to what *will* execute and increases the cognitive burden of engineers by requiring them to reconstruct a mental-model of the process to know when to manually notify others.

Addressing these limitations falls within the domain of Workflow Management (WfM), which has evolved to facilitate collaboration in the face of long-running tasks where delays caused by people or external processes are present. Over time, the scope of WfM increased to manage processes across heterogeneous systems and organizational boundaries reliably [10, p.18-19]; rebranding into Business Process Management (BPM) products [11, p.3].

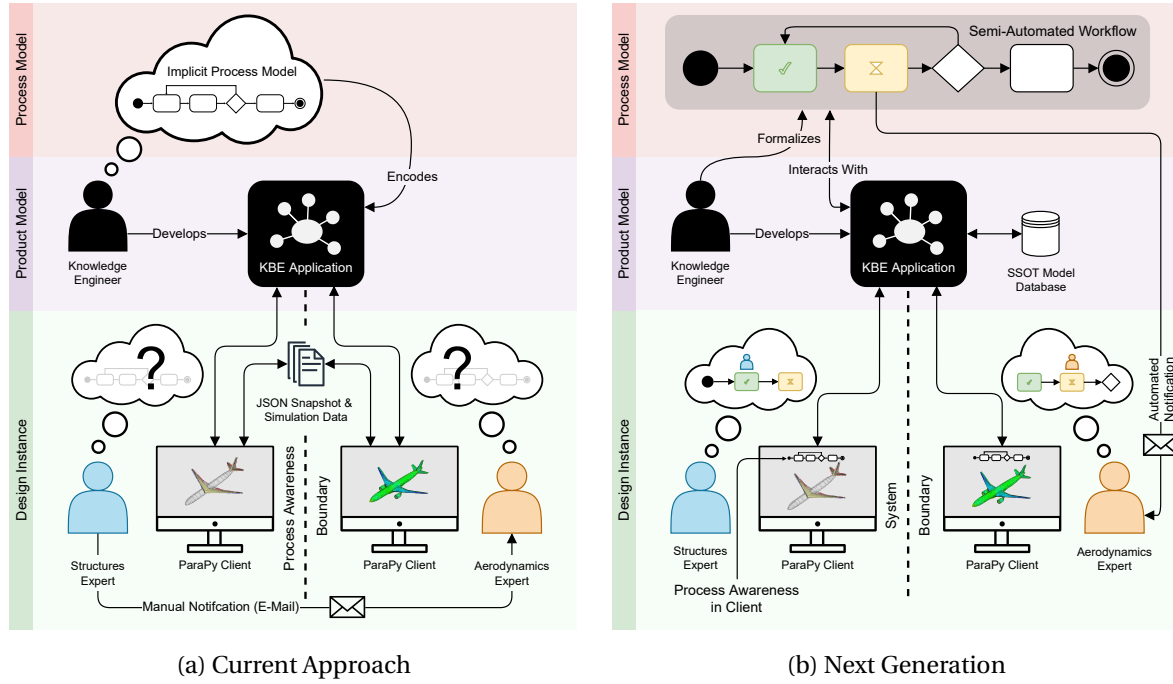


Figure 1.1: Current vs. Next Generation Process Modelling Approaches for KBE

At present WfM is underutilized in engineering due to the rigidity of dictating a “top-down” process on ad-hoc design activities [12]. Hence, there is a derived challenge to be able to maintain the dynamism of KBE applications, while integrating WfM techniques. This thesis then aims to synthesize the product modelling capability of KBE and the process modelling of WfM technology in an effort to overcome their respective limitations. This next generation approach, given by Figure 1.1b, envisions that a knowledge engineer formalizes intended process models that then interact with the KBE application. The synthesized software is also capable of exploiting the demand-driven behavior of KBE to dynamically “trigger” workflows. It is expected that doing so will reduce the rigidity of describing the entire process “top-down” while simultaneously alleviating the current lack of process awareness in KBE applications by automatically notifying users of pending tasks and providing visual feedback through process models.

## 1.1 Problem Statement

Lacking process orchestration capabilities in KBE, the current generation of applications suffer from: (a) increased development time, (b) reduced reusability, and (c) increased cognitive burden for users. These detrimental problems can best be explained through examination of a real world KBE application used within a collaborative process. The purpose of the application is to analyze the structural integrity of houses with respect to earthquakes to be able to determine if structural stiffness need to be applied. Overall, the analysis starts with a user responsible for inputting the geometry of the house to the KBE application. Afterwards, another user is responsible for modeling the foundation of the house. Subsequently, one user continues creating a Finite Element Model (FEM) of the house, while another performs an analysis to derive the ground properties of the house. Finally, once all users have implemented their tasks, the FEM simulation is run, and its results are used while generating a report.

In the current approach given by Figure 1.2a, a monolithic KBE application is used to facilitate this process. However, without process model to guide them, the four specialists involved experience a cognitive burden due to needing to understand how to use the monolithic application, while also

needing to coordinate when to do their respective tasks amongst each other. This can be seen as a loss of the “big picture” which leads designers to having a lack of awareness of: (a) tasks that need to be done, (b) the history of information, (c) how information is consumed, and (d) changes to processes [13, p.258]. The attempt to support this collaborative process without process orchestration has also made this particular application bulky. Not only have developers needed to provide more documentation on how the application should be used, but they have also needed to resort to ad-hoc solutions to implement features such as model persistence and geometry checking. Integration of the latter within the application also means that such a capability to check geometry becomes less re-usable. Ultimately, these ad-hoc solutions also increase application development time. Instead, it is desired to be able to orchestrate several tools and KBE applications flexibly within a formalized process to take advantage of re-usable services while benefiting from generic solutions of WfM for process orchestration. The desired next generation approach is given by Figure 1.2b.

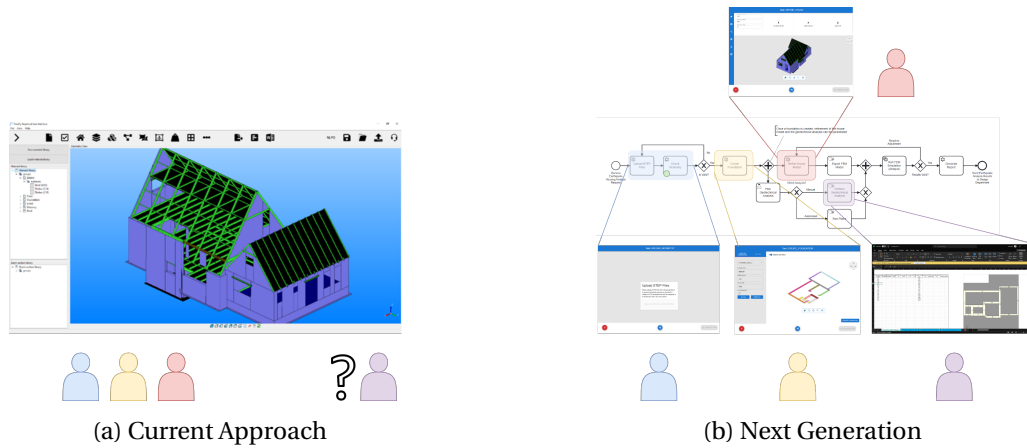


Figure 1.2: Current vs. Next Generation Approach for Earthquake Analysis Process

In the desired approach, the formalized process becomes the backbone for facilitating this collaborative activity by providing users with process awareness, while the KBE application becomes leaner as developers can focus on implementing task specific logic and interfaces while refactoring functionalities such as the geometry checker into separate re-usable applications. Furthermore, with a majority of communication in design activities being asynchronous in nature, meaning occurring not at the same time, [14, p.543] simply adopting information-centric software without process orchestration is not sufficient [15, p.43]. As a result, using workflows for solving collaboration challenges in engineering seems intuitive. However, as will be covered later in thesis using process centric approaches such as workflows, reduces flexibility, Figure 1.3. Therefore, the challenge is to maintain flexibility when synthesizing WfM and KBE methodologies to solve the current problem at hand.

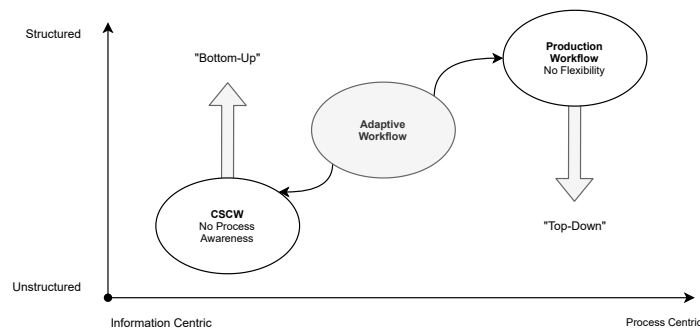


Figure 1.3: The Collaborative Work Spectrum, Adapted from [16]

## 1.2 Research Objective & Questions

Due to the lack of previous research on the integration of KBE and WfM for the purpose of collaboration, several open questions remain. Although the identified limitations from literature provide an additional motivation for researching the synthesis, the primary goal of this research is bounded by a KBE-centric view to improve its collaborative usage and process orchestration capabilities. The formal goal of this research is therefore:

*“To improve the collaborative usage of KBE applications by means of increasing process awareness through process modeling and visualization, and facilitating the integration of KBE with business processes.”*

Nonetheless, the literature study revealed that WfM could benefit from KBE technology to increase its capability to deal with dynamism. Furthermore, using a KBE application as an information backbone might reduce the overhead required to define tool specific interfaces. Several research questions are phrased to enrich the scientific understanding of creating a KBE-WfM synthesized software architecture to address the goal and sub-goals. These research questions stem from the abstract question: “How can a synthesis between KBE and WfM be achieved?”

- RQ-1.** What technology is suitable to expose the information of a KBE application to external services while supporting dynamic types, lazy evaluation, and dependency tracking?
- RQ-2.** How can the runtime cache of a KBE application be persisted to support the transactional nature of tasks where each task leads to a new state of the model?
- RQ-3.** What is the time and memory complexity associated with persisting the runtime caches of a KBE application?
- RQ-4.** Can runtime dependency information be used to establish external steering of a WfMS to modify the control-flow dynamically at runtime based on data dependencies?
- RQ-5.** If external steering is possible, does it show promise for increasing workflow flexibility?

## 1.3 Thesis Structure

This thesis summarizes the research done to push the start of the art in collaborative KBE. First the theoretical content needed to perform this work is presented by Chapter 2. Subsequently, the methodology and theories developed during the application phase of the thesis work is detailed in Chapter 3. Afterwards, verification of the software prototype through experiments and case studies are performed as explained by Chapter 4. These are then reflected upon by Chapter 5. Finally, conclusions of this research are presented by Chapter 6.

## 2 Theoretical Content

### 2.1 Knowledge Based Engineering

Since the word “knowledge” is used in day-to-day life, its semantic meaning within computer science and literature is often overlooked. To get a grasp of what KBE is, as well as how it is related to other software tools, it is prudent to look into the nature of information and knowledge. For this purpose, a common illustration of the terms Data, Information, Knowledge, and Wisdom, the DIKW taxonomy, can be used. The purpose of this taxonomy, presented by Figure 2.1, is to relate these terms and to describe the process used to transform one entity at the lower-level in the hierarchy to one at a higher-level [17, p.164].

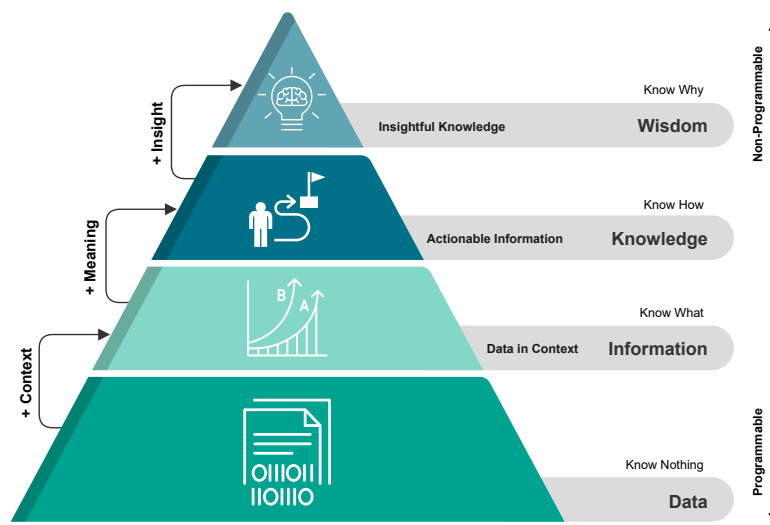


Figure 2.1: Data, Information, Knowledge, Wisdom (DIKW) Taxonomy [17]

In Figure 2.1, abundance and programmability decreases from the lowest-level classification, data, to the most valuable: wisdom. An important distinction is between data and information which allows one to roughly differentiate Computer Aided Design (CAD) software from KBE. While the former provides geometric abstractions such as curves, extrudes, and fillets for the user to describe what the product looks like, the latter enables users to create so-called “information models” by defining what the product consists of using Object-Oriented (OO) representation. This places product data in context by drawing relationships between entities and establishes a shared understanding of it amongst engineers. As described by [18], information models are primarily useful for designers to describe their domain. For engineers this could be the difference between dealing with curves built from Cartesian coordinate data in CAD software, as opposed to interacting with an airfoil object in a KBE system. In the words of [19]: “(KBE) allows an engineer to model a geometric shape by using his own jargon instead of using points, lines and surfaces. An engineer could for instance model a wing by stating its length, profile, twist, sweep, etc. and the KBE application, based on the knowledge it contains, creates the associated points, lines and surfaces.” This is why in this thesis, the classes and slots of a KBE application are seen as an information model.

While information describes “what” the product is at a given moment in time—in this case the wing geometry and metadata—knowledge has an actionable component to it that provides a description of “how” to create the product. This is another fundamental difference between CAD and KBE, where the former describes “what” the resultant geometry is, whereas KBE describes “how” to create it and “what” the product model represents [20, p.2, 21, p.211]. Describing the “how” often takes the form

of rules, formula, or other parsable representations; allowing one to classify KBE as weak artificial intelligence since the knowledge is modelled explicitly [22, p.1263]. Additionally, since a KBE application has the capability to generate rather than simply communicate the present state of a design it is referred to as generative modelling [21, p.209].

### 2.1.1 Core Technologies

Although a pure scripting approach could theoretically be used to create an application like the Multi-Model Generator (MMG), KBE systems offer several benefits that simplify application development. Besides providing direct access to a CAD kernel, KBE systems have three core technologies that optimize information access. Namely, these consist of: lazy evaluation, runtime caching, and dependency tracking [21, p.222], which are summarized as follows:

**Lazy Evaluation:** The ability to execute relevant portions of the code as required, instead of running it eagerly at launch.

**Runtime Caching:** Memoizing evaluated information such that subsequent requests for the same values do not result in recomputations. [23, p.4]

**Dependency Tracking:** Maintaining a record of the data flow used to evaluate the information requested. This allows a KBE system to understand which caches to invalidate when changes are requested and not recompute everything [23, p.4] [24, p.262].

Unlike scripts that follow an imperative programming paradigm, where one dictates what to run next explicitly, KBE applications have no pre-determined “start” or “end” [21, p.241]; allowing the process to instead emerge at runtime based on requests. This emergence is enabled by the declarative programming paradigm provided by Object-Oriented Programming (OOP) languages. This paradigm implicitly resolves the data flow necessary to satisfy a given demand by recursing until previously evaluated data is reached. Doing so allows complex systems to be modelled bottom-up or “composed” from elementary building-blocks, such as the HLPs in the MMG application, which reference the information they require. This allows KBE systems to lazily evaluate only the information requested, thereby, efficiently providing multiple “views” of a central product model.

While the property of emergent data flow is inherited from the declarative features of the programming languages KBE systems are built on-top of, arguably their most beneficial feature is unique, the so-called dependency tracking mechanism that maintains a record of this data flow. This allows a KBE application to determine what needs to re-evaluate when a design change is made. Using this functionality alleviates the burden of engineers from writing explicit imperative code to describe how to maintain all views of the product model or from having to re-run the entire design process. Coupled with the runtime caching mechanism that memoizes the results of computations, a reduction of the overall runtime of a design process can be achieved on subsequent runs—assuming that the caching overhead is less than task execution time—by re-evaluating only tasks that are no longer valid [21, p.246]. Reducing overall runtime is especially useful when dealing with complex products where the tasks are time-consuming, either as a byproduct of the level of fidelity or the sheer number of parts.

Besides potential runtime improvements, the dependency tracking mechanism has a benefit of preventing mistakes during Engineering Change Management (ECM) activities by ensuring that maintaining consistency is no longer a manual activity. On the other hand, for fully-automated processes it allows one to expose only the design variables that should be controlled by the process orchestration layer [25, p.5]. In other words, reducing the need to define variables to check consistency constraints, resulting in simpler simulation workflow definitions.



### 2.1.2 Limitations

The systematic phase of the literature study revealed limitations of KBE systems that hamper its usability in collaboration. The reader may refer to articles that discuss further limitations of KBE systems [7, 26, 27], however, the relevant limitations for this research are:

1. KBE applications become “black-boxes” that provide limited context for understanding what happens under-the-hood. [2, 27, 28]
2. KBE systems have weak workflow integration [2, 29]
3. KBE systems have limited web collaborative solutions [2, 9, 26, 30, 29]

The black-box nature of KBE applications is perhaps the most widely published and criticized limitation. The notion of a “black-box” is summarized by how a KBE application “produces some output with some input, but nobody knows what happens in between” [27, p.5]. One cause of black-box applications is the ad-hoc development process that often embeds knowledge in the product model, as demonstrated by Figure 1.1a. Although, representing knowledge in code allows greater expressiveness, malleability, and debuggability [27, p.8], it has a detrimental affect toward traceability [31]. The reduced traceability makes it difficult for engineers to understand the process used to arrive at results, which negatively affects collaboration. In the worst case scenario, alienation of resources can occur due to people loosing track of the “big picture” to which their work is contributing; resulting in a loss of productivity [16, p.14]. Similarly, [32, p.3] identified this problem in complex Multi-Disciplinary Optimization (MDO) workflows that lack a top-level overview, and stated that this leads to inconsistencies, hampers determination of design trends, proves detrimental to decision-making, and undermines trust amongst stakeholders.

As identified by [25, p.8] monolithic KBE applications, often prevent an optimizer from directly controlling coupling variables as only input and outputs are directly accessible. Analogously, an engineer wanting to perform a manual task to modify an intermediate result during the execution of a KBE application would be prevented from doing so. The next generation approach of this thesis, Figure 1.1b aims to overcome this by exposing intermediate results and allowing external decision-making. However, the lack of this functionality could have prevented stronger workflow integration in KBE systems in the past, simply because the opportunities to externally control the application are limited.

This relates to the final identified limitation that addresses the restricted collaborative usage of KBE systems due to the apparent lack of web-based solutions. Even though there are solutions for collaborative web-based design and manufacturing systems, such user-friendly solutions for knowledge-based systems (KBS) are underdeveloped [30, p.261]. Developing a web-based approach for KBE necessitates a “dynamic information transfer environment”, but updating this information is a critical issue [27, p.11]. Interfacing KBE with PLM could provide this information transfer environment; however as identified by the third limitation, lifecycle management of KBE models is challenging. Although, a clear reason is not provided by [28], one can hypothesize that the user-defined information model in a KBE model varies per application; making it harder to manage than the rigid pre-defined information model of a CAD system. This is why focusing on information modeling in this research is paramount to enabling rich collaborative features with KBE.

### 2.1.3 Relevant Literature

According to [2], the Integrated Management of Product Heterogenous Data (iProd) project was expected to overcome the lack of workflow integration in KBE. Starting in 2011, this project aimed to improve the efficiency and quality of the Product Development Process (PDP) by harnessing Knowledge Management (KM), Knowledge-Based Engineering (KBE), and Process Integration and Design

Automation (PIDO) [33, p.2]. A collaborative knowledge base was constructed, that became a SSOT with a single innovative interactive interface [34, p.459].

The core of the iProd project was to create a flexible Service Oriented Architecture (SOA) software framework to harness reasoning capabilities on formal knowledge models [35, p.2]. Within this scope, research at the Delft University of Technology focused on utilized the ontology modelling, reasoning, and querying functionalities of Semantic Web Technologies (SWT) to reduce the overhead required to define simulation workflows. [36] created a framework to capture product and process knowledge in High-Level Engineering Services (HESs) and High-Level Activities (HLAs) that could then be reasoned upon to instantiate specific simulation workflows that were implemented in the Optimus PIDO platform. Building on top of the HES and HLA concept, [37] created the InFoRMA framework to advise, formalize, and integrate MDO architectures based on product and process knowledge. [25] subsequently utilized this framework to assist in splitting apart a black-box monolithic ParaPy application, the Hinge-System Design and Optimization Tool (HDOT), to expose intermediate variables to the process orchestration layer. Doing so allowed the optimizer to deduce the complex coupling between these variables and enabled the user to have greater flexibility to define multi-objective optimizations.

Similar to this research, the iProd project aimed at becoming the backbone for collaborative software tools such as PLM [35]. Even though [34] remarks that human interaction is required since a design cannot be made automatically and requires human input, the primary human-system coupling mechanism in iProd is based on capturing changes to the knowledge base when the framework is used. Thus, there seems to be limited process orchestration capability to incorporate human decision-making inside the managed process. Leaving human decision-making unmanaged, perhaps allows more flexibility to create ad-hoc “workflows” outside the managed process, yet it reduces task awareness in the virtual enterprise as constituents are not aware of the human tasks being executed. Also, doing so enlarges the divide between automated and manual tasks in a design process. Going forward, the PDP should include human-system coupling in workflows, since creative tasks that are not desirable or possible to automate are still a significant contributor to the success of a design.

Achieving human-system coupling in design overlaps with the field of Decision Based Design (DBD) where the guiding philosophy to utilize decision workflows to incorporate human judgement [38, p.2]. Within this field, it was observed that KBE does not yet address the challenges of decision workflows in the design of complex systems [38]. Subsequently, the PSIDES KBE platform was created to address a lack of reusable and executable decision knowledge and user classification through a decision template approach. Ontologies were used to represent a central knowledge base, that could provide users of different knowledge levels with decision support [38]. However, central process orchestration through a workflow was not the focus of this research. On the other hand, [11], focused on a formal mathematical representation to incorporate a workflow component in Knowledge-Based Systems (KBS). The KBS4IL developed as part of this research aimed to bring a decision support system to the logistics domain [11, p.8].

Relating to the desired ability of a KBE application to influence control flow, [39] addressed the complexities of ECM activities in virtual enterprises through KBE-based workflows. Here, the dependency tracking capabilities of KBE were harnessed to simulate the impact of an engineering change committed to a PLM-based SSOT. Therefore, the ECM impact analysis capability greatly resembles the desired functionality of informing users of the impact of their requested change. However, dependencies determined from the KBE application were not cached in the model database. Instead, the KBE application was constantly triggered to evaluated proposed changes [39, p.350]. Doing so could potentially lead to a similar amount of overhead as keeping a KBE application alive. However, as

this depends on how often the KBE application is invoked as well as its runtime performance this research can provide insight into when or if shutting down a KBE application is worthwhile.

Overall, the literature on synthesizing KBE and WfM technology is limited and has primarily focused on harnessing the knowledge capture aspect of KBE technology to simplify the definition of simulation workflows. As such, there is a lack of systemic literature on supporting human-system coupling through workflows in PDP. Even though the work of [36] determined that using Business Process Model and Notation (BPMN) was ideal—a visual modeling language that has rich support for human and manual tasks—manual tasks within the framework were not implemented. Perhaps due the origin of KBE being rooted in full-automation, the present body of research gravitates toward fully-automated simulation workflows and re-use of product / process knowledge. While design process modelling is a “top-down” endeavor where high-level steps get refined into detailed ones, product modelling is the opposite and takes the form of a “bottom up” approach. [40, p.132]. Therefore, there is a mismatch of modelling approaches which could explain the limited research available on the topic of stronger workflow integration in KBE. Consequently, understanding how to solve this mismatch requires new research that integrates concepts from other fields since KBE research cannot yet answer this question.

## 2.2 Workflow Management

Understanding how to address the mismatch of product and process modelling approaches naturally “flows” toward exploring the field of WfM. The “top-down modelling approach of WfM is becoming increasingly relevant due its knack for coordinating complex processes that necessitate a subdivision of activities to execute them efficiently across distributed resources. As product complexity grows, the functionalities required to support them become too large to implement in a single application. Therefore, tools that specialize for a specific task are often stitched together with application code [41, p.26]. WfM is then a means to perform the stitching of these tools—often referred to as “services”—in a scalable and reliable way such that the impact of failures and delays are minimized.

With web-based applications, cloud-hosted software platforms, machine-learning models, and simulations growing in complexity, a lot of effort has gone into various types of workflow products. As a result, WfM is a trending part of modern IT infrastructure [42] and the field of research surrounding it is quite massive. As with the usage of “knowledge”, the term “workflow” has become commonplace in natural language, which makes its meaning ambiguous. Resolving this ambiguity is crucial before delving into the field of WfM in greater detail. At its core a workflow entails an assemblage of tasks in order to do “work” or accomplish a goal. The formal definition according to the Workflow Management Coalition (WfMC) is “The automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules.”

### 2.2.1 Task Definition

An aspect of workflow terminology that deserves special attention is the fundamental building-block: a task. Formally, it is the smallest distinguishable part of the process that is atomic and cannot be subdivided further [16, p.7, 32]. However, [43, p.29] makes references to “block” tasks which represent sub-processes. This can lead to subjectivity due to how a “task” can then be viewed differently depending on context. For example, a manager could view running a CFD analysis as an atomic task, however, in the viewpoint of the CFD engineer, the analysis comprises many steps such as meshing, grid convergence studies, and the analysis of results. Therefore, a further classification is required for which [44, p.6] suggest the application of the widely used ACID properties, originally developed for databases, to characterize a task from the perspective of a WfMS. [45, p.289-290] originally coined the

ACID transaction properties, which can be redefined to determine the boundary of what constitutes a task [44, p.6]. With this idea, an adaptation of the ACID properties is presented by Figure 2.2 below [16, p.166].

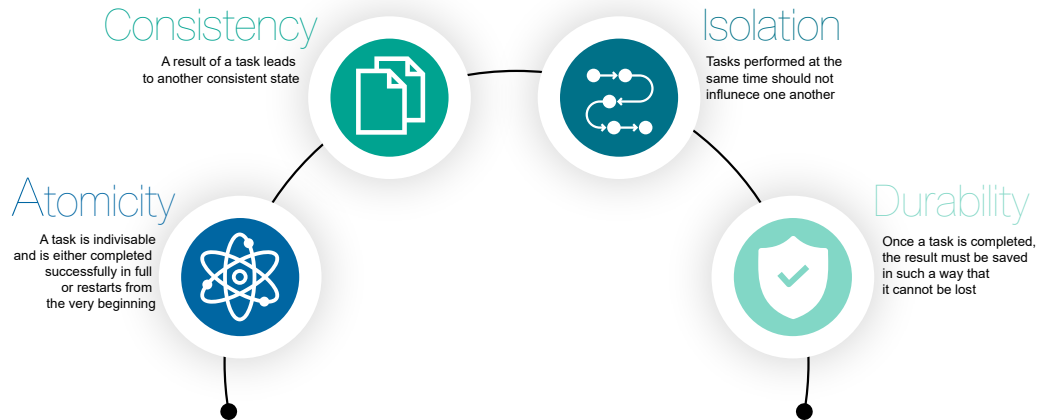


Figure 2.2: ACID Transaction Properties Adapted to Reflect Task Boundary

### 2.2.2 Fault Tolerance

Fault tolerance refers to the ability of a software application to handle and recover from errors. It is important for this research to recognize that the present usage of KBE applications is not particularly fault-tolerant due to the lack of transactional safety and/or application state persistence. The aforementioned ACID transaction properties represents a prevalent approach to implementation of fault tolerance in software applications [46, p.295]. The primary motivation of utilizing ACID transaction properties in a workflow is to ensure that each task performs recoverable actions on product data. In essence, it prevents the creation of corrupted partial states due to the failure of a task.

An alternative to transactional safety is the implementation of workflow persistence, which involves either manual or automatic creation of “checkpoints” to store the full state of the workflow. As described by [47] workflow persistence provides a “point of recovery for the workflow instance in the event of system failure, or to preserve memory by unloading workflow instances that are not actively doing work.” As a side note, the aspect of “unloading” from this quote resembles the desired functionality of shutting down the KBE application to reduce the memory footprint of the workflow. Of significance to this research, is understanding how to prove the fault tolerance of KBE applications when used within collaborative processes.

### 2.2.3 Control Flow vs. Data Flow

Another important concept that is prevalent in workflows is the difference between control flow and data flow. Theoretically, control flow dictates the execution order of tasks using the aforementioned basic mechanisms of sequence, selection, parallelization, and iteration. On the other hand, data flow describes the data dependencies between tasks such that the data produced by one task is consumed by the other [48, pg.73]. Whereas a data edge is essential to the validity of the computation, a control edge can be removed if sufficient concurrency can be achieved with the underlying computing architecture [48, pg.74].

The most common type of data and control flow in workflows is the so-called Directed Acyclic Graph (DAG) where the tasks are nodes, while the edges represents data and control dependencies [49, p.1]. Here, the acyclic nature means that one cannot trace a closed loop from any of the edges between

tasks. Consequently, iteration with an acyclic iteration model is not possible and instead requires a cyclic execution model. Figure 2.3 provides an overview of the different execution models to visually depict the various control and data flow possibilities within each.

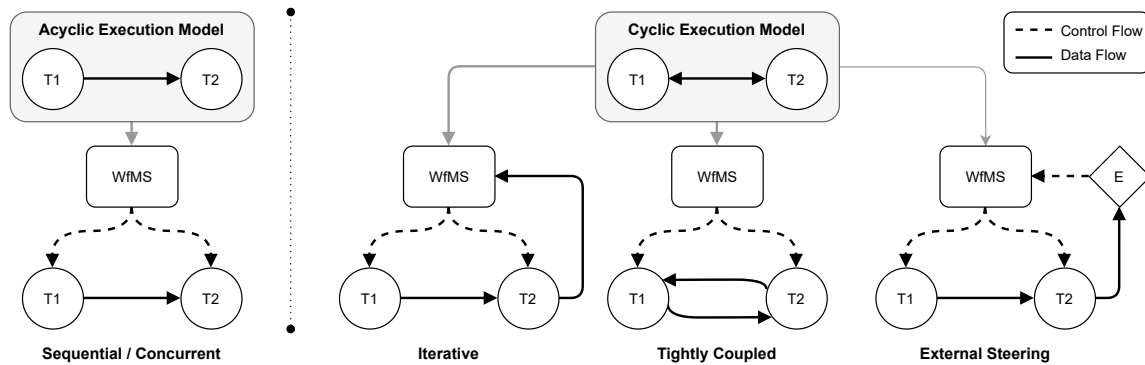


Figure 2.3: Workflow Execution Models Categorized by Control and Data Flow [50]

Amongst the execution models presented by Figure 2.3, an important one to highlight is cyclic external steering. Although similar to the acyclic execution model, a selection mechanism is used to incorporate the decision-making of a user or system to “steer” the control flow based on the data output of T2. Since a cycle can be created if a re-run of T1 is decided, it is classified as a cyclic execution model. An interesting application of this model would be to use a KBE application to externally steer a WfMS based on data dependencies. The reason for doing so is a fundamental premise of this research; namely that the declarative backward control flow approach of KBE is advantageous to complex product modelling, as opposed the imperative approach of WfM that requires an up-front forward description of data and control flow.

## 2.2.4 Historical Perspective

Having highlighted important concepts in WfM, this section describes its historical development to put these concepts in perspective. The origins of WfM can be traced to the 1970s when researchers at Xerox PARC working on a project called “Office Automation Systems had the idea to support business processes through generic tools and methods using petri-net based mathematical representations of workflows [16, p.27]. Following from this initial work, the need for such systems increased in mid-1980s during the digitalization of business information as a way to assign and track electronic work to humans. It was the digital equivalent of dropping mail in somebodys in-box to assign work to them and therefore heavily focused on the primary form of information: the document [10, p.17]. However, WfMS were only recognized as a standard part of ICT systems as the first commercial systems became available in the 90s [11, p.3, 51, p.5-14].

Over time, as the amount and heterogeneity of digital information grew, WfMS began to take on increasing automation duties to process it, notify distributed resources of pending tasks, and await the completion of long-running tasks before continuing an automated process [10, p.18]. To prevent ambiguity over the functionality of a WfMS, standardization activities began with the formation of the WfMC in the mid 90s and The Workflow Patterns Initiative in the late 90s [16, p.149, 43, p.16]. The next phase of WfMS evolution related to the growth of applications using SOA in the late 90s and early 2000s. Moreover, the onset of globalization resulted in the “outsourcing of activities to external businesses. As a result, Business Process Management (BPM) was envisaged by WfM vendors to model, analyze, and run these complex processes across organizational boundaries [10, p.18]. Subsequently, many WfMS rebranded as BPMS [11, p.3]. Today, there are tremendous amounts of workflow products available on the market and their intended usage has outgrown the initial vision of automating tasks in an office.

### 2.2.5 Specialized Systems to Automate Workflows

While WfM vendors transitioned their office automation systems into comprehensive Business Management Systems (BPMS), scientific fields requiring workflow technology developed their own specialized systems. As a result, nowadays WfMS can be generalized under two different categories: business process and scientific or computational workflows [6, p.231]. Nonetheless, one of the outcomes of the standardization activities of the WfMC was the creation of a workflow reference model that provides a basis for WfMS functionalities. This model defines five interfaces linked to a central enactment service responsible for the runtime execution of workflows. These interfaces can be seen in all workflow systems regardless if they are intended for business processes or scientific workflows and they are summarized from [52] as follows:

1. **Process Definition Tools** to model a process in such a way that can be interpreted by the workflow engine.
2. **Workflow Client Applications** to allow humans to perform work items and provide results back in a machine parsable manner.
3. **Invoked Applications** to run automated tasks on a variety of applications.
4. **Other Workflow Enactment Service(s)** to add interoperability with other WfMS such that tasks can be passed between them.
5. **Administration & Monitoring Tools** to provide task awareness to resources and gather insight on how to improve the process definition.

To select a suitable WfMS for fulfilling the desired functionalities it is necessary to create a finer-grained classification and understand the strengths of each type. Having observed the functionalities of a wide array of WfMSs during the ILR, it was observed that these specialized products can be classified into four categories. Table 2.1 below summarizes the result of this classification which is discussed on the next page.

Table 2.1: Categorization of Specialized WfMSs Based on Role and Relevance

Category	Primary Role	Relevant For	Manual Tasks	Examples
BPM	Modelling and managing business processes	Orchestration, Choreography, Process Modelling & Optimization	Yes	Camunda, Kissflow, ProcessMaker, Bonita
Scientific WfM	Abstractly defining workflows for HPC	Large / Extreme Scale Simulations	No <sup>i</sup>	Pegasus, Taverna, Kepler, Galaxy
PIDO	Defining and executing optimization workflows	MDO, Design Exploration	No	Optimus, ModeFrontier, pSeven, RCE
Data Pipeline	Batch-processing large volumes of data (ETL Tasks)	Data Science, Machine Learning	Limited	Luigi, Prefect, Airflow, Kubeflow

<sup>i</sup>Implementing human-in-the-loop workflows in scientific WfMSs is currently state of the art [50]



Starting with BPMS, one can regard them as specialized WfMS that are catered to supporting complex business processes that span organizational boundaries. As such they often come equipped with more fault-tolerance and workflow persistence features since the likelihood of failures is higher when executing process logic across managerially independent and geographically distributed systems. Furthermore, they often come equipped with more sophisticated process modelling tools to more easily express the business process. A popular choice is to use BPMN as will be discussed by Section 2.2.6.

Moving on, Scientific Workflow Management Systems (SWfMSs) are specialized to allow users to describe highly-parallelized computation workflows through abstractions that can later compile to run on specific High Performance Computing (HPC) hardware. Due to their focus on automation, they do not handle work items (manual tasks). Although current research activities are focusing on incorporating human-in-the-loop work items to allow workflows to scale to extreme scale [50]. In a similar category but targeted toward engineering problems are PIDO systems, which are the natural choice for optimization workflows. Unfortunately, once again they emphasize full-automation and hence have no support for work items. Although their optimization capabilities are valuable for engineering work, the features of a BPMS fulfill the intended desire to coordinate human-system coupled processes. Furthermore, it seems easier to add optimization capability to a complex BPMS, than it is to add BPMS features to a PIDO platform. Therefore, use of a BPMS is found to be ideal for this research at this stage.

Finally, with an ever-increasing amount of data in modern society, the last group of WfMS specialize to support streaming processing of large data sets. For example, Spotify uses the Luigi platform to process data to give customers recommendations, and highlight popular music per genre, amongst a host of other use-cases [53]. Therefore, these Data Pipeline tools provide abstractions to easily compose so-called Extract, Transform, Load (ETL) tasks in a DAG, that like an assembly line continuously processes chunks of data as it gets handed off from a source to a destination. However, streaming processing is less common in engineering design problems as the data sets are heavily interdependent and heterogeneous as compared to highly uniform datasets such as Spotify's listening statistics. As a result, discussing ETL tasks in detail is outside the scope of this thesis; the reader may instead refer to [54, p.401-421].

### 2.2.6 Modeling Techniques

Process definition tools are one of the standardized interfaces of a WfMS, as described by Section 2.2.5, and allow modeling processes in both a human-readable and machine parsable representation. Often they consist of graphical notations that depict the execution order of activities in a business processes and thus provide means to provide task awareness to users of a KBE application. This section explains the modeling techniques available that could be applied to formalize the envisioned process model of a Knowledge Engineer in Figure 1.1b.

According to [43, p.6], business process modelling requires three dominant perspectives: control flow, data, and resource. As explained by Section 2.2.3, although the data and operations performed on it may necessitate a certain flow; the control flow takes precedence over data flow. Consequently, the activities contained within a business process and the order in which they are executed are defined by the control flow perspective. Meanwhile, the data perspective defines the data and information required during the execution. Finally, the resource view maps humans and systems to activities to later determine the resource responsible for executing a task. Due to the vision of using the declarative KBE approach for data modeling, an emphasis is placed for now on the control flow perspective. Table 2.2 summarizes the modeling techniques available for each of these dominant perspectives.

Table 2.2: Modeling Techniques for Each Dominant Business Process Perspective [43, p.6]

Control Flow Perspective	Data Perspective	Resource Perspective
BPMN	UML Class Diagrams	Use Cases
UML Activity Diagrams	ER Diagrams	Role-Activity Diagrams
Petri Nets & State Charts	Object-Role Models	Organizational Charts
EPCs		X.500

One of the control flow techniques in Table 2.2 is BPMN, which has become a standard recognized in ISO/IEC 19510:2013 in 2013 and is backed by the Object Management Group (OMG) [55]. The goal of this modeling notation is to be easily understood by business users, analysts, and developers, while consolidating other notations such as UML Activity Diagrams, IDEF, and Event-Process Chains (EPCs) [56, p.1]. While BPMN depicts what types of documents and artifacts flow through tasks, it is not a data flow language [56, p.20]. This means that expressing data dependencies through a BPMN diagram is not its primary function. Nonetheless, [36] compared BPMN to other control flow languages in Table 2.2, based on five categories for its applicability to represent simulation workflows in KBE. The result of a trade-off was that BPMN was the most applicable language for that purpose due to its completeness [36, p. 142]. Another study, found that BPMN was most suitable for its ability to model complex processes [57]. These conclusions are verified by the vast modeling capability of BPMN; an example being its support of both process orchestration and choreography [56, p.315].

On the surface, BPMN has only five elementary categories: flow objects, data, connecting objects, swimlanes, and artifacts. However, the main graphical elements consist of the flow objects which are activities, events, and gateways, along with connecting objects that define the control flow between them [56, p.25, 58, p.67]. Specializations of these elements, such as sub-processes, allow multiple levels of abstraction to be defined. As [58, p.273] describes, this allows both strategic process models—useful for fast comprehension—and operational process models—having sufficient detail for automation—to be defined. Figure 2.4 below provides a sample strategic process model of a recruiting process. The pools and lanes in the diagram provide clarity on the activities performed by internal and external participants. For example, dashed arrows connect the “application submission” activity to denote a process boundary where a message flow is required to involve an external resource.

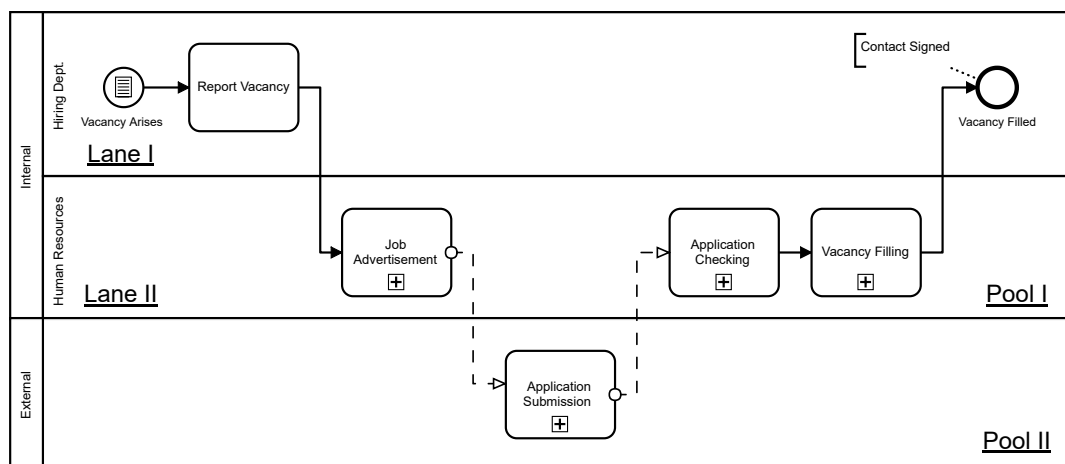


Figure 2.4: BPMN Diagram of a Recruiting Process at Strategic Level [58, p.281]



### 2.2.7 Limitations & Research Trends

Due to applicability of WfM to multiple fields, the limitations and research trends identifiable within it are quite large. As a result, the sole limitation of relevance for this research is:

- WfM is underutilized in engineering due to its rigidity [2, 12, 59, 60, 61, 62]

The desire to formalize business processes surrounding the usage of a KBE applications brings with it the possibility of over-constraining its usage. After the commercialization of WfM, the latter half of the 90s were focused on “dynamic workflows” to adjust process definitions quickly to new requirements [51, p.14]. As [63] mentions, real-life work is much more dynamic and rich than what can be represented in process models, therefore users should be able to modify workflow models at runtime. Although the topic of “dynamic” or “adaptive” workflows has been an ongoing research trend since this initial effort; the gathered research in this thesis has provided evidence suggesting that limited progress has been made to bring these methodologies to engineering workflows. For example, [59] researched how to assist collaboration activities in the aerospace sector. The research gap identified by their work discusses how the topic of data flow and workflows in complex product development is lacking sufficient depth [59, p.30]. [2, p.7] adds that WfMSs force companies to organize into pre-defined structures and are unable to cope with deviations in a design process from the formalized model. Therefore, the field of WfM itself is underutilized in engineering, which might explain the weak integration of workflows in KBE as discussed in Section 2.1.2.

There are two conflicting interests in workflows at play here: (a) the high abstraction-level of “top-down” models detach them from the information required by engineering tasks, (b) the lack of flexibility prevents ad-hoc changes [62, p.1120]. These two interests conflict due to how creating a detailed workflow to address the high abstraction-level would end-up reducing its flexibility even further. This conflict severely impacts engineering processes as they require ad-hoc modifications, the activities within them are difficult to plan ahead of time, and the data transmitted between tasks is highly interdependent [12, p.1472]. The latter aspect of data interdependence can impact data validity when enforcing a control flow on a process [61, p.2].

To prevent such issues two flavors of KBE-driven workflows are envisioned: (a) **Dynamic Workflows** where the KBE application externally steers the WfMS based on its dependency tracking and slot evaluation mechanisms, (b) **Emergent Workflows** where static control flow does not exist ahead of time and instead “emerges” as a consequence of the transformations applied on data at runtime. The focus of this research is on the first approach. A solution from literature for introducing dynamism in workflows was identified that involves the representation of control-flow as a composition of smaller scale workflows. These smaller scale workflows, called worklets are essentially self-contained and complete workflow processes that are meant to handle specific tasks [64, p.135]. Furthermore, worklets allow control flow modification by being able to be dynamically selected based on contextual data at runtime. Essentially a top-level parent process can be dynamically altered by changing the selection of which worklet to run during process execution. The benefits of this approach are:

- **Ease of Adoption:** Same process modeling methodology can be used for both workflow and worklets as there is difference between dynamic and “normal” workflow elements
- **Ease of Modeling:** Makes composition of workflows easier since individual worklets are less complex to build and verify than monolithic models
- **Ease of Comprehensibility:** Provides workflow views of differing granularity for different stakeholders
- **Ease of Evolution:** New worklets can be added without modifying the macro-level workflow

## 3 Methodology

The present research derives from existing methodologies and theories from KBE, WfM, CE, and Computer Science (CS). This knowledge was used to formulate the overarching hypothesis of this work as:

*“The use of a KBE application to externally steer a WfMS for parts of a process can address the lack of process awareness when using KBE applications while simultaneously reducing the rigidity of WfM.”*

The rationale of this hypothesis comes from the declarative demand-driven nature of KBE, which allows the expression of logic without dictating an execution order [21, p.241] as opposed to the imperative goal-driven nature of WfM. Due to this property, KBE simplifies the creation of multi-use applications, and is expected to allow a workflow to be quickly adapted or extended. The hypothesis is further reinforced by MDO which accredits use of KBE with a reduction in the number of required consistency constraints and simplifying the integration of heterogeneous tools to conduct multi-fidelity analysis [21, p.214].

The desired synthesis would only involve users in a process when necessary, allowing low-level tasks to execute behind-the-scenes, which relates to the concept of conversational composition where so-called gray-boxes are created [46, p.279]. These “gray boxes” are expected to introduce better process awareness through high-level BPMN representations that can provide engineers with a meaningful task overview as depicted by Figure 1.1b. WfM can also provide KBE with better fault tolerance capabilities from its ability to deal with individual task failures and retry them whilst maintaining an audit log of what happened during the entire design process. However, to support this functionality the KBE application should support transactional usage and/or persistence such that changes made to a KBE application during a task that fails, can be “un-done” or ‘rolled-back’ to a previous state of the application before failure.

These considerations result in the following high-level requirements on the software prototype:

- The information of a KBE application must be accessible to the process orchestrator and all involved resources (people & tools).
- The information along with its dependency relationships must be persistable to support fault-tolerance and the desired hybrid workflow concept.
- The KBE application should be able to introduce ad-hoc control flow during a running process to dynamically chain together tasks while simultaneously using the runtime cache and dependency tracking to re-run only what is required during iteration.

These requirements are then directly related to the sections in this chapter, namely first the information modelling methodology will be discussed, followed by the persistence mechanism. Finally, the process orchestration methodology used to enable dynamic worklet publication will be discussed.

### 3.1 Experimental Set-up

To meet the goals of this research, better understand the validity of the hypothesis, and answer the research questions, a software prototype is needed. Therefore, the experimental set-up used to conduct this research is a modern software development environment using the Python programming language, the underlying programming language of the ParaPy SDK. Alongside Python, the open-source Zeebe BPMS is chosen to provide both BPMN modeling and service task definition capability

through its gRPC-based Python client [65]. These features were important to accelerate the prototyping phase. Finally, to handle data-flow between services in workflows, GraphQL a query language developed by Facebook was selected for its granular query capability, ability to cater for multiple demands through a single end-point, and for its potential to become the dominant API of the future [66, p.32].

To iteratively verify the software prototype at each phase of development, Test-Driven Development (TDD) was applied to ensure the building blocks of the prototype function properly. Additionally, Docker was used to standardize the installed packages and operating system configuration in a so-called container and Docker Compose was used to orchestrate multiple containers through a single configuration file. Doing so provided a glimpse into how this software prototype might one day be operated in the Cloud as this containerization mimics how each service within the architecture is isolated from on another.

## 3.2 Information Modeling

The cornerstone of being able to integrate KBE applications within collaborative processes is to enable effective communication of information. To do this the so-called information model of the KBE application consisting of its classes and slots must be formalized and provisioned. The latter meaning that it should be accessible during a process. The aim in this thesis was to find an approach where the KBE application can be used as-is within processes. More formally, the requirements from the information modelling approach are:

1. Allow tools and the process orchestrator to obtain information from the KBE application
2. Allow the KBE application developer to define what to expose from the information model
3. Allow the KBE application developer to change how information is represented
4. Prevent burdening the KBE application developer and prevent mistakes by not forcing them to redefine their types
5. Allow KBE language features to be represented in the exposed information model.

ParaPy currently comes with a generic REST API, that exposes the data model of a KBE application, including instances and their slots. However, due to the lack of exposing application specific types and slots, one could not communicate the domain specific abstractions used by engineers, or simply the information model. As a result, consuming this API from the perspective of the service developer is difficult without a prior understanding of “what” information is available before querying everything. To overcome both of these issues, the state-of-the-art query language, GraphQL, is used to expose the information model of a KBE application in a type-safe way, meaning that the types of classes and slots are represented in the language.

What also distinguishes GraphQL is its selection set concept that allows obtaining partial results from the backend as a client. While it is possible to obtain type-safe APIs using REST through OpenAPI specification, or newer Remote Procedure Call (RPC) implementations such as gRPC or tRPC; none of these technologies support the selection set concept. What makes the selection set concept powerful is it enables data to be queried flexibly by allowing a user to select exactly what they need. This not only alleviates under-fetching and over-fetching problems—relating to getting too little data and too much data respectively—but also alleviates burden during information modeling as use-case specific types do not need to be generated. This is best illustrated by Figure 3.1.

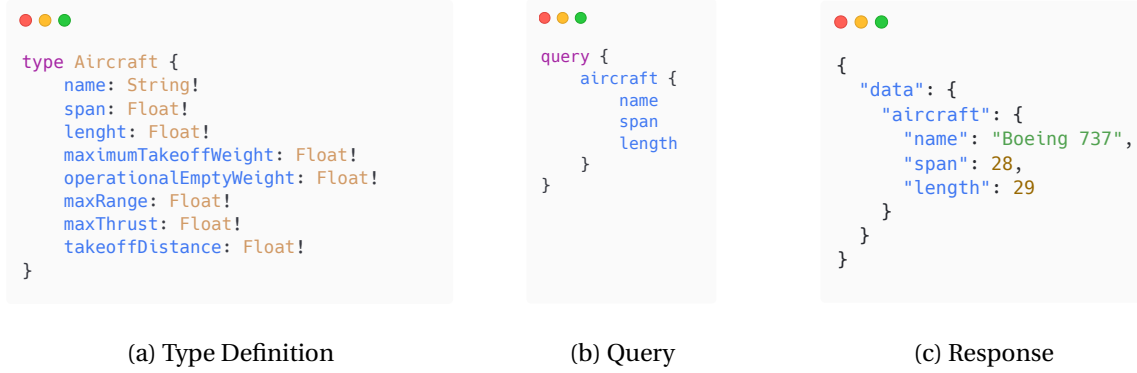


Figure 3.1: Example of a Selection Set in a Query to Obtain Aircraft Data

Here, a basic Aircraft schema is defined with several fields. A representation of a potential query is then provided where a selection set including the name, span, and length fields is used. This means that the response only includes these fields, and not the remaining Aircraft fields defined in the schema. This essentially allows a client to query exactly what it requires from each type in the schema. This avoids the necessity to predict how clients will consume data, and instead allows GraphQL to expose capabilities of the backend [67]. This allows a high degree of flexibility when consuming information. For an overview of example queries please consult Chapter C.

Driving the response of each query are so-called resolvers which are functions responsible for returning the value of a given field. The returned value can either be a scalar with no fields, or an object type. If the latter is returned, then execution continues recursively until no fields remain. [67]. To better understand the similarities between the GraphQL type system and ParaPy, a UML diagram given by Figure 3.2 was created.

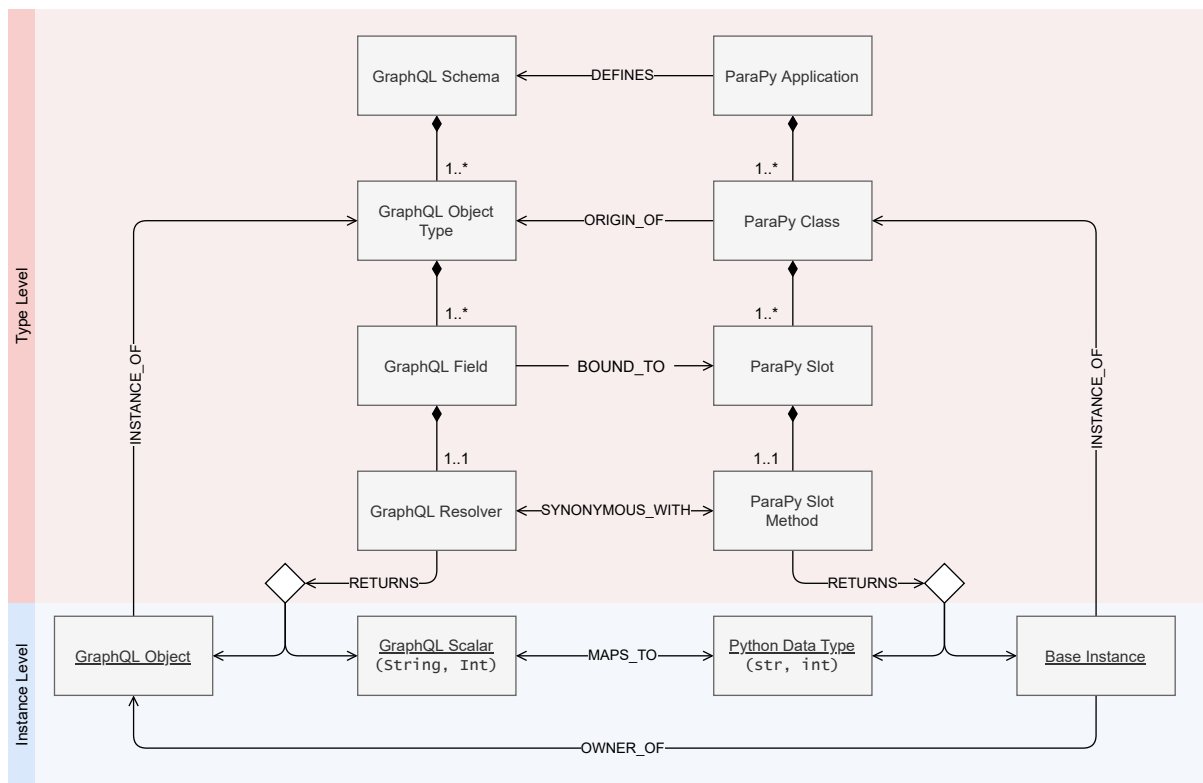


Figure 3.2: UML Class/Object Diagram Depicting Relationship Between ParaPy &amp; GraphQL

Here one can see how the aforementioned resolvers are analogous to ParaPy slot methods which contain logic on how to return a given value. Each slot method is then bound to a slot, which in GraphQL called fields. GraphQL Object Types are then composed of these fields which are analogous to ParaPy classes. The collection of these Object Types is then called a GraphQL Schema, which resembles the collection of classes which make up a ParaPy application. What is interesting to note, is that resolvers can return either an object or a scalar data type. This is similar to how ParaPy slot methods can return either base instances or Python data types. What this enables is a heirarchal representation of product data.

### 3.2.1 Schema Autogeneration

While GraphQL seems to be a good fit for representing the information model of a KBE application, generating GraphQL schemas and writing resolvers for each field is a strenuous activity. To not burden the KBE application developer with learning the intricacies of GraphQL and its best practices, a transpiler, which parses application source code and automatically generates resolvers and types was developed. Such API autogeneration techniques are becoming commonplace in modern databases or database services such as Hasura, Edge DB, and Neo4J.

These techniques allow users to maintain a single model of their domain and use it to create database entities, define API endpoints, and perform validation / authorization on those endpoints. Use of the KBE-GraphQL transpiler created in this research allows a knowledge engineer to deploy an API for their application in seconds without modifying their application. Furthermore, the deployed GraphQL API allows tools and workflow services to communicate with the KBE model in a type-safe way, whilst also being compatible with KBE-specific language features such as dynamic types, lazy evaluation, and dependency tracking.

At the core of this functionality is the origin (class) discovery algorithm to traverse the application source code from a provided root type, and Figure 3.3. This algorithm operates on type-level hence allowing multiple design instances of the KBE application to be served through the same API. The basic principle, which is further explained by Chapter E, is to start from the root class of a ParaPy application, and traverse the application source code via the edges between slots and their return types. In Figure 3.3 one can see how starting from the root class denoted by “C”, all “discoverable” edges are traversed in the first iteration. What the latter means is that the return type of some slots is not known without so-called type inferencing. Although this will be explained subsequently, for now it suffices to say that it allows one to derive type information from the context of neighboring source code. Furthermore, it must be said that compared to regular traversal, inferring the return type of slots is an expensive per-call operation. Therefore, the algorithm shown below attempts to “batch” inferencing calls by discovering as many edges as possible in one iteration by going depth-first through the type-tree and caching all slots that require inferencing. The inferencer is then run at the end of each iteration.

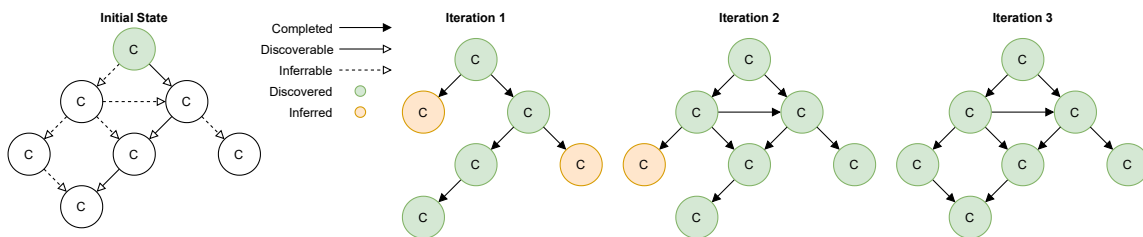



Figure 3.3: Visualization of Origin Type Discovery Algorithm

## Type Inferencing

While transpilation prevents a developer from needing to manually create types and resolvers, it would require a fully type-annotated code-base in order to function without inferencing. A type annotation in Python is a way to instruct type-checkers of what the return type of an assignment of function is. This is shown by Figure 3.4, with for example the radius slot. However, adding these annotations for each slot would be laborious and unnecessary, as type information can be “inferred” for slots that make use of the radius slot. For example, in Figure 3.4, this would be how the return type of the area slot can be inferred to be a `float` due to its use of the radius slot along with floating-point arithmetic operations being used. Therefore, type inferencing is a way to prevent developers from writing unnecessary type-hints by making use of contextual type information and a type inference system has the benefit of reducing the verbosity of code, making it easier to read and write, due to prevention of “silly” and “common” type hints [68]. In this research, type inferencing is used to prevent cluttering KBE application source code with unnecessary type hints just for the purpose of being able to serve a GraphQL API for the KBE application. The *Pyright* inferencer was selected due to its efficiency as well its capability and impressive inferencing capability.

i



```
class Circle:

    radius: Input[float] = Input() # User-defined type annotation

    @Attribute
    def area(self): # Inferred type: float
        return math.pi * self.radius ** 2
```

Figure 3.4: Example of When Type Inferencing is Required

## Slot Resolvers

Whereas the GraphQL Schema provides type information about what data is available, resolvers are responsible for encoding the knowledge of how to provide and modify that data. Therefore, without resolvers, the schema would simply be a description of what information the KBE application contains. To make it executable, one needs to define resolvers which are synonymous to slot methods used in ParaPy applications that encode knowledge of how to return values. In this research, since the GraphQL backend runs within the same Python process as the KBE application, the `SlotResolver` is responsible for directly retrieving the return value of the correct slot, and then converting this data into a suitable format to be represented in the response. For slots that are settable by the user, a resolver is also responsible for processing the input data received via a mutation operation, and converting that data type to the data-type used within the KBE application.

The knowledge of how to resolve certain slot return types is then encoded in these slot resolvers. To keep the code-base extendible to the introduction of more advanced slot return types or new language constructs in ParaPy, new `SlotResolver` classes can be defined without requiring changes to the transpiler. Therefore, this can be seen as an adherence to the Open Closed Principle (OCP) since as long as new `SlotResolver` types are introduced that adhere to this “interface”, then the remainder of the code-base does not need to change. Listing B.1 can be consulted for further clarification.

<sup>i</sup><https://github.com/microsoft/pyright>

### 3.2.2 Modeling Approach

While the desire to use KBE applications as a starting point for information modeling may be desirable to get going quickly, it may be desirable to separate the KBE application information model, from the exposed information. This can be due to numerous reasons, including but not limited to a desire to restrict sensitive information, representing slots different with different names, and filtering undesired slots from ending up in the final information model. During the architecting phase of the transpiler, it was envisioned to prevent an approach where source code is generated and saved on the users workspace. This was done to prevent conflicts if the user modifies the generated source code and then decides to regenerate a new schema. Instead, an approach was preferred where the generated schema in-memory.

However, to allow the user to modify how information of the KBE application is represented in the schema, both a filtering and override mechanism were introduced. The former allows a user to define filter functions for both classes and slots. For example, this allows one to ignore all ParaPy-derived slots, such as those originating from the Base class from appearing in the schema. This functionality is supplemented by an override mechanism that allows definition of custom GraphQL types that are linked to their ParaPy class. Therefore, when the transpiler encounters an overridden type, the user-defined GraphQL type is used as a starting point. An example is provided below by Figure 3.5, which demonstrates how a slot can be represented differently in the schema by defining custom getter and setter functions. In this use-case, the desire is to maintain backwards compatibility for tools consuming a `span` field, when the KBE application has changed to begin using `half_span`.

```
from parapy.core import Base, Input
from parapy.wfm import graphql as gql

class Aircraft(Base):

    half_span: Input[float] = Input(5)

@gql.model(origin=Aircraft, name=Aircraft.__name__)
class AircraftModel(gql.Base):

    span: float = gql.auto(
        slot=Aircraft.half_span,
        getter=lambda v: v * 2,
        setter=lambda v: v / 2,
        deprecation_reason="Span is no longer used in calculations.",
    )

schema = gql.Schema(root=Aircraft, overrides=(AircraftModel,))
gql.deploy(schema)
```

Figure 3.5: Example of the GraphQL Schema Modeling Approach with Overrides



### 3.3 Model Persistence

As workflows are state-machine where tasks are atomic entities that transform one state to the next, supporting multiple states of a KBE model is necessary to maintain the fault-tolerant nature of workflows. The current JSON-based snapshot capability of ParaPy was insufficient for this research as it only persists the modifications made to a model in terms of JSON-representable inputs and is not capable of persisting computed values or the runtime dependencies between caches. Furthermore, a desired functionality of exploiting the dependency tracking capability of KBE in workflows required the ability to persist the full runtime state of an application, including dependencies. Therefore, a challenge was to find a way to persist the full state of a ParaPy application. This problem is particularly challenging for the following reasons:

- KBE models are complex objects that contain a high degree of references
- The KBE object graph may contain cycles due to aggregation
- Slots contain arbitrary data types defined by the user and should not be constrained

These challenges, and most importantly the last challenge have led to the creation of unique solutions that take advantage of the KBE paradigm to most notable be able to consistently persist partial application states. Before, delving into the specifics of these solutions, this thesis makes a distinction between the object graph and the dependency graph within a KBE application. The former represents product tree, whereas the latter represents the dependency relationships between this data. Looking at Figure 3.6, one can see the presence of an unserializable cache value. Additionally, whereas the dependency graph can be assumed to be acyclic, the object graph can have a high degree of cycles, due to aggregation. In this example, an aggregation is already present when passing down the value of the root airfoil points down to the curve. Recognizing and serializing such aggregations is not straightforward when using file formats such as XML, HDF5, or JSON and databases. As a result, it was decided to use the built-in binary persistence module of Python, `pickle` for its ability to resolve these aggregations. Furthermore, it was selected for its capability to represent a wide range of Python objects out of the box.

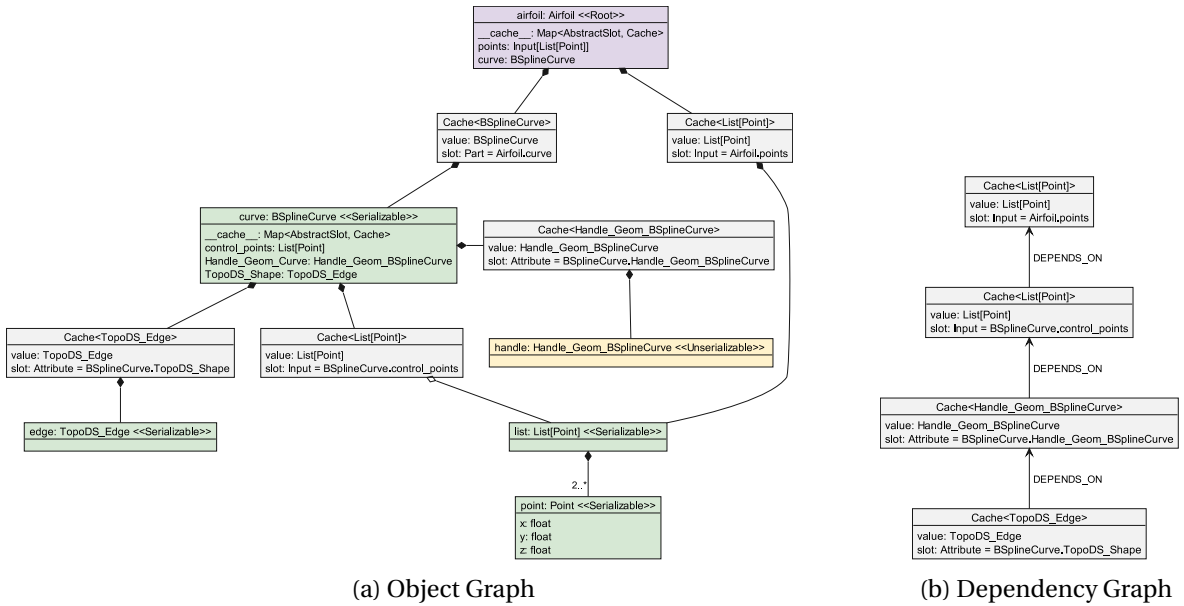


Figure 3.6: UML Diagram of Object and Dependency Graph for an Airfoil



### 3.3.1 Persistence Architecture

From the specific challenges given previously, a persistence architecture was needed that overcome the recursion limits present in the `pickle` module of Python as well as dealing with unserializable data. To do this several enhancements were made to the `pickle` module as follows:

- Creation of a plugin system to enable side-effecting during serialization
- Addition of a rules system to define serializability
- Look-ahead serializer to robustly determine serializability

The first item in the above started as a way to gather all geometries present within a KBE application during serialization and then be able to serialize those geometries individually afterwards. The second item was necessary to define rules to influence the serialization of certain data types. For example, these rules can be used to toggle if geometry is persisted or not. Finally, the last item was necessary to be able to detect unserializable values before causing the main serializer to fail.

An important design decision during the architecting process was creating a plugin to postpone the traversal of the dependency graph until after serialization of the object graph had finished. This was done in order to be able to serialize the complex objects created by KBE applications, as otherwise the serializer must recurse until the end of a dependency chain to save a single value. This greatly increases the number of stack frames required to serialize ParaPy objects and would make it infeasible to persist large objects.

### 3.3.2 Handling Geometry

A primary reason for needing a way to side-effect during serialization was to be able to handle geometries. The underlying CAD kernel of ParaPy defines a topology as the relationships between geometric entities in order to be able to link them together to represent complex shapes [69]. Because of this a shape is represented as a composition tree of sub-shapes that may contain solids, shells, faces, wires, edges, and vertices. Furthermore, these sub-shapes may be arbitrarily organized into compounds. Once a geometry is instantiated by a slot, ParaPy provides direct access to the underlying topology of the shape including its potential sub-shapes. However, this capability poses a problem for serialization as although in the CAD kernel understands that these sub-shapes are pointers to the same objects in memory, inside the Python interpreter they lead to the creation of new unique objects. As a result, the serialization mechanism would in turn serialize shapes that are identical multiple times. Persisting sub-shapes individually would then be costly both in runtime and memory consumption.

To address this problem, the properties of the “shape set” data structure, provided by the CAD Kernel are used, which guarantees uniqueness of contained shapes. Therefore, working principle of this approach is to add all evaluated shapes in the KBE model to the shape set and keep track of the identifiers provided back by the CAD kernel to elements within this set. Due to the underlying data structure, if a shape has been previously added, it will return the same identifier. As a result, this allows data deduplication of “deduping” of the geometry contained within a KBE model. This methodology is implemented as a plugin within the aforementioned persistence architecture. This plugin allows geometry to be added to the shape set during traversal of the object graph at serialization time. Once serialization is complete, the shape set is serialized and stored alongside the KBE model’s object graph. This process is reversed during deserialization, whereby first the shape set is deserialized, and then subsequently accessed during object graph traversal and re-instantiation at deserialization time.

### 3.3.3 Graph Contraction

Previously, in Section 3.3.1 it was explained how an important architectural choice was made to separate the serialization of the object and dependency graph. Although this choice was made for good reason to mitigate the recursion limit, it led to a challenge in dealing with “gaps” or “lost” edges formed in the dependency graph due to parts of the object tree not being serializable. This is illustrated below by Figure 3.7.

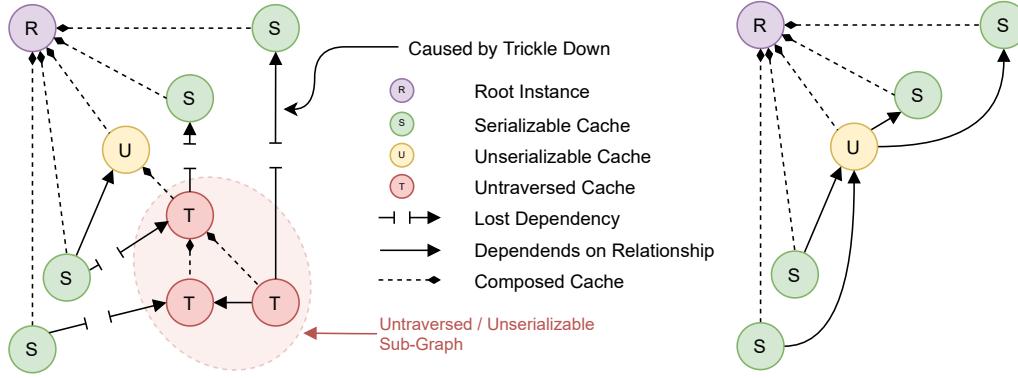


Figure 3.7: Depiction of Untraversed Sub-Graph Requiring Contraction

Here what can be seen is how when there exists an unserializable base instance in the object graph, the dependency relationships of its composed caches are not serialized. This creates a broken dependency between the composed caches and their precedents, which becomes problematic on deserialization when these precedent caches are changed. These situations with untraversed caches typically occurs in the following situations:

- Rule preventing base instance from being serialized
- Base instances contained within an unserializable container, such as Listing 3.1

The former situation occurs often in the ParaPy `geom` package as certain shapes such as the `BSplineCurve` have unserializable inputs and are created within an `Attribute` slot, meaning that they lack the child rules necessary to be able to re-evaluate these unserializable inputs. As a result, it is safer to mark these geometric classes as unserializable.

Regardless of the situation in which these inconsistencies occur, the solution to this problem was to implement a graph contraction algorithm where the contraction takes place along the composition edges of the composed caches. This algorithm is explained further in Chapter E. Besides restoring reactivity of the application after deserialization, the graph contraction algorithm can also be used to simplify the dependency graph. Therefore, this can be used to reduce the storage space required for persisting the state of a KBE application. Especially in situations where a lot of intermediate geometries are required to create the desired final shape, the contraction algorithm can be applied to remove all intermediate caches that are not needed. For example, in the Multi-Model Generator (MMG), the generation of a wing involves several intermediate geometries as depicted by Figure 3.8. Replicating the wing generation steps of the MMG, for a simple wing consumed 130.493 kB. Without intermediate slots, the serialized model consumed 2.486 kB, which corresponds to a reduction of 98.1 %. The reason for this drastic reduction in size stems from the sheer number of intermediate slots and dependencies required to arrive at final result.

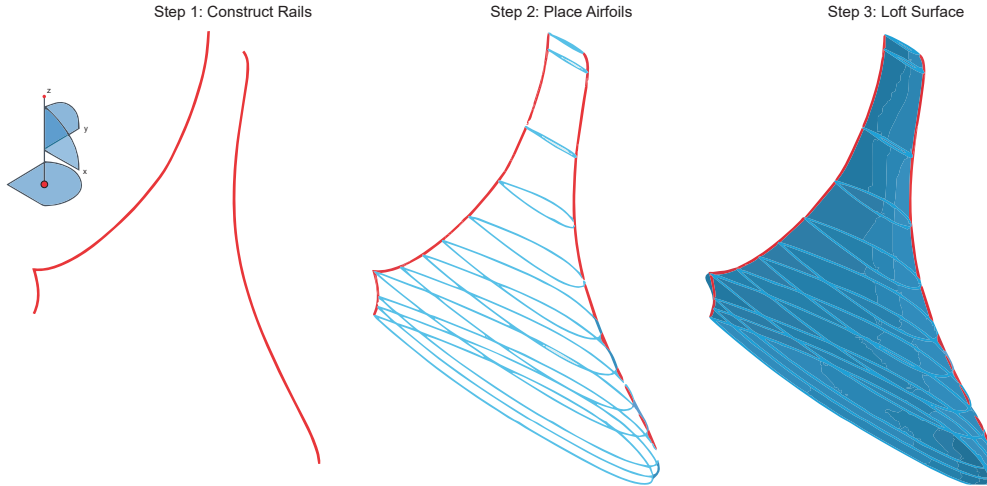


Figure 3.8: Wing Construction Methodology Used in the MMG [70]

### 3.3.4 Consistency Maintenance

Alongside the graph contraction algorithm, it was necessary to augment the slot evaluation mechanism of ParaPy to cope with unserializable values. The problem is best illustrated by Listing 3.1, whereby an identity comparison is performed using an object that not serialized due to being a part of an unserializable container. Here, once the `identity_comparison` slot is evaluated, it will use the cached value of the object, but will at the same time cause the unserializable slot to reevaluate. This returns a new object, resulting in the failure of the identity comparison.

---

```

1  class ConsistencyExample(Base):
2
3      @Attribute
4      def unpickleable(self):
5          return [lambda: None, Circle()]
6
7      @Attribute
8      def pickleable(self):
9          return unpickleable[-1]
10
11     @Attribute
12     def identity_comparison(self):
13         return unpickleable[-1] is pickleable

```

---

Listing 3.1: Example Illustrating the Need for Identity Consistency Maintenance

The working principle of this augmentation is to check the precedents of a slot evaluation for the presence of unserializable caches. If such a cache exists, then all unserializable precedents of that cache as well as their respective dependent caches are eagerly invalidated. Although simple to implement when a slot evaluation calls a single cache, this algorithm becomes a bit more complicated when considering call stacks where previously accessed caches need to be invalidated. To accommodate these situations, all visited caches during an evaluation are collected. If any of these caches would be affected by the invalidation of the unserializable precedents, then the slot evaluation will be terminated. The slot evaluation is then retried from the originating slot.

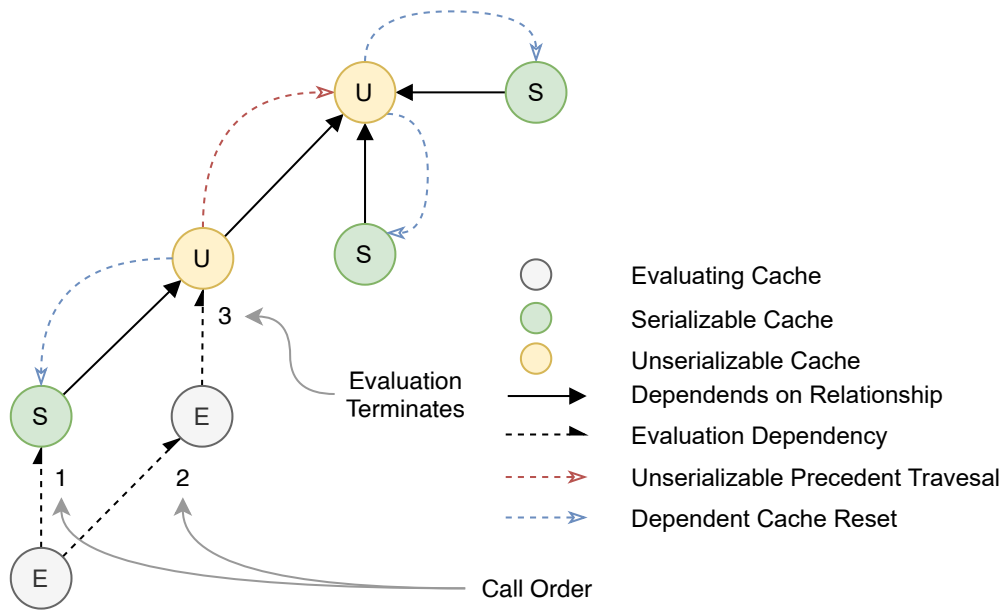


Figure 3.9: Visualization of Consistency Maintenance Algorithm

### 3.4 Process Orchestration

While the information modeling and model persistence phases of the prototype development were significant to being able to orchestrate processes, the primary goal of this research is to enable a knowledge engineer to formalize the intended process of a KBE application in order to improve its collaborative usage. Furthermore, a derived challenge is to be able to facilitate processes flexibly without over-constraining them with control flow.

Before delving into the methodology developed to achieve these goals, this thesis recognizes three different types of workflows: (a) static, (b) dynamic, and (c) emergent workflows. Static workflows are characterized by a predominantly constant control-flow path during runtime. Dynamic workflows are then those which are characterized by how the eventual flow taken in a workflow is dependent on logic. This thesis recognizes that logic can be encoded in the business process, however for simplicity only dynamism encoded inside such inside a central, workflow-steering KBE app in the form of business rules is considered. Examples of such dynamism are given below by Figure 3.10, and further explanation can be consulted from Chapter D.

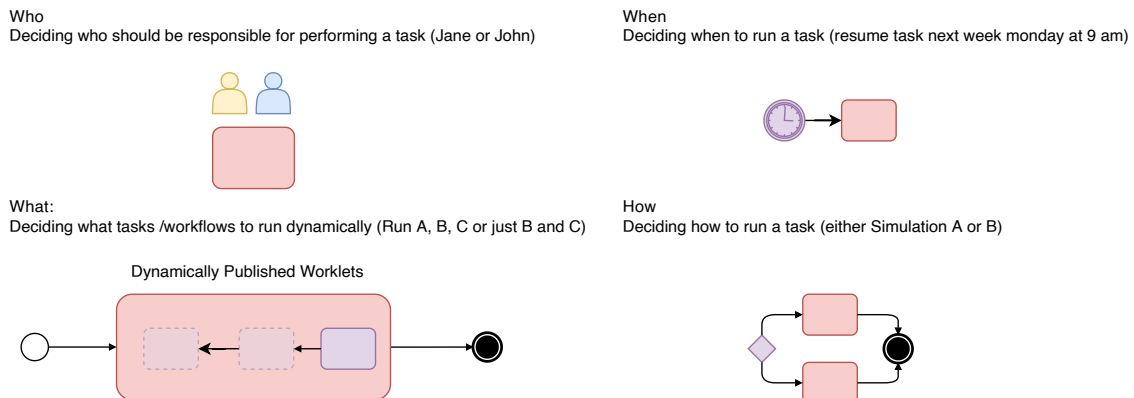


Figure 3.10: Types of Dynamism in Workflows

Additionally, as previously mentioned in the theoretical content, this thesis recognizes two forms of

control flow: (a) forward, (b) backward. Forward control flow, or goal-driven processes are typical when the flow is predictable and (relatively) static. It's characterized as predefined formulation of tasks, and for which the control flow is governed by the process layer. Backwards flow (demand-driven) is typical when the objectives are clear but the means are complicated to define upfront and a result of logic. The latter flow occurs inside KBE applications, which are declarative and where slot flow happens in reverse and will dictate the means. It's characterized by having no predefined formulation of task order, and for which the control flow is governed by the KBE app. These types of flows then result in two distinct forms of dynamic KBE workflows: (a) KBE assisted, and (b) KBE controlled, Figure 3.11.

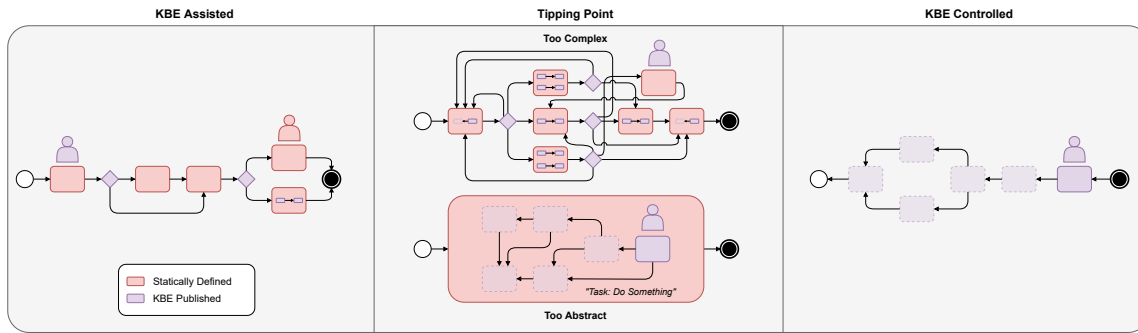


Figure 3.11: Tipping Point Between KBE Assisted and KBE Controlled Workflows

Observing Figure 3.11, one can define KBE assisted workflows as exhibiting forward control flow that is augmented with the aforementioned forms of dynamism. On the other hand KBE controlled workflows are a form of external steering where the slot evaluation, runtime cache, and dependency tracking mechanism of the KBE application is used to dynamically alter the control flow of the process.

Often, one can recognize a tipping point where a certain KBE-driven workflow methodology will be favorable. The tipping point is subjective, but it depends on (a) the ability of process owners to articulate the process, and (b) the fluidity that process owners expect. This is due to how, forward control flow predefines the possible paths that can be taken in a workflow, leading to a form of rigidity and, thus, limited re-use. To illustrate the differences between these two approaches, this thesis will focus on either extreme.

### 3.4.1 Process Modelling

Having defined the taxonomy of this thesis with respect to workflows, the next important topic is to discuss how processes are modelled. This thesis focuses on manual formalization of processes, where a user utilizes BPMN software to compose workflows out of a sequence of tasks. These BPMN models are then linked to the KBE applications through accessing their respective GraphQL APIs. The envisioned methodology has a clear distinction between the responsibilities of the BPMN model and the KBE product model:

- **Process Model:** Define “How” / “When” users and services should interact with the product model
- **Product Model:** Define “What” users and services should interact with and “How” some computed values should be provided (geometry, engineering rules)

This thesis utilized the *Camunda*<sup>ii</sup> a BPM software for both process modelling and orchestration. As shown by Figure 3.12 a visual BPMN editor is used to develop process models. To make it easier

<sup>ii</sup><https://camunda.com/>

for developers to create these models, task templates were defined for typical tasks such as starting, saving, loading, shutting down, and querying a KBE application Figure 3.13.

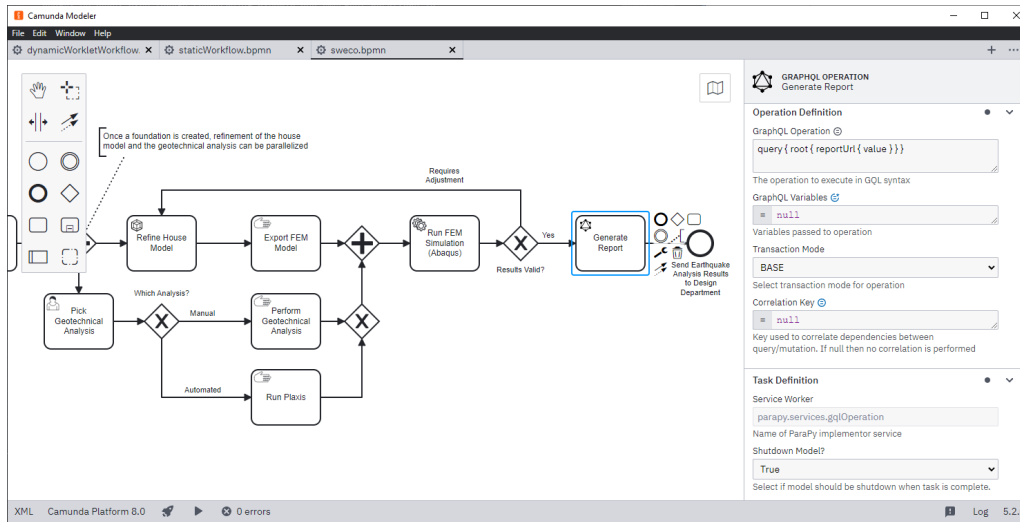


Figure 3.12: Screenshot of Process Modeling / Formalization in Camunda Modeler

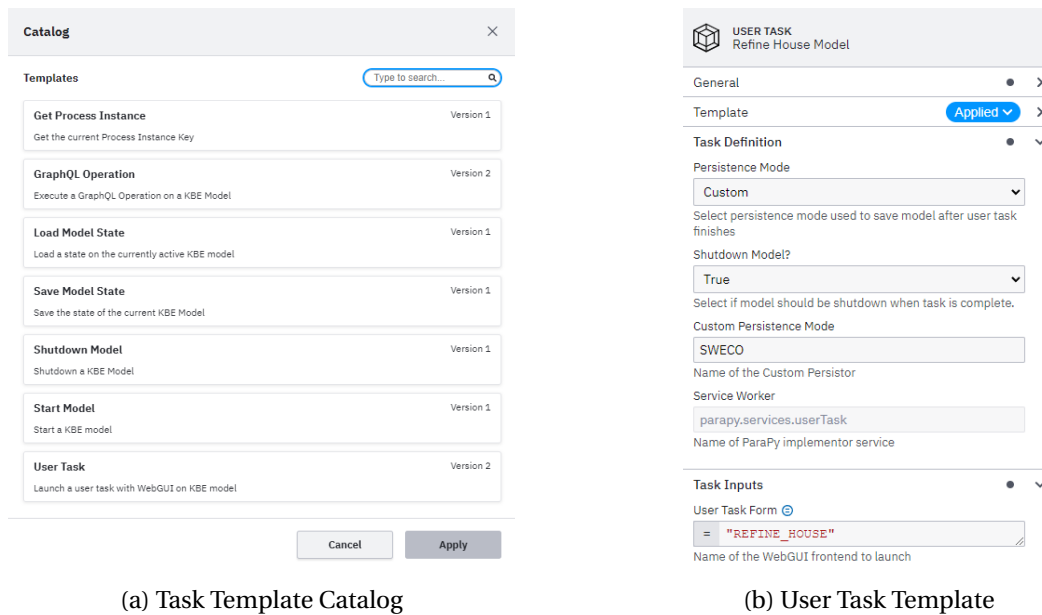


Figure 3.13: Screenshots of Task Template Implementation to Assist Process Modeling

### 3.4.2 Service Architecture

An important enabler of the collaborative KBE software prototype is the adoption of a service oriented architecture which exploits *Camunda Zeebe*'s horizontal scalability model. Although currently, horizontal scaling was out of scope for this thesis, all major components within the architecture: task workers, persistence, KBE application instances, and the workflow engine itself, are decoupled such that they can be scaled independently from one another, as seen in Figure 3.14. Furthermore, in this diagram it is important to note that the generic KBE services are lightweight “controllers” written in Python that do not need the ParaPy runtime to function. They simply send instructions to the model via GraphQL API. To illustrate this, an example task service is provided by Listing 3.2. Furthermore,

simulation tools and arbitrary services can also be defined by the end-user using the same gRPC based-Python client. This allows all involved services within this architecture to be scaled based on load. For example, if a KBE application is very frequently used, then multiple service instances can be deployed to cater to this demand.

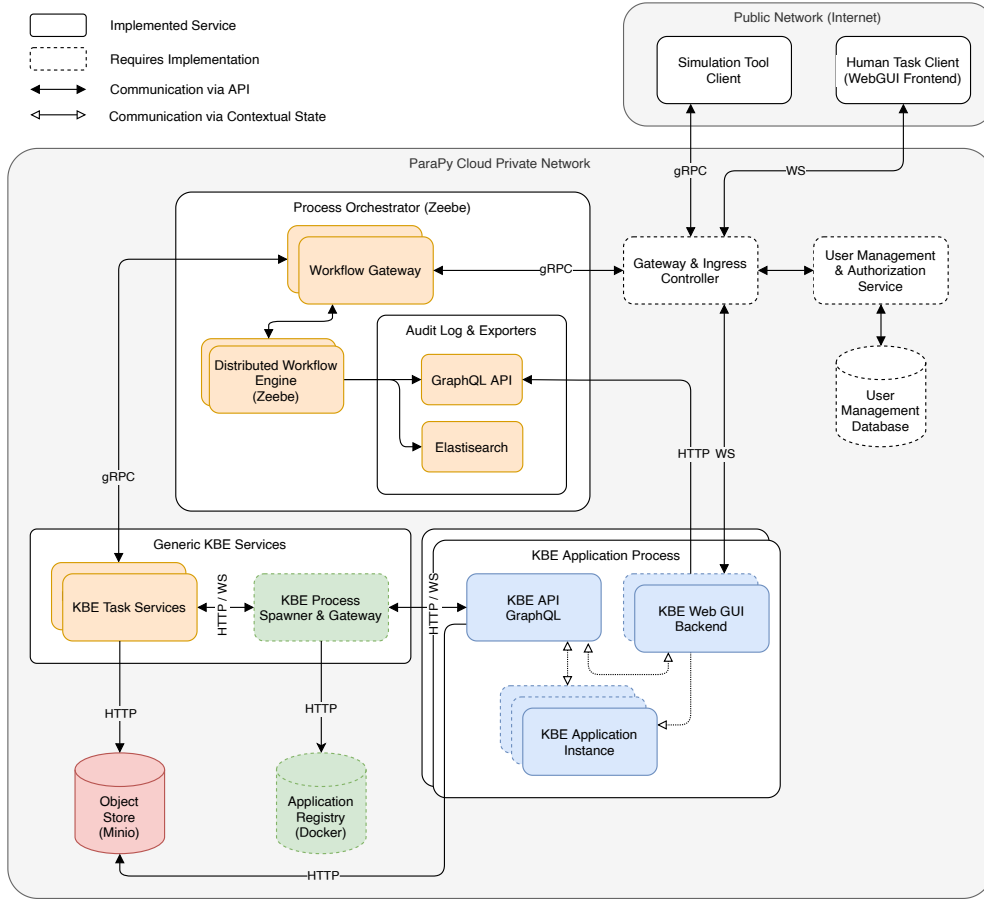


Figure 3.14: Service Architecture of KBE-WfM Synthesized Software

```

1 @router.task(task_type="parapy.services.saveModel", timeout=TIMEOUT)
2 async def save_model(job: zeebe.Job, model: Model, logger: Logger) -> None:
3     persistor = get_persistor(job) or "PICKLE"
4     state = await model.save(persistor=persistor)
5
6     job.stateVariables["model"].update(
7         {"state": state, "persistor": persistor}
8     )

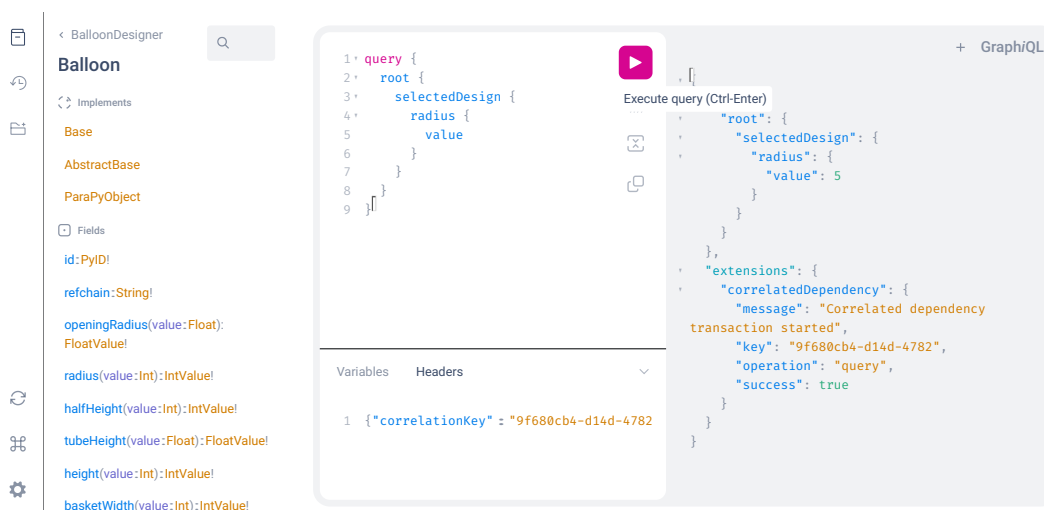
```

Listing 3.2: Example of a Generic KBE Task Service

### 3.4.3 Correlated Dependencies

To support the need of flexibly integrating tools with KBE applications, a methodology has also been developed to be able to maintain dependency tracking even when changing (mutating) the values of a KBE application from an external tool. The basic principle is to track the queried inputs of a tool, with a `correlationKey` as seen in Figure 3.15a. Subsequently, when the same key is used on a mutation, the KBE application treats the mutated caches as dependent on the queried caches, Figure 3.15b.

In this example, the height of the balloon becomes dependent on the radius, due to the performed correlated dependency transaction. This concept is used later in Section 3.4.7.



(a) Start of Transaction



(b) End of Transaction

Figure 3.15: Depiction of How a Correlated Dependency Transaction is Executed

### 3.4.4 Data Management

Another important aspect to mention in process orchestration is the management of data to capture the evolution of changes throughout the process. Zeebe, make use of JSON process variables that are scoped to their current process instance and logged in the so-called audit log. These process variables are used to execute business logic in the process models. Loosely, this can be envisioned as a sort of “Git” versioning approach for processes. However, Zeebe’s use of an in-memory database means that the process variables are limited to 34 MB [65]. As a result, it is necessary to store large files and persisted models externally to the process engine. For this purpose *Minio*<sup>iii</sup> an open source object store, is used. Furthermore, to keep track of external data, several `stateVariables` as defined shown by Figure 3.16 are used to track the latest state of the KBE application within the process instance.

<sup>iii</sup><https://min.io/>

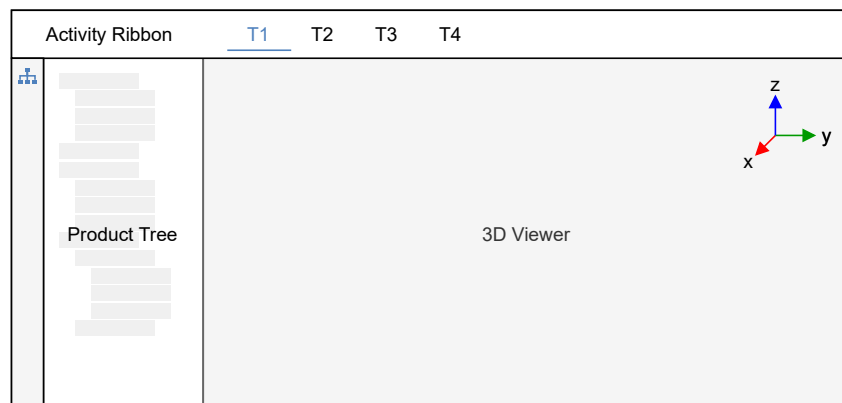




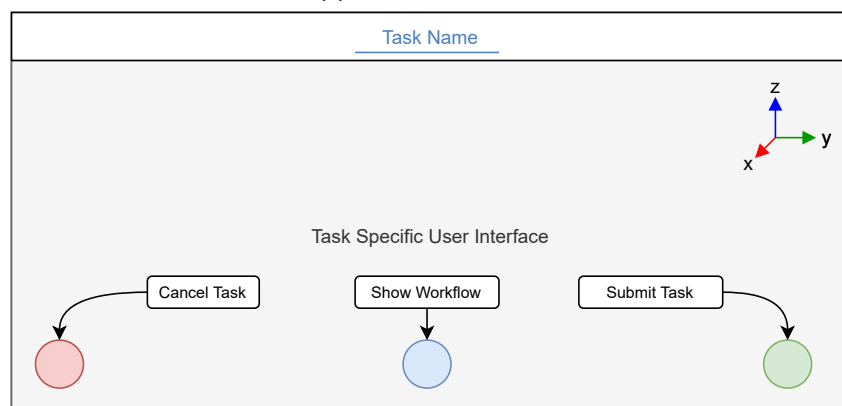
Figure 3.16: Example Process Variables from a Process Instance

### 3.4.5 Handling User Tasks

The final component to facilitating collaboration on processes is to be able to break apart the monolithic user interfaces of traditional KBE applications into task specific interfaces intended for one person. This difference is highlighted by Figure 3.17.



(a) Generic Interface



(b) Task Specific Interface

Figure 3.17: Comparison of Generic KBE User Interface vs. Task Specific Interface

Here, the emphasis is placed on allowing the developer to place anything within the canvas using ParaPy's WebGUI technology. However, to facilitate tasks, three buttons are added by default to the user interface. Namely, these are buttons to cancel or submit the task and to launch a viewer of the current active workflow. The way in which these user interfaces are launched during a process is depicted by Figure 3.18. The working principle relies on the use of a GraphQL Subscription, which makes use of a websocket connection to maintain a connection between the launched user-interface and the task service. Once a user-task is required, the task service sends a subscription query, Figure 3.19, with task inputs. Once a connection is established, the KBE Application launches a WebGUI, and responds to the task service with a dynamic URL. This URL can then be sent to the user via any notification service. Once the user finishes their task, the open subscription completes with the outputs of the task.

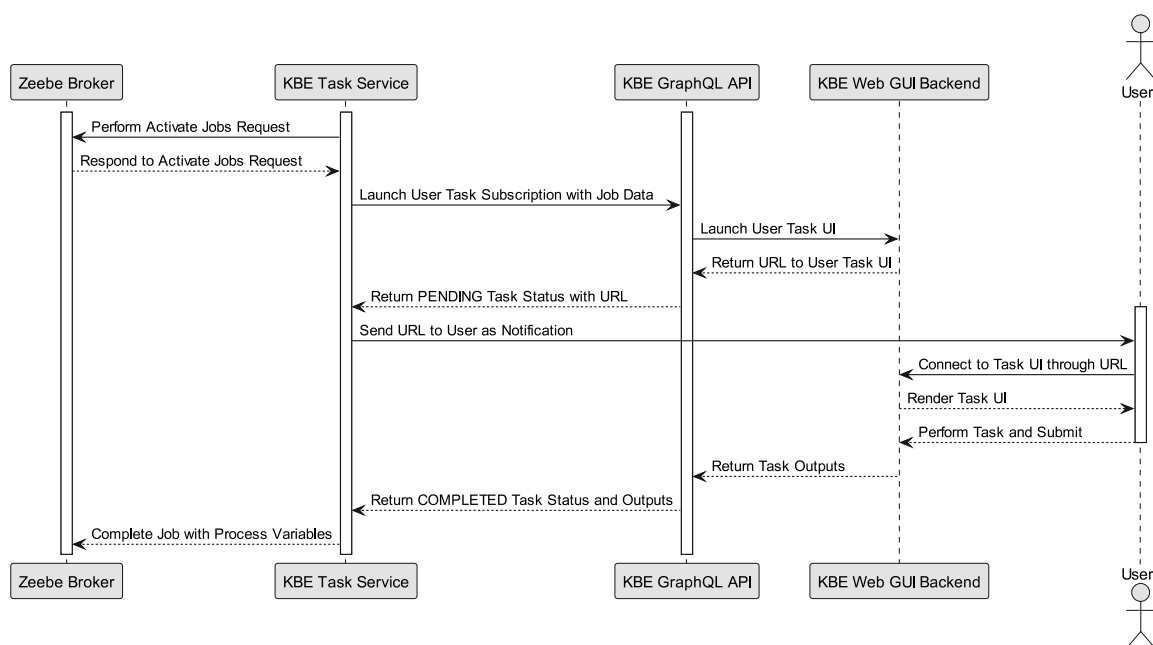


Figure 3.18: UML Sequence Diagram Depicting User Task Handling

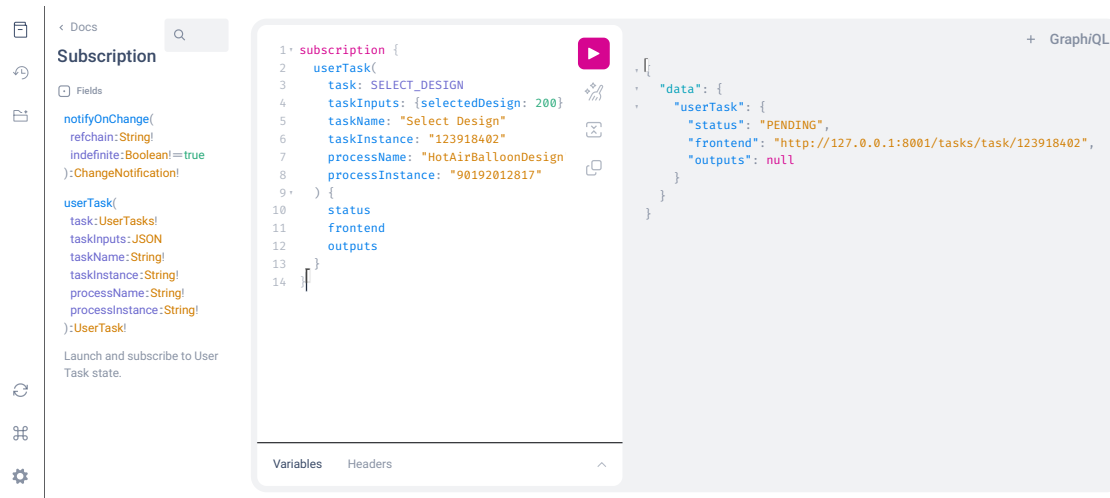


Figure 3.19: Depiction of How a User Task Subscription is Launched

### 3.4.6 Dynamic Workflow

To support the ultimate form of dynamicism, referred to as KBE controlled workflows, the KBE application can be responsible for publishing instructions to the workflow manager to execute certain workflows in a demand-driven manner. From the literature study, it was observed how “worklets” which are small workflows to achieve a single goal, can be used to increase process flexibility. This research extends on this concept put forth by [61, 64], by linking worklets to slots within a KBE application. The goal is to allow runtime caching and dependency tracking to be used to influence the execution of these worklets to prevent re-doing tasks unnecessarily.

To accomplish this, the Worklet decorator was added to the ParaPy SDK. Figure 3.20 below, demonstrates this decorator through a basic use-case that illustrates how the slot evaluation mechanism of the KBE application is used to dynamically chain together worklets, which are defined by developers in BPMN. In this example, if one were to query for the volume of the cylinder, then the `top_area` slot will be called, which in turn is another worklet. This results in the `calculateArea` worklet being published to the process orchestrator, which receives inputs from the KBE application, executes the worklet, and then returns its outputs. After the outputs are received by the KBE application, they are cached on the `top_area` slot. Therefore, when the initial query for the volume is repeated, the `calculateArea` worklet is skipped, and instead the `calculateVolume` worklet is published. Due to the re-use of the slot evaluation mechanism of ParaPy, the runtime dependencies between these cached values is known. Therefore, these worklets are only re-run if their precedent slot values change. This concept enables dynamism as task execution order becomes tied to the dependencies within the KBE application and not to the process model. For example, if there would be a scenario where only the height of the model is modified in a subsequent task, then only the `calculateVolume` worklet would need to be re-run by the system.

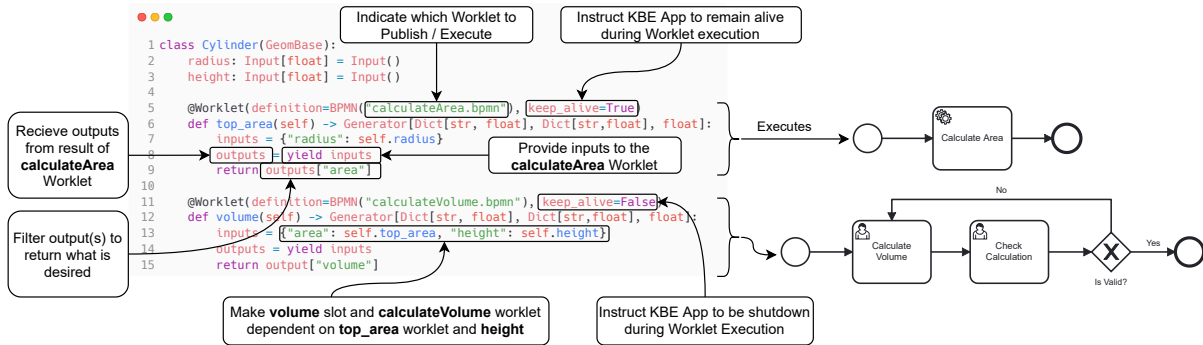


Figure 3.20: Worklet Decorator Developed for the ParaPy SDK

The worklet methodology entails that a KBE application publishes a message to the process orchestrator containing the inputs and process model required to respond to the initial query of the task service. Within the same message, the KBE application provides information to the process orchestrator to know if it can be shutdown. Once the worklet finalizes, if the KBE application was shutdown then it is first relaunched. Afterwards, the KBE application spawns a background thread to poll the process orchestrator for a finalization task. Since service tasks in Zeebe poll for a given task name, the finalization task contains a `workletKey` unique for that given worklet. A detailed explanation of the sequence flow can be seen in Figure 3.23. Overall, the worklet concept allows the KBE application to establish external steering. Even though it is not materialized as such in the process orchestrator, one can think this chain of worklets as a First In Last Out (FILO) call stack. This is because the execution order of worklets is dictated by the logic encoded within the slot methods. Furthermore, since these slot methods happen to often call other slots, a call stack is formed within the KBE runtime. With this stack analogy, worklets execute sequentially until the initial demand is satisfied.

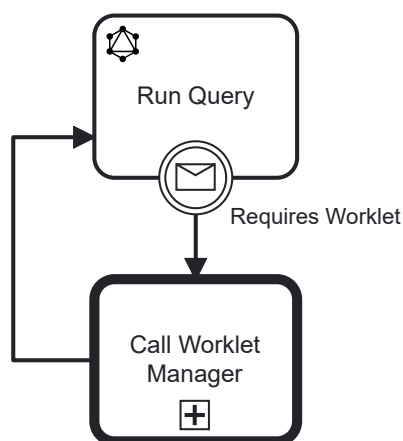


Figure 3.21: BPMN Pattern Created to Enable the Worklet Concept

To enable the worklet concept, a boundary event must be added to the query task to intercept the publication message by the KBE application. This boundary event can be seen in the Run Query in Figures 3.21 and 3.24. The use of this event, makes worklets an opt-in functionality, meaning that it is possible to restrict the workflow to only running pre-defined tasks if desired. In this situation, an exception will be raised that the query cannot complete without the worklet. Although it would have been possible to introduce the worklet concept through message brokers, using BPMN to accomplish this orchestration makes communicating the process model to non-developers easier. The hypothesized benefits of the worklet concept include: (a) ability to run simulation outside of the KBE model while maintaining dependency tracking, (b) Ability to intermix human inputs with KBE slot evaluation (conversational composition), and (c) ability to extend KBE slot evaluation behavior in process models.

### 3.4.7 Emergent Workflow

The ability for a workflow to be completely agnostic to task order is a powerful concept to support highly dynamic business processes that change from day to day to suit requirements. Support for such processes in BPMN is accomplished with ad-hoc subprocesses which allows CMMN modeled sub-processes to be embedded in BPMN models. Since ad-hoc subprocesses are not currently supported by Zeebe, they were not directly researched in this thesis. Nonetheless, to support the functionality of an ad-hoc subprocess, a methodology was developed to allow a tools or humans to run anything for a given task while still exploiting the KBE application dependency tracking mechanism to afterwards derive a logical sequence of tasks and what data those tasks depend on.

In this research, the ability to derive a logical order of tasks post-execution is referred to as the Emergent Workflow. The working principle is that the KBE dependency tracking system correlates queried slots to mutated slots in a so called correlated dependency operation. These correlated dependencies associate the transformation that was applied to the KBE model to the individual that performed them. This allows one to derive an order of tasks after execution that was not explicitly modeled. Figure 3.22 below demonstrates how the query and mutation applied by two ad-hoc tasks allows these tasks to be placed sequentially based on these data dependencies.

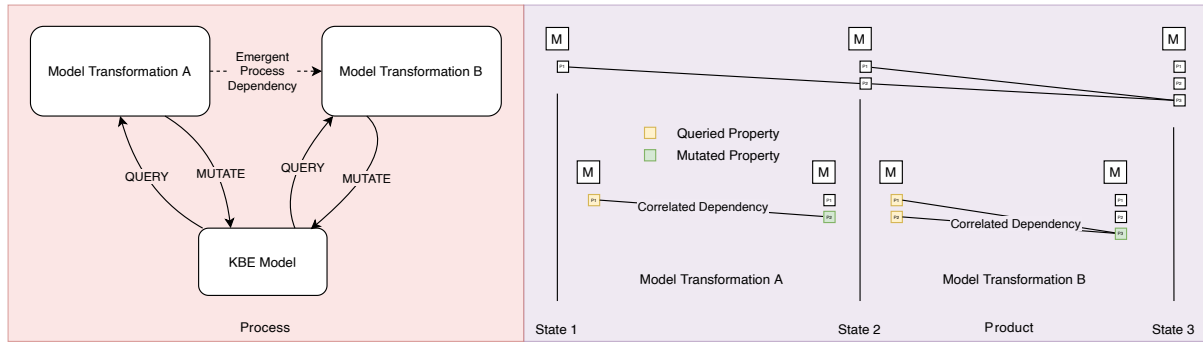


Figure 3.22: Correlated Dependency Concept for Emergent Workflow Orchestration

In this example, since Model Transformation B depends on property P2 which is set by Model Transformation A, one can infer that Model Transformation B is dependent on Model Transformation A. As a consequence, one can use this information for several purposes:

- Allow ad-hoc processes to be later formalized into a statically defined process model if desired
- Provide an audit-log of what happened, who set values, and how were those values consumed
- Inform users of what tasks will require re-evaluation if a change is made
- Provide an impact analysis of approximately how long a proposed change to the design will take.

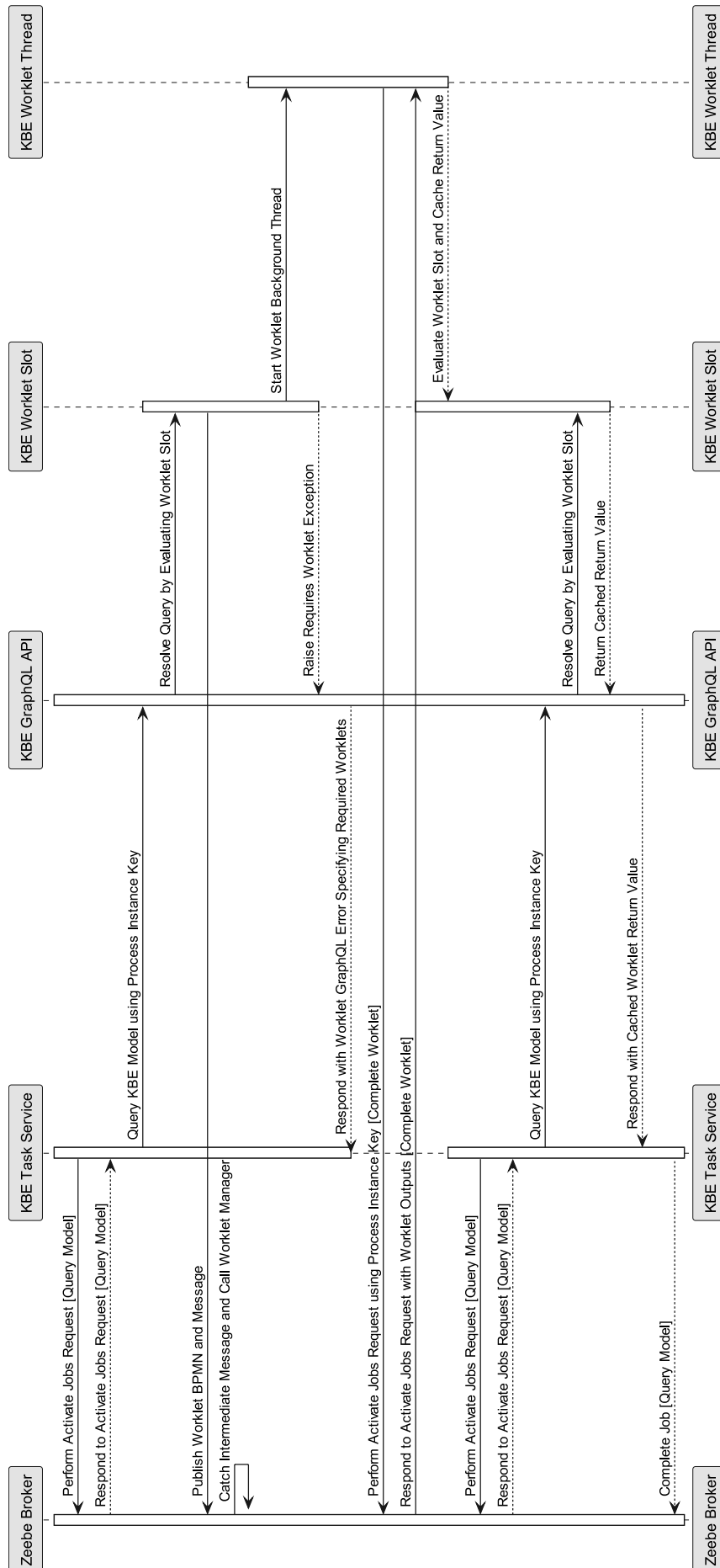


Figure 3.23: UML Sequence Diagram Depicting Worklet Execution

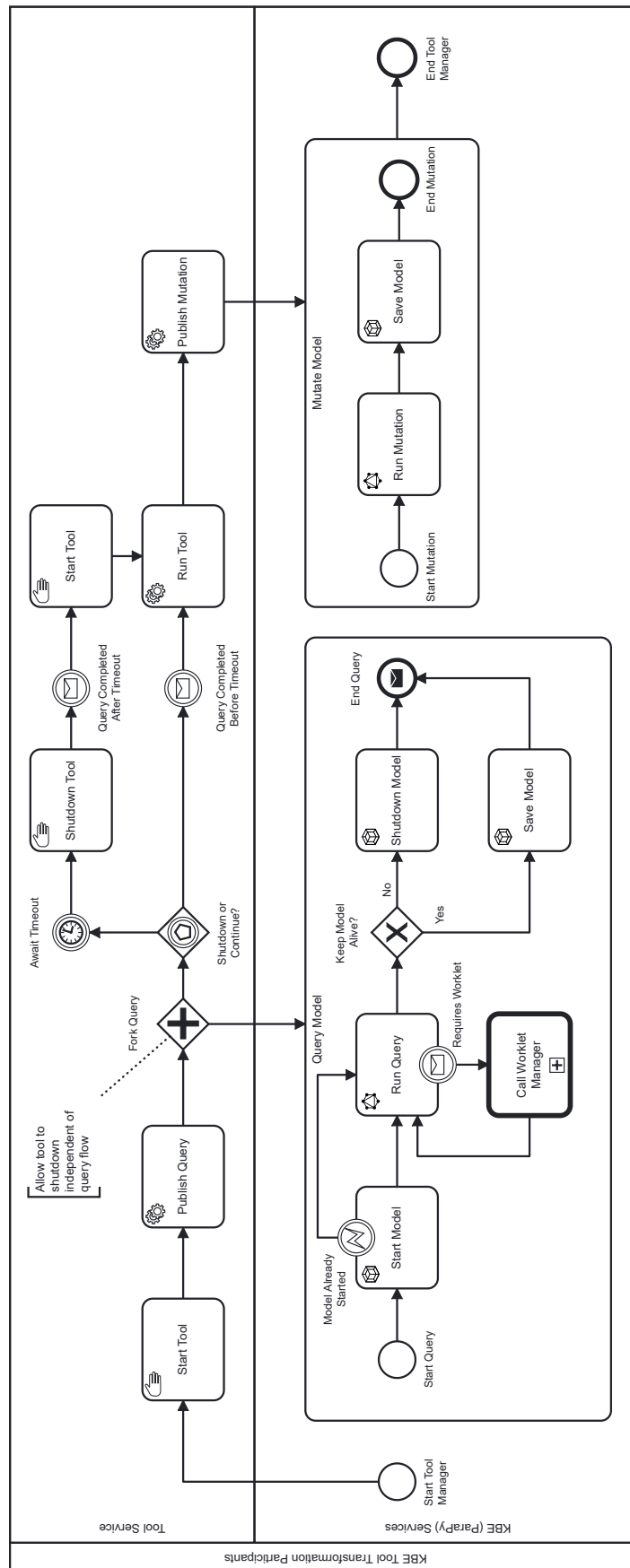


Figure 3.24: Process Model of a Generic Tool Transformation

## 4 Results

After developing the software prototype, in order to verify its proper function several experiments were performed. Similar to the Chapter 3 section, this chapter will concentrate on explaining the results of these experiments on the topics of (a) information modeling, (b) model persistence, and finally (c) process orchestration. After the presentation of these results, an explanation of the verification steps before and after these experiments is explained.

### 4.1 Information Modeling

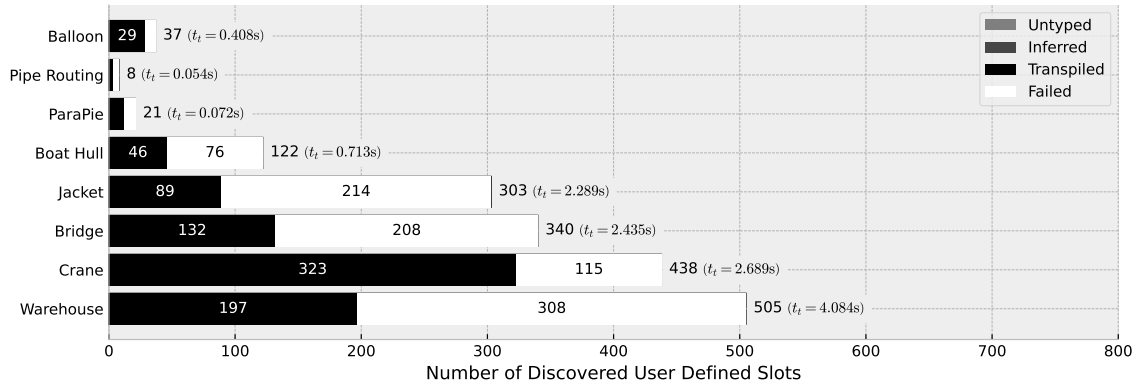
The starting point for coupling a KBE application to a business process starts with the information modeling phase. To test the performance of how well the GraphQL transpiler can expose the information of a KBE application, 8 example applications were selected. These applications were specifically selected by their complexity measured in terms of the number of user-defined slots in the source code. The purpose was to see if the transpiler was able to consistently represent the information of a KBE application across varying code-base sizes.

Furthermore, these experiments were repeated with different configurations to observe the effect of type inferencing and the allowance of dynamic typing. The latter should not be confused with the dynamic type in KBE applications, and instead refers to the availability of a strong type for a given slot. Strong typing can be obtained in three ways: (a) deriving it from the default value, (b) type annotations, (c) type inferencing. This means that the transpiler is able to ahead-of-time (i.e. before runtime) determine the return type of a slot instead of representing it as JSON which has the potential to return any data type, granted that it is serializable. When dynamic typing is allowed, the functionality of the GraphQL API, resembles the REST API of ParaPy, whereby a user is not able to know what the exact data type of a slot is when formulating a query.

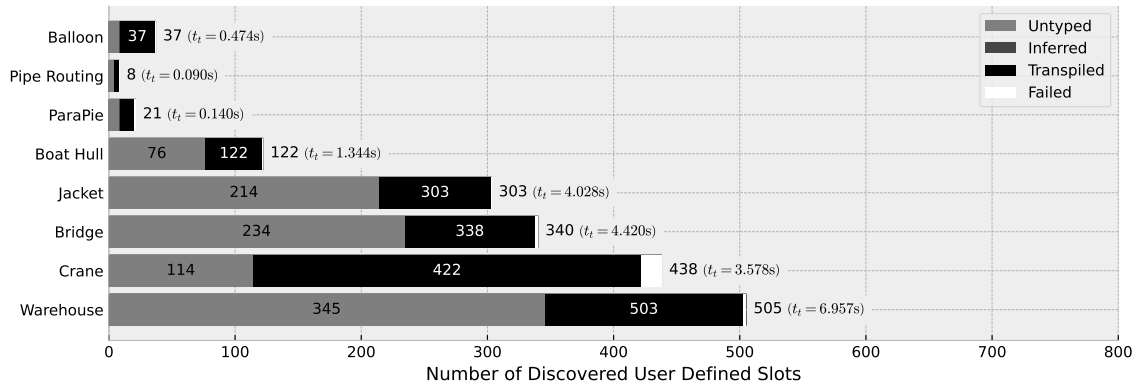
The first experiment, given by Figure 4.1a shows that the majority of applications are only partially representable in the GraphQL schema. The only two exceptions to this in this experiment were the balloon application and the crane application, which notably made strong use of default values in most input slots. By contract, the remainder of the applications often had required inputs with no default value, which meant that it was possible to derive the expected return type of these slots. As a result, a significant portion of the applications were not representable in the schema. On the flip side, when dynamic typing is allowed, given by Figure 4.1b, the previous failures are almost all resolved, with more than 90 % discovered slots being representable. However, use of these dynamic typed slots carry the risk of runtime errors in the situation that a slot return value is not JSON serializable. Nonetheless, use of dynamic typing in the schema allows more slots to be queried. It is also worthy to note that the transpilation time,  $t_t$ , increases with the number of representable slots. This is due to the necessity to create more GraphQL object types to represent these slots in the schema.

The same difference between statically typed, and dynamically typed schemas can be seen in the third and fourth experiment, given by Figure 4.1d respectively. However, in both of these experiments, type inferencing was enabled. This resulted in the ability to discover more user-defined slots contained within classes that were previously untraversed by the transpiler due to lack of return types from other slots linking to these classes. However, this comes at the cost of increased transpilation time, due to the need to run the type inferencer, represented by the inferencing time,  $t_i$ . Interestingly, for simpler applications, the majority of transpilation time is due to type inferencing. Whereas on larger applications, such as the warehouse application, the time required for type inferencing is overshadowed by the time required to create GraphQL object types. Chapter B provides a profiling result to clarify this finding.

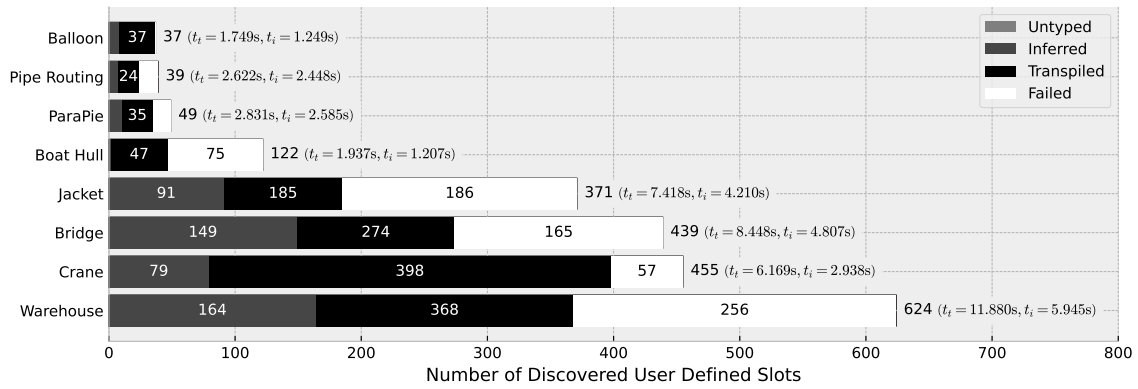




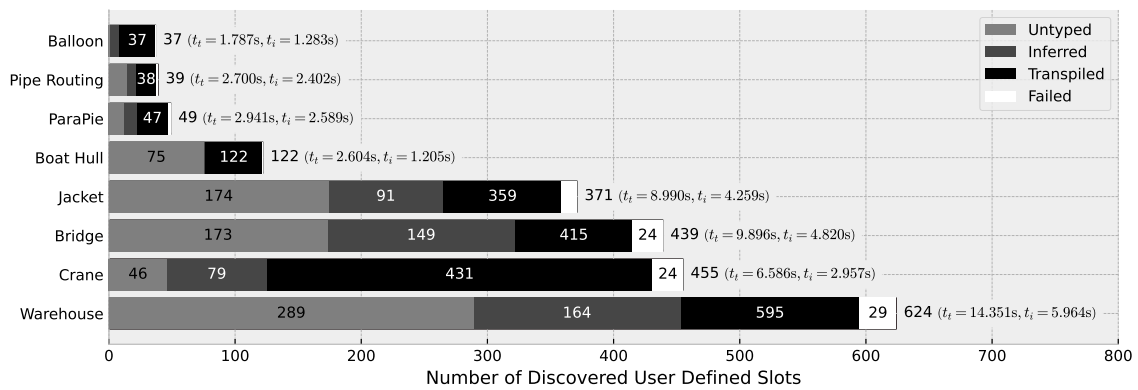
(a) No Inferencing, Static Typing Enforced



(b) No Inferencing, Dynamic Typing Allowed



(c) Inferencing Enabled, Static Typing Enforced



(d) Inferencing Enabled, Dynamic Typing Allowed

Figure 4.1: Schema Transpilation Performance Measured on Example KBE Applications

## 4.2 Model Persistence

Similar to the testing of the information modeling, the behavior of the developed model persistence extension to ParaPy was tested using several example applications ranging from simple to complex geometries, as shown by Figure 4.2. The purpose of these experiments was to determine if the aforementioned problems in Chapter 3, due to the recursion limits of Python, were overcome by the implemented approach. This testing was important to guarantee that ParaPy models are persistable no matter the complexity of the application.

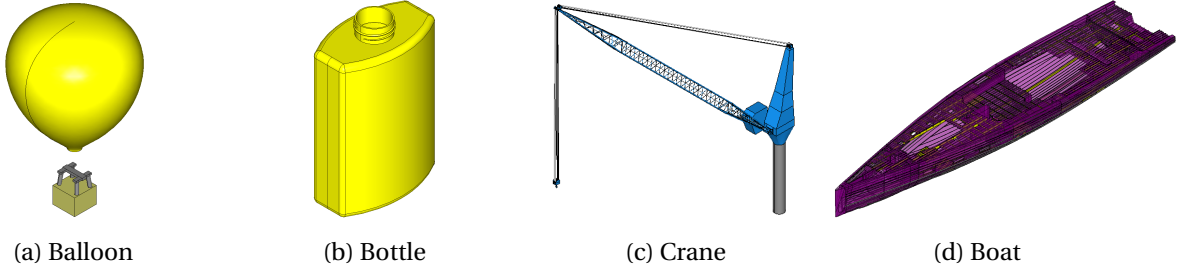


Figure 4.2: KBE Applications Used to Test Model Persistence

These example applications are then benchmarked based on several timings: the evaluation time,  $t_e$ , the serialization (pickling) time,  $t_p$ , the deserialization (unpickling) time,  $t_u$ , the re-evaluation time,  $t_r$ , and finally, the time gained by serializing  $\Delta t$ . The latter is measured based on the difference between the sum of the evaluation and serialization time with the sum of the deserialization and re-evaluation time. In essence, this shows how much time could be gained by restoring the state of a ParaPy application from a persisted representation instead of recreating it from scratch. The results of this benchmark are provided by Table 4.1 below.

Table 4.1: KBE Serialization Performance on Real Use Cases

Model	$t_e$ [s]	$t_p$ [s]	$t_u$ [s]	$t_r$ [s]	$\Delta t$ [s]	Size [KB]
Balloon	0.01000	0.00700	0.01500	0.00012	-0.01212	49
Bottle	0.12700	0.01000	0.01500	0.00039	0.10161	99
Crane	1.38100	0.52200	0.79300	0.00200	0.03861	8297
Boat	35.36400	5.15100	5.08700	0.02100	23.78800	76498

In the results, one can see how all applications serialize and deserialize faster than the evaluation time. However, in the case of the balloon the application, there is no time gained by loading the model from a persisted representation. This contrasts with the boat application where there is a large time gain of 23.79 s. This can be explained by the geometric complexity of the two models. Whereas the balloon application comprises only of elementary shapes such as boxes, and a revolved surface; the boat application requires the computation of many boolean computations. As a result, the time consumed to perform these calculations is much higher than the time required to serialize the resulting shape.

Besides these real applications, synthetic benchmarks were performed to better understand the time and memory complexity of serialization. Furthermore, these benchmarks were used to determine the dominant parameters influencing these complexities during serialization. Four situations were benchmarked. Namely, (a) full geometry, where the entire geometry is serialized, (b) partial geometry, where only the end-result is serialized using graph contraction, (c) no geometry, where no geometry is serialized, and finally (d), where no geometry is serialized, and a partial evaluation is per-

formed where only a derived property of the geometry is accessed. The geometry used in the presented benchmarks is a sequence of boxes that are instantiated on top of each other, and then fused together using a `FusedSolid`. The partial evaluation in question is accessing the center of gravity of the resultant shape.

These synthetic benchmarks results are provided by Figure 4.3. Since the same geometry has to be created in each of the 4 cases, the evaluation time is roughly constant. What varies is the total restore time which is the summation of serialization, deserialization, and reevaluation times. As can be seen when comparing Figure 4.3a to Figure 4.3b, the lack of persisting intermediate geometries along with the use of the graph contraction algorithm to remove the need to serialize intermediate dependency edges, reduces the total restore time. Furthermore, it also reduces memory consumption since the dependencies between intermediate results are not serialized. In the final two cases, given by Figure 4.3c and Figure 4.3d, one can also see that the  $O(n)$  memory complexity can be reduced to  $O(1)$  in the best case where all intermediate results and geometry are excluded from the serialization. This comes at the cost of achieving no runtime speedup when deserializing as evaluating the geometry will take roughly the same time to re-evaluate, Figure 4.3c. However, in Figure 4.3d one can see the effect of only requesting the evaluation of the center of gravity of the `FusedSolid`. Since no geometry is needed to access this persisted value, the most significant speed-up is achieved in this case.

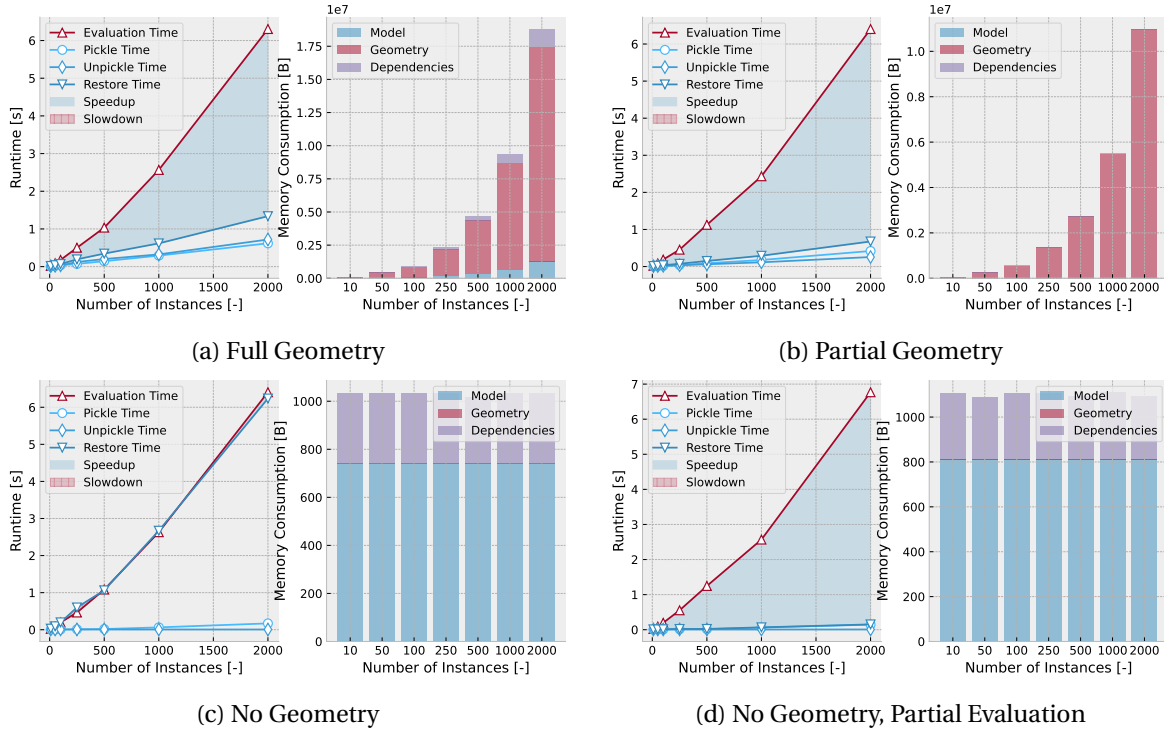


Figure 4.3: Synthetic KBE Model Serialization Benchmark Results

Overall, the synthetic benchmarks reveal that the time complexity of serializing ParaPy KBE models is  $O(n)$ . Since the memory complexity is also  $O(n)$ , one can quickly verify this result by ensuring that the increase in serialization time is proportional to an increase in memory consumption as given by Table 4.1. For example, the boat application consumes roughly 10x the memory consumption as compared to the crane application. This same ratio exists between the two applications for the serialization time. In addition to this back-of-the-envelope verification, a curve fitting library in Python was used to determine whether a linear or quadratic curve fit the measured values. The result was that a linear curve was a better fit of the data provided in Figure 4.3.

## 4.3 Process Orchestration

The process orchestration phase of the experiment is the most significant to verify that the primary goal of this research is achieved: to improve the collaborative usage of KBE applications and facilitating the integration of KBE applications with business processes. For this purpose, two case studies are explored. First the earthquake analysis case study introduced at the start of this report is used to provide a glimpse into a real-world application of the software prototype. However, to concentrate on the differences between static and dynamic workflows, a simpler case study on the design of a hot air balloon is also explored.

### 4.3.1 Case Study: Earthquake Analysis

As presented at the start of this report, the business process surrounding the earthquake analysis involves several users, tasks, and most notably the parallel flow as given by Figure 4.4. To properly facilitate this process as compared to using a monolithic KBE application interface, several task interfaces were defined, which can be seen in Figure 1.2b. Additionally, the external tasks that were previously coordinated by users, have now been explicitly modelled in business process. These are namely, the STEP file upload whereby a CAD draftsman uploads the geometry of the house to analyze, the geotechnical analysis—performed either manually through *Excel* or via *Plaxis*—and the Finite Element Model (FEM) simulation run via *Abaqus*.

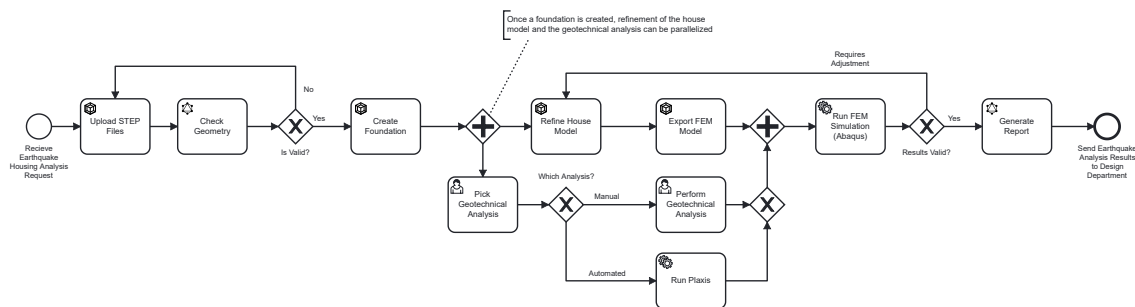


Figure 4.4: Earthquake Analysis Workflow BPMN Diagram

Formalizing these tasks, allows the process orchestrator to be used to “keep the ball rolling” during the process, alleviating the cognitive burden on users to keep track of who is doing what tasks. Furthermore, the segregation of the monolithic user interface into several task specific interfaces, allows a user to more easily understand what they should be doing at each task. In this research, this is referred to as process awareness.

After implementing this case study, which took 3 days of development effort to accomplish, several benefits compared to using the monolithic KBE application were observed. First, it became possible to split apart the house geometry upload and checking logic into a separate tool. This had the benefit of simplifying the KBE application, which no longer required such logic. Additionally, this opens the door for the re-usability of the geometry checker in other KBE apps. Finally, the use of state variables along with the process bucket, allowed data from multiple states of the process to be persisted. Along with the audit log of the process orchestrator, this makes it possible to retrospectively understand how the final results in the generated report was obtained.

### 4.3.2 Case Study: Hot Air Balloon

Although useful to verify the function of the software prototype on a real-world application, the complexity of the earthquake analysis case study proved detrimental to effectively testing workflow flexibility due to the time required to develop the workflow itself. Therefore, to practically test the specific characteristic of worklets at increasing process flexibility, a simpler case study was required. For this purpose, an example developed by ParaPy, the Hot Air Balloon KBE application was selected as it exemplifies a Made-to-Order (MTO) design process, whereby a customer places an order, and the design department works to create a design that fulfills the requirements of that order. As a baseline, the application has 4 steps: Design of Experiment (DoE) Generation, Design Selection, Design Refinement, and Reporting, Figure 4.5.

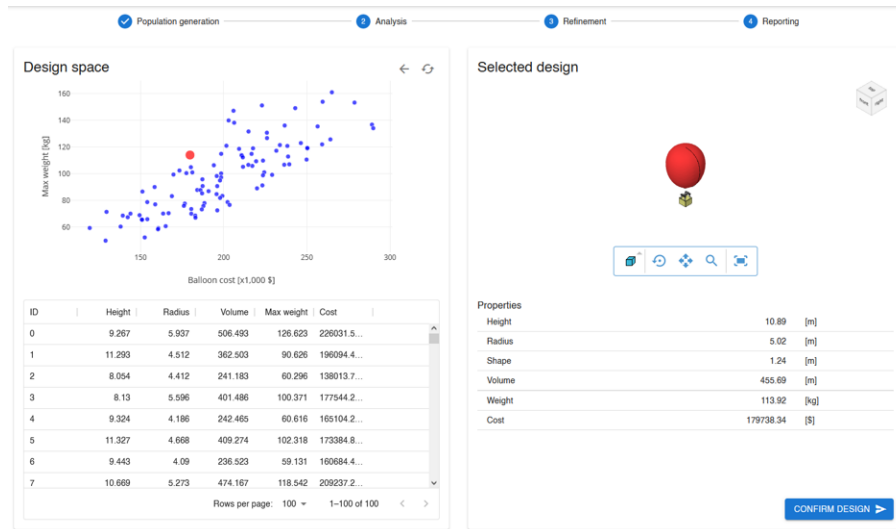


Figure 4.5: User Interface of the Monolithic Hot Air Balloon Application

Applying the same methodology as the earthquake analysis, the resulting formalization of the process is given by Figure 4.6. As can be seen in Figure 4.6 the application is a natural fit to integrate into a collaborative process as several decision and approval gates could be added. This contrasts with the original application which defined an intended sequence of tasks via the “stepper” widget at the top of the user interface. As a result, the original application did not enforce a control flow for how the application should be used. Although this made it possible to support ad-hoc usage, as users could freely navigate between the various steps of the application, the lack of representing these conditional flows in a process model, did not allow users to understand what will happen in the process based on certain decisions.

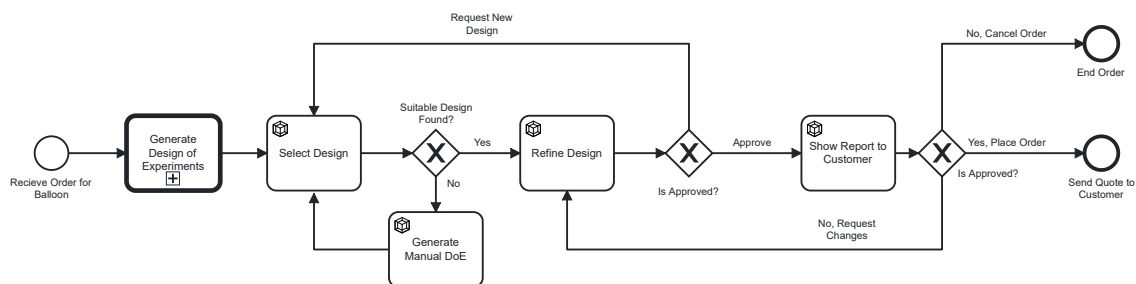


Figure 4.6: Hot Air Balloon Case Study BPMN Diagram

An important result from this case study is to realize that the addition of control flow to model decisions within the process as well as the segmentation of the process into tasks, ends up enforcing a rigid process that inhibits ad-hoc usage. This is because, whereas before users would be able to flexibly navigate between steps of the application, they are now forced into following the process defined in Figure 4.6. For example, a user was previously able to “jump” from the reporting step back to design selection. In the modeled static workflow, to accomplish the same, the user must first propose changes, leading back to the refine design step, where another user must then reject the design in order to end up back at design selection.

To counter this rigidity, the worklet concept discussed previously by Section 3.4.6 was applied. Specifically, the various segments of the static workflow were separated into worklets as depicted by Figure 4.7. Furthermore, the Worklet decorator was applied to modify the source code of the balloon application to link slots to these worklets. The resulting dynamic workflow is then given by Figure 4.8. Note how the entire design process is now handled by a single call activity, and a new task to apply changes is added if the design is rejected.

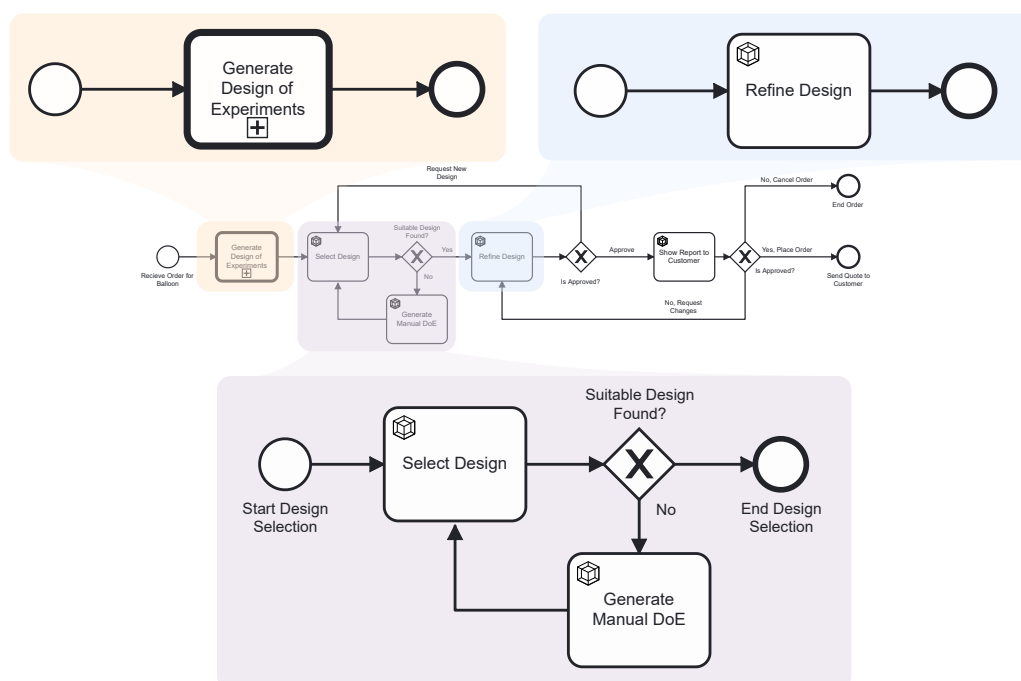


Figure 4.7: Segmentation of Hot Air Balloon Design Process into Worklets

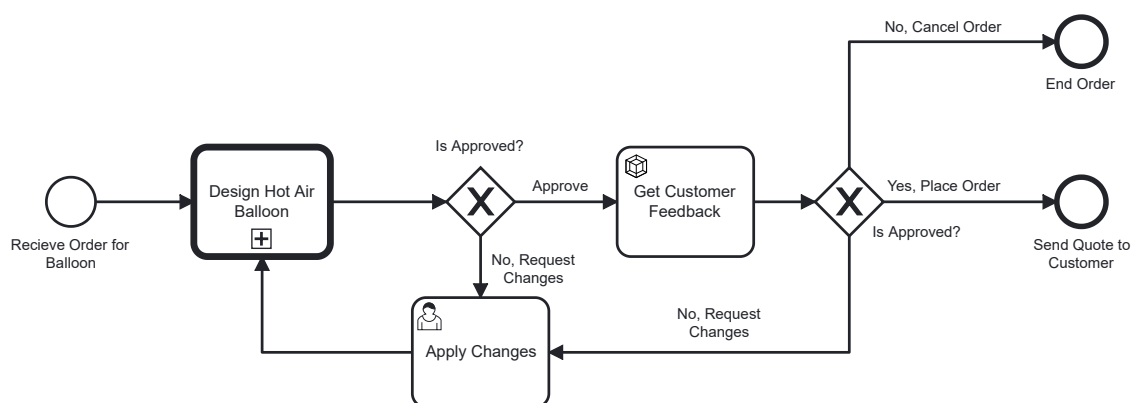


Figure 4.8: KBE Controlled Hot Air Balloon Case Study BPMN Diagram

The dynamic adaptation of this process is then an example of a KBE controlled workflow as the design process control flow is handled by the KBE application. Hence, the KBE application is able to exploit its runtime caching and dependency tracking mechanisms to externally steer the process. The result is the ability to support ad-hoc design activities, similar to that of the original application, while still benefiting from process orchestration for portions of the business process. Take for example how the design call activity in Figure 4.8, which represents the dynamic portions of the workflow, co-exists with the static portions of the workflow representing the customer feedback loop. As a result, depending on the feedback of the customer, the applied change to the design determines which worklet executes. For example, if only a change to the color of the balloon is requested, then no worklets are required to run. However, a change to the dimensions of the balloon require the design refinement worklet. These aspects are discussed further in Section 5.3.

## 4.4 Verification

The most important goal of this research was to improve the collaborative usage of KBE applications through process formalization and orchestration. Since experimentation in this research has been limited to isolated case studies, it is not possible to validate the software in real-world usage. Nonetheless, the case studies allow the requirements of the software prototype to be verified. For example, the earthquake analysis case study allowed verification of the general principle for how a monolithic KBE application can be modified and later integrated into a real-world use case. Furthermore, it allowed assessment of how well the developed software prototype can adapt to new features and functionalities. Besides highlighting the differences between static and dynamic workflows, the hot air balloon case study was also used to verify the functionality of external tool integration into a KBE application. Whereas in the original application, the DoE generation step was embedded into the KBE application, in the modeled process, this logic is externalized into a tool transformer, enabled through Figure 3.24. This allowed testing of the correlated dependencies method to maintain dependency tracking, even for external tools.

Besides the case studies, the entire development process of the software prototype was verified using automated unit tests written and run continuously in compliance with Test Driven Development (TDD). The most technically challenging functionalities: GraphQL transpilation and model serialization were tested rigorously using this methodology. In total 272 tests were written, 154 for the transpiler and 118 for the serialization capability. To quantify the quality of these tests the code coverage metric was used. This demonstrates the percentage of source code lines touched by tests compared to the total number of source code lines. The code coverage metric, which provides a percentage of source code lines that are covered by tests, was 74% and 97% for the transpiler and serializer respectively. Achieving higher code coverage on the transpiler was less critical as verifying schemas visually was much easier than testing the intricacies of the graph contraction algorithm. Finally, as explained in the beginning of the chapter, several real-world use cases were used to verify the time complexity of serialization; demonstrating that even the most complex geometry, the Royal Huisman Boat Application, was serializable, as demonstrated by Table 4.1.

On the service and process orchestration side, manual integration testing was conducted during development to verify the functionality of all services. For this purpose, the audit log and fault tolerance capabilities of Zeebe were helpful in revealing incorrect variable names, invalid workflow token execution, and improperly configured tasks. Furthermore, a demonstration of the capabilities of the software was given live to employees of ParaPy. Here fellow software engineers had the opportunity to ask questions and verify that the software was exhibiting the intended behavior as desired since the start of the project. Namely, the ability to flexibly integrate KBE applications within business processes.



# 5 Discussion

After conducting experiments on the developed software prototype, this section will discuss these results individually before providing answers to the posed research questions. Afterwards, the contributions of this research will be discussed along with its limitations.

## 5.1 Information Modeling

An important cornerstone of this research is the ability to expose the information of a KBE application through a strongly typed GraphQL API. This allows any service in a collaborative process to access and change the data of a KBE application to be able to use it as an information backbone in workflows. Through experimentation on various case studies as well as the example applications, it was observed that KBE applications at the moment do make use of extensive type-hints. As a result, an important technology to be able to represent types in a schema statically has been the use of type inferencing. This technology has not only allowed the discovery of more slots in KBE application source codes, but also increased the percentage of these slots that can be represented statically.

While use of dynamically typed slot return values is possible, it is the opinion of the author to discourage such usage as it burdens tool developers with additional type checking and increases the chances for mistakes. Overall, for the case studies experimented on in this thesis, the use of a KBE application as an information model within workflows functions properly, however the generated schemas are often not catered to use within a process. The main issue is that the current generation of KBE applications have been developed without the present use-case in-mind. As a result, the generated information models are either too verbose to be consumed easily in a workflow or, do not contain the process specific values needed to make decisions within the process model. Therefore, either the development of KBE applications in the future should make use of this technology during development to model the application around the intended process, or a “wrapper” ParaPy class can be used to embed the original application into a suitable representation for process orchestration.

## 5.2 Model Persistence

The result of generic serialization capability added to ParaPy as part of this thesis shows that of the applications sampled, all of them are able to be persisted. The creation of the persistence capability was perhaps the most difficult endeavor of this work. However, several fruitful ideas have come out of this effort to take advantage of KBE’s unique position of dependency tracking to accomplish partial persistence capabilities.

Besides creating an extendible architecture persistence capability, two important achievements were obtained: the graph contraction algorithm, and the consistence maintenance methodology. These two concepts combined provide a powerful way to deal with serializable values as well as enable users to selectively choose which parts of their application they would like to save. From a more critical perspective, the use of `pickle` means that a KBE application must be alive in order to be queried for information. While such a limitation is not critical at the moment, the memory consumption of hosting ParaPy applications in the cloud might mean that the current methodology results in higher cost as compared to using a database to store application data.

Especially with the earlier remark on how the full information model of a KBE application is not necessarily useful in workflows, it might be prudent to explore the creation of lighter-weight alternative to the full KBE runtime. Such a lighter-weight alternative would not come with a CAD kernel, but instead support the core technologies of KBE. Regardless of the implementation however, use of per-



sistence technology has been vital to the implementation of the worklet methodology to increase process flexibility. As a result, the author believes that such a breakthrough in database technology is required to support these workflows. For lack of a better word, there is a need to create a tailored engineering database that has the ability to calculate computed values, if needed launch a CAD kernel, and track dependencies between data elements.

### 5.3 Process Orchestration

The results show that the primary goals of the research have been fulfilled. On both fronts, namely static and dynamic workflows the integration of KBE into business processes has been demonstrated. Furthermore, the hypothesis of this research related to the ability of external steering with a KBE application to improve the flexibility of a process has been verified. This was observed in the hot air balloon case study by how the static definition of the process resulted in constraining the control flow to a rigid way of working. This meant that during more ad-hoc design activities where simply the KBE application is desired to be used flexibly, users would have been forced to go through the order of steps defined in order to perform an iteration Figure 5.1.

This contrasts with the use of the worklet methodology, which allowed for the design process to make use of runtime caching and dependency tracking to cope with ad-hoc changes. For example, when a query was performed to change the population size of the initial design of experiments, the full three worklets were published. However, for a simple change for modifying the radius only the Refine Design worklet was published. This put less of a burden on users as only those individuals necessary to perform the given design modification were notified.

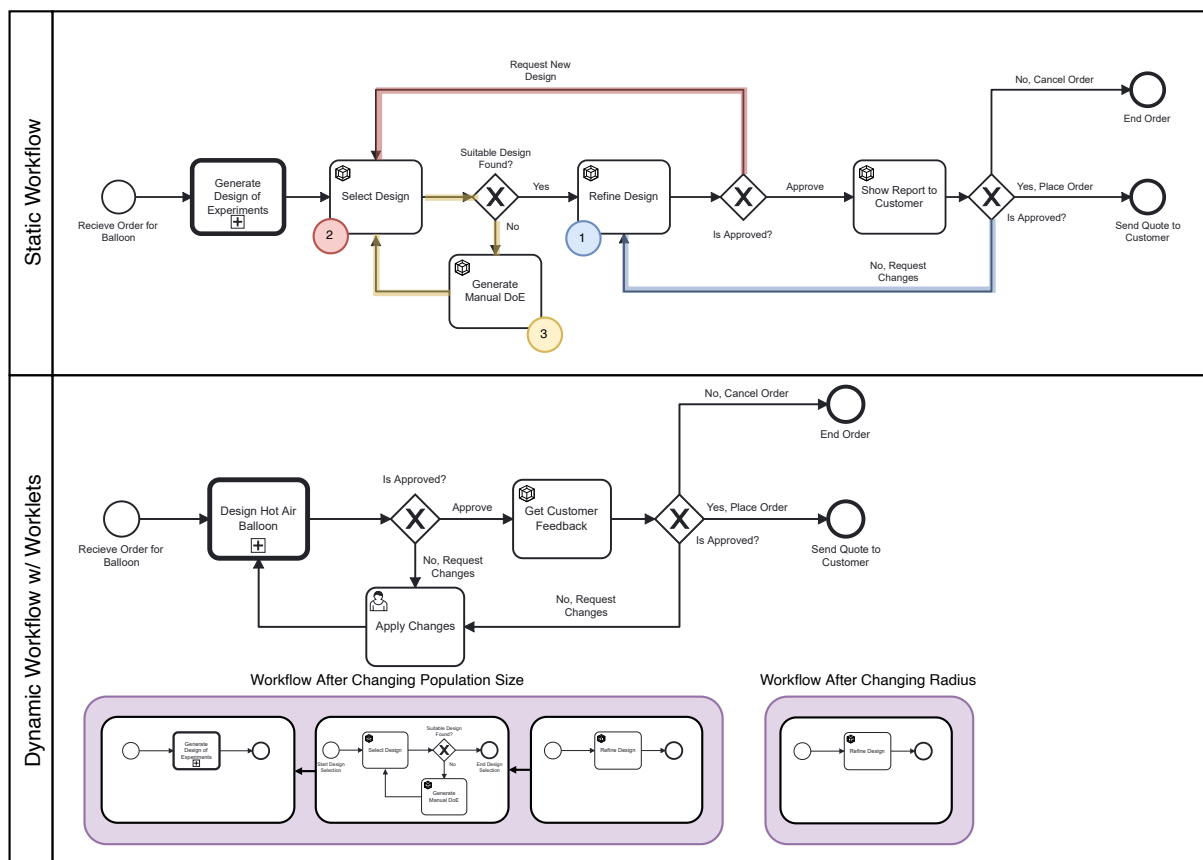


Figure 5.1: Differences in Iterative Usage on Static vs. Dynamic Workflows

In light of this result, use of KBE control is advised when the design process must account for ad-hoc changes and is highly iterative. Furthermore, although this present research has focused on collaboration, the same underlying technology can be applied to lazily launch simulations within an automated design process.

Besides the ability to collaborate and integrate the KBE application within business processes, through the hybrid workflow concept, KBE applications also gain new functionalities from WfMS that were not available before. Most importantly is the ability to view exactly what has happened, what is currently running, and what could run in the future. This task awareness is provided by the *Operate* frontend application of Camunda given by Figure 5.2.

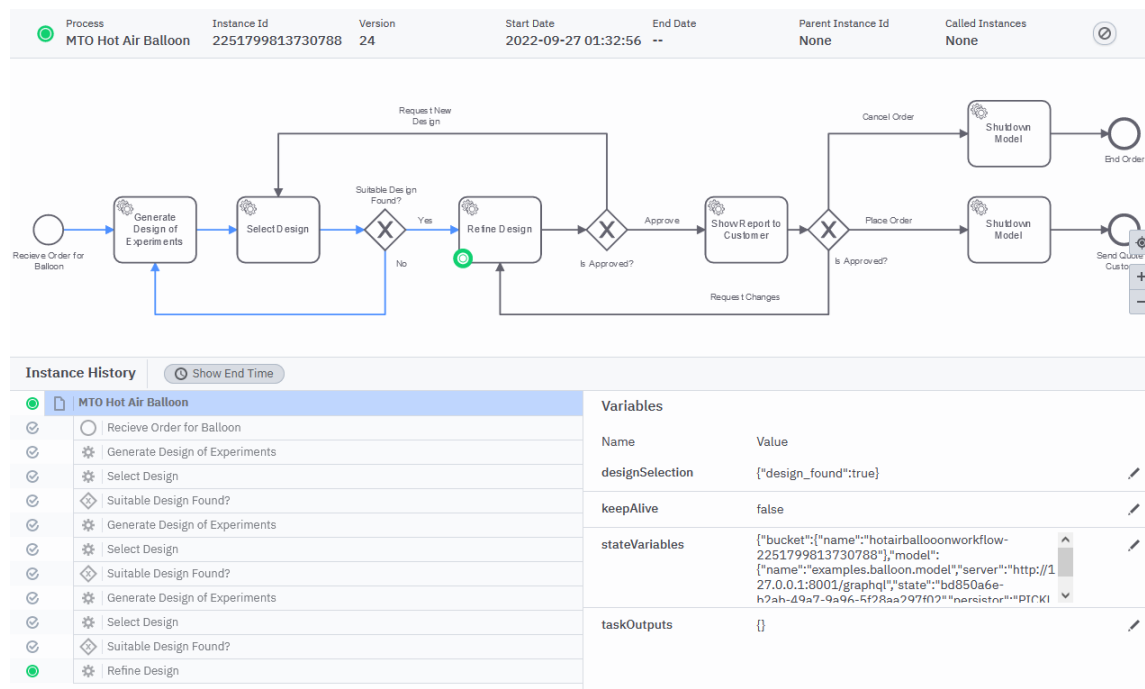


Figure 5.2: Hot Air Balloon Workflow Status, Audit Log, & Process Variables

During the adaption of the original KBE applications, a notable challenge was in synchronizing variable and task names between their implementation in the KBE application and the process model; causing the highest number of failures during development. In the future, to improve user experience the KBE application source code should be parsed to provide auto-completion functionality in the process model during variable name assignment. As a final remark, in complicated use cases the implicit order of tasks within the KBE application could become an issue when the same developer is not in charge of both creating the product and process model. Such issues can be fixed either through more significant adoption of the worklet approach—thereby allowing the KBE model to define overall workflow control flow—or to statically define required inputs in tasks to allow a process linter to detect invalid task ordering. However, the latter is not preferable as it would increase tool interface complexity in the process model.

## 5.4 Answers to Research Questions

Based on the developed software prototype and its use within use cases, the research questions posed by this thesis can be answered as follows:

**LQ-1:** What technology is suitable to expose the information of a KBE application to external services while supporting dynamic types, lazy evaluation, and dependency tracking?

*As demonstrated by Section 3.2 and Section 4.1, the use of GraphQL in this research has shown promise for exposing the information of a KBE application. The key features of GraphQL that make it a suitable technology are its hierarchical structure that mirrors the product tree of a KBE application, selection sets which support both lazy evaluation and dependency tracking, and type unions that allow for the flexibility of modeling dynamic types. Besides these features, the strongly typed nature of GraphQL makes it easier to query for information within a KBE application by being able to know ahead of time what information is available and the type of data that will be returned.*

**LQ-2:** How can the runtime cache of a KBE application be persisted to support the transactional nature of tasks where each task leads to a new state of the model?

*As demonstrated by Section 3.3 and Section 4.2, the use of Python's builtin binary persistence module `pickle` has demonstrated applicability for persisting the runtime cache of a KBE application. However, due to the current limitation of Python with respect to recursion, the recursive implementation of `pickle` required handling of the object and dependency graph separately. Nonetheless, transaction usage in processes has been achieved by allowing the full state of a ParaPy application to be persisted. Therefore, if failures occur a previous state can be used to load a version of the application state before the failure occurred.*

**LQ-3:** What is the time and memory complexity associated with persisting the runtime caches of a KBE application?

*As demonstrated by Section 4.2 the time and memory complexity of persisting the runtime caches of a KBE application is  $O(n)$ . It was observed that the graph contraction algorithm can be used along with partial persistence functionality to reduce the memory required to persist these caches. In extreme cases, where no geometry is needed and intermediate results can be ignored, an  $O(1)$  memory complexity can be achieved. However, real-world applicability for such use cases is expected to be limited.*

**LQ-4:** Can runtime dependency information be used to establish external steering of a WfMS to modify the control-flow dynamically at runtime based on data dependencies?

*Yes, as demonstrated by Section 3.4.6 and its implementation in Section 4.3.2, external steering can be established using the worklet concept. This enables the runtime caching and dependency tracking mechanisms to be used to dynamically publish worklets using demand-driven evaluation. This allows for runtime control flow to change based on previous execution results.*

**LQ-5:** If external steering is possible, does it show promise for increasing workflow flexibility?

*Yes, as demonstrated by Section 4.3.2, the application of the worklet concept allowed for more flexible support of an ad-hoc design process where several iterations were performed. Without the worklet approach the additional control flow added on top of the KBE application, made the process rigid as a strict order of tasks needed to be followed in order to perform design iterations. By adoption of the worklet concept, such restrictions on control flow are reduced as the process does not have to execute from start to finish, and instead smaller portions of the process can be run based on context. However, the flexibility gained by this concept comes at the cost of reduced process awareness due to the current lack of visualization techniques.*

## 5.5 Research Contributions

This thesis project has from the beginning has strived to establish the next generation of collaborative KBE. Through a methodology that has focused on process formalization and orchestration, several key contributions have been made in order to achieve this goal. First, an information modeling approach making use of GraphQL has been developed to be able to flexibly query KBE models to support collaborative activities. Furthermore, generic persistence capability has been added to ParaPy to support transactional usage as well as retaining the full history of a design. Moreover, the implementation of the worklet concept in KBE has opened doors to a new form of process automation making use of runtime caching and dependency tracking to influence control flow based on context. This has shown promise to increase the flexibility of workflows. Finally, this research has theorized about emergent workflows whereby the correlated dependency technique may be applied in the future to enable true flexibility, whereby no control-flow is dictated ahead of time. Figure 5.3 provides an overview of the research contributions.

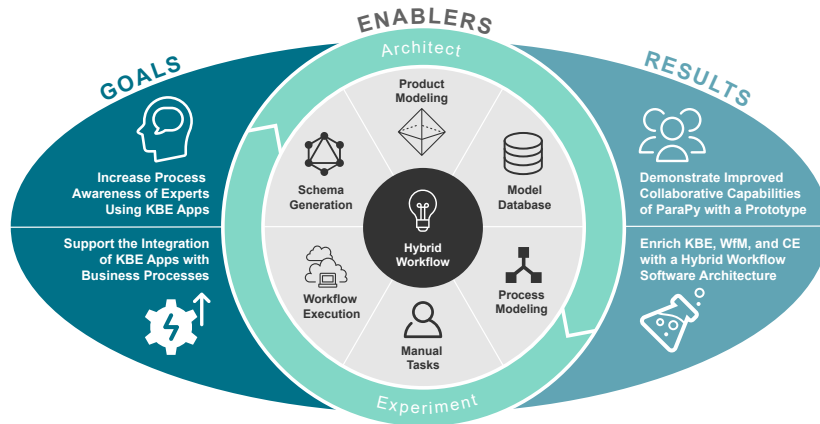


Figure 5.3: Goals, Enablers, Results of the Conducted Research

## 5.6 Limitations & Future Work

Although able to represent a fair part of KBE applications, the GraphQL transpiler does have its limitations. While some aspects are not yet modeled due to limitations with the GraphQL specification, others are not implemented due to time intensity. In the future, more slot resolvers can be implemented to be able to represent a larger portion of KBE applications. Additionally, a considerable portion of application launch time is schema generation. Therefore, persistence capability for the in-memory schema could be added to support faster launch times.

One limitation of the current persistence capability is lack of support for meshses. Using the plugin architecture defined, this can be solved by creating a plugin. However, what is more problematic is the lack of so-called “delta-encoding” when making use of the persistence capability in processes. This essentially means that every time the application is persisted, a full copy of the data is created. What delta encoding enables is the ability to only store the changed data between states. This can be especially useful for large applications saved multiple times in a process.

In terms of process orchestration, the worklet concept has revealed that although external steering using a KBE application has been achieved, currently the worklet “call stack” is not representable within the available BPMN features in Camunda. Without such visualization techniques it is difficult to provide users with an overview of what is expected to happen and what has executed. Future work should explore custom visualization techniques. Furthermore, unit tests should be added to the process orchestration layer to provide confidence for developers when building new features.

## 6 Conclusion

This thesis achieves the synthesis between KBE and WfM, pushing the state-of-the-art into next-generation solutions. Based on the research goal, a strenuous process has led to improving the collaborative usage of KBE applications through novel means and hybrid workflows, thereby increasing process awareness through process modeling and visualization, and facilitating the integration of KBE with business processes. The new software prototype that is developed addresses an important gap in the literature for a tool that can automate the design process, its knowledge and data through the development of novel techniques for the KBE platform of ParaPy.

The developed software prototype addresses three key high-level requirements as verified in this thesis: (a) providing access of KBE application information to the process orchestrator and all connected resources, (b) allowing dependency relationships to be serializable to support fault tolerance and a hybrid workflow concept, and (c) dynamically chaining together tasks through ad-hoc control flow during running processes. To enable these high-level requirements, new advances are realized, which include a KBE-GraphQL transpiler that parses application source code and automatically generates resolvers and types to reduce the burden on developers. In addition, advancements to the serialization of KBE applications is made through the introduction of a graph contraction and consistency maintenance algorithm. Also, a way to maintain dependencies when coupling external tools to KBE applications is introduced through the correlated dependency concept. Finally, a way to link worklets to slots in KBE applications has enabled the usage of the runtime caching and dependency tracking mechanism to dynamically alter the control flow at runtime.

Based on the results of the experiments for the software prototype for information modeling, model persistence and process orchestration, the main findings for these topics are summarized below:

- **Information Modeling:** Based on 8 example applications with different geometric complexities, it is found that if both type-inferencing and dynamic typing are enabled then almost all slots in these example applications can be represented.
- **Model Persistence:** The tested models are all found to be serializable regardless of geometric complexity. The time and memory complexity of serialization is found to be  $O(n)$ .
- **Process Orchestration:** A real-world application based on an earthquake analysis case verifies that the collaborative usage of KBE applications can be improved while facilitating the integration of KBE applications with business processes. In addition, the case study of a hot air balloon demonstrates how the worklet concept enables greater flexibility for ad-hoc usage in design processes.

Overall, the advances that are put forward in this thesis enable and empower a knowledge engineer to formalize the intended process of a KBE application to improve its collaborative usage. More specifically, these advances allow integrating process awareness to improve the collaborative usage of KBE applications, meaning that there is sufficient awareness on what has happened, what is currently running, and what could run in the future in the process.

# References

- [1] Knut Erik Bang and Tore Markeset. "Impact of Globalization on Model of Competition and Companies' Competitive Situation". In: *Advances in Production Management Systems. Value Networks: Innovation, Technologies, and Management*. Ed. by Jan Frick and Børge Timenes Laugen. Vol. 384. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 276–286. ISBN: 978-3-642-33979-0 978-3-642-33980-6. DOI: 10.1007/978-3-642-33980-6\_32.
- [2] Josip Stjepandi, Nel Wognum, and Wim J.C. Verhagen, eds. *Concurrent Engineering in the 21st Century*. Cham: Springer International Publishing, 2015. ISBN: 978-3-319-13775-9 978-3-319-13776-6. DOI: 10.1007/978-3-319-13776-6.
- [3] Dirk Ahlers et al. "Challenges for Information Access in Multi-Disciplinary Product Design and Engineering Settings". In: *2015 Tenth International Conference on Digital Information Management (ICDIM)*. Jeju Island, South Korea: IEEE, Oct. 2015, pp. 109–114. ISBN: 978-1-4673-9152-8. DOI: 10.1109/ICDIM.2015.7381865.
- [4] Dimitri N. Mavris and Olivia J. Pinon. "An Overview of Design Challenges and Methods in Aerospace Engineering". In: *Complex Systems Design & Management*. Ed. by Omar Hammami, Daniel Krob, and Jean-Luc Voirin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–25. ISBN: 978-3-642-25203-7.
- [5] F. Mas et al. "A Review of PLM Impact on US and EU Aerospace Industry". In: *Procedia Engineering* 132 (2015), pp. 1053–1060. ISSN: 18777058. DOI: 10.1016/j.proeng.2015.12.595.
- [6] Massimo D'Auria and Roberto D'Ippolito. "Process Integration and Design Optimization Ontologies for Next Generation Engineering". In: *On the Move to Meaningful Internet Systems: OTM 2013 Workshops*. Ed. by David Hutchison et al. Vol. 8186. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 228–237. ISBN: 978-3-642-41032-1 978-3-642-41033-8. DOI: 10.1007/978-3-642-41033-8\_31.
- [7] Wim J.C. Verhagen et al. "A Critical Review of Knowledge-Based Engineering: An Identification of Research Challenges". In: *Advanced Engineering Informatics* 26.1 (Jan. 2012), pp. 5–15. ISSN: 14740346. DOI: 10.1016/j.aei.2011.06.004.
- [8] G. La Rocca and M.J.L. Van Tooren. "Enabling Distributed Multi-Disciplinary Design of Complex Products: A Knowledge Based Engineering Approach". In: *J. of Design Research* 5.3 (2007), p. 333. ISSN: 1748-3050, 1569-1551. DOI: 10.1504/JDR.2007.014880.
- [9] Dirk Jodin and Christian Landschützer. "Knowledge-Based Methods for Efficient Material Handling Equipment Development". In: *12th IMHRC Proceedings*. Gardanne, France, 2012, p. 20.
- [10] Layna Fischer and Workflow Management Coalition. *Workflow Handbook 2005*. Lighthouse Point, FL: Future Strategies Inc., 2006. ISBN: 978-0-9703509-8-5.
- [11] A I Pavlov and A B Stolbov. "The Workflow Component of the Knowledge-Based Systems Development Platform". In: *Proceedings for the 2nd Scientific-practical Workshop Information Technologies: Algorithms, Models, Systems*. Vol. 2463. Irkutsk, Russia: CEUR, Sept. 2019, p. 11.
- [12] Matthew Daniels et al. "An Engineering Prototype Workflow Management System". In: *IFAC Proceedings Volumes* 46.9 (2013), pp. 1471–1476. ISSN: 14746670. DOI: 10.3182/20130619-3-RU-3018.00428.
- [13] Claudia Eckert, Anja Maier, and Chris McMahon. "Communication in Design". In: *Design Process Improvement*. Ed. by John Clarkson and Claudia Eckert. London: Springer London, 2005, pp. 232–261. ISBN: 978-1-85233-701-8 978-1-84628-061-0. DOI: 10.1007/978-1-84628-061-0\_10.



- 
- [14] Weiming Shen, Jean-Paul Barthès, and Junzhou Luo. “Computer Supported Collaborative Design: Technologies, Systems, and Applications”. In: *Contemporary Issues in Systems Science and Engineering*. Ed. by Mengchu Zhou, Han-Xiong Li, and Margot Weijnen. Hoboken, NJ, USA: John Wiley & Sons, Inc., Apr. 2015, pp. 537–573. ISBN: 978-1-119-03682-1 978-1-118-27186-5. DOI: 10.1002/9781119036821.ch14.
  - [15] Carlos Vila et al. “Workflow Methodology for Collaborative Design and Manufacturing”. In: *Cooperative Design, Visualization, and Engineering*. Ed. by Yuhua Luo. Vol. 4674. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 42–49. ISBN: 978-3-540-74779-6 978-3-540-74780-2. DOI: 10.1007/978-3-540-74780-2\_5.
  - [16] Wil van der Aalst and Kees Max van Hee. *Workflow Management: Models, Methods, and Systems*. Cooperative Information Systems. Cambridge, Mass: MIT Press, 2002. ISBN: 978-0-262-01189-1.
  - [17] Jennifer Rowley. “The Wisdom Hierarchy: Representations of the DIKW Hierarchy”. In: *Journal of Information Science* 33.2 (Apr. 2007), pp. 163–180. ISSN: 0165-5515, 1741-6485. DOI: 10.1177/0165551506070706.
  - [18] A. Pras and J. Schoenwaelder. *On the Difference between Information Models and Data Models*. Tech. rep. RFC3444. RFC Editor, Jan. 2003, RFC3444. DOI: 10.17487/rfc3444.
  - [19] A.W. Reijnders. “Integrating Knowledge Management and Knowledge-Based Engineering”. MA thesis. Delft, The Netherlands: Delft University of Technology, Oct. 2012.
  - [20] Martyn Pinfold and Craig Chapman. “The Application of KBE Techniques to the FE Model Creation of an Automotive Body Structure”. In: *Computers in Industry* 44.1 (Jan. 2001), pp. 1–10. ISSN: 01663615. DOI: 10.1016/S0166-3615(00)00079-8.
  - [21] “Knowledge Based Engineering”. In: *Multidisciplinary Design Optimization Supported by Knowledge Based Engineering*. John Wiley & Sons, Ltd, 2015, pp. 208–257. ISBN: 978-1-118-89707-2.
  - [22] P. C. Gembarski. “Three Ways of Integrating Computer-Aided Design and Knowledge-Based Engineering”. In: *Proceedings of the Design Society: DESIGN Conference 1* (May 2020), pp. 1255–1264. ISSN: 2633-7762. DOI: 10.1017/dsd.2020.313.
  - [23] Dave Cooper and Gianfranco LaRocca. “Knowledge-Based Techniques for Developing Engineering Applications in the 21st Century”. In: *7th AIAA ATIO Conf, 2nd CEIAT Int'l Conf on Innov and Integr in Aero Sciences, 17th LTA Systems Tech Conf; Followed by 2nd TEOS Forum*. Belfast, Northern Ireland: American Institute of Aeronautics and Astronautics, Sept. 2007. ISBN: 978-1-62410-014-7. DOI: 10.2514/6.2007-7711.
  - [24] C.B Chapman and M Pinfold. “Design Engineeringa Need to Rethink the Solution Using Knowledge Based Engineering”. In: *Knowledge-Based Systems* 12.5-6 (Oct. 1999), pp. 257–267. ISSN: 09507051. DOI: 10.1016/S0950-7051(99)00013-1.
  - [25] Akshay Raju Kulkarni, Maurice Hoogreef, and Gianfranco La Rocca. “Combining Semantic Web Technologies and KBE to Solve Industrial MDO Problems”. In: *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*. Denver, Colorado: American Institute of Aeronautics and Astronautics, June 2017. ISBN: 978-1-62410-507-4. DOI: 10.2514/6.2017-3823.
  - [26] Gianfranco La Rocca. “Knowledge Based Engineering: Between AI and CAD. Review of a Language Based Technology to Support Engineering Design”. In: *Advanced Engineering Informatics* 26.2 (Apr. 2012), pp. 159–179. ISSN: 14740346. DOI: 10.1016/j.aei.2012.02.002.
  - [27] Jayakiran Esanakula, Naga venkata Sridhar, and Vootukuri Rangadu. “Knowledge Based Engineering: Notion, Approaches and Future Trends”. In: *American Journal of Intelligent Systems* 2015 (Jan. 2015), pp. 1–17. DOI: 10.5923/j.ajis.20150501.01.

- 
- [28] P. Bermell Garcia and I S. Fan. “Practitioner Requirements for Integrated Knowledge-Based Engineering in Product Lifecycle Management”. In: *International Journal of Product Lifecycle Management* 3.1 (2008), p. 3. ISSN: 1743-5110, 1743-5129. DOI: 10.1504/IJPLM.2008.019968.
  - [29] Eugen Rigger, Kristina Shea, and Tino Stankovic. “Task Categorisation for Identification of Design Automation Opportunities”. In: *Journal of Engineering Design* 29.3 (Mar. 2018), pp. 131–159. ISSN: 0954-4828, 1466-1837. DOI: 10.1080/09544828.2018.1448927.
  - [30] Jayakiran Reddy Esanakula et al. “Online Knowledge-Based System for CAD Modeling and Manufacturing: An Approach”. In: *Intelligent Systems, Technologies and Applications*. Ed. by Sabu M. Thampi et al. Vol. 910. Singapore: Springer Singapore, 2020, pp. 259–268. ISBN: 9789811360947 9789811360954. DOI: 10.1007/978-981-13-6095-4\_19.
  - [31] Kelbey Wheeler. “Methodological Support for Knowledge Based Engineering Application Development: Improving Traceability of Knowledge into Application Code”. MA thesis. Delft, The Netherlands: Delft University of Technology, Mar. 2020.
  - [32] Imco van Gent et al. “Knowledge Architecture Supporting the next Generation of MDO in the AGILE Paradigm”. In: *Progress in Aerospace Sciences* 119 (Nov. 2020), p. 100642. ISSN: 03760421. DOI: 10.1016/j.paerosci.2020.100642.
  - [33] Nick Tzannetakis and Roberto d’Ippolito. *iProd Fact Sheet*. Feb. 2011.
  - [34] Maurice F. M. Hoogreef et al. “An Application of the IProd Software Framework to Support the Product Development Process in Te Automotive and Aerospace Domain”. In: *Tools and Methods of Competitive Engineering: Proceedings of the Tenth International Symposium on Tools and Methods of Competitive Engineering - TMCE 2014, May 19 - 23, Budapest, Hungary*. Delft: Faculty of Industrial Design Engineering, Delft University of Technology, 2014, p. 14. ISBN: 978-94-6186-177-1 978-94-6186-176-4.
  - [35] Nick Tzannetakis and Roberto d’Ippolito. *iProd Publishable Summary 2014*. Apr. 2014.
  - [36] P.K.M Chan. “A New Methodology for the Development of Simulation Workflows”. MA thesis. Delft, The Netherlands: Delft University of Technology, Mar. 2013.
  - [37] Maurice F M Hoogreef. “A Multidisciplinary Design Optimization Advisory System for Aircraft Design”. In: *5th CEAS Air & Space Conference*. CEAS, 2015, p. 15.
  - [38] Zhenjun Ming et al. “PDSIDESA Knowledge-Based Platform for Decision Support in the Design of Engineering Systems”. In: *Journal of Computing and Information Science in Engineering* 18.4 (Dec. 2018), p. 041001. ISSN: 1530-9827, 1944-7078. DOI: 10.1115/1.4040461.
  - [39] Dag Bergsjö, Amer Catic, and Johan Malmqvist. “Implementing a Service-Oriented PLM Architecture Focusing on Support for Engineering Change Management”. In: *International Journal of Product Lifecycle Management* 3.4 (2008), p. 335. ISSN: 1743-5110, 1743-5129. DOI: 10.1504/IJPLM.2008.027010.
  - [40] Dante Pugliese, Giorgio Colombo, and Maurizio Saturno Spurio. “About the Integration between KBE and PLM”. In: *Advances in Life Cycle Engineering for Sustainable Manufacturing Businesses*. Ed. by Shozo Takata and Yasushi Umeda. London: Springer London, 2007, pp. 131–136. ISBN: 978-1-84628-934-7. DOI: 10.1007/978-1-84628-935-4\_23.
  - [41] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas behind Reliable, Scalable, and Maintainable Systems*. First edition. Boston: O’Reilly Media, 2017. ISBN: 978-1-4493-7332-0.
  - [42] Tristan Pollock. *Automated Workflows Are Eating the World*. <https://venturebeat.com/2020/11/21/workflows-are-eating-the-world/> Technology News. Nov. 2020.



- 
- [43] Nick Russell, Wil van der Aalst, and Arthur Ter Hofstede. *Workflow Patterns: The Definitive Guide*. Cambridge, MA: MIT Press, 2015. ISBN: 978-0-262-02982-7.
  - [44] Reijers HA (Hajo). “Design and Control of Workflow Processes : Business Process Management for the Service Industry”. In: (2002). DOI: 10.6100/IR557134.
  - [45] Theo Haerder and Andreas Reuter. “Principles of Transaction-Oriented Database Recovery”. In: *ACM Computing Surveys* 15.4 (Dec. 1983), pp. 287–317. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/289.291.
  - [46] Michael Rosen, ed. *Applied SOA: Service-Oriented Architecture and Design Strategies*. Indianapolis, IN: Wiley Pub, 2008. ISBN: 978-0-470-22365-9.
  - [47] Microsoft. *Workflow Persistence*. Mar. 2017.
  - [48] Patrick R. Schaumont. *A Practical Introduction to Hardware/Software Codesign*. Boston, MA: Springer US, 2010. ISBN: 978-1-4419-5999-7 978-1-4419-6000-9. DOI: 10.1007/978-1-4419-6000-9.
  - [49] Athanassios M. Kintsakis, Fotis E. Psomopoulos, and Pericles A. Mitkas. “Reinforcement Learning Based Scheduling in a Workflow Management System”. In: *Engineering Applications of Artificial Intelligence* 81 (May 2019), pp. 94–106. ISSN: 09521976. DOI: 10.1016/j.engappai.2019.02.013.
  - [50] Rafael Ferreira da Silva et al. “A Characterization of Workflow Management Systems for Extreme-Scale Applications”. In: *Future Generation Computer Systems* 75 (Oct. 2017), pp. 228–238. ISSN: 0167739X. DOI: 10.1016/j.future.2017.02.026.
  - [51] Stefan Jablonski and Christoph Bussler. *Workflow Management: Modeling Concepts, Architecture and Implementation*. London: ITP, Internat. Thomson Computer Press, 1996. ISBN: 978-1-85032-222-1.
  - [52] David Hollingsworth. *Workflow Management Coalition The Workflow Reference Model*. Tech. rep. TC-1003. Avenue Marcel Thiry 204, 1200 Brussels, Belgium: Worklow Management Coalition, June 1996.
  - [53] Michael Thelin. *Luigi Is Now Open Source: Build Complex Pipelines of Tasks*. Developer Blog. Sept. 2012.
  - [54] Brian Knight, ed. *Professional SQL Server 2005 Integration Services*. Wrox Professional Guides. Indianapolis, IN: Wrox/Wiley Pub, 2006. ISBN: 978-0-7645-8435-0.
  - [55] International Organization for Standardization. *Object Management Group Business Process Model and Notation*. ISO/IEC/IEEE Standard 19510:2013. ISO/IEC JTC1 Information technology, July 2013, p. 507.
  - [56] *Business Process Model and Notation (BPMN), Version 2.0*.
  - [57] Marcelo Bernardino Araújo and Rodrigo Franco Gonçalves. “Selecting a Notation to Modeling Business Process: A Systematic Literature Review of Technics and Tools”. In: *Advances in Production Management Systems. Initiatives for a Sustainable World*. Ed. by Irenilza Nääs et al. Cham: Springer International Publishing, 2016, pp. 198–205. ISBN: 978-3-319-51133-7.
  - [58] Jakob Freund and Bernd Rücker. *Real-Life BPMN: Using BPMN and DMN to Analyze, Improve, and Automate Processes in Your Company*. Trans. by James Venis, Kristen Hannum, and Jalynn Venis. 4th edition. Berlin: Camunda, 2019. ISBN: 978-1-08-630209-7.
  - [59] Wei Han. “Modelling for Data Management & Exchange in Concurrent Engineering - A Case Study of Civil Aircraft Assembly Line Development Process”. MA thesis. Cranfield University, Nov. 2013.

- [60] K. Rouibah, S. Rouibah, and W. M. P. Van Der Aalst. "Combining Workflow and PDM Based on the Workflow Management Coalition and STEP Standards: The Case of Axalant". In: *International Journal of Computer Integrated Manufacturing* 20.8 (Dec. 2007), pp. 811–827. ISSN: 0951-192X, 1362-3052. DOI: 10.1080/09511920600930038.
- [61] Pouria G. Bigvand and Alexander Fay. "Optimal Path-Finding in a Context-Aware Workflow Support System for Process and Automation Engineering of Plants". In: *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Limassol: IEEE, Sept. 2017, pp. 1–8. ISBN: 978-1-5090-6505-9. DOI: 10.1109/ETFA.2017.8247672.
- [62] Pouria G. Bigvand and Alexander Fay. "A Workflow Support System for the Process and Automation Engineering of Production Plants". In: *2017 IEEE International Conference on Industrial Technology (ICIT)*. 2017, pp. 1118–1123. DOI: 10.1109/ICIT.2017.7915519.
- [63] M.M. Kwan and P.R. Balasubramanian. "Dynamic Workflow Management: A Framework for Modeling Workflows". In: *Proceedings of the Thirtieth Hawaii International Conference on System Sciences*. Vol. 4. Wailea, HI, USA: IEEE Comput. Soc. Press, 1997, pp. 367–376. ISBN: 978-0-8186-7743-4. DOI: 10.1109/HICSS.1997.663409.
- [64] Michael Adams. "Dynamic Workflow". In: *Modern Business Process Automation*. Ed. by Arthur H. M. Hofstede et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 123–145. ISBN: 978-3-642-03122-9 978-3-642-03121-2. DOI: 10.1007/978-3-642-03121-2\_4.
- [65] Camunda. *Zeebe*. <https://docs.camunda.io/docs/1.0/product-manuals/zeebe/zeebe-overview>. Software Documentation. 2021.
- [66] Cloud Elements. *The State of API Integration 2021*. Apr. 2021.
- [67] *GraphQL Specification*. Oct. 2021.
- [68] Benjamin C. Pierce and David N. Turner. "Local Type Inference". In: *ACM Transactions on Programming Languages and Systems* 22.1 (Jan. 2000), pp. 1–44. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/345099.345100.
- [69] Open Cascade Technology. *Modeling Data*. [https://dev.opencascade.org/doc/overview/html/occt\\_user\\_guides\\_\\_modeling\\_data.html](https://dev.opencascade.org/doc/overview/html/occt_user_guides__modeling_data.html). Software Documentation. Oct. 2022.
- [70] Gianfranco La Rocca, T.H.M. Langen, and Y.H.A. Brouwers. *The Design and Engineering Engine. Towards a Modular System for Collaborative Aircraft Design*. Vol. 5. Jan. 2012.
- [71] ParaPy. *ParaPy Documentation*. 2020.
- [72] Dave Cooper. *Genworkd GDL: A User's Manual*. 2021.
- [73] D. Cooper and R.E.C. Van Dijk. "Gendl Meets X3DOM: The Declarative Web, All the Way Down". In: *Web3D '12: Proceedings of the 17th International Conference on Web3D Technology, Los Angeles (CA), USA, 04-05.08.2012*. New York (NY): A.C.M., 2012, p. 187. ISBN: 978-1-4503-1432-9.
- [74] Siemens Digital Industries Software. *Rulestream for Engineer-to-Order Process Automation: Enabling Manufacturers to Rapidly Engineer Products to Unique Customer Specifications*. 2017.
- [75] TechnoSoft Inc. *The Adaptive Modeling Language. A Technical Perspective*. 2003.
- [76] TechnoSoft Inc. *AMEnterprise Product Sheet*. 2004.
- [77] Jeffrey Zweber et al. "Towards an Integrated Design Environment for Hypersonic Vehicle Design and Synthesis". In: *AIAA/AAAF 11th International Space Planes and Hypersonic Systems and Technologies Conference*. Orleans, France: American Institute of Aeronautics and Astronautics, Sept. 2002. ISBN: 978-1-62410-123-6. DOI: 10.2514/6.2002-5172.

- 
- [78] Universal Robots. *JT File Format for UR Robots*. <https://www.universal-robots.com/articles/ur/application-installation/jt-file-format-for-ur-robots/>. Support Page. June 2021.
- [79] A. Molina, J. Aca, and P. Wright. “Global Collaborative Engineering Environment for Integrated Product Development”. In: *International Journal of Computer Integrated Manufacturing* 18.8 (Dec. 2005), pp. 635–651. ISSN: 0951-192X, 1362-3052. DOI: 10.1080/09511920500324472.
- [80] Uber. *Cadence Use Cases*. <https://cadenceworkflow.io/docs/use-cases/>. Software Documentation. 2021.
- [81] Camunda. *Camunda Compared to Alternatives: Guide to The Process Automation Landscape*. July 2021.
- [82] Camunda. *The Camunda Platform Manual*. <https://docs.camunda.org/manual/7.15/>. Software Documentation. 2021.
- [83] NSA Cybersecurity Directorate. *Welcome to WALKOFF's Documentation*. <https://walkoff.readthedocs.io/en/latest/>. Software Documentation. 2019.
- [84] Sartography. *What Is SpiffWorkflow?* <https://spiffworkflow.readthedocs.io/en/latest/>. Software Documentation. 2021.
- [85] Prefect. *Prefect Core*. <https://docs.prefect.io/core/>. Software Documentation. 2021.
- [86] Uber. *Cadence Overview*. <https://cadenceworkflow.io/docs>. Software Documentation. 2021.
- [87] Alfrick Opidi. *GraphQL vs. REST: A Comprehensive Comparison*. Technology Blog. Sept. 2020.
- [88] SmartBear. *The State of API 2020 Report*. Aug. 2020.
- [89] Ajay. *Why GraphQL*. Technology Blog. Nov. 2017.
- [90] Imco van Gent et al. “A Critical Look at Design Automation Solutions for Collaborative MDO in the AGILE Paradigm”. In: *2018 Multidisciplinary Analysis and Optimization Conference*. Atlanta, Georgia: American Institute of Aeronautics and Astronautics, June 2018. ISBN: 978-1-62410-550-0. DOI: 10.2514/6.2018-3251.

# A Market Studies

## A.1 Knowledge Based Engineering

A market study was performed to understand if modern KBE systems currently provide workflow functionalities. For this purpose, commercial KBE systems that have a coupled integration to a CAD kernel were analyzed. Functional compliance during the study was grouped into the categories of: Process Orchestration (PO), Human-System (HS) coupling, and Model Database. The latter describes if the KBE system can persist and share product state between multiple applications. The PO metric, measures if a KBE system provides explicit process modelling to enable an overview of tasks within an application. An implicit process orchestration capability signifies the default behavior of a KBE system to automate a design process through an emergent flow that is abstracted from the user. Meanwhile, HS Coupling determines the primary mechanism for providing users with information and asking them for decisions. Finally, the Model Database capability determines if a KBE system can persist model information and transfer it to multiple clients without running. The KBE systems included in this market study as well as how they fulfill these needs are provided by Table A.1.

Table A.1: Market Study of Commercial KBE Systems with Coupled Integration

KBE System	CAD Kernel	Persistence	PO	HS Coupling	Model Database
ParaPy [71]	Open Cascade	JSON Snapshot	Implicit	Forms	No
GDL [72, 73]	SMLib	Binary Model	Implicit	Forms	No
Rulestream [74]	Siemens NX	JT Model	Explicit	Forms	Yes, through Siemens TeamCenter
AML [75, 76, 77]	Parasolid	XML Model	Implicit	Forms	Yes, through AMEnterprise

At present, the ParaPy SDK and the Genworks GDL KBE systems have no capability to persist and transfer the information of the product model without the KBE system running. Although ParaPy provides support for transferring data through STEP and IGES, the information model containing evaluated attributes is not transferred. Furthermore, the persistence mechanism of ParaPy is a light-weight JSON snapshot that records user-inputs to be able to reconstruct the model at launch. This means that evaluated attributes and geometry is not available in the snapshot. In the case of GDL, its use of a binary persistence format means that model information can only be accessed when GDL is running. However, unlike ParaPy, the saved “world” can be launched instantly without requiring the KBE system to re-evaluate [72, p.20].

On the contrary, Rulestream and AML have a model database functionality. Rulestream enables this functionality by persisting model geometry and metadata to JT files [74, p.4] which can be used for product visualization, collaboration, and data sharing with Siemens PLM software [78]. AML enables collaboration on a distributed KBE model through an Object Request Broker (ORB) which allows multiple users to be connected across different time zones, and edit models collaboratively [77, p.2]. Although AML through AMEnterprise has functionalities to enable geographically distributed teams to collaborate, [79] found in their study that AML is not suitable to use in a web-browser, and must be installed locally on the remote machine. As a result, this hampers usability of AML in a virtual enterprise, where not every company will desire to install the client on their machines. Nonetheless, in their white-paper, AML advertise “a single underlying object-oriented database”, thus the possibility exists to create a web-based client using this underlying technology [75, p.5]. With the exception of AML, all other KBE systems in this survey provide capabilities to deploy the application on the web.

Most KBE systems analyzed have implicit process orchestration and all use forms as their primary mechanism for human-system coupling; allowing a user to be roughly guided through a process by

sequence of forms. Only RuleStream has a type of explicit process modelling by separating product and process trees, allowing a design procedure to be represented separately to the product model [40, p.132]. Therefore, this market study aligns with the observation of that KBE applications have weak workflow integration [2, p.262].

## A.2 Workflow Management

As the synthesis of KBE and WfM is a new field of research, the features required are not available from a single product. However, creating a bespoke workflow engine would be time-consuming and error-prone. Therefore, extending open-source workflow engines can utilize the tremendous research and development that has gone into them. These were evaluated based on the presence of features listed below and a summary of their capabilities is presented by Table A.2. Polling capability enables execution of long-running tasks and “fault-tolerance” signifies if a WfMS has built-in support for workflow persistence and compensation actions.

The Camunda Platform is a BPMS written in Java, that has a strong open-source community and maintains the BPMN-js library for modelling and visualization purposes. It is the only surveyed system that provides all the desired features. Its counterpart, Zeebe, is poised to enable greater horizontal scaling capability And enhanced usability through auto-generated clients utilizing gRPC. This is similar to Uber’s Cadence workflow manager that uses Apache Thrift instead of gRPC for client generation. A unique aspect of Cadence is its ability to persist full-state of multithreaded applications. Interestingly, Cadence does not have BPMN modelling capability and argues that low-code platforms are limited when processes grow beyond simple “hello-world” use-cases [80]. Camunda confirms this by stating that low-code platforms lack the capability for end-to-end orchestration of complex business processes and do not scale well [81, p.7]. However, they assert that the combination of BPMN with the presence of a flexible and modular interface for developers prevents the pit-falls of previous low-code products [81].

The remainder of the surveyed workflow managers are Python-based which is advantageous to easily interface with the ParaPy SDK. Although Prefect has a feature-set that geared toward data pipelines and ETL tasks instead of WfMS, it is the most complete and easy-to-use Python package available and has a GraphQL API. The advantages of the latter will be covered in Section A.3. It also supports long-running tasks through a configurable “heartbeat” and has optional flow-state caching. Comparatively, the NSA’s WALKOFF prioritizes security and has no persistence capabilities, while SpiffWorkflow introduces BPMN capabilities but is not useable on its own for this research. A potential solution is to overcome the lack of a BPMN engine in Prefect by using the parser of SpiffWorkflow. However, the most straight-forward solution is to use Zeebe through the Python gRPC client or Camunda through its REST API.

Table A.2: Survey of Open-Source Workflow Management Systems

Workflow Engine	Language	Type	API	Visualization	Polling	Fault Tolerance
Camunda [82]	Java	BPMS	REST	BPMN	Yes	Yes
Zeebe [65]	Java	BPMS	gRPC	BPMN	Yes	Yes
WALKOFF [83]	Python	WfMS	REST	Proprietary	Yes	No
SpiffWorkflow [84]	Python	WfMS	None	BPMN	No	No
Prefect [85]	Python	Data Pipeline	GraphQL	Proprietary	Yes	Yes
Cadence [86]	Go	WfMS	Thrift	Proprietary	Yes	Yes



### A.3 Web Application Programming Interfaces

Application Programming Interfaces (APIs) expose the functionality of a service to clients through a representation that is both human and machine-readable. The human-readable component allows creation of a Domain Specific Language (DSL) to add semantic meaning to what the service provides. Therefore, an API created for a KBE product model can facilitate information exchange in a workflow. Over the years, several popular API description languages have emerged as illustrated by Figure A.1.

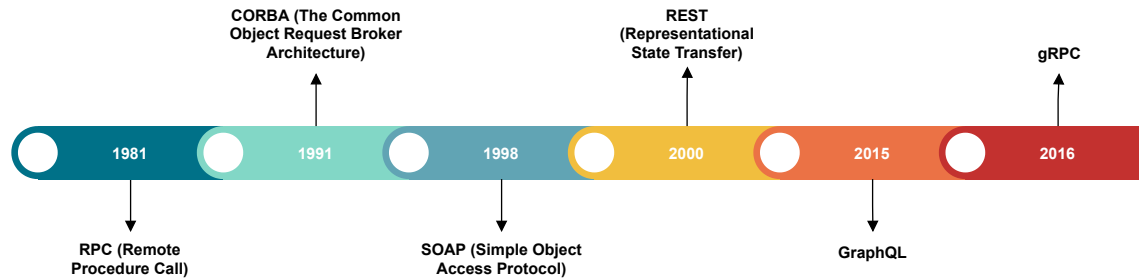


Figure A.1: Timeline of Popular API Description Languages [87]

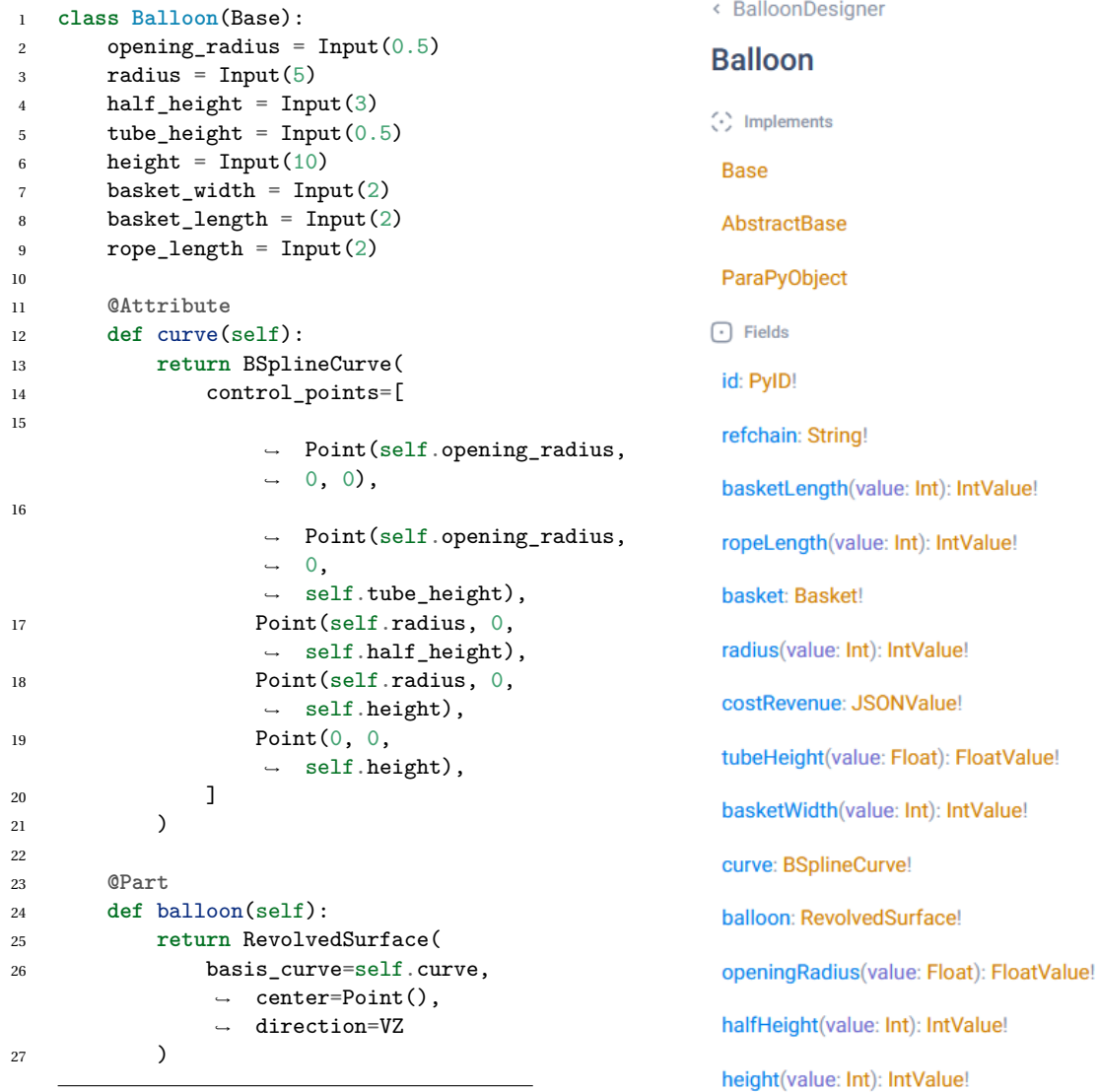
Since its release in 2000, REST has become the backbone of the modern web and is the standard choice of 82 % of developers [88, p.49]. A REST API, exposes several URI “end-points” that accept requests to be encoded in a JSON body. Since each end-point exposes a single data-type such as “users” or “comments”, under-fetching and over-fetching problems are common. The former means that requests to multiple end-points are needed to retrieve a user’s comments. The latter means that unnecessary data is received in the response that must then be filtered. These problems cause overhead for client devices and incur additional data bandwidth.

Addressing these problems for mobile devices motivated Facebook to develop GraphQL [89]. Instead of forcing the client to do extra processing to normalize data, GraphQL allows all data types to be served over a single end-point, enabling clients to ask for exactly the data they expect to receive. The “graph” part of the name originates from its usability for resolving queries such as obtaining the friends of an individual who have all seen a particular movie; thereby traversing the edges of a data graph. In a recent survey of industry professionals, the highest demanded customer features were APIs that fit a specific business need and better documentation [66, p.18]. GraphQL’s query capability, typing system, integral documentation, and ability to be served over a single end-point make it attractive to supply semantically rich and flexible APIs.

The most recent addition to the API landscape is Google’s gRPC, which supports bi-directional streaming and uses a binary format that is more efficient than the JSON strings used by REST and GraphQL. However, it lacks the same degree of tooling that is available for the GraphQL ecosystem. These tools enrich the developer experience through code-completion and type-hinting features available as plugins for most Integrated Development Environments (IDEs). A unique feature of the GraphQL ecosystem is Apollo Federation, which tackles the limitations of monolithic schemas by enabling geographically distributed development teams to relate their GraphQL schemas to form a unified API. Here, the versionless principle of GraphQL allows companies to develop their APIs gradually; letting the federated schema naturally emerge over time. This could provide the flexibility needed to allow “ad-hoc” extensions and speed-up the slow development pace of CPACS [90, p.16]. Overall, 75 % of respondents state that GraphQL will become the dominant API of the future, and might achieve critical mass in the coming years [66, p.32]. Combined with its impressive features, the state-of-the-art nature of GraphQL make it attractive to investigate as the API description language to serve KBE product models.

## B Information Modeling

The purpose of this section is to provide additional information for how the information modeling through the use of GraphQL functions behind the scenes. First off it is useful to examine how the generated GraphQL schema relates to application source code. This relationship is given by Figure B.1. Here, one can observe how slots within the KBE application are represented by fields on the resultant GraphQL schema. Furthermore, input slots (or any settable slot for that matter) can be recognized by the presence of a value argument. This allows these fields to be used in mutations to perform updates to the inputs of the application. An example is the `openingRadius` field. One can also notice in the schema that inheritance is implemented via interfaces. For example, the `Balloon` inherits from `Base`. In the GraphQL schema, one can then see the Method Resolution Order (MRO) starting with `Base`, and progressing onto the superclasses `AbstractBase` and `ParaPyObject`. Representing inheritance this way is beneficial to be able to query the common attributes of classes when they are used in type unions. An example of this is provided later in Chapter C.



(a) ParaPy Source Code

(b) GraphQL Schema

Figure B.1: Comparison of Source Code and Transpiled GraphQL Schema



As mentioned in Section 3.2.1, without slot resolvers the generated schema would not have the knowledge required to be able to serve queries. The basic function of a slot resolver as demonstrated below by Listing B.1, is to convert between GraphQL and ParaPy types. Each slot resolver defines a `can_resolve` method which is called by the transpiler to determine if the current `SlotResolver` is capable of representing the provided slot. Using this methodology makes it possible to add new slot resolvers without needing to modify existing code, which makes it easier to improve the coverage of the transpiler gradually over time. In this case, Listing B.1 provides the source code of a `BaseResolver`, which is responsible for obtaining a base instance from a given parent, here referred to as an owner. One can see in Line 21, how a `getattr` call is made using the encoded slot name to retrieve the requested instance. Then the encoded GraphQL type is instantiated by passing the obtained instance as its owner. In this way, the product tree can be traversed as each accessed child will put itself as an owner until the selection set of the query is fully resolved.

Whats more is that a `SlotResolver` is responsible for determining the correct return type of the generated resolver method. This returned resolver function is defined as a closure, which makes it convenient to access data such as the GraphQL Object Type as well as the name of the ParaPy slot this resolver should be bound to. On a side note, a way to understand where GraphQL gets its name is to examine the nodes and edges of a GraphQL schema, Figure B.2. Essentially, an Object type relates to other types through its fields which are then the edges of a graph.

---

```

1  class BaseSlotResolver(OptionalSlotResolver[V], register=True):
2      @staticmethod
3      @exclude_union
4      @exclude_sequence
5      @exclude_builtin_sequence
6      def can_resolve(slot_data: SlotData) -> bool:
7          return_type = strip_optional(slot_data.return_type)
8          if get_origin(return_type) is None:
9              bases = extract_baseclasses(return_type)
10             return len(bases) == 1
11         return False
12
13     def compile_resolver(
14         self,
15     ) -> Callable[..., Awaitable[V]]:
16
17         gql_type = self.graphql_type
18         slot_name = self.slot.__name__
19
20         async def resolver(self: Base):
21             instance = getattr(self.owner, slot_name)
22             if instance is None:
23                 return None
24             return gql_type(owner=instance)
25
26         resolver.__annotations__.pop("self")
27         resolver.__annotations__["return"] = self.return_annotation
28         return resolver
29
30     @cached_property
31     def graphql_type(self) -> Type[Any]:
32         base_class = extract_baseclasses(self.slot_data.return_type).pop()
33         return self.type_map[base_class]

```

---

Listing B.1: Example Slot Resolver used to Access ParaPy Base Instances

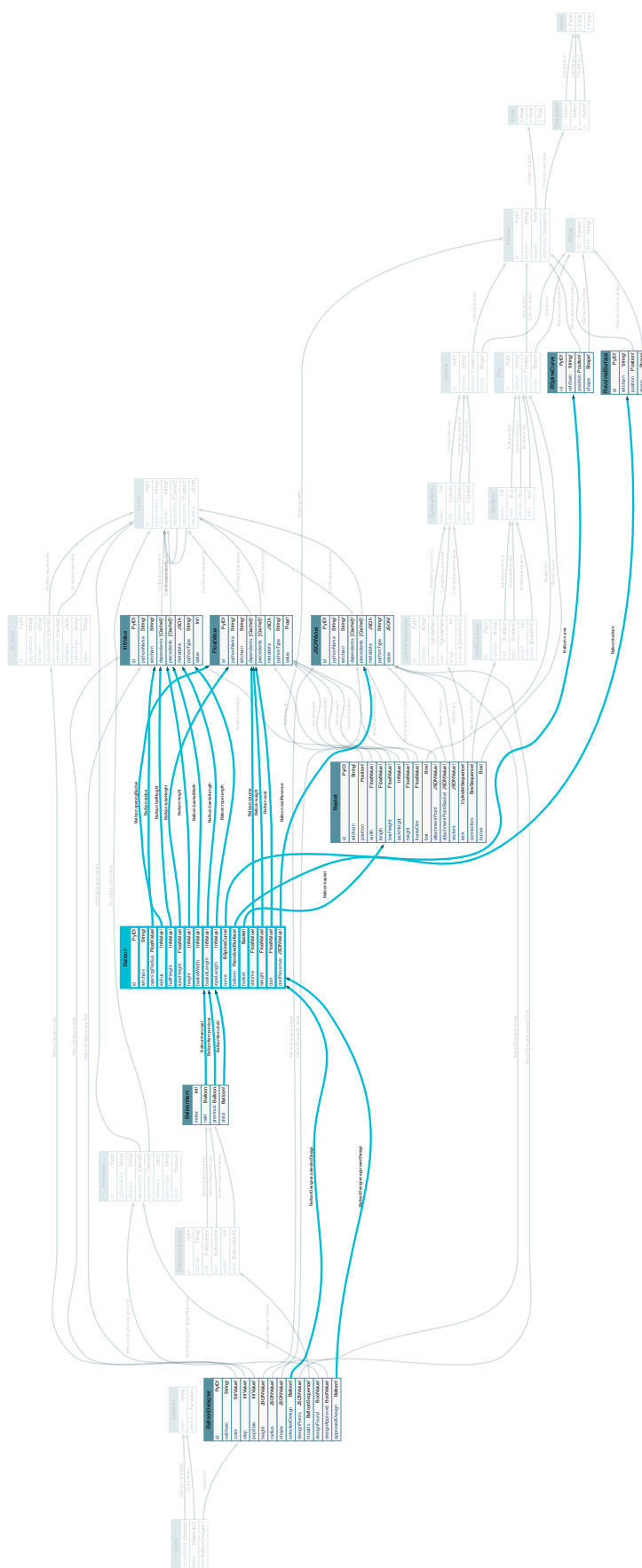


Figure B.2: Transpiled Hot Air Balloon Schema Visualized in GraphQL Voyager

With the model persistence functionality inplace, it was quickly realized that the performance of the GraphQL API startup time is often worse than the time it takes for an application instance to be relaunched. This is detrimental in situations where a KBE application is restarted often within a process. Although the intended usage for the GraphQL API is for a single process to be kept alive that is capable of serving multiple design instances at once, it is still worthwhile to investigate what consumes the most time. For this purpose the profiling tool pyinstrument is used to determine which calls within the transpilation algorithm consume the most time. The results of this analysis on the Warehouse application for both with and without inferencing are given by Figures B.3 and B.4 respectively. Here it is striking to see that the majority of time is spent creating GraphQL object types during schema launch. In fact, the transpilation algorithm developed for the thesis consumes just 1.1 % of the total transpilation time. Even with inferencing enabled, the majority of transpile time is still due to creation of GraphQL object types. One can argue that the high number of generics types used to model the transpiled schema could be causing this slowdown. Nonetheless, this requires further investigation.

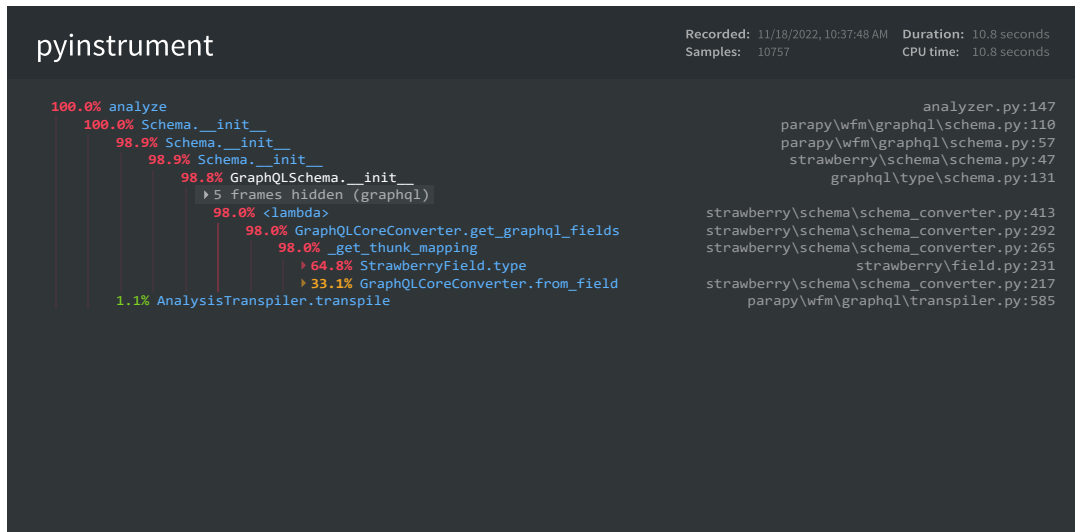


Figure B.3: Profiling Result of Warehouse Application (Without Inferencing)

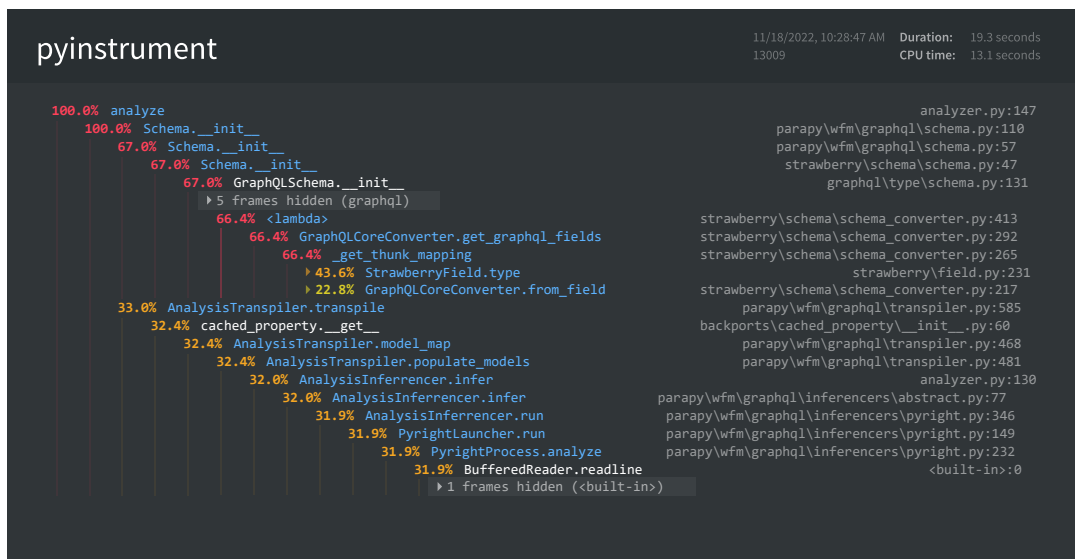


Figure B.4: Profiling Result of Warehouse Application (With Inferencing)

## C Example Queries

The purpose of this appendix is to demonstrate how the information of a KBE application can be accessed via the GraphQL API. Furthermore, the aim is to demonstrate how this query language satisfies the primary language features in KBE. Starting off with a simple query given by Figure C.1, the `is_sphere` slot of a KBE application is queried using the provided syntax code. Notice how the provided response structurally resembles the provided query.



Figure C.1: Query Formulation for Requesting a Boolean Value Slot

If one were to then desire to change the value of `is_sphere` a so-called mutation would be required. An example of a mutation is then provided by Figure C.2. Here the operation type is specified as a mutation and a value argument is passed, in this case a `true` value to change the value of the slot from the previous query. Note that a selection set is always required when dealing with scalar values such as booleans as currently GraphQL does not support scalar unions. Therefore, for this reason along with the desire to expose further metadata within a value, an Object type is chosen to represent scalars.



Figure C.2: Mutation Formulation for Updating a Boolean Value Slot

## Dependency Tracking

As previously mentioned it is desirable to access the dependents or precedents of the caches of a ParaPy instance. In the example below, the same `is_sphere` slot is queried for dependent values. At the same time a dynamic type in the KBE application is queried which depends on the truthfulness of `is_sphere`. While dynamic types will be covered in the subsequent section, the inline fragment used is required as otherwise the specific field `radius` is not otherwise accessible on the other available type in this schema, which is a `Box`. As a result, if one were to leave out the inline fragment, then the `radius` field would not be queriable, and GraphQL or other GraphQL linters would raise an error.

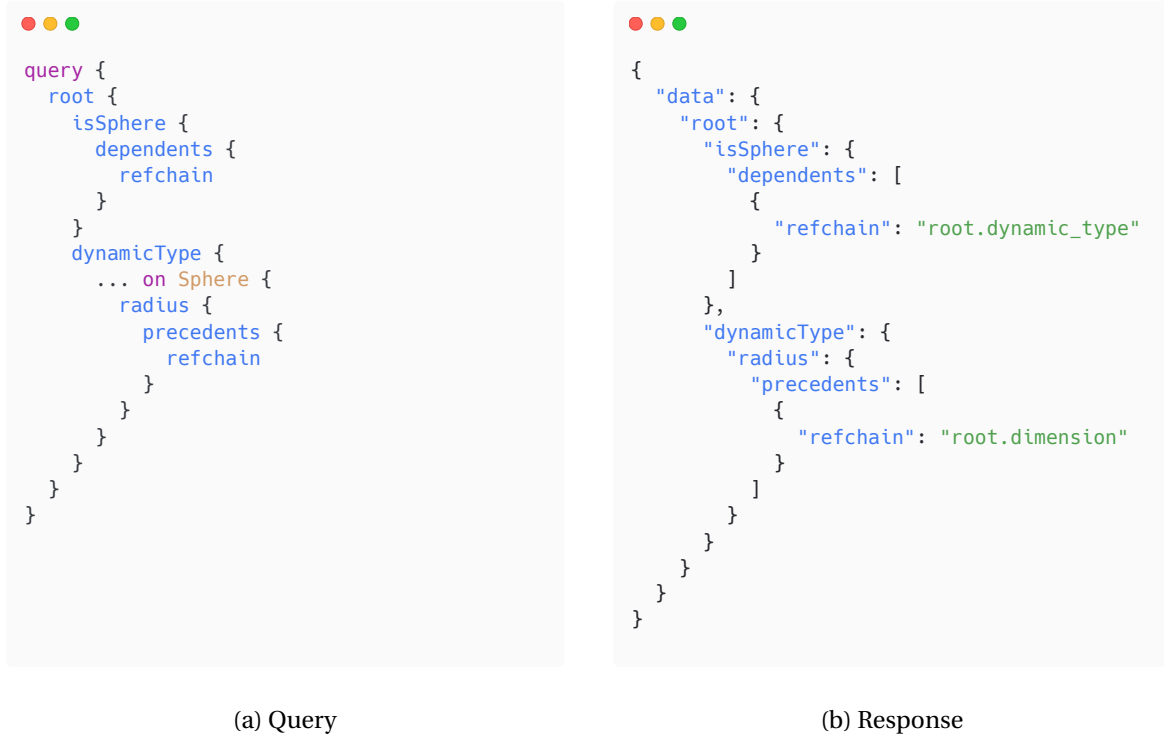


Figure C.3: Query Formulation for Accessing Dependency Information

## Dynamic Type

Similar to the previous example, an inline fragment is once again required to access the different slots of a field returning a type union. The difference here is that the query includes both the `Box` and `Sphere` type. Such a query could be required at runtime to handle the unknown return type. For example, if a tool were being developed to analyze the volume of these shapes, the tool would need to be aware that the returned type can either be a `Box` or a `Sphere`. If the tool then had the capability to calculate the volume of both of these types, then it would perform a query like the one given below. Although only one type will be returned, the tool would perform the query with the intention of fetching either of the two types. As a result, type unions in GraphQL are a powerful concept that provide support for the dynamic type construct in the KBE paradigm.

```
query {  
  root {  
    isSphere {  
      value  
    }  
    dynamicType {  
      ... on Sphere {  
        radius {  
          value  
        }  
      }  
      ... on Box {  
        width {  
          value  
        }  
        length {  
          value  
        }  
        height {  
          value  
        }  
      }  
    }  
  }  
}
```

(a) Query

```
{  
  "data": {  
    "root": {  
      "isSphere": {  
        "value": true  
      },  
      "dynamicType": {  
        "radius": {  
          "value": 1  
        }  
      }  
    }  
  }  
}
```

(b) Response

Figure C.4: Query Formulation for a Requesting a Dynamic Type

## Sequence

Another important aspect to support in KBE applications is the ability to index specific items from within a sequence. Although GraphQL has the Relay Specification<sup>i</sup>, for engineers, especially those using ParaPy are used to working with Python. As a result, adoption of the Python slices seemed more intuitive. As can be seen from the example below, the first two items in the sequence can be queried through the GraphQL API.

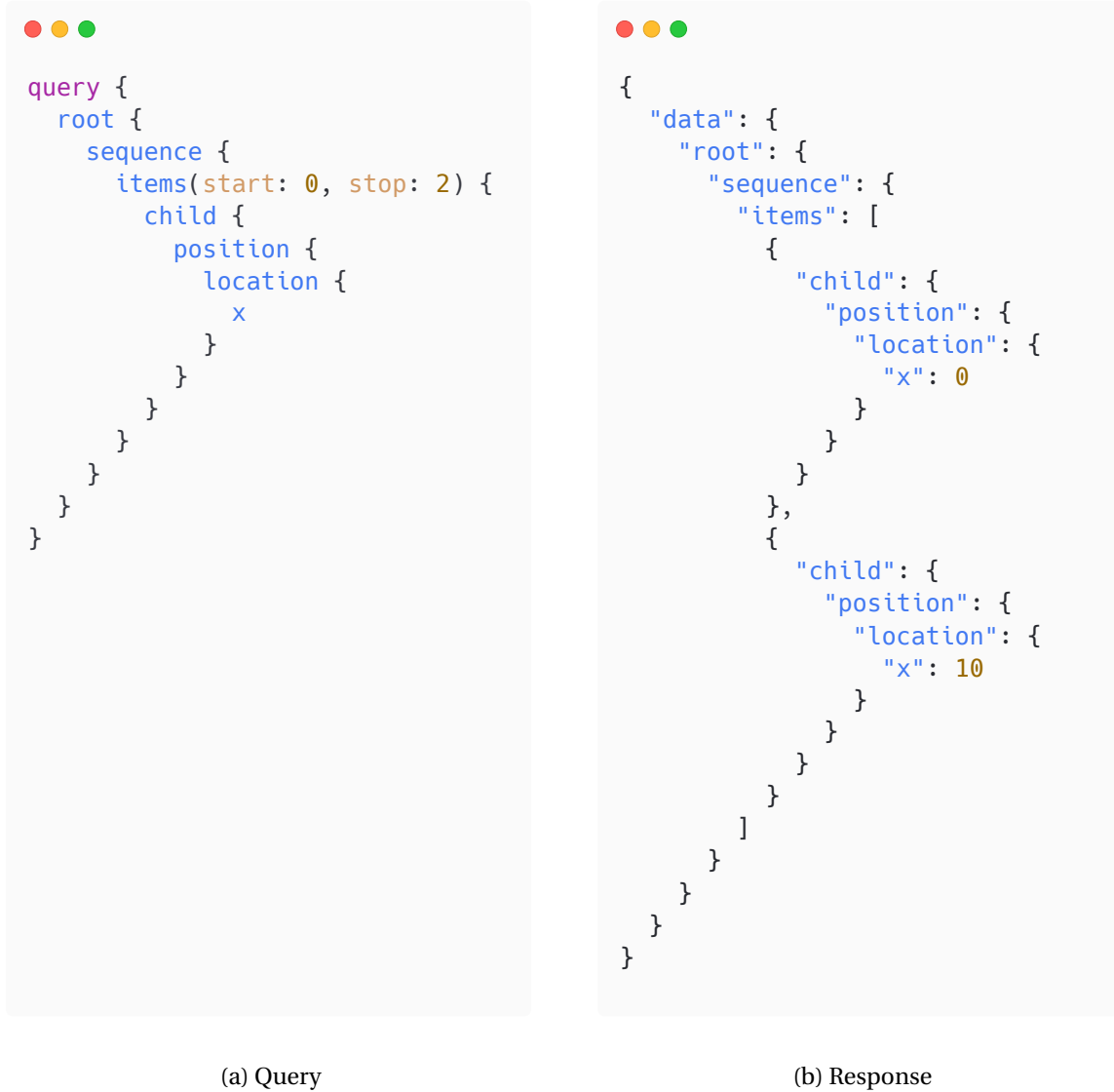


Figure C.5: Query Formulation for Accessing Items of a Sequence

Similarly, slice syntax can be used to mutate a specific item in a sequence as shown below. Here the width of the 5th item in sequence is changed to a value of 2. What is important to grasp, is that the ability to perform such fine-grained queries and mutations is a cornerstone to enabling the correlated dependency feature.

<sup>i</sup><https://relay.dev/docs/guides/graphql-server-specification/>



```
mutation {  
  root {  
    sequence {  
      items(start: 4, stop: 5) {  
        index  
        child {  
          width(value: 2) {  
            value  
          }  
        }  
      }  
    }  
  }  
}
```

(a) Mutation

```
{  
  "data": {  
    "root": {  
      "sequence": {  
        "items": [  
          {  
            "index": 4,  
            "child": {  
              "width": {  
                "value": 2  
              }  
            }  
          }  
        ]  
      }  
    }  
  }  
}
```

(b) Response

Figure C.6: Mutation Formulation for Updating an Item in a Sequence

## Dynamic Sequence

Dynamic type functionality can also be used on sequences as ParaPy supports the notion of a dynamic sequence. In the query below, the sequence can either consist of a Square or a Box. While the syntax for accessing the dynamic type is similar to before, an interesting aspect to this query is the inclusion of the `GeomBase` interface. What this shows is that even with the presence of dynamic types, common fields between the Square and Box can be queried if the inline fragment is set to a common interface. In this case, both Box and Sphere derive from `GeomBase`, making it possible to query the position on both return types.

```
query {
  root {
    dynamicSequence {
      items(start: 0, stop: 2) {
        child {
          __typename
          ... on Box {
            width {
              value
            }
          }
          ... on Sphere {
            radius {
              value
            }
          }
          ... on GeomBase {
            position {
              location {
                x
              }
            }
          }
        }
      }
    }
  }
}
```

(a) Query

```
{
  "data": {
    "root": {
      "dynamicSequence": {
        "items": [
          {
            "child": {
              "__typename": "Box",
              "width": {
                "value": 1
              }
            },
            "position": {
              "location": {
                "x": 0
              }
            }
          },
          {
            "child": {
              "__typename": "Sphere",
              "radius": {
                "value": 1
              }
            },
            "position": {
              "location": {
                "x": 10
              }
            }
          }
        ]
      }
    }
  }
}
```

(b) Response

Figure C.7: Query Formulation for Accessing Items of a Dynamic Sequence

## Geometry

Another feature of the GraphQL API is the ability to ask for the geometry of a specific shape within the product tree. What makes this feature worthwhile is the ability for the client to specify in what exchange type it would like to receive the shape. In this case STEP is used and the query specifies that the dimensions should be in mm. Note that the output is truncated for in order to be able to represent the response on the page. In the future, this feature could allow heterogeneous tools to query for the geometry in a suitable exchange format, and make use of the geometry generated by the KBE application. As a result, a KBE service could be deployed that is in charge of generating geometries for optimizations. A potential use-case could be the MMG.



```

query {
  root {
    dynamicType {
      ... on DrawableShape {
        shape {
          STEP(unit: MM)
        }
      }
    }
  }
}

```

```

{
  "data": {
    "root": {
      "dynamicType": {
        "shape": {
          "STEP": "ISO-10303-21;..."
        }
      }
    }
  }
}

```

(a) Query

(b) Response

Figure C.8: Query Formulation for Accessing Geometry

## D Examples of Dynamism

The purpose of this appendix is to provide examples for when KBE assisted dynamism can be introduced into workflows. Use of KBE assistance to achieve greater dynamism in workflows is often a necessity if the business rules needed to reach a decision are difficult to formalize outside of code. Starting off with an example where one may need formulate rules within the KBE application to determine who should be assigned as task, Figure D.5. In this hypothetical scenario, an human-in-the-loop optimization is running where certain constraint violations require manual intervention from different engineers. Since the business logic of what type of constraint violations are possible lives within the KBE application, it may be wiser to keep such logic closer to the application itself.

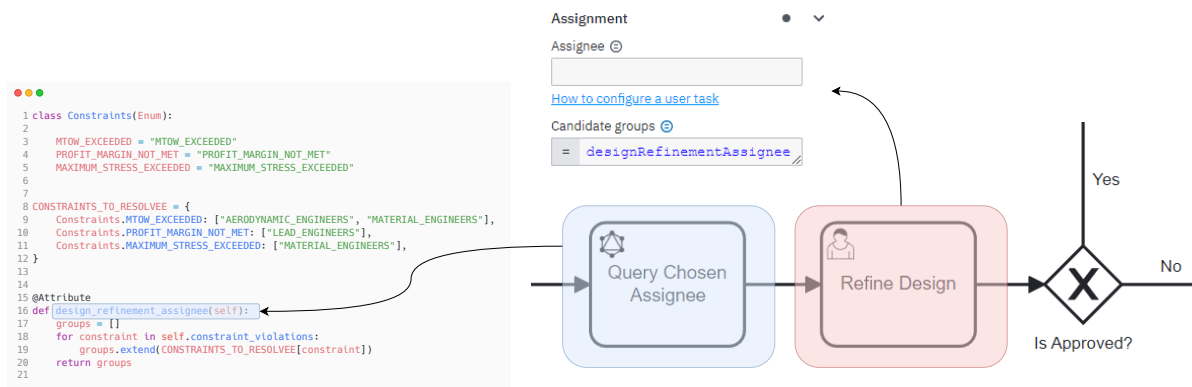


Figure D.1: Example of a Dynamic Selection of Assignee (Who)

Another form of dynamism, is the selection of which worklet to run. This should not be confused with the `@Worklet` decorator, however the worklet terminology is reused as Figure D.2 depicts a situation where the KBE application is responsible for deciding which simulation to run. In this hypothetical case, the decision is based off of a flag variable to determine if a high fidelity simulation is required.

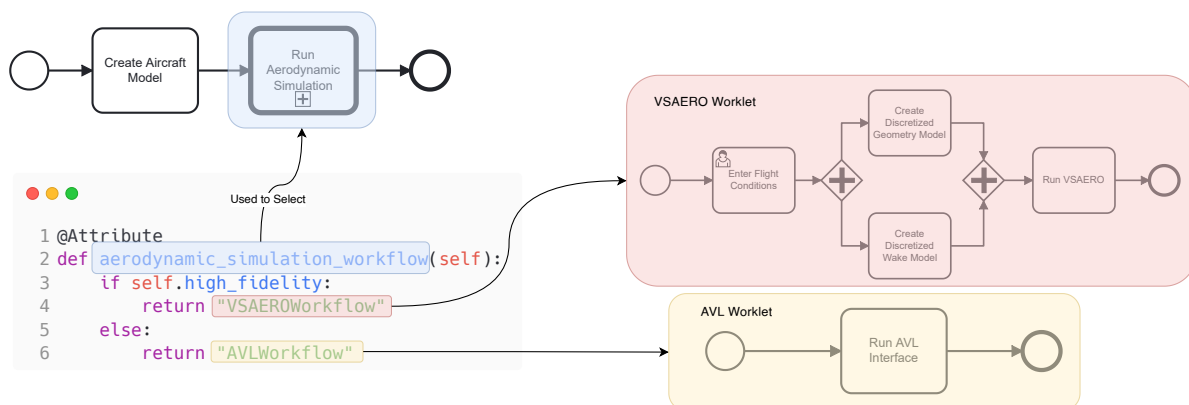


Figure D.2: Example of a Dynamic Selection of Worklet (What)

Another example of KBE assisted dynamic control flow is the usage of the runtime cache to skip a certain task. Figure D.3 below demonstrates how the design of experiments task can be skipped based on if the design points are already available.

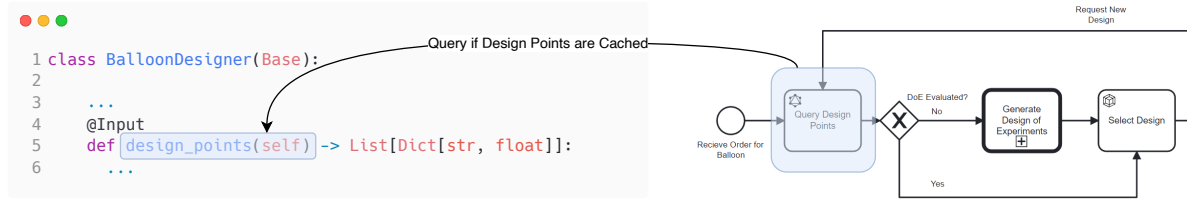


Figure D.3: Example of Dynamic Task Skipping (What)

Moving onto timing related dynamism, Figure D.4 depicts an example situation where the report must be shown to a customer within business hours. In this particular use-case the calculation for determining the timer delay necessary to send the report at the right time would be complicated to express in the business process itself. As a result, use of the KBE application can help in this situation.

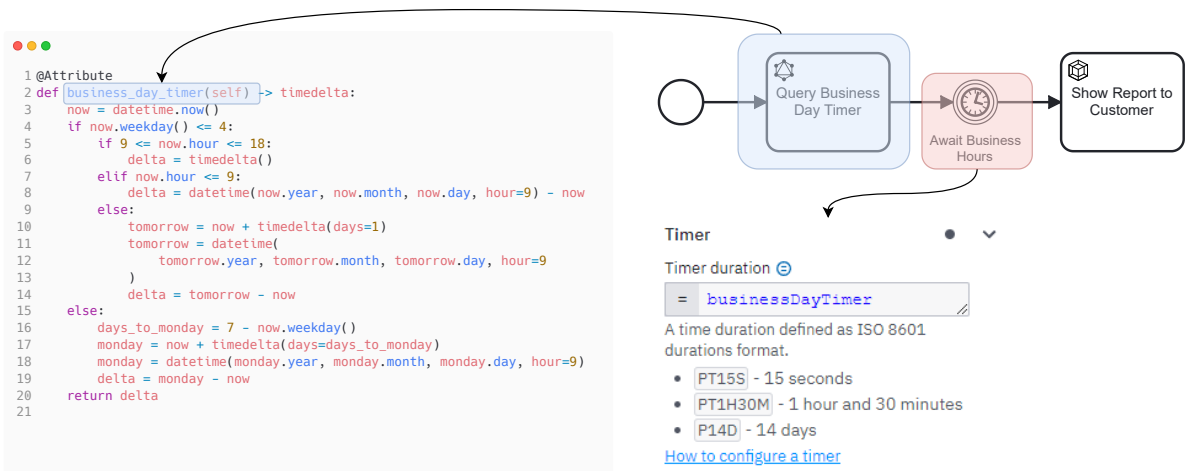


Figure D.4: Example of Dynamic Timing (When)

Finally, the last form of dynamism recognized is “how”, whereby a dynamic selection is made between a number of options. In Figure D.5, the KBE application is responsible for making a call to a backend service to determine if an automated tool is currently available. If the tool is available, then the workflow will make use of it. Otherwise, a user task is will be activated.

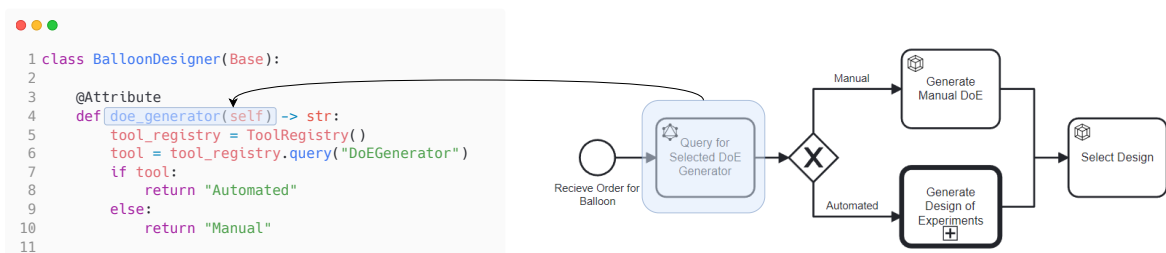


Figure D.5: Example of a Dynamic Task Selection (How)

# E Algorithm Formalization

The purpose of this appendix is to explain two algorithms that were developed during this thesis. Expressed as psuedo-code the first algorithm below is the one implemented by the GraphQL transpiler to create GraphQL Object Types (Models) and traverse through KBE application source code, Figure E.1. Notable aspects of this algorithm is how slot inferencing, which is an expensive per-call operation is deferred to infer as many slots at the same time as possible.

```
Input: Initial Set of Origins:  $O \leftarrow \{o_i \dots o_n\}$ 
Input: Initial Set of GraphQL Models:  $M \leftarrow \{m_i \dots m_n\}$ 
Input: Map of Origins to GraphQL Models:  $f : O \rightarrow M$ 

/* Outer Loop for Handling Deferred Inferencing */
while  $O \neq \emptyset$  do
   $I \leftarrow \emptyset$  /* Set of slots to infer */
  /* Inner Loop for Depth-First Origin Discovery */
  while  $O \neq \emptyset$  do
    /* Remove an origin from set */
     $O \leftarrow O \setminus \{o_n\}$ 
    if  $f(o_n) \notin M$  then
      /* Create new model type and update mapping */
       $m_n \leftarrow \text{CreateModel}(o_n)$ 
       $M \leftarrow M \cup \{m_n\}$ 
       $m_n \mapsto f(o_n)$ 
    end
    forall  $s \in \text{GetSlots}(m_n)$  do
       $D \leftarrow \{d \in \text{DiscoverOrigins}(s) \mid f(d) \notin M\}$ 
      if  $D \neq \emptyset$  then
        /* Update set with discovered origins */
         $O \leftarrow O \cup D$ 
      else
        /* Defer slot inferencing by adding to set */
         $I \leftarrow I \cup \{s\}$ 
      end
    end
  end
  /* Run inferencer and update set with inferred origins */
   $O \leftarrow O \cup \{i \in \text{InferOrigins}(I) \mid f(i) \notin M\}$ 
end
```

Figure E.1: Graph Contraction Algorithm

The second algorithm on graph contraction, Figure E.2, which is visually represented by Figure E.3, allows simplification of the dependency graph from an original representation given by Figure E.4a to Figure E.4b. The algorithm works by traversing the dependencies of composed caches of an unserializable base instance until a serialized value is found. To prevent traversing the same path multiple times, discovered dependencies are back propagated along the initial traversal path. Therefore, akin to how a Merkle tree stores the hashes of its descendant nodes, a record is maintained for all visited caches that stores its discovered dependents. To prevent premature contraction, the number of outgoing paths from a node is counted to make sure that results are back propagated only when all paths have been traversed.

**Input:** Set of Serialized Caches:  $S \leftarrow \{s_i \dots s_n\}$   
**Input:** Set of Unserializeable Base Caches:  $U \leftarrow \{u_i \dots u_n\}$   
**Input:** Map of Serialized Caches to Precedents:  $f_p : s_i \rightarrow \{p_i \dots p_n\}$   
**Input:** Map of Serialized Caches to Dependents:  $f_d : s_i \rightarrow \{d_i \dots d_n\}$

```

1  $f_c : c_i \rightarrow \{d_i \dots d_n\}$  /* Map of Caches to Contracted Dependents */

/* Outer Loop to Iterate of All Unserializeable Base Caches */
2 forall  $u \in U$  do
3    $V_d \leftarrow \emptyset$  /* Set of Visited Dependents */
4    $V_p \leftarrow \emptyset$  /* Set of Visited Precedents */

5    $\rho \leftarrow (u)$  /* Path Sequence to Current Cache */
6    $T \leftarrow ((u, \rho))$  /* Sequence of Cache, Path Pairs to Contract */

/* Inner Loop to Traverse over Dependencies */
7 while  $|T| \neq 0$  do
8    $(c_n, \rho_n) \leftarrow \text{PopLast}(T)$ 
9   forall  $d \in \text{Dependents}(c_n)$  do
10    /* Use Previous Contraction Result */
11    if  $f_c(d) \neq \emptyset$  then
12      forall  $e \in f_c(d)$  do
13         $f_d(u) \leftarrow f_d(u) \cup \{e\}$ 
14         $f_p(e) \leftarrow f_p(e) \cup \{u\}$ 
15      end
16      Backpropagate( $f_c(d), \rho_n$ )
17    end
18    /* Found New Dependency */
19    else if  $d \in S$  then
20       $f_d(u) \leftarrow f_d(u) \cup \{d\}$ 
21       $f_p(d) \leftarrow f_p(d) \cup \{u\}$ 
22      Backpropagate( $f_c(d), \rho_n$ )
23    end
24    /* Reached Previously Traversed Dependency */
25    else if  $d \in V_d$  then
26      EnqueuePath( $d, \rho_n$ )
27    end
28    /* Add Current Cache and Path to Iteration Sequence */
29    else
30       $T \leftarrow (t_0, \dots, t_i, \dots, (d, \rho_d))$ 
31       $V_d \leftarrow V_d \cup \{d\}$ 
32    end
33    /* Prevent Visiting Dependent as Precedent */
34     $V_p \leftarrow V_p \cup \{d\}$ 
35  end
36  forall  $p \in \text{Precedents}(c_n) \notin V_p$  do
37    if  $p \in S$  then
38       $f_d(p) \leftarrow f_d(p) \cup \{u\}$ 
39       $f_p(u) \leftarrow f_p(u) \cup \{p\}$ 
40    end
41  end
42 end
43 end

```

Figure E.2: Graph Contraction Algorithm



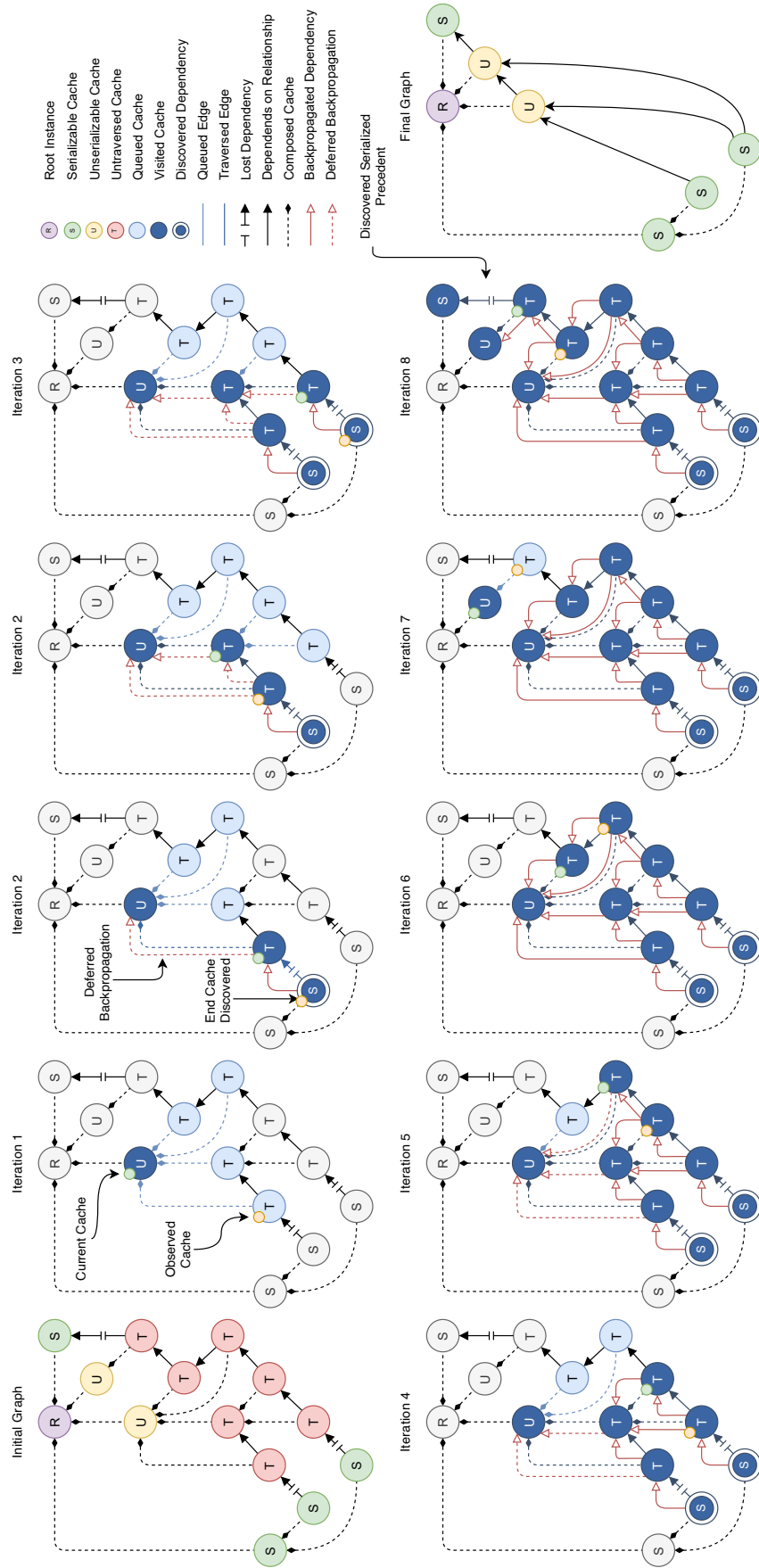
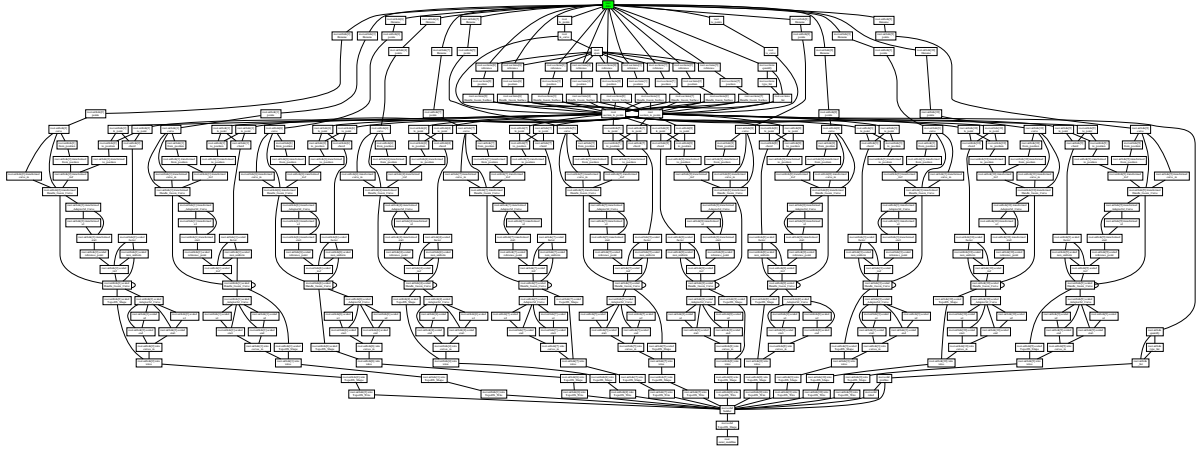
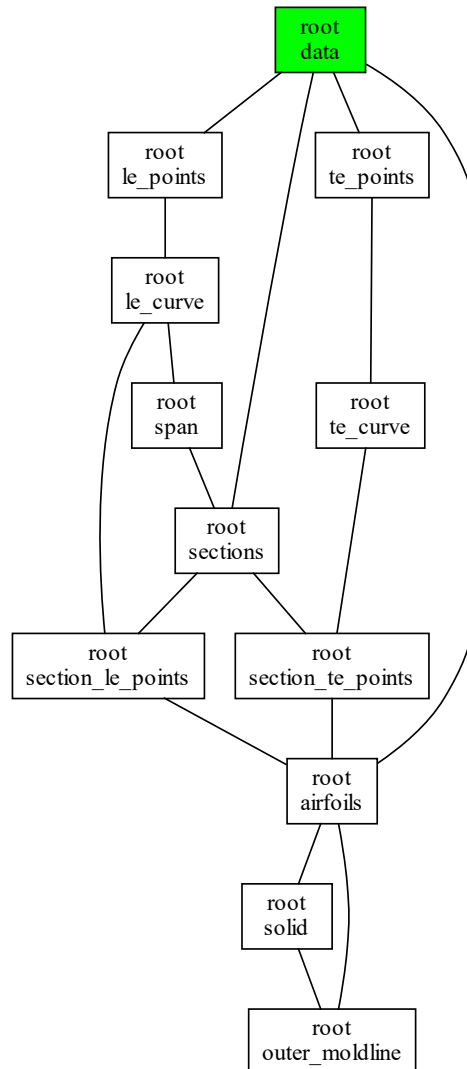


Figure E.3: Visualization of Graph Contraction Algorithm



(a) Before



(b) After

Figure E.4: Wing Model Dependencies Before and After Graph Contraction