DTU Compute
Department of Applied Mathematics and Computer Science

A Platform-Independent Method for Fully Automating Business Process Execution

Master Thesis

A Platform-Independent Method for Fully Automating Business Process Execution

Master Thesis
March, 2026

By
Nicolai Pavliuc

Cover photo: Vibeke Hempler, 2012
Published by: DTU, Department of Applied Mathematics and Computer Science, Richard Petersens Plads, Building 324, 2800 Kgs. Lyngby Denmark
https://www.compute.dtu.dk/

Approval

This thesis has been prepared over five months at the Department of Applied Mathematics and Computer Science, at the Technical University of Denmark, DTU, in partial fulfilment for the degree Master of Science in Engineering, MSc Eng.

It is assumed that the reader has a basic knowledge in the areas of computer science.

Nicolai Pavliuc - s240366

.....

Signature

.....

Date

Abstract

Business Process Management Systems (BPMS) are widely adopted in organizations to automate and orchestrate business processes. This thesis addresses vendor lock-in in BPMS by proposing a platform-independent modeling, validation, and artifact-generation pipeline that targets multiple execution engines. The core idea is to separate *what* a business process means from *how* a specific BPMS requires it to be packaged and configured. The thesis defines a vendor-neutral modeling set centered on BPMN 2.0 and complements it with additional models required for execution (e.g., organization, data, user forms, formal expressions, and complex decision logic). A set of modeling guidelines and validation rules ensures consistency between them.

A concrete prototype is implemented that generates engine-specific deployment bundles for *Camunda 7* and *Bonita*. The generator transforms the vendor-neutral model set into executable artifacts such as updated BPMN definitions, forms, DMN files, and engine-specific configuration files. The approach is evaluated through end-to-end demonstrative validation: representative case processes are modeled once, passed through the pipeline, and the generated outputs are deployed and executed on both target engines. Results show that the proposed approach can produce deployable and runnable process applications without manual correction of the generated core definitions for the evaluated cases, supporting the feasibility of vendor-neutral modeling with automated multi-engine generation.

Acknowledgements

Giovanni Meroni, Associate Professor, Department of Applied Mathematics and Computer Science, DTU Compute
Thesis supervisor. I thank him for his guidance and feedback throughout this work.

Nicolai Pavliuc, MSc Eng in Computer Science and Engineering, DTU
Thesis author.

Contents

1 Introduction

1.1 Context and Motivation

Business Process Management Systems (BPMS) are widely adopted in organizations to automate and orchestrate business processes [1, 2]. Typically, the automation process follows a structured path. First, business requirements are captured. Then, a BPMN 2.0 (Business Process Model and Notation) diagram is created to represent the control flow of the process [3]. Finally, this BPMN model is manually extended with engine-specific execution details such as user task form references, lane/actor assignments, service task connector/delegate configuration for REST and email, decision model (DMN) bindings for business rule tasks, and other vendor-specific attributes required for deployment on a specific BPMS such as Camunda, Bonita, Oracle BPM, etc. These technical extensions are not part of the BPMN standard but are required for deployment. As a result, BPMN models become tightly coupled to the chosen engine, making migration, reuse, and maintenance difficult.

1.2 Problem Statement

The current practice of embedding engine-specific execution configuration into BPMN models creates strong coupling between process definitions and the selected BPMS [4]. When organizations migrate to another platform, they have to manually re-implement execution details (e.g., form references, actor assignment mechanisms, connector/delegate configurations for REST calls, etc.), making migration complex, costly, and error-prone. At the same time, the information required to automate processes is most of the time not represented through structured, platform-independent models (e.g., data models, form specifications, organizational models, and decision models) [5, 6]. As a result, execution requirements are harder to validate and keep consistent, leading to reduced maintainability, limited reuse, and poor portability across BPMS platforms. The last element has a strong negative effect since it causes vendor lock-in [7].

1.3 Goal and Scope

The goal of this thesis is to develop and evaluate a platform-independent pipeline for process automation that reduces coupling between BPMN process models and a specific BPMS. The approach captures execution-relevant requirements in a structured set of complementary, vendor-neutral models, validates their consistency [8, 9], and generates BPMS-specific deployment artifacts for target engines.

This thesis targets a vendor-neutral perspective: the platform-independent models are treated as the primary source of truth, and engine-specific artifacts are generated from them rather than modeled manually using vendor extensions. To demonstrate portability across genuinely different platforms, the prototype generates artifacts for two distinct BPMS engines that follow different execution and configuration approaches (i.e., not closely related forks). Empirical validation is done on versions of BPMS that are free and publicly accessible. Validation on enterprise and paid versions of BPMS is not feasible within the scope of this thesis.

In terms of tooling and modeling choices, the work prioritizes freely available and open-source standards and tools to ensure reproducibility and accessibility. In the initial information-requirements phase, though, enterprise BPMS documentations were also consulted.

The implemented pipeline targets commonly used BPMN constructs and automation needs found in practical BPMS deployments, rather than aiming for exhaustive coverage of the full BPMN 2.0 specification [3].

1.4 Research Questions

The overall research question that this thesis addresses is:

> Can business processes be modeled and executed in a platform-independent way that reduces engine-specific coupling and enables automated deployment across multiple BPMS engines?

To address this question, the following sub-questions will be investigated:

1.4.1 RQ1.1: How can business requirements be systematically captured and represented using complementary modeling languages?

A platform-independent approach requires identifying what execution information is needed beyond only BPMN control flow. This might be information about forms, data, user assignment, etc. Without a precise list of information requirements, modeling choices become ad hoc, leading to gaps or a need for extra vendor-specific extensions. This question establishes the information foundation needed before selecting modeling languages and defining transformation rules [5, 10].

1.4.2 RQ1.2: How can synchronization and consistency between models be defined and maintained?

Once information is split across models, consistency becomes a first priority: changes in one model may invalidate assumptions in others, and the generated engine-specific artifacts can quickly become outdated or incorrect if these dependencies are not managed [8, 9]. This motivates defining explicit consistency relations and synchronization mechanisms.

1.4.3 RQ1.3: How can platform-independent models be transformed into engine-specific execution artifacts?

A vendor-neutral modeling approach only becomes practically useful if the platform-independent model set can be systematically parsed into platform-specific artifacts that a concrete BPMS can deploy and execute [11, 12]. This step is unavoidable because engines require concrete deployment inputs such as executable process definitions, decision models, form files, actor/identity mapping, etc. Therefore, RQ1.3 is necessary to specify what target artifacts are required per engine, and define and implement the mapping/transformation logic that turns platform-independent models into deployable, engine-specific execution artifacts.

1.5 Contributions

This thesis follows a design-oriented approach in which an artifact is built: a vendor-neutral pipeline that systematically captures execution requirements in platform-independent models, validates their consistency, and generates engine-specific deployment artifacts for multiple BPMS platforms.

1.5.1 Contributions list

- **Comparison of BPMS platforms**: The thesis provides a structured comparison of multiple BPMS platforms being used in the industry, summarized in a consolidated table (Appendix B). The table highlights the different features/functionalities supported by different BPMS and how these are implemented (e.g., whether DMN

is supported or what attributes can be specified on a User task), making portability challenges explicit.

- **Vendor-neutral pipeline:**
  - **Information-requirements catalogue and vendor-neutral modeling guidelines** Based on the cross-engine analysis, the thesis identifies a common set of execution-relevant information that repeatedly appears across different BPMS deployments. Modeling languages and modelling guidelines are proposed to capture each information type.
  - **Consistency and linking method for complementary models** To avoid fragmentation across multiple models, the thesis proposes a method for linking the complementary models and defines consistency assumptions/rules that help keep them aligned
  - **Artifact-generation algorithm and prototype implementation** The thesis defines an algorithmic transformation pipeline that takes platform-independent models plus a selected target engine and generates the concrete execution artifacts required for deployment. A Java-based prototype that targets two different BPMS is used for validation.

1.6 Roadmap

The thesis is organized into ten chapters. Chapter 1 introduces the context, problem statement, scope, and research questions. Chapter 2 provides background and related work. Chapter 3 presents the overall approach for conducting the research. Chapter 4 focuses on comparing BPMS platforms, defining information requirements, picking modelling languages to capture them together with guidelines to do the modelling. That section is fully centered around RQ1.1. Chapter 5 describes how the models are linked and validated. That section is fully centered around RQ1.2. Chapter 6 details the artifact-generation approach and the mappings from the platform-independent representation to engine-specific execution artifacts. That section is fully centered around RQ1.3. Chapter 7 goes through the implementation of the prototype for the proposed pipeline. Chapter 8 focuses on evaluating the proposed pipeline. Chapter 9 discusses results of evaluation, trade-offs, limitations, etc. Finally, Chapter 10 concludes the thesis and outlines directions for future work.

2 Background and State of the Art

2.1 Business Process Management and BPMS

2.1.1 Business processes and process automation

A *business process* can be viewed as a coordinated set of activities performed to achieve a business objective, typically producing a service or outcome for an internal or external customer. In workflow standardization, a process definition is treated as an explicit representation of work that can be modeled and (partly) enacted by an information system, including activities, their relations, start/termination criteria, and relevant information about participants, data, and supporting applications [2].

Business Process Management (BPM) is the discipline of designing, executing, monitoring, and improving business processes using methods and technology. A common lifecycle view emphasizes that processes are managed as assets: they are explicitly modeled, deployed into an operational environment, observed through execution data, and refined iteratively based on performance and compliance needs [1].

Process automation refers to implementing process behavior such that work is routed and coordinated by software, with well-defined handoffs between human and automated steps. This aligns with classic workflow management perspectives, where automation is concerned with directing work items and integrating people and systems according to an explicit process definition [2, 1].

2.1.2 What a BPMS provides

A *Business Process Management System (BPMS)* provides technical support for BPM across the lifecycle, typically spanning design-time modeling, deployment, execution, and monitoring/analysis [1]. In practice, BPMS platforms commonly provide the following capabilities:

- **Modeling / design-time support:** tools to create process models, validate structure, manage versions, and support collaboration. BPMN is frequently used as the modeling notation due to its standardized syntax and exchange format [3].
- **Deployment and configuration:** mechanisms to package and publish process definitions together with required runtime resources (e.g., forms, connector configuration, decision models) [1].
- **Runtime execution:** a process engine that interprets the process model, manages process instances, persists state, schedules timers, and routes control flow through gateways and events [1].
- **Human task management:** worklists/inboxes, assignment and claiming, escalation, and task lifecycle management for user work [1].
- **Integration mechanisms:** invoking external systems (e.g., REST services, messaging, email) through connectors, delegates, or external worker patterns, depending on the platform (e.g. Camunda REST integration[1]).
- **Monitoring and analytics:** operational visibility into instance state, audit trails, KPIs, and logs that support supervision and continuous improvement [1].

[1] https://docs.camunda.org/get-started/archive/java-process-app/service-task/

These capabilities make process behavior explicit and observable, enabling organizations to manage execution, compliance, and improvement in a structured way rather than embedding process logic implicitly inside application code [1].

2.1.3 Executable processes vs. conceptual process models

Process models can be created with different intentions. A *conceptual* (descriptive) process model is primarily a communication artifact. It documents or explains how work is performed, often intentionally abstract, and leaves out technical bindings. In contrast, an *executable* process model is intended to be used by a process engine and must therefore be sufficiently precise to determine runtime behavior, including unambiguous routing and task execution semantics [3].

2.2 BPMN 2.0 for Automation

2.2.1 Core BPMN concepts

Business Process Model and Notation (BPMN) is an OMG standard that defines a graphical notation and an XML-based representation for specifying business processes [3]. BPMN provides a set of core flow elements for describing *how work proceeds* through a process:

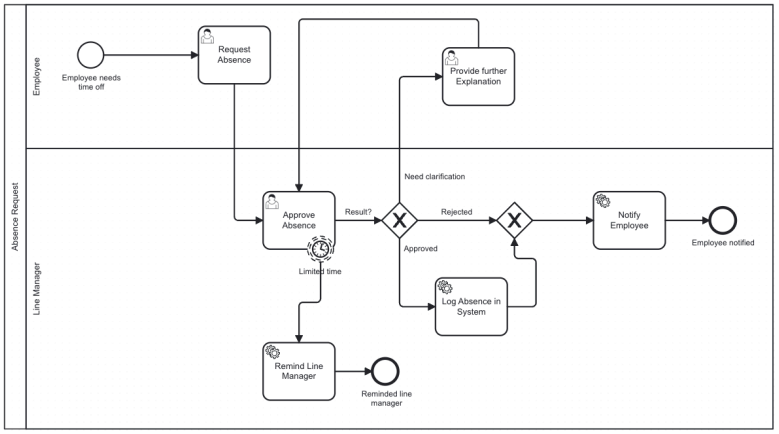


Figure 2.1: BPMN 2.0 diagram example.

- **Tasks:** units of work performed in the process. Tasks can represent human work (e.g., *User Task*) or automated work (e.g., *Service Task*), depending on the nature of the activity [3].
- **Events:** occurrences that affect the process flow. Common event types include *Start Events* (instantiate a process), *Intermediate Events* (e.g., timers or messages that model waiting/communication), and *End Events* (terminate a path of execution) [3].
- **Gateways:** control-flow routing constructs used to split and merge paths (e.g., exclusive choice, parallel split/join) [3].
- **Sequence Flow:** directed edges that define the execution order between flow nodes (activities, events, gateways) [3].
- **Subprocesses:** hierarchical structuring mechanisms that encapsulate a portion of the process, supporting decomposition and reuse of process fragments [3].

BPMN also includes additional constructs often used to improve understanding and collaboration, such as **Pools and Lanes** for representing participants and responsibility boundaries, and **Message Flows** for communication across participants [3].

2.3 The BPMN Execution Gap

2.3.1 What BPMN standardizes (control flow)

BPMN standardizes core control-flow concepts such as activities, sequence flows, gateways, events, etc. [3]. This enables BPMN to represent what happens next in a process in a platform-independent way.

However, BPMN does not fully standardize how execution concerns, such as UI rendering, connector frameworks, organizational identity resolution, etc., should be handled. Consequently, BPMN alone is rarely sufficient to deploy and run a real-world process end-to-end across different engines without extra configuration.

2.3.2 What execution requires beyond just BPMN diagram

Some of the common examples are:

- **Forms/user interaction:** a user task requires a concrete UI (embedded/external form or schema-driven form) and mappings between UI fields and process variables (e.g., Camunda Forms[2]).
- **Organizational models:** executable tasks must be bound to users, groups, or roles. Although assignments are possible with standard BPMN, some engines require organization artifacts in the deployment (e.g., Bonita [3]).
- **Service integrations:** service tasks require an implementation mechanism (e.g., delegate code, external service calls, etc.) and configuration of inputs/outputs and error handling (e.g., Camunda Service Tasks [4]).
- **Complex decision models:** The use of gateways might not be enough in some cases. Solutions for more complex decision logic might be needed (e.g., DMN).

These and other elements are essential for execution, but are not standardized or part of BPMN itself. This motivates the need for vendor-specific extensions or additional artifacts needed for deployment.

2.3.3 Vendor extensions and coupling: why portability breaks

BPMN includes explicit extensibility mechanisms (e.g., extension elements and attributes) that allow tools and engines to attach non-standard metadata to BPMN models [3]. This is practical for execution, but it introduces coupling: once a model contains vendor-specific form references, assignment rules, connector configurations, or decision bindings, it becomes tied to the conventions of that specific engine. Consequently, migration across engines becomes more than moving BPMN XML. It requires translating engine-specific execution metadata and associated deployment artifacts (forms, connector configurations, role mappings, decision references, etc.) into the target platform's configuration model. As mentioned earlier, this portability limitation is a primary motivation for the vendor-neutral pipeline proposed in this thesis.

[2] https://docs.camunda.org/get-started/archive/java-process-app/forms/
[3] https://documentation.bonitasoft.com/bonita/latest/identity/organization-overview
[4] https://docs.camunda.org/get-started/archive/java-process-app/service-task/

2.4 Engine Configuration Differences

Although pretty much every BPMS uses BPMN, they differ substantially in how they encode and deploy execution details. Control-flow constructs are generally comparable across engines, but differences become pronounced for configuration beyond BPMN.

For example, BPMS platforms differ in how user tasks connect to forms. Camunda 7 allows building entire forms directly in the BPMN's XML using embedded form definitions[5], while Bonita attaches only form references to task elements and manages form definitions separately[6]. Task assignment can be expressed as assignee/candidate semantics, role/group expressions, or explicit actor mapping artifacts. Bonita[7], for instance, treats actor mapping as an explicit configuration concern separate from the BPMN control-flow model. The number of such differences is countless.

2.5 State of the Art

There is a lot of research on business process automation, but much of it targets *only parts* of the overall pipeline from requirements to deployment. The reviewed prior work can be grouped as follows:

2.5.1 Model-driven generation from BPMN to executable artifacts

Some prior work uses Model-Driven Engineering (MDE) transformations to derive executable artifacts from process models. Zafar et al. present a framework that transforms BPMN into SoaML and then generates executable Web services and code [13]. This shows how an MDE pipeline can reduce manual effort by moving from a high-level process model to implementation artifacts via intermediate models. Yet, the approach is tied to a particular technology stack (e.g., service implementation generation) so it does not address the multi-engine portability challenge in which different BPMS products require different sets of execution metadata and deployment artifacts.

In an earlier work, Ouyang et al. study the systematic translation from BPMN process models to BPEL Web services and discuss structural challenges when translating between flow-based languages that support parallelism [14]. These mappings demonstrate that executable representations can be generated from BPMN, but they assume a specific execution target and environment. Modern BPMS engines, however, typically execute BPMN directly while depending on additional, engine-specific configuration that is outside the BPMN standard.

Mos and Jacquin propose a platform-independent deployment mechanism where BPMN tasks reference *abstract services* resolved later via mapping composites [15]. This reduces the need to embed platform-specific service bindings into BPMN.

Overall, the works above focus primarily on service tasks rather than the broader set of execution artifacts (e.g., user task UIs, organizational mappings, decision bindings).

2.5.2 Deriving user interaction artifacts from process models

Since many business processes are centered around human interaction, several works investigate deriving user interaction artifacts from process models. Díaz et al. propose augmentations and transformation rules that support automatic generation of user interfaces from BPMN and empirically evaluate the effect of such automation on development

[5] https://docs.camunda.org/manual/7.20/user-guide/task-forms/
[6] https://documentation.bonitasoft.com/bonita/latest/process/forms
[7] https://documentation.bonitasoft.com/bonita/latest/process/actors

effort and outcomes [16]. Cruz et al. similarly propose a model-driven approach that derives integrated software design models (use cases, domain models, and user interface models) from business process models [17].

These 2 works show that UI-related artifacts can be derived from process models, but they treat UI artifact generation as a separate task and do not aim to produce end-to-end, multi-engine BPMS deployment packages that include connectors, roles/actors, and other engine-specific configuration.

2.5.3 Data-aware and artifact-centric approaches

Another stream of research focuses on integrating BPMN control-flow with rich data semantics to obtain precise and verifiable specifications. Artifact-centric approaches such as BAUML, model business processes around evolving business artifacts using UML and OCL. This emphasizes artifact lifecycles, data structures, and the semantics of tasks [10]. De Giacomo et al. propose a method to link BPMN processes with UML class models by using OCL operation contracts to specify task effects, with the goal of achieving executable models grounded in a precise data semantics [5]. Follow-up work studies verification of data-object manipulation against combined process and data models [6]. More recently, König et al. discuss extending BPMN data objects with variable-like concepts to better align process execution with data handling needs [18].

These approaches contribute to data-process alignment and consistency checking. Their scope is, however, limited to these dimensions.

2.5.4 Traceability and synchronization across modeling views

Bouzidi et al. propose an integrated intermediate model that relates BPMN and UML use case models, and define transformation rules that can derive trace links and support synchronization after changes [19]. Such approache is relevant when different stakeholders maintain different views of the same process and alignment must be preserved over time.

2.6 Thesis Positioning

The contributions described above generally address a subset of concerns in isolation. Some focus on service task generation, others on UI derivation, data semantics, or traceability. The remaining gap, and the focus of this thesis, is an integrated approach that starts from structured requirements, maintains consistency across complementary models, and automatically generates the full set of deployable artifacts needed to execute the same process on multiple BPMS engines without embedding vendor-specific configuration directly into the BPMN model. This thesis is positioned as an *integrated vendor-neutral modeling and generation pipeline* for business process automation.

3 Approach

This chapter presents the overall approach/methodology used to answer the research questions. It focuses on the research methodology and overall artifact architecture rather than concrete implementation details.

3.1 DSR

The thesis follows a design-oriented research approach inspired by Design Science Research (DSR) [20], where knowledge is produced through building and evaluating a purposeful artifact. The produced artifact is a vendor-neutral modeling, validation, and generation pipeline for business process automation. The artifact is built in an iterative manner, through a series of Build–Evaluate cycles. At the end, an evaluation of the artifact implementation is done.

3.2 High level overview

The work towards developing the artifact aligns directly with the three research subquestions. They have to be answered in order.

3.2.1 Stage 1: Platform-independent modeling strategy (RQ1.1)

1. **BPMS comparison.** A prerequisite for platform-independent execution is understanding what contemporary BPMS engines require in practice to deploy and run processes, what features are supported, and how they are implemented (e.g. in case of forms, decision tables, etc.). This results in a comparison table between different BPMS engines.
2. **Identifying information requirements.** The comparison table is then reviewed to reveal what main information types commonly appear to be supported across the different BPMS engines. The result is a list of information requirements. Some examples of information requirements are: user forms, business data, email templates, etc.
3. **Modeling language selection.** For each information requirement, a suitable modeling language or representation is selected. The selection is guided by criteria such as expressiveness, standardization, and tool support.
4. **Modeling guidelines and conventions.** For each information category, concrete modeling guidelines are defined (naming conventions, required attributes, reference identifiers, and any constraints needed later by validation and generation). These guidelines ensure that models are sufficiently precise and machine-processable.

The output of Stage 1 is a set of models that jointly capture all execution-relevant information required for business process automation.

3.2.2 Stage 2: Ensuring consistency between models (RQ1.2)

Because execution information is distributed across multiple models, before generating engine-specific artifacts, it is necessary to ensure that the models are consistent with each other. This stage introduces validation dimensions to ensure that.

Key points of the stage are:

- **Review of linking rules.** Linking rules between models are already introduced in the last step of Stage 1. This step is an overview of the methods used.

- **Syntax and semantic validity of each model by itself.** This is an overview of the steps and methods used to validate the models independently.
- **Semantic validity of the models together.** This is an overview of steps and methods used to validate consistency between pairs, triples, etc. of models.

The output of Stage 2 is a platform-independent model set that is not only expressive but also *internally consistent* according to defined cross-model constraints.

3.2.3 Stage 3: Engine-specific artifact generation (RQ1.3)

Given a consistent platform-independent model set, there are still a few steps to go before the generation of engine-specific artifacts. These generally involve answering the questions about how to parse models, whether intermediary representations are needed, how to resolve references (linking points between models), how to enrich the models with computed values, etc.

This is followed by the possible approaches to generating BPMS artifacts.

3.2.4 Stage 4: Implementation

In order to validate the proposed artifact, it is of course necessary to implement it. The implementation will be done in Java, and will be able to generate deployment artifacts for 2 different BPMS engines.

3.2.5 Relation to the thesis structure

The next three chapters correspond to the three research sub-questions and describe each stage in depth: Chapter 4 details the information requirements and modeling strategy (RQ1.1), Chapter 5 focuses on the consistency and validation (RQ1.2), and Chapter 6 puts focus on the generation algorithm (RQ1.3). Chapter 7 describes a real implementation of the artifact.

4 Platform-independent modeling strategy

This chapter aims to answer the first research sub-question (RQ1.1) (Stage 1 in Chapter 3). The output is a coherent set of models that jointly capture execution-relevant information required for automation. Artifact-wise, the chapter provides the list of information requirements, the modeling language selection, and the modeling guidelines and conventions.

4.1 BPMS comparison

4.1.1 Motivation and scope

A prerequisite for platform-independent execution is understanding what contemporary BPMS engines require in practice to deploy and run processes. Therefore, the first step is a structured comparison of representative platforms. The aim is not to rank vendors, but to compare them, analyze how they deal with execution-relevant information.

The comparison includes both open-source and commercial suites. The primary inclusion criteria are (i) evidence of real-world adoption, and (ii) practical accessibility for hands-on inspection (download, trial, or online modeler) of core functionality.

4.1.2 Market scan and selection of representative platforms

To obtain an initial overview of relevant platforms, Gartner Peer Insights' category for *Business Process Management Platforms* [21] was used. This provided a good starting point to identify widely used products and alternatives that are actively maintained and discussed by practitioners.

From this long list, six platforms that together cover different styles of BPMS usage, from developer-centric, embeddable workflow engines to suites with strong low-code tooling, were selected. Table 4.1 summarizes the selection rationale at a high level. The detailed comparison is captured in the catalogue described later in this section.

Table 4.1: Selected BPMS platforms and their positioning.

Platform	Positioning
Camunda 7	Developer-centric BPMN execution engine
Flowable	Developer-centric BPMN execution engine
jBPM / RHPAM	Developer-centric engine with enterprise suite distribution
Bonita	Low-code, business-user-oriented BPM suite
Bizagi	Low-code, business-user-oriented BPM suite
Oracle BPM	Enterprise-grade BPM suite

4.1.3 Brief characteristics of the selected platforms

Camunda Platform 7. Camunda Platform 7[1] provides a BPMN 2.0 process engine running on the JVM and is designed to be embedded into Java applications and common

[1] https://docs.camunda.org/manual/7.24/

runtime containers, with strong integration into the Spring ecosystem. Historically, Camunda's engine originates from a fork of Activiti (2013), after which Camunda maintained an independent open-source engine codebase [22]. Camunda 7 is a representative baseline for a developer-first approach to process automation.

Flowable. Flowable[2] is a lightweight Java process engine that deploys and executes BPMN 2.0 process definitions. It emerged as a fork created by core Activiti developers [23], making it closely related to Camunda in terms of historical ancestry. Including Flowable allows for an interesting comparison where many BPMN constructs overlap, while extension mechanisms and modelling toolchains differ.

jBPM / Red Hat Process Automation Manager (RHPAM). jBPM[3] is an open-source BPM suite centered around a Java workflow engine capable of executing processes expressed in BPMN 2.0 and designed to run embedded or as a service. In enterprise contexts, Red Hat Process Automation Manager[4] builds on jBPM. RHBAM also introduces a rule-engine integration angle absent from the others (Drools coupling) and represents the Red Hat enterprise deployment model. It is useful for examining how BPMN execution interacts with declarative business rules and how suite packaging affects the boundary between process and decision logic environments.

Bonita. Bonita[5] provides BPMN-based studio-style process modeling and a runtime environment. It is a representative of the "artifact-heavy" modeling style, where there are many artifacts besides BPMN, such as an organization model, business data model, etc. Including Bonita helps expose questions about portability.

Bizagi. Bizagi[6] represents the business-user end of the spectrum, where execution semantics are configured through platform UI rather than code. Useful for identifying which BPMN extension points are absorbed into platform tooling and which are left to developers.

Oracle BPM Suite. Oracle BPM Suite[7] is Representative of large-enterprise stack coupling, where BPMN processes are first-class citizens within a broader middleware ecosystem (SOA, BPEL, Oracle DB integration). Useful for examining how enterprise infrastructure assumptions shape what the BPMN engine is expected to handle natively versus delegate outward.

4.1.4 Comparison table

A key observation across the selected platforms is that all of them are centered around BPMN as the core control-flow representation. This shared foundation makes it possible to also structure the comparison around the BPMN 2.0 standard element categories. The table can be found in Appendix B.

4.2 Identifying information requirements

The BPMS comparison table in Appendix B shows that engines converge on BPMN for the *orchestration and control-flow skeleton*, but diverge in how they model and bind ex-

[2] https://www.flowable.com/
[3] https://docs.jbpm.org/latest/jbpm-docs/html_single/
[4] https://access.redhat.com/products/red-hat-process-automation-manager/
[5] https://documentation.bonitasoft.com/bonita/latest/
[6] https://help.bizagi.com/platform/en/index.html?get_started.htm
[7] https://www.oracle.com/middleware/technologies/bpm.html

ecution details. In practice, these are realized through platform-specific properties, XML extensions, external configuration artifacts, or code hooks. For instance, even BPMN data-related annotations are frequently treated as documentation rather than runtime semantics in common engines [24]. The BPMN model alone is therefore insufficient for automation portability.

In this thesis, *information requirements* is the set of execution-relevant information items that must be recorded in order to deploy and run a process, in addition to the BPMN control-flow model. These can later be mapped to modeling languages and ultimately generated into engine-specific execution artifacts.

4.2.1 Scope and selection of requirements

In principle, the space of information requirements can become very large if the aim is to cover every vendor feature (e.g., advanced monitoring, proprietary worklists, custom transaction semantics, domain-specific connectors, etc.). Covering everything is beyond the scope of this thesis. Instead, the focus is on a set of core requirements that repeatedly appear across the selected BPMS (analyzed in Appendix B) so are essential for end-to-end automation in typical process scenarios. At the same time, these information requirements should be modelled expressive enough so that they can be later mapped back to heterogenious BPMS.

The selected requirement categories are summarized in Table 4.2 and explained below.

Table 4.2: Core information requirement categories addressed in this thesis.

Category	Brief description
Business data	Domain objects used by the process, attributes and their types (e.g., `Car`, `Invoice`, `LeaveRequest`).
Process variables	Instance variables (e.g. `car1` of type `Car`) that can be accessed and modified during process execution
Data contracts	Activity-level preconditions (required variables) and postconditions (produced variables). E.g., a task that requires a `Car` object could be `DriveToGarage`
Business data states	Expected states of data before/after tasks (e.g. which fields are present/null). States are identified before runtime. E.g. a `Car` can be in the state `new`, `damaged`, or `repaired`
User forms	Forms that need to be filled out by a user. E.g., a form for a `Car` repair request might ask for the car's make, model, and mileage.
Organization	Roles, groups, participants, and associations between them. E.g., `John` is a `CarMechanic` and a member of the `Garage` team
Expressions	Expressions in some formal language that can be used in e.g., conditional statements (e.g,. checks if `car1` is in the state `damaged` to determine if a specific flow should be followed)
Decision logic	Refers to complex decision logic (e.g., determining the cost of `car1` repair based on the car's make, model, mileage, insurance coverage, etc.)
Emails	Typical data needed for sending emails (e.g., subject, body, additional parameters). An email might be sent e.g., when `car1` is repaired to inform the customer about the repair status.
REST API calls	Commonly used way to integrate with external systems (e.g., REST API calls). An external system might be e.g., a CRM system to update the customer's car repair status.

4.2.2 What needs to be captured and how it is realized in the BPMSs

Business data. The platform-independent model must capture the domain objects used by the process, together with their attributes (e.g., `Car`, `Invoice`, `LeaveRequest`). Each attribute has an explicit type (e.g., `mileage: String`). Domain objects may be complex (e.g., a `Car` contains an `Engine` attribute) or include collections (e.g., a `Car` has `List<Wheel> wheels`).

Different BPMS engines impose different assumptions regarding variable typing and persistence. Bonita, for example, provides a strict native Business Data Model (BDM)[8] that must be defined and deployed for each process. Camunda, in contrast, does not prescribe a formal domain data model and leaves type enforcement and persistence structure to the developer.

For compatibility across heterogeneous engines, the platform-independent model set must therefore record business data types explicitly and assign stable identifiers that can be referenced consistently from other models.

[8] https://documentation.bonitasoft.com/bonita/latest/data/define-and-deploy-the-bdm

From a business process perspective, explicit data modeling is fundamental to ensuring consistency and validation [25].

Process variables. The platform-independent model must capture the process variables that flow through the process. These may be of primitive types or complex objects (e.g., `car1` of type `Car`). In different BPMS, they are either left entirely to the developer (e.g., in Flowable) or fully controlled and statically typed (e.g., in Bonita).

For compatibility across heterogeneous engines, the platform-independent model must record variable names and, where applicable, bind them to business data types defined in the business data model.

Data contracts. The platform-independent model should capture what goes in and out of an activity: the process variables (including their types and states) that serve as inputs and outputs. Multiple variables, as well as collections, can flow in and out of an activity. In different BPMS, these are typically realized as input/output mappings, validation rules, or scripts. These contracts enable consistency validation (e.g., missing required data before a task), generation of user forms that request the right fields, and generation of integration mappings (e.g., REST call templates).

Business data states. Data state refers to a specific combination of attribute values in a process variable. For example, for a `Car` variable: certain fields may be present or null (e.g., `mileage` is null when the car is new), a field may have a specific value (e.g., `status` = “damaged”), or a field may fall within a certain range (e.g., `mileage` between 0 and 1000 for a new car). As a minimum, the state should represent which fields are null and which are not null. The possible states are identified before runtime.

User forms (human task UI). For human tasks, automation requires a UI artifact (form) and a binding from the BPMN user task to that form. Forms must capture user input and validate it against required fields and additional rules (e.g., variable type, value range). While BPMN defines user tasks, form definitions and form references are commonly realized through vendor form references or embedded forms (vendor-specific properties or extension elements) [26, 27].

Human-task interfaces (forms) often appear in deployments and are key to execution reliability in practice [28].

Organization representation. The platform-independent model must capture actors, roles, groups, and associations between them. Regarding bindings, BPMN lanes should be assignable, and then these assignments are propagated to tasks unless overridden. Managing user identity information (e.g., email addresses, phone numbers) is beyond the scope of this thesis. Different BPMS differ in how rich organizational representation is. Oracle BPM and Bonita[9], for example, support more sophisticated organizational models. In terms of assignments, different BPMS implement this through actor mapping, work allocation rules, and identity provider binding.

Expressions. The platform-independent model must support expressions with at least limited expressivity: enough for conditional sequence flows, conditional events, assignments, and similar constructs. Expressions should be able to reference process vari-

[9] https://documentation.bonitasoft.com/bonita/latest/process/actors

ables during runtime and, where relevant, variables that engines introduce implicitly (e.g., `processInstanceId` for the current process instance). BPMN allows formal expressions syntactically, but the choice of expression language (e.g., UEL, FEEL, or other) and evaluation context differ between platforms. Different BPMS typically embed expressions in BPMN properties [29, 30].

Decision logic. The platform-independent model must support both complex decision logic and the use of expressions for this. Many BPMS externalize business rules from control-flow via DMN decision tables and business rule tasks. The OMG standard for this is DMN (Decision Model and Notation). It defines a portable representation for decisions and decision tables [31]. In Camunda, business rule tasks reference deployed DMN decisions (e.g., by key), and the engine evaluates the decision table and uses the result in the process. Other engines, e.g., Red Hat Process Automation Manager, also support Drools rules for decision logic.

Decision logic separated from control-flow is widely adopted as it enhances modularity and maintainability of processes [32].

Emails. The platform-independent model must support email templates whose subject and body can use expressions and dynamic content based on passed parameters. Emails should conform to the RFC 5322 format. Existing BPMS typically provide mail connectors, extensions, or listeners and support subject/body templates with parameter substitution. This is not standardized in BPMN and is often implemented via vendor tooling or libraries [33].

Email interactions frequently correspond to executable workflow activities, justifying their native support in many BPM platforms [34].

REST API calls. The platform-independent model must support REST calls that can use expressions and reference process variables, and that conform to the HTTP protocol (method, URL, headers, and body) for making REST calls. Different BPMS realize REST integration via connectors, service tasks, or code delegates, but invocation configuration (method, URL, authentication, payload mapping) is platform-specific (e.g. Bonita[10]).

Similary to Email tasks, REST integration have a huge widespread in modern BPMS engines [35].

Optional: Pre-runtime (global) variables. Pre-runtime or global variables are less of an information requirement in the usual sense than a configuration mechanism. It can be necessary to supply values (e.g., SMTP server settings) to the process before it is started. Other information requirements may reference these global variables. The references are then resolved before runtime.

Not included: Scripts. Although all reviewed BPMS support scripts, they are not included in the information requirements catalogue. This is because each uses a different programming language for that purpose. Choosing a less expressive language would be an oversimplification and would not capture the full complexity of scripted logic. Choosing a highly expressive language would make it impossible to parse into the language(s) supported by each BPMS.

[10] https://documentation.bonitasoft.com/bonita/latest/process/rest-connector

4.2.3 Output of this step

The output of this section is a consolidated information requirements list: a finite set of execution-relevant categories that the platform-independent model set must represent. This catalogue directly informs the next step (modeling language selection).

4.3 Modeling language selection

The next step is to select a representation for each requirement category. The goal is not to introduce yet another proprietary meta-format, but to assemble a platform-independent model set from available modeling languages/formats.

4.3.1 Selection criteria

The selection is guided by the following criteria:

1. **Expressiveness:** the language/format must be able to capture the information requirement precisely enough so that the minimum described in the previous section can be realized.
2. **Standardization and longevity:** whenever possible, preference is given to established standards maintained by recognized standardization bodies (e.g., OMG, IETF, W3C, The Open Group) to reduce vendor lock-in and to maximize future-proofing.
3. **Open and accessible tooling:** the language should have mature, freely available tooling (editors, validators, parsers) to ensure that model authoring is feasible without proprietary dependencies.
4. **Transformation friendliness:** the language should have well-defined syntax/semantics and stable serialization formats so models can be parsed, validated, and transformed reproducibly.
5. **Familiarity and stakeholder usability:** because models are authored and reviewed by humans, visual notations are preferred when they provide an ergonomic representation of the concept being modeled. Visual tools improve stakeholders' ability to comprehend complex workflows by leveraging human spatial reasoning [36]
6. **Reproducibility:** models and auxiliary artifacts should be stored as versionable text files such that generation results can be reproduced from the same repository state.

Table 4.3 summarizes the modelling choices made for each information requirement. The remainder of this section justifies the main choices and discusses key alternatives.

4.3.2 Information requirement 0: the process model

The primary (essential) information requirement is the process model itself. BPMN is selected as the platform-independent notation for this information requirement because it has become the de facto standard for business process diagrams and is widely supported across contemporary BPMS products [37, 3]. This also aligns with the comparative findings of Section 4.1, where all representative platforms either natively support BPMN or provide BPMN interchange as an import/export format (e.g., Bonita).

It is important to mention, though, that BPMN is used here primarily as a platform-independent source of truth for control-flow. Capturing references to many of the other information requirements models is offloaded to an additional complementary model - the config file. More information on it and why it was used can be found at the end of this section.

Another important detail is that not the full BPMN 2.0 specification will be used. BPMS comparison revealed that many engines have poor support for collaborations and chore-

Table 4.3: Overview of modeling language selection per information requirement.

Requirement category	Selected representation	Rationale (high-level)
Process control-flow	BPMN 2.0	De-facto standard; supported across BPMS; standardized interchange [37, 3].
Business data	UML class diagrams	Standardized; well-supported tooling; visual [38].
Process variables	BPMN data object references	Visual, close to process context; reuses BPMN [3, 18].
Data contracts (in/out)	BPMN data object references	Visual, close to process context; Standard BPMN practice for showing produced/required data [3].
Data states	BPMN data object references + JSON file following JSON Schema	Visual, close to process context; reuses BPMN [3, 18].
Forms	JSON Schema files	Broad tooling; supports form generation [39, 40].
Organization	ArchiMate	Standard EA language; direct constructs for actors, roles, groups; visual [41].
Expressions	Unified Expression Language (UEL)	Deliberately limited; [42, 29].
Decision logic	DMN	OMG standard; visual [43].
Emails	FreeMarker (FTL) + JSON file following JSON Schema (RFC 5322-aligned fields)	Mature templating for text (FTL); structured metadata (JSON Schema); [44, 45].
REST invocation	JSON file following JSON Schema (OpenAPI inspired)	Standard-inspired [46].

ographies (e.g., RedHatPAM, OracleBPM). Therefore, the thesis deliberately targets *private executable* business processes only [3]. Concretely, the platform-independent process model must be a single-participant executable process (one pool). Call activities are excluded, too.

4.3.3 Business data: UML class diagrams

UML [38] class diagrams have been selected for business data representation. First of all, they allow capturing all the required data for this information requirement (domain objects as classes, attributes as fields, and relationships as associations). Secondly, it's a well-established standard in the field of software engineering and is supported by a broad ecosystem of tooling. Additionally, UML can be represented programmatically if the right tooling is used (e.g., PlantUML). Lastly, it is a visual notation, which is preferred when it provides an ergonomic representation of the concept being modeled.

Entity-relationship (ER) modeling is a strong alternative. It was not selected, though, as attribute types cannot be captured in the ER model. Using ER would require an additional artifact to capture the attribute types, which is not necessary with UML. Another strong candidate is capturing business data as a JSON file following an OPENAPI-inspired JSON Schema. Although it allows capturing all the required information, it is not a visual notation. Other modeling languages considered (e.g., ArchiMate) are intentionally higher-level and do not allow enumerating domain object fields. Standard BPMN data objects are excluded, too, as they are not expressive enough [3].

4.3.4 Process variables, data contracts, and data state via BPMN data objects

For several information requirements, there is no single universally adopted standard language that all BPMS engines and stakeholders use in practice. This is particularly the case for: process variables, data contracts, and possible data states.

A purely textual configuration file could define process variables and their types, but such a solution is not user-friendly and disconnects data usage from the process context. Instead, this work uses a construct that is already part of BPMN: data objects (more precisely, data object references connected via data associations). Furthermore, BPMN data objects are by design intended to indicate data required or produced by activities [3]. By adopting clear naming conventions, data object references can encode additional information:

- **Process variables:** a variable identifier and its business type, e.g., `car1: Car`.
- **Data contracts:** input/output variables connected to tasks via BPMN data associations.
- **Data state:** a state tag appended to the variable/type, e.g., `req1: AbsenceRequest [submitted]`.

This approach has three advantages. First, it remains visual and close to the process model. Business users can see which task reads or produces which data. Second, it is transformation-friendly. Since data object references are explicit BPMN elements, the diagram can be scanned and all variables and their occurrences can be collected and presented as a synthesized list without requiring authors to maintain a separate register. Third, the approach is aligned with existing research that extends BPMN data objects with variable identifiers and discusses their semantics [18]. Similarly, the idea of reasoning about (and verifying) data object manipulation and states across tasks has been studied in the literature [6]. Using naming conventions for data objects combines and extends the ideas of these two research papers.

The concrete formatting conventions for data object reference labels (variable name, type, optional state, and further constraints) are specified in Section 4.4 to ensure that the representation is parseable.

4.3.5 Data states as a JSON file

While the state tag in BPMN (e.g., `[submitted]`) provides a compact visual notation, automation also requires a precise definition of what each state means. For example, a state may imply that certain fields must be present (non-null).

Therefore, it is necessary to introduce another notation format for this information requirement that would specify which states exist for a given business type, and for each state, which fields are required. For the scope of this thesis, this definition is limited to that. It is, of course, possible to extend the definition to also support more complex constraints, such as having different states in case of a certain field having a certain value or a certain range of values, etc.

The information requirement is modeled as a JSON file that has to follow a JSON Schema designed for this purpose. JSON is selected because it is standardized, widely supported across languages and toolchains, and well-suited for validation and parsing [47]. Compared to XML, JSON is less verbose for the same hierarchical content and avoids document-level complexity that is unnecessary [48]. YAML is a strong alternative to JSON. It is even less verbose and can be parsed to JSON. There is, however, no widely

supported 'YAML Schema' standard, and JSON schemas still have to be used. Therefore, to avoid additional parsing complexity, YAML is not used, however it could be an option [49].

4.3.6 Forms: JSON Schema file

User tasks require a mechanism to capture input data (and potentially display existing data) during execution. The modeling approach selected here is intentionally scoped. It is not meant to be a UI design tool, but to capture the data needed for automation, including basic structural constraints (types, required fields, required ranges for numeric fields, etc.)

JSON Schema files were selected as the preferred notation format for forms. Although this is a text format, there is a wide range of tools available that allow designing the schema visually[11]. JSON Schema is standardized (draft 2020-12)[12] and supported by a broad ecosystem of tooling, including validators and libraries that can automatically generate forms from schemas (e.g., React-based generators) [40].

Two alternative approaches were considered. First, XForms is a W3C standard for declarative forms, but it did not achieve mainstream adoption in modern web development, and mainstream browsers have historically not provided native client-side support, requiring plugins or transformations [50, 51]. Second, forms could be derived from other information requirement models (business data types and data states). While the second option is acceptable for basic presence/absence constraints for inputs, it is insufficient for richer form requirements such as string length bounds, regex patterns, numeric ranges, etc. JSON Schema, therefore, provides the necessary expressiveness needed for this information requirement.

4.3.7 Organization: ArchiMate

Process execution requires knowledge about who performs human work. It is therefore necessary to represent the organizational concepts necessary to bind work to participants (e.g., users, groups, and roles).

ArchiMate is selected to represent organization because it is a standardized enterprise architecture language and provides constructs that map directly to the required concepts, e.g., *Business Actor* and *Business Role* [41]. Groups can be represented as *Business Collaboration*. The modeling language is also visual and is supported by a broad ecosystem of tooling (e.g., Archi, ArchiMate Editor).

One of the alternatives considered was UML, it however does not have pre-existing constructs for e.g., roles and groups. Another alternative was to use SCIM, a standard for identity provisioning [52]. It is, however, non-visual and more protocol-oriented. The third alternative was using RBAC (Role-Based Access Control), a standard for access control. It is, however, also a non-visual tool and does not support groups.

4.3.8 Expressions: Unified Expression Language (UEL)

Many execution semantics require the use of expressions (e.g., conditional sequence flows, variable mappings, etc.). To support portability, the expression language should be sufficiently expressive for common conditions and property access, but not so powerful that parsing to other languages becomes infeasible.

It was decided to use the Unified Expression Language (UEL), standardized in the Jakarta Expression Language specification, as the canonical expression language for the platform-

[11] https://json.ophir.dev/

[12] https://json-schema.org/draft/2020-12/schema

independent layer [42]. UEL is deliberately constrained compared to general-purpose scripting languages, yet widely used in Java-centric ecosystems.

One of the alternatives was Groovy. It is a powerful JVM scripting language and is attractive for flexibility, but its general-purpose nature makes portable translation difficult [53]. As an alternative, XPath may be considered. The BPMN 2.0 specification states that “BPMN designates XML Schema and XPath as its default data structure” [3]. It is standardized and expressive for navigating structured data, but it is primarily designed for XML/JSON navigation contexts and does not map uniformly to BPMS expression facilities [54]. FEEL, a standardized expression language for decision tables, was also considered. Initial review, however, showed that it will be challenging to parse it into other languages.

4.3.9 Decision logic: DMN

Decision logic is separated from control-flow using DMN, an OMG standard designed specifically to model decisions and decision requirements independently of process orchestration [43]. DMN supports decision tables, and this is the main focus for this information requirement. Although DMN’s standard expression engine is FEEL, UEL will be used instead, as discussed in the previous subsection. Tooling exists that allows working with DMN models visually (e.g. Camunda DMN Modeler).

Alternative approaches were considered. First, not using any extra modeling language for complex decision logic at all. This is, however, not a good idea, as it would require embedding the decision logic directly into the process model, which would likely make it hard to maintain and understand. Another alternative was scripting decision logic. However, this approach is not visual and thus less user-friendly.

4.3.10 Email: FreeMarker templates + JSON file following JSON Schema

The selected representation separates: (i) the email body template, and (ii) the email metadata (recipient, subject, cc/bcc, etc.).

For the body template, FreeMarker[13] (FTL) is selected. FreeMarker is a mature template language designed for text generation with explicit variable interpolation and control constructs.

Alternative template engines were considered. One of them, Velocity[14], could in principle be used as well. It is, however, less predictable for error handling and has less community support.

For the metadata, a JSON file following a specific JSON Schema is used. The schema is aligned with the conceptual structure of Internet email messages as standardized in RFC 5322 (e.g., header fields such as `From`, `To`, `Subject`, `Date`) [45].

4.3.11 REST invocation: OpenAPI-inspired schema

A JSON file following a specific JSON Schema is used. The schema is inspired by the OpenAPI Specification (OAS) [46]. OpenAPI provides a standardized, language-agnostic description format for HTTP APIs. The schema also specifies the minimal set of required fields for a REST invocation (following the HTTP standard [55]).

4.3.12 Extra: Config file and pre-runtime (global) variables

As mentioned at the beginning of this section, it was decided to offload some of the referencing between the main process model and other information requirement models to a complementary model. This model is a JSON configuration file that should contain these

[13] https://freemarker.apache.org/docs/index.html
[14] https://velocity.apache.org/engine/1.7/overview.html

references (e.g., between a user task and a JSON Schema file for the form). The file follows a specific JSON Schema designed for this purpose.

The reason for having the configuration file is to avoid embedding these references in the BPMN model's XML. That would be a less user-friendly approach. In addition to that, those references are something that is resolved before runtime, so it makes sense to have them in a separate file.

Using a separate file for the configuration is also a good idea as it allows adding additional information. This includes adding e.g., pre-runtime (global) variables that then can be referenced from other information requirements models.

4.4 Modeling guidelines for each information requirement

This section specifies modeling guidelines for the platform-independent model set introduced in sections 4.2 to 4.3.

4.4.1 Process control-flow (BPMN)

The control-flow of a process should be modeled in BPMN. In general, the modelling at this step should be done as usual, following the OMG BPMN 2.0 standard [3]. However, there are some extra guidelines that should be followed in this thesis.

G0 — BPMN Subset Only BPMN elements part of *private executable business processes* [3] can be used. This excludes collaborations and choreography elements. Call activities are excluded, too.

The BPMN process must contain only one pool. The pool should contain at least one lane (explicit).

The BPMN XML should use `http://www.omg.org/spec/BPMN/20100524/MODEL` namespace URI and `bpmn:` namespace prefix (these are the defaults for e.g. Camunda Modeller [15]).

G1 — Standard BPMN only. The BPMN diagram should not include any vendor-specific extensions (e.g., proprietary XML attributes and XML elements). In case of Timer events, only the ISO-8601 standard must be used ([3]).

G2 — Naming. Process, Pool (LaneSet), lanes, activities, events, and nodes must have a name. Names should not repeat. When there is only one lane in the pool, `_default` name should be used for the lane.

These rules are required to avoid ambiguous references during transformation.

G3 — Expressions. If runtime expressions occur in BPMN (e.g., gateway conditions), they **must** follow the expression rules defined in Section 4.4.7. The expressions themselves should be wrapped in `conditionExpression` element, e.g.

```
<bpmn:conditionExpression xsi:type="bpmn:tFormalExpression">
    ${~req1.status~ == "NEEDS_CLARIFICATION"}
</bpmn:conditionExpression>
```

Listing 4.1: Example of a formal expression in BPMN XML

[15] https://camunda.com/download/modeler/

G4 — User assignment. There should be no user assignments in the BPMN model using XML attributes such as `humanPerformer`, `substitutionGroups`, `etc.`. This information should be captured later, in the configuration file.

G5 — Tasks. User tasks should have an outgoing data object reference. The guidelines for data object references are defined in Section 4.4.3.

User tasks must be used when referencing forms. Service tasks must be used when referencing email and REST configuration files. Business rule tasks must be used when referencing DMN files. More information on references is in Section 4.4.11.

4.4.2 Business data (UML)

Business data is modeled using UML class diagrams [38]. Only a subset of the language is used. Irrelevant elements of the diagram are ignored in the following stages.

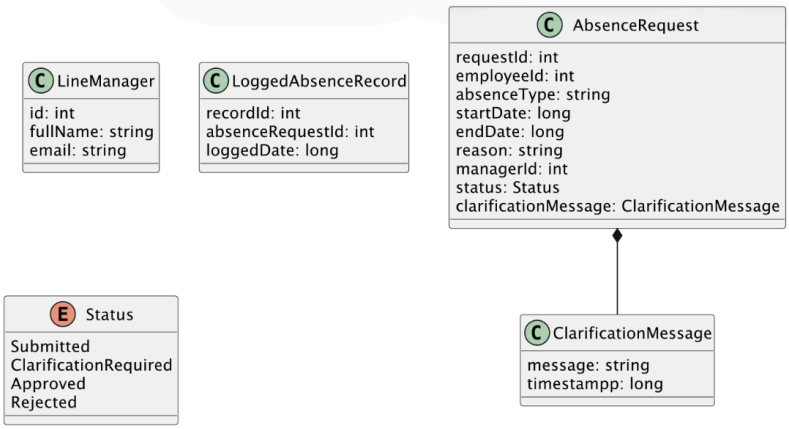


Figure 4.1: UML class diagram example.

G6 — Allowed constructs. Only the following UML class diagram constructs are permitted:

- **Classes with typed attributes.** Attributes must follow the format:

  `<attributeName> : <ElementType>`

  The element type denotes the type of the individual element. Multiplicity must not be written in the attribute declaration. Collection semantics are derived exclusively from the multiplicity specified on the corresponding association.

  In the case of composition or aggregation, the whole must declare the part explicitly as a typed attribute.

  Language-specific collection types (e.g., `List<T>`) are not allowed.

- **Enumerations**, for finite categorical domains. Enumeration literals must follow the format `<literalName>`. Enumeration values are treated as strings in subsequent transformation stages.

G7 — Relationships and multiplicities. Relationships are restricted to binary associations of the following types:

- **Composition.** The filled diamond must be placed at the composite (whole) end.

- **Aggregation.** The hollow diamond must be placed at the aggregate (whole) end.

Multiplicity must be written only on the part side of the association.

In this modeling subset, multiplicity is used solely to distinguish single-valued from multi-valued relationships:

- If the part side is annotated with $*$, the relationship is interpreted as multi-valued (collection).

- If no multiplicity is written on the part side, the relationship is interpreted as single-valued.

Lower-bound semantics (i.e., required vs. optional) are not expressed in UML. Requiredness constraints are defined separately in `states.json`. The UML diagram therefore defines structural shape (single vs. collection), while state-dependent requiredness is handled externally.

Multiplicity must not be written in the attribute declaration.

Composition and aggregation differ semantically as follows:

- **Composition** denotes strong ownership. A part may belong to at most one composite, and its lifecycle is considered dependent on the whole.

- **Aggregation** denotes weak ownership. The part may exist independently of the whole and may be referenced by multiple aggregates.

All relationships are unidirectional and navigable only from the whole to the part. Every composition or aggregation must have a corresponding typed attribute declared in the whole class.

The selected relationship subset aligns with widely used UML interpretations of composition and aggregation in practice [56].

The metamodel for business data modelling is available in Appendix A.1.

4.4.3 Process variables, data contracts, and data state (BPMN data object references)

Process variables (runtime instances), data contracts (input/output), and data state annotations must be captured using BPMN's data object references and connected via BPMN's data associations to BPMN activities. This applies only to process variables that are instances of business data types described in Section 4.4.2.

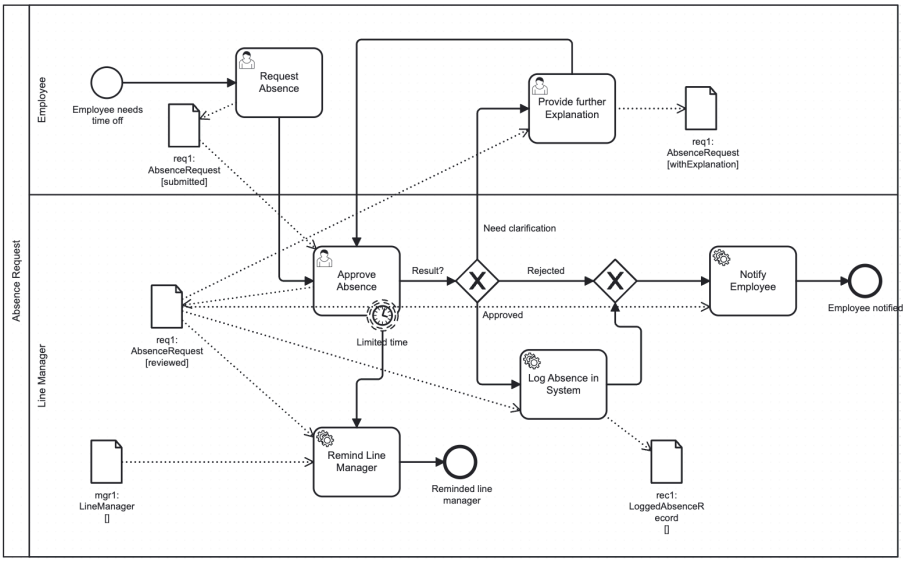


Figure 4.2: BPMN 2.0 diagram example with data object references.

G8 — Required label format. Each data object reference must follow the label format:

```
variableName: Type [state]
```

where:

- `variableName` is the process variable identifier.
- `Type` refers to a UML class/enum from Section 4.4.2.
- `[state]` is the state tag for the referenced type (see Section 4.4.4). If the type has only one (default state), use [] with nothing inside.

G9 — Multiple inputs/outputs. If an activity has multiple inputs/outputs, they should be represented as multiple data object references, each with a different label. Each reference should have an association to the activity.

G10 — Mandatory outputs for user tasks. For each user task that references a form, one outgoing data object reference must be present. This makes the produced/updated process variable explicit at the modeling level and enables later validation against form schemas and state constraints.

G11 — Optional inputs/outputs as runtime guards. For non-user tasks (e.g. service tasks, events), adding data object references as inputs/outputs is optional but recommended. When present, they serve as runtime guards. They document what data is required/produced and can be checked for consistency across the process.

G12 — Collections and multi-instance. If an activity manipulates a collection (e.g., multi-instance processing), the variable name should reflect collection semantics. This thesis adopts the convention of using a capitalized variable name (e.g., `Requests: AbsenceRequest [submitted]`) to signal a collection-valued variable. Additionally, the `collection` version of the data object BPMN element should be used.

For non-collections, the `variableName` should start with a lower case letter.

Reason for label convention . The BPMN 2.0 Specification contains special attributes for data objects to express multiplicity (e.g. `isCollection`) as well as the name of the data states. Prodviding this attributes changes the visual representation of the data object on the diagram. In reality, however, not many tools support this (e.g., Camunda Modeller, BPMN.io[16], JBoss Modeller[17]), so data objects' visual representation remains the same. This is the reason why the label convention is introduced.

The EBNF Grammar for label convention can be viewed in Appendix A.2.

4.4.4 Data states (JSON file following JSON Schema)

G13 — Writting the JSON file. The JSON file should follow the schema - see Appendix A.3.

In general, it should contain a collection of objects each describing possible states of business data types from Section 4.4.2.

- each object from collection contains:
  - `name`: UML class name,
  - `states`: list of states,
  - each state defines:
    - `name`: state identifier,
    - `requiredFields`: array of attribute names that must be non-null in that state.

In case when a business data type has only one state (the default state), the name of the only state should be `_default`. It still remains necessary to list the required fields for it.

```
{
  "classes": [
    {
      "name": "AbsenceRequest",
      "states": [
        {
          "name": "submitted",
          "requiredFields": ["requestId","employeeId","absenceType","startDate
              ","endDate"]
        },
        {
          "name": "reviewed",
          "requiredFields": ["requestId","employeeId","absenceType","startDate
              ","endDate","status","managerId"]
        },
        // ... other states ...
      ]
    },
    // ... other business data types ...
  ]
}
```

Listing 4.2: Example of `states.json` file.

[16] https://demo.bpmn.io/s/start
[17] https://tools.jboss.org/features/bpmn2.html

4.4.5 Forms (JSON Schemas)

G14 — One JSON schema per form. For each form, exactly one JSON Schema file (Draft 2020-12[18]) should be used.

G15 — Alignment with business data and states. JSON Schema properties should align with the UML class of the business data type (Section 4.4.2). Namely, the properties should be the subset of that UML class's attributes and the types should match. The `required` property in the JSON Schema should contain at least the fields listed in the states file for that specific state (see Section 4.4.4).

```
1  {
2    "$schema": "https://json-schema.org/draft/2020-12/schema",
3    "title": "AbsenceRequest",
4    "type": "object",
5    "properties": {
6      "requestId": {"type": "integer"},
7      "employeeId": {"type": "integer"},
8      "absenceType": {"type": "string"},
9      "startDate": {"type": "integer"},
10     "endDate": {"type": "integer"},
11     "reason": {"type": "string"},
12     "managerId": {"type": "integer"},
13     "status": {"type": "string", "enum": ["PENDING", "APPROVED", "REJECTED",
           "NEEDS_CLARIFICATION"]},
14     "clarificationMessage": {"$ref": "#/$defs/ClarificationMessage"}
15   },
16   "required": ["requestId", "employeeId", "absenceType", "startDate", "
       endDate"],
17   "$defs": {
18     "ClarificationMessage": {
19       "type": "object",
20       "properties": {
21         "message": {"type": "string"},
22         "timestampp": {"type": "integer", "description": "Unix timestamp"}
23       },
24       "required": []
25     }
26   }
27 }
```

Listing 4.3: Example of JSON Schema `request_absence_form.json` file.

The JSON Schema file above corresponds to the `Request Absence` user task in the BPMN diagram (Figure 4.2 and Figure 4.3). The `properties` property in the JSON Schema (Listing 4.3 lines 5-14), contains all the attributes of the UML class `AbsenceRequest` (Figure A.2). The state of the `Request Absence` is `submitted`. The `required` property of the JSON Schema (Listing 4.3 line 16) contains the attributes that are required for the `submitted` state.

[18] https://json-schema.org/draft/2020-12

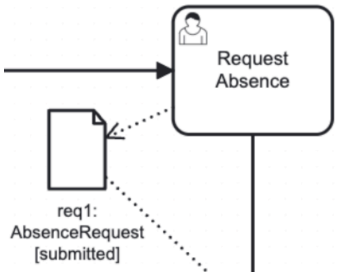


Figure 4.3: Zoomed in view of the data object reference coming out of the `Request Absence` user task in the BPMN diagram (Figure 4.2).

4.4.6 Organization (ArchiMate)

Supported ArchiMate elements are: Business Actor, Business Role, Business Collaboration, Assignment Relationship, Aggregation Relationship, Composition Relationship. Other elements can be used, but they will be ignored in the rest of the pipeline.

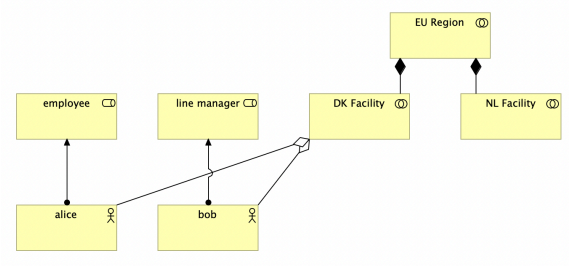


Figure 4.4: ArchiMate organization model example.

G16 — Required element types. The organization model must contain at least:

- one **Business Actor** (individual user),
- one **Business Role** (role in the organization),
- one **Business Collaboration** (group in the organization).

G17 — Structural constraints.

- Roles should not form hierarchies (roles are siblings only).
- Actors should not form hierarchies (actors are siblings only).
- Collaborations (groups) can form hierarchies (groups can contain sub-groups).

G18 — Relationships.

- All actor-to-role links must be modeled using **Assignment** relationships (arrow pointing from actor to role).
- All actor-to-group links must be modeled using **Aggregation** relationships (diamond pointing to the group).
- All group-to-group links must be modeled using **Composition** relationships (diamond pointing to the parent group).

Each actor must be assigned to at least one role and at least one group.

The metamodel for organization modelling is available in Appendix A.1.

4.4.7 Expressions (UEL)

G19 — UEL syntax. Expressions must be written using the standard `${ ... }` delimiter defined by the Jakarta Expression Language specification [42]. Within BPMN XML, they are placed in `bpmn:tFormalExpression` where applicable.

G20 — Variable markers. When referencing process variables inside UEL, this thesis introduces an additional marker for portability reasons. That is, process variables are wrapped in `~... ~`. This convention is not part of UEL. It is a generator hint enabling platform-specific rewriting of variable access paths (e.g., mapping `req1.status` to an engine-specific variable representation).

```
<!-- ... -->
<bpmn:conditionExpression xsi:type="bpmn:tFormalExpression">
  ${~req1.status~ == "REJECTED"}
</bpmn:conditionExpression>
<!-- ... -->
```

Listing 4.4: Example of BPMN condition expression using UEL with variable marker.

G21 — Pre-runtime resolution of global variables. Expressions can contain references to global variables. They have the following form:

`${...~globalVariables.X~...}`

The reference part can be resolved before deployment (pre-runtime). That part of the expression is simply replaced with the actual value of the global variable, located in the configuration file (more on that later inSection 4.4.11). If the expression contains no operators beyond the value lookup, it can be reduced to a plain string during generation. Listing 4.5 shows an example. The value of the `url` parameter is ultimately resolved to `https://example.org/post`. More on this in Section 4.4.11.

```
{
  // ... other properties ...
  "request": {
    "method": "POST",
    "url": "${~globalVariables.notificationServiceBaseUrl~}/post",
    // ... other properties ...
  }
}
```

Listing 4.5: Example of REST template file using UEL expression referencing global variables.

EBNF Grammar for use of markers is available in Appendix A.2.

4.4.8 Decision logic (DMN decision tables)

Each DMN file must contain exactly 1 decision table.

G22 — Inputs as expressions. UEL expressions can be used inside the decision table (Section 4.4.7). When referencing process variables that work as inputs for decision tables, they should be described as UEL expressions with extra markers (Section 4.4.7).

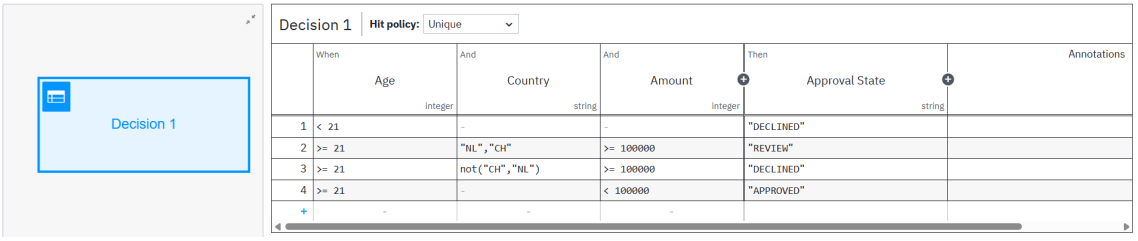


Figure 4.5: Example of a DMN model containing a decision table (`Decision 1`) for a business rule task called `Evaluate Loan Request`

```
<!-- ... -->
<decision id="loan_approval_dmn" name="Decision 1">
  <decisionTable id="DecisionTable_1pdtx17">
    <input id="Input_1" label="Age">
      <inputExpression id="InputExpression_1" typeRef="integer">
        <text>${~loa1.age~}</text>
      </inputExpression>
    </input>
    <input id="InputClause_02g1qz8" label="Country">
      <inputExpression id="LiteralExpression_1u1do00" typeRef="string">
        <text>${~loa1.country~}</text>
      </inputExpression>
    </input>
    <!-- ... -->
```

Listing 4.6: `Decision 1` decision table XML code

4.4.9 Emails (JSON file + FreeMarker template)

G23 - JSON file The JSON file should follow the JSON schema made for this purpose - see Appendix A.3. In general, mandatory data should be specified (e.g., recipients, subject, etc.) as well as parameters that are used in the FreeMarker template. UEL expressions can be used inside the JSON File only (see Section 4.4.7).

```
{
  "type": "email",
  "id": "remindLineManager",
  "headers": {
    "from": "no-reply@company.example",
    "to": "${~mgr1.email~}",
    "subject": "Pending Absence Request for ${~req1.employeeId~}"
  },
  "body": {
    "templateRef": "remind-line-manager.ftl",
    "parameters": {
      "managerName": "${~mgr1.fullName~}",
      "reviewLink": "${~globalVariables.appBaseUrl~}/requests/${~req1.
          requestId~}"
    }
  }
}
```

Listing 4.7: Example of a JSON file for describing an email (`remind_line_manager_email.json`)

G24 — Parameter completeness. Every template variable used by the FTL body must be supplied through `body.parameters` in the JSON file. There should be exactly 2 files per email (JSON and Template file).

```
Dear ${managerName},

You have a pending absence request that requires your attention.

Review link:
${reviewLink}
```

Listing 4.8: Example of an FTL file used as email template (`remind_line_manager_email.ftl`)

4.4.10 REST integration - JSON file)

G25 — JSON file guidelines. The JSON file should follow the JSON schema made for this purpose - see Appendix A.3. In general, the request method, URL, headers, and body should be specified. UEL expressions can be used (see Section 4.4.7).

```
{
  "type": "rest",
  "id": "notifyEmployee",
  "description": "Send final decision to employee.",
  "request": {
    "method": "POST",
    "url": "${~globalVariables.notificationServiceBaseUrl~}/post",
    "headers": {
      "Content-Type": "application/json"
    },
    "body": "{\"recipient\":\"${~req1.employeeId~}\",\"...\"}"
  }
}
```

Listing 4.9: Example of a JSON file for describing a REST call (`notify_employee_rest.json`)

4.4.11 Configuration file - JSON file

G26 — JSON file guidelines. The JSON file should follow the JSON schema made for this purpose - see Appendix A.3. In general, the configuration file is the place where the references between the main process model and other information requirements models are stored. Besides that, additional configuration can be stored here. An example of a configuration file is shown in Listing 4.10.

```
{
  "globalVariables": [
    { "name": "notificationServiceBaseUrl", "value": "https://httpbin.org"
        }, //...
  ],
  "processes": [
    {
      "name": "absenceRequest",
      "config": {
        "lanes": [
          {
            "name": "Employee",
            "assignee": ["${ref:organization.alice}"],
            "vendor_specific_attributes": {}
          }, //...
        ],
        "events": [ // ...],
        "tasks": [{
            "name": "Request Absence",
            "formRef": "${file:forms/request_absence_form.json}"
```

```
20            },{
21              "name": "Remind Line Manager",
22              "emailJsonRef": "${file:emails/remind_line_manager_email.json}",
23              "emailFtlRef": "${file:emails/remind_line_manager_email.ftl}"
24            },{
25              "name": "Provide further Explanation",
26              "formRef": "${file:forms/provide_further_explanation_form.json
                    }",
27              "vendor_specific_attributes": {
28                "camunda:test1": "someValue1",
29                "camunda:test2": "someValue2"
30              }
31            },{
32              "name": "Notify Employee",
33              "restCallRef": "${file:rests/notify_employee_rest.json}",
34              "vendor_specific_attributes": {}
35            },
36            // ...
37          ]
38        }
39      }
40    ],
41    "smtpConfig": {
42      "host": "smtp.gmail.com",
43      "port": 587,
44      // ...
45    }
46  }
```

Listing 4.10: Example of a JSON file for describing a configuration (`config.json`)

- **Global variables.** `globalVariables` defines name–value pairs that may be referenced in other model files (email configuration, REST configuration, BPMN configuration, DMN expressions) (e.g., lines 2–3 in Listing 4.10). Expressions are not permitted in `config.json` values.
- **Process configuration.** Each process must define a `name` and a `config` section. The configuration allows attaching metadata to BPMN elements by referencing them via their `name`. It is possible to configure lanes, events, and tasks (e.g., lines 11, 18, 21, 25, 32 in Listing 4.10).
- **Lane assignment.** For lanes, it is possible to configure either a single assignee or a list of assignees. Each assignee must reference an element from the organization model (e.g., actor, group, or role) using the structured format `${ref:organization.<id>}` (e.g., line 12 in Listing 4.10).
- **User tasks.** Tasks representing user tasks must include a form reference `${file:forms/...json}` (e.g., line 19 in Listing 4.10). User tasks must have at least one assignee resolved either directly or indirectly via the enclosing lane.
- **Email service tasks.** Service tasks used for sending emails must include both:
  - a JSON email definition reference (`emailJsonRef`), and
  - a corresponding FreeMarker template reference (`emailFtlRef`).

  (e.g., lines 22–23 in Listing 4.10). If at least one email task is present in the configuration, the `smtpConfig` section must also be specified (e.g., line 41 in Listing 4.10).

- **REST service tasks.** Service tasks used for sending REST calls must include a REST configuration reference `${file:rests/...json}` (e.g., line 26 in Listing 4.10). The referenced file defines the HTTP method, URL, headers, and optional payload.
- **Business rule tasks.** Business rule tasks must include:
  - a DMN file reference (`dmnRef`), and
  - the target variable in which the decision result is stored (`dmnResultVariable`).

  The result variable must use the marker-based expression syntax (e.g., `${~loa1.approvalState~}`).
- **Task typing constraint.** Each task configuration may represent at most one task type (user, service, or business rule). Multiple task types must not be combined in a single task configuration object.
- **Vendor-specific attributes.** Optional, engine-specific attributes may be configured for lanes, events, and tasks (e.g., lines 27–30 in Listing 4.10). These attributes are treated as opaque metadata and passed unchanged to the engine-specific generation stage.
- **Expression restriction.** Expressions are not permitted inside `config.json` except in explicitly defined reference fields (e.g., `formRef`, `emailJsonRef`, `restCallRef`) and in `dmnResultVariable`.

4.5 Chapter summary

This chapter introduced the platform-independent modeling strategy. A structured set of complementary models was introduced. The core artifact is a BPMN 2.0 process model, which is augmented by explicit definitions for business data, state models, forms, integration tasks (email and REST), decision models, organizational structures, and a central configuration file that binds these elements together. By maintaining all execution-relevant information in explicit, validated, platform-independent models, the framework will enable the automated generation of engine-specific deployment bundles. The next chapters will focus on the problem of validation and generation.

5 Ensuring consistency between models

This chapter answers RQ1.2: “How can synchronization and consistency between models be defined and maintained?” The input for this chapter is the platform-independent model set described in Chapter 4. While this separation of concerns increases clarity and portability, it introduces a fundamental challenge. Correctness is no longer determined by a single file, but by the combined interpretation of several interdependent artifacts.

In software and systems engineering, this problem is typically discussed under *inconsistency management* and *multi-view/model consistency*. Prior work emphasizes that inconsistencies are not rare corner cases but a natural consequence of evolving, multi-perspective specifications, and that systematic detection and feedback reduce late rework [9, 8]. A practical conclusion from this literature is that consistency checking should be performed early and continuously, ideally close to the moment a change is introduced [57]. In this thesis, the artifact realizes these ideas through an explicit validation pipeline. So the output of this chapter remains the same as the input, however, now also validated.

5.1 Linking strategy across models

This is a review of the linking strategies used in the context of this thesis. The guidelines for modeling were already presented in Chapter 4 and introduced different ways in which models for different information requirements reference each other. In general 2 different ways can be highlighted:

Config file as a linking hub. The main way to connect models is using the config file (e.g. Listing 4.10). The configuration file allows e.g., to reference a form file from a user task, a DMN file from a business rule task, or a REST call from a service task. Treating the config as an explicit link artifact makes dependencies visible and keeps the BPMN diagram from accumulating engine-specific details.

References between models. In many of the models, it is possible to include UEL expressions with markers. They can be used to reference process variables, or, e.g., global variables from the configuration file.

5.2 Consistency dimensions

To keep validation structured (so the rest of the chapter is not ad-hoc), the following categories of consistency are defined:

Syntax validity. Each artifact is well-formed at the *format* level: the file can be parsed (e.g., valid XML/JSON, parsable template syntax). Syntax checks do not require other artifacts. They answer “can this file be read and recognized as belonging to its language?”

Semantic validity (standalone). Each artifact conforms to its schema or metamodel when considered in isolation (e.g., BPMN structural validity, JSON Schema conformance, DMN metamodel, thesis-specific modeling guidelines). This goes beyond parseability. The structure and content are meaningful within that language.

Semantic validity (referential consistency). References across artifacts resolve and are compatible (user task ↔ form, form ↔ UML class, lane ↔ role, etc.). This category captures the "multi-model" nature of the artifact. Correctness depends on relationships, not only on file-level validity.

These dimensions intentionally separate *what can be checked locally* from *what requires the full model set*. This also enables better error messages: an invalid JSON file is a different class of issue than a valid JSON file that references a non-existent BPMN task.

5.3 Pipeline (detailed)

This section describes the validation pipeline used by the artifact. Importantly, the pipeline largely enforces rules already defined earlier in the thesis (especially the modelling guidelines), but turns them into concrete, repeatable checks. This also aligns with the general inconsistency-management view that rules should be explicit and enforceable rather than informal [9, 58].

5.3.1 Syntax solo

- **BPMN** – well-formed XML (parseable; BPMN namespaces and tags).
- **Config, states file, forms files, email config file, REST config file** – well-formed JSON.
- **DMN** – well-formed XML (parseable; DMN namespaces and tags).
- **Email templates** – valid FTL (FreeMarker) syntax.
- **Data object references** – labels follow the proprietary syntax rules (variable name, domain object, state).
- **Expressions** – valid UEL syntax with markers for process variables and global variable references.
- **Organization** – well-formed XML (parseable; ArchiMate namespaces and tags).
- **Business data** – valid UML syntax (a text-based notation is required for validation, e.g. PlantUML[1]).

5.3.2 Semantic solo

- **BPMN** – conforms to BPMN metamodel (e.g. via a BPMN parser); restrictions from Section 4.4 are enforced (e.g. no engine-specific elements; nodes and lanes have names).
- **Config, states file, email config file, REST config file** – conform to the corresponding JSON Schema.
- **DMN** – conforms to DMN metamodel (parseable with a DMN parser).
- **Email template:** – The template must be renderable using the parameters defined in the associated email config.
- **Expressions** – parseable with a UEL parser.
- **Organization** – conforms to ArchiMate and restrictions from Section 4.4.
- **Business data** – conforms to UML and restrictions from Section 4.4.
- **Forms** – well-formed JSON and conformance to the form JSON Schema standard.

[1] PlantUML language reference: https://plantuml.com/.

It is also worth mentioning that, in a real implementation of the artifact, some other validation rules may be enforced, e.g., depending on the BPMS, reserved words/keywords may not be used in the UML model for the attributes (e.g., in Bonita, `timestamp` is a reserved word).

5.3.3 Semantic multi

This stage validates cross-model consistency (referential and semantic alignment across artifacts). Since elements are joined primarily by `name` (and, where applicable, by typed variable paths), the overall correctness depends on (i) unambiguous naming and (ii) the resolvability of all references and expressions across the model set.

Configuration file (JSON).

- **Config → BPMN:** Every process, lane, event, and task referenced in the config (by `name`) must exist in the BPMN model. Process, tasks, lanes, and events are linked to BPMN flow nodes and data object references by name. (Section 4.4.1).
- **Config → Forms, DMN, Email, REST (file references):** All file reference expressions (`formRef`, `dmnRef`, `emailJsonRef`, `emailFtlRef`, `restCallRef`) must follow the `${file:path}` form and resolve to existing files (Section 4.4.11).
- **Config → Organization (ref expressions):** All assignees expressed as `${ref:organization.<name>}` (lane-level or task-level) must resolve to elements in the ArchiMate model by name (Section 4.4.6, Section 4.4.11).
- **Config → Global variables:** Every occurrence of `globalVariables.X` inside a UEL expression must reference an entry that exists in the config (Section 4.4.7, G21).
- **Config (DMN result variable):** For business rule tasks, `dmnResultVariable` must be a UEL expression using variable markers (G20) and must target an existing writable process variable (defined via BPMN data objects and UML attributes; see below). The target attribute must exist in the corresponding UML class and be type-compatible with the DMN decision output (Section 4.4.11, Section 4.4.7, Section 4.4.8).

BPMN (process control-flow).

- **BPMN → UML (business data):** Every BPMN data object `typeName` (from the data object reference label, Section 4.4.3) must exist as a UML class in the business data model (Section 4.4.2).
- **BPMN → States:** Every BPMN data object that specifies a `stateName` (state tag in the label) must have that state defined in the states file for its class type. The class must appear in the states file and the state must appear in that class's state list (Section 4.4.4).
- **BPMN expressions:** If UEL expressions occur in BPMN (e.g. gateway conditions), they must follow the UEL syntax rules (G19) and use the variable marker convention (G20). Each referenced variable root must correspond to a process variable represented by a BPMN data object, and each referenced attribute path must exist in the corresponding UML class (Section 4.4.7, Section 4.4.2).

Business data (UML) and data states (JSON file).

- **States → UML:** Every class referenced in the states file must exist as a class in the UML diagram. Every `requiredFields` entry in a state must correspond to an attribute of that class (Section 4.4.2, Section 4.4.4).
- **Forms ↔ States (requiredness):** If a data object has a state with `requiredFields`, those fields must appear in the form's `required` array for the task that produces/updates that data object. Additionally, the task must be associated (via config and BPMN) with the same UML class to which the state definition applies (Section 4.4.5, G15).

Forms (JSON Schema).

- **Config → Forms:** Every `formRef` in config tasks (e.g. `${file:forms/request_absence_form.json}`) must resolve to an existing form file (Section 4.4.11).
- **Forms → BPMN → UML:** Form schema properties must align with the UML class of the data object type associated with the task (via config and BPMN). Properties that do not exist in the UML class cause validation failure. JSON Schema property types must be type-compatible with the corresponding UML attribute types (Section 4.4.5, G15).

Decision logic (DMN).

- **Config → DMN:** Every `dmnRef` in config tasks or events (e.g. `${file:dmn/loan_approval.dmn}`) must resolve to an existing DMN file (Section 4.4.8, Section 4.4.11).
- **DMN input expressions:** DMN inputs that reference process variables must be expressed as UEL expressions with variable markers (G22). Each referenced process variable must exist in the BPMN data objects and resolve to attributes in the UML model (Section 4.4.8, Section 4.4.7, Section 4.4.2).
- **DMN output alignment:** The decision table output(s) must be compatible with the variable targeted by `dmnResultVariable`. The declared `typeRef` of the DMN output must be type-compatible with the UML attribute type of the target process variable (Section 4.4.8, Section 4.4.2).

Email (JSON + FreeMarker).

- **Config → Email files:** Every `emailJsonRef` and `emailFtlRef` in config tasks must resolve to existing email JSON and FTL template files (Section 4.4.9, Section 4.4.11).
- **Template parameter completeness:** For every email-related task, there must be a pair: email configuration JSON file and FTL file. Every FreeMarker variable used in the template must be provided through `body.parameters` in the email JSON (G24). Missing parameters invalidate the model (Section 4.4.9).
- **No UEL in templates:** UEL expressions must not appear inside the FreeMarker template body (G24). All dynamic values must be expressed in the email JSON (headers or `body.parameters`) using UEL where needed (Section 4.4.9, Section 4.4.7).
- **Email JSON expressions:** UEL expressions inside the email JSON (e.g. in headers or parameter values) must satisfy the UEL syntax and marker rules (G19–G21) and must reference resolvable process variable paths and/or resolvable global variables.

Variable paths must resolve to UML attributes via BPMN data objects (Section 4.4.7, Section 4.4.2).

- **Config (SMTP):** If any config task uses `emailJsonRef`, the config must define `smtpConfig` (Section 4.4.11).

REST (JSON).

- **Config → REST:** Every `restCallRef` in config tasks (e.g. `${file:rests/notify_employee_rest.json}`) must resolve to an existing REST configuration file (Section 4.4.10, Section 4.4.11).
- **REST expressions:** UEL expressions used in REST configuration files (e.g. URLs or payload fields) must satisfy the UEL syntax and marker rules (G19–G21). All referenced process variable paths must resolve to UML attributes via BPMN data objects, and all global variable references must resolve to entries in `globalVariables` (Section 4.4.10, Section 4.4.7, Section 4.4.2).

5.4 Reporting, determinism, and limits

The validation pipeline produces error messages so that problems can be located and fixed quickly. Errors point to the model and describe the failed rule (e.g., “task has `formRef` but no BPMN data output”). Validation should be run every time the model set is changed, to catch errors early.

Cross-model checks are designed to be deterministic and order-independent. They do not depend on iteration order over files, and name-based resolution assumes uniqueness so that references are unambiguous.

Some constraints cannot be fully validated statically. For example, REST and email configs can be checked for syntax and schema, but not for service reachability. FreeMarker templates can be rendered with dummy data to catch missing variables, but not all branches without runtime data.

5.5 Chapter summary

This chapter defined how consistency is maintained across the platform-independent model set. It introduced the linking strategy, consistency dimensions (syntax, standalone semantics, and referential consistency), and the validation pipeline (syntax solo, semantic solo, semantic multi). The outcome is a validated model set that can be used as a reliable basis for engine-specific generation.

6 Engine-specific artifact generation

This chapter describes the *generation* stage of the thesis artifact (RQ1.3 ”How can platform-independent models be transformed into engine-specific execution artifacts?”). The input is a validated platform-independent model set and the output is a set of engine-specific, deployable artifacts for a target BPMS.

6.1 Purpose and scope of generation

What “generation” means. Generation transforms the validated platform-independent model set into deployable engine-specific artifacts. Input models are parsed, references are resolved, and engine-specific mappings are applied to produce a deployment bundle for the target BPMS.

Boundary: what is generated vs. what remains manual. The information requirements from earlier chapters do not cover absolutely everything an engine needs for a production environment. The main objective is therefore to generate a deployable baseline that captures the process structure and core execution behavior (process definition, forms, connectors, organizational mapping, etc.). Examples of concerns that typically remain manual include:

- Identity store configuration (e.g., containing detailed user data, e.g., user email, password, etc.)
- Form styling
- Partially runtime configuration (e.g., database connection details, etc.)
- External system endpoints and networking details
- etc.

Extending the information requirements to eliminate all post-generation manual work would be a substantial effort and is beyond the scope of this thesis.

6.2 High-level generation pipeline

This section presents the generation pipeline at a high level. The pipeline assumes that the inputs are already consistent according to the validation and cross-validation rules defined in Chapter 5.

6.2.1 Pipeline stages

1. **Load and parse platform-independent artifacts.**
   Information requirements captured in models, namely, BPMN, UML, ArchiMate, JSON Schema, JSON, and DMN files, are each loaded into an internal representation suitable for downstream processing.

2. **Prepare an enriched internal representation.**
   This is the opportunity to resolve references between different models with their final values, where possible.

   The first mandatory step is resolving pre-runtime (global variables). As explained in previous chapters, many models can reference global variables defined in a config file using UEL with markers. These references should be resolved pre-runtime

when they are statically decidable. For example, in Listing 4.7 we have a part of a JSON file that looks like this:

```
"reviewLink":"${~globalVariables.appBaseUrl~}/requests/${~req1.requestId~}"
```

It should be resolved to
`"reviewLink": "http://localhost:8080/requests/${~req1.requestId~}"`
where the global variable is defined as `"appBaseUrl": "http://localhost:8080"`.

Another example of model enrichment is the BPMN representation: we update lanes and user tasks with assignee information by resolving it from the organization model (via the config file). Assignees from lanes are then propagated to user tasks if not explicitly set for those tasks in the config file.

In general, the extent to which internal representations can be enriched before the artifact generation step is directly related to the list of BPMS engines that a given pipeline implementation will support. For DMN files, for example, the obvious approach would be to update the BPMN internal representation with the DMN file reference and result variable name for all business rule tasks. However, some engines support DMN natively while others do not, in which case a script emulating the decision table may be needed. Thus, from this stage onwards, the pipeline is less strictly defined.

Overall, the purpose of enriched internal representations is that they will be used as a basis for artifact generation, so it is important that they are as complete as possible. Otherwise, the generator would have to read the same information from the same source multiple times for each engine, so these data structures should be enriched as much as possible.

3. **Apply engine-specific mapping rules and emit engine artifacts.**
   It is possible to reuse the enriched internal representations by applying engine-specific mapping rules to them and emitting the artifacts. Another approach would be to emit internal representations as files and then update them by applying engine-specific mapping rules. The decision should be made based on the list of target BPMS and what is more convenient for the implementation.

   Examples of mapping rules are:

   - Expression dialect conversion: UEL to Groovy, etc.
   - Adding BPMS-specific attributes to BPMN: e.g. `<camunda:formKey="...">`, etc.
   - Artifact format mapping: e.g., Organization to BPMS-specific format, etc.

   The exact set of mapping rules and files that should be emitted is determined by the targeted BPMS engines.

6.3 Generation as M2M and M2T transformations

The pipeline described above is a Model-Driven Engineering (MDE) pipeline: the enrichment stage corresponds to model-to-model (M2M) transformation, and the emission of artifacts to model-to-text (M2T) transformation. In this thesis, Chapter 7 describes an implementation that takes a fully programmatic approach: the pipeline is realised as direct code that parses models, enriches internal structures, and emits files.

An alternative is to implement the same pipeline using dedicated MDE transformation and code-generation tools. OMG specifies languages for these categories, for example, QVT for M2M and MOFM2T for M2T.[1][2] The MDE literature discusses M2M vs. M2T and the relationship between them [59]. For M2M, model transformation languages such as ATL [60] are commonly used. For M2T, template-based or visitor-based code generators (e.g., Acceleo[3], or Xtend[4] in the Eclipse ecosystem) can produce concrete artifacts from a model. Such tools could be used to realise the enrichment and emission stages instead of hand-written programmatic code.

6.4 Summary

This chapter defines artifact generation as the transformation of a validated platform-independent model set into deployable, engine-specific artifacts. It described the purpose and scope of generation (a deployable baseline rather than full production configuration), the high-level pipeline (load and parse, enrich internal representations, apply engine-specific mapping rules, and emit), and the MDE view of that pipeline as M2M and M2T.

[1] https://www.omg.org/spec/QVT/1.3/PDF
[2] https://www.omg.org/spec/MOFM2T/1.0/PDF
[3] https://eclipse.dev/acceleo/
[4] https://eclipse.dev/Xtext/xtend/documentation/

7 Implementation

This chapter documents a concrete prototype implementation of the pipeline described in this thesis. The primary objective is to demonstrate the end-to-end process, starting from modelling of information requirements, continuing with validation of models, and ending with artifact generation.

The implementation is written in Java 17 and uses Maven for dependency management. Two BPMS are targeted: Camunda 7 and Bonita. The codebase is available on GitHub (Appendix C).

The chapter uses *AbsenceRequest*[1] BPMN process as an example. This is a common example used in literature and tutorials. The BPMN model alone is, of course, not enough for a deployment, so additional information requirements were added to make the process executable. This is the same BPMN process that was used in examples from the previous chapters (Figure 4.2).

The purpose of this chapter is not to provide exhaustive code-level documentation. Instead, it focuses on how the Java prototype realizes the pipeline stages described earlier.

7.1 Modelling of information requirements

The detailed modelling of information requirements for *AbsenceRequest* was already largely covered in Section 4.4. This section puts focus on other aspects, e.g., the tooling used.

7.1.1 Process model

The process is modelled as a BPMN 2.0 diagram. Camunda Modeler[2] was used to make the model. Any other BPMN-compliant tool could be used too (e.g., `bpmn.io`[3]).

7.1.2 Business data model

Business data is modelled using UML class diagrams, more specifically PlantUML[4]. PlantUML was selected because it is text-based and can be easily parsed programmatically. In principle, any other UML-compliant tool could be used too, as long as it can be parsed.

```
@startuml
class AbsenceRequest {
    requestId: int
    employeeId: int
    absenceType: string
    startDate: long
    endDate: long
    reason: string
    managerId: int
    status: Status
    clarificationMessage: ClarificationMessage
}
class ClarificationMessage {
    message: string
    timestampp: long
```

[1] https://camunda.com/try-modeler/
[2] `https://camunda.com/download/modeler/`
[3] `https://demo.bpmn.io/new`
[4] `https://plantuml.com/class-diagram`

```
}
enum Status {
    Submitted
    ClarificationRequired
    Approved
    Rejected
}
AbsenceRequest *-- ClarificationMessage

//...
@enduml
```

Listing 7.1: Part of the `business_objects.puml` file used for capturing business data information, corresponding to Figure A.2

7.1.3 Organization model

The organization is represented in ArchiMate[5]. The Archi[6] tool was used to make the model. Any other ArchiMate-compliant tool could be used too.

7.1.4 Forms

Three JSON Schemas for the forms were created for the process. Tools such as `json.ophir.dev`[7] can be used to design the schemas visually.

7.1.5 DMN files

DMN files can be created using tools such as Camunda Modeller or `dmn.io`[8]. Note that business rule tasks do not appear in this particular process (`AbsenceRequest`). Examples using business rule tasks can be found in the GitHub repository (Appendix C LoanRequest example).

7.1.6 Other models

The config file, state model, email config file, and REST config file were handwritten in JSON format by following their corresponding JSON Schemas. The template file for the email task was handwritten in FreeMarker format. Data object references do not need separate files as they are part of the BPMN model. The same applies to expressions. They can be part of other models.

The complete set of model files for AbsenceRequest looks as follows:

```
models/
|-- email/
|   |-- remind_line_manager_email.ftl
|   `-- remind_line_manager_email.json
|-- form/
|   |-- approve_absence_form.json
|   |-- provide_further_explanation_form.json
|   `-- request_absence_form.json
|-- rest/
|   `-- notify_employee_rest.json
|-- business_objects.puml
|-- config.json
|-- diagram_1.bpmn
|-- diagram_1-1.0.proc
|-- organization.archimate
`-- states.json
```

[5] https://pubs.opengroup.org/architecture/archimate3-doc/
[6] https://www.archimatetool.com/
[7] https://json.ophir.dev/
[8] https://dmn.io/

Listing 7.2: Complete set of model files for `AbsenceRequest` process

7.2 Project overview

7.2.1 More details on technology stack

The implementation is written in Java 17 and uses Maven for dependency management. The dependencies are primarily used for parsing and validation: Jackson for JSON parsing[9], FreeMarker for template processing[10], and a JSON Schema validator implementation (NetworkNT)[11]. Additionally, Camunda's BPMN model API[12] is used to validate BPMN 2.0 conformance.

7.2.2 Architecture and module structure

The `run` method is the main function of the project. It orchestrates the pipeline. As can be seen in the code snippet below (Listing 7.3), the implementation is mostly focused on the validation (lines 2-4) and generation of artifacts (lines 6-10).

```
1  public void run() throws Exception {
2    validateModelsSolo();
3    parseModelsToInternalDataStructures();
4    crossValidateModels();
5
6    enrichInternalRepresentationsOfModels();
7    switch (targetBPMS) {
8        case "camunda" -> camundaGenerateArtifacts();
9        case "bonita" -> bonitaGenerateArtifacts();
10   }
11 }
```

Listing 7.3: Orchestration of the pipeline in `Main.run()`.

The `Main` class also defines the file paths to the model files (corresponding to the Listing 7.2 directory in the project) and holds the internal representation data structures of them. A simplified excerpt is:

```
1  class Main {
2    // Paths to model files
3    private final String bpmnFilePath = ".../diagram_1.bpmn";
4    private final List<String> formPaths = List.of(".../request_absence_form.
         json", ...);
5    private final List<String> restActionPaths = List.of(".../
         notify_employee_rest.json");
6    private final List<String[]> emailConfigsPaths = List.of(new String[]{"...
         ftl","...json"});
7    private final String organizationPath = ".../organization.archimate";
8    private final String processConfigPath = ".../config.json";
9    private final String statesPath = ".../states.json";
10   private final String businessDataPath = ".../business_objects.puml";
11   private final List<String> dmnPaths = List.of();
12
13   // Internal representation data structures of the model files
14   private OrganizationData organizationData;
15   private ConfigFile configFileData;
16   private BusinessDataStates statesData;
```

[9] https://github.com/FasterXML/jackson
[10] https://freemarker.apache.org/
[11] https://github.com/networknt/json-schema-validator
[12] https://docs.camunda.org/manual/latest/user-guide/model-api/bpmn-model-api/

```
  private BusinessData businessData;
  private BpmnData bpmnData;

  // ...
}
```

Listing 7.4: Simplified excerpt of `Main`: paths and in-memory data.

Of all internal representation data structures, probably the most notable is the `BpmnData` structure. It is the top-level structure that represents the BPMN model. While full-fidelity BPMN object models exist, the prototype employs a simpler pragmatic representation tailored to the needs of the project.

Code snippets for the `BpmnData` structure are shown below:

```
record BpmnData(String id, String targetNamespace,
                Collaboration collaboration,
                List<ProcessDef> processes) {}
```

Listing 7.5: `BpmnData`: top-level BPMN structure.

```
record ProcessDef(String id, String name, boolean isExecutable,
                  LaneSet laneSet,
                  Map<String, FlowNode> nodesById,
                  Map<String, DataObjectReference> dataObjectsById,
                  Map<String, SequenceFlow> flowsById) {}
```

Listing 7.6: `ProcessDef`: data structure for a process

```
record Lane(String id, String name, List<String> flowNodeRefs,
            Map<String, String> vendorAttributes,
            String resolvedActor) {}
```

Listing 7.7: `Lane`: data structure for a lane

```
record FlowNode(String id, String type, String name,
                Map<String, String> vendorAttributes,
                List<DataInputAssociation> dataInputAssociations,
                List<DataOutputAssociation> dataOutputAssociations,
                String resolvedActor, String resolvedFormRef,
                String resolvedEmailConfigFileName,
                String resolvedRestCallFileName,
                // ...
```

Listing 7.8: `FlowNode`: data structure for a flow node (e.g., task, event, etc.)

For the rest of the project, a strict separation of concerns is maintained between (i) platform-neutral helpers and (ii) engine-specific helpers and generators. This decomposition makes the implementation extensible to adding other target-BPMS engines.

Platform-neutral helper modules (e.g., `BpmnHelper`, `FormHelper`, `ProcessConfigHelper`, `EmailHelper`, `RestHelper`, `DmnHelper`) handle parsing, validation, and internal representation enrichment in a target-independent manner.

Engine-specific helper modules (e.g., `CamundaBpmnHelper`, `CamundaEmailHelper`, `CamundaRestHelper`) handle the mapping rules for specific BPMS on the same internal representation data structures.

Engine-specific generators then implement generation and emission. In the current prototype this includes e.g. `CamundaBpmnGenerator`, `BonitaProcGenerator`, `BonitaFormGenerator`,

BonitaBomGenerator, BonitaOrganizationGenerator, BonitaProcessConfigGenerator, etc.

7.3 Validation of models

As shown in Listing 7.3, the validation part is composed of 3 methods: `validateModelsSolo()`, `parseModelsToInternalDataStructures()`, and `crossValidateModels()`.

7.3.1 Validation of models in isolation

The `validateModelsSolo()` method is responsible for validating the models in isolation, both for syntax and semantics. This was described in Section 5.3.1 and Section 5.3.2, respectively.

Code snippet for the `validateModelsSolo()` method is shown below:

```
1  private void validateModelsSolo() {
2    BpmnValidator.validate(new File(bpmnFilePath));
3    ProcessConfigHelper.validateConfigFile(processConfigPath);
4    StatesHelper.validateStatesFile(statesPath);
5    formPaths.forEach(path -> validateOrThrow(() -> FormHelper.validate(path))
         );
6    OrganizationDataHelper.validate(organizationPath);
7    dmnPaths.forEach(DmnHelper::validate);
8    emailConfigsPaths.forEach(config -> validateOrThrow(() -> EmailHelper.
         validate(config[0], config[1])));
9    restActionPaths.forEach(path -> validateOrThrow(() -> RestHelper.validate(
         path)));
10 }
```

Listing 7.9: Solo validation: syntax and semantic checks.

In the prototype, the concrete mechanisms are as follows. (i) BPMN is validated using Camunda's model API: the BPMN XML is parsed and checked against the BPMN metamodel.[13] (ii) JSON-based artifacts (forms, REST configs, email configs, state model, and process config) are parsed with Jackson and validated against JSON Schema using NetworkNT's validator.[14] (iii) DMN files are parsed as XML and checked for required DMN structure (e.g., namespace and presence of decisions), using standard Java XML tooling (DOM). (iv) Email validation additionally performs a template-parameter check by rendering the FreeMarker template with dummy parameters to ensure referenced variables are available.

7.3.2 Validation of semantic consistency across the models

The `crossValidateModels()` method is responsible for validating semantic consistency between the models, as described in Section 5.3.3. In this particular implementation, for convenience, the `parseModelsToInternalDataStructures()` is called before `crossValidateModels()`.

Parsing to internal representation

Code snippet for the `parseModelsToInternalDataStructures()` method is shown below:

```
1 private void parseModelsToInternalDataStructures() throws Exception {
2   organizationData = OrganizationDataHelper.loadFromFile(organizationPath);
3   configFileData = ProcessConfigHelper.loadConfigFile(processConfigPath);
4   statesData = StatesHelper.loadDataFromFile(statesPath);
5   businessData = BusinessDataHelper.loadFromFile(businessDataPath);
6   bpmnData = BpmnHelper.parseBpmnFile(bpmnFilePath);
```

[13] https://docs.camunda.org/manual/latest/user-guide/model-api/bpmn-model-api/
[14] https://github.com/networknt/json-schema-validator

```
}
```

Listing 7.10: Parsing raw models to internal representations.

To create an instance of `BpmnData`, DOM-based traversal of the BPMN 2.0 XML is used. The configuration file (`config.json`) is parsed using Jackson into a `ConfigFile` Java object instance. In case of `businessData`, a custom lightweight parser is used to make a `BusinessData` Java object instance. Similar parsers are used to parse organization and state models into `OrganizationData` and `BusinessDataStates` Java object instances, respectively.

In general, what models should be parsed to internal representation, and which should be left in raw formats, is a matter of design choice for each implementation. In this case, for example, the JSON Schemas for forms (e.g. `request_absence_form.json`) are left as raw JSON objects, as this is enough for validating against the JSON Schema Standard (draft 2020-12) and then using them for form generation. More sophisticated models, such as BPMN 2.0 XML, are parsed to internal representations, as this is more convenient for the subsequent mapping and generation steps.

Cross-validation
Cross-validation enforces referential integrity and semantic alignment across the model set. It ensures that the separate models form a coherent executable specification rather than a collection of independent files. The implementation of the cross-validation is shown in the code snippet below and generally follows the steps described in Section 5.3.3.

```
private void crossValidateModels() {
  ProcessConfigHelper.validateLaneAssignees(configFileData, organizationData);
  ProcessConfigHelper.validateTaskFormReferences(configFileData, formPaths);
  ProcessConfigHelper.validateRestCallReferences(configFileData,
      restActionPaths);
  ProcessConfigHelper.validateDmnReferences(configFileData, dmnPaths);
  List<FormDataObjectPair> pairs =
    ProcessConfigHelper.validateTasksWithFormsHaveDataOutputs(configFileData,
        bpmnData);
  ProcessConfigHelper.validateDataObjectTypesExistInPuml(bpmnData, statesData)
      ;
  BusinessDataHelper.validateStatesExistInBusinessData(statesData,
      businessData);
  // ...
}
```

Listing 7.11: Cross-validation: referential and semantic consistency across artifacts.

In case validation fails, the method throws an exception. Exceptions are accompanied by descriptive messages of what went wrong and where. For example:

```
  throw new IllegalStateException(
    String.format("State '%s' of class '%s' requires field '%s', " +
                  "but field '%s' does not exist in the PUML class
                      definition. " +
                  "Available fields in '%s': %s",
            state.name, className, requiredField, requiredField, className,
                pumlFieldNames));
```

Listing 7.12: Exception message for in `BusinessDataHelper.validateStatesExistInBusinessData`

7.4 Generation of artifacts

In this implementation of the pipeline, the artifact generation is performed in 2 steps. First, the internal representation of the models is enriched. Next, the engine-specific artifact generation is performed. These two actions correspond to lines 6-10 in Listing 7.3.

7.4.1 Enrichment of internal representation

The enrichment of the internal representation is performed in the `enrichInternalRepresentationsOfModels()` method. Enrichment was described in Section 6.2. At this step, only vendor-neutral enrichments are applied, so that next they can be used by any of the engine-specific generators later on.

```
1  private void enrichInternalRepresentationsOfModels() {
2    processedEmailFiles = EmailHelper.resolveGlobalVariablesInEmailConfigs(
         emailConfigsPaths, configFileData);
3    processedRestFiles = RestHelper.resolveGlobalVariablesInRestActions(
         restActionPaths, configFileData);
4    bpmnData = BpmnHelper.resolveGlobalVariablesInSequenceFlows(bpmnData,
         configFileData);
5    processedDmnFiles = DmnHelper.resolveGlobalVariablesInDmnFiles(dmnPaths,
         configFileData);
6    bpmnData = BpmnHelper.updateLanesWithAssignees(bpmnData, configFileData,
         organizationData);
7    bpmnData = BpmnHelper.updateUserTasksWithLaneAssignees(bpmnData);
8
9    //...
10 }
```

Listing 7.13: Enrichment of internal representation.

7.4.2 Engine-specific artifact generation

This implementation targets 2 BPMS: Camunda 7 and Bonita.

Camunda artifact generation

For Camunda, the artifact generation is performed in the `camundaGenerateArtifacts()` method (Listing 7.3). The emitted artifacts are intended to be copied into a bootstrapped Camunda 7 (Spring/Maven) project (more details in Appendix C README.md file).

```
1  private void camundaGenerateArtifacts() {
2    //...
3    CamundaBpmnGenerator.updateUserTasksFormKeyInDocument(bpmnDocument, bpmnData
         );
4    CamundaBpmnGenerator.generateCamundaSpecificAttributes(bpmnDocument);
5    //...
6    DmnHelper.saveProcessedDmnFiles(processedDmnFiles, "src/main/resources/out/
         camunda");
7    //...
8  }
```

Listing 7.14: Camunda artifact preparation and generation.

The full implementation of the `camundaGenerateArtifacts()` method is quite long and can be browsed in the Git repository (Appendix C), but overall its a mix of methods resposible for either (i) *preparation for generation* (engine specific mapping rules) and (ii) *artifact emission* (writing files). Some of the interesting preparation and generation steps are described below.

Forms For user tasks, the prototype generates Camunda embedded forms[15] from JSON form definitions. The generator maps each form field into an HTML input element and uses `cam-variable-name` and `cam-variable-type` attributes to bind submitted values to process variables.

An excerpt from a generated form (`request_absence_form.html`) is shown below (simplified):

```
<form role="form" name="AbsenceRequest">
  <label for="req1_requestId">Request Id *</label>
  <input type="number" id="req1_requestId"
         cam-variable-name="req1_requestId"
         cam-variable-type="Long" required>
  <!-- ... -->
  <button type="submit">Submit</button>
</form>
```

Listing 7.15: Excerpt from a generated Camunda embedded form.

The corresponding user task is then updated in the BPMN XML with a `camunda:formKey` attribute that references the generated HTML file:

```
<bpmn:userTask camunda:assignee="alice"
  camunda:formKey="embedded:app:forms/provide_further_explanation_form.html"
  id="Activity_1083e10" name="Provide further Explanation">
```

Listing 7.16: BPMN user task with Camunda form key.

Generated forms do not compromise on any information from the original form definitions made as JSON Schemas. This inclues e.g. valiging `required` fields, enforcing `type` constraints as well as support for nested objects and enums (`<select>`s are generated). Furthermore, `cam-variable-name` and `cam-variable-type` attributes allow prefilling the form with the process variables that already exist in the process instance.

Although in this implementation, embedded forms are used, this is, of course, not the only option. Camunda 7 supports other form representations, such as inline forms, Camunda Forms [16] and external forms [17]. Generating any of those would require a similar approach for the form generation step. In case of external forms, widespread generation tools are available, such as `RJSF` [18].

Emails For email-related service tasks, the prototype emits (i) a processed configuration JSON and (ii) the corresponding FreeMarker template, and updates the BPMN service task to reference a Spring-managed delegate bean during runtime through `camunda:delegateExpression`. The email config and template references are passed via `camunda:inputOutput`.

```
<bpmn:serviceTask camunda:delegateExpression="${sendEmailDelegate}"
  id="Activity_1q9he4w" name="Remind Line Manager">
  <bpmn:extensionElements>
    <camunda:inputOutput>
      <camunda:inputParameter name="configJson">remind_line_manager_email.json
          </camunda:inputParameter>
```

[15] https://docs.camunda.org/manual/7.24/reference/forms/embedded-forms/
[16] https://docs.camunda.org/manual/7.24/reference/forms/camunda-forms/
[17] https://docs.camunda.org/manual/7.24/reference/forms/external-forms/
[18] https://react-jsonschema-form.readthedocs.io/en/stable/

```
      <camunda:inputParameter name="templateFtl">remind_line_manager_email.ftl
          </camunda:inputParameter>
    </camunda:inputOutput>
  </bpmn:extensionElements>
</bpmn:serviceTask>
```

Listing 7.17: Camunda service task with email delegate and input parameters.

The `sendEmailDelegate` is a universal Spring-managed delegate bean that is emitted as part of the Camunda artifact generation. During runtime, the delegate reads the `configJson` and `templateFtl` parameters and uses them to send the email using the JavaMail API [19].

REST calls For REST calls, the prototype emits a processed REST configuration JSON file and updates the BPMN service task to reference a Spring-managed delegate bean during runtime through `camunda:delegateExpression`. The REST config reference is passed via `camunda:inputOutput`. The implementation is similar to the one for emails (the Spring bean takes care for making the REST API call).

DMN For DMN decisions, the prototype emits a processed DMN file and updates the BPMN business rule task to reference the DMN decision table through `camunda:decisionRef` and `camunda:decisionVariable` attributes.

Expressions Camunda 7, by default, supports UEL expressions, so these are supported out of the box by the runtime.

A practical issue that remains to be addressed is the use of process variables inside expressions. Since Camunda 7, out of the box, has poor support for JSON objects, primitive instance variables can only be used. This, for instance, means that expressions of type `${~req1.status~ == "NEEDS_CLARIFICATION"}` cannot just be rewritten to `${req1.status == "NEEDS_CLARIFICATION"}`. Instead, process variables are flattened to primitive variables, for instance, `req1.status` would be rewritten to `req1_status`. As a result, the expression `${~req1.status~ == "NEEDS_CLARIFICATION"}` would be rewritten to `${req1_status == "NEEDS_CLARIFICATION"}`. Consequently, only flattened process variables are used during runtime (e.g., as can already be seen in Listing 7.15).

BPMN The prototype augments the BPMN document with Camunda extension attributes so that the process is executable on the engine. `bpmnData` is first enriched and then traversed for generating the final `.bpmn` file. For example, on *lanes* and *user tasks*, it adds `camunda:assignee` with the resolved assignee (from the process config and organization model). The same user tasks also receive `camunda:formKey` pointing to the generated embedded HTML form (see Listing 7.16). On *service tasks*, the generator adds `camunda:delegateExpression` and `camunda:inputOutput` for email and REST (as in the paragraphs above). On *business rule tasks*, it adds `camunda:decisionRef` and `camunda:decisionVariable` for DMN. *Sequence flows* get resolved condition expressions in UEL. Additional vendor-specific attributes from the process config (e.g. `camunda:class`, custom extension attributes) are written onto the corresponding flow nodes. Finally, the generator adds engine-specific attributes to the document (e.g., async execution, job priority) where applicable.

[19] `https://docs.oracle.com/en/java/javase/17/docs/api/java.mail/jakarta/mail/package-summary.html`

DMN In case of DMN files, only expressions are resolved (as described in Section 7.4.2). The updated DMN files are then emitted to the output directory.

Output The output is written under `out/camunda`. A typical output directory contains the following files:

```
out/camunda/
|-- diagram_1.bpmn (main process)
|-- request_absence_form.html
|-- approve_absence_form.html
|-- provide_further_explanation_form.html
|-- notify_employee_rest.json
|-- remind_line_manager_email.json
|-- remind_line_manager_email.ftl
|-- RestCallDelegate.java
|-- SendEmailDelegate.java
`-- config.json
```

Listing 7.18: `AbsenceRequest` process output directory structure.

DMN files are included in the output directory as well, but the `AbsenceRequest` process does not have any business rule tasks, so no DMN files are emitted. The `config.json` file is emitted as well (in its original form) as it is used by the `SendEmailDelegate.java` to grab the SMTP configuration.

Engine-specific emission for Bonita
The Bonita artifact generation is performed in the `bonitaGenerateArtifacts()` method (Listing 7.3). The emitted artifacts are intended to be inserted into Bonita Studio project files (more details in Appendix C README.md file).

The overall method is similar to the one for Camunda, however, the artifact preparation and emission differ a lot. In contrast to Camunda's relatively flexible runtime variable handling, Bonita imposes stricter, more explicit configuration requirements. As a consequence, the number of generated artifacts is larger, and the generator must explicitly use constructs such as task contracts and operations that map inputs into business data.

```
private void bonitaPrepareAndGenerateArtifacts() {
  BonitaProcGenerator.addDataObjectsToPoolInDocument(procDocument, bpmnData);
  //...
  BonitaOrganizationGenerator.generateOrganizationFile(organizationData, "src/
      main/resources/out/bonita");
  BonitaBomGenerator.generateBomFile(businessData, "src/main/resources/out/
      bonita");
  BonitaProcessConfigGenerator.generateBonitaProcessConfigFile(procDocument,
      bpmnData, "src/main/resources/out/bonita", organizationData);
  //...
}
```

Listing 7.19: Bonita backend: preparation and artifact emission.

Some of the interesting preparation and generation steps are described below.

Organization Bonita manages actors, groups, and roles in a separate organization file rather than embedding them in the process definition. The prototype generates this file from the `organizationData` object and emits `Organization.xml` in the Bonita output directory for use during deployment. Since Bonita's organizational model is very similar to the organizational model used in the thesis, all the information can be mapped pretty much one-to-one.

One difference, though, is that Bonita's organizational model allows additional user (actor) information to be added, from which some attributes are required to be provided, such as the user's password. To ensure that the organization file is valid, a basic password is generated for each user.

```
  <?xml version="1.0" encoding="ASCII"?>
<organization:Organization xmlns:organization="http://documentation.bonitasoft
    .com/organization-xml-schema/1.1">
  <users>
    <user userName="alice">
      <personalData/>
      <professionalData/>
      <password encrypted="false">bpm</password>
      <customUserInfoValues/>
    </user>
    <!-- ... -->
  <groups>
    <group name="DK Facility" parentPath="/EU Region">
      <displayName>DK Facility</displayName>
    <!-- ... -->
  <memberships><membership>
      <userName>alice</userName>
      <roleName>employee</roleName>
      <groupName>DK Facility</groupName>
      <groupParentPath>/EU Region</groupParentPath>
    <!-- ... -->
```

Listing 7.20: Parts of the `Organization.xml` file generated for the `AbsenceRequest` process.

Assignment of assignees for the tasks works differently in Bonita, too, and is more sophisticated. For instance, it is not possible to directly add a `assignee` attribute to the BPMN user task that would reference an entity in the Bonita organization model. Instead, actor mappings are first created in a separate configuration file:

```
  <configuration:Configuration xmlns:configuration="http://www.bonitasoft.org/
    ns/bpm/configuration" ...>
  <actorMappings>
    <actorMapping name="Line Manager"><groups/><memberships/><roles/>
      <users>
        <user>bob</user>
      </users>
    </actorMapping>
    <actorMapping name="Employee"><groups/><memberships/><roles/>
      <users>
        <user>alice</user>
      </users>
    </actorMapping>
    <!-- ... -->
```

Listing 7.21: Parts of the `_ZvkdoO7sEfCoIMCqMDCGog.conf` file generated for the `AbsenceRequest` process.

This implementation uses the convention of naming the mapping by the name of the lane or user task where the assignee is placed.

Once the actor mappings are created, the BPMN Pool where these mappings are used is updated to reference them.

```
<!-- Pool -->
```

```
  <actors name="Line Manager" xmi:id="_ZxH80O7sEfCoIMCqMDCGog" xmi:type="
      process:Actor"/>
  <actors name="Employee" xmi:id="_fIi1xMazhD9GtRrWYbZa35" xmi:type="
      process:Actor" initiator="true"/>
<!-- ... -->
```

Listing 7.22: Parts of the `diagram_1-1.0.proc` file generated for the `AbsenceRequest` process.

Then, the lanes are updated to reference, in this case, the actors.

```
<!-- Pool -->
  <elements actor="_vQbp8P0Jbxk9ffpoY4qm8M" name="Line Manager" xmi:id="
      _ZwOk8O7sEfCoIMCqMDCGog" xmi:type="process:Lane">
<!-- ... -->
```

Listing 7.23: Part of the `diagram_1-1.0.proc` file generated for the `AbsenceRequest` process with updated `Line Manager` lane.

Bonita Configuration As mentioned earlier, Bonita uses additional configuration files for deployment. The generator creates a `_<processId>.conf` file. The file contains mappings (discussed earlier) as well as the process dependencies (used by connectors for emails and REST calls).

Business object model Bonita uses a business object model (BOM) to manage the business data objects. The prototype generates this file from the in-memory `businessData` and emits `bom.xml` in the Bonita output directory for use during deployment. Type mapping is implemented. Primitive types map mostly one-to-one (e.g., Bonita allows specifying the length of the STRING type, but this information is not available in the business data model). In case of Enums, they are mapped to `STRING`. For class relations, composition and aggregation are supported. Attribute descriptions, unique constraints, queries, and indexes are generated empty.

```
  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<businessObjectModel xmlns="http://documentation.bonitasoft.com/bdm-xml-schema
    /1.0" modelVersion="1.0">
    <businessObjects>
        <businessObject qualifiedName="com.company.model.AbsenceRequest">
            <fields>
                <field collection="false" length="255" name="requestId"
                    nullable="true" type="INTEGER"/>
                <field collection="false" length="255" name="employeeId"
                    nullable="true" type="INTEGER"/>
                <field collection="false" length="255" name="absenceType"
                    nullable="true" type="STRING"/>
                <field collection="false" length="255" name="startDate"
                    nullable="true" type="LONG"/>
                <field collection="false" length="255" name="endDate" nullable
                    ="true" type="LONG"/>
                <field collection="false" length="255" name="reason" nullable=
                    "true" type="STRING"/>
                <field collection="false" length="255" name="managerId"
                    nullable="true" type="INTEGER"/>
                <field collection="false" length="255" name="status" nullable=
                    "true" type="STRING"/>
                <relationField collection="false" fetchType="LAZY" name="
                    clarificationMessage" nullable="true" reference="com.
                    company.model.ClarificationMessage" type="COMPOSITION"/>
```

```
            </fields>
            <uniqueConstraints/>
            <queries/>
            <indexes/>
        </businessObject>
    <!-- ... -->
```

Listing 7.24: Parts of the `bom.xml` file generated for the `AbsenceRequest` process.

Forms All the information contained in the JSON Schemas is used to generate Bonita forms [20]. This includes selectors for enums, required fields validation, type contraint and nested objects support. The forms are generated as mini bundles consisting of assets and the main form definition file in a Bonita-specific format. During runtime, form inputs are prefilled with the process variables that already exist in the process instance.

Pool variables, contracts and operations Pool variables are variables that are shared between the tasks in the process. They have to be explicitly declared in the BPMN Pool before the deployment. For this purpose, the process variables (in-memory `bpmnData`) are used to generate the pool variables:

```
<elements name="absenceRequest" xmi:id="_ZvkdoO7sEfCoIMCqMDCGog" xmi:type="
    process:Pool">
  <data className="com.company.model.LoggedAbsenceRecord" dataType="
      _ZuF28u7sEfCoIMCqMDCGog" name="rec1" ... />
  <data className="com.company.model.AbsenceRequest" dataType="
      _ZuF28u7sEfCoIMCqMDCGog" name="req1" ... />
  <data className="com.company.model.LineManager" dataType="
      _ZuF28u7sEfCoIMCqMDCGog" name="mgr1" ... />
  <!-- ... -->
</elements>
```

Listing 7.25: Pool variable declaration for for the `AbsenceRequest` process.

Another difference in Bonita is that human tasks interacting with forms require an explicit *contract* that defines expected inputs. The generator, therefore, extends the human (user) tasks in the BPMN file by inserting contract structures. An abbreviated example is shown below.

```
<contract xmi:type="process:Contract">
  <inputs name="req1Input" type="COMPLEX" dataReference="req1" xmi:type="
      process:ContractInput">
    <inputs name="requestId"  type="INTEGER" xmi:type="process:ContractInput"/
        >
    <inputs name="employeeId" type="INTEGER" xmi:type="process:ContractInput"/
        >
    <inputs name="absenceType"              xmi:type="process:ContractInput"/>
    <inputs name="startDate"                xmi:type="process:ContractInput"/>
    <!-- ... -- >
  </inputs>
</contract>
```

Listing 7.26: Bonita contract structure for `Request Absence` human task (abbreviated).

Furthermore, Bonita does not implicitly propagate form input values into pool variables. Instead, explicit *operations* must be generated. The generator therefore emits operation

[20] `https://documentation.bonitasoft.com/bonita/latest/pages-and-forms/forms`

nodes (Groovy/Java-method-based) that perform assignments such as `req1.setRequestId(req1Input.requestId)`.

```
<operations xmi:type="expression:Operation">
  <leftOperand  content="req1" type="TYPE_VARIABLE" xmi:type="
      expression:Expression"/>
  <rightOperand content="req1Input?.requestId" interpreter="GROOVY" xmi:type="
      expression:Expression"/>
  <operator expression="setRequestId" type="JAVA_METHOD" xmi:type="
      expression:Operator">
    <inputTypes>java.lang.Integer</inputTypes>
  </operator>
</operations>
```

Listing 7.27: Bonita operation mapping contract input to business object field.

REST and Email connectors Bonita uses `connectors` for service tasks to allow for integration with external systems. The configuration code for the connectors is generated and inserted straight into the BPMN file. An example of a REST connector is shown below:

```
  <connectors definitionId="rest-post" ...>
  <configuration definitionId="rest-post" ...>
    <parameters key="url" ...>
      <expression content="https://httpbin.org/post" .../>
    </parameters>
    <parameters key="contentType" ...>
      <expression content="application/json" ...>
    </parameters>
    <parameters key="body" ...>
      <expression content="{&quot;recipient&quot;:&quot;req1.employeeId&quot
          ;,&quot;title&quot;:&quot;Absence Request Update&quot;,&quot;content
          &quot;:&quot;Dear ${req1.employeeId}, your absence request #${req1.
          requestId} has been ${req1.status}. Thank you for your submission.&
          quot;}"...>
        <referencedElements content="req1.employeeId" interpreter="GROOVY"
            name="req1.employeeId" ...>
          <referencedElements content="req1" name="req1" returnType="com.
              company.model.AbsenceRequest" type="TYPE_VARIABLE"...>
            <!-- ... -->
```

Listing 7.28: Bonita REST connector configuration.

A similar approach is used for email connectors, where the template is generated and inserted straight into the BPMN file. As an alternative to REST and Email connectors, raw Groovy code could be used instead.

Expressions Unlike Camunda, Bonita does not support UEL, so expressions are passed into Groovy. In situations where process variables are used in the original UEL expressions with markers, the resolution is much simpler. Unlike Camunda 7, Bonita supports nested objects natively, so UEL expressions such as `${~req1.status~ == "NEEDS_CLARIFICATION"}` are parsed into `req1.status == "NEEDS_CLARIFICATION"`. For example:

```
  <condition automaticDependencies="false" content="req1.status == &quot;
      NEEDS_CLARIFICATION&quot;" ...>
    <!-- ... -->
</condition>
```

Listing 7.29: Bonita expression resolution for `req1.status` process variable.

Decision logic Decision logic in Bonita can be implemented either using Condition Decision Tables[21] or raw Groovy code. This implementation uses Groovy code that is generated and inserted straight into the BPMN file. Since business rule tasks are not supported in Bonita, these are mapped to `xmi:type="process:Activity"` generic task nodes.

BPMN The BPMN process model is emitted in a Bonita-specific format, `.proc`, which is Bonita's own process definition format used by Bonita Studio/Engine. Lots of additions to the model are already described in the previous paragraphs. Prior to emission, nodes are additionally updated with the engine-specific code (if any) from the `config.json` file. Additional engine-specific metadata is also added to the XML so it is ready for deployment.

Output structure The output is written under `out/bonita`. A typical output directory contains the following files:

```
out/bonita/
   approveAbsenceForm/
      approveAbsenceForm.json
      assets/...
   provideFurtherExplanationForm/...
   requestAbsenceForm/...
   _ZvkdoO7sEfCoIMCqMDCGog.conf (process configuration)
   .index.json (forms index)
   bom.xml (business object model)
   diagram_1-1.0.proc (process definition)
   Organization.xml (organization file)
```

Listing 7.30: `AbsenceRequest process` output directory structure for Bonita.

7.5 Discussion and summary

This chapter described a concrete implementation of the pipeline introduced earlier in chapters 4, 5, and 6. A concrete way of modelling information requirements was shown in Section 7.1. Validation of models was described in Section 7.3. Engine-specific artifact generation was described in Section 7.4. Two BPMS were targeted: Camunda 7 and Bonita.

Ovearll, artifact generation was successfully implemented for both engines, without any compromises. All information captured in the vendor-neutral model set could be mapped to the 2 targeted BPMS. Evaluation of the implementation is covered in the following chapter.

[21] https://documentation.bonitasoft.com/bonita/latest/process/transitions#_condition_decision_table

8 Evaluation

This chapter evaluates the thesis artifact: a vendor-neutral modeling, validation, and artifact-generation pipeline that produces engine-specific deployment bundles from a platform-independent model set. The evaluation goal is to assess whether the artifact achieves its intended objective in practice, namely, generating deployable and executable artifacts for multiple BPMS engines.

Because the contribution is an engineering artifact, the evaluation is conducted as end-to-end demonstrative validation consistent with Design Science Research (DSR). Representative processes are modeled using the platform-independent approach, passed through the prototype pipeline, and the generated outputs are deployed and executed on the target engines.

8.1 Evaluation design

8.1.1 Methodological framing

The evaluation follows a design-oriented validation approach consistent with Design Science Research (DSR) [20] where knowledge is produced by building a purposeful artifact and evaluating it against its objective. In this thesis, the artifact is the pipeline described in chapters 5 to 7. As a result, the evaluation focuses on whether the artifact works as intended in realistic usage scenarios.

The selected evaluation method is demonstrative. The pipeline is applied to multiple case processes, and success is determined by whether the generated artifacts can be deployed and executed on two substantially different engines. This aligns with DSR evaluation forms such as demonstration and functional testing.

8.1.2 Evaluation questions

The evaluation is organized around the following questions:

- **EQ1 (Executability):** Can the pipeline generate engine-specific artifacts that deploy without manual correction of the generated core definitions?
- **EQ2 (Semantic preservation):** Does runtime behavior match the intended process logic expressed in the platform-independent models?
- **EQ3 (Completeness):** Does the platform-independent model set capture sufficient execution-relevant information to produce deployable artifacts (forms, variables, assignments, decisions, integrations) for the targeted feature subset?
- **EQ4 (Portability):** Can the same platform-independent source models be used to generate artifacts for multiple engines?
- **EQ5 (Reduced coupling):** Are engine-specific execution details isolated within the generation layer, rather than embedded throughout the platform-independent core models?

8.1.3 Operational criteria

8.1.4 Evaluation protocol

For each case process, the following end-to-end protocol is applied:

1. **Create the platform-independent model set**: This is done manually using appropriate modeling tools.

2. **Run the Java implementation**: The modelling set is manually passed to the Java implementation. The output is engine-specific deployment files.
3. **Packaging:** Deployment files are manually copied into the appropriate engine project files.
4. **Deployment:** Deployment is done using the standard engine mechanisms.
5. **Execution:** Start the process and execute representative paths.

8.2 Evaluation setup and case selection

8.2.1 Target engines

Two BPMS were selected: Camunda 7 and Bonita. The choice is deliberate because the engines differ significantly in paradigm and deployment model. They are not forks of each other.

Camunda 7 is developer-centric and designed to be embedded into custom applications. It is typically used by developers who integrate the process engine directly into Java-based systems, implement service tasks as Java classes or delegates, and manage deployment through build pipelines and application packaging.

In contrast, Bonita follows a studio-centric and model-driven paradigm. It provides an integrated development environment where business objects, organization models, contracts, and forms can be defined through graphical interfaces. Many artifacts, such as the Business Object Model (BOM) and organization definitions, are generated and managed by the platform itself and are not typically edited manually.

Demonstrating successful generation and execution across both engines therefore provides evidence that the proposed approach is not narrowly coupled to a single vendor paradigm, but is capable of supporting both developer-centric and studio-centric execution environments.

- Camunda Platform 7 (Community Edition) 7.24.0
- Bonita Community Edition: 2025.2

8.2.2 Case origin and selection rationale

Four case processes were modeled and executed. The case processes originate from two categories:

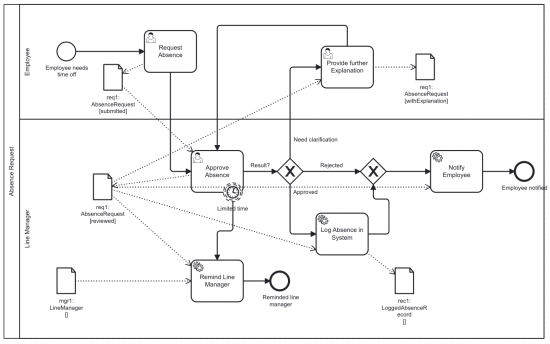
- **Examples from vendor documentation**. These were used to ensure that the pipeline can handle modeling patterns commonly presented as “typical” by vendors and that resemble realistic starter processes. 3 processes were picked: `AbsenceRequest`[1], `LoanRequest`[2], and `ErrorHandling`[3].
- **Real business context (UNDP case):** This case was used to validate the pipeline on a real organizational workflow (2nd step in UNDP 7-step process[4]). This case was derived from the author’s professional engagement at UNDP and represents an authentic organizational workflow.

[1] https://camunda.com/try-modeler/

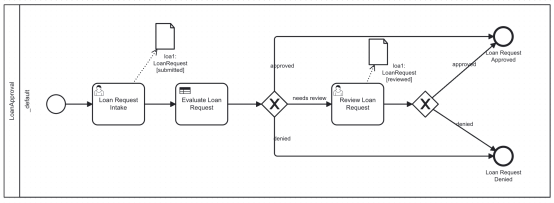
[2] https://documentation.flowable.com/latest/model/dmn/example/part1-decision-table#step-5c-exclusive-gateway

[3] https://camunda.com/try-modeler/

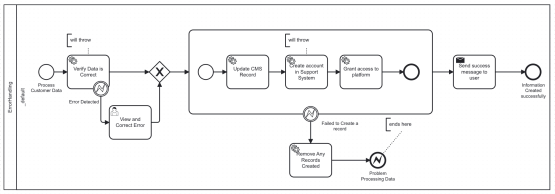
[4] https://www.undp.org/smart-facilities/publications/undp-7-step-process-smart-and-green-energy-solutions-implementation



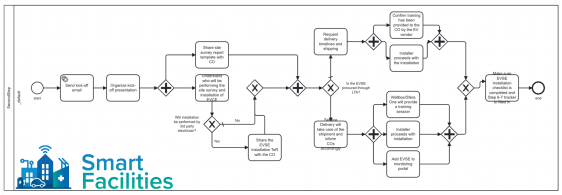
(a) Case1: AbsenceRequest



(b) Case2: LoanRequest



(c) Case3: ErrorHandling



(d) Case4: SecondStep

Figure 8.1: Evaluation case process diagrams: AbsenceRequest, LoanRequest, ErrorHandling, and SecondStep (UNDP).

8.2.3 Scenario categories covered

The BPMN 2.0 specification defines a comprehensive set of modeling constructs [3]. This thesis does not claim complete BPMN coverage. Instead, the evaluation focuses on an execution-relevant subset commonly used in automation- oriented process models. The matrix below shows BPMN constructs covered by each case:

Table 8.1: BPMN constructs covered by evaluation cases.

BPMN construct	Case 1	Case 2	Case 3	Case 4
Start Event	Y	Y	Y	Y
Timer Boundary Event (Non-Interrupting)	Y	N	N	N
Error Boundary Event (Non-Interrupting)	N	N	Y	N
End Event	Y	Y	Y	Y
Error End Event	N	N	Y	N
User task	Y	Y	Y	N
Service task	Y	Y	Y	Y
Business rule task	N	Y	N	N
Send task	N	N	Y	N
Task (abstract)	N	N	N	Y
Expanded Subprocess	N	N	Y	N
Exclusive Gateway	Y	Y	Y	Y
Parallel Gateway	N	N	N	Y
Normal (Sequence) Flow	Y	Y	Y	Y
Conditional (Sequence) Flow	Y	Y	Y	Y
Default (Sequence) Flow	N	N	N	Y
Exception Flow	N	N	Y	N
Association	Y	Y	Y	N
Pool (with Process)	Y	Y	Y	Y
Lane	Y	Y	Y	Y
Data Object	Y	Y	Y	N
Text Annotation	N	N	Y	N

Beyond BPMN control-flow constructs, the evaluation also covers execution-level artifacts and integration mechanisms relevant for automation scenarios. These are summarized in Table 8.2.

Table 8.2: Execution and integration features covered by evaluation cases.

Feature	Case 1	Case 2	Case 3	Case 4
Form for user tasks	Y	Y	Y	N
Process variables, business data, states	Y	Y	Y	Y
Decision tables (DMN)	N	Y	N	N
REST integration	Y	N	N	N
Email integration	Y	N	N	N
Organization mapping	Y	Y	Y	Y
Use of expressions	Y	Y	Y	N

8.2.4 Scope boundaries and non-evaluated constructs

This evaluation does not cover all BPMN constructs defined in the full BPMN 2.0 specification (as scoped in Section 4.3.2) The following constructs were not empirically evaluated:

- Groups
- Some event types (e.g., Cancel, Compensation, Conditional, Signal, etc.)
- Some gateway types (e.g., Event-based Gateway, Inclusive Gateway, etc.)
- Tasks with multiple instances (Parallel, Sequential, etc.)

Minor syntactic variations Certain additional event types and boundary-event variants were not explicitly tested. However, these follow structural mapping principles already demonstrated for the evaluated subset and are therefore considered incremental extensions rather than architectural limitations.

While extending support to the above constructs is technically feasible, such extensions remain future work and are not empirically validated in this evaluation.

8.3 Results

This section reports the outcomes of executing the protocol in Section 8.1.4 for each case and each engine.

For each case, the full protocol was followed successfully. Making the model set and running the Java implementation occured similary for both engines. In the packaging step, files had to be manually copied into the appropriate locations depending on the engine. For the deployment step, in case for Camunda, the entry point of the application had to be update to refernce the id of the BPMN process file. In case for Bonita, organization and bom files had to be manually deployed (a button click) from the Bonita Studio interface (so that Bonita generates additional project files), before the process could be started.

8.3.1 Deployment and execution outcomes

The generated artifacts were deployed to both engines using their standard deployment mechanisms. For each (case, engine), the process instance could be started and representative paths could be executed:

- Completion of user tasks through generated/linked forms
- Gateway routing and alternative branches
- Variable creation and propagation across tasks
- Decision evaluation using decision logic (DMN and Groovy script)
- REST API calls
- Email sending
- Timer and error behavior
- Process completion

8.3.2 Relation to evaluation questions

- Executability (EQ1) - Satisfied: For each case and engine, the generated artifacts were deployed and executed without manual correction of the generated core definitions.
- Semantic preservation (EQ2) - Satisfied: For each case and engine, the observed runtime behavior matched the intended control-flow and data-flow for the exercised scenarios.
- Completeness (EQ3) - Satisfied: For each case and engine, the platform-independent model set captured sufficient execution-relevant information to produce deployable artifacts for the targeted feature subset. There was no need to manually edit or add any additional artifacts to make the deployment work.
- Portability (EQ4) - Satisfied: For each case and engine, the vendor-independent source models were the same when generating artifacts for both engines.

- Reduced coupling (EQ5) - Partially satisfied: The `config.json` from the vendor-neutral model set allows defining engine-specific attributes for tasks, events, and lanes. This can be used in case the platform-independent model set is not expressive enough to capture and then map some of the information to the target BPMS. One might need to pass some raw vendor-specific extensions (e.g. `<camunda:executionListener>`) directly to final BPMN model. This was for instance used in Case 3, `Create account in Support System` task.

Overall, the evaluation demonstrates that the artifact generally achieves its intended objective in practice, namely, generating deployable and executable artifacts for multiple BPMS engines.

8.4 Threats to validity

Several threats and limitations should be acknowledged:

- **Engine coverage:** only two engines are evaluated. While they differ substantially, additional engines are required to strengthen generalizability.
- **Scope of BPMN and features:** the prototype targets a practical subset of BPMN constructs and execution requirements. Unsupported constructs are not evaluated.
- **Number of representative cases:** the evaluation included only four cases. Additional cases would provide more evaluation coverage and strength.
- **No user study:** the evaluation demonstrates technical feasibility but does not measure usability, adoption likelihood, or productivity impact for practitioners (e.g. possibly manual transition between BPMS is more optimal).
- **No formal semantic proof:** correctness is validated operationally via execution, not via formal equivalence proofs.

Further limitations and implications are discussed in Chapter 9.

8.5 Summary

This chapter evaluated the thesis artifact using an end-to-end demonstrative protocol consistent with Design Science Research. Four case processes were modeled using the platform-independent approach and processed through the prototype pipeline, producing engine-specific deployment bundles for Camunda 7 and Bonita. Overall, the evaluation demonstrates that the artifact generally achieves its intended objective in practice, namely, generating deployable and executable artifacts for multiple BPMS engines.

9 Discussion

This chapter interprets the evaluation results from Chapter 8.

9.1 Interpretation of results

The evaluation demonstrated that the proposed vendor-neutral modeling and generation framework is technically feasible. For the selected case processes, platform-independent models were successfully transformed into deployable artifacts for both Camunda 7 and Bonita. The generated artifacts could be deployed and executed without manual correction of core process definitions.

Relation to EQ1: Executability Generated artifacts can indeed be deployed and executed without manual correction of the core definitions. However, this finding is scoped to the specific Java-based implementation in which engine-specific artifacts are generated programmatically. Alternative implementations (e.g., using M2M or M2T transformation frameworks) were not evaluated. In addition, the evaluation did not include very large or highly complex processes that use all possible BPMN constructs. For example, none of the 4 evaluate cases included BPMN Signal events. Executing the protocol on such a case would possibly reveal issues (or not).

Relation to EQ2: Semantic preservation The evaluation indicates that runtime behavior matches the intended process logic expressed in the platform-independent models. At the same time, processes using BPMN constructs not covered by the evaluation may not exhibit the same guarantees. For example, Event-based Gateways were not evaluated. Furthermore, larger and more complex processes may require automated testing or simulation tools to systematically verify all execution paths. Such techniques were beyond the scope of this thesis.

Relation to EQ3: Completeness Completeness is satisfied within the evaluated scope. However, the same limitations discussed for EQ1 and EQ2 apply.

Relation to EQ4: Portability Only two engines were evaluated. Extending the framework to additional vendors appears plausible but requires careful qualification. Engines that are developer-oriented, API-accessible, file-based in deployment, and well-documented in terms of artifact structure are likely easier to support (e.g., Flowable). Engines that rely heavily on proprietary studio environments or tightly integrated web-based tooling may require alternative integration strategies. In principle, artifact formats could be reverse-engineered, exposed APIs could be used, or scripted automation could be implemented. However, such integration would require a detailed study of each vendor's architecture and constraints. Therefore, while the conceptual framework is generalizable, concrete support for additional engines must be evaluated individually. This thesis does not claim universal compatibility.

Relation to EQ5: Reduced coupling Overall, the artifact demonstrates reduced coupling between process models and specific engines. The ability to define engine-specific attributes in the configuration file represents a pragmatic trade-off between strict platform independence and practical support for engine-specific execution metadata.

Relation to RQ1.1: How can business requirements be systematically captured and represented using complementary modeling languages? The evaluation implicitly validates the information-requirements list and modeling choices. The models contained sufficient execution-relevant information to enable deployment on the target engines for the evaluated cases. This supports the claim that a BPMN-only model is insufficient for portability and that complementary models are necessary to bridge the execution gap.

Relation to RQ1.2: How can synchronization and consistency between models be defined and maintained? Similarly, the evaluation implicitly validates the synchronization and consistency mechanisms between models, as successful deployment and meaningful execution were achieved without manual correction. This suggests that the defined linking and validation strategy is sufficient within the evaluated scope.

Relation to RQ1.3: How can platform-independent models be transformed into engine-specific execution artifacts? The evaluation demonstrates that the platform-independent models can be transformed into engine-specific execution artifacts. This is done by the Java implementation.

9.2 Limitations

Several limitations should be acknowledged in context of evaluation.

Engine coverage As noted earlier, only two engines were evaluated. Although they differ substantially, additional engines would be required to strengthen claims of generalizabilit

Scope of BPMN and features The prototype targets a practical/commonly used subset of BPMN constructs and information requirements. Many BPMN constructs were not evaluated. The definition of what is “practical/commonly used” is inherently subjective. Although multiple BPMS systems were analyzed in the comparative analysis (Section 4.1), certain details may have been overlooked. It is also difficult to determine which constructs are most frequently used in practice, as organizations typically do not publicly disclose detailed process implementations. Such analysis would require large-scale empirical research on its own.

At the same time, the results of evaliation should not be interpreted as proving broad industrial portability for arbitrary processes or for the full BPMN 2.0 specification. Rather, the evaluation demonstrates portability within tested subset of constructs and within the information requirements that can be covered by the vendor-neutral modelling set.

Number of representative cases The evaluation included four cases. Although they are representative (including both documentation and real business cases, multiple BPMN constructs, and information requirements), additional, fully documented deployment cases (not just BPMN models) would strengthen the evaluation. They would reveal potential gaps that vendor-neutral modelling set and modelling guidelines do not cover.

No user study The evaluation did not include a real user study involving people (e.g., assessing usability). This omission was primarily due to time constraints. As a result, the thesis demonstrates technical feasibility but does not provide empirical evidence regarding usability for non-technical modelers, productivity gains, or long-term maintainability in real organizations. These questions remain open for future investigation.

9.3 Tradeoffs

The proposed approach involves deliberate trade-offs. These are discussed in the following sections.

9.3.1 Expressiveness vs. portability

Portability across engines requires constraining the range of information captured in the vendor-neutral modeling set. The thesis therefore prioritizes representations that can be parsed/transformed predictably (e.g., UEL for expressions, JSON artifacts validated by JSON Schema, explicit references). The trade-off is that more expressive mechanisms such as expressive, general-purpose scripting languages (e.g., Groovy) are intentionally excluded because they are difficult to translate reliably across engines.

9.3.2 Multi-model decomposition vs. modeling overhead

Separating execution-relevant information into multiple complementary models increases modeling effort and repository complexity. Users must maintain BPMN, business data models, schemas, etc. However, this decomposition provides two benefits. First, each model can be validated using appropriate techniques (e.g., schema validation for forms). Second, engine-specific coupling is avoided in the source BPMN. In practice, the approach trades a single, vendor-specific modeling experience for a modular, tool-agnostic model set.

9.3.3 Determinism vs. flexibility

A generation pipeline benefits from deterministic conventions such as strict naming, and explicit linking. This makes the pipeline reproducible and improves error reporting. The trade-off is reduced flexibility in ad-hoc modeling styles. For instance, if a modeling team prefers informal task names or implicit assumptions, the pipeline will require additional discipline from team members to follow the guidelines.

9.3.4 Upfront modeling effort vs. downstream benefits

A central tradeoff of the proposed approach concerns upfront modeling effort versus downstream portability. The initial modeling effort is high and demands discipline, maintenance of multiple models. However, this upfront cost yields several downstream benefits, such as reduced vendor lock-in, a centralized and consistent source of truth, automated artifact generation. The tradeoff may be particularly acceptable for organizations planning migration between engines, using multi-engine environments, or having teams with sufficient technical maturity to maintain structured models.

9.4 Generalizability

9.4.1 Generalizability to other BPMS engines

The evaluation covered 2 engines. The conditions and limitations for extension to additional vendors are already discussed in Section 9.1 under EQ4. In summary, the conceptual framework is generalizable, but concrete support for additional engines must be evaluated individually. This thesis does not claim universal compatibility.

Process types and organizational context

The pipeline is most suitable for processes with a clearly defined business data model and explicit lifecycle states. Highly ad-hoc processes or strongly UI-driven workflows with minimal formal structure may benefit less from this modeling strategy. For example, the current modeling strategy for forms does not support detailed UI/UX specifications such as layout or styling. In such cases, additional manual customization would be required.

9.5 Lessons learned

- BPMN control flow is not the portability barrier. Portability challenges arise mainly at the level of execution metadata (e.g., form bindings, task assignments, integration configuration). Making these concerns explicit artifacts exposes coupling and makes it manageable.
- Validation is a first-class requirement in multi-model automation. Without validation, generation may succeed syntactically but produce deployment bundles that fail at runtime or behave incorrectly.
- The BPMS engines industry is highly diverse. Although most follow the same BPMN standard, other features are implemented differently. Comparing BPMS engines is a huge task on its own. Defining a universal, practical, information requirements catalogue is a compromise between completeness and practicality.
- The implementation of the artifact generation part of the pipeline differs a lot on the target BPMS. Adding support for a new engine requires careful analysis before implementation.

9.6 Summary

The thesis demonstrates that a vendor-neutral modeling and generation framework is technically feasible. For the selected case processes, platform-independent models were successfully transformed into deployable artifacts for both Camunda 7 and Bonita. The generated artifacts could be deployed and executed without manual correction of the generated core definitions.

The results should, however, be interpreted within the defined scope of the evaluation. Portability is demonstrated for a specific subset of BPMN constructs, information requirements, and two target engines. The findings therefore provide empirical support for the feasibility of a vendor-neutral modeling and generation strategy, rather than proof of universal portability across all BPMS platforms or all possible process types.

10 Conclusions and Future Work

This chapter concludes the thesis by restating the main contribution and result, summarizing the implemented artifact, highlighting key findings from the evaluation and discussion, giving explicit answers to the research questions, and outlining a prioritized roadmap for future work.

10.1 Main contribution and result

Business process automation platforms often combine BPMN control flow with execution-relevant configuration in engine-specific formats and tool ecosystems. As a result, process solutions are often difficult to migrate across BPMS engines without re-implementing configuration, integration logic, and deployment packaging, which contributes to vendor lock-in.

This thesis showed that a substantial part of execution-relevant process configuration can be captured in a structured, platform-independent model set and automatically transformed into engine-specific deployment artifacts. More specificaly ,the thesis proposed and implemented a vendor-neutral modeling, validation, and artifact-generation pipeline that (i) captures execution requirements beyond BPMN in complementary,
machine-processable models, (ii) enforces solo and cross-model consistency between models, and (iii) generates deployable bundles for different BPMS. The evaluation in Chapter 8 and the interpretation in Chapter 9 demonstrate technical feasibility of the pipeline (and its implementation), while also making explicit the boundaries within which the results should be interpreted.

10.2 Artifact summary

The pipeline is the main artifact of the thesis.

10.2.1 Platform-independent model set

An information requirements catalogues was formed based on a comparative analysis of representative BPMS engines. These information requirements are then captured in a platform-independent model set. The pipeline provides guidelines for how to model each information requirement. The model set consists of:

- **BPMN model** as the primary notation for control flow and data contracts.
- **A configuration file (JSON)** for linking BPMN elements to complementary artifacts (e.g., forms, REST/email config files).
- **Business data model (UML)** for capturing the type of data that flows through the process.
- **Data states model (JSON)** for encoding minimal lifecycle constraints (e.g., required fields per state).
- **Forms (JSON Schemas)** for capturing the structure of the data that is collected from the user.
- **Organization (ArchiMate)** for capturing the organization structure that can later be used for e.g. task assignments.

- **Expression (UEL)** as a vendor-neutral expression language that can be used e.g. for conditional statements and variable access.
- **Decision logic (DMN)** for separating decision logic from orchestration.
- **Email (JSON + FTL)** for capturing the email template and configuration used for sending emails.
- **REST (JSON)** for making REST calls to external systems.

The key principle is that each artifact has a clear responsibility and provides stable linking points so that cross-model references are machine-resolvable and can be validated before generation.

10.2.2 Validation and generation pipeline

After the platform-independent model set is created, the pipeline validates and generates engine-specific artifacts. An implementation of the pipeline would consist of the following steps:

1. **Load and parse** all platform-independent artifacts into internal representations.
2. **Validate syntax and standalone semantics** of each artifact type (“solo” checks).
3. **Validate cross-model consistency** (“multi” checks) such as referential integrity, alignment of variables, state constraints, and linking correctness.
4. **Enrich** internal representations (e.g. resolving references that can be statically resolved) for the last time in a vendor-neutral way.
5. **Generate and emit** engine-specific artifacts

The outcome is a deployable baseline for a target engine, i.e., a bundle that captures core execution behavior and can be deployed and executed.

10.3 Answers to the research questions

10.3.1 RQ1.1: How can business requirements be systematically captured and represented using complementary modeling languages?

Business requirements can be captured systematically by deriving an evidence-based information-requirements catalogue from a comparative analysis of BPMS engines and then assigning a suitable modelling language for each information requirement.

This was described in Chapter 4.

10.3.2 RQ1.2: How can synchronization and consistency between models be defined and maintained?

Synchronization and consistency can be maintained by defining explicit linking rules between artifacts and enforcing them via validation that checks both standalone and cross-model consistency.

This was described in Chapter 5.

10.3.3 RQ1.3: How can platform-independent models be transformed into engine-specific execution artifacts?

Platform-independent models can be transformed into engine-specific execution artifacts by combining a deterministic parsing and enrichment stage that produces consistent internal representations, and an engine-specific mapping and emission stage that produces

BPMS-specific artifacts. In this thesis, the generation stage is a MDE-style pipeline (M2M + M2T).

This was described in Chapter 6.

10.3.4 RQ1 Overall: Can business processes be modeled and executed in a platform-independent way that reduces engine-specific coupling and enables automated deployment across multiple BPMS engines?

Yes, within clearly defined scope boundaries (Section 10.5).

10.4 Contributions

- **Vendor-neutral pipeline for capturing execution requirements in a platform-independent model set, validating and generating engine-specific artifacts.** This pipeline is the main artifact of the thesis.
- **Prototype pipeline implementation.** A concrete end-to-end implementation in Java that loads, validates, enriches, and generates artifacts from the platform-independent model set.
- **Comparison of BPMS engines.** Multiple BPMS were compared in terms of feature support and how they follow the BPMN standard. This resulted in a comparison table - (Appendix B).

10.5 Scope boundaries

The evaluation (Chapter 8) demonstrates *technical feasibility* of the pipeline. However, it does not include empirical user studies. Therefore, productivity impact, adoption barriers, are not quantified. Additionally, the implementation supports a subset of BPMN constructs sufficient for the evaluated case processes, and does not claim full BPMN 2.0 coverage. Only two BPMS engines were implemented and evaluated. Although they differ substantially, broader generalizability requires additional engine implementations and comparative validation. These boundaries motivate the roadmap below.

10.6 Future work roadmap

- **Further evaluation.** The evaluation should be extended to more BPMS engines and more case processes. This would suggest existing gaps in the information requirements catalogue and the model set.
- **Evaluation with user studies.** A user study should be done to quantify the productivity impact, adoption barriers, and other relevant metrics.
- **Further BPMS and industry analysis.** More detailed BPMS analysis should be done to futher refine the information requirements set for a better balance between completeness and practicality. Futher analysis should also be done on what kind of processes are most frequently used in practice, what are the most common features used, what patterns exist, etc.
- **Turning the prototype into a tool.** The prototype could be further developed into a production-ready tool with improved usability.

10.7 Closing remarks

In conclusion, this thesis contributes to the field of business process automation by providing a vendor-neutral pipeline for capturing execution requirements in a platform-independent

model set, validating and generating engine-specific artifacts.

Bibliography

[1] Wil M. P. van der Aalst. “Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management”. In: *Lectures on Concurrency and Petri Nets*. Ed. by Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg. Vol. 3098. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 1–65. DOI: 10.1007/978-3-540-27755-2_1. URL: https://link.springer.com/chapter/10.1007/978-3-540-27755-2_1.

[2] *Workflow Management Coalition Terminology & Glossary*. Specification WFMC-TC-1011. Workflow Management Coalition (WfMC), Feb. 1999. URL: https://wfmc.org/wp-content/uploads/2022/09/TC-1011_term_glossary_v3.pdf.

[3] Object Management Group. *Business Process Model and Notation (BPMN) — Version 2.0.2 (Formal Specification)*. OMG Document: formal/2013-12-09. Dec. 2013. URL: https://www.omg.org/spec/BPMN/2.0.2/PDF.

[4] Christopher Drews and Birger Lantow. “A Framework for Effort Estimation in BPMS Migration”. In: *Proceedings of the CEUR Workshop on Business Processes and Services (BPMS) – Vol. 1898*. CEUR Workshop Proceedings, 2017, p. 18051. URL: https://ceur-ws.org/Vol-1898/paper7.pdf.

[5] Giuseppe De Giacomo et al. “Linking Data and BPMN Processes to Achieve Executable Models”. In: *Advanced Information Systems Engineering (CAiSE 2017)*. Vol. 10253. Lecture Notes in Computer Science. Springer, 2017, pp. 612–628. DOI: 10.1007/978-3-319-59536-8_38.

[6] José Miguel Pérez-Álvarez et al. “Verifying the manipulation of data objects according to business process and data models”. In: *Knowledge and Information Systems* 62 (2020), pp. 2653–2683. DOI: 10.1007/s10115-019-01431-5.

[7] Justice Opara-Martins, Reza Sahandi, and Feng Tian. “Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective”. In: *Journal of Cloud Computing* 5 (2016), p. 4. DOI: 10.1186/s13677-016-0054-z.

[8] Steve Easterbrook and Bashar Nuseibeh. “Using ViewPoints for inconsistency management”. In: *Software Engineering Journal* 11.1 (1996), pp. 31–43. DOI: 10.1049/sej.1996.0004.

[9] George Spanoudakis and Andrea Zisman. “Inconsistency Management in Software Engineering: Survey and Open Research Issues”. In: *Handbook of Software Engineering and Knowledge Engineering, Volume I: Fundamentals*. Ed. by S. K. Chang. World Scientific, 2001, pp. 329–380. DOI: 10.1142/9789812389718_0015.

[10] Montserrat Estañol. “Artifact-centric Business Process Models in UML: Specification and Reasoning”. PhD thesis. Universitat Politècnica de Catalunya, 2016.

[11] Tom Mens and Pieter Van Gorp. “A Taxonomy of Model Transformation”. In: *Electronic Notes in Theoretical Computer Science* 152 (2006), pp. 125–142. DOI: 10.1016/j.entcs.2005.10.021.

[12] Jean Bézivin. “Model Driven Engineering: An Emerging Technical Space”. In: *Generative and Transformational Techniques in Software Engineering*. Ed. by Holger Giese. Vol. 4143. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 36–64. DOI: 10.1007/11877028_2. URL: https://link.springer.com/chapter/10.1007/11877028_2.

[13] Iqra Zafar et al. “A Novel Framework to Automatically Generate Executable Web Services From BPMN Models”. In: *IEEE Access* 7 (2019), pp. 93653–93677. DOI: 10.1109/ACCESS.2019.2927785.

[14] Chun Ouyang et al. “From BPMN Process Models to BPEL Web Services”. In: *Proceedings of the 4th IEEE International Conference on Web Services (ICWS 2006)*. IEEE, 2006, pp. 285–292. DOI: 10.1109/ICWS.2006.67.

[15] Adrian Mos and Thierry Jacquin. “A Platform-Independent Mechanism for Deployment of Business Processes Using Abstract Services”. In: *17th IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW 2013)*. IEEE, 2013, pp. 71–78. DOI: 10.1109/EDOCW.2013.25.

[16] Estefanía Díaz et al. “Generating Graphical User Interfaces from BPMN Models with Stereotypes: A Model-Driven Approach”. In: *Proceedings of the 11th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2019)*. 2019. DOI: 10.1145/3319499.3328242.

[17] Estrela Ferreira Cruz and António Miguel Rosado da Cruz. “Deriving Integrated Software Design Models from BPMN Business Process Models”. In: *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*. SCITEPRESS, 2018, pp. 571–582. DOI: 10.5220/0006852005710582.

[18] Maximilian König et al. “Data Objects with Variables in BPMN”. In: *Proceedings of the Best BPM Dissertation Award, Doctoral Consortium, and Demonstrations & Resources Forum co-located with the 22nd International Conference on Business Process Management (BPM 2024)*. Vol. 3758. CEUR Workshop Proceedings. 2024. URL: https://ceur-ws.org/Vol-3758/paper-29.pdf.

[19] Aljia Bouzidi, Nahla Haddar, and Kais Haddar. “Traceability and Synchronization Between BPMN and UML Use Case Models”. In: *Ingénierie des Systèmes d’Information* 24.2 (2019), pp. 215–228. DOI: 10.18280/isi.240214.

[20] Alan R. Hevner et al. “Design Science in Information Systems Research”. In: *MIS Quarterly* 28.1 (2004), pp. 75–105. DOI: 10.2307/25148625. URL: https://www.jstor.org/stable/25148625.

[21] Gartner Peer Insights. *Business Process Management Platforms Reviews and Ratings*. 2026. URL: https://www.gartner.com/reviews/market/business-process-management-platforms.

[22] Camunda. *Camunda Engine Evolution since Activiti Fork*. 2016. URL: https://camunda.com/blog/2016/10/camunda-engine-since-activiti-fork/.

[23] Flowable. *Flowable and Activiti: What the Fork?!* 2016. URL: https://www.flowable.com/blog/engineering/flowable-and-activiti-what-the-fork.

[24] Camunda. *User Task Assignment based on a DMN Decision Table*. 2020. URL: https://camunda.com/blog/2020/05/camunda-bpm-user-task-assignment-based-on-a-dmn-decision-table/.

[25] Alicia Martín-Navarro, María Paula Lechuga Sancho, and José Aurelio Medina-Garrido. “BPMS para la gestión: una revisión sistemática de la literatura”. In: *Revista Española de Documentación Científica* 41.3 (Sept. 2018), e213. ISSN: 0210-0614. DOI: 10.3989/redc.2018.3.1532. URL: http://dx.doi.org/10.3989/redc.2018.3.1532.

[26] CIB seven Documentation. *User Task Forms (Task Forms Documentation)*. 2026. URL: https://docs.cibseven.org/manual/2.0/user-guide/task-forms/.

[27] BPMN Model Interchange Working Group (MIWG). *BPMN Model Interchange (BPMN MIWG submission)*. 2015. URL: https://www.omgwiki.org/bpmn-miwg/lib/exe/fetch.php?media=20150611_submission.pdf.

[28] Alicia Maria Martín Navarro et al. “Is User Perception the Key to Unlocking the Full Potential of Business Process Management Systems (BPMS)?” In: *Journal of Organizational and End User Computing* 37.1 (2025). ISSN: 1546-2234. DOI: https://doi.org/10.4018/JOEUC.364099. URL: https://www.sciencedirect.com/science/article/pii/S1546223425000115.

[29] CIB seven Documentation. *Unified Expression Language (UEL) in BPMN Conditions*. 2026. URL: https://docs.cibseven.org/manual/2.0/user-guide/process-engine/expression-language/unified-expression-language/.
[30] Camunda. *Using FEEL for Expressions (Part 1)*. 2020. URL: https://camunda.com/blog/2020/09/feel/.
[31] Object Management Group (OMG). *DMN — Decision Model and Notation (Version 1.3, About)*. 2021. URL: https://www.omg.org/spec/DMN/1.3/About-DMN.
[32] Wei Wang et al. "Business process and rule integration approaches—An empirical analysis of model understanding". In: *Information Systems* 104 (2022), p. 101901. ISSN: 0306-4379. DOI: https://doi.org/10.1016/j.is.2021.101901. URL: https://www.sciencedirect.com/science/article/pii/S0306437921001162.
[33] Camunda. *Email Integration for Processes (Camunda BPM Mail Extension)*. 2016. URL: https://camunda.com/blog/2016/06/camunda-bpm-mail/.
[34] Christian Pïëch et al. "A Survey on Business Process Model and Notation (BPMN) 2.0". In: *Proceedings of the 13th International Workshop on Modeling in Software Engineering (MISE 2021)*. Vol. 2622. CEUR Workshop Proceedings. CEUR-WS.org, 2021, pp. 1–17. URL: https://ceur-ws.org/Vol-2622/paper1.pdf.
[35] Eleni Itsou. "A Framework for Assessing Business Process Digitalization: A Case Study in the Greek Public Sector". MA thesis. Thessaloniki, Greece: University of Macedonia, 2023. URL: https://dspace.lib.uom.gr/bitstream/2159/29678/4/ItsouEleniMsc2023.pdf.
[36] Jana Koehler et al. "The Role of Visual Modeling and Model Transformations in Business-driven Development". In: *Electronic Notes in Theoretical Computer Science* 211 (2008). Proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006), pp. 5–15. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2008.04.025. URL: https://www.sciencedirect.com/science/article/pii/S1571066108002405.
[37] Object Management Group (OMG). *BPMN — Business Process Model and Notation (Version 2.0.2, About)*. 2014. URL: https://www.omg.org/spec/BPMN/2.0.2/About-BPMN.
[38] Object Management Group. *Unified Modeling Language (UML) — Version 2.5.1 (About)*. 2017. URL: https://www.omg.org/spec/UML/2.5.1/About-UML.
[39] JSON Schema community. *JSON Schema — Draft 2020-12*. 2020. URL: https://json-schema.org/draft/2020-12.
[40] rjsf-team. *react-jsonschema-form documentation*. 2026. URL: https://rjsf-team.github.io/react-jsonschema-form/docs/.
[41] The Open Group. *ArchiMate 3.2 Specification (Reference Cards / Personal PDF Edition)*. 2022. URL: https://www.opengroup.org/sites/default/files/docs/downloads/n221p.pdf.
[42] Eclipse Foundation / Jakarta EE. *Jakarta Expression Language Specification — Version 4.0*. 2020. URL: https://jakarta.ee/specifications/expression-language/4.0/jakarta-expression-language-spec-4.0.
[43] Object Management Group. *Decision Model and Notation (DMN) — Version 1.3 (Formal Specification)*. Publication date on OMG spec page: February 2021. Feb. 2021. URL: https://www.omg.org/spec/DMN/1.3/PDF.
[44] Apache FreeMarker. *Apache FreeMarker Manual*. 2026. URL: https://freemarker.apache.org/docs/index.html.
[45] Pete Resnick. *Internet Message Format*. RFC 5322. IETF, 2008. URL: https://datatracker.ietf.org/doc/html/rfc5322.

[46] OpenAPI Initiative. *OpenAPI Specification v3.1.0*. 2021. URL: https://spec.openapis.org/oas/v3.1.0.html.
[47] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. IETF, 2017. URL: https://datatracker.ietf.org/doc/html/rfc8259.
[48] W3C. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C Recommendation. 2008. URL: https://www.w3.org/TR/xml/.
[49] YAML Language Development Team. *YAML Ain't Markup Language (YAML) — Revision 1.2.2*. 2021. URL: https://yaml.org/spec/1.2.2/.
[50] Wolfgang Niedl et al. "Processing XForms in HTML5-Enabled Browsers". In: *Balisage: The Markup Conference*. 2013. URL: https://www.balisage.net/Proceedings/vol10/html/Niedl01/BalisageVol10-Niedl01.html.
[51] W3C. *XForms 1.0 Frequently Asked Questions*. 2003. URL: https://www.w3.org/MarkUp/Forms/Group/xforms-1-faq.html.
[52] Phil Hunt et al. *System for Cross-domain Identity Management: Protocol*. RFC 7644. IETF, 2015. URL: https://datatracker.ietf.org/doc/html/rfc7644.
[53] The Apache Software Foundation. *The Apache Groovy programming language*. 2026. URL: https://groovy-lang.org/.
[54] W3C. *XML Path Language (XPath) 3.1*. W3C Recommendation. 2017. URL: https://www.w3.org/TR/xpath-31/.
[55] Roy T. Fielding and Julian Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. IETF, June 2014. URL: https://datatracker.ietf.org/doc/html/rfc7231.
[56] B. M. Byrne. "The Use of UML Class Diagrams to Teach Object Oriented Analysis and Design". In: (2015). Used here for concise pedagogical definitions of aggregation vs. composition. URL: https://uhra.herts.ac.uk/id/document/16860.
[57] Alexander Egyed. "Instant consistency checking for the UML". In: *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, 2006, pp. 381–390. DOI: 10.1145/1134285.1134339.
[58] Sebastian J. I. Herzig and Christiaan J. J. Paredis. "A Conceptual Basis for Inconsistency Management in Model-based Systems Engineering". In: *Procedia CIRP* 21 (2014), pp. 52–57. DOI: 10.1016/j.procir.2014.03.192.
[59] Edward D. Willink. "A text model - Use your favourite M2M for M2T". In: *Proceedings of MODELS 2018 Workshops (MODELS 2018), Copenhagen, Denmark, October 14, 2018*. Ed. by Regina Hebig and Thorsten Berger. Vol. 2245. CEUR Workshop Proceedings. CEUR-WS.org, 2018, pp. 89–102. URL: https://ceur-ws.org/Vol-2245/ocl_paper_1.pdf.
[60] Frédéric Jouault and Ivan Kurtev. "Transforming Models with ATL". In: *Satellite Events at the MoDELS 2005 Conference*. Vol. 3844. Lecture Notes in Computer Science. Springer, 2006, pp. 128–138. DOI: 10.1007/11663430_14.

A Formal Specification of the Platform-Independent Model Set

A.1 Meta models

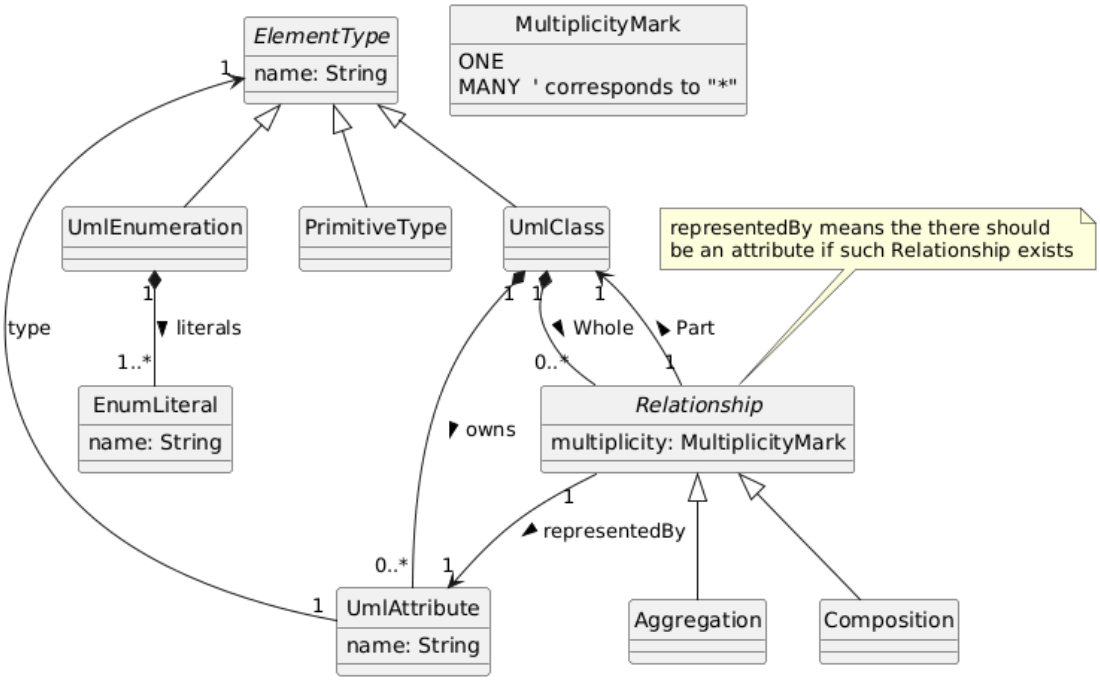


Figure A.1: Meta model for modeling business data

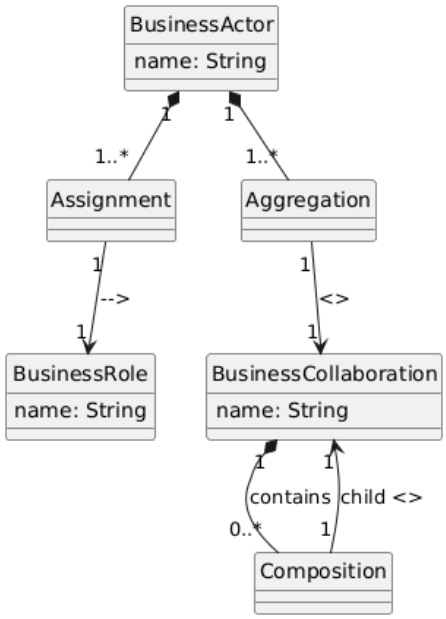


Figure A.2: Meta model for modeling organization

A.2 EBNF Grammars

```
DataObjectLabel ::= VariableName ":" TypeName StateTag ;

StateTag        ::= "[" StateName? "]" ;

VariableName    ::= SingularName | CollectionName ;

SingularName    ::= LowercaseLetter IdentifierRest ;
CollectionName  ::= UppercaseLetter IdentifierRest ;

TypeName        ::= UppercaseLetter IdentifierRest ;
StateName       ::= Identifier ;

Identifier      ::= Letter (Letter | Digit | "_")* ;

Letter          ::= "A".."Z" | "a".."z" ;
LowercaseLetter ::= "a".."z" ;
UppercaseLetter ::= "A".."Z" ;
Digit           ::= "0".."9" ;
```

Listing A.1: EBNF Grammar for data object labels

```
Expression      ::= "${" ExprBody "}" ;

ExprBody        ::= { ExprChunk } ;

ExprChunk       ::= OpaqueSegment
                  | ProcessVarRef
                  | GlobalRef ;

ProcessVarRef   ::= "~" ProcessVarPath "~" ;
GlobalRef       ::= "~globalVariables." GlobalKey "~" ;

ProcessVarPath  ::= Identifier { "." Identifier } ;
GlobalKey       ::= Identifier ;

OpaqueSegment   ::= { AnyChar - ("~" | "}") } ;
Identifier      ::= Letter (Letter | Digit | "_")* ;
Letter          ::= "A".."Z" | "a".."z" ;
Digit           ::= "0".."9" ;
```

Listing A.2: EBNF Grammar for markers used as part of UEL

A.3 JSON Schemas

JSON Schemas are uploaded separately with the report and are also available on GitHub:

- **Email validation schema:** https://github.com/pavliuc75/thesis_public/blob/main/JSON_schemas/email_validation_schema.json
- **REST call validation schema:** https://github.com/pavliuc75/thesis_public/blob/main/JSON_schemas/rest_validation_schema.json
- **Configuration file validation schema:** https://github.com/pavliuc75/thesis_public/blob/main/JSON_schemas/config_validation_schema.json
- **Business data states validation schema:** https://github.com/pavliuc75/thesis_public/blob/main/JSON_schemas/states_validation_schema.json

B BPMS Comparison Table

The `bpms_comparison.docx` file is uploaded separately with the report and is also available on GitHub: https://github.com/pavliuc75/thesis_public/blob/main/bpms_comparison_table.docx.

C Prototype implementation

Project files are available on GitHub. General instructions are listed in the README file on the main branch: https://github.com/pavliuc75/thesis_public/tree/main.

The project setups with the cases from the evaluation part are available on these branches:

- Case1: AbsenceRequest (the main and most representative case used throughout the report): https://github.com/pavliuc75/thesis_public/tree/absence_request
- Case2: LoanRequest: https://github.com/pavliuc75/thesis_public/tree/loan_request
- Case3: ErrorHandling: https://github.com/pavliuc75/thesis_public/tree/error_handling
- Case4: SecondStep: https://github.com/pavliuc75/thesis_public/tree/second_step

These 4 branches will be ”frozen” after the thesis is submitted. Later, I will likely add a new branch to continue expanding on the implementation part (outside of this thesis).

D Use of Generative AI

Generative AI was used to assist in working on the project.

For the report paper, it was mostly used for finding information and references (e.g., prompts: ”What other popular BPMS are out there besides Camunda? Give references”, ”is there some parseable way to represent UML diagrams?”). ChatGPT was used.

When working on the Java implementation, GitHub Copilot and ChatGPT were used. Copilot was used for line auto-completion (no prompts) as an assistant. ChatGPT was occasionally used for debugging, information search, and help documenting Java code (e.g., prompts: ”this is an error log after I added an email controller to the service task, what is wrong?” or ”how do I check if xml file is valid in Java?”.

Technical
University of
Denmark

Richard Petersens Plads, Building 324
2800 Kgs. Lyngby
Tlf. 4525 1700

https://www.compute.dtu.dk/