TU WIEN Informatics

Brücke zwischen DEMO und BPMN: Ein Metamodell-basierter Ansatz zur Synchronisation von Geschäftsvorgängen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Susanne Hirber, BSc
Matrikelnummer 11919162

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Henderik Proper, PhD

Wien, 28. Jänner 2025

Susanne Hirber	Henderik Proper

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Bridging DEMO and BPMN: A Metamodel-Based Approach for Synchronization of Business Transactions

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Business Informatics

by

Susanne Hirber, BSc
Registration Number 11919162

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Henderik Proper, PhD

Vienna, January 28, 2025

Susanne Hirber	Henderik Proper

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Susanne Hirber, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 28. Jänner 2025

Susanne Hirber

Danksagung

Ich möchte mich bei allen bedanken, die mich während meiner Masterarbeit unterstützt und begleitet haben.

Ein besonderer Dank gilt meinem Betreuer Henderik Proper, der mir bei der Ausarbeitung dieser Arbeit mit fachlichem Feedback, hilfreichen Rückmeldungen und klarer Orientierung sehr geholfen hat.

Ein herzliches Dankeschön geht außerdem an das Team hinter der Simplified Modeling Platform. Auf meine Fragen zur Nutzung der Plattform wurde stets schnell und freundlich reagiert, und ich habe mich bei der Arbeit mit dem System sehr gut unterstützt gefühlt.

Ich danke auch meiner Familie und meinen Freund:innen für ihre Unterstützung, nicht nur während dieser Masterarbeit, sondern während meines gesamten Studiums. Ihr habt mir immer den Rücken gestärkt, mich motiviert, wenn es schwierig wurde, und wart da, wenn ich euch gebraucht habe. Dafür bin ich sehr dankbar.

Acknowledgements

I want to thank everyone who supported and accompanied me during my master's thesis.

First and foremost, I would like to thank my supervisor, Henderik Proper, for his valuable feedback, clear guidance, and consistent support throughout the process of writing this thesis.

I would also like to thank the team behind the Simplified Modeling Platform. They always responded to my questions quickly and helpfully, and I felt well supported while working with the system.

Finally, I thank my friends and family for their support, not only during this thesis but throughout my entire studies. You have always had my back, encouraged me during difficult times, and were there whenever I needed you. I'm truly grateful for that.

Kurzfassung

In der Praxis werden häufig verschiedene Modellierungssprachen eingesetzt, um unterschiedliche Blickwinkel auf ein Unternehmen abzubilden. DEMO legt den Fokus auf die Verantwortung von Akteuren und deren gegenseitige Verpflichtungen, während BPMN beschreibt, wie Prozesse im Detail ablaufen. Obwohl beide Modelle oft denselben Geschäftsbereich betreffen, werden sie meist getrennt voneinander gepflegt. Das führt schnell zu Inkonsistenzen und einem hohen manuellen Pflegeaufwand.

Ziel dieser Arbeit ist es, einen Ansatz zu entwickeln, der DEMO- und BPMN-Modelle automatisch synchron hält. Anders als bei klassischen Modelltransformationen, die nur in eine Richtung funktionieren, erlaubt das vorgestellte System eine Synchronisation in beide Richtungen. Änderungen im einen Modell werden erkannt und - soweit möglich - automatisch auf das andere Modell übertragen. Dafür wurde ein lauffähiger Prototyp in der Simplified Modeling Platform umgesetzt. Das System verwaltet Modellinhalte und ihre visuellen Darstellungen, erkennt Veränderungen und nutzt sogenannte Brückenelemente, um die Verbindung zwischen den Modellelementen nachzuvollziehen. Klare Regeln sorgen dafür, dass die Synchronisation nachvollziehbar und konsistent abläuft.

Während Änderungen im DEMO-Modell vollständig automatisch in BPMN übernommen werden können, erfordert die Synchronisation in der umgekehrten Richtung die Interaktion des Nutzers, um passende DEMO-Elemente auszuwählen und Mehrdeutigkeiten aufzulösen.

Der Prototyp wurde auf Korrektheit, Laufzeitverhalten und Stabilität getestet. Die Ergebnisse zeigen, dass sich kleine bis mittelgroße Modelle zuverlässig und effizient synchronisieren lassen. Eine detaillierte Analyse der Komplexität und Benchmarks bestätigen zudem die technische Skalierbarkeit des Ansatzes. Auch wenn sich die Arbeit auf DEMO und BPMN konzentriert, bietet der gewählte Lösungsansatz eine gute Grundlage, um in Zukunft weitere Modellierungssprachen miteinander zu verbinden.

Abstract

In enterprise modeling, different languages are often used to capture different perspectives of an organization. While DEMO focuses on actor commitments and coordination logic, BPMN models the operational flow of processes. In practice, these models are often maintained separately, even when they describe overlapping business behavior. This leads to inconsistencies and manual duplication of effort.

This thesis presents a practical approach for keeping DEMO and BPMN models synchronized over time. Unlike traditional one-way transformation approaches, this system continuously synchronizes DEMO and BPMN models by identifying and applying changes in both directions - automatically where rules are clear, and with user support where ambiguity exists. We implemented a working prototype on top of the Simplified Modeling Platform, which supports the creation, update, and deletion of model elements and connections. To manage this process, the system uses bridge elements to track links between models and applies clearly defined synchronization rules. The overall architecture is designed to be modular and efficient, enabling smooth synchronization across multiple modeling scenarios.

From DEMO to BPMN, changes are applied automatically using the semantics of the DEMO transaction pattern. In the reverse direction, from BPMN to DEMO, the system provides user guidance to resolve ambiguity.

The prototype was evaluated in terms of correctness, runtime performance, and robustness. Results show that the system handles small to medium-sized models efficiently and reliably. A detailed complexity analysis and benchmark study confirm the scalability of the approach. While the implementation is tailored to DEMO and BPMN, the methodology provides a foundation for extending synchronization to other modeling language pairs in the future.

Contents

CHAPTER 1

Introduction

1.1 Problem overview

Enterprises, such as commercial companies and government agencies, frequently use multiple modeling languages simultaneously to document and analyze different aspects of their business. For example, an enterprise may use DEMO to structure its transactional workflows [1], BPMN to describe business processes [2], and ArchiMate to model the enterprise architecture [3].

While each language serves a specific purpose and provides unique insights, the models created with them are usually not connected. In practice, they are often maintained separately, even if they describe similar parts of the organization. This can be inefficient because changes in one model need to be manually copied into others, which can lead to redundancy, inconsistencies, and a higher risk of misalignment [4].

In dynamic enterprise environments where models need frequent updates to reflect organizational changes or process improvements, interoperability becomes a key challenge [4]. Misalignment between models can lead to misunderstandings among stakeholders, hinder collaboration, and reduce an organization's agility.

A theoretical solution to this challenge is to unify all models under a single integrated modeling language. However, this is impractical in most real-world scenarios. As France and Rumpe point out, modeling languages often differ significantly in their abstraction levels, intended purposes, and target audiences [5]. Trying to replace them with a common language can lead to a loss of important meanings, and may encounter resistance from existing tools and habits.

Instead of introducing yet another modeling language, a more practical approach is to connect the ones that are already in use. By building bridges between existing languages, we can synchronize overlapping parts of models. This allows each language to continue

serving its own purpose while maintaining overall alignment. In this way, the strengths of the individual languages are not lost, and we avoid the time-consuming management of completely separate models. Our goal is to make them work together in a clear and traceable way [6].

This thesis follows that path. Rather than designing something new from scratch, we focus on how two established but quite different modeling languages can be kept in sync. We implement a working prototype to explore how such a bridge can function in practice, and we examine the technical and conceptual challenges that come with trying to keep these languages aligned over time.

1.2 Scope of the thesis

To keep the exploration clear and manageable, we focus on synchronizing models between two specific modeling languages: DEMO (Design and Engineering Methodology for Organizations) and BPMN (Business Process Model and Notation).

More precisely, we focus on synchronizing a subset of elements that represent business transactions - a central construct in DEMO that reflects how coordination and commitments occur in organizations. These transaction steps - such as Request, Promise, Execute, and Accept - provide a semantically rich and stable basis for mapping between languages. In BPMN, their counterparts are represented by activities, message events, pools, and data objects.

The synchronization logic is defined at the metamodel level, meaning it is based on the structure of the modeling languages themselves, not just on specific examples. This makes the approach more robust and adaptable. It also means that, in the future, the same method could be applied to other elements or even to other modeling languages [7].

The prototype developed in this thesis supports both directions: from DEMO to BPMN and from BPMN back to DEMO. Because DEMO models typically provide clearer semantics and a structured transaction logic, the transformation from DEMO to BPMN can be done automatically. The reverse direction, from BPMN to DEMO, is more complex. BPMN models tend to leave out some of the meaning that DEMO requires, so this transformation sometimes needs user input to resolve ambiguities or reconstruct missing information [8, 9].

To ensure the system remains manageable and reliable, the synchronization mechanism is limited to the happy path, that is, to well-formed models that follow the normal success flow. Special cases such as incomplete transactions, invalid modeling patterns, or semantic conflicts outside the standard flow are explicitly excluded from the current scope. Handling such exceptions would require additional logic and validation mechanisms, and is left for future work.

The implemented system supports all major types of changes to the models: creating, updating, or deleting elements, attributes, and connections. Where needed, conflict

detection helps ensure that changes don't overwrite each other in unintended ways. The entire synchronization process has been implemented as a working prototype that integrates with a model management platform and supports visualization of changes.

1.3 Research Questions

Based on the practical scope outlined above, we now define the key research questions that specify the central challenges of this thesis. These questions shape the direction of the work and provide a structured basis for its design, implementation, and evaluation.

Central research question

> How can a metamodel-based synchronization mechanism be designed and implemented to enable consistent, traceable, and partially automated transformation between DEMO and BPMN models?

Sub-questions

To address this central question, we formulate the following sub-questions, which focus on specific aspects of the problem:

1. Which transaction-related elements in the DEMO and BPMN metamodels can be mapped to each other in a semantically meaningful and technically feasible way, and what synchronization rules are needed to support this mapping?
2. How can model differences - such as added, removed, or modified elements, attributes, and connections - be detected using a delta-based comparison approach to support bidirectional synchronization?
3. How can detected changes be transformed and applied to the target model in a way that updates semantics, visuals, and bridge mappings consistently?
4. Which types of changes can be synchronized automatically, and in which cases is manual intervention required due to ambiguity or model-specific constraints (particularly in BPMN-to-DEMO transformations)?
5. What is the complexity of the resulting synchronization mechanism?

1.4 Methodology Overview

This thesis follows a design-science research approach. The goal is to create and evaluate a working prototype that enables synchronization between DEMO and BPMN models. To ensure that this process is scientifically grounded, the methodology combines two perspectives: Hevner'fs Design Science Research (DSR) [10] and Algorithm Engineering [11, 12].

Hevner's Design Science Research provides the overall structure for this kind of research. It focuses on solving real-world problems by designing, building, and evaluating artifacts, in our case, a synchronization mechanism. According to Hevner et al. [10], such research should be relevant to practice, built on a solid knowledge base, and rigorously evaluated. Our work follows this idea: we define a relevant modeling problem, design synchronization rules based on existing modeling knowledge, implement a solution, and test it to ensure it works reliably. While we do not follow every formal step of the DSR process [13], the core principles are applied. Some steps were merged, for example, "define objectives" is integrated into the problem analysis, and demonstration and evaluation are treated as a single internal-testing phase, since external stakeholder evaluation was beyond the scope of this thesis.

In addition, we adopt ideas from Algorithm Engineering [11, 12], a methodology that focuses on building algorithms through repeated testing and refinement. This fits well with how the synchronization prototype was developed: rather than being built all at once, it was implemented in small, iterative steps. After each step, the behavior of the prototype was tested, evaluated, and improved. This back-and-forth between theory and practice helped us improve both the performance and correctness of the solution.

The research process followed in this thesis involved five main phases:

1. **Problem Analysis and Requirements:** We begin by analyzing the problem of synchronizing DEMO and BPMN models. A review of existing approaches shows that there is no clear solution for keeping models in sync across these languages. Based on this, we define key requirements and select DEMO's transaction pattern as a stable structure for transformation.

2. **Rule Design at the Metamodel Level:** We then define synchronization rules that describe how DEMO elements can be mapped to BPMN constructs such as activities, events, or pools. These rules are designed to preserve meaning while allowing flexibility.

3. **Prototype Implementation and Iteration:** We developed a prototype that puts these rules into practice. It compares models, identifies differences, and applies the correct synchronization. The prototype supports both directions, from DEMO to BPMN and back. The implementation is carried out in small steps, with frequent testing and improvements, following the principles of Algorithm Engineering.

4. **Evaluation and Testing:** The prototype is evaluated through internal testing. We check whether the synchronizations are correct, whether the models remain consistent, and how the system performs with increasing model size. This evaluation helps us assess the quality of the solution and points to areas for future improvement. It also satisfies the DSR requirement to rigorously evaluate the designed artifact [10].

5. **Documentation and Contribution:** All results are documented in this thesis. The main contribution is twofold: a working synchronization prototype and a

methodology that shows how structured design and iterative refinement can be used to solve synchronization problems in model-driven engineering.

This methodology combines conceptual design with practical testing. It ensures that the solution is not only rigorously grounded in existing knowledge but also works reliably in practice.

1.5 Structure of the Thesis

The work is divided into ten chapters, each of which builds on the previous one and leads the reader from the conceptual motivation to the technical implementation and evaluation of the synchronization prototype.

Chapter 2 provides an overview of the two modeling languages involved in this work, DEMO and BPMN, and summarizes relevant literature on bridging modeling languages. It outlines the state of the art in DEMO-BPMN transformations and identifies the research gap addressed by this thesis.

Chapter 3 introduces the Simplified Modeling Platform (SMP), which serves as the technical foundation for the prototype. It explains how the platform supports custom modeling languages and how it is used to manage and manipulate DEMO and BPMN models in this work.

Chapter 4 presents the metamodels and modeling constructs relevant to the synchronization logic. It explains the core elements and relationships of DEMO and BPMN as they are implemented in SMP, and highlights how these definitions are used to structure models and support synchronization.

Chapter 5 defines the functional and non-functional requirements for the synchronization prototype. These requirements guide the development and evaluation of the system and ensure that all relevant aspects, such as performance, traceability, and user interaction are considered.

Chapter 6 describes the architecture of the prototype. It introduces the folder structure used to organize original models, synchronized snapshots, and bridge mappings. It also explains the delta-based synchronization logic and outlines the three main scenarios supported by the system: initial model creation, DEMO-to-BPMN updates, and BPMN-to-DEMO updates.

Chapter 7 defines the synchronization rules and mappings that govern how model elements and connections are translated between DEMO and BPMN. These rules form the conceptual core of the transformation logic and are expressed both as structured mappings and procedural descriptions.

Chapter 8 details the internal implementation of the prototype. It presents the main synchronization functions, their modular structure, and their computational complexity.

Each function is analyzed in terms of runtime behavior and backend interaction, using both theoretical and empirical benchmarks. This chapter also reflects on the algorithmic complexity of the overall synchronization mechanism.

Chapter 9 evaluates the prototype in terms of performance, correctness, and robustness. It presents benchmark results for all synchronization scenarios, test coverage for key components, and a discussion of known limitations. The chapter also reflects on early generalization insights and how the approach could be extended to other modeling languages in future work.

Chapter 10 concludes the thesis by summarizing the main contributions and findings. It reflects on the practical implications of the synchronization prototype and outlines possible directions for future research, including improved scalability, extended language support, and user interface integration.

CHAPTER 2

State of the Art

This chapter provides a structured overview of the two modeling languages used in this thesis: DEMO and BPMN. After introducing each language individually, existing research on bridging modeling languages in general is presented, followed by a focused discussion on efforts to integrate DEMO and BPMN.

2.1 Design and Engineering Methodology for Organizations

The Design and Engineering Methodology for Organizations (DEMO) is a modeling method that focuses on the transactions between actor roles to reveal the essential structure of an organization. It was introduced by Dietz as part of his theory of Enterprise Ontology [14] and later developed into a more practical, human-centered approach for understanding and designing organizations [1].

The main idea behind DEMO is to describe what an organization is, not just what it does. It models how people make agreements, take on responsibilities, and communicate in a structured way. These interactions are organized into transactions, which define who agrees to do what, when, and under which conditions they apply.

A central component of DEMO is the so-called ψ-theory, which provides a theoretical foundation for understanding social interactions in organizations. It formalizes communication and production acts and offers a consistent way to represent the commitments made within an enterprise [14]. As a result, DEMO models are known for being complete, coherent, consistent, and concise [1]. DEMO not only represents standard process flows but also includes exceptions such as cancellations or reversals.

One of DEMO's key strengths is that it separates essential business logic from technical or IT-related details. This makes it easier to model organizational processes that remain

valid even as supporting technologies change [15]. DEMO provides a highly abstract and semantically rich view of workflow logic, focusing on coordination and production aspects rather than implementation details [16].

This makes DEMO a helpful tool for projects where people from different backgrounds, like business, architecture, and IT, need to work together; its formal structure reduces misunderstandings and supports informed decisions [17].

However, DEMO also has some limitations. Because it does not include control flow or detailed execution logic, DEMO models cannot be enacted directly in typical workflow engines and usually have to be transformed into an executable notation such as BPMN first [18]. DEMO can also be difficult to learn: stakeholders often require guidance or tool support to interpret and use DEMO models [17]. These factors have limited DEMO's adoption, particularly in environments that favour practical, tool-supported notations [19].

2.2 Business Process Model and Notation

Business Process Model and Notation (BPMN) is one of the most widely used standards for modeling business processes. It was developed and is maintained by the Object Management Group (OMG) to provide a common visual language that connects business analysts and technical developers [2]. Its popularity comes from the fact that it is both easy to understand for business users and detailed enough to support process execution in technical systems.

At its core, BPMN uses a flowchart-like notation to describe how business activities are carried out, how decisions are made, and how people and systems interact. It includes modeling elements like tasks, events, gateways, and message flows, which help represent workflows both within and across organizations. This makes it possible to show things like parallel steps, alternative paths, subprocesses, and exception handling in a clear and structured way.

One reason for BPMN's widespread adoption is its flexibility. It can be used to create simple overviews for managers or detailed process definitions for developers. Since the introduction of BPMN 2.0, the standard also supports execution semantics, allowing BPMN diagrams to be deployed into Business Process Management Systems (BPMS). This capability has been adopted by several tools that generate executable models directly from BPMN [20]. As a result, BPMN is not only used for documenting processes but also for automating them in execution environments.

However, BPMN has some limitations, for example that it focuses on how processes are executed, but not why they exist. It lacks a formal way to represent organizational commitments or the deeper business semantics behind actions [1]. In contrast to conceptual modeling approaches like DEMO, BPMN operates at a more technical level and often leaves out the organizational context and actor responsibilities behind the process steps.

From a methodological point of view, BPMN is typically used in a bottom-up way to improve existing processes. DEMO, on the other hand, is used in a top-down way to model the essential structure of an organization [14]. While DEMO focuses on stability and conceptual clarity, BPMN supports flexibility and execution. This makes the two languages complementary: DEMO helps define what an organization should do, while BPMN shows how to implement those ideas in practice.

Although BPMN provides a standardized syntax, its diagrams are often modeled inconsistently without clear conventions [21]. DEMO defines a complete transaction pattern - including steps such as Initial, Request, Requested, Promise, Promised, State, Stated, Accept, Accepted - that make the internal logic and agreements behind organizational actions explicit. In contrast, BPMN lacks a standardized way to capture such transaction stages or formalize organizational commitments, which often leaves the rationale behind process steps implicit. These differences have motivated research efforts to combine the strengths of both languages. Several studies have explored how DEMO's semantic richness can be used to enhance BPMN models by clarifying roles, commitments, and process intentions [22, 9]. Such approaches aim to make sure that the processes modeled for execution still reflect the original business intentions, ensuring semantic traceability between conceptual and operational layers.

To summarize the complementary nature of the two languages, Table 2.1 highlights key differences in abstraction, use cases, and limitations.

Table 2.1: Comparison of DEMO and BPMN in terms of focus, strengths, and limitations [23].

Aspect	**DEMO**	**BPMN**
Abstraction level	High-level, ontology-based	Operational, workflow-oriented
Primary focus	Commitments, actor roles, transactions	Control flow, activity execution
Typical use	Top-down conceptual modeling	Bottom-up process improvement
Strengths	Semantic clarity, theoretical foundation, model stability [1]	Tool support, expressiveness, simulation and execution capabilities [24]
Limitations	Steep learning curve, limited adoption, abstract	Semantic ambiguity, missing commitments, focus on implementation [21]
Scope representation	Explicit via actor roles and composite transactions	Often implicit unless documented separately [2]

2.3 Overview of existing work in bridging Modeling Languages

The challenge of connecting different modeling languages has been widely discussed in research. Many organizations use more than one modeling approach, which often leads to inconsistencies, misunderstandings, or extra effort to keep models aligned. To solve this, several studies propose bridges that transform or synchronize models across languages.

One research direction focuses on connecting value modeling with enterprise architecture. De Kinderen et al. presented a method that uses DEMO's transaction patterns to bridge

e3Value and ArchiMate [25]. Their idea is to translate the economic logic of e3Value into operational structures in ArchiMate, with DEMO acting as the intermediate step. In a follow-up study, they describe how this transformation can be formalized using metamodel mappings to ensure semantic consistency across the different modeling layers [26].

Other approaches explore how BPMN can be transformed into formats that support execution or simulation. Boukelkoul and Maamri developed a transformation framework that converts BPMN models into DEVS (Discrete Event System Specification) models, allowing formal verification and simulation of business processes [27]. Their method defines clear transformation rules to ensure modularity and make the resulting models easier to validate. Similarly, Barjis proposes translating BPMN into Petri nets to support simulation and analysis of dynamic behavior in enterprise systems [28].

There are also efforts to generate workflows directly from conceptual models. Figueira and Aveiro introduced DEMOBAKER, a transformation framework that converts DEMO models into executable business process workflows [29]. Their approach introduces a new action rule syntax for DEMO, enabling the automated generation of workflow logic. This work demonstrates how conceptual enterprise models can serve as a foundation for executable processes, bridging the gap between high-level design and operational execution.

A few efforts go beyond one-way transformations and attempt true bidirectional synchronization. For instance, Mazanek and Hanus [30] implemented a bidirectional mapping between BPMN and BPEL using functional logic programming. Their approach enables models to be updated in either direction while maintaining consistency, making it a rare example of executable two-way synchronization in practice.

In the broader context of model-driven engineering, research on multi-view model synchronization addresses the challenge of maintaining consistency between related models. Grossmann et al. [31] propose a framework for detecting and resolving conflicts during concurrent model evolution, highlighting the complexity of update propagation in co-dependent models.

The next section will focus specifically on bridging DEMO and BPMN, and how their complementary strengths can be combined.

2.4 Model Transformations between DEMO and BPMN

Several researchers have explored ways to integrate DEMO and BPMN, recognizing that the two languages serve different purposes but can complement each other when combined properly.

One of the earliest efforts was made by Caetano et al., who analyzed how DEMO's ontology-based structure can improve the consistency of BPMN models [32]. They pointed out that BPMN often lacks clear constraints on transaction logic, which can lead to incomplete or inconsistent process models. DEMO's stricter semantics help ensure that transaction logic in BPMN models is both complete and semantically precise.

Heller [33] built on this idea in his master's thesis by investigating how DEMO methods can be used to generate BPMN models. He proposed a structured mapping approach that preserves the semantic integrity of DEMO during the transformation process. The thesis shows that using DEMO as a foundation can reduce ambiguity in BPMN models and provide clearer links between business intentions and executable processes.

A more formal transformation method was presented by Mráz et al. [22], who introduced a set of transformation rules for converting DEMO's transaction pattern into BPMN. Their method extends the ideas of Caetano and Heller, aiming to automate the transformation process. However, their approach does not fully handle more complex structures such as nested transactions or actor hierarchies.

To support such transformations in practice, Gray et al. [34] developed a DEMO modeling tool on the ADOxx platform. The tool allows users to model DEMO structures and partially transform them into BPMN collaboration diagrams. A key focus was on translating the Organization Construction Diagram (OCD) from DEMO into BPMN Pools and Message Flows, while maintaining semantic consistency.

The same authors later evaluated the tool in real-world modeling sessions [35]. Their results showed that users could successfully create BPMN diagrams from DEMO models and appreciated the tool's usability. However, the study also revealed that more complex constructs, such as nested transactions or exception handling, still require manual intervention. Building on these results, De Vries and Bork [23] identified nine distinct transformation scenarios that support converting DEMO's Coordination Structure Diagram (CSD) and Process Structure Diagram (PSD) into BPMN. Their fine-grained rule set improves coverage and consistency in BPMN diagrams derived from DEMO, addressing earlier structural limitations.

Other authors have contributed method-oriented or tooling-related perspectives. For example, Rodrigues [36] developed a modeling methodology for translating DEMO transaction models into BPMN process diagrams. His thesis emphasized semantic preservation and consistency, proposing practical guidelines rather than a tool-based implementation.

Some researchers have taken a different approach by embedding DEMO concepts directly into BPMN models to address BPMN's semantic limitations. Van Nuffel et al. [8] proposed an extension of the BPMN metamodel that integrates DEMO's transaction structure. Their goal was to give BPMN the formal expressiveness needed to represent enterprise transactions explicitly, beyond just modeling activity sequences.

More recently, Guerreiro and Dietz [9] introduced a framework that aligns BPMN process flows with DEMO's transaction coordination structure. Their approach clarifies the commitments and communication acts within a process, making BPMN models more transparent and semantically precise. Building on this, Guerreiro and Sousa [37] created SemantifyingBPMN, a tool that generates BPMN diagrams directly from DEMO transaction patterns. The tool ensures that BPMN models include key coordination acts

like request, promise, and acceptance - making business intentions traceable in technical process models.

To evaluate the impact of DEMO on BPMN modeling in practice, Náplava and Pergl [18] conducted an empirical study in an academic setting. Their findings confirmed that BPMN models created with DEMO principles were more structured and less ambiguous. The study concluded that DEMO can help improve the quality and consistency of BPMN models, especially in contexts where process logic and actor commitments need to be made explicit.

2.5 Research Gap

While previous studies have significantly advanced our understanding of model transformation, most approaches remain limited to one-way conversions or isolated tool support. Several works focus on translating DEMO models into BPMN to improve process clarity or to generate executable workflows. Others embed DEMO concepts into BPMN or use DEMO to validate BPMN model consistency. These approaches usually stop at initial model generation, once the models are created, they evolve independently.

None of the literature reviewed addresses the challenge of keeping DEMO and BPMN models consistent over time as they change. No existing approach supports dynamic, bidirectional synchronization where updates in one model are automatically propagated to the other. Moreover, aspects such as conflict detection, semantic alignment during model evolution, and traceable change propagation are not addressed in current tools or frameworks.

This thesis addresses this research gap by introducing a synchronization approach that supports bidirectional updates between DEMO and BPMN models. The proposed method goes beyond initial transformation and enables ongoing co-evolution of models - an essential feature in real-world environments where business requirements and models continuously change. By combining structural mappings with synchronization logic and conflict-aware mechanisms, this work contributes a practical solution for maintaining semantic and structural alignment between heterogeneous models. It also lays the foundation for further research into cross-model consistency, tool interoperability, and model-driven enterprise design.

CHAPTER 3

The Simplified Modeling Platform

To implement and evaluate the synchronization prototype developed in this thesis, we rely on the Simplified Modeling Platform (SMP), a model-driven development environment developed in the context of enterprise engineering research [38]. SMP supports the creation, visualization, and transformation of models and is particularly well-suited for setups that involve multiple modeling notations.

Unlike traditional modeling tools that are tied to fixed languages such as BPMN or UML, SMP is designed to be notation-agnostic. This means that modeling languages are not hard-coded into the platform but can be defined externally and loaded dynamically. In practice, languages like DEMO and BPMN are made available through declarative configuration files, known as notation scripts, which describe the structure, semantics, and appearance of model elements [39]. For the purposes of this thesis, we rely on existing implementations of both DEMO and BPMN that are already defined within the platform and publicly available as notation scripts in the official Simplified Modeling Platform repository [40].

3.1 Platform Architecture and Capabilities

SMP is designed to be lightweight and extensible, offering both a graphical user interface for manual modeling and a programmatic interface for automation. It is deployed as a multi-layer web application and can be accessed directly through the browser. The backend, built in Go, exposes a WebSocket-based API that enables model operations such as creating elements, setting attributes, adding connections, and manipulating diagrams.

As illustrated in Figure 3.1, the platform is organized into a frontend (user interface), an application server (containing notation and transformation logic), and a backend database. Developers can add new transformation functionality, access stored models, and connect to the application server through external scripts or services.

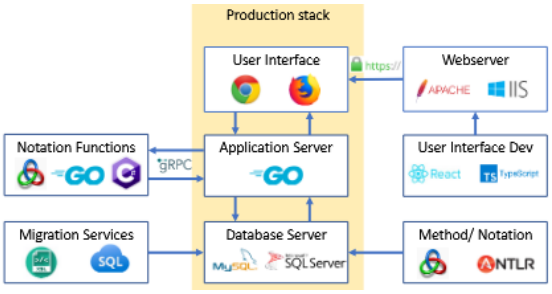

Figure 3.1: Architecture of the Simplified Platform, adapted from [38].

Models in SMP are structured hierarchically. Each model is composed of one or more folders, which contain semantic model elements, and each folder is associated with one or more diagrams for visual representation. This separation allows for clear organization of model content and makes it possible to maintain and compare different versions of the same model. These structural features are crucial for implementing synchronization mechanisms, which rely on tracking differences between model snapshots.

3.2 Use of SMP in This Thesis

SMP serves as the technical foundation for all modeling activities in this work. We interact with the platform exclusively through its API to load, compare, and synchronize DEMO and BPMN models. Because the platform already provides notation definitions for both modeling languages, we are able to focus entirely on implementing and evaluating synchronization logic, without needing to extend the modeling language definitions themselves.

The prototype makes use of the platform's hierarchical structure to store multiple model versions (original and synchronized), and relies on its visual API to generate layouts and update diagrams automatically. All changes to model elements, attributes, and connections are performed programmatically, allowing for repeatable and traceable synchronization operations.

By leveraging the flexibility, automation capabilities and internal model structure of SMP, we are able to develop and evaluate a prototype for a bridge-based synchronization mechanism in a real modeling environment. The implementation builds directly on the platform's backend API, and all the results presented in the following chapters were achieved with this setup.

CHAPTER 4

Meta-models and Modeling constructs

This chapter introduces the metamodels that form the basis of the synchronization prototype and describes how the DEMO and BPMN modeling languages are implemented within the SMP. By working at the metamodel level, the synchronization logic can operate on generic modeling constructs rather than specific instances.

4.1 DEMO

To support synchronization between DEMO and BPMN, it is important to understand the foundational concepts of DEMO. DEMO is based on the principle that companies can be modeled by their essential transactions - what people request, decide and produce. To represent this, DEMO provides four interrelated aspect models: the Construction Model (CM), the Process Model (PM), the Action Model (AM), and the Fact Model (FM) [14]. In this thesis, the focus lies specifically on the CM and PM, as they provide the necessary structural and behavioral elements for synchronization.

The **Construction Model (CM)** defines the static organizational structure of an enterprise. It identifies which actor roles exist, what transaction types they are responsible for, and how these roles and transactions are interconnected. In essence, it models who does what within the organization. DEMO uses three key artifacts to visualize the CM: the Organizational Construction Diagram (OCD), which shows actor roles and their transaction links; the Transaction Product Table (TPT), which lists the services or products delivered; and the Bank Contents Table (BCT), which outlines the types of information produced during execution [1].

Figure 4.1 shows an example of a simplified Construction Model. Actor role A00 initiates transaction type T01, which is executed by actor role A01. Initiators and executors are

connected via solid lines, with a small black diamond marking the executor. Actor role A07, which is connected via a dashed line, has access to the history of T01, indicating an informational dependency rather than a responsibility.

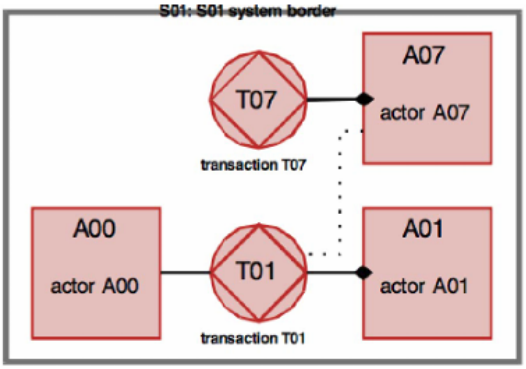

Figure 4.1: Example of a DEMO Construction Model [1]

While the CM defines who is involved in which transactions, the **Process Model (PM)** describes how these transactions are carried out over time. Each transaction in DEMO is modeled as a structured interaction between two actors: the Initiator, who expresses a need, and the Executor, who fulfills that need [14]. To ensure semantic clarity, DEMO standardizes the structure of each transaction using the Complete Transaction Pattern, which divides the process into three phases: the Order Phase, where the initiator makes a request and the executor responds (e.g., by promising); the Execution Phase, in which the executor performs the agreed-upon task; and the Result Phase, where the result is communicated and accepted by the initiator. Each of these phases is modeled by coordination acts and coordination events. Coordination acts (C-acts) represent communicative intentions between actors, such as a promise or a request. Coordination events (C-events), on the other hand, denote the observable results or conclusions of these actions. Together, they form the semantic backbone of transaction modeling in DEMO.

In this thesis, we focus only on the so-called happy flow, which represents the ideal course of a transaction without any interruptions or exceptions. This flow follows a defined sequence of C-acts and C-events [23, 9]:

(in) Initial → [rq] Request → (rq) Requested → [pm] Promise → (pm) Promised → Execution → [st] State → (st) Stated → [ac] Accept → (ac) Accepted.

This path forms the foundation for our prototype implementation and guides the transformation logic toward BPMN.

Figure 4.2 illustrates the Complete Transaction Pattern with the happy flow highlighted in green. Although the happy flow serves as the primary reference for our mapping and synchronization logic, DEMO also provides mechanisms to model exceptions. For example, an executor may decline a request, or an initiator may reject the outcome of a

transaction. In such cases, the process can be either retried or terminated. DEMO defines four revocation patterns to handle these situations in a structured way. These patterns extend the Complete Transaction Pattern to account for deviations and rollbacks, and are also visualized in Figure 4.2 [1].

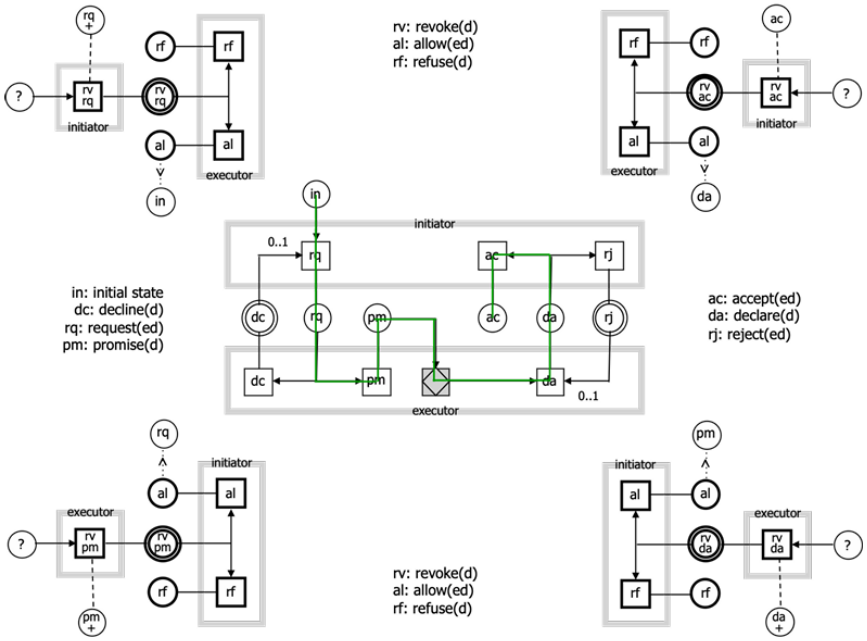

Figure 4.2: The Complete Transaction Pattern in DEMO, with the happy flow highlighted [1]

In DEMO's Process Model diagrams, C-events are shown as circular nodes, while C-acts are represented as rectangular boxes. A gray box containing a diamond marks a production act, indicating that a product or service is being delivered. The direction of arrows between elements indicates their semantic role: arrows from a disk to a box are response links, meaning that the act is a response to a previous event; arrows from a box to a disk are causal links, meaning the act triggers the event. Light-gray backgrounds are used to group the steps that belong to the initiator or executor [1].

Beyond structure and behavior, DEMO also includes two additional aspect models: the **Action Model (AM)** and the **Fact Model (FM)**. The AM describes how actors should behave during a transaction based on organizational rules. It distinguishes between action rules, which define how to respond to certain coordination events, and work instructions, which specify how to carry out concrete production acts. These elements help ensure that transactions are executed in a consistent and policy-compliant manner [14].

The FM, on the other hand, focuses on traceability. It captures the facts and outcomes of transactions over time, documenting what has occurred within the organization. This

historical perspective supports auditing, compliance, and strategic analysis [14].

While the AM and FM are not directly used in the synchronization prototype, they complete the DEMO methodology and provide important context for understanding its broader modeling capabilities.

4.1.1 Implementation of DEMO in the Simplified Platform

The DEMO metamodel used in this thesis is implemented in the Simplified Modeling Platform, which supports diagram-based modeling through customizable metamodel definitions. In line with the theoretical foundations, the implementation focuses on two aspect models: the CM and the PM. These are realized through two separate diagram types within the platform, the Process Structure Diagram and the Transaction Process Diagram.

The **Process Structure Diagram (PSD)** corresponds to the CM. It defines the static structure of the organization and includes the following element and connection types, as defined in the official DEMO metamodel [40]:

```
toolbox "PSD" ToolboxPSD37
on diagram (ProcessStructureDiagram37)
comment "Toolbox with all elements for the PSD diagram"
(
  element TransactionKind37,
  element ElementaryActorRole37,
  element CompositeActorRole37,
  connection WaitLink37,
  connection ResponseLink37
)
```

Each element type is associated with specific attributes that reflect its role within DEMO. The suffix “37” in the element and connection names refers to version 3.7 of the DEMO methodology, which is the version implemented in the SMP and used throughout this thesis. An ElementaryActorRole37 has a Name attribute and a boolean attribute SelfInitiating, which indicates whether the actor can independently initiate transactions. The CompositeActorRole37, which represents a group of elementary roles, includes only the Name attribute. The TransactionKind37 element is shared across both the PSD and TPD diagrams and includes several attributes such as Name, ProductKindIdentification, ProductKindFormulation, and TransactionSort, which is an enumeration that classifies the transaction, with the default value set to “Original”.

The **Transaction Process Diagram (TPD)** represents the PM and captures the dynamic behavior of transactions. It includes both C-acts and C-events, modeled using the following types:

```
toolbox "TPD" ToolboxTPD37
on diagram (TransactionProcessDiagram37)
comment "Toolbox with all elements for the TPD diagram"
(
  element TransactionKind37,
  element TransactionProcessStepKind37,
  connection WaitLink37,
  connection ResponseLink37,
  connection CasualLink37,
  connection ReversionLink37
)
```

The central element in this diagram is the TransactionProcessStepKind37, which represents an individual transaction step. It includes attributes such as Name, Abbreviation, and StepKind, where StepKind is based on the STEPKIND enumeration. This enumeration defines the various phases a transaction step can represent, such as C-acts (Request, Promise, Execute, Accept) and C-events (Requested, Promised, Executed, Stated, Accepted). Only the subset of step kinds relevant to the happy flow is used in this prototype. The different connection types define relationships between steps. While WaitLink37 and ResponseLink37 are used in both PSD and TPD to express temporal or triggering dependencies, the CausalLink37 represents logical consequences, and the ReversionLink37 models rollback behavior. All connections link two TransactionProcessStepKind37 elements and are interpreted semantically based on the associated step kinds. In addition to the connections defined directly in the PSD and TPD toolboxes, we also include important actor-transaction links from the Organisation Construction Diagram (OCD). These connections - Initiator37e and Executor37e - connect an ElementaryActorRole37 to a TransactionKind37. Although these are technically defined in the OCD, we use them in the prototype to specify which actors are responsible for which transaction steps, and ultimately to support the role assignment logic during BPMN model generation. This implementation enables the prototype to accurately model both the static structure and dynamic behavior of transactions in accordance with the DEMO methodology.

4.2 BPMN

Following the theoretical description and implementation of DEMO, we now turn to the second modeling language used in this thesis: **Business Process Model and Notation (BPMN)**. BPMN focuses on the control flow of processes. It captures the order in which activities are performed, the events that influence execution, and the communication between participants. The language supports a wide range of scenarios, from simple sequential workflows to complex parallel and collaborative processes.

At the core of BPMN are the Flow Objects, which define the most important behavioral elements of a process. These include events, activities, and gateways. Events are used to

indicate something that happens during the process. A Start Event marks the beginning of a process, an End Event indicates its conclusion, and Intermediate Events represent something that happens between the start and the end, such as a delay, a timer, or the arrival of a message [2].

Activities represent the actual work being performed within a process. These can be atomic tasks or larger sub-processes. In our implementation, we concentrate on simple tasks that describe individual work units. Gateways are used to model decision points and parallelism in the process. They define how the process can diverge into multiple branches (splits) or converge from several paths into one (joins). BPMN supports various types of gateways such as exclusive (XOR), inclusive (OR), and parallel (AND) gateways, each with different execution semantics.

To define how elements are connected, BPMN uses three types of connections, which include Sequence Flows, Message Flows, and Associations. Sequence Flows define the execution order between flow objects within the same process. Message Flows connect elements across different participants (e.g., between pools) and represent communication or interaction. Associations are used to link artifacts such as text annotations or data objects to the process elements without affecting the control flow.

BPMN also introduces structural constructs to represent organizational context. A Pool denotes a major participant in the process (e.g., a company or department), while Lanes subdivide Pools into internal roles or units. These Lanes help clarify who is responsible for which part of the process.

In addition to control flow, BPMN allows for modeling information flow through Data Objects and Data Stores. Data Objects represent transient information passed between activities, while Data Stores represent persistent data repositories shared across the process.

BPMN also provides artifacts such as Groups and Text Annotations, which are used for documentation and visual grouping but do not influence execution behavior.

Figure 4.3 shows the subset of BPMN elements used in this thesis. These include specific Flow Objects (such as Start Events, Activities, and Intermediate Message Events), Connecting Objects (Sequence and Message Flows), structural elements (Pools and Lanes), and data-related elements (Data Objects and Data Stores). The selected elements are sufficient to model the so-called happy flow of a DEMO transaction in BPMN, without involving branching or exception handling.

To support this targeted subset, we created a simplified metamodel. Figure 4.4 illustrates the reduced BPMN metamodel. Flow-related elements such as Activities and Events are highlighted in blue, and their connections via Sequence and Message Flows are highlighted in yellow. All executable process steps (Activities and Events) are grouped under the abstract superclass Flow Node, which inherits metadata from the abstract class Object (orange). Structural and data-related components, such as Pool, Lane, and DataObject, are shown in green.

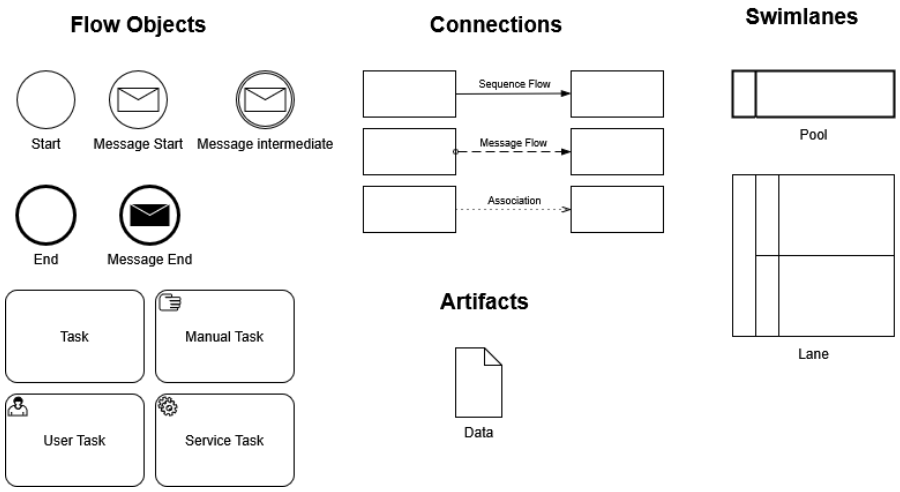

Figure 4.3: Visual representation of BPMN elements used in this thesis implementation. The selected subset includes only those constructs necessary to represent the happy flow of a transaction.

Flows in this model always connect two Flow Nodes through a source and target association. Events are further classified by their type (Start, Intermediate, End), and Message Events are treated as a specialized subclass that enables inter-participant communication. Each Pool must contain at least one Lane, and both Pools and Lanes can contain Flow Nodes. DataObjects are associated with Activities to represent input and output information relevant to task execution.

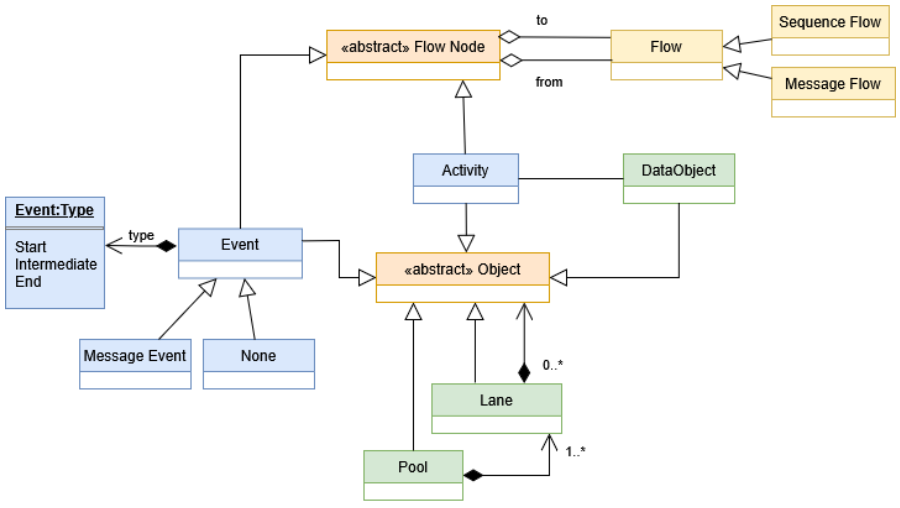

Figure 4.4: Simplified BPMN metamodel showing only the elements used in this thesis

By avoiding advanced constructs such as gateways, choreographies and compensation events, the metamodel remains focused and aligned with the assumptions of the DEMO

Happy Flow. The selected elements provide sufficient expressiveness to represent the core logic of DEMO transactions in BPMN while maintaining the simplicity of the implementation.

4.2.1 Implementation of BPMN in the Simplified Platform

As with DEMO, the BPMN concepts used in this thesis are implemented using the modeling infrastructure provided by the Simplified platform. The platform supports the definition of custom metamodels, including diagram types, model elements, attributes, and connection rules [40]. For our prototype, BPMN modeling is supported through a diagram type called Business Process Diagram. According to the metamodel, this diagram formally contains only a limited set of element types:

```
diagram "Business Process Diagram" BusinessProcessDiagram
contains (
    BusinessProcess,
    DataObject,
    Lane
)
```

While central BPMN elements such as Activity, Event, and Pool are not explicitly listed here, they are still defined in the metamodel and can be created within this diagram. The Simplified platform does not enforce strict validation of diagram structure at runtime, meaning that modelers can freely include these elements in a Business Process Diagram without triggering any errors. This flexibility allows us to build semantically complete BPMN models, despite structural limitations in the metamodel definition.

Some semantic relationships defined in the official BPMN 2.0.2 [41] specification are only partially supported in the Simplified metamodel. For example, there is no formally defined containment or linking relationship between Activity elements and Pool or Lane elements. To address this limitation, the Simplified development team extended the metamodel in our environment by introducing a custom connection type called ActivityPoolLink, which we use to connect Activity elements to Pool elements. We also apply this connection to Event elements in the same way. Although this connection is not formally part of the metamodel definition, the editor accepts it and allows it to function correctly. This mechanism enables the prototype to determine which flow elements belong to which actors and is essential for supporting automated role assignments during synchronization. Similarly, Sequence Flows are formally defined only between Activity elements, which would normally restrict flows between Event and Activity elements. In practice, however, the modeling environment accepts these combinations, and they behave as expected, despite lacking formal support in the metamodel schema.

The metamodel provides a variety of element types and attributes. Event elements include attributes such as Flow Dimension (Start, Intermediate, End) and Type Dimension (e.g.,

Message, None), allowing us to distinguish different event roles. Activity elements are described using attributes like Task Type (e.g., User, Manual, Service) and behavioral flags such as Ad-Hoc, Compensation, and Collapsed, which we do not use. DataObject elements include a Data Object Type attribute, which can be set to DataObject, DataStore, DataInput, or DataOutput. For our purposes, we only use DataObject and DataStore to reflect transactional outputs.

In our implementation, we use only a subset of these constructs to capture the transactional structure of the DEMO "happy flow". Pools and Lanes are used to represent Initiator and Executor roles, and Activities reflect discrete transaction steps. We include Start, Intermediate Message, and End Events to capture the temporal and communicative structure of the process. DataObjects and DataStores are linked to Activities via Association flows, representing coordination facts (C-Facts) and production facts (P-Facts). Finally, Sequence Flows and Message Flows model process logic and inter-role communication. Although some of these links are not strictly supported by the metamodel's semantic constraints, they are accepted by the modeling environment and function correctly in practice. This makes the Simplified platform sufficiently expressive for the scope of this prototype.

In summary, the BPMN implementation in the Simplified platform provides the necessary flexibility and modeling capabilities to support the transformation and synchronization of DEMO transactions into BPMN process flows.

CHAPTER 5

Prototype Requirements

To ensure that synchronization between DEMO and BPMN behaves as intended, a set of functional and non-functional requirements has been defined. These requirements are derived from the previously defined research questions, the methodological goals outlined in Chapter 1, and the intended scope of the prototype. They specify what the system must be capable of (functionally) and what qualities it should exhibit (non-functionally), such as performance, stability, and extensibility. By making these expectations explicit, the requirements serve as a basis for the implementation as well as a reference framework for evaluating the success of the prototype. They also ensure that the system can be maintained, adapted to future modeling languages, and reused in broader enterprise modeling contexts.

5.1 Functional Requirements

- **Model Synchronization:** The prototype should be able to synchronize DEMO and BPMN models in three practical situations: when no BPMN model exists yet and needs to be created from a DEMO model; when the DEMO model has changed and those changes need to be reflected in the BPMN model; and when the BPMN model has changed and the DEMO model needs to be updated accordingly. While these are treated as three distinct scenarios in the implementation, they conceptually fall into two synchronization directions - from DEMO to BPMN and from BPMN to DEMO. The first case (initial creation) is simply a special case of the DEMO to BPMN direction, triggered when the target model is still empty.

- **Detecting Changes:** The prototype should be able to recognize what has changed in a model since the last synchronization. This includes newly created elements, deleted elements and elements whose attributes or connections have been changed.

- **Mapping Elements and Connections:** When creating or updating models, the system must apply clear synchronization rules that define how a DEMO element corresponds to one or more BPMN elements and vice versa. This also includes the handling of connections between elements and ensuring their correct representation.
- **Handling Bridges Between Elements:** The prototype should keep track of which elements in DEMO are linked to which elements in BPMN. This is done using bridge elements that must be automatically created and updated during synchronization.
- **Showing Elements Visually:** Each element in the model must also have a visual representation in the diagram.
- **Dealing with Ambiguities:** In cases where it is not obvious how a BPMN element can be mapped to a DEMO step, the system should suggest a reasonable option (based on the surrounding context), but also give the user the opportunity to confirm or adjust this decision.
- **Tracking Synchronization State:** The prototype should record when the last synchronization happened, so it can figure out whether anything has changed since then.
- **Conflict Detection:** If both models have changed since the last synchronization, the system should detect the conflict, notify the user, and abort synchronization to avoid inconsistent updates.

5.2 Non-functional Requirements

- **Performance:** Synchronization should be fast and not cause unnecessary delays. Especially for medium-sized models, the system should work smoothly without noticeable lag.
- **Code Structure and Maintainability:** The prototype should be implemented in a clean and structured way. Important functionality should be separated into helper functions so that the code is easy to understand, test, and extend.
- **Flexibility for Future Use Cases:** Although the current focus is on DEMO and BPMN, the architecture and logic should be able to support other modeling languages or more advanced synchronization rules in the future without major revision.
- **Consistency and Reliability:** The prototype must ensure that DEMO and BPMN models remain consistent when synchronization is triggered by the user. Synchronization does not happen automatically, but only upon explicit user action (e.g. calling a function or pressing a synchronization button in future versions).

Once synchronization has been triggered, the system should pass on changes to the models correctly.

- **Clarity for the User:** All system output (e.g. logs or user prompts) should be clear and helpful. When the system asks the user for input, it must do so in a way that makes the decision understandable.
- **Error Handling and Stability:** The system should not crash if something goes wrong (e.g., if an expected element is missing). Instead, it should log the issue and continue running wherever possible.

Together, these requirements lay the groundwork for the architecture of the prototype, which will be described in the next chapter.

CHAPTER 6

System Architecture for Model Synchronization

The architecture of the prototype is designed to support reliable, bidirectional synchronization between DEMO and BPMN models. It is implemented on top of the Simplified Modeling Platform (introduced in Chapter 3), which organizes modeling content into models, folders, and diagrams. To enable version tracking, change detection, and transformation operations, the prototype defines a structured folder-based architecture within a single model. In the context of the Simplified Modeling Platform, a folder is a structural container used to group semantic model elements and diagrams. These folders allow users to manage different model versions, organize content modularly, and compare snapshots without interference. This folder-based organization plays a central role in the synchronization logic of the prototype.

6.1 Folder Structure

To manage synchronization between DEMO and BPMN models, we define seven dedicated folders within the prototype. Each folder serves a specific role in tracking changes, storing snapshots, or connecting model elements across versions and metamodels. The organization follows a two-layered architecture: the upper layer contains the user-editable models, while the lower layer holds their synchronized snapshots and the internal bridge elements.

The top layer includes the following:

- **DEMO**: This folder contains the original DEMO model created and maintained by the user.
- **BPMN**: This folder contains the user-editable BPMN model.

Beneath these, the lower layer supports version control and traceability:

- **DEMO Synchronized**: Stores a snapshot of the DEMO model at the time of its last successful synchronization. This snapshot serves as a reference for detecting subsequent changes.
- **BPMN Synchronized**: Analogously, this folder contains the last synchronized version of the BPMN model.
- **DEMO Bridge**: Contains internal bridge elements that link elements in the current DEMO model to their corresponding elements in the DEMO Synchronized folder.
- **BPMN Bridge**: Contains bridge elements linking elements in the BPMN model to their counterparts in BPMN Synchronized.
- **Bridge**: Serves as the cross-metamodel bridge folder. It stores bridge elements that connect elements in the synchronized DEMO and BPMN models. These mappings support traceability between languages and serve as the basis for executing transformation rules.

This layered structure is illustrated in Figure 6.1. The screenshot shows how each modeling language (DEMO and BPMN) is represented by a group of three folders: the user-editable model, its synchronized snapshot, and a bridge folder for intra-model mapping. In addition, the central Bridge folder connects elements across the two languages. Diagrams are stored in the editable folders and serve as the interface for modeling.

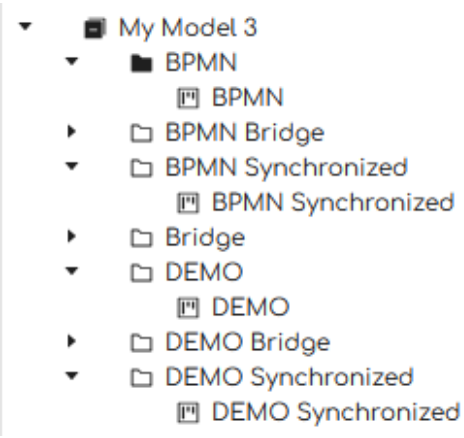

Figure 6.1: Folder structure of the synchronization prototype in the Simplified Modeling Platform

Each synchronization scenario follows the same general process. Changes are not directly applied from one modeling language to the other. Instead, they always pass through a

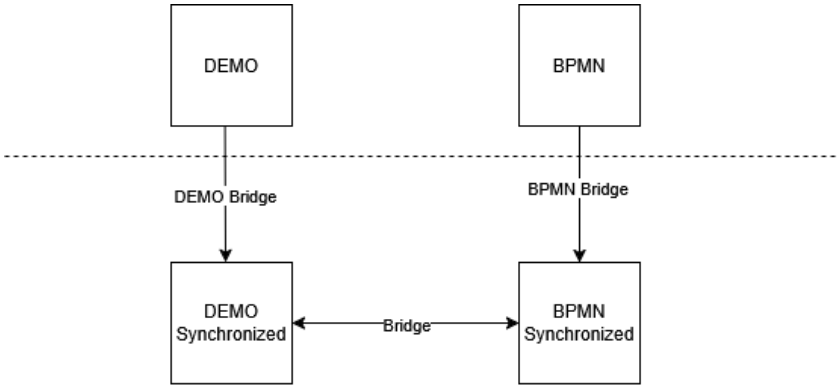

Figure 6.2: Synchronization Flow

series of intermediate snapshots to preserve traceability and enable delta-based comparison. This flow can be summarized as follows:

$$\textit{Original} \rightarrow \textit{Synchronized} \rightarrow \textit{Synchronized (Other)} \rightarrow \textit{Original (Other)}$$

This structure guarantees that all changes are first captured, transformed, and verified before being reflected in the target model. The next section introduces the concept of delta-based synchronization, which formalizes this transformation process.

6.2 Delta Calculation

Delta-based synchronization forms the technical foundation for Scenarios 2 and 3, where existing models must be kept in sync after user changes. This section formalizes how changes between versions of a model are detected, transformed, and applied.

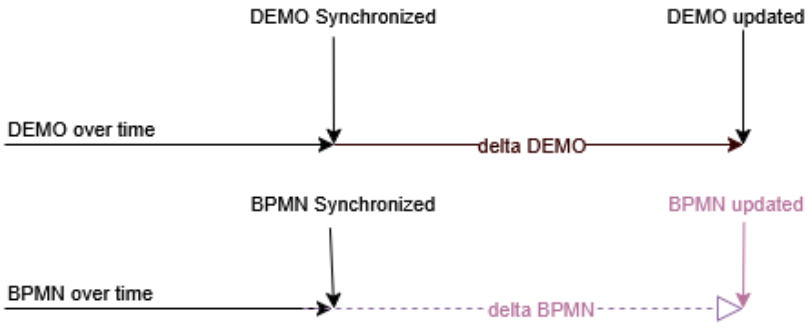

Figure 6.3: Delta calculation

Figure 6.3 shows this concept of delta-based synchronization. Each row represents the state of a model at a specific point in time. In the upper part of the diagram, the

current (updated) version of the DEMO model is compared to its last synchronized counterpart. The resulting difference is captured as a delta, denoted $\Delta^{(t)}_{\text{DEMO}}$, which includes all additions, deletions, and attribute modifications that occurred since the last synchronization.

Formally, the delta is defined as:

$$\Delta^{(t)}_{\text{DEMO}} \equiv M^{(t)}_{\text{DEMO}} \rightarrow M^{(t-1)}_{\text{DEMO}}$$

where M^{t}_{DEMO} represents the current model and M^{t-1}_{DEMO} the last synchronized version. The delta can be further divided into:

- Δ^{+}_{DEMO}: newly created elements and connections,
- Δ^{-}_{DEMO}: removed elements and connections,
- $\Delta^{\circ}_{\text{DEMO}}$: modified attribute values.

The lower part of the diagram shows how this delta is used to update the target model. The changes in DEMO are transformed according to predefined synchronization rules and applied to the synchronized BPMN model, resulting in a corresponding delta $\Delta^{(t)}_{\text{BPMN}}$. This transformed delta represents the changes that must be reflected in BPMN as a result of the modifications in DEMO.

This transformation process can be described mathematically as:

$$\Delta^{(t)}_{\text{BPMN}} = T(\Delta^{(t)}_{\text{DEMO}}), \qquad M^{t}_{\text{BPMN}} = M^{t-1}_{\text{BPMN}} \oplus \Delta^{(t)}_{\text{BPMN}}$$

Here, T denotes the transformation function that maps changes from DEMO to BPMN, and $\oplus$ represents the application of the transformed delta to the model.

This approach improves performance by processing only the actual changes and ensures traceability and semantic consistency between DEMO and BPMN over time. The following section explains how this delta-based logic is applied in practice through three synchronization scenarios.

6.3 Synchronization Scenarios

The prototype's synchronization logic is built around three different scenarios. Each scenario reflects a specific direction and trigger for synchronization and follows the general transformation flow presented above. Together, these scenarios ensure that updates across DEMO and BPMN models are handled in a consistent and traceable way.

6.3.1 Scenario 1: Creating BPMN from DEMO

This scenario applies when a DEMO model has been created, but no corresponding BPMN model exists yet. In this case, the prototype initiates a one-time synchronization to generate the initial BPMN model based on the DEMO structure and semantics. The process begins by copying the current DEMO model to the DEMO Synchronized folder to create a reference point for future comparisons. Based on this snapshot and using predefined synchronization rules, the system then generates a new synchronized BPMN model in the BPMN Synchronized folder. Finally, this generated model is copied into the user-editable BPMN folder, allowing the user to continue working with the BPMN representation. This scenario establishes the initial link between DEMO and BPMN models and defines the basic state for the subsequent delta-based synchronization.

6.3.2 Scenario 2: Updating BPMN from DEMO Changes

In this scenario, both DEMO and BPMN models already exist, but the user has updated the DEMO model. The prototype must now detect and propagate these changes to keep the BPMN model consistent. The process starts by comparing the current DEMO folder with the DEMO Synchronized folder to determine which elements and connections have been added, removed, or updated. Before applying any updates, deletion changes are first propagated to the BPMN Synchronized model to remove outdated elements and maintain the validity of existing bridge connections. Once deletions are handled, the updated DEMO model is copied into the DEMO Synchronized folder. Based on the calculated delta and the synchronization rules, the prototype then updates the BPMN Synchronized model accordingly. In the final step, the system compares the updated BPMN Synchronized model with the BPMN folder and applies the resulting delta, ensuring that the user-editable BPMN model reflects the latest DEMO changes. This scenario enables unidirectional synchronization from DEMO to BPMN, ensuring that changes in the source model are accurately mirrored in the target.

6.3.3 Scenario 3: Updating DEMO from BPMN Changes

This scenario mirrors Scenario 2 but applies in the reverse direction. When the user modifies the BPMN model, the system ensures that corresponding updates are reflected in the DEMO model. As in the previous case, synchronization begins with a comparison between the current BPMN model and its last synchronized version. The resulting delta is used to identify removed, updated, and newly created elements. Deletion changes are first applied to the DEMO Synchronized model to eliminate any DEMO elements that correspond to removed BPMN elements. Next, the updated BPMN model is copied into the BPMN Synchronized folder to reflect the current state. Using the synchronization rules, the system then applies the transformed changes to the DEMO Synchronized model. Finally, the prototype compares this updated DEMO Synchronized folder with the DEMO folder and applies any necessary changes so that the DEMO model remains aligned with BPMN.

6.4 Software Architecture Overview

Having described the folder-based model structure, the synchronization flow, and the delta-based logic, we now turn to the internal software architecture that implements this behavior. Figure 6.4 illustrates the main components of the prototype and how they interact with the Simplified Modeling Platform (SMP).

At the core of the system is the Synchronization Controller, which determines the active synchronization scenario and orchestrates the end-to-end process. It invokes the Model Loader to retrieve model data from the SMP via WebSocket API calls and populate an in-memory structure known as the BridgeContext. This context stores all relevant data needed for synchronization - including model elements, visual representations, semantic connections, and bridge mappings.

Once the models are loaded, the Delta Calculator compares the current and synchronized versions to identify created, deleted, or updated elements and connections. Detected changes are then passed to the Change Applier, which applies them to the target model, updates visual representations, and maintains bridge consistency.

All logic for translating DEMO to BPMN (and vice versa) is encapsulated in the Synchronization Rules Engine. This module ensures that semantic transformations respect the modeling constraints of each language. In cases where mappings are ambiguous or incomplete, the system consults the User Interaction Module to involve the user in resolving the decision.

Together, these components form a modular and traceable architecture that supports efficient, conflict-aware synchronization across two modeling languages.

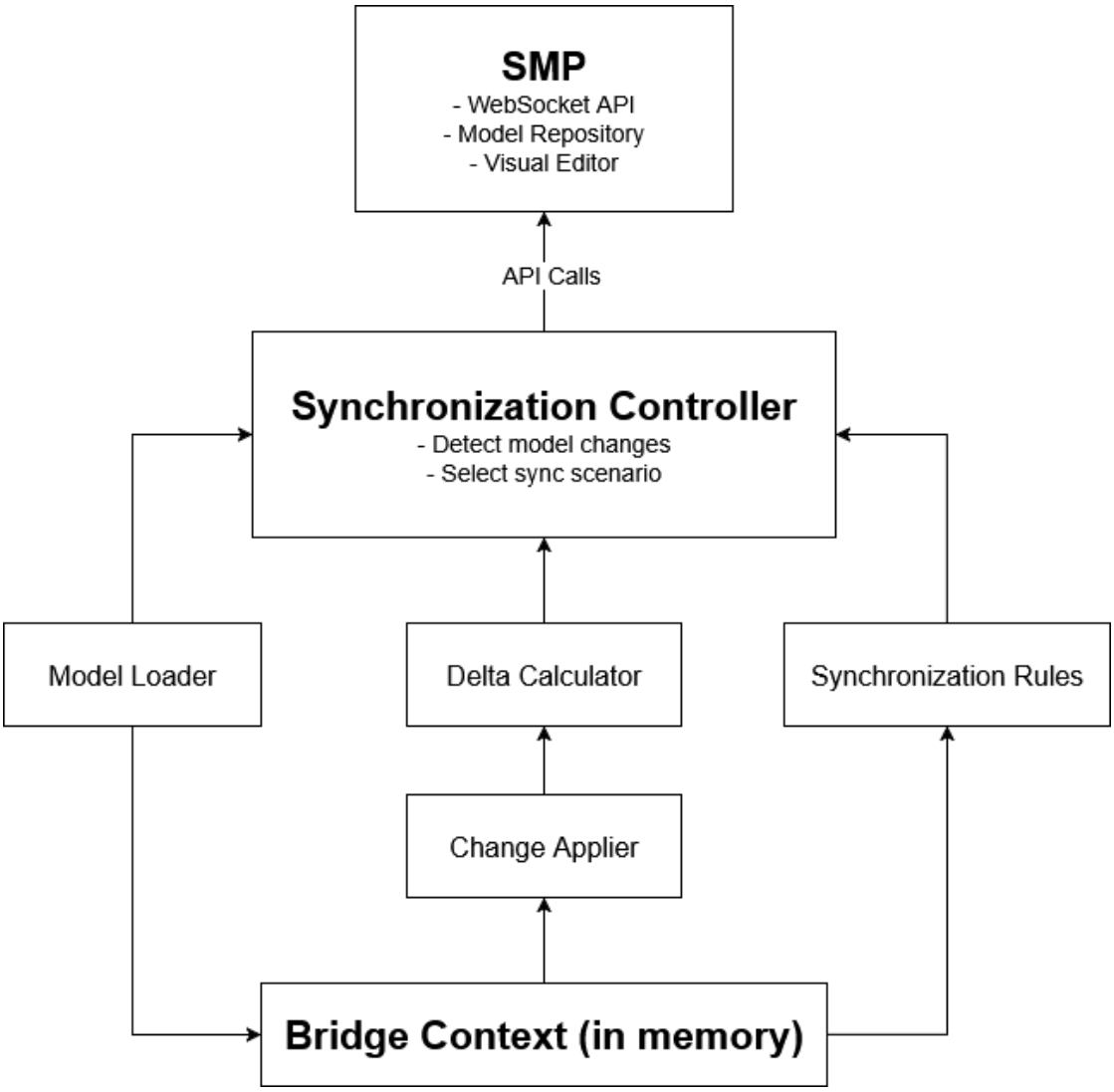

Figure 6.4: Software architecture of the synchronization prototype. The system interacts with the Simplified Modeling Platform via API calls and organizes synchronization logic into modular components that operate on a shared in-memory context.

CHAPTER 7

Synchronization Rules and Mappings

This chapter introduces the synchronization rules that translate DEMO model elements into their BPMN counterparts and vice versa. These rules build on the architectural framework and synchronization scenarios described in the previous chapter 6 and form the core logic used during model creation, update, and deletion. They also directly address Research Questions 1 and 2, which focus on defining conceptually valid mappings and enabling consistent, traceable, bidirectional synchronization between DEMO and BPMN models.

7.1 Mapping Rules Between DEMO and BPMN Elements

The mapping rules in this chapter are partly inspired by prior work on DEMO to BPMN transformations [33, 22, 23]. However, unlike these approaches, we formalize the rules to support continuous synchronization in both directions, including change detection, attribute propagation, and consistent connection updates. The formalization consists of structured rule templates that define how each DEMO element type (e.g., Transaction Step Kind or Actor Role) is translated into one or more BPMN elements, including logic for visual placement and semantic connections. Table 7.1 shows the main mappings, while the remaining cases are explained individually in the text. These rules are applied consistently during both initial model generation and incremental updates, depending on the active synchronization scenario.

7.1.1 Element Mapping Rules

The following table 7.1 summarizes the refined mappings used throughout the system.

DEMO Element	BPMN Element(s)	Attributes
`TransactionProcessStepKind37`: Step Kind = **Initial**	Start Event	- Flow Dimension = Start
`TransactionProcessStepKind37`: Step Kind = **Request**	Activity + DataObject	- Activity: Task Type = User - DataObject: Data Object Type = DataStore (C-Fact)
`TransactionProcessStepKind37`: Step Kind = **Requested**	Intermediate Message Event	- Flow Dimension = Start, Type Dimension = Message
`TransactionProcessStepKind37`: Step Kind = **Promise**	Activity + DataObject	- Activity: Task Type = User - DataObject: Data Object Type = DataStore (C-Fact)
`TransactionProcessStepKind37`: Step Kind = **Promised**	Intermediate Message Event	- Flow Dimension = Intermediate, Type Dimension = Message
`TransactionProcessStepKind37`: Step Kind = **Execute**	Activity + DataObject	- Activity: Task Type = Manual - DataObject: Data Object Type = DataObject (P-Fact)
`TransactionProcessStepKind37`: Step Kind = **State**	Activity + DataObject	- Activity: Task Type = Service - DataObject: Data Object Type = DataStore (C-Fact)
`TransactionProcessStepKind37`: Step Kind = **Stated**	Intermediate Message Event	- Flow Dimension = Intermediate, Type Dimension = Message
`TransactionProcessStepKind37`: Step Kind = **Accept**	Activity + DataObject	- Activity: Task Type = User -DataObject: Data Object Type = DataStore (C-Fact)
`TransactionProcessStepKind37`: Step Kind = **Accepted**	End Message Event	- Flow Dimension = End, Type Dimension = Message
`ElementaryActorRole`	Pool	- Name = Identification

Table 7.1: Mapping rules between DEMO and BPMN elements

7.1.2 Naming Strategy for BPMN Elements

To ensure consistency, the prototype uses a hybrid naming strategy when creating BPMN elements from DEMO elements. Instead of copying the DEMO identification directly, each BPMN element receives a name based on the Step Kind and the Identification attribute of the corresponding DEMO element.

This guarantees that BPMN models remain semantically clear and understandable on their own, that element names correspond to their functional roles, and that traceability to the original DEMO model is maintained. For example, a DEMO element with Step Kind "Promise" and identification "Order123" results in a BPMN Activity named "Order123 (Promise Product)" and an associated Data Object named "Promise C-Fact".

The following table 7.2 summarizes the naming conventions for each supported Step

Kind:

DEMO Step Kind/Element	Naming Convention for BPMN Elements
Initial	<Identification> (Start Transaction)
Request	<Identification> (Request Product), Request C-Fact
Requested	<Identification> (Receive Request)
Promise	<Identification> (Promise Product), Promise C-Fact
Promised	<Identification> (Promise Sent)
Execute	<Identification> (Execute Product), P-Fact Product
State	<Identification> (State Product), State C-Fact
Stated	<Identification> (State Sent)
Accept	<Identification> (Accept Product), Accept C-Fact
Accepted	<Identification> (Accepted)
ElementaryActorRole	Name taken from the Name attribute (used as Pool name)

Table 7.2: Naming conventions for BPMN elements generated from DEMO elements

Reverse Mapping (BPMN to DEMO)

When BPMN elements are transformed into DEMO elements, the naming strategy is applied in a simplified and consistent way. For BPMN Pools, the Identification of the Pool is directly reused as the Identification of the corresponding DEMO ElementaryActorRole. For example, a BPMN Pool named "Sales Department" will result in a DEMO actor with the same name.

For BPMN Activities and Events, the prototype reuses the BPMN element's Identification as the identification of the corresponding DEMO element. However, the semantic meaning, such as whether the element represents a Request, Promise, or Accept step is stored in the Step Kind attribute. The Identification is therefore kept clean and free of semantic prefixes, avoiding redundancy and improving clarity.

This approach ensures that BPMN-to-DEMO synchronization produces models that are readable, structurally consistent with the DEMO methodology, and traceable to their original BPMN sources.

7.2 Connection Rules

In addition to element mappings, the system applies a set of rules to synchronize connections:

- **Message Flow:** Created between a BPMN Activity and a Message Event when the corresponding DEMO connection is between a C-Act and a C-Event.
  For example, a connection between a Request (C-Act) and Requested (C-Event) leads to a Message Flow from a BPMN Activity to a Message Start Event. Similarly, a connection from Declare to Declared is mapped to a Message Flow from a Service Task to an Intermediate Message Event.
- **Sequence Flow:** Used between BPMN Activities or between BPMN Events when the corresponding DEMO connection is between two C-Acts or two C-Events.
  For example, a connection from Promise (C-Act) to Execute (C-Act) results in a Sequence Flow between the corresponding User Task and Manual Task. Likewise, a connection between Promised and Declared (both C-Events) is translated into a Sequence Flow between their respective Intermediate Message Events.
- **Response Link:** When a new BPMN Message Flow is created between an Activity and a Message Event that are both bridged to distinct DEMO elements, the system creates a Response Link between the corresponding C-Act and C-Event.
- **Follow Link:** When a new BPMN Sequence Flow is added between two Activities (or two Events) that are bridged to valid consecutive DEMO steps, the prototype creates a Follow Link connection between them.

Deleted connections are handled symmetrically when synchronizing from DEMO to BPMN: if a connection between two DEMO elements is removed, the corresponding BPMN connection is also deleted. When synchronizing in the other direction (from BPMN to DEMO), the system checks whether the source and target BPMN elements are bridged to different DEMO elements. Only if this is the case, the corresponding DEMO connection will be removed. Otherwise, no change is made, as both BPMN elements may represent parts of a single DEMO element (e.g., a DataStore and an Activity linked to the same transaction step).

7.2.1 Actor-Role Assignment Connections

In addition to message and sequence flows, the system also synchronizes actor-role assignments between DEMO and BPMN models.

In the DEMO metamodel, actor-role assignments are explicitly represented through semantic and visual connections: Initiator37e and Executor37e, linking a TransactionKind to either an Elementary or Composite Actor Role. These connections are visible in the diagram and form a key part of the model structure.

In contrast, BPMN represents actor assignments implicitly via ActivityPoolLink connections, which associate Activities or Events with Pools. These connections are purely semantic in the current metamodel implementation and are not visually represented in the diagram. However, they are used internally by the synchronization logic to track pool assignments.

When synchronizing from DEMO to BPMN, the system uses the Initiator37e and Executor37e links to determine which Pool each Activity or Event should be placed in. This ensures that the resulting BPMN elements appear in the visually correct pool based on the actor-role semantics of the DEMO model.

Conversely, when synchronizing from BPMN to DEMO, the system determines the relevant actor-role assignment by analyzing the vertical position (Y value) of the new Activity or Event and matching it to the correct Pool. Based on this, it infers the correct actor role and creates the corresponding Initiator37e or Executor37e connection in the DEMO model. The decision is also guided by the inferred Step Kind (e.g., Request, Promise, Execute): for example, a Promise step leads to an Executor37e connection from the TransactionKind to the actor, while a Request step leads to an Initiator37e connection from the actor to the transaction.

If an ActivityPoolLink is missing in the BPMN model at the time of synchronization, the system generates it automatically based on the element's position, maintaining semantic consistency even though the link is not visually rendered.

7.3 Handling Updates and Special Cases

Attribute Updates

When an element's attributes change (e.g., name or task type), these changes are propagated directly to the corresponding elements. If an attribute does not yet exist in the BPMN/DEMO counterpart, it is created.

Step Kind Change in DEMO

If the "Step Kind" attribute is changed in a DEMO element, the prototype treats the element as deleted and recreated. This avoids inconsistencies in synchronization logic since a change in step kind often implies a completely different BPMN structure.

New BPMN Elements

When a new BPMN element is added by the user (e.g., an Activity or Message Event), the system analyzes its incoming and outgoing connections to infer a likely DEMO step kind. Because BPMN itself does not encode the organizational commitments that DEMO makes explicit, this inference can be ambiguous; the user is therefore asked to confirm or override the suggestion to ensure conceptual validity. By default, if the element is a

BPMN Activity with no connections, the prototype proposes the Step Kind Request; if it is a Message Event, it is proposed as an Initial step.

Deletion in BPMN

If BPMN elements such as Activities or Message Events are deleted, the corresponding DEMO elements are removed as well. If only a DataObject or DataStore is deleted, the corresponding DEMO element is preserved and only the bridge is removed.

CHAPTER 8

Prototype Implementation and Performance analysis

This chapter presents the internal implementation of the implemented prototype. Key data structures, such as the `BridgeContext`, are introduced along with the core synchronization algorithms used. Each major function is analyzed with respect to time and space complexity, and its performance is empirically validated through benchmark tests. To isolate computational performance, all benchmarks in this chapter are executed using mocked backend calls, to avoid real API requests during testing. This approach ensures that the internal efficiency of the synchronization logic can be evaluated without external influences. In addition, the number of required API calls per function is analyzed. The next chapter 9 evaluates the overall runtime behavior of the three scenarios using real API calls, as well as the system's correctness, robustness, and limitations.

The full implementation of the prototype, is part of the repository [42].

Benchmark Setup

All benchmarks were executed on a local development machine with the following configuration:

- **CPU:** Intel Core i5-8250U @ 1.60GHz (4 cores, 8 threads)
- **RAM:** 16 GB
- **OS:** Windows 11
- **Go Version:** 1.24rc2

8.1 BridgeContext

To support efficient and consistent synchronization between DEMO and BPMN models, we introduce a central coordination structure called the BridgeContext. This shared in-memory structure contains all model elements, visual representations, connections, and bridge mappings required for bidirectional synchronization across both modeling languages and their respective synchronized copies.

The BridgeContext fulfills several key purposes. Most importantly, it significantly improves performance by avoiding repeated backend calls. Instead, all relevant model data is preloaded into structured maps and lists, allowing comparisons, synchronizations, and updates to be performed entirely in memory.

It also simplifies the implementation. Rather than passing multiple identifiers, element lists, or lookup maps to every function, most operations just accept the shared `ctx` object (BridgeContext instance). Since it is passed by reference, all updates, such as creating or deleting elements, are visible system-wide without requiring additional reloading.

Before any synchronization logic can run, the prototype loads all relevant model data into the BridgeContext. The BridgeContext stores data such as:

- `CopyDemoMap`: ID → DEMO Copy element
- `BpmnToDemoMap`: BPMN ID → DEMO ID (bridge mapping)
- `ConnectionsBySourceID`: Element ID → []Connection
- `VisualsByID`: Element ID → Visual Representation

This includes both semantic and visual elements, connection data, and bridge mappings from all involved folders - originals, synchronized versions, and bridges - for DEMO and BPMN. Because all synchronization steps operate entirely in memory, the quality and completeness of this initial loading phase directly affect the responsiveness and scalability of the system.

To avoid unnecessary memory usage, only the data needed for the current synchronization scenario is loaded. For example, to detect Scenario 1 the prototype only requires the original DEMO model and its diagram, while Scenarios 2 and 3 additionally require elements from the synchronized folders, bridge mappings, connection metadata, and visuals. The loading logic is implemented in scenario-specific functions like `LoadForComparison`, `LoadForSyncFromDemo`, and `LoadForSyncFromBPMN`, ensuring that no redundant data is retrieved.

By centralizing access to the model state and managing all updates through a single structure, the BridgeContext plays a critical role in keeping the synchronization architecture modular, efficient, and maintainable.

Time Complexity

Let n be the number of model elements, m the number of connections, and b the number of bridge elements.

Operation	Complexity	Notes
Element parsing and mapping	$O(n)$	Populate lists and maps
Connection parsing and indexing	$O(m)$	Fill source/target index maps
Bridge construction	$O(b)$	Build and reverse bridge mappings
Visual mapping	$O(n)$	Assign visuals to elements
Total	$O(n+m+b)$	Linear in input sizes

Table 8.1: Time Complexity of BridgeContext Initialization

Space Complexity

All data is loaded and stored in memory via lists and maps inside the BridgeContext. Since the structure holds the full set of elements, connections, and bridges, space complexity is also linear:

$$O(n+m+b)$$

Backend/API Performance

The following table 8.2 summarizes the estimated number of API calls required to initialize the BridgeContext:

API Call	Frequency
`GetModelElement`	~ 7 (DEMO, BPMN, synchronized versions, bridges)
`GetModelVisualElement`	~ 1
`GetModelConnection`	~ 1
`GetModelVisualConnection`	~ 1
`GetNotationMetaSet`	~ 2 (DEMO + BPMN)
Total API Calls	$\approx$ 12

Table 8.2: Estimated Frequency of API Calls for BridgeContext Initialization

8.2 Overall Synchronization Function

The central coordination logic of the prototype is implemented in the `SyncModels` function. It manages all three synchronization scenarios: generating a BPMN model from DEMO (Scenario 1), updating BPMN based on DEMO changes (Scenario 2), and updating DEMO based on BPMN changes (Scenario 3). It also includes logic to handle conflicts (when both sides have changed) and to detect when no synchronization is needed at all.

The function receives an initialized `BridgeContext` as input and populates it with only the model data required for the current synchronization step. It begins by loading only the original DEMO and BPMN elements to determine whether the BPMN model exists. If it does not, Scenario 1 is triggered. Otherwise, it loads the rest of the elements and compares the original and synchronized versions of each model using bridge mappings to detect added, removed, or updated elements and connections.

If changes are detected on both sides, the system aborts the synchronization to prevent unintentional overwriting. In such cases, the user must manually resolve the conflict by editing one of the models until only one side contains changes. If changes are found only on one side, the function applies the appropriate synchronization scenario using specialized helper functions. For instance, Scenario 2 applies DEMO-side changes to BPMN using functions like `ApplyCreateAndUpdateChangesToBPMN()`, while Scenario 3 performs the reverse.

Because the `BridgeContext` provides centralized, in-memory access to all model elements, the system ensures consistency and avoids redundant API calls. Each helper function involved in the process is described in detail in the following sections, along with its performance characteristics.

The following pseudocode summarizes the logic implemented in the `SyncModels` function:

Algorithm 8.1: Synchronization Logic (`SyncModels`)

Input : Model and diagram IDs for DEMO and BPMN
Output : Synchronized state between DEMO and BPMN models

```
1  Initialize BridgeContext with model identifiers;
2  LoadMinimalModelData() to load original DEMO and BPMN elements;
3  if BPMN has no elements and DEMO has elements then
4  |   // Scenario 1: Generate BPMN model from DEMO;
5  |   LoadForScenario1() to fetch DEMO visuals and connections;
6  |   CopyAllElementsAndConnections() from DEMO to CopyDEMO;
7  |   CreateBPMNFromDEMO() using CopyDEMO as input;
8  |   CopyAllElementsAndConnections() from BPMN Copy to BPMN;
9  |   return
10 end
11 LoadForComparison() to load copies, bridges, and connections;
12 CompareFolders() for DEMO and BPMN to detect changes;
13 if both DEMO and BPMN have changes then
14 |   Log conflict and abort synchronization;
15 |   return
16 end
17 if only DEMO has changes then
18 |   // Scenario 2: Apply DEMO changes to BPMN;
19 |   LoadForSyncFromDemo() to load required visuals and bridges;
20 |   ApplyConnectionDeletionsToBPMN();
21 |   ApplyDeletionChangesToBPMN();
22 |   CopyChangesToFolder() with DEMO changes to CopyDEMO;
23 |   ApplyCreateAndUpdateChangesToBPMN();
24 |   ApplyConnectionCreationsToBPMN();
25 |   CompareFolders() for BPMN and apply final changes to BPMN;
26 |   return
27 end
28 if only BPMN has changes then
29 |   // Scenario 3: Apply BPMN changes to DEMO;
30 |   LoadForSyncFromBPMN() to load required visuals and bridges;
31 |   ApplyDeletionChangesToDEMO();
32 |   CopyChangesToFolder() with BPMN changes to CopyBPMN;
33 |   ApplyCreateAndUpdateChangesToDEMO();
34 |   CompareFolders() for DEMO and apply final changes to DEMO;
35 |   return
36 end
37 Log: No changes detected - models are already synchronized;
```

The execution time and memory consumption of `SyncModels` depend entirely on which

scenario is triggered and which functions are invoked. For instance, Scenario1 is typically the most expensive, as it involves full model generation. Scenario 2 and Scenario 3 operate incrementally based on detected deltas. The performance characteristics and estimated API interactions of each function involved are analyzed in the subsequent sections. A detailed performance analysis and benchmark comparison of the three synchronization scenarios is provided in Chapter 9.

8.3 Scenario 1

Below we discuss the functions that are required for scenario 1.

8.3.1 CopyAllElementsAndConnections

The CopyAllElementsAndConnections function duplicates all model elements, attributes, visuals, and semantic connections from a source folder into a target folder. In Scenario 1, it is used twice: first to copy the original DEMO model into a synchronized DEMO Copy, and then to copy the BPMN Copy into the final BPMN folder. Each copied element is linked to its source via bridge elements, and all created elements and visuals are registered in the BridgeContext for subsequent synchronization.

Algorithm 8.2: CopyAllElementsAndConnections

Input: Source folder, target folder, diagram ID, bridge folder, direction flag, context `ctx`

```
foreach element in source folder do
    Create new model element in target folder;
    Copy attributes from original to new element;
    Create bridge element and connect original to copy;
    Register new element, bridge and connections in ctx;
end
foreach visual in source folder do
    Create new visual for copied element;
    Register visual in ctx;
end
foreach connection in source folder do
    Map source and target elements to copied IDs;
    if mapped source and target exist then
        Create new connection in target folder;
        Create visual connection in diagram;
        Register connection in ctx;
    end
end
```

Time Complexity

Let n be the number of model elements, k the average number of attributes per element, and m the number of semantic (non-bridge) connections.

Operation	Complexity	Notes
Element creation	$O(n)$	One call to `SaveModelElement` per element
Attribute copying	$O(n \cdot k)$	Each attribute saved via `SaveModelAttribute`
Visual creation	$O(n)$	One `SaveModelVisualElement` call per visual
Connection creation	$O(m)$	One `SaveModelConnection` call per connection
Visual connection creation	$O(m)$	One call per connection
Bridge creation	$O(n)$	For each element, we create a bridge element with 2 connections (One from source to bridge, one from bridge to target)
Total	$O(n \cdot k + m)$	Linear in model size

Table 8.3: Time Complexity of Copy Operation

As shown in Table 8.3, the function scales linearly with the number of elements and connections. Since most real-world models have moderate k and m, performance is suitable even for large diagrams.

Space Complexity

All elements, attributes, visuals, and connections are stored in memory via maps and lists inside the BridgeContext. Therefore, space complexity is $O(n + m)$, with additional $O(n)$ for bridge elements and connections.

Empirical Benchmarking

To empirically validate the expected linear complexity, benchmarks were run for varying model sizes. As shown in Table 8.4, the observed runtime increased approximately linearly with the number of elements and connections, supporting the theoretical complexity.

Each benchmark was executed using mocked save functions to isolate computation and memory effects from I/O delays. The input consisted of synthetic model elements with a moderate number of attributes and associated visual and semantic connections.

Elements	Connections	Avg. Time (ms)	Growth factor
100	50	2.24	1.0×
500	250	12.76	5.7×
1000	500	25.45	11.4×
2000	1000	58.43	26.1×

Table 8.4: Empirical runtime of the copy function across increasing input sizes

The results confirm the theoretical complexity of $O(n \cdot k + m)$.

API Call	Approximate Frequency
`SaveModelElement`	n elements + b bridge elements
`SaveModelAttribute`	$n \cdot k$
`SaveModelVisualElement`	n element visuals
`SaveModelConnection`	m semantic connections + $2b$ bridge links
`SaveModelVisualConnection`	m visual connections
Total API Calls	$\approx (2n + n{\cdot}k) + (2m + 3b)$

Table 8.5: Estimated frequency of API calls for `CopyAllElementsAndConnections`

Backend/API Performance

8.3.2 CreateBPMNFromDEMO

The `CreateBPMNFromDEMO` function transforms the copied/synchronized version of the DEMO model into a corresponding BPMN Synchronized model. It performs a full forward mapping of all Transaction Process Step Kind (TPSK) elements, Elementary Actor Roles, and their semantic connections. The function follows the synchronization rules defined in Chapter 7 and creates the appropriate BPMN elements, including Pools, Events, Activities, and DataObjects. Each element is registered in the shared `BridgeContext` for future synchronization.

Algorithm 8.3: CreateBPMNFromDEMO

```
   Input: Bridge context ctx with loaded DEMO Copy model
 1 Sort all DEMO Transaction-Process-Step-Kind (TPSK) elements into
    sortedSteps;
 2 foreach element in ctx.CopyDemoMap do
 3 |   if element is a DEMO Actor then
 4 |   |   Create a BPMN Pool and bridge it to the actor;
 5 |   end
 6 end
 7 foreach element in sortedSteps do
 8 |   Determine Step Kind;
 9 |   switch Step Kind do
10 |   |   case Initial do
11 |   |   |   Create Start Event;
12 |   |   end
13 |   |   case Request, Accept do
14 |   |   |   Create User Activity and C-Fact;
15 |   |   end
16 |   |   case Promise do
17 |   |   |   Create User Activity and C-Fact;
18 |   |   end
19 |   |   case Execute do
20 |   |   |   Create Manual Activity and P-Fact;
21 |   |   end
22 |   |   case State do
23 |   |   |   Create Service Activity and C-Fact;
24 |   |   end
25 |   |   case Requested, Promised, Stated do
26 |   |   |   Create Intermediate Message Event;
27 |   |   end
28 |   |   case Accepted do
29 |   |   |   Create End Message Event;
30 |   |   end
31 |   end
32 |   Bridge and register created BPMN elements;
33 end
34 foreach connection between DEMO steps do
35 |   if both ends are bridged then
36 |   |   Create and register a Sequence or Message Flow;
37 |   |   Link each Activity/Event to its corresponding Pool;
38 |   end
39 end
```

Time Complexity

Let n be the number of DEMO elements, k the number of TPSK steps, and m the number of semantic connections.

Operation	Complexity	Notes
Element separation	$O(n)$	Filter pools and TPSK steps
TPSK sorting	$O(k \cdot \log k)$	For layout and order enforcement
Pool creation	$O(p)$	Typically one per actor role
BPMN element creation	$O(k)$	Multiple elements per TPSK step
Attribute setting	$O(k)$	Per element, usually one per attribute
Connection creation	$O(m)$	Includes ActivityPoolLink creation
Visual creation	$O(k)$	One per BPMN element
Total	$O(n+p+k \cdot \log k+m)$	Efficient for structured flows

Table 8.6: Time Complexity of BPMN Creation from DEMO

As shown in Table 8.6, the function scales efficiently. The only non-linear factor is the sorting step, which arranges TPSK steps based on a fixed ordering. In practice, k remains moderate, and the function performs well even for large models.

Space Complexity

Each generated BPMN element, connection, and visual is stored in memory within the `BridgeContext`. Space complexity is therefore linear:

$$O(k + m)$$

Empirical Benchmarking

To empirically validate the performance and confirm the expected time complexity of $O(n + k \cdot \log k + m)$, we ran benchmarks with increasing numbers of TPSK steps and corresponding DEMO connections. Each synthetic input included realistic payloads, visual representations, and actor mappings. All save operations were mocked to isolate the algorithm's logic and memory consumption from backend overhead.

TPSK Steps	Connections	Avg. Time (ms)	Growth factor
100	99	8.00	1.0×
500	499	124.77	15.6×
1000	999	279.29	34.9×
2000	1999	881.54	110.2×

Table 8.7: Empirical performance of BPMN creation from DEMO model

The results confirm that the implementation scales as expected. The runtime increases linearly with the number of TPSK elements, though with higher constant factors because

each step produces multiple BPMN constructs (e.g., activities, events, data objects, and bridge links). Nevertheless, the growth remains predictable and well within acceptable limits. For instance, creating a BPMN model from 2000 TPSK steps with nearly 2000 connections took under 1 second on average. This confirms that the transformation can be executed interactively even for large-scale models.

Backend/API Performance

API Call	Approximate Frequency
`SaveModelElement`	n BPMN elements + b bridge elements
`SaveModelAttribute`	$n \cdot k$
`SaveModelVisualElement`	n element visuals
`SaveModelConnection`	m semantic connections + $2b$ bridge links
`SaveModelVisualConnection`	m visual connections
Total API Calls	$\approx\ (2n + n{\cdot}k)\ +\ (2m + 3b)$

Table 8.8: Estimated API-call frequency for `CreateBPMNFromDEMO`

8.4 Scenarios 2 and 3

For scenario 2 and 3 we need following functions:

8.4.1 CompareFolders

The `CompareFolders` function compares two versions of a model (original and synchronized) and identifies all structural and attribute-level differences. It supports both DEMO and BPMN models and works in either synchronization direction. The comparison includes both model elements and semantic connections. The result is a list of `Change` objects representing created, deleted, or updated elements and connections. It serves as the central mechanism for identifying deltas in both synchronization directions.

Algorithm 8.4: CompareFolders

Input: Preloaded maps from `ctx`: original/copy elements, bridges, and all connections
Output: List of Change objects (Created, Updated, Deleted)

```
1  Initialize empty list changes;
2  Initialize sets for matchedOriginal, matchedCopy, createdElementIDs,
     deletedElementIDs;
3  // Step 1: Compare Model Elements;
4  foreach mapped element pair (original → copy) do
5      if element type is not Diagram then
6          Mark both elements as matched;
7          Compare attributes using CompareAttributes;
8          if any attribute differs then
9              Add Updated change with list of modified attributes;
10         end
11     end
12 end
13 foreach element in copy map do
14     if element is unmatched and not a Diagram then
15         Add Deleted change for element;
16     end
17 end
18 foreach element in original map do
19     if element is unmatched and not a Diagram then
20         Add Created change for element;
21     end
22 end
23 // Step 2: Compare Semantic Connections;
24 Filter out bridge-related connections from ctx.AllConnections;
25 Build map of valid connections in original and copy models;
26 foreach connection in original model do
27     Map source and target to copy IDs using bridge map;
28     if corresponding connection is missing in copy then
29         Add Created change for connection;
30     end
31 end
32 foreach connection in copy model do
33     Map source and target to original IDs using bridge map;
34     if corresponding connection is missing in original then
35         Add Deleted change for connection;
36     end
37 end
38 return changes
```

To enable consistent and traceable synchronization between DEMO and BPMN models, we introduce a unified way of representing detected differences: the Change struct. Every change found between models is encapsulated in one of these Change objects. This approach standardizes the way differences are handled throughout the synchronization logic.

Each Change describes what type of element was affected (ModelElement or ModelConnection), what kind of change occurred (Created, Deleted, or Updated), and which specific model object was involved. It also includes the Identification of the Element or the Connection. If the change involves a connection, the struct includes information about the connection type, source and target IDs. For updates, a list of attribute-level differences is included via the Changes field, which holds all modified name-value pairs.

This uniform structure allows update functions to process all changes in the same way, whether they're updating elements, attributes, or connections. It also serves as the basis for analyzing the complexity of the entire synchronization mechanism, since the number of Change objects directly reflects how many model modifications need to be processed.

Time Complexity

Let n be the number of model elements, k the average number of attributes per element, and m the number of semantic (non-bridge) connections.

Operation	**Complexity**	**Notes**
Element comparison	$O(n \cdot k)$	Includes attribute differentiation and identity checks
Connection comparison	$O(m)$	Filters, indexes, and compares connections
Total	$O(n \cdot k + m)$	Linear in structure and size of the model

Table 8.9: Time Complexity of Folder Comparison

Space Complexity

Temporary maps for matched IDs and attributes, as well as connection maps, are built in memory. Thus, the space complexity is linear in the number of elements and connections:

$$O(n + m)$$

Empirical Benchmarking

To empirically validate the performance and scalability of the `CompareFolders` function, we conducted again a benchmark using synthetic DEMO models of increasing size.

As shown in Table 8.10, the runtime increases in line with model size. Although the complexity includes both $O(n \cdot k)$ for attribute comparisons and $O(m)$ for connection mapping, the function completes the comparison of 2,000 elements with 1,000 semantic

TPSK Steps	Connections	Avg. Time (ms)	Growth factor
100	50	0.29	1.0×
500	250	1.73	6.0×
1000	500	3.89	13.4×
2000	1000	11.12	38.3×

Table 8.10: Empirical performance of the `CompareFolders` function

connections in under 12 milliseconds on average. This confirms that the function performs efficiently and is suitable for interactive use even with large models.

Backend/API Performance

This function performs all comparisons in-memory and does not trigger any API calls. All model data is assumed to be preloaded via `BridgeContext`.

8.4.2 CopyChangesToFolder

The CopyChangesToFolder function applies the list of detected changes to the target model folder (DEMO or BPMN, original or synchronized version). It is used in both Scenario 2 and Scenario 3, depending on which model serves as the source. For each change, the function creates or deletes model elements, updates attributes, handles visual representations, and ensures that all bridge connections and internal maps within the BridgeContext are kept consistent.

Algorithm 8.5: CopyChangesToFolder

Input: BridgeContext `ctx`, list of `Change` objects, target folder ID
Output: Model folder updated with all changes (create, update, delete)

```
// Step 1: Delete Connections
foreach change in changes do
    if change is a deleted ModelConnection then
        Delete connection from backend;
        Remove it from all context maps and lists in ctx;
    end
end
// Step 2: Apply Element Changes
foreach change in changes do
    if change is a deleted ModelElement then
        Delete element and visual;
        Clean up bridge mappings and remove from maps;
    else if change is a created ModelElement then
        Create element and visual from original;
        Create bridge and update context maps;
    end
    else if change is an updated ModelElement then
        Update attributes and save changes;
        Replace or update context entries;
    end
end
// Step 3: Create New Connections
foreach change in changes do
    if change is a created ModelConnection then
        Resolve mapped source and target elements;
        Create semantic and visual connection;
        Register in context maps;
    end
end
```

Time Complexity

Let n be the number of changed elements, k the average number of attributes per updated element, and m the number of changed connections.

The complexity is driven by the number of changes rather than full model size. Efficient use of maps ensures fast lookup and insertion operations.

Operation	Complexity	Notes
Element creation	$O(n)$	One call to `SaveModelElement` per element
Visual creation	$O(n)$	Only if a corresponding visual exists
Element deletion	$O(n)$	Lookup + removal from model and maps
Attribute updates	$O(n \cdot k)$	Set each changed value via API
Connection updates	$O(m)$	Created or deleted connections + visuals
Bridge creation	$O(n)$	One bridge per created element (if needed)
Total	$O(n \cdot k + m)$	Depends on number of deltas

Table 8.11: Time Complexity of Applying Changes

Space Complexity

All comparisons and changes operate in-memory, assuming that all elements, visuals, and connections are preloaded into the BridgeContext. No on-demand fetches are performed. Space complexity is $O(n + m)$, reflecting temporary storage of change results.

Benchmark and Runtime Behavior

To evaluate the performance and scalability of the `CopyChangesToFolder` function, we conducted a benchmark using synthetic change sets of varying sizes. Each run applied a realistic mix of model changes, including element creation, updates, and deletions, on a pre-initialized context containing a base model with mapped elements and visuals.

For each model size, the change set consisted of:

- 60% newly created model elements,
- 20% updated elements (e.g., identification changed),
- 20% deleted elements.

All benchmark executions used mocked save and delete functions to simulate backend interaction while preserving realistic computational overhead (e.g., context updates, ID mapping, bridge handling).

Changes Applied	Base Elements	Avg. Time (ms)	Growth factor
100	50	1.01	1.0×
500	250	7.18	7.1×
1000	500	16.63	16.5×
2000	1000	40.98	40.6×

Table 8.12: Empirical runtime performance of `CopyChangesToFolder` on mixed change sets

The results demonstrate that the function exhibits predictable scaling behavior consistent with its theoretical complexity of $O(n \cdot k + m)$. Even for 2,000 mixed changes, the operation completes in under 41 milliseconds on average, confirming that the function is suitable for incremental synchronization in large models.

Backend/API Performance

The following API calls are triggered when applying each type of change, depending on its operation. Table 8.13 summarizes the estimated frequency of each call:

API Call	Approximate Frequency
`SaveModelElement`	n_c created elements
`SaveModelAttribute`	$n_u \cdot k$ updated attributes
`SaveModelVisualElement`	n_c element visuals
`SaveModelConnection`	m_c created connections
`SaveModelVisualConnection`	m_c visual connections
`DeleteModel`	$n_d + m_d$ deletions (elements and connections)
Total API Calls	$\approx\ 2n_c + n_u\,k + 2m_c + n_d + m_d$

Table 8.13: Estimated API-call frequency for `CopyChangesToFolder`

Here, n_c is the number of created elements, n_u is the number of updated elements, n_d is the number of deleted elements, k is the average number of updated attributes per element, m_c is the number of created connections, and m_d is the number of deleted connections.

8.4.3 Apply DEMO Changes to BPMN

This synchronization step applies changes from the DEMO Copy model to the BPMN Copy model. It handles newly created, updated, and deleted elements and connections, and ensures the BPMN model remains consistent. The following core functions are executed in sequence to apply the detected changes:

- `ApplyCreateAndUpdateChangesToBPMN`
- `ApplyConnectionCreationsToBPMN`
- `ApplyConnectionDeletionsToBPMN`
- `ApplyDeletionChangesToBPMN`
- `CreateBPMNElementFromDemo`
- `deleteBPMNElementAndConnections`

Together, these functions form the logic for propagating structural and behavioral changes across the models.

Algorithm 8.6: ApplyDEMOChangesToBPMN(ctx, demoChanges)

Data: BridgeContext `ctx`, list of DEMO-side `demoChanges`

```
1 ApplyConnectionDeletionsToBPMN(ctx, demoChanges);
2 ApplyDeletionChangesToBPMN(ctx, demoChanges);
3 CopyChangesToFolder(ctx, demoChanges, ctx.CopyDemoFolderId);
4 ApplyCreateAndUpdateChangesToBPMN(ctx, demoChanges);
5 ApplyConnectionCreationsToBPMN(ctx, demoChanges);
```

This composite update pipeline ensures that all structural and attribute-level modifications are propagated to the BPMN model in a consistent and conflict-free manner.

Time Complexity

Let n be the number of changed elements, k the average number of updated attributes per element, and m the number of changed connections.

Operation	Complexity	Notes
Mapping and creation of BPMN elements	$\mathcal{O}(n)$	Includes visual creation and bridge registration
Attribute updates (SetSingleAttribute)	$\mathcal{O}(n \cdot k)$	Per changed attribute
BPMN element deletion	$\mathcal{O}(n)$	Includes visual and connection cleanup
Connection creation/deletion	$\mathcal{O}(m)$	Per semantic + visual connection
Total	$\mathcal{O}(n \cdot k + m)$	Linear in number of changes

Table 8.14: Time Complexity of BPMN Update Logic

Space Complexity

The space requirements are minimal as updates work in-place within the `BridgeContext`. Each created or modified element is registered in existing maps and lists. Overall space complexity: $\mathcal{O}(n + m)$.

Benchmark and Runtime Behavior

To evaluate the performance and scalability of the BPMN update logic, we conducted a benchmark using synthetic DEMO change sets of varying sizes. Each run applied a realistic mix of model modifications, including element creation, updates, and deletions, on a pre-initialized context with 1:1 mapped DEMO and BPMN elements, including bridge connections and visuals.

For each input size, the change set consisted of:

- 50% updated DEMO elements (e.g., Identification or Attribue value changed),
- 25% deleted elements,

- 25% newly created elements.

All benchmark executions used mocked save and delete functions to isolate backend performance and simulate realistic in-memory synchronization behavior (e.g., bridge cleanup, attribute handling, visual and connection management).

Changes Applied	Base Elements	Avg. Time (ms)	Growth factor
100	50	0.43	1.0×
500	250	3.32	7.7×
1000	500	8.20	19.0×
2000	1000	17.82	41.5×

Table 8.15: Empirical runtime of BPMN update from DEMO changes under mixed workloads

The results confirm that the implementation scales consistently with the expected complexity of $O(n \cdot k + m)$. The total runtime increases linearly with the number of elements, but with higher constant factors due to the fixed set of operations required per element (e.g., multiple attribute updates, bridge links, visuals, and connections). Since all benchmarks used mocked backend functions, the results isolate internal logic performance. Still, the overall trend supports the correctness of the calculated complexity class.

Backend/API Performance

API Call	Approximate Frequency
`SaveModelElement`	$n_c + n_u$ elements (created + updated)
`SaveModelAttribute`	$n_u \cdot k$ updated attributes
`SaveModelVisualElement`	n_c visuals for created elements
`SaveModelConnection`	m_c created connections
`SaveModelVisualConnection`	m_c visuals for created connections
`DeleteModel`	$n_d + m_d$ deletions (elements + connections)
Total API Calls	$\approx (2n_c + n_u k) + (2m_c) + (n_d + m_d)$

Table 8.16: Estimated API-call frequency for `Apply DEMO Changes → BPMN`

This update pipeline forms the core of Scenario 2, ensuring that the BPMN model remains synchronized with changes made to the DEMO model, while maintaining consistency across semantic and visual representations.

8.4.4 Update of DEMO based on BPMN Changes

This synchronization scenario ensures that structural and behavioral changes made to the BPMN Copy model are accurately propagated to the DEMO Copy model. It handles

creation, update, and deletion of both elements and connections based on the calculated delta. The following core functions are responsible for this behavior:

- `ApplyCreateAndUpdateChangesToDEMO`
- `ApplyDeletionChangesToDEMO`
- `createDEMOFromNewBPMNElement`
- `createActorFromPool`
- `promptAndCreateTPSK`

Each function is responsible for a specific change type, and together they ensure all structural and behavioral changes from the BPMN Copy are consistently propagated to the DEMO Copy.

Algorithm 8.7: UpdateDEMOFromBPMN(ctx, bpmnChanges)

Data: BridgeContext `ctx`, list of BPMN-side `bpmnChanges`

```
1 ApplyDeletionChangesToDEMO(ctx, bpmnChanges);
2 CopyChangesToFolder(ctx, bpmnChanges, ctx.CopyBpmnFolderId);
3 ApplyCreateAndUpdateChangesToDEMO(ctx, bpmnChanges);
```

This algorithm ensures a complete cycle from change detection to application, including both forward and reverse synchronization steps within the Synchronized-Original model pair.

Time Complexity

Let n be the number of created or deleted BPMN elements, k the average number of attributes inferred or updated per DEMO element, and m the number of changed connections.

Operation	Complexity	Notes
BPMN → DEMO element creation	$\mathcal{O}(n)$	Includes inference, visual, and bridge creation
Attribute inference and application	$\mathcal{O}(n \cdot k)$	e.g., Step Kind, Name, Task Type
DEMO element deletion	$\mathcal{O}(n)$	Includes visual and bridge cleanup
DEMO connection creation/deletion	$\mathcal{O}(m)$	Per visual + semantic connection
Total	$\mathcal{O}(n \cdot k + m)$	Efficient handling of user-validated inference

Table 8.17: Time Complexity of DEMO Update Logic

Space Complexity

All elements, visuals, and connections are updated in-place within the `BridgeContext`. Intermediate maps for bridging and lookup are reused throughout the update cycle.
Overall space complexity: $\mathcal{O}(n + m)$.

Benchmark and Runtime Behavior

To evaluate the performance and scalability of the DEMO update logic (Scenario 3), we conducted a benchmark using synthetic BPMN change sets of varying sizes. Each run applied a realistic mix of model modifications on a pre-initialized context with 1:1 mapped BPMN and DEMO elements, including bridge connections and visuals.

For each input size, the change set consisted of:

- 50% updated BPMN elements,
- 25% deleted elements,
- 25% newly created elements.

All benchmark executions used mocked save and delete functions to isolate backend performance and simulate realistic synchronization behavior, including attribute inference, pool-to-actor mapping, bridge management, and visual layout computation.

Changes Applied	Base Elements	Avg. Time (ms)	Growth factor
100	50	0.65	1.0×
500	250	4.12	6.4×
1000	500	9.83	15.2×
2000	1000	20.01	30.8×

Table 8.18: Empirical runtime of DEMO update from BPMN changes under mixed workloads

The results confirm that the DEMO update logic performs efficiently and scales as expected with the number of changes. Even when applying 2,000 mixed changes, the update completes in just over 20 milliseconds on average. This confirms that interactive synchronization from BPMN to DEMO remains feasible even for large-scale models.

Backend/API Performance

Since all model updates are persisted through the Sirius Web backend, each creation, deletion, or modification triggers API calls. The following table summarizes their expected frequency per update cycle:

API Call	**Approximate Frequency**
`SaveModelElement`	$n_c + n_u$ elements (created + updated)
`SaveModelAttribute`	$n_u \cdot k$ inferred or modified attributes
`SaveModelVisualElement`	n_c visuals for new elements
`SaveModelConnection`	m_c created connections
`DeleteModel`	$n_d + m_d$ deletions (elements + connections)
Total API Calls	$\approx (2n_c + n_u k) + (m_c) + (n_d + m_d)$

Table 8.19: Estimated API-call frequency for `UpdateDEMOFromBPMN`

CHAPTER 9

Evaluation

In this chapter, we evaluate the performance and scalability of the three synchronization scenarios in the prototype based on the implemented logic presented in Chapter 8. While the design aims to support efficient and maintainable bidirectional synchronization between DEMO and BPMN models, it is important to empirically validate how the system works under realistic conditions. This includes measuring real runtime behavior (including the correct API calls), identifying potential bottlenecks, and discussing the system's current limitations.

9.1 Performance Benchmarks

9.1.1 Benchmark Scenarios

To evaluate the overall performance of the synchronization prototype under realistic conditions, we defined three benchmark tests, one for each synchronization scenario. Unlike the function-specific benchmarks in Chapter 8, which were executed using mocked backend calls to isolate internal computation, these tests were performed against the actual SMP backend. This setup provides a more accurate picture of end-to-end runtime behavior, including the full overhead of API interactions.

- Scenario 1: Generate BPMN from DEMO
- Scenario 2: Update BPMN based on DEMO changes
- Scenario 3: Update DEMO based on BPMN changes

For each scenario, we used models of different sizes, from small (around 10 transaction steps) to medium (50 steps) and large (about 100 steps). The runtime was measured

using the built-in timing tools in Go, and each test was run multiple times to reduce fluctuations. These measurements help us understand how well the system scales and whether performance remains acceptable even with growing model sizes.

9.1.2 Runtime Measurements

Table 9.1 summarizes the measured execution times for each scenario. All runtimes in this table are measured in seconds (s).

Model Size	Scenario 1 (s)	Scenario 2 (s)	Scenario 3 (s)
10 TPSK steps	22.5499	17.5479	14.5501
50 TPSK steps	101.1064	81.587	48.3815
100 TPSK steps	209.3951	131.6153	86.2833

Table 9.1: Synchronization Runtime by Scenario and Model Size

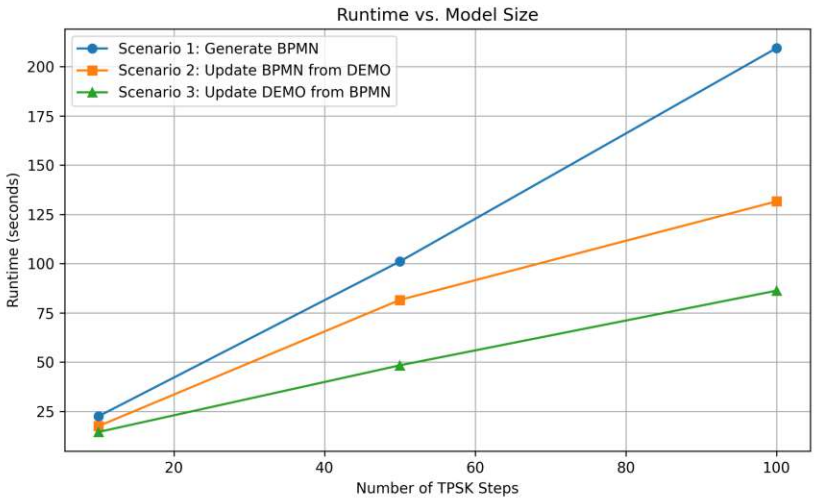

Figure 9.1: Measured runtime of each synchronization scenario across increasing model sizes.

As the results in Table 9.1 and Figure 9.1 show, the runtime increases consistently across all three scenarios as the number of elements and changes in the model grows. For each case, the original model contained 10, 50, or 100 transaction steps (TPSK steps), and for Scenario 2 and 3 the same number of changes were randomly applied before synchronization. This setup allows us to observe how the system behaves as both model complexity and update workload increase.

Scenario 1, in which a completely new BPMN model is generated from a DEMO model, shows the highest runtime in all test cases. For example, generating a BPMN model from 100 DEMO steps takes over 209 seconds. This is expected, as this scenario always

creates all elements, visuals, and connections from scratch, including setting attributes and generating bridge mappings. The effort scales with the total size of the DEMO model.

Scenario 2, where BPMN is updated based on changes in DEMO, performs slightly better but still shows significant runtime growth. Here, the system compares the original and modified DEMO models, detects the changes, and applies updates to the BPMN model accordingly. For 100 changes, the process takes around 131 seconds. Although only part of the model is modified, each change can involve multiple BPMN elements (e.g., tasks, message events, data objects) and may require removing and recreating elements, which makes this scenario relatively expensive.

Scenario 3, which updates DEMO based on changes in the BPMN model, performs best across all sizes. Even for 100 changes on 100 elements, the runtime stays below 90 seconds. This is mainly due to the more compact and stable structure of DEMO models. Since multiple BPMN elements can correspond to a single DEMO step, each change on the BPMN side typically affects only one DEMO element - or in some cases, none at all. For instance, if a BPMN DataStore is deleted, this does not require any update in the DEMO model. As a result, fewer operations are needed during synchronization.

Overall, the runtime results show that the system handles small to medium-sized models well, but synchronization costs grow noticeably with the number of elements and changes, especially in scenarios involving BPMN generation or updates. From a scalability perspective, Scenario 1 and Scenario 2 exhibit roughly linear growth (O(n)) in runtime as the number of elements or changes increases, while Scenario 3 scales more efficiently due to fewer element reconstructions and simpler transformation logic on the DEMO side.

9.2 Correctness and Robustness

Beyond performance, it is also important to evaluate whether the prototype consistently delivers correct and reliable synchronization results across the tested scenarios. This includes verifying that elements are transformed according to the defined rules, that bridge connections remain consistent and that the system behaves predictably even with repeated or complex change processes.

During testing, the system consistently maintained the intended one-to-one or one-to-many mappings between DEMO and BPMN elements. In Scenario 1, all BPMN elements generated from DEMO models were complete, visually placed and correctly linked to their source elements. For example, Request and Promise steps reliably led to corresponding tasks and data objects, and pools and events were correctly created and positioned based on actor roles and transaction contexts.

Scenario 2 showed that changes in the DEMO model were accurately transferred to the BPMN model. The system correctly transformed new steps into their BPMN equivalents, updated existing elements based on attribute changes and removed elements and visual representations when they were deleted. Semantically, the synchronization matched

the defined synchronization logic. Visually, most elements were placed in the correct locations, especially when contextual connections were present. In some cases, however, the placement was less precise.

Scenario 3 also provided correct results when synchronizing BPMN to DEMO. All created, updated and deleted BPMN elements were mapped to their DEMO equivalents as intended. The resulting DEMO models were semantically valid and structurally consistent across all test cases. As with Scenario 2, visual placement generally followed the expected order when connections provided sufficient context, while isolated elements were added in default positions. Nevertheless, no semantic inconsistencies were observed in any of the test runs.

In terms of robustness, the system proved stable on repeated runs and supported different orders of creates, updates and deletes. Changes could be made in any order, and the synchronization logic correctly maintained bridge consistency even when multiple dependent elements were changed. In no case did the prototype result in broken connections, invalid references or invalid model states.

Overall, the evaluation confirms that the synchronization logic is correct and robust within the defined scope. The system handles typical modeling changes in a reliable and predictable way, maintaining consistency between DEMO and BPMN models.

9.3 Unit Tests and Test Coverage

To ensure that the synchronization logic is reliable and robust, we developed unit tests for all core helper functions. These tests validate the correctness of attribute handling, element creation, connection management, bridge logic, and data structure initialization.

Tests are organized in the `Helper_functions_test` package and correspond to each major component in the `Helper_functions` package. This includes `compare_test.go`, `copy_test.go`, `create_bpmn_test.go`, and others. Each test targets a specific function or logic module and verifies the internal behavior using synthetic input data. The system state is validated via the `BridgeContext` structure, ensuring that expected model elements, connections, visuals, and bridges are correctly registered or updated.

All tests use mocked versions of the Sirius Web API, enabling isolated validation of logic without triggering backend calls. The `Mocks.go` file provides reusable stubs and simulated responses for typical API operations, such as saving or deleting model elements and visuals.

The test suite includes not only successful execution scenarios but also a range of edge cases and failure conditions. This includes:

- Changes with missing bridge mappings (e.g., a new BPMN element without corresponding DEMO mapping),

- Connection deletions where either the source or target element is missing from the model,
- Visual creation errors when no diagram is set or when element IDs are invalid,
- Attribute mismatches between original and copy elements,
- Partial or duplicate model state due to inconsistent ID registration,
- Unsupported element types (e.g., unknown BPMN or DEMO types),
- Prompt-based logic for Step Kind inference in ambiguous BPMN insertions.

Testing these edge cases helps verify that the system handles unexpected or incomplete input reliably and does not break under inconsistent states.

Based on the coverage analysis tools in the IntelliJ IDEA, the unit test suite achieves 100% file coverage and approximately 70.5% statement coverage. Most synchronization-critical modules, such as `compare.go`, `creation.go`, and `update_bpmn.go`, exceed 90% statement coverage. Only supporting infrastructure files like `BridgeContext.go`, which contain basic data initialization, have significantly lower coverage. This confirms that the essential transformation and synchronization logic is well-covered and reliably tested.

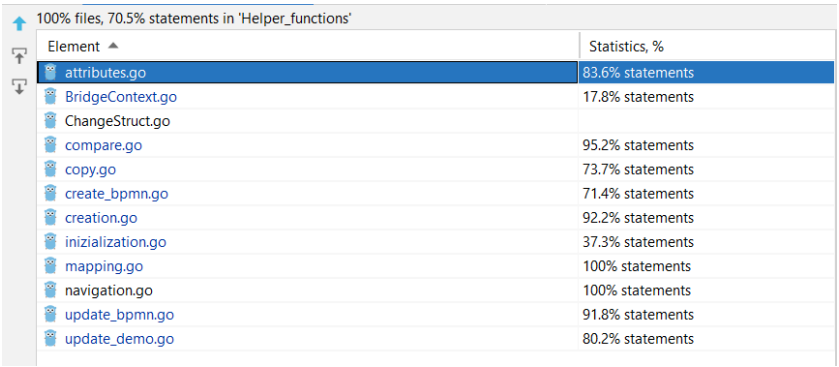

Figure 9.2: Test coverage of helper functions (visualized in GoLand).

Overall, the unit tests played an important role in the implementation and debugging of the prototype. In practice, these tests helped uncover issues like missing attribute propagation or broken bridge links early in development. As the synchronization logic was refactored and extended, the tests ensured consistent system behavior and helped validate incremental improvements.

9.4 Limitations and Future Work

While the prototype shows that synchronization between DEMO and BPMN models is possible, several limitations became apparent during development and evaluation.

Originally, we intended to evaluate the system using models with up to 1000 TransactionProcessStepKind (TPSK) elements. However, we observed that the platform becomes increasingly unstable at this scale. In particular, Scenario 1 resulted in long response times due to the high number of semantic and visual elements created. Scenario 1 took almost 40 minutes to run. When executing Scenario 2 on these models, the system broke down (it was not able to run). This indicates performance bottlenecks in the underlying modeling platform that should be addressed to support large-scale synchronization in the future.

Another limitation concerns how changes between models are detected. Currently, the implementation compares all items in the original and synchronized folders to identify created, updated and deleted items. This comparison is very computationally intensive. While the simplified platform supports timestamps for the creation and last modification of elements, attributes and connections, it does not support timestamps for deletions. This makes efficient detection of deleted elements impossible. A more robust change tracking mechanism, including deletion metadata, would improve performance and scalability.

The prototype also relies on structural context (e.g. connections to previous or next steps) to determine the visual placement of elements. When new elements are added without such context, they are placed in a default position. As a result, some diagrams appear visually cluttered or misaligned. Future work could improve the visual layout strategies to ensure clearer positioning even in context-free cases.

The current implementation focuses only on the "happy flow" of DEMO transactions. Future work could extend the prototype to cover the entire structure of the transaction diagram, including edge cases such as rollback or cancellation scenarios.

Although the prototype successfully demonstrates DEMO-BPMN synchronization, the logic was developed on a language-specific basis. In future work, the synchronization approach could be generalized to support other modeling languages such as ArchiMate or e^{3}value. This would contribute to a broader vision of cross-language model synchronization in enterprise modeling.

Another limitation is the lack of a user interface for interacting with the synchronization logic. At the moment, the prototype must be executed manually as a backend script. This means that end users cannot trigger synchronization directly from the modeling environment. A useful future enhancement would be to integrate a simple button or UI control into the Web frontend that allows users to start the synchronization process on demand.

Finally, the runtime and performance evaluation was based on test models that were manually created for the purpose of evaluation. While this approach is useful for benchmarking and testing the system under controlled conditions, these models do not reflect the complexity, structure, or modeling styles found in real-world projects. We did not have access to actual DEMO or BPMN models from companies, which limits the practical relevance of the evaluation. As a result, the findings may not fully capture the challenges that arise in real modeling scenarios. Future evaluations using real enterprise

models would provide more meaningful insights into the prototype's applicability in practice.

CHAPTER 10

Conclusion

In this thesis, we set out to address a common challenge in enterprise modeling: how to keep models in different languages, like DEMO and BPMN, consistent and synchronized. Rather than creating yet another new language, we focused on connecting two existing ones through a metamodel-based synchronization approach that respects their individual structures and semantics.

We developed a working prototype that supports bidirectional synchronization between DEMO and BPMN models. The system can detect changes, apply transformations, while maintaining visual and semantic consistency using bridge mappings and structured synchronization rules. Our focus was not on one-time transformations, but on enabling continuous synchronization, supporting ongoing model evolution in both directions.

Throughout the process, we combined theoretical foundations with practical implementation, using the Simplified Modeling Platform to build and test our approach. We followed principles from design science [10] and algorithm engineering [11] to iteratively design, evaluate, and improve the prototype. Our system supports both automated synchronization, where semantics are clear, and manual guidance where ambiguity exists.

The evaluation showed that our approach performs well for small to medium-sized models, correctly applying changes and preserving model structure and traceability. At the same time, we identified limitations in scalability, visual placement logic, and platform constraints.

Overall, we demonstrated that model synchronization between different languages is feasible.

All source code, unit tests, and benchmarking scripts used in this thesis are available in the corresponding GitLab repository [42].

10.1 Recap of research questions

At the beginning of this thesis (Chapter 1), we introduced five research questions on which we build the design and development of the synchronization prototype. These questions covered both conceptual and technical challenges involved in keeping DEMO and BPMN models aligned. In this section, we revisit each question and explain how it was addressed throughout the work.

RQ1: Which transaction-related elements in the DEMO and BPMN metamodels can be mapped to each other in a semantically meaningful and technically feasible way, and what synchronization rules are needed to support this mapping?

In order to synchronize DEMO and BPMN models, we first needed to define clear and consistent mappings between the two languages. This was done based on DEMO's Complete Transaction Pattern, which provides a structured way of describing coordination between actors. For each DEMO transaction step (e.g., *Request*, *Promise*, *Execute*, *Accept*), we identified corresponding BPMN elements such as Activities, Events, Pools, and DataObjects. These mappings were formalized as synchronization rules, which are applied consistently in both directions. The full overview is shown in Table 7.1.

RQ2: How can model differences - such as added, removed, or modified elements, attributes, and connections - be detected using a delta-based comparison approach to support bidirectional synchronization?

To keep track of what has changed between two models, the prototype compares the current version with its last synchronized snapshot. This comparison is done at the semantic level, so we compare actual model elements, not visuals. As a result we get a categorized list of changes: elements or connections that were created, deleted, or updated. This delta-based approach is used in both directions and ensures that the system only processes what actually changed. The logic for this comparison is explained in Chapter 6 and is the foundation for the incremental synchronization used in Scenarios 2 and 3.

RQ3: How can detected changes be transformed and applied to the target model in a way that updates semantics, visuals, and bridge mappings consistently?

After identifying what has changed in one model, the next step is to apply those changes to the other model in a consistent and structured way. This involves more than just copying over elements - it also means updating their attributes, placing them correctly in the diagram, and maintaining the bridge connections that keep both sides in sync. To handle this, the prototype uses a set of modular functions that take care of the different tasks step by step: from semantic transformation to visual placement to updating internal mappings. These functions were designed to work together smoothly and ensure that

every change is applied in a traceable and predictable way. The full process is explained in Chapter 8.

RQ4: Which types of changes can be synchronized automatically, and in which cases is manual intervention required due to ambiguity or model-specific constraints (particularly in BPMN-to-DEMO transformations)?

Not all transformations can be done fully automatically. For the direction from DEMO to BPMN, the mapping is straightforward because DEMO models include clear semantics about each transaction step. This allows the system to generate BPMN elements without user input. The reverse direction, from BPMN to DEMO, is more complex. BPMN elements often lack the semantic detail needed to infer exactly what kind of transaction step they represent. In these cases, the system suggests a likely option based on surrounding elements (e.g., incoming and outgoing connections), but still asks the user to confirm or correct it. This interactive behavior, described in Chapter 7, strikes a balance between automation and correctness.

RQ5: What is the complexity of the resulting synchronization mechanism?

To evaluate the scalability and performance of the system, we analyzed the complexity of the main synchronization functions and ran a series of runtime benchmarks (see Chapter 8and 9). Most core operations, such as detecting changes, applying transformations, and maintaining bridge mappings, scale linearly with the number of elements or connections involved. The benchmark results confirm that the system can handle realistic model sizes efficiently. In addition, the development process followed the principles of Algorithm Engineering, using iteration and testing to refine performance and reliability.

In summary, each of the research questions has been addressed both in theory and in practice. The resulting prototype shows that continuous synchronization between DEMO and BPMN models is not only technically feasible, but can also be implemented in a way that is robust, traceable, and adaptable for future extensions and modeling languages.

Overview of Generative AI Tools Used

I used ChatGPT (OpenAI, GPT-4, version June 2025) as a supporting tool for this thesis project. It helped me to understand certain aspects of the Go programming language, as I had not worked with it before. It also gave me guidance in structuring the whole thesis, e.g. in organizing the chapters and refining the formulation of the research questions. I also occasionally used the it to improve sentence structure and check grammar. Nevertheless, all technical ideas, transformational logic and written content were developed by me and carefully reviewed before inclusion.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] J. L. G. Dietz and H. B. F. Mulder, *Enterprise Ontology: A Human-Centric Approach to Understanding the Essence of Organisation*. Springer, 2024.

[2] M. Dumas, M. L. Rosa, J. Mendling, and H. A. Reijers, *Fundamentals of Business Process Management*. Berlin, Germany: Springer, 2 ed., 2018.

[3] T. O. Group, *ArchiMate® 3.2 Specification*. The Open Group, 2023.

[4] G. Doumeingts, D. Chen, B. Vallespir, and J.-L. Faure, "Interoperability in enterprise modelling: Requirements and roadmap," *Annual Reviews in Control*, vol. 31, no. 2, pp. 221–231, 2007.

[5] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *Future of Software Engineering (FOSE '07)*, pp. 37–54, 2007.

[6] A. Albani and J. L. G. Dietz, "Requirements for supporting enterprise interoperability in dynamic environments," in *Enterprise Interoperability*, pp. 345–356, Springer, 2006.

[7] D. Cetinkaya and A. Verbraeck, "Metamodeling and model transformations in modeling and simulation," in *Proceedings of the 2011 Winter Simulation Conference (WSC)*, pp. 3048–3058, IEEE, 2011.

[8] D. Van Nuffel, H. Mulder, and S. Van Kervel, "Enhancing the formal foundations of bpmn by enterprise ontology," in *International Workshop on Cooperation and Interoperability, Architecture and Ontology*, pp. 115–129, Springer, 2009.

[9] S. Guerreiro and J. Dietz, "Demo enhanced bpmn," *arXiv preprint arXiv:2410.08215*, 2024.

[10] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004.

[11] J. Mendling, B. Depaire, and H. Leopold, "Theory and practice of algorithm engineering," *CoRR*, vol. abs/2107.10675, 2021.

[12] J. Mendling, H. Leopold, H. Meyerhenke, and B. Depaire, "Methodology of algorithm engineering," *CoRR*, vol. abs/2310.18979, 2023.

[13] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *Journal of management information systems*, vol. 24, no. 3, pp. 45–77, 2007.

[14] J. L. G. Dietz, *Enterprise Ontology: Theory and Methodology*. Springer, 2006.

[15] M. Op't Land and J. L. Dietz, "Benefits of enterprise ontology in governing complex enterprise transformations," in *Advances in Enterprise Engineering VI: Second Enterprise Engineering Working Conference, EEWC 2012, Delft, The Netherlands, May 7-8, 2012. Proceedings 2*, pp. 77–92, Springer, 2012.

[16] F. Huňka, "DEMO methodology and its potential for collaboration," in *AIP Conference Proceedings*, vol. 2186, p. 060007, 2019.

[17] C. Décosse, W. A. Molnar, and H. A. Proper, "What does demo do? a qualitative analysis about demo in practice: founders, modellers and beneficiaries," in *Advances in Enterprise Engineering VIII: 4th Enterprise Engineering Working Conference, EEWC 2014, Funchal, Madeira Island, Portugal, May 5-8, 2014. Proceedings 4*, pp. 16–30, Springer, 2014.

[18] P. Naplava and R. Pergl, "Empirical study of applying the demo method for improving bpmn process models in academic environment," in *2015 IEEE 17th Conference on Business Informatics*, vol. 2, pp. 18–26, IEEE, 2015.

[19] J. L. G. Dietz and M. O. Land, "The adoption of demo: A research agenda," in *Enterprise Architecture and Business Transformation*, Lecture Notes in Business Information Processing, pp. 231–242, Springer, 2010.

[20] "Business process model and notation (bpmn) version2.0.2 – execution semantics." `https://www.omg.org/spec/BPMN/2.0.2/`, 2014. Formalizes the execution semantics for all BPMN elements.

[21] B. Silver and B. Richard, *BPMN method and style*, vol. 2. Cody-Cassidy Press Aptos, 2009.

[22] O. Mráz, P. Náplava, R. Pergl, and M. Skotnica, "Converting demo psi transaction pattern into bpmn: a complete method," in *Advances in Enterprise Engineering XI: 7th Enterprise Engineering Working Conference, EEWC 2017, Antwerp, Belgium, May 8-12, 2017, Proceedings 7*, pp. 85–98, Springer, 2017.

[23] M. D. Vries and D. Bork, "Identifying scenarios to guide transformations from demo to bpmn," *D. Aveiro, D. Guizzardi, R. Pergl, H. Proper (Eds.): 10th Enterprise Engineering Working Conference (EEWC 2020)*, 2021.

[24] K. Grigorova and K. Mironov, "Comparison of business process modeling standards," *Int. J. Eng. Sci. Manag. Res*, vol. 1, no. 3, pp. 1–8, 2014.

[25] S. de Kinderen, K. Gaaloul, and H. A. Proper, "Bridging value modelling to archimate via transaction modelling," 2014.

[26] S. de Kinderen, K. Gaaloul, and H. E. Proper, "Transforming transaction models into archimate," 2012.

[27] S. Boukelkoul and R. Maamri, "Optimal model transformation of bpmn to devs.," in *AICCSA*, pp. 1–8, 2015.

[28] J. Barjis, "Enterprise modeling and simulation within enterprise engineering," *Journal of Enterprise Transformation*, vol. 1, no. 3, pp. 185–207, 2011.

[29] C. Figueira and D. Aveiro, "A new action rule syntax for demo models based automatic workflow process generation (demobaker)," in *Advances in Enterprise Engineering VIII: 4th Enterprise Engineering Working Conference, EEWC 2014, Funchal, Madeira Island, Portugal, May 5-8, 2014. Proceedings 4*, pp. 46–60, Springer, 2014.

[30] S. Mazanek and M. Hanus, "Constructing a bidirectional transformation between bpmn and bpel with a functional logic programming language," *Journal of Visual Languages and Computing*, vol. 22, no. 1, pp. 66–89, 2011.

[31] G. Grossmann, M. Stumptner, and W. Mayer, "Change propagation and conflict resolution for the co-evolution of business processes," *International Journal of Cooperative Information Systems*, vol. 24, no. 1, pp. 1–42, 2015.

[32] A. Caetano, A. Assis, and J. Tribolet, "Using demo to analyse the consistency of business process models," *Advances in Enterprise Information Systems II*, pp. 133–146, 2012.

[33] Štěpán Heller, "Usage of demo methods for bpmn models creation," Master's thesis, Czech Technical University in Prague, Prague, Czech Republic, 2016. Master's thesis.

[34] T. Gray, D. Bork, and M. De Vries, "A new demo modelling tool that facilitates model transformations," in *Enterprise, Business-Process and Information Systems Modeling: 21st International Conference, BPMDS 2020, 25th International Conference, EMMSAD 2020, Held at CAiSE 2020, Grenoble, France, June 8–9, 2020, Proceedings 21*, pp. 359–374, Springer, 2020.

[35] T. Gray and M. De Vries, "Empirical evaluation of a new demo modelling tool that facilitates model transformations," in *Advances in Conceptual Modeling: ER 2020 Workshops CMAI, CMLS, CMOMM4FAIR, CoMoNoS, EmpER, Vienna, Austria, November 3–6, 2020, Proceedings 39*, pp. 189–199, Springer, 2020.

[36] D. Rodrigues, "Methodology for modeling business processes in bpmn inspired by demo," Master's thesis, Instituto Superior Técnico, Universidade de Lisboa, 2017.

[37] S. Guerreiro and P. Sousa, "A framework to semantify bpmn models using demo business transaction pattern," 2020.

[38] M. A. Mulder, R. Mulder, F. Bodnar, M. van Kessel, and J. Gomez Vicente, "The simplified platform, an overview," in *Modellierung 2022 Satellite Events*, pp. 223–234, Bonn: Gesellschaft für Informatik e.V., 2022.

[39] M. A. Mulder, R. Mulder, and F. Bodnar, "Towards a demo description in simplified notation script," in *Enterprise Engineering Working Conference*, pp. 53–70, Springer, 2022.

[40] S. M. Platform, "Demo and bpmn toolbox definitions." `https://gitlab.com/teec2/simplified/notationscripts/-/tree/main`, 2024. Accessed May 2025.

[41] O. M. Group, "Business process model and notation (bpmn), version 2.0." `https://www.omg.org/spec/BPMN/2.0`, 2013. Accessed: 2024-05-26.

[42] S. Hirber, "Synchronization prototype for demo and bpmn." `https://gitlab.com/masterthesis4802607/DEMO_BPMN_Bridge/-/tree/main?ref_type=heads`, 2025. Accessed: 2025-06-18.