A Model-Driven Engineering and Software Product Line Approach to Support Interoperability in Systems of Information Systems.

Boubou Thiam Niang

N° d'ordre NNT : 2024LYO20005

THÈSE de DOCTORAT DE L'UNIVERSITÉ LUMIÈRE LYON 2

École Doctorale : ED 512

Informatique et Mathématiques

Discipline : Informatique

Soutenue publiquement le 28 mars 2024 par :

Boubou Thiam NIANG

A Model-Driven Engineering and Software Product Line Approach to Support Interoperability in Systems of Information Systems.

Devant le jury composé de :

Christophe DANJOU, Professeur, École Polytechnique de Montréal, Président

Christelle URTADO, Professeure, IMT Mines Alès, Rapporteuse

Abdelhak-Djamel SERIAI, Maître de conférences HDR, Université Montpellier, Rapporteur

Yacine OUZROUT, Professeur, Université Lumière Lyon 2, Examinateur

Jessie GALASSO-CARBONNEL, Professeure Adjointe, McGill University, Examinatrice

Giacomo KAHN, Maître de conférences, Université Lumière Lyon 2, Examinateur

Amel BENNACEUR, Directrice de recherche, The Open University, Examinatrice

Jannik LAVAL, Maître de conférences HDR, Université Lumière Lyon 2, Directeur de thèse

Contrat de diffusion

THÈSE de DOCTORAT DE L'UNIVERSITÉ DE LYON

Opérée au sein de :

l'Université Lumière Lyon 2

Ecole Doctorale ED512
Informatique et Mathématiques

Discipline : Informatique

Soutenue publiquement le 28 mars 2024, par :

Boubou Thiam Niang

A Model-Driven Engineering and Software Product Line Approach to Support Interoperability in Evolving Systems of Information Systems

---

Devant le jury composé de :

Mme Chritelle Urtado Professeure, IMT Mines Alès	**Rapporteure**
M. Abdelhak-Djamel Seriai Maître de conférences, HDR, Université de Montpellier	**Rapporteur**
M. Christophe Danjou Professeur, Polytechnique Montréal	**Examinateur**
Mme. Amel Bennaceur Director of Research, Associate Professor in Computing, The Open University	**Examinatrice**
Mme. Jessie Galasso-Carbonnel Professeur adjointe, McGill University	**Examinatrice**
M. Jannik Laval Maître de conférences, HDR, Université Lumière Lyon 2	**Directeur de thèse**
M. Giacomo Kahn Maître de conférences, Université Lumière Lyon 2	**Co-Encadrant de thèse**
M. Yacine ouzrout Professeur, Université Lumière Lyon 2	**Co-Encadrant de thèse**
M. Christope Bortolaso Responsable de la recherche chez Berger-Levrault, Berger-Levrault	**Invité**

ABSTRACT

Modern information systems consist of various components that require seamless communication and coordination. Organizations face difficulties adapting to dynamic changes while engaging with diverse industry partners. The challenge arises from the fact that interoperability mechanisms are often created manually and in an ad hoc manner. These mechanisms must be reusable to avoid time-consuming and error-prone processes in an ever-changing environment. The lack of reusability is because interoperability mechanisms are often integrated into business logic components, which creates a strong coupling between components, making maintenance difficult without affecting the overall operation of the system.

Berger-Levrault, our industrial partner, primarily serves public institutions. The company actively maintains interoperability, especially in the context of frequent reforms in the local public sector. In addition, companies have grown through acquisitions, resulting in a diverse range of legacy applications with variations in language, architecture, norms, and industry standards. The primary goal is to create an adaptable interoperability solution that facilitates seamless communication between the components of the information system and the external environment. Challenges include adapting to changing rules and standards, managing variable data volumes, and integrating connected objects within public institutions.

This thesis examines data exchange flows between constituents and systems, analyzes their characteristics and requirements, and proposes cost-effective approaches for implementing and evolving interoperability mechanisms while minimizing the impact on overall information system operations. The methodology adopted begins with a reified vision of interoperability mechanisms, where exchange mechanisms are extracted from the business logic constituents and considered first-class constituents called interoperability connectors. To achieve this, reverse engineering extracts functionality from existing interoperability mechanisms and reifies it as a tangible constituent, the connector, within the information system. For the analysis, we create a repository comprising projects selected transparently, guaranteeing a minimal number of projects and covering all the Enterprise Integration Patterns from different sources. The proposed metamodel confirms and validates this reification regarding completeness and extensibility. The completeness of the connector metamodel is validated through a well-defined process, while another process guarantees the metamodel's extensibility. The extensible metamodel reveals connectors as common entities, leading to the ConPL approach, a software product line framework adapted to connectors. The PhaDOP tool was utilized to implement this approach, and a proof-of-concept was demonstrated with a specific use case. Performance tests were conducted on the proposed connector representation structure. The ConPL framework is validated through an industrial use case.

Keywords: Interoperability, Model-driven Engineering, Software Product Line, System-of-System, Software generation.

RÉSUMÉ

Les systèmes d'information modernes requièrent une communication et une coordination sans faille entre leurs composants. Les organisations ont du mal à s'adapter aux changements fréquents lors de leurs engagements avec divers partenaires industriels en raison de la création manuelle et au cas par cas des mécanismes d'interopérabilité. Ces mécanismes doivent être les réutilisables pour éviter des processus chronophages et sujets aux erreurs dans un environnement en constante évolution. Leur manque de réutilisation résulte de leur intégration fréquente dans des composants de logique métier, créant un couplage fort et rendant la maintenance complexe sans altérer le fonctionnement global du système.

Berger-Levrault est éditeur de logiciels, fournit principalement les institutions publiques, maintenant activement l'interopérabilité pour s'ajuster aux réformes fréquentes du secteur public local. La croissance par acquisitions d'entreprises entraîne une diversité d'applications héritées, avec des variations dans le langage, l'architecture, et les normes. L'objectif principal est de créer une solution d'interopérabilité adaptable facilitant l'interaction entre les composants du système d'information et les systèmes externes. Les défis incluent l'adaptation aux règles changeantes, la gestion de volumes de données variables, et l'intégration de nouveaux composants dans les institutions publiques.

Cette thèse analyse les flux d'échange de données entre composants et systèmes, propose une approche pour implémenter et faire évoluer les mécanismes d'interopérabilité, minimisant l'impact sur les opérations du système d'information. La méthodologie démarre par une vision réifiée des mécanismes d'interopérabilité, extrayant les mécanismes d'échange des constituants de la logique d'entreprise pour les considérer comme des connecteurs d'interopérabilité de première classe.

Pour ce faire, on procède par rétro-ingénierie pour extraire les fonctionnalités des mécanismes d'interopérabilité existants et les réifie sous la forme d'un composant tangible, le connecteur, au sein du système d'information. L'analyse, s'appuie sur référentiel créé, comprenant des projets sélectionnés de manière transparente, garantissant un nombre minimal de projets et couvrant tous les modèles d'intégration d'entreprise provenant de différentes sources. Le métamodèle proposé confirme et valide cette réification en termes de complétude et d'extensibilité. La complétude du métamodèle de connecteur est validée par un processus bien défini, tandis qu'un autre processus garantit l'extensibilité du métamodèle. Le métamodèle extensible révèle que les connecteurs sont des entités communes, ce qui conduit à l'approche ConPL, un cadre de ligne de produits logiciels adapté aux connecteurs. L'outil PhaDOP a été utilisé pour mettre en œuvre cette approche, et une preuve de concept a été démontrée avec un cas d'utilisation spécifique. Des tests de performance ont été effectués sur la structure de représentation des connecteurs proposée. Le cadre ConPL est validé par un cas d'utilisation industriel.

Mots clés: Interopérabilité, Ingenierie Dirigée par le Modèle, Lignes de Produit Logiciel, Système de système, Génération de logiciel.

Acknowledgements

First and foremost, I express my deep gratitude to my thesis supervisor, Jannik Laval, for giving me the precious opportunity to embark on this fascinating journey. I am equally thankful to my two laboratory co-supervisors, Giacomo Kahn and Yacine Ouzrout, for their constant support and encouragement during moments of doubt.

I would like to thank Christophe Bortolaso and Nawel Amokrane for their supervision, bridging the gap between the company's needs and the laboratory's requirements effortlessly.

I would like to express my gratitude to Abdelhak Djamel Seriai and Christelle Urtado for agreeing to act as rapporteurs for my thesis. I would also like to thank Amel Bennaceur, Jassie Galasso-Carbonnel, and Christophe Danjour for agreeing to act as examiners on the jury.

I thank Lilia Gzara for her dedicated support as an internal member of my thesis committee over the past three years.

My warmest thanks to all the staff at IUT Lumière Lyon 2; teaching alongside you has been a privilege and I'm grateful for the opportunities you've given me. Special thanks to Cyrille Dolce for the teaching administration process, for his support, and for the memorable tickets to the OL matches.

I thank my colleagues at the company, in particular Anas Shatnawi, for the past discussion on software reuse and Nicolas Hlad, for their contributions during the weekly discussion sessions, which helped me significantly to progress towards the conclusion of the thesis. I thank Benoit Verhaeghe, a team member with whom I shared experiences from Vaise to Limonest.

Recognizing that a PhD student's journey can be difficult in isolation, I'm grateful to my fellow PhD students at the Limonest site - Clément, Elodie, Mehdi, and others - for the moments shared. Greetings to the other PhD students on the GL team Quentin, Gabriel, and Ikram and a special mention to our non-DRIT neighbors, Sébastien, Jean Michel, Youssef, Thierry, and Eleric.

My warmest thanks go to my laboratory colleagues, especially Bilgesu and Baddredine, my classmates Randa and Meirem, and all the PhD students at the Limonest site. My thanks also go to Vincent Chautet, the laboratory director, and Guy, our IT manager.

Of course, I want to salute the woman of my life, my home thesis supervisor, Aissata's mom, for her unfailing support. My gratitude goes to my daughter, Honey, for the joy she has brought me over the last few months.

I thank my French family - Barbara, Christelle, Mohamed, Pablo Clara - and I salute the memory of Hubert Jourdain, a remarkable person.

I honor the memory of my grandparents, who instilled in me a passion for teaching. Salutations to my family in Mali—my mother, aunts, and uncles—for their enduring support throughout this journey.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

Humans are social creatures who flourish in families, friends, and colleagues' communities. In the TV show *"The Walking Dead"*, people unite to create communities to survive in the long run. However, building a community requires working together to achieve common goals, which is only possible through various forms of interaction. In the past, people used to send letters to take news from distant relatives, which took days or even weeks to receive a response. Similarly, traveling on horseback was another interaction that took several days, leaving the point of departure a source of worry. While these methods were acceptable in the past, they would be intolerable in urgent situations, such as waiting to hear from a sick family member. These examples illustrate how interaction involves both the means of communication and the ability to express oneself effectively, whether through signs, writing, or speech.

Furthermore, in some cases, effective interaction requires more than just expression. For example, when dealing with people from different cultures and languages professionally, adopting a common language is crucial for mutual understanding. English has become the language of global business, making communication between people from different countries and cultures possible. Without a common language, people would need to learn several languages to interact with individuals from different linguistic backgrounds, which could be time-consuming and impractical. Using interpreters is one solution to make communication possible, but even with a shared language, misunderstandings can still occur due to cultural differences. Different cultures have distinct customs and practices that can affect the interpretation of interaction. For instance, bowing one's head is a sign of respect in some cultures, while it is regarded as disrespectful in others if you're talking to someone older.

Thus, interaction involves both the means of communication and the ability to express oneself effectively, as well as an understanding of cultural differences. These factors can make interaction complex, especially when dealing with individuals from distant or different cultural backgrounds.

In the digital world, individuals are represented as software entities and communities as software systems. This software and systems may be located anywhere in the world, but they need to be able to interact to accomplish complex

tasks. This need for interaction between software components has led to software interoperability, which is essential because few software components work in isolation, and the notion of a system would be meaningless without interoperability. The challenges addressed in this thesis aim to understand the current mechanisms enabling interoperability and explore possible solutions to facilitate the implementation of interoperability solutions in a context where the world is increasingly connected, with constant changes disrupting our established practices.

This introductory chapter presents a context that shows the need for interoperability and clarifies the various terminologies used in the field. We will highlight the difficulties in implementing interoperability solutions within distributed and evolving information systems in a software-industrial context.

1.1 Research context

In today's business environment, companies need to work with partners from different industries, which requires them to adapt to various technical and organizational changes such as mergers, acquisitions, or upgrades in technology. Companies' information systems have become complex and dispersed across servers and clouds worldwide. Even though these components are different, they must work together seamlessly to provide customers with the required solutions.

Besides, companies must evolve from a technical and organizational point of view through mergers, acquisitions of subsidiaries, or technological migrations. Nowadays, information systems are distributed and consist of various independent components spread across servers and clouds worldwide. Despite their design and development independence, frequent changes, heterogeneity, and location, all components must work together to provide solutions to customers. It is also essential to ensure interaction with third-party information systems. Therefore, it is necessary to enable interoperability between information systems and their sub-components.

Interoperability between information systems and their constituent is essential for interaction with third-party systems.

Interoperability is a commonly used term, yet its interpretation can vary depending on the context. Nevertheless, ensuring seamless interaction between different systems remains crucial [SPE17]. Numerous definitions exist in the literature, providing a diverse understanding of this concept.

Broader Definition: Interoperability is the ability of diverse organizational entities to collaborate, encompassing both technical and business aspects [LLA+20]. This inclusive definition spans global technical and human considerations.

To delve into a more Information and Communications Technology (ICT) [SL99] focused perspective, let's explore a definition provided by the European Union Commission:

ICT Focus Definition: interoperability is *"the ability of various organizations to work together towards common goals by sharing information and knowledge through their respective ICT systems.* Organizations referred to in this definition include public administration units, any entity acting on their behalf, or EU institutions or bodies [1].

In contrast to the broader definition, the European Union Commission emphasizes technical considerations, such as data exchange, while acknowledging the significance of organizational and business process levels. Developing a software engineering-centric vision of interoperability based on these diverse viewpoints is crucial. Considering a standards-based definition is essential in the context of software engineering:

Software Engineering focus Definition: Within the domain of software engineering, the standard ISO/IEC TR 15944-14:2020 [2] defines interoperability as *"the capability to communicate, run programs, and transfer data among different functional units with minimal need for users to understand the unique traits of these units"*. Additionally, following a software-centric viewpoint, Wegner's definition in [Weg96] characterizes interoperability as *the ability for software to cooperate despite differences in language, interface, or execution platform"*.

The standard and Wegner definitions emphasize the technical aspect of interoperability while neglecting the human aspect. This allows us to focus on the part of the systems we consider in this thesis.

Regarding systems, it is essential to note that the scope and challenges of interoperability vary among different types of systems. These include domains such as the Internet of Things (IoT) [NAG19], Cyber-Physical Systems (CPS) [GA18], Digital twins (DT) [Pir21], and Information Systems (IS) [MDCB17].

[1] `https://commission.europa.eu/system/files/2022-11/other_staff_working_paper_en_v4.pdf`

[2] `https://www.iso.org/obp/ui/#iso:std:iso-iec:tr:15944:-14:ed-1:v1:en:term:3.28`

This manuscript primarily delves into interoperability within the realm of information systems.

According to De Coursy [DC92], an information system (IS) is *"a structured set of resources, including hardware, software, personnel, data, and procedures, used to collect, process, and distribute information about a specific environment. This definition emphasizes the merging of technical and social elements within an information system and points to the broader interoperability considerations"*.

In addition, modern information systems are increasingly distributed and consist of independent software components located worldwide. An information system is "a set of ICT, such as computers, software, and databases, used to perform specific tasks and to interact with and inform different actors in different organizational or social contexts" [BCK15]. This definition emphasizes the complete autonomy of each component within the IS, such as a System-of-System paradigm.

System of Information Systems (SoIS)[MMHA15], a subset of System-of-Systems[KRU+03], connects multiple information systems to create significant organizational value. In this context, redefining interoperability for information systems emphasizes the ability of systems and their components to exchange data and use functionality in a complementary manner. This ensures the independent functioning of each component even when the system is disassembled, emphasizing the autonomy of each component within an information system.

With this in mind, let's revisit the definition of interoperability for a SoIS:

Proposed Definition of Interoperability: Interoperability represents a quality within information systems and their constituent components to exchange data and use their functionalities complementary. This quality ensures that each IS component operates independently, even if the system is disassembled.

Indeed, despite the diversity and complexity of available interoperability mechanisms, they require regular updates to adapt to the evolution and diversity of contemporary information systems. In this thesis, *interoperability mechanisms* refers to the implemented source code enabling communication and coordination between systems and their constituent elements. When these mechanisms are updated, there is an inherent risk of work interruption within the company [BVT22]. This underscores the interdependence among system constituents, indicating a strong coupling between them.

1.2 Defining fundamental Concepts and positioning

Expanding the definition of interoperability in ICT considerations marks our first step in positioning this manuscript. However, it is crucial to understand the multifaceted aspects of interoperability by considering its different levels and layer [LLA+20].

1.2.1 Interoperability: Levels, Layers, and Vision

Interoperability includes data, service, process, and business [CD06, BEF+07]. Each level is subdivided into technical, syntactic, semantic, and organizational layers [GON19]. These layers delineate seamless data exchange, coordination between functionalities, service sequence for business needs, and inter-organizational interactions.

The technical layer emphasizes interoperability capabilities like data transportation using protocols, while the syntactic layer manages the structure of exchanged information. The semantic layer concerns the meaning of exchanged information, and the organizational layer involves defining business objects and structures.

In addition to level and layer of interoperability, it is possible to enable interoperability from a vision that allows you to frame some terminologies [MMP00]. Communication between systems and their constituents is of interest for data exchange. Coordination is employed for interoperating at the service level through functionalities. Interoperability between two systems at the process level falls under the cooperation vision, while collaboration terminology is employed for the organizational level.

This thesis centers on interoperability at the data and service levels, framing this work within the scope of interoperability. Each of the three layers is considered, viewing interoperability through the lenses of communication and coordination.

1.2.2 Distinguishing Interoperability, Integration, and Alignment

The terms interoperability and integration, less frequently alignment, might be interchangeably used despite differences in their meanings. This section aims to elucidate the significance of each term and outline their practical distinctions.

Table 1.1 summarizes the nuances among interoperability, integration, and alignment across various aspects, including their definitions, focuses, objectives, scopes, dependencies, and examples.

Aspect	Interoperability	Integration	Alignment
Definition	The ability of different systems or devices to connect and communicate with each other	The process of bringing different systems, applications, or components together to function as a unified whole	The process of ensuring that the goals, strategies, and actions of various entities are harmonized to achieve a common objective
Focus	Focuses on the ability of systems to exchange and interpret data	Focuses on merging or combining separate systems into a cohesive unit	Focuses on synchronizing objectives, strategies, and actions among different entities within an organization
Objective	Enables different systems to work together seamlessly.	Aims to create a unified, often seamless, user experience by combining functionalities of multiple systems	Aims to ensure consistency, coherence, and mutual support among various entities' goals and strategies
Scope	Primarily concerned with communication and data exchange between systems	Encompasses a broader range of activities, including data sharing, process coordination, and functionality merging	Encompasses strategic, operational, and cultural aspects to align different entities toward shared goals
Dependency	Doesn't necessarily require a deep integration of systems but emphasizes the ability to communicate and exchange information effectively	Often requires a more profound connection between systems, potentially involving shared databases, APIs, or middleware	Involves creating shared values, objectives, and practices among different entities, potentially influencing decision-making and resource allocation
Example	Ensuring that a mobile app can retrieve data from various backend systems, even if they use different protocols or technologies	Combining the functionalities of an e-commerce platform with a customer relationship management (CRM) system to provide a unified view of customers and transactions	Harmonizing the marketing, sales, and product development departments to work towards a unified customer-centric approach

Table 1.1: highlights the differences between interoperability, integration, and alignment

Interoperability focuses on facilitating communication and data exchange between systems, integration involves merging systems into a cohesive unit, and alignment concentrates on ensuring that different entities within an organization work harmoniously toward shared goals.

1.3 Research Motivations and Objectives

This thesis project was conducted in collaboration with Berger-Levrault, a company specializing in software solutions primarily for the public sector. The project addresses the pressing need for efficient information sharing and exchange within ongoing reforms in the local public sector. These reforms include restructuring local authorities, the implementation of legislation such as the NOTRE law, the establishment of Territorial Hospital Groups in 2016, and the Digital Republic initiative. To address these changes, an adaptable interoperability solution must be developed to facilitate streamlined data exchanges between software applications and the external environment.

Throughout these transformations, Berger-Levrault's applications need to adjust to evolving rules and standards while advancing the integration of public service digitization. These changes have a substantial impact on established data exchanges. Consequently, there is a need for flexible exchange architectures capable of accommodating diverse data volumes and types, particularly during peak periods. For instance, millions of payslips are edited and exchanged between various system constituents at the end of each month.

Through collaborative research with Berger-Levrault and the DISP laboratory, critical challenges were identified:

These transformations involve a substantial volume of exchanged data, subject to variations, especially during 'peak periods' like electronic voting during elections. The nature of these exchanges may also be affected, especially in the realm of connected objects. Public institutions increasingly utilize such objects for managing city equipment or user services. The increased volume of exchanged data necessitates exchange architectures capable of supporting this load and the variability in data production types and frequencies. This requires distributed, adaptable, or even self-adaptable architectures to ensure system fault tolerance while preventing potential congestion.

Research conducted in partnership with Berger-Levrault and the DISP laboratory has revealed certain hurdles:

Lack of visibility into existing interoperability exchanges: Most current exchanges lack traceability, and monitoring mechanisms mainly focus on low-level technical information without correlation to business information. Few methods focus on evaluating effective data interoperability post-implementation, and fewer still are equipped with suitable tools.

Complexity in maintaining exchanges: The lack of traceability and evolving exchange configurations present challenges in identifying faults, hindering effective alert systems' implementation. Additionally, the lack to capitalize on information complicates maintenance.

Manual development of exchange system modules: This approach is costly and does not meet the reactivity requirements of business areas subject to frequent change, requiring the development of adaptable exchange systems using dynamic interoperability pivots. This leads to considerable delays, for example, when migrating existing applications.

This thesis aims to develop a comprehensive approach to the lifecycle of interoperability mechanisms, improving the reliability and resilience of exchange systems to ensure interoperability. The thesis statement is that:

Any information system can be transformed into a System of Information Systems by maximizing the separation between business logic computation and communication and coordination. By doing so, we can automatically generate connectors between business logic constituents, making them loosely coupled and reifying communication-specific instructions regardless of their characteristics. Subsequent changes to communication specifications will only affect the connectors, leaving the constituents responsible for the core business logic unaffected.

The proposed solution intends to streamline all application exchanges and services, optimizing software and infrastructure resources within public institutions.

1.4 Structure of the manuscript

The manuscript is structured in the following manner:

- Chapter 2 provides an in-depth examination of general interoperability concepts and proposed solutions for implementing interoperability mechanisms.
- Chapter 3 introduces an extensible metamodel that depicts reified interoperability mechanisms derived from existing systems. It provides a comprehensive view of the connector, presenting its constituents at a high level for better understanding by both technical and non-technical stakeholders.
- Chapter 4 explains the methodology used to evaluate the completeness and extensibility of the connector's metamodel through a heuristic process. Additionally, experiments were conducted to assess the performance of the reified messaging connector and compare it with interoperability mechanisms based on the messaging style.
- Chapter 5 presents the *ConPL* framework, a software product line approach designed to implement interoperability connectors. The framework utilizes Model-Driven Engineering and Delta-Oriented Programming in this context.
- Chapter 6 proposes the *PhaDOP* framework and demonstrates its practical application in implementing Software Product Lines. Emphasizes the utilization of Delta-Oriented Programming at the model level to execute the *ConPL* framework through a fundamental use case.

- Chapter 7 involves the implementation of an industrial use case following the *ConPL* framework and utilizing the PhaDOP framework.
- Chapter 8 provides an overarching perspective on the research conducted, offers insights into future research directions on evolving interoperability mechanisms in System-of-Information Systems, and concludes the manuscript.

Chapter 2. State of the Art

2.1 Introduction

Chapter 1 establishes the core concepts of the thesis, defines fundamental terminologies, and contextualizes the present work within the realm of interoperability concerning System of Information Systems (SoIS), specifically focusing on the ICT dimension of Information Systems. Our primary objective revolves around addressing interoperability in line with the System-of-System requirements. This pursuit leads us to elevate interoperability mechanisms into primary system entities called connectors. However, the literature often utilizes various terms to describe these connectors, disregarding SoS considerations. The initial segment of this chapter surveys to analyze frequently used terms for interoperability mechanisms, emphasizing their distinctions. Subsequently, we delve into current industrial practices in the second section. The third part explores prevalent standards adopted in pursuing interoperability, while the final section scrutinizes diverse approaches employed for implementing efficient interoperability solutions.

2.2 Exploration of Key terminology in Describing Interoperability Mechanisms

Our exploration of the interoperability literature has revealed various terms used to describe interoperability mechanisms. These terms often attempt to characterize both the type of interoperability challenge and the role played by the mechanism implemented. However, there is considerable variation; different terms may refer to identical concepts. The work of [MMP00] introduced a taxonomy of software connectors derived from analyzing interactions between existing components. This taxonomy classifies software connectors according to service and interaction types. This section endeavors to clearly define the most commonly used terminologies and to emphasize their differences. This helps position the concept of connector used in this manuscript by clarifying why we use this term and not others.

2.2.1 Middleware

The main difference is that using the middleware terminology more generally goes beyond application layer consideration by considering the infrastructure layer [SMR+12]. While the application level is limited to the application level, The infrastructure layer includes another non-functional level covering (security, logging service, and transaction), runtime level management level (lifecycle or binding), and kernel level.

Middleware [CBB+00] is a technology that acts as a bridge between service providers and requesters. It provides standardized mechanisms for communication, data exchange, and type marshaling. It uses higher abstraction than messaging, making it easier to build interoperable applications.

The term *middleware* has been used in software engineering since the late 1960s, and it can refer to a wide range of modern software components. Mediators, connectors, and APIs are examples of interoperability mechanisms that can be categorized as middleware, focusing on application layer interoperability.

2.2.2 Mediator

The concept of a mediator was first introduced in the paper by [Wie92] to solve the problem of integrating heterogeneous data sources in distributed information systems. This paper defines the mediator as a software module utilizing specific data sets or subsets of knowledge to generate information for higher-level applications. Initially, the definition was framed in a vertical architecture vision with several layers, but this changed over time.

Three years later, the concept of mediator emerged as a design pattern in the book on best-known design patterns by [Bec95], which was classified as a behavioral pattern. This perspective as a behavioral pattern was reinforced in another paper by [SI10], who used the term *mediating connector* or *mediator* and defined it as one or a collection of components responsible for overseeing and reconciling behavioral mismatches. The mediator forwards interaction messages between components, facilitating protocol translation or adaptation when necessary.

The definition of [SI10] underscores that a mediator is a component, and combinations of mediators can be formed. It encompasses the information exchanged, spanning data and service-level interoperability, and addresses technical layers through protocol conversion. s.

Furthermore, the Ph.D. thesis by [Ben13] delineates the mediator as intermediary software that facilitates the collaboration of functionally compatible

components without requiring modifications. This underscores the role of the mediator in addressing various concerns, including coordinating component behaviors to prevent issues like deadlocks, translating data for meaningful exchanges between components, and managing communication among distributed components to navigate network-related challenges like concurrency and fault tolerance.

While Bouloukakis in [BGNI19] refers to the term "mediator adapter", retaining its definition and role, Gio in [Gio12] aligns himself with the work above regarding the meaning of mediators, but emphasizes semantic matching. However, for this discussion, we will ignore this reduction in the role of mediators.

A mediator can facilitate behavior matching, data exchange, and protocol conversion.

2.2.3 Adaptor

According to [YS94], an adapter is code that acts as an intermediary between two components, designed to mitigate the disparities between their interfaces. The software adapter can reconcile the gap between an application and functionally compatible but incompatible interfaces. For interaction between two components, their interfaces must be exposed, usually defined by method signatures. Then, a limited set of rules specifies the correspondence between these interfaces.

As for the mediator, [Bec95] presents an adaptor as a design pattern, precisely a structural architectural pattern. Its role is to convert an interface of a class into another interface that clients expect. So, the adaptor permits classes to work together even if their interfaces are incompatible.

One difference between a mediator and an adaptor is that the adaptor is a supplementary thing added to a component to glue it together, not an independent entity.

However, [Bec95] affirms that an adaptor is a wrapper. However, the current literature shows some differences between the two terminologies.

2.2.4 Wrapper

A wrapper, or an interface wrapper, encapsulates or contains an object or functionality, providing a simplified or modified interface to interact with that object. It acts as a protective layer around an object, allowing controlled access or modification to its functionalities. According to [CBB+00], a wrapper consists of two parts: an adapter that provides additional functionality for an application program at essential external interfaces and an encapsulation mechanism that

binds the adapter to the application and protects the combined components. The software adapter is part of the wrapper. It intercepts all invocations to provide additional functionalities such as synchronization between the local and distributed object, transaction control, events monitoring, and exception handling.

Wrapping is an approach to protecting legacy software systems and commercial off-the-shelf software products that require no modification of those products. In summary, wrappers and software adapters share similarities but serve different purposes within a system.

Like a wrapper, a software adapter mediates interactions between components or systems by compensating for differences in their interfaces. However, an adapter specifically focuses on facilitating interoperability between different systems or components with distinct interfaces. It acts as a bridge, enabling communication and interaction between disparate elements by translating or transforming data and method calls to ensure compatibility and seamless operation.

While wrappers and software adapters involve encapsulation and interface modification, a wrapper primarily focuses on providing a modified interface for an object. In contrast, a software adapter enables interoperability between diverse systems or components.

Concerning the difference between a mediator and an adaptor, [Men07] states that the mediator is an independent component that manages interactions and communication between components. At the same time, a wrapper concentrates on encapsulating and providing controlled access to an object's functionalities.

2.2.5 Application Programming Interface (API)

API stands for Application Programming Interface. APIs are software tools that facilitate the communication between two computer applications. APIs can connect entirely different products and services by providing a standard layer. It refers to a specific set of rules and guidelines that a software program can follow to access and utilize the resources and services that another software program implements that API [Sch05]. According to [BC19], APIs are a set of protocols that determine how software components communicate.

It's important to note that APIs differ from adapters. While an API defines how systems can interact, an adapter helps to connect systems or components that wouldn't naturally work together due to interface differences.

The wrapper concept differs from the API. A wrapper wraps the original API with an additional layer, simplifying its usage.

Mediators, on the other hand, complement APIs in designing modular, maintainable, and scalable systems. While an API focuses on defining the interface and communication protocols, a mediator focuses on managing and facilitating communication and interaction between different components or objects within a system.

2.2.6 Software connector

Following the literature, [MMP00] states that connectors are to be explicit and tangible elements of the system. In other words, it is a fundamental element at the architectural and implementation level of the system. [YS94] discuss connectors as first-class system entities in the related work. Their connectors are first-class, reusable components in their own right and can support n-party interactions. Connectors are polymorphic in that any component's port whose protocol is compatible with a given connector's role can be plugged into that role.

The mediator focuses on the object, while the connector focuses on the system. Knowing the SoS context component is also a system.

According to the literature, [MMP00] emphasizes that connectors should be explicit and tangible elements of a system. This means that connectors should be visible and present at the architectural and implementation levels of the system. Yellin [YS94] also discusses connectors as first-class entities in the related work. Their connectors are reusable components that can support multiple interactions. Connectors are polymorphic, meaning that any component's port whose protocol is compatible with a given connector's role can be plugged into that role.

It is important to note that the mediator focuses on the object, while the connector focuses on the system. In a System of Systems (SoS) context, components should also be considered part of the more extensive system.

2.2.7 Summary of terminologies

We have made two summary tables to recapitulate the findings of our survey on diverse terminologies. The first table, Table 2.1, focuses on the crucial aspects of using each terminology. The second table, called Table 2.2, highlights the similarities and differences between the terminologies used. This allows us to draw a more comprehensive conclusion for each concept.

In this comparative analysis of commonly used terminologies in software development, integration, and interoperability, we classify them based on their categories, subcategories, and primary roles. Upon comparing them based on primary roles, we observe that middleware provides general communication ser-

Terminology	Category	Subcategory	Role
Middleware	Integration	Integration Support	Provides services for communication, data management, messaging, security, and transactions between different software systems.
Connector	Interoperability	Interoperability	Establish connections between disparate systems or components, managing data transformation, protocol translation, or interface adaptation for interaction.
Mediator	Design Pattern	Behavioral Design Patterns	encapsulate interaction logic between objects, promote loose coupling, and manage interactions through a mediator object.
Adapter	Design Pattern	Structural Design pattern (Interface Compatibility)	Facilitates interaction between entities with incompatible interfaces by converting one interface into another expected by clients.
Wrapper	Extension	Functionality Extension	Encapsulates and delegates existing class/-component functionalities, allowing modification or extension of behavior without altering its original structure.
API	Interface	Interface Specification	Defines protocols, tools, and routines for software interaction, serving as an intermediary for communication between different software components.

Table 2.1: Overview of different concepts used in interoperability and integration

vices, which include security and transactions. On the other hand, connectors or software mediators facilitate interaction between different systems and components, enabling data transformation, protocol translation, or interface adaptation. The software connector solves the problem of interface adaptation that a mediator, adaptor, or wrapper addresses. Mediator and adaptor are design patterns where the mediator encapsulates the interaction between objects, and the adaptor proposes interface adaptation if required for interaction with other interfaces. Wrapper adds a layer of abstraction instead of offering several interfaces, as in the adapter case. Lastly, APIs serve as an intermediary for communication between service or microservice software components, not between objects like the mediator or adaptor.

While software development and integration use distinct concepts, some functionality overlaps, connectors, and APIs might be included in middleware services. Still, each concept plays a unique role in system design, integration, and interaction.

2.3 Industrial practice/architecture for interoperability

Now that we have summarized and clarified the different terminologies for designing interoperability mechanisms, we propose surveying different industry architectures.

Terminology	Purpose/Role	Commonality	Difference
Middleware	Facilitates communication between systems	Provides services for software integration	Differs in the range of services offered, including APIs, connectors, and more for system communication
Connector	Establishes connections between systems	Enables interoperability	Differs from other concepts by focusing specifically on connecting disparate systems
Mediator	Manages interaction between objects/components	Promotes loose coupling and manages interactions	Differs by specifically managing interaction and communication among objects
Adapter	Facilitates interaction between incompatible interfaces	Allows entities with different interfaces to work together	Differs by converting interfaces to enable interaction between incompatible entities
Wrapper	Encapsulates and extends functionalities	Adds new functionalities without altering the original structure	Differs by encapsulating and delegating functionalities to modify behavior
API	Defines protocols for software interaction	acts as an intermediary for different software components to communicate	Differs by specifying rules for software interaction and system communication

Table 2.2: Difference analysis commonly used concepts in interoperability and integration

2.3.1 Point-to-Point architecture

Point-to-Point architecture is commonly used to achieve ad-hoc interoperability between software components [ABG+19]. This architecture is accidental [Boo06], meaning it is not a planned strategy but the result of combining several ad hoc interaction flows. It is often used when there is no clear strategy because it is relatively easy to set up, especially for small-scale systems. However, it is a risky choice because the scale of a system can rarely be predicted for years to come.

In Point-to-Point interaction, systems and components interact directly with each other. While setting up a small-scale system is easy, this approach has disadvantages. For instance, if a component needs to interact with multiple applications, it must ensure that all integration logic is implemented for each of them. This includes protocol conversions, data transformation, transfer, and more.

As time passes, multiple individual interfaces must be changed simultaneously, leading to an ongoing maintenance burden for keeping the interfaces up-to-date. The result is a highly complex system that can become difficult to manage, leading to performance issues and management problems [Sch05]. Ad-

ditionally, it becomes challenging to impose standards, and adding a new link is always problematic, requiring new developments and adapted infrastructures.

2.3.2 Hub-Spoke architecture

The Hub-and-Spoke architecture is a popular choice for software practitioners who want to avoid the complexity that can arise with point-to-point architectures [Sch05]. This architecture allows multiple systems to be integrated using a central hub to facilitate communication. The hub provides a uniform interface to all participating systems, making it easier to maintain due to fewer dependencies.

In a Hub-and-Spoke architecture, the central hub is an intermediary between the different systems or "spokes". The applications communicate through the hub, using interfaces based on the Hub/Spoke architecture. The central hub is responsible for routing messages to the various systems and applications, which includes tasks like translation, transformation, and message redirection.

Compared to point-to-point interaction, the number of connections is significantly smaller for hub-and-spoke, making it easier to manage and [Ris07]. The spokes in the architecture are also decoupled from each other, which strengthens their connection with the hub instead of relying on multiple communication channels with other applications.

While the Hub-and-Spoke architecture solves the problems of point-to-point communication and offers advantages such as decoupling, there are also some disadvantages. One disadvantage is that there is a single point of failure - if the central system fails, all communication stops. Additionally, since all communication passes through the central hub, it can lead to overloading and bottlenecks.

2.3.3 Message-Oriented Middleware (MOM)

Message-Oriented Middleware (MOM) [YQC+19] is a middleware architecture that promotes system communication by enabling asynchronous message passing. Its main objective is to facilitate the reliable exchange of messages between applications or components. MOM enables systems to communicate without requiring both parties to be available simultaneously, thus ensuring reliable message delivery through features such as queuing or guaranteed delivery.

The main difference between MOM and Hub-and-Spoke architecture is that MOM primarily focuses on reliable asynchronous messaging. In contrast,

Hub-and-Spoke architecture emphasizes using a centralized hub for managing and orchestrating communication flow among multiple systems.

2.3.4 Service-Oriented Architecture (SOA)

Service-Oriented Architecture (SOA) is an architectural approach for building and integrating enterprise applications [LL09]. It utilizes reusable and interoperable services, known as software interfaces, for distributed computing. This enables remote system interaction and seamless data exchange, facilitating loose coupling among services that communicate and cooperate across diverse technology or platform foundations.

SOA provides advantages such as flexibility, reusability, and scalability. However, studies have identified challenges [PED19], particularly with interoperability when integrating legacy systems or constituents not initially aligned with SOA principles. It relies on standards, making it more suitable for systems or constituents adhering to them.

Another challenge is managing the proliferation of services, interactions, and dependencies as the system expands. This growth can foster a point-to-point integration style, resulting in a convoluted, spaghetti-like architecture.

Nevertheless, SOA offers superior reusability, scalability, and governance compared to point-to-point integrations, especially in intricate environments requiring multifaceted system communication. Its structured and systematic approach contrasts with the ad-hoc nature of point-to-point integrations, often leading to maintenance challenges as the system evolves and expands.

2.3.5 Enterprise Service Bus (ESB)

Enterprise Service Bus (ESB) represents a middleware architecture designed to enable interoperability among diverse applications or services within an enterprise [Cha04]. As a central hub, ESB facilitates communication between multiple applications or services.

It is essential to understand that ESB is often seen as a tool supporting the implementation of Service-Oriented Architecture (SOA) principles rather than serving as a direct alternative to SOA [Men07]. ESB indirectly connects applications through its centralized structure rather than establishing direct connections. This central hub incorporates essential logic to facilitate interaction and integration among systems. It empowers functionalities like routing, invocation, and mediation, ensuring secure and reliable interaction among disparate distributed applications and services. Mediation, a core aspect of ESB, encompasses trans-

formations or translations across various resources, including transport protocols, message formats, and content.

In contrast, Message-Oriented Middleware (MOM) provides reliable asynchronous messaging between systems, focusing primarily on message-based communication and dependable message delivery.

Choosing between ESB and MOM hinges on specific integration needs and the complexity of an organization's architecture. ESB offers a comprehensive integration platform with broader capabilities than MOM. In some cases, these technologies can complement each other within an organization's integration strategy, each serving distinct purposes based on the requirements of the enterprise architecture.

2.3.6 Summarize of the architectural style

After examining various industry architectures for interoperability, it becomes evident that each architecture offers a unique approach with advantages and challenges.

Point-to-point architecture [ABG+19], despite being easy to set up, poses risks and complexities as systems scale over time. Its ad-hoc nature leads to increased maintenance burdens and challenges in enforcing standards.

Hub-and-spoke architecture [Sch05] addresses some of these issues by centralizing communication, reducing complexity, and minimizing the number of connections. However, it introduces single points of failure and potential bottlenecks.

Message-oriented middleware (MOM) [YQC+19] focuses on reliable asynchronous messaging, ensuring dependable message delivery between applications without requiring simultaneous availability.

Service-oriented architecture (SOA) [LL09] emphasizes reusable and interoperable services for distributed computing, offering scalability and reusability. However, it faces challenges with legacy integration and managing proliferating services and dependencies.

Enterprise Service Bus (ESB) [Cha04] is a comprehensive integration platform that indirectly connects applications through a centralized hub. It facilitates system interaction and integration while providing functionalities like routing, mediation, and secure interaction among distributed applications.

Choosing the appropriate architecture requires carefully evaluating specific integration needs, scalability, reliability requirements, and the complexity of the enterprise's architecture. Each architecture presents a trade-off between

Architecture	Key Features	Advantages	Challenges
Point-to-Point	Ad-hoc, direct interaction, multiple individual interfaces	Easy setup for small-scale systems	Increased maintenance, scalability challenges, complexity
Hub-and-Spoke	Central hub, uniform interface, fewer connections	Reduced complexity, easier maintenance	Single point of failure, potential bottlenecks
Message-Oriented Middleware	Asynchronous messaging, reliable message passing	Dependable message delivery, no simultaneous availability	Focused primarily on message-based communication
Service-Oriented Architecture	Reusable, interoperable services, loose coupling	Flexibility, reusability, scalability	require standard adoption, legacy integration issues, managing proliferation of services
Enterprise Service Bus	Centralized hub, mediation, comprehensive integration	Facilitates interaction and integration among systems	Indirectly connects applications, requires careful evaluation

Table 2.3: Difference analysis commonly used concepts in interoperability and integration

advantages and challenges, emphasizing the importance of aligning these factors with the enterprise's objectives to achieve optimal interoperability and scalability while managing inherent complexities.

Table 2.3 highlights the distinctive features, advantages, and challenges associated with each architecture, providing a quick comparative view for better understanding and evaluation based on specific needs and priorities within an enterprise.

2.4 Survey of Standard for interoperability

Various standards help interoperability between different devices and systems in different industries. This section highlights some of the most notable ones.

Hyper Text Transfer Protocol (HTTP) [FGM+97], and its secure version HyperText Transfer Protocol Secure (HTTPS) [Dan09] are widely used protocols for web communication and data transfer over the internet.

Message Queuing Telemetry Transport (MQTT) [SM17] is primarily used in IoT (Internet of Things) applications because of its efficiency and lightweight messaging system for machine-to-machine communication. MQTT for Wireless Sensor Networks (MQTT-S) [HTSC08] is a wireless network sensor extension

designed to be run on low-end, battery-operated sensor/actuator devices and operate over bandwidth-constrained WSNs such as ZigBee-based networks. MQTT for sensor network (MQTT-SN) [SCT13] is another extension designed to be as close as possible to MQTT but adapted to the peculiarities of a wireless communication environment like low bandwidth, high link failures, short message length, etc. It is also optimized for implementing low-cost, battery-operated devices with limited processing and storage resources.

Advanced Message Queuing Protocol (AMQP) [Pra21] is designed for message-oriented middleware and is helpful in scenarios where reliability and interoperability are crucial in communication between applications. AMQP is an open standard for enterprise messaging designed to support messaging for almost any distributed and business application.

Constrained Application Protocol (CoAP) [BCS12] is a protocol specifically tailored for constrained devices and is designed for low-power and low-bandwidth networks. CoAP provides a RESTful protocol for communication, enabling interoperability in IoT environments.

Open Platform Communications Unified Architecture (OPC UA) [PTD+19] is a machine-to-machine communication protocol that is widely used in industrial automation. It is defined in the IEC specification 62541. In OPC-UA, every node in the server's address space is described. This information can be queried and used by a client along with the received data.

Data Distribution Service (DDS) [SPCF04] is a middleware that enables highly dynamic distributed systems to publish and subscribe to data in a data-centric way [PTD+19]. It is standardized by the Object Management Group (OMG) [1]. Compared to OPC-UA, DDS is more focused on data. In DDS, data is published into the DDS domain, and subscribers can subscribe to data from that domain without knowing where the information came from or how it is structured, as the information package already describes itself.

Digital Imaging and Communications in Medicine (DICOM) [MDG08] is a standard designed for healthcare to ensure interoperability between medical imaging devices and systems. DICOM facilitates the exchange of medical images and related information.

Health Level Seven International (HL7) [DAB+01] is a standard in healthcare that aims to facilitate the sharing, integration, exchange, and retrieval of electronic health information between different healthcare systems. There are various versions of HL7, including v1, v2, and v3. The most recent version is HL7-FHIR [Sar19], where FHIR stands for Fast Healthcare Interoperability Re-

[1] `https://www.dds-foundation.org/`

source. HL7-FHIR is a newer standard developed by HL7 to address some of the limitations and complexities of previous versions. FHIR is designed to be lighter, more flexible, and more developer-friendly than earlier HL7 standards. It leverages modern web standards like RESTful APIs (Representational State Transfer) [Pat17] and uses a resource-based model, making it easier for different systems to understand and exchange healthcare information.

ISO 20022 [MM20] is a global standard for financial messaging. It standardizes the format and structure of financial messages, enhancing interoperability and communication within the financial industry.

JavaScript Object Notation (JSON) [PRS+16] is a lightweight data-interchange format that transmits data between a server and a web application. Its simplicity and readability enhance interoperability.

Specific standards exist in various industries and domains to ensure compatibility, seamless communication, and interoperability between systems, applications, and devices. Other standards may also be necessary to facilitate efficient data exchange and communication depending on the context and requirements.

This chapter provides an overview of some of the used interoperability standards. The primary aim of this section is to demonstrate that these standards are constantly evolving. For instance, the shift from HTTP to HTTPS involves the addition of SSL or TLS [SL+16]. MQTT has several variations that can be used for energy conservation, while HL7-FHIR is used to integrate Restful APIs. These updates highlight the challenges involved in maintaining interoperability for legacy systems. Nonetheless, certain standards like OPC UA for machine-to-machine communication in the industrial sector and HL7 for healthcare are specific to particular domains. As a result, it is crucial to combine standards since domain stakeholders may need to interact with counterparts from other domains. While standards are helpful, they do not solve all interoperability issues.

2.5 Interoperability mechanisms implementation approach

This section will review the approaches proposed in the scientific literature for enabling efficient interoperability solutions. These approaches primarily focus on establishing effective interoperability between systems and their constituents. The proposed approaches include (1) first-class connectors, (2) reconfiguration capabilities - including runtime considerations, (3) automatic synthesis, (4) model-driven adaptation, and (5) variability management.

2.5.1 Connector as first-class entity

The concept of an exogenous connector emphasizes the separation of concerns between computational and communication-oriented code, leading to the system-of-systems concept. Communication-oriented codes are code-specific for business logic, and communication-oriented codes are codes specific to interaction between components. Achieving independence among constituents necessitates the externalization of all dependencies between them. In this context, dependencies refer to source code within a constituent that relies on another to function. By eliminating these dependencies, strong coupling between constituents is avoided. The primary approach going in this line is mentioned in this section.

ArchJava, as presented in [ACN02], stands as an early proposal that delineates the distinction between components and connections. ArchJava introduces the concepts of components, connections, and ports as a language extending the object-oriented programming paradigm's class notion. Its fundamental emphasis lies in ensuring communication integrity. This is established by restricting a component's direct invocation of methods from other components, allowing such access solely along designated connections between ports. This stringent control ensures that a component can only initiate calls to authorized counterparts. However, ArchJava confronts limitations primarily centered around system evolution challenges, mainly when dealing with smaller-scale applications. Its applicability is confined to programs written in a single language and running on a Java Virtual Machine (JVM).

The notion of an exogenous connector in [LEW05, AL17] promoted the separation of entities that play a purely functional role from components that implement communications. This decoupling allows flexibility and scalability since acting on the connector without impacting the business logic is possible.

Lau et al. in [LEW05] limitation. Implemented in Java. Using hierarchy pair to pair connector

[AL17] (difference with X-MAN) approach focuses on SOA systems. For this reason, unlike X-MAN, our approach is distributed (i.e., multi-process), so services are mapped onto different network addresses, and the control flow is distributed over a network (copy/Past)

In [BVT22] Towards an intelligent connector for dynamic interoperability in an agile enterprise. For now, conceptual framework. Implementation is ongoing. Target industries 4.0

2.5.2 Dynamic or runtime reconfiguration

The second aspect is about implementing connectors dynamically by reconfiguring component orchestration. Several studies, such as those by [SMR+12, IRHBJ16, RBVL18, RBVL18], present approaches that simplify connector development and address design-time or runtime reconfiguration to change interactions between entities. These approaches tackle interoperability solutions by focusing on two fundamental combination paradigms: choreography and orchestration [Pel03]. The approaches presented in this study offer features such as reflexive component support, which allows configuration changes of the assembly at runtime using a script. These approaches can make the system dynamic through static configuration and even at runtime. This will enable developers to focus solely on the business logic, mixing services based on different technologies.

Seinturier *et al.* [SMR+12] propose FraSCAti, an open-source platform that aims to simplify the development of service-oriented architecture (SOA) [LL09] based on the Service Component Architecture (SCA) [CZ13] standard. With FraSCAti, developers can focus on the business logic, mixing services based on different technologies such as Bundle, Java, script, and BPEl. The platform offers features like reflexive component support, which allows configuration changes of the assembly at runtime using a script. To expose services on the web via the SCA standard, developers need to specify them in an XML file. However, FraSCAti is limited to Java technologies, and it does not support all the languages and protocols specified for SCA. This means that the possibility of interactions is limited. FraSCAti is specific to service orchestration, where the connector is centralized, and it cannot implement a connector based on event-driven communication. Moreover, connectors are not considered to be first-class entities of the system, and generation is not addressed.

Ibañez *et al.* is also interested in the dynamic reconfiguration of interactions between services based on service orchestration in [IRHBJ16]. The authors propose reconfiguring applications using the GCMScript language [BHR15]. The Grid Component Model (GCM) [BCD+09] exposes control interfaces that allow changes of components at runtime and structure of services. This allows adding or removing components or links at runtime and makes dynamic reconfiguration possible. However, as explained by FraSCAti, this approach focuses on the orchestration of services where the notion of connector is implicit. Furthermore, it is essential to know the cost of adding a new component: the number of possible new connectors it generates or the impact of the number of combinations to be tested.

Ciatto *et al.* [CMO+18] proposes the ReSpecTX language, toolchain, and standard library for programming the coordination of multi-agent systems and distributed applications in general. The ReSpecTX standard library is designed as a constantly evolving collection of interaction mechanisms providing a reference library of reusable and composable interaction patterns.

The ReSpecTX [DNO98], is based on the ReSpecT language inheriting and extending its semantics while pushing it beyond the limits of other coordination languages through features such as modularity, composability, and tools. However, the shortcoming of ReSpecTX is that the set of ready-to-use composable coordination mechanisms provided by its standard library needs to be constantly extended to cope with an increasing number of application scenarios and their typical interaction patterns.

2.5.3 Automatic synthesis

There are a lot of papers that are interested in the automatic synthesis of connectors. Bulej *et al.* [BB03] was one of the first works to deal with the generation of connectors. In this paper, the authors propose the basic ideas for connector generation and specify the data structures and interfaces necessary for generation in a platform-independent language CORBA interface description language (CORBA IDL) [CFP+01]. The proposed model follows the top-down approach of software design allows for defining different types of connections. Although this approach was pioneering in this field, it does not go as far as the generation of connectors.

Inverardi *et al.* [IT13] propose a method for automatically synthesizing modular connectors, a composition of independent mediators. The mediators are primitive sub-connectors that perform a mediation pattern corresponding to the solution of a recurrent interoperability mismatch. However, the approach assumes that a network system (NS) is accompanied by an AI-based specification of its interaction mechanisms. This may be considered for internal interoperability, but it isn't easy to require from a customer. The interaction mechanisms of an NS express the order in which input and output actions are executed when the NS interacts with the environment. This approach allows for greater modularity but is only design time according to [BN17]. The number of possible primitive sub-connectors limits the modularity. In addition, this approach is specific to peer-to-peer communication. Namely, the interactions between peers constitute the network system called the choreography. For example, an event-driven connector cannot be created.

Autili *et al.* [AIT18] propose a formal approach to the application of choreography realizability through the automatic synthesis of distributed Coordination Delegate (CD). The CDs are additional software entities for the participants in the choreography. The process takes as input a choreography specification in the form of a state machine and automatically generates a set of Coordination Delegates (CDs). Furthermore, another limitation of this approach is that it is choreography-specific, as connectors develop peer-to-peer connections. We seek more flexibility, such as generating a connector for event-driven communication.

Bencomo *et al.* [BBG+13] focuses on automatically generating connectors on the fly. This approach combines machine learning techniques and ontologies to discover the component's functionalities with which to interact and automatically create a mediation software component.

Bennaceur *et al.* [BI14] the authors present an approach based on ontological reasoning and constraint programming to infer mappings between component interfaces automatically. This approach requires the exposure of the signatures of the different features. Then, machine learning techniques are combined with the domain ontologies of the two services to explore the behaviors of the two functions to generate the computed mediator. The weakness of this approach is that it requires domain ontologies that are not easy to obtain from clients for an information system composed of several applications from different application domains. In addition, the discovery algorithm can be time-consuming and resource-intensive for an interface containing many function signatures.

2.5.4 Model-driven approaches

The fourth aspect concerns approaches that aim to make connectors more modular and promote reuse and scalability. In [ADSG+18b], the authors define an approach to the automatic synthesis of service adapters using Enterprise Integration Patterns (EIPs) [HW04]. The approach proposes a Service-Role Adapter, a metamodel used to represent the component matching. Moreover, the Adapter Component Metamodel [ADSG+18a] represents the structure of an adapter as a chain of adapter components implementing the considered EIPs. Each of these is realized as an appropriate composition of adaptation primitives. The authors of [BGNI19] introduce a solution for the automated synthesis of mediators that ensure the interoperability of heterogeneous things, the Data exchange (DeX) connector model. The objective is to devise a generic connector that comprehensively abstracts and represents the semantics of the various middleware protocols. Works in [AIS+19] define a model-based framework for mediators and an auto-

mated approach to mediator synthesis. The objective is to devise a generic connector that comprehensively abstracts and represents the semantics of the various middleware protocols. The interaction abstraction step consists of taking as input an interaction process, a choreography specification, and its ontology to produce a coordination delegate.

2.5.5 Exploiting variability and code generation

The fourth group of work has focused on reusability and variability management to facilitate the implementation of connectors.

Jongmans *et al.* [JSS+12] propose Reo, a graphical and exogenous coordination language for compositional construction of Web services using constraint automata. The framework takes the behavioral description of the services as input in the form of constraint automata, the WSDL interfaces, and the description of their interaction in Reo. It generates all the Java code needed to orchestrate the services in practice. A proxy is automatically generated for each web service to manage the communication between this service and the Reo circuit. This is interesting for direct communications between web services but is not variable to allow the implementation of other types of connectors, such as file transfer, shared database, or event-driven. Furthermore, it lacks variability and abstraction from the target language since the connector cannot be generated in Java.

Reo is improved in Jongmans *et al.* [JSS+14], but the approach still present some limitation. The machine-readable interface definition for the web services supported by its current tools is WSDL with RPC-literal bindings, as specified in the WS-I base profile. The current technical implementation of Reo excludes circuits with channels that modify the data passing through them. This confirms that Reo focuses on a relatively low level of interoperability, such as communication protocol mismatch. The syntactic aspect, which concerns the data format, and the semantic aspect, which affects the meaning given to the information, are not addressed. In addition, Reo focuses on choreography and does not cover other types of connectors, such as file transfer.

Autili *et al.* [ADSG+18c] describe a model-driven approach to manage the evolution of choreographies through variability. The approach consists of synthesizing coordination and adaptation software entities to represent and control the interactions of the services participating in the choreography. The use of model-driven provides the opportunity to evolve the coordination logic to allow for a modular evolution of the choreography in response to possible changes in

context. At the design stage, a BPMN2 modeler will allow variation points to be specified directly in the choreography diagrams.

It defines a new version of coordination delegates, namely, evolvable coordination delegates (eCDs) presented above in [AIT18], which will manage the evolution of the choreography in response to changes in context. However, only one variation can be enabled at runtime, and the execution of a variation is treated as atomic by the CDs. Use the meta-class to solve the BPMN2 lake to represent variability. To do this, the variation point metamodel must present a part of the BPMN2 metamodel, which the authors have extended to support the specification of variation points and variants of the choreography. It will be beneficial to use the feature model and the metamodel to present variability and manage variability to create other types of variability, not only for choreography.

2.6 Summary

After reviewing the state of the art, we have identified several limitations that may restrict open and evolving interoperability.

Approaches that allow the dynamic reconfiguration of services, while interesting, are specific to orchestration, where the connector is centralized. These solutions are not designed to exploit variability to generate several types of connectors for event-driven architectures. Also, these approaches do not consider the exogenous connector, making it difficult to change a single connector without affecting other combinations.

Approaches that address automatic connector synthesis, mainly focus on choreography, which involves several peer-to-peer communications to create a network system. However, these connectors are not preferable when dealing with many services, as they can create a spaghetti code that is difficult to maintain. Moreover, these approaches do not allow the generalization of other types of connectors, such as asynchronous event-driven communication through a single broker.

Some approaches require resources that a client may be unable to provide, such as a domain ontology or a state machine specification. It can be limiting for partners who need help providing these resources.

While a model-driven approach takes a big step towards scalable and maintainable connectors, some are only interested in low-level interoperability or require predefined scenarios. Also, the variability is not sufficiently exploited to facilitate scalability further, even if the abstraction effort is made.

Our proposal is to confirm that the connector is an independent component. We will do this by proposing a connector model that covers the different possible connectors while remaining extensible for future evolutions. We also want to exploit the commonalities of the connectors to facilitate the development of specific connectors through model-driven engineering.

Chapter 3. Reifying Interoperability Mechanism: An Extensible Metamodel for Software Connectors

This chapter centers on interoperability mechanisms as a tangible constituent of information systems. We begin by presenting two basic examples of interoperability mechanisms implemented ad hoc or using independent constituents. We will then guide you through analyzing and modeling reified connectors. We will show how we built a repository of projects involving interoperability mechanisms for our study. Ultimately, interoperability mechanisms will no longer be considered merely source code scattered throughout business applications but as integral system components, bringing us closer to a System of System information system. This study focuses on asynchronous interaction.

3.1 Motivation for Reifying Interoperability Mechanisms

Reification, originating from philosophy and social theory, involves attributing real and tangible characteristics to abstract concepts or ideas. Vandenberghe in [Van01] defines the reification as *"the process of transforming human properties, relations, processes, actions, and concepts into things"*. It encompasses the transformation of the theoretical into the physical, highlighting the inclination to treat abstract notions as independently existing entities.

In software engineering, reification involves translating abstract concepts or ideas into tangible manifestations within a software system [BC10]. This process requires converting abstract notions, such as higher-level concepts or ideas, into implementable code or models. Reification is commonly used to design software models or systems representing real-world entities or abstract concepts. By representing these concepts as classes, objects, data structures, or interfaces, software engineers can manipulate and use them within the software.

In Object-Oriented Programming, reification is achieved by representing abstract concepts or entities as classes and objects [EKLG+03]. This approach enables developers to model real-world concepts in software by transforming abstract ideas into manipulable entities within the code. This practice significantly

enhances code organization, readability, and maintainability by aligning the software structure with the conceptual model of the problem domain.

For instance, consider the abstract concept of a *vehicle*. A class named *Vehicle* can be crafted through reification to encapsulate shared properties and behaviors among all vehicles. This class serves as a blueprint for creating specific instances or objects representing individual vehicles, such as cars, trucks, or motorcycles. Developers can interact with these entities using specific methods, properties, and behaviors defined within the software.

This chapter focuses on the concrete manifestation of interoperability mechanisms. Typically, interoperability is achieved by embedding interaction mechanisms in business logic code in an ad-hoc manner. Although these embedded mechanisms allow disparate systems to communicate and coordinate, they require significant effort to modify or update. Here, the concept of *Reification* seeks to transform these embedded interoperability mechanisms within business code into tangible system components.

These scenarios highlight the need to minimize human intervention in component interactions. Achieving interoperability in such evolving environments is challenging and requires adaptive interoperability mechanisms. In addition, these mechanisms must accommodate legacy applications that rely on outdated technologies and ensure their compatibility with updated components.

Reifying interoperability mechanisms means recognizing their explicit and complex role within the system. For example, in today's digital landscape, two prevalent contexts-ubiquitous computing and system-of-systems (SoS)-shed light on this complexity. Ubiquitous computing involves components distributed across multiple platforms, enabling spontaneous interactions without prior familiarity [LY02]. Conversely, an SoS consists of independent constituent systems that self-manage, operate, and evolve, resulting in emergent behavior directed toward specific goals [BS06]. The reification process emphasizes interoperability mechanisms through dedicated constituents called *Interoperability connectors*. Each connector operates independently, allowing modeling, deployment, generation, and monitoring to separate from the business logic constituents.

Definition of Connector *An interoperability connector is a primary component that plays a critical role in enabling interoperability between the system's business constituents or other connectors. Its primary function is to enable communication and coordination despite potential differences in technical, syntactical, and semantic specifications between the interconnected constituents.*

This definition shows that connectors are constituents like any other, with the difference that they only include functions relating to communication and coordination. Connectors can be combined to create other connectors for emergent behavior.

Figure 3.1 illustrates a scenario where components from two systems need to communicate or coordinate utilizing interoperability mechanisms embedded within the business logic constituent. Conversely, in Figure 3.2, interoperability between constituents of the two systems is facilitated through the reified concept of the *Connector*.

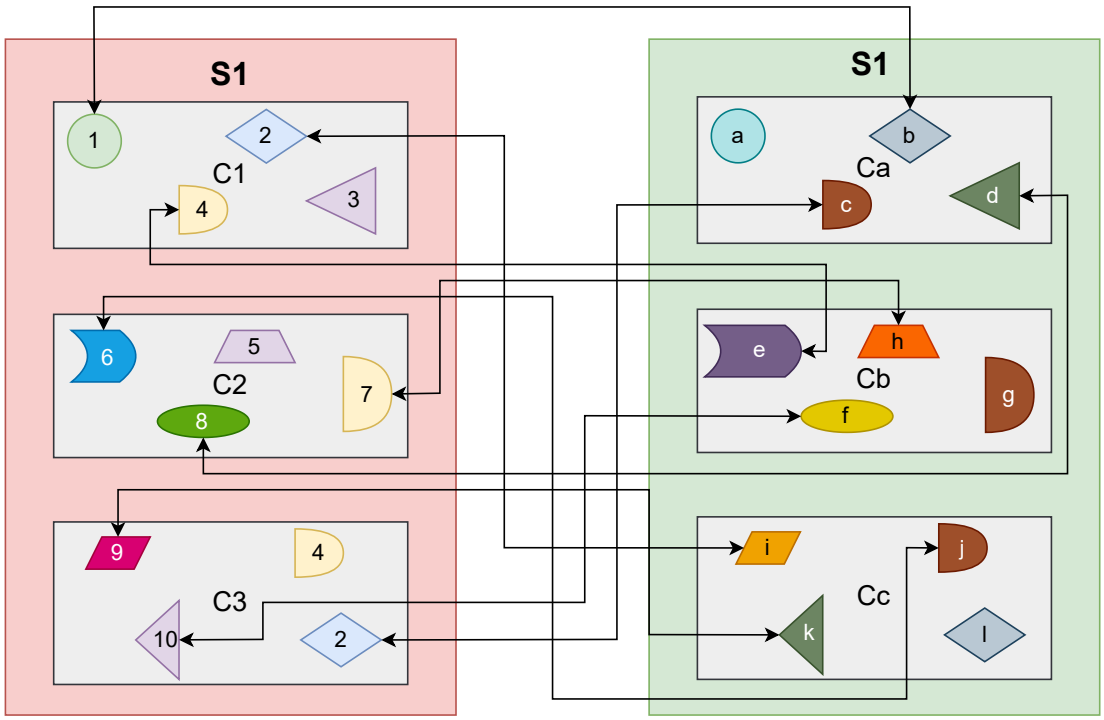


Figure 3.1: Scenario showing interoperability between constituents of two systems through interoperability mechanisms embedded in business constituent

Figure 3.1 illustrates a scenario where components from two systems need to communicate or coordinate utilizing interoperability mechanisms embedded within the business logic constituent. Conversely, in Figure 3.2, interoperability between constituents of the two systems is facilitated through the reified concept of the *Connector*.

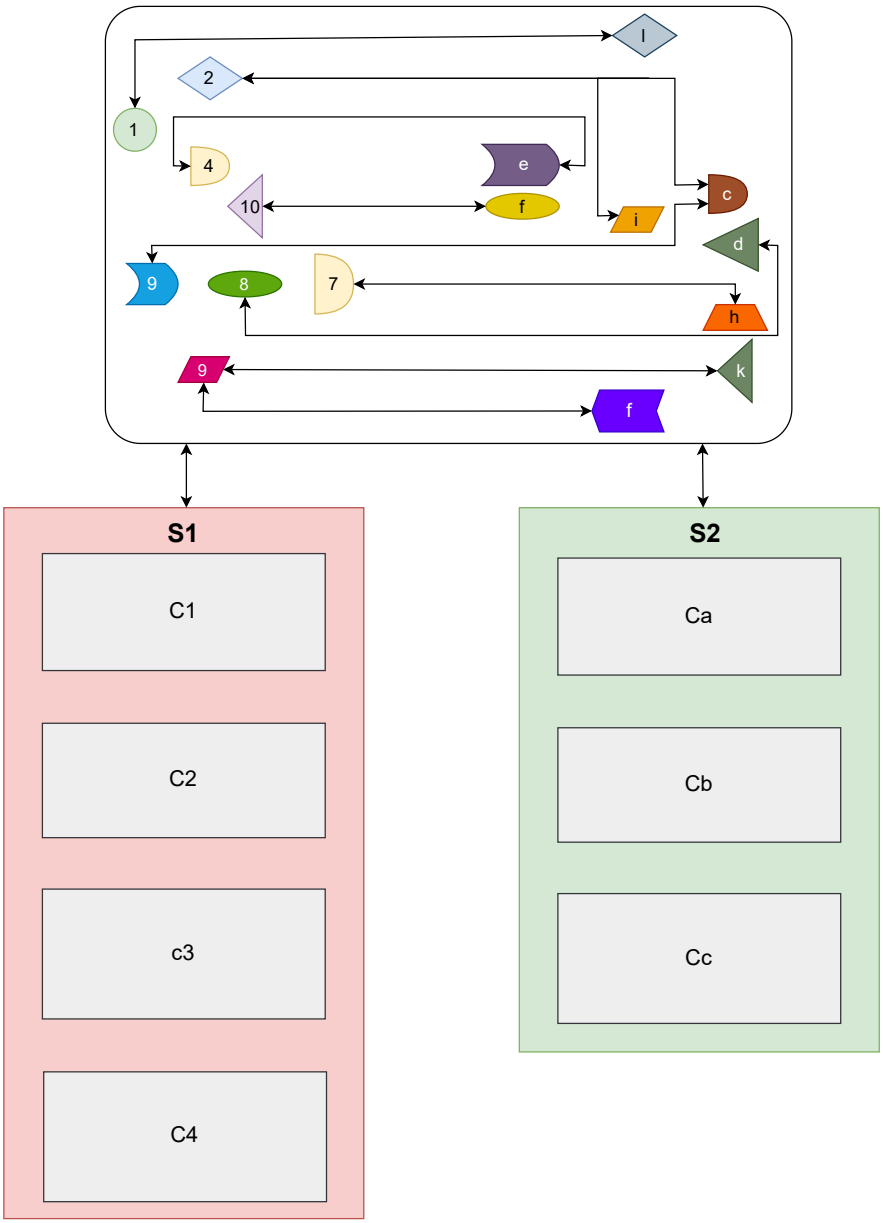


Figure 3.2: Scenario showing interoperability constituents of two systems through *reified* called *Interoperability connector*

In Figure 3.1, there are two systems, S1 and S2, each containing several business logic constituents—C1, C2, and C3 for system S1, and Ca, Cb, and Cc for

system S2. In contrast, Figure 3.2 depicts systems S1 and S2, each encompassing multiple business logic constituents—C1, C2, C3, and C4 for system S1, and Ca, Cb, and Cc for system S2.

Both Figure 3.2 and Figure 3.1 depict constituents that embed sets of interoperability mechanisms that are source code snippets. These mechanisms, represented by different geometric shapes, colors, and numbers, denote code blocks, programming languages, and identifiers. Different shapes indicate different code blocks, sizes within the same shape indicate similar codes of different lengths, and identical shapes but different shapes indicate the same code block in a different language. Mechanisms with no incoming and outgoing forms represent dead code.

In comparing the two scenarios, the differences in connector usage are apparent.

Figure 3.1 embeds interoperability mechanisms within the business logic code, while Figure 3.2 externalizes these mechanisms within the reified *connector*. Another difference is that in scenarios with a reified connector, interoperability exists once in the connector instead of duplicated across constituents when the connector is not reified.

In addition, when considering a scenario with a reified connector and focusing on constituent C4, updates only require adding missing existing interoperability mechanisms without introducing new ones. Conversely, a scenario without a reified connector might require an update to component C4, potentially impacting the overall functionality of system S1.

Furthermore, reified connectors allow the removal of dead code without affecting the constituent. However, removing interoperability mechanisms would require updating and deploying the respective constituent for non-reified connectors. Additionally, a scenario with a reified connector aligns with the characteristics required for a System-of-System, ensuring technical, managerial, and geographic independence.

3.2 Methodology for the Reification of Interoperability Connectors

In our context, reifying interoperability mechanisms involves extracting the source code related to interactions and separating it from the business logic code, creating a distinct entity known as a connector.

The process of specifying interoperability mechanisms involves several steps. First, identifying interoperability mechanisms involves analyzing implemented use cases to create a repository, including conceptual and implementation elements. Conceptual components include specifications, hypotheses, and interaction-related

principles, while implementation components are model entities or code snippets scattered throughout the source code.

Next, understanding existing interoperability mechanisms involves extracting information related to the interaction from the repository without business logic. This step helps to identify different concepts and their characteristics in terms of entities and their properties.

The next step is the classification of interoperability concepts, where rules are established to classify each idea into concepts such as message endpoint, routing concept, and data transformation. Subcategories can be created based on interoperability requirements such as frequency, maximum document size, and synchronous or asynchronous communication types.

The subsequent step is the classification of interoperability concepts, where rules are established to classify each idea into concepts like message endpoint, routing concept, and data transformation.

Finally, interoperability mechanisms are materialized by considering them as first-class constituents within the systems. This results in reified interoperability mechanisms, called connectors, which emerge from studying recurring patterns in system interactions. Connectors represent the concretization of interoperability mechanisms. In addition, a metamodel is introduced to outline key interoperability specifications, using insights from existing solutions found in the literature, vendor offerings, and industry practices.

Together, these steps contribute to a comprehensive understanding of interoperability mechanisms and the creation of tangible connectors through metamodels.

3.2.1 Building a Repository for Analyzing Interoperability Mechanisms

Interoperability remains a pivotal aspect of software engineering because software systems must interact to accomplish various tasks [AIS+19]. Despite its importance, no established repository of interoperability mechanisms or connectors exists. This absence is primarily due to the ad hoc development of most interoperability mechanisms, often implicit and hidden in the business logic. Proposing a common interoperability solution would require sharing a portion of an organization's business source code, a prospect generally met with reluctance by industry stakeholders. Furthermore, identifying these mechanisms is challenging due to their implicit nature, and proposing a repository is equally complex.

The book *Enterprise Integration Patterns* (EIP) by Hohpe and Woolf [HW04] serves as a comprehensive guide to messaging patterns relevant to the implementation of interoperability solutions. Focusing primarily on messaging for asynchronous interaction, EIP presents sixty-five patterns categorized and identified by specific names, visual representations, or icons. These patterns can potentially form a connector metamodel that includes a set of relevant entities and their relationships.

The reification introduced in this chapter extends the concepts of EIP but with a difference in granularity. While EIP encapsulates a set of partially reified concepts, our metamodel serves as a reified connector. From a Unified Modeling Language (UML) [MG00] class diagram perspective, EIP can be seen as a collection of entities, while the reified connector is an aggregation of these entities represented in the form of a metamodel.

The entity representing the reified connector in the metamodel incorporates the properties described by EIP and supplements them with properties derived from industry or literature, considering that EIP dates back to 2003.

Analyzing and Collecting Data on Messaging-Based Interactions: The initial data is sourced from Enterprise Integration Patterns (EIP), which includes various concepts essential for achieving interoperability. EIP provides messaging patterns designed to address common interoperability challenges within enterprise software systems, offering a consistent approach and terminology to design scalable and maintainable interoperability solutions.

Although EIPs do not explicitly present the concept of connectors proposed in this thesis, they lay the foundations for potential connector functionalities. However, the current research emphasizes reification at the connector level rather than solely on its constitutive functions. In this context, the connector itself is considered the system, and the representation of EIP pattern entities is feasible, albeit not encompassing all connector system constituents.

It is crucial to recognize that the list of presented patterns serves as a starting point. While the EIP may not fully cover specific needs, solution vendors can implement new patterns in their industrial solutions, contributing to the enrichment of the dataset. The dataset aims to encompass various models from different origins, including vendor-neutral, vendor-specific, and industrial-specific solutions.

Vendor-neutral solutions: are interoperability solutions and patterns not tied to any specific vendor's products or technologies. These solutions operate inde-

pendently and harmonize seamlessly with diverse vendors' systems or platforms, ensuring adaptability and compatibility across varying environments. Examples of such solutions include Apache Camel, Spring Cloud Data Flow, and Spring Integration.

Apache Camel is an open-source integration framework facilitating seamless integration across diverse systems, applications, and protocols. It offers a versatile platform for implementing intricate routing and mediation rules in numerous integration scenarios [Cam21].

Spring Integration extends the Spring framework to enable the creation of enterprise integration solutions. It enables the development of messaging solutions and facilitates integration between disparate systems or applications [Pan15].

Spring Cloud Data Flow is a framework that streamlines the development and orchestration of data processing pipelines and microservices on modern runtime platforms [GG21].

Vendor-specific solutions: are designed to be interoperable with a specific vendor's technologies, products, or services. These solutions are typically customized to take advantage of specific features, functionalities, or proprietary aspects of a particular vendor's ecosystem, providing deeper integration but potentially limiting interoperability across multiple platforms. Examples include Microsoft BizTalk Server and MuleSoft Anypoint Platform [MMS19, AMS22].

Microsoft BizTalk Server is an integration server product developed by Microsoft that provides a platform for building and managing integration solutions that facilitate communication and data exchange between disparate systems. Microsoft BizTalk Server requires Windows Server OS and Microsoft SQL Server[PR22], with proper licensing essential for compliance with Microsoft policies [Dau22].

MuleSoft Anypoint Platform [1] is an integration platform that enables businesses to connect applications, data, and devices across on-premises, cloud, and hybrid environments. Anypoint Studio is required as the development environment, along with proper licensing or subscription plans based on intended usage and capacity within the platform.

Industrial-specific solutions: refer to proprietary interoperability solutions that are specialized or prevalent interoperability solutions within a particular company. These solutions address industry-specific challenges, requirements, or

[1] `https://www.mulesoft.com/`

standards, ensuring effective communication and compatibility within that specific industrial need. For instance, BL-MOM used in [ALL+20, LATN+23], is an example of such a solution. It has been developed by the Berger-Levrault company [2]. BL-MOM is a messaging-based library that aims to support the implementation of a new connector based on the publish-subscribe communication pattern. It is based on the RabbitMQ broker [Tos15].

Figure 3.3 summarizes the data collection process for building the connector repository toward reification.

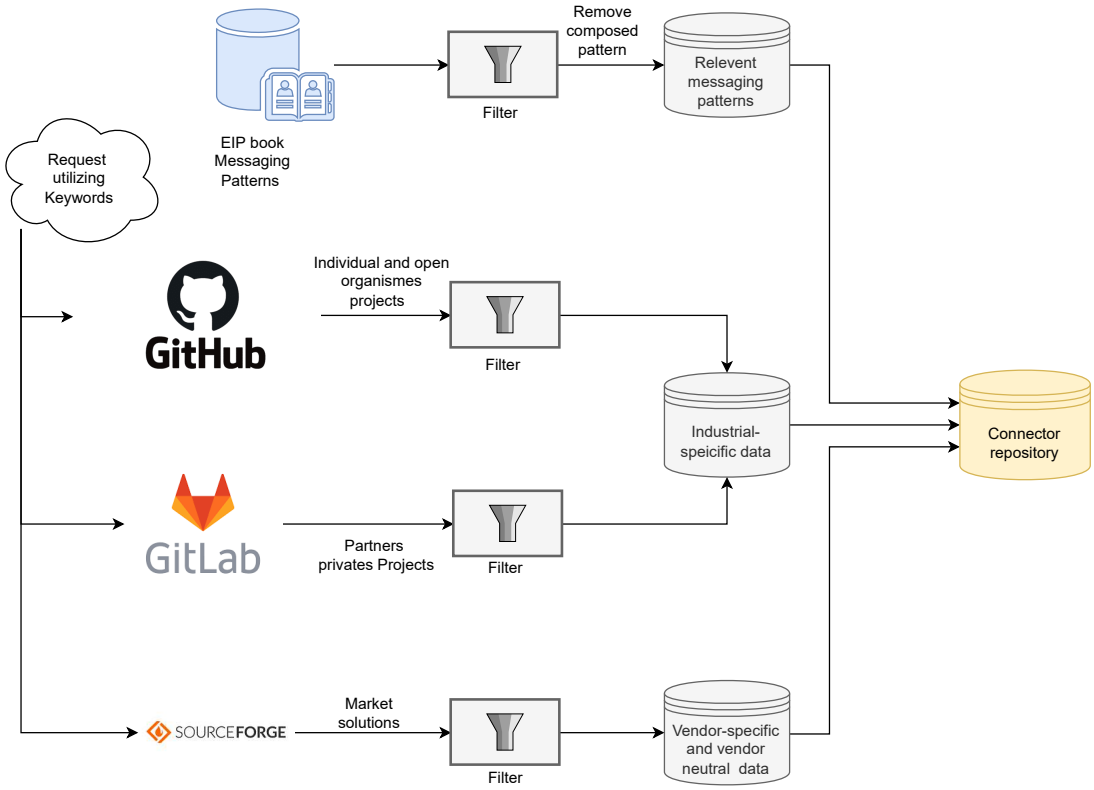


Figure 3.3: Overview of Messaging Data Collection Process for Building a Connector Repository

The data collection process involves simplifying the Enterprise Integration Patterns (EIP) by eliminating complex patterns, especially those that can be achieved by combining other patterns. For example, patterns such as the *normalizer* can be created by combining a router and a set of transformers to translate

[2] `https://www.berger-levrault.com/fr/`

messages into a unified format, resulting in a focus on basic patterns [3]. Data collection is initiated by requesting information from various projects and existing solutions using keywords such as connector, mediator, middleware, interoperability, integration, and specific project names such as Apache Camel, RabbitMQ, Apache Kafka, Spring Cloud Data Flow, and Spring Integration.

To gather information, we explore both organizational and individual open projects on GitHub [4], while private interoperability projects from our industry partner, Berger-Levrault, are accessed through the company's private GitLab [5]. Additional market solutions are identified on SourceForge [6]. The results obtained from the requests are refined based on expert knowledge, which includes removing non-relevant projects (e.g., empty projects). The connector repository is built once relevant data is collected from these various sources.

3.2.2 Concretization of the Reification: Metamodel for the Messaging Connector

In the previous subsection, we focused on collecting data about the interoperability of constituents and systems, particularly regarding messaging interaction styles. Our analysis was EIPs, vendor-specific solutions, vendor-neutral solutions, and industrial-specific solutions, which showed that interaction can be achieved effectively through a reified system constructed from reified entities derived from EIPs. This study highlighted the crucial importance of the messaging connector as a primary entity in the system.

This section proposes a metamodel that precisely defines the reified messaging connector. The metamodel outlines the characteristics and relationships among the entities that constitute the messaging connector.

As defined by Garcia [GMFFGS09], a metamodel is an abstract representation that encapsulates a software system's structure, entities, relationships, and constraints. The proposed metamodel captures the essential components, relationships, and rules that govern the structure of the messaging connector.

In Model-Driven Engineering (MDE) [S+06], the metamodel holds significant importance by formally describing elements, their interconnections, and properties within a modeling language or framework.

[3] `https://www.enterpriseintegrationpatterns.com/patterns/messaging/`
[4] `https://github.com/`
[5] `https://about.gitlab.com/`
[6] `https://sourceforge.net/`

Before introducing the metamodel for the reified messaging connector, it is crucial to address the rationale behind choosing a metamodel to represent the messaging connector effectively and discuss the potential benefits of leveraging model-driven engineering to its full realization.

3.2.3 The Importance of Using a Metamodel to Represent the Reified Messaging Connector

Connectors undergo reification, transforming into visible components within the system, as shown in Figure 3.11. Their tangible forms may include representations such as classes, black boxes, or other visualizations. However, this thesis focuses on using and reusing the connector and its functionalities as integral elements. Several scenarios can be envisioned to achieve this goal.

Challenges Associated with Implementing a Universal Mega-Connector for Exchange Flows in Programming: The concept of a mega-connector designed to handle all possible interactions is intriguing but presents several challenges. Firstly, it requires predetermined technological decisions, such as building the mega-connector in a specific programming language like Java. Secondly, this solution may be disproportionate to most exchange flows. Maintaining extensive code can be cumbersome, primarily when only a fraction is used for each flow. The complexity of flows increases the risk of regression, where changes in one flow affect others that remain unchanged. Additionally, identifying and rectifying issues reactively between components within such a complex system can be challenging. Onboarding new developers may also face obstacles due to the high learning curve and associated costs. Moreover, relying on a single mega-connector creates a single point of failure, which could disrupt the exchange flow if the server hosting the mega-connector crashes.

Challenges in Utilizing a Catalog of Pre-Built Connectors Developed in Programming: Although pre-built connectors developed in programming may be appropriate in a static environment, they face challenges in scalability and maintainability within the dynamism of contemporary systems. Handling multiple pre-built connectors at the code level, each tailored to a specific language requires the maintenance of numerous connectors similar to catalog items. Customizing connectors to meet specific requirements, such as adapting them to languages like C++ or Java, presents significant challenges. For example, transitioning from File Transfer Protocol (FTP) to Secure File Transfer Protocol

(sFTP) due to the introduction of Secure Shell (SSH) [Coh02] requires updating all catalog connectors and adjusting unit tests across multiple projects.

In the context of the continuous evolution of systems and their constituents, both scenarios have notable limitations. The constraints posed by specific technologies, complexities in maintenance, scalability concerns regarding adaptability, and the inherently static nature of these approaches collectively suggest using a metamodel.

3.3 Introducing the Metamodel of the Messaging Connector

This section introduces the messaging connector metamodel developed based on the connector repository built from EIP and existing interoperability solutions.

Figure 3.4 presents the messaging connector metamodel class diagram.

The presented metamodel provides a comprehensive exploration of a connector, affirming the proposed reification for interoperability mechanisms. It emphasizes achieving interoperability through a messaging style, particularly asynchronous communication. The overarching *Connector* entity within the system is a composite that integrates diverse entities through composition relationships, covering all common elements. Within the composite *Connector* class, various component classes, including *Message*, *InputEndpoint*, *OutputEndpoint*, *Channel*, *Transformer*, and *Router* are intrinsically connected to the connector. Their existence is contingent on the connector itself, dictated by the nature of the composition relationship.

Following the entity overview, a thorough exploration of the relationships among the main entities is provided. The *OutputEndpoint*, serving as the sender, can publish messages across multiple messages *Channels*, while the *InputEndpoint*, functioning as the receiver, can subscribe to one or several messages *Channels*. The *OutputEndpoint* can produce messages across multiple Message Channels, while a given recipient may consume varying numbers of messages, occasionally none, based on immediate needs.

A message *Channel*, facilitated by the message *Queue*, can handle zero or more *Messages*. In contrast, message *Routers* can receive data from zero or more *Channels* and direct this data to zero or more output *Channels*.

Message *Transformers* are designed to take one message from an input message *Channel* and provide one or more *Messages* to an output message *Channel*. They are not confined to specific input or output message *Channels*, message *Transformers* can connect to multiple input and output Message Channels.

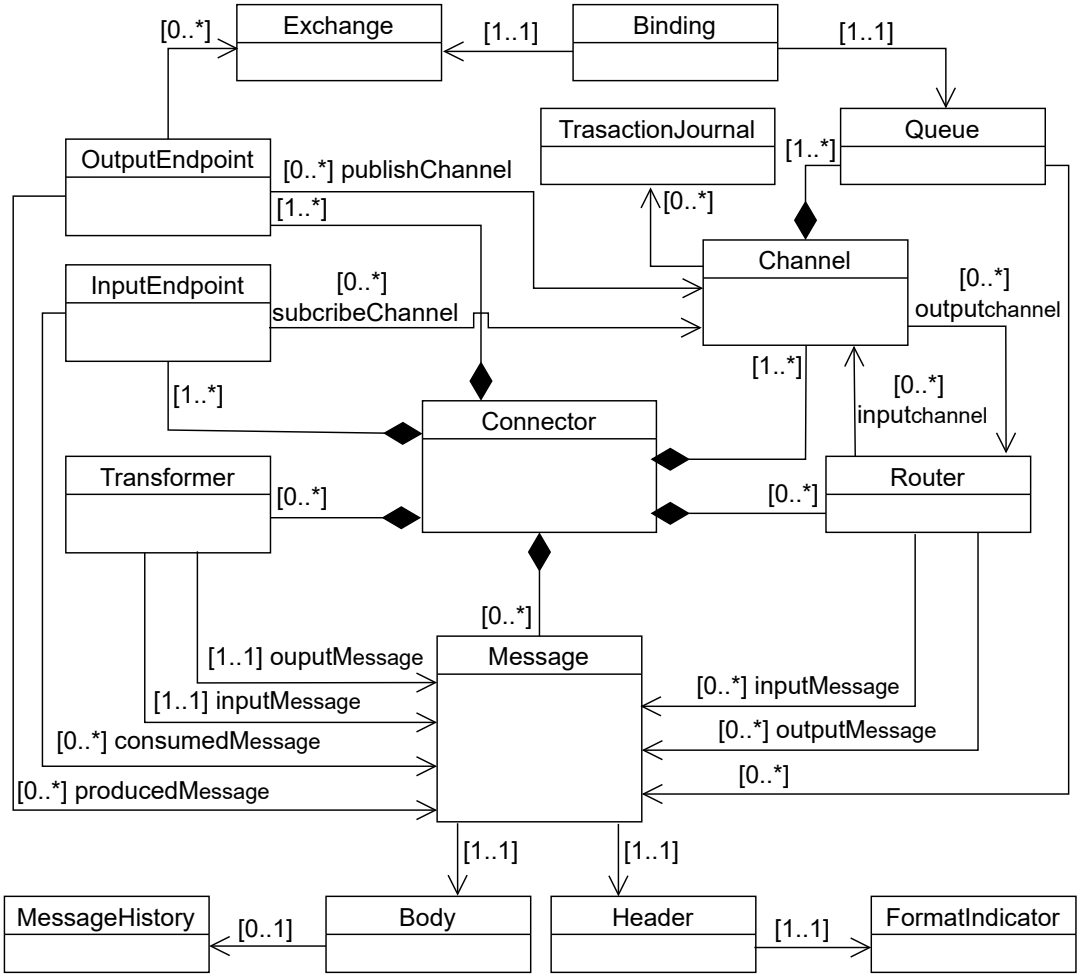


Figure 3.4: Metamodel Overview: Core Entities Shared Among All Messaging Connectors

In addition to this overview, it is necessary to examine the metamodel in detail to understand the capacity of the entities and the metamodel itself.

3.3.1 Detailed presentation of the metamodel:

The entities represented in the metamodel 3.4 can take different forms based on communication and coordination needs, allowing for creating different types of connectors. This observation emphasizes that connectors share common entities that can be adapted to specific interaction requirements.

This statement reveals that the reified messaging connector is as a Software Product Line (SPL)[CN02]. A detailed exploration of this concept is undertaken in chapter5. Before delving into the specifics, the details of each component class within the composite class *Connector* will be presented for clarity, focusing on one entity at a time.

***Message*:** A message consists of two parts: the header and the body. The header contains essential technical information such as the format indicator for identifying the version and document format, a timestamp, a correlation identifier for grouping, a return address, an expiration timestamp, and a priority level. On the other hand, the body contains the essence content of the message that contains data or instructions for processing by the receiving system.

Figure 3.5 provides a detailed view of the metamodel, zooming in on the specifics of the message entity.

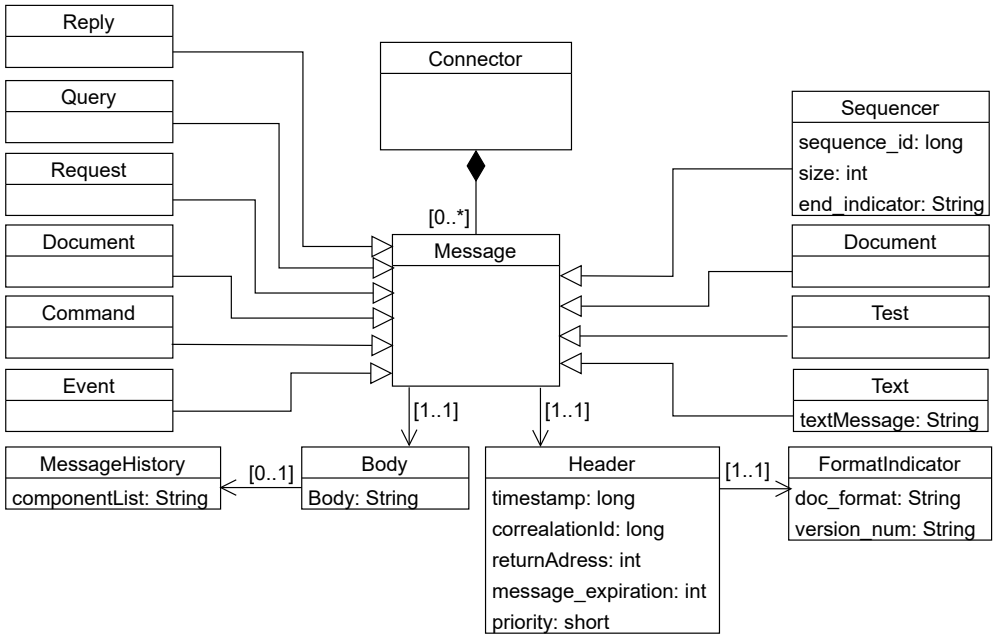


Figure 3.5: Messaging Connectors: A Comprehensive Overview Focused on Message Entities

Table 3.1 describes the different variants of the message entity, with some example [7]. It provides a clear overview of each message variant and its distinctive characteristics, aiding in understanding the metamodel entities.

Message type	Description	Example
Document	Facilitate the transfer of a data structure between constituents without explicitly instructing the receiver on handling the data.	Product
Event	Notifies occurrences with event details and a reference, emphasizing timing and content distinctions from document messages.	UserUpdated
Command	Contains precise instructions for a specific recipient, promoting sender-receiver coupling but maintaining decoupling through message *Queues*.	ConfirmOrder
Request	An inquiry for specific action or information, allowing negotiation and distinct from commands in terms of politeness and compliance expectation.	PostUser
Query	This is a specific request for information retrieval. It involves posing questions or searches to a system or database without commanding an action.	SelectUserById
Reply	Responds to previous queries, commands, or requests.	UserReply
Text	Short written messages exchanged between system constituents. These messages are often informal and used for quick communication.	Hello
Sequencer	Divides large amounts of data into message sequences for partial transfer, requiring identifiers for order and size.	Hello "sequenced" in H — e — l — l — o
Test	Serves for system testing can be of various types, such as *Events*, *Commands*, or *Documents*.	Abcd1efg0

Table 3.1: Summarizing variants of the *Message* entity

OutputEndpoints: is a specialized endpoints that send messages to other constituents or systems within a connector. While a connector typically includes at least one *OutputEndpoint*, scenarios may demand more. For instance, a dedicated *OutputEndpoint* can be crucial for internal communication within the same system, while another can transmit data to external systems.

An *OutputEndpoint* can transmit messages directly to a message *Channel* or through an *Exchange*. The *Exchange* is a constituent, receives messages from producers, and forwards them based on routing criteria to *Queues* or other *Exchanges*. It can be *Direct* (dedicated to a specific Queue), *Fanout* (broadcasting to all Queues), or *Topic* (matching specific criteria).

Figure 3.6 illustrates the metamodel, highlighting *OutgoingEndpoints* and variant entities.

The descriptions of *OutputEndpoint* variants are listed in Table 3.2.

[7] `https://thehonestcoder.com/types-of-messages-in-message-driven-systems/`

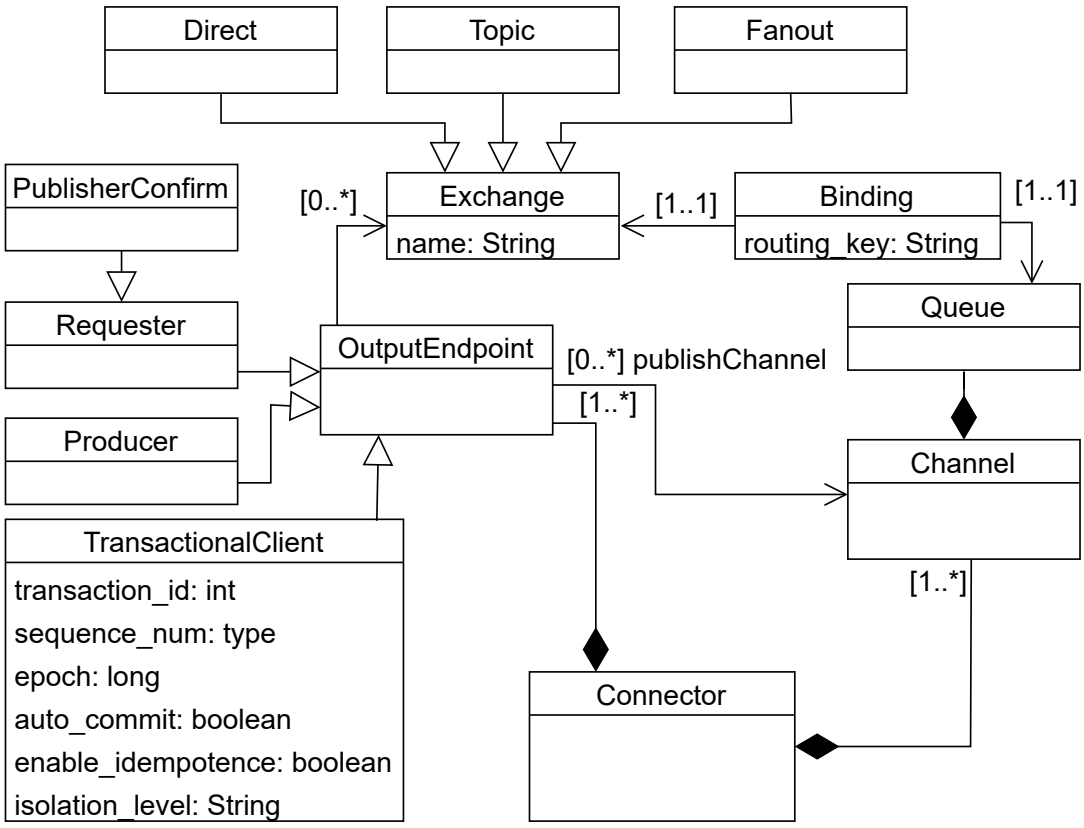


Figure 3.6: Messaging Connectors: A Comprehensive Overview Focused on OutputEndpoint Entities

Output Endpoint Type	Description
Requester	Dispatches one-to-one requests to a message queue within a *Message channel*. Messages sent by a *Requester* are directed to a single *Replier*, which acknowledges reception. Its primary use is for asynchronous implementation of Request-Reply patterns.
Producer	Publishes *Event* messages utilizing or not an *Exchange* within a *Publish-Subscribe channel*, a type of *Channel* that delivers a copy of a particular event to each, targeted, or set of subscribed receiver. This process adheres to the fire-and-forget principle, where messages are dispatched without waiting for or expecting a reply.
Publish Confirms	Publishes *Event* messages to multiple recipients, expecting asynchronous responses. This is a variant of *Producer*.
Transactional Client	Enables clients to set transactional boundaries during interactions with the messaging system. Both the sender and receiver can operate within these transactions. Messages are not instantly added to the *Message channel* upon sending; they wait until the sender commits the transaction. Similarly, they persist in the channel until the receiver commits the transaction.

Table 3.2: Summarizing variants of the *OutputEndpoint* entity

InputEndpoint: is a specialized endpoint intended for receiving messages from other systems or constituents. A connector includes at least one Input Endpoint, although some scenarios may necessitate multiple instances. A dedicated *InputEndpoint* may be essential for internal communication within the same network, while another can be reserved for receiving data from external systems. This serves as the entry point of the system, reinforced with necessary protection against external threats.

Figure 3.7 provides an overview of the metamodel, emphasizing *InputEndpoints* and their variant entities.

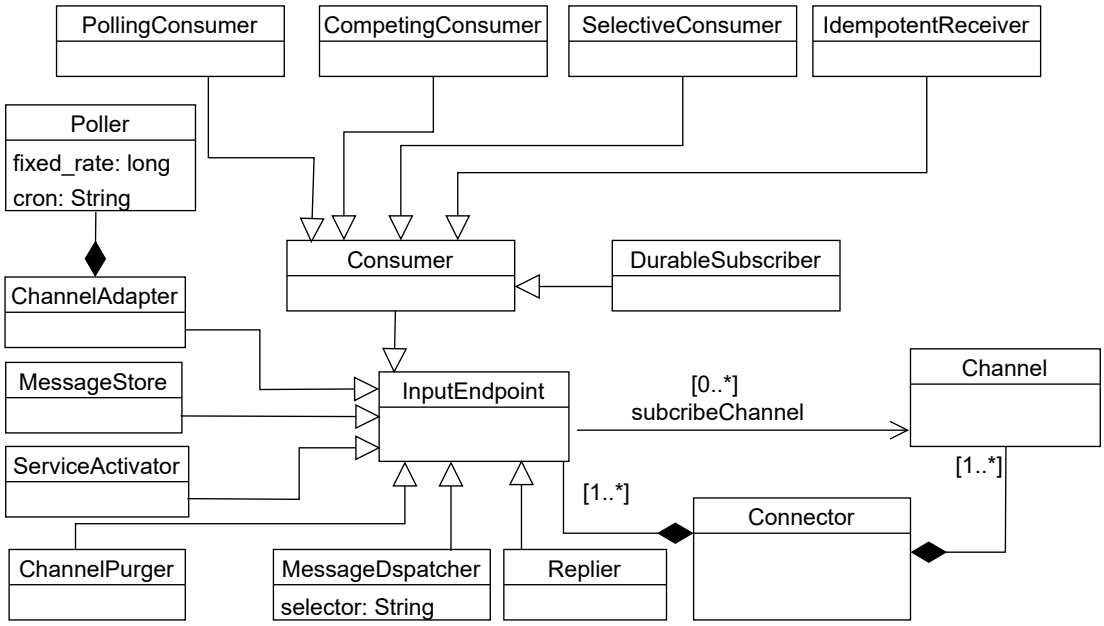


Figure 3.7: Messaging Connectors: A Comprehensive Overview Focused on InputEndpoint Entities

Table 3.3 provides a concise overview of the various *InputEndpoint* types and their respective functionalities within the messaging system.

Channel: A connector includes at least one *Channel*, sometimes referred to as a *Destination*. The Message *Channel* serves as a conduit for transmitting messages between systems. When system constituents exchange data, they utilize a *Channel* to facilitate communication. Even without knowing the exact recipient, the sending application sends data through a message *Channel*, expecting the

Input Endpoint Type	Description
Replier	Responsible for receiving requests or queries and asynchronously sending one-to-one replies to a specific message *Queue* within a message *Channel*.
Consumer	Subscribes to *Event* messages from one or multiple message Queues within a message *Channel*, adhering to the Publish-Subscribe pattern.
Competing consumer	Dedicated to processing messages from a single message *Queue*, essential for concurrent task handling.
Selective consumer	Exclusively retrieves messages from a specified *Topic* (*Exchange* adhering to a particular pattern, contrasting a general event Consumer.
Polling consumer	Actively seeks to receive a message when prepared, useful when preserving messages in the *Queue* is more efficient.
Durable subscriber	Capable of directing the messaging system to store published messages while the subscriber remains disconnected.
Idempotent receiver	Receiver capable of handling duplicate messages
Message dispatcher	Consumes messages from a message *Channel* and internally distributes them to processors handling input and output messages.
Service activator	Connects messages within the message *Channel* to the appropriate service, operating in one-way or two-way mode.
Channel adapter	Links constituents of the system to the messaging system, facilitating message transmission and reception based on data.
Channel purger	Clears undesired messages from a *Channel*, ensuring the restoration of the system to a stable state.
Message store	Stores details of every message in a central database-like repository, primarily for business analytics and auditing.

Table 3.3: Summarizing variants of the InputEndpoint entity

receiver to search for the specific data type within it. This approach enables constituent producers of shared data to interact with those intending to consume it. Messages in Message *Channels* are stored in separate *Queues*, providing isolation. These *Message channels* can consist of one or more multiple *Queues*, each holding messages awaiting consumption.

Figure 3.8 provides an overview of the metamodel, emphasizing *Channel* and their variant entities.

Table 3.4 provides a concise overview of the various *Channel* types and their respective functionalities within the messaging system.

These *Channels* are commonly implemented using popular brokers and messaging systems like Java Messaging service JMS [HBS+02], OpenMQ [ABKM06], Apache Kafka [Gar13], RabbitMQ [Tos15], ActiveMQ [CC19], ZeroMQ [S+15].

Router: In the connector, multiple messages *Routers* direct messages from senders to specific receivers, reading from diverse source channels and delivering to target channels based on predefined criteria. This enables flexible and adaptable message routing within a system, which is crucial for building resilient

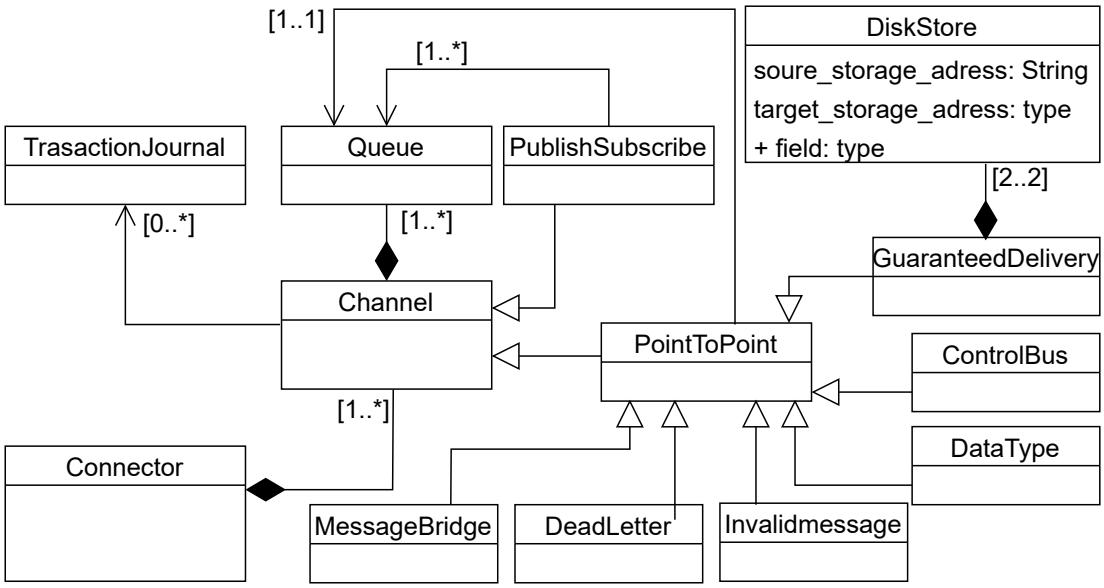


Figure 3.8: Messaging Connectors: A Comprehensive Overview Focused on Channel Entities

Channel Type	Description
Point-to-Point	Handle one message simultaneously, from a single sender to a sole consumer.
Publish-Subscribe	Enable communication between multiple senders and receivers. Support multiple queues, behaving like a Point-to-Point channel with several queues. Receivers consume messages based on subscriptions to specific queues, with *Exchange* types such as Direct-, Header-, Topic-, and Fanout-Type.
Data Type	Similar to Point-to-Point channels but exclusively accepts data of a specific type (e.g., invoice, order, medical prescription).
Invalid message	Contains messages that do not match the expected data type and are therefore rejected.
Dead Letter	olds messages that expire and become irrelevant after a specified time.
Guaranteed delivery	specialized Point-to-Point channel with persistence capabilities, ensuring message delivery even in case of message *Channel* failure. Uses disk storage at both sender and receiver ends.
Message Bridge	Facilitates seamless communication between different systems or channels by serving as an intermediary that bridges the gap between them.
Control Bus	Dedicated to transmitting data relevant to managing components involved in the message flow, such as the monitoring system.

Table 3.4: Summarizing variants of the Channel entity

and easily maintainable messaging systems without directly impacting senders or receivers.

Figure 3.9 provides an overview of the metamodel, emphasizing *Router* and their variant entities.

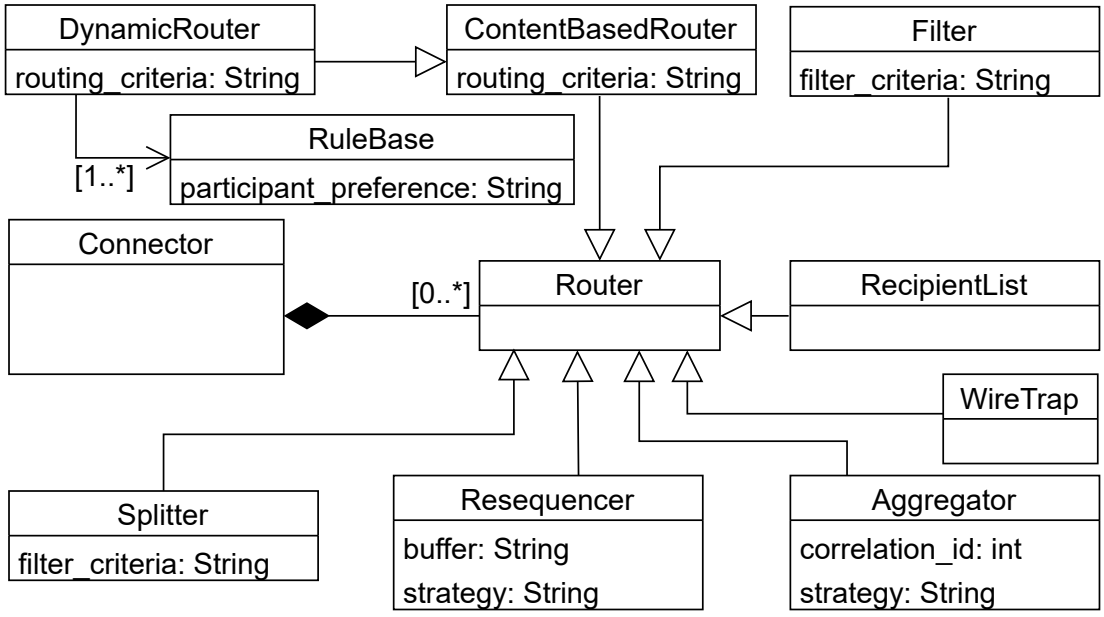


Figure 3.9: Messaging Connectors: A Comprehensive Overview Focused on Router Entities

Table 3.5 provides a concise overview of the various *Router* types and their respective functionalities within the messaging system.

Transformer: The connector can includes message *Transformer* to convert data from one format to another. This is necessary when the sender and receiver use different message formats and versions or employ distinct fields within the message structure.

Figure 3.9 provides an overview of the metamodel, emphasizing *Router* and their variant entities.

Table 3.6 provides a concise overview of the various *Router* types and their respective functionalities within the messaging system.

3.3.2 Revealing the Concrete Connector: A Comprehensive Overview

The process of utilizing data from various interoperability mechanisms leads to the formation of a tangible connector. This concept is supported by different metamodels discussed in Section 3.3. These models challenge the idea of connectors being solely embedded mechanisms within business logic code. In-

Router Type	Description
Content-Based Router	Directs messages based on their content by evaluating predefined conditions or criteria and dynamically routing them to specific message *Channels* based on their content characteristics.
Dynamic Router	Directs messages based on rules within a dynamic rule base, allowing flexible routing. Updated actively by consumers, enabling directives for relaying messages to specified message Channels.
Filter	Selectively processes or allows the passage of messages based on predefined criteria or conditions. It screens incoming messages, permitting only those that meet specific criteria to proceed, providing a mechanism for refining and directing the data flow within a system. For example, let pass only those that are ordered and block invoices.
Aggregator	Merges messages from a channel into a single output message using aggregation strategies. Correlation and completeness conditions are employed to organize messages into the correct order after aggregation. Popular strategies include *wait for all*, *time out*, *first best*, and *time out with override*, catering to specific scenarios.
Splitter	Break down a message into smaller sub-messages that can be processed individually. Each component is then directed to its respective output message channel.
Resequencer	Manages potentially out-of-order messages, reordering and publishing them while maintaining order. Requires order preservation in the output channel.
Recipent list	Identifies the intended recipients of an incoming message and transmits it to all associated message *Channels*. It relies on a list of recipients instead of message content and requires explicit subscription. This differs from Publish-Subscribe patterns, which involve no recipient information published in a queue.
Wiretap	Variant of *Recipient list* pattern with dual output *Channels*. Broadcasts messages from input *Message channels* for duplication on the *Control bus* channel 3.5.

Table 3.5: Summarizing variants of the Router entity

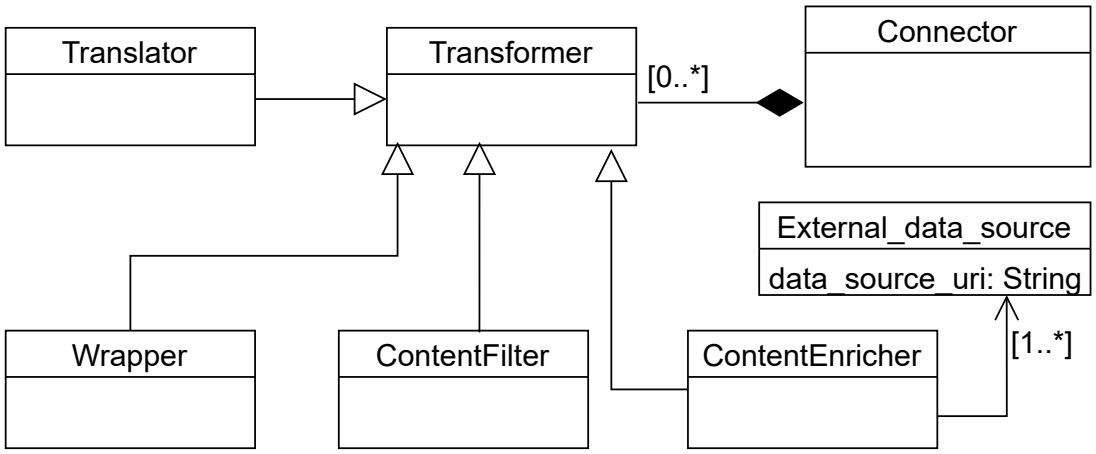


Figure 3.10: Messaging Connectors: A Comprehensive Overview Focused on Transformer Entities

stead, the concrete *Connector* is viewed as an actual constituent of the system, modeled to showcase its physical existence.

Transformer Type	Description
Message Translator	Reads data from a source message*Message channel* and converts it to a different format for an output *Message channel*. For instance, this may include data serialization from JSON to XML and vice versa.
Content Filter	Rremoves some original data, useful for scenarios like eliminating unnecessary information from a file, keeping the payload, removing sensitive information, or simplifying a message tree structure.
Content Enricher	Differs from the *Content Filter* by inputting a basic message and accessing an external data source to append missing information and augment the message.
Envelope Wrapper	Packages data in a messaging-compliant envelope, addressing scenarios where some constituents may require surplus header information for proper interpretation. Internally used by the connector for routing messages correctly to suitable *Message channels*. A *Wrapper* with essential processing details can be utilized, and an *Unwrap transformer* unpacks the message for receiver consumption.

Table 3.6: Summarizing variants of the Transformer entity

This section presents the internal constituents of a real connector to enhance technical and non-technical stakeholders' understanding of the connector concept.

Figure 3.11 depicts a holistic view of the reified messaging *Connector*. It highlights its complexity as a comprehensive system rather than just a part of a business component.

The Messaging Connector comprises several constituents, each serving distinct functions. The presented perspective emphasizes the connector's capabilities, including Endpoint, Channel, Router, Transformer, and Message features. It is important to note that these functionalities, including Data Type Channel and Dead Letter, can be customized to meet specific requirements. This section introduces variants, such as point-to-point channels, which are publish and subscribe channels.

We can view the connector as a software product line by identifying commonalities and variants in the metamodel. This global representation illustrates the messaging connector as a tangible entity, with each high-level component aligning with a corresponding class of the messaging connector metamodel in a broader context.

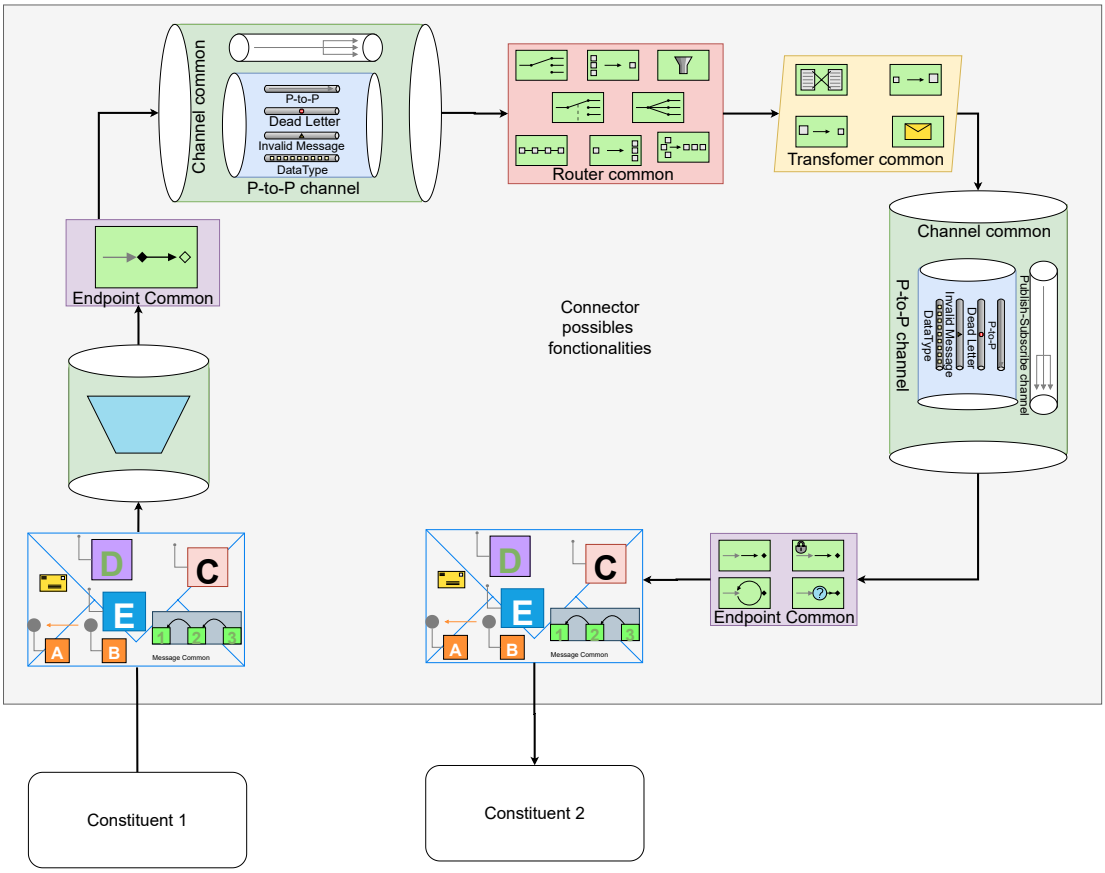


Figure 3.11: A Holistic View of the *Refied messaging connector*

3.4 Summary

In this chapter, we explore the idea that interoperability mechanisms should be seen as tangible components of a system rather than embedded source code in business logic. This approach is called reification. To verify the reification of interoperability mechanisms, we collected data from various projects that use messaging-style (asynchronous) interactions. We established a connector repository, which serves as a reference as interoperability mechanisms currently lack one. We collected data from various stakeholders, including vendor-specific, vendor-neutral, and industrial-specific projects and Enterprise Integration Patterns (EIP). Based on this data, we proposed a metamodel to represent interoperability mechanisms. The metamodel details reveal that messaging connectors have several common entities that can take different variants based on interaction

requirements, resulting in several variabilities. This finding leads us to explore Software Product Line Engineering in Chapter 5. We presented a holistic view of the reified connector to make it understandable for technical and non-technical stakeholders. The next step is to finalize a concrete use case for experimentation purposes. This experiment will enable us to compare the proposed connector to existing solutions not based on messaging style. The experimental phase will encompass performance and load tests, focusing on a use case from Berger-Levrault.

Chapter 4. Validating the Completeness and Extensibility of the Messaging Connector Metamodel and Conducting performance tests on the Reified Messaging Connector

The validation process for the reified connector is a crucial step. This chapter focuses on three main aspects: assessing the completeness of the constructed metamodel designed for interoperability connectors regarding coverage within the connector repository, affirming its extensibility capabilities, and subjecting the reified connector to a load test for validation.

The objectives of the evaluation are threefold. According to the connector repository, the proposed metamodel must include specifications for every potential connector to enable messaging interactions. It is also indispensable to ensure that the metamodel can integrate new connectors and is extensible. Finally, it is crucial to validate the proposed connector to ensure it can support critical scenarios from industry partners when many messages are exchanged in a short time interval. This scenario involves comparing the performance with interoperability mechanisms that do not rely on messaging style with another mechanism.

4.1 Assessing the Scope of Connector Metamodel Coverage

In the previous Chapter 3, we presented a reified metamodel for messaging connectors. This metamodel is initially derived from the enterprise integration patterns (EIP) [HW04], complemented by specific interoperability solutions, including vendor-specific, vendor-neutral, and industrial-specific approaches.

It has become increasingly important to explore additional sources of EIP, first introduced in 2003 to address integration challenges in enterprise scenarios. With the emergence of new practices and challenges such as IoT, ubiquitous environments, and cloud computing, it is necessary to update interoperability requirements continuously. The development of a new version of EIP has been ongoing since 2006 [Hoh06], with the latest update released in January 2017 [1]. This underlines the importance of continuously updating interoperability specifications.

[1] urlhttps://www.enterpriseintegrationpatterns.com/patterns/conversation/

To validate the current version of the metamodel, a two-step procedure is used. The first step involves examining the connector repository created in Chapter 3, as shown in Figure 3.3. This examination ensures that the process used during the repository construction guarantees the diversity and representativeness of entities within the metamodel.

Next, it is necessary to evaluate each subset of connectors based on the established metamodel. This evaluation closely examines their compliance with specific criteria to ensure alignment.

4.1.1 Validation of the connector repository building process

To create the messaging connector repository, we queried search engines, including Google Search, Github, Gitlab, and SourceForge. We utilized Google to obtain implementation examples of interoperability patterns represented in the metamodel available on individual blogs or vendor websites such as SAP [2], Microsoft [3], and IBM [4] for pedagogical purposes or to showcase their expertise. Google has also facilitated access to additional implementation examples on blog sites or tutorials, such as Baeldung [5], which provide concrete implementation examples. Research on Google complements other search systems we use. For example, a demonstration of an implementation available on GitHub may also be located on the RabbitMQ [6] site.

Conversely, GitHub provides users with the ability to discover open-source solutions developed by various contributors, including Confluent.io, a provider of data streaming solutions based on Apache Kafka technology, Apache Camel, an open project offering implementations of numerous enterprise integration models in Java or DSL, and Spring.io, which features several projects aiding in software integration, such as Spring Cloud Stream, Stream Integration, and Spring Cloud Data Flow. GitLab is predominantly used to access proprietary industrial solutions from partners willing to share their source code. SourceForge is a centralized platform for managing open-source software projects and identifying keywords for further research on vendor sites like MuleSoft and Boomi.

[2] https://learning.sap.com/learning-journeys/developing-with-sap-integration-suite/using-integration-patterns_fdd8f683-da3d-4abe-a29d-a6f6fd06cc14

[3] https://learn.microsoft.com/en-us/azure/architecture/patterns/competing-consumers

[4] https://www.ibm.com/docs/en/integration-bus/10.0?topic=solutions-developing-integration-by-using-patterns

[5] https://www.baeldung.com/

[6] https://rabbitmq.com/

In addition to online research, we actively seek specific examples of implementing the use cases proposed in the EIP book in physical and digital formats. Exploring the websites of messaging technologies has enabled us to uncover implementation examples related to the technologies under consideration, such as RabbitMQ, Apache Kafka, ActiveMQ, and Java Messaging System (JMS) and its broker version, OpenMQ. A reference to the specific website is pending.

A search is conducted for use cases involving the entities listed in tables 3.1, 3.2, 3.2, 3.4, 3.5, and 3.6 on each system, including Google Search, Github, Gitlab, and SourceForge. The tables contain entity names based on the first column of each table. The selection criteria for the connector repository include clear flow diagrams and source code, the ability to compile and run the code, an understandable specification, and a minimum threshold of code quality. If multiple projects meet these criteria, additional factors are considered in the final selection, with expert visual analysis playing a crucial role in the decision-making process.

Figure 4.1 illustrates the process used to build the connector repository.

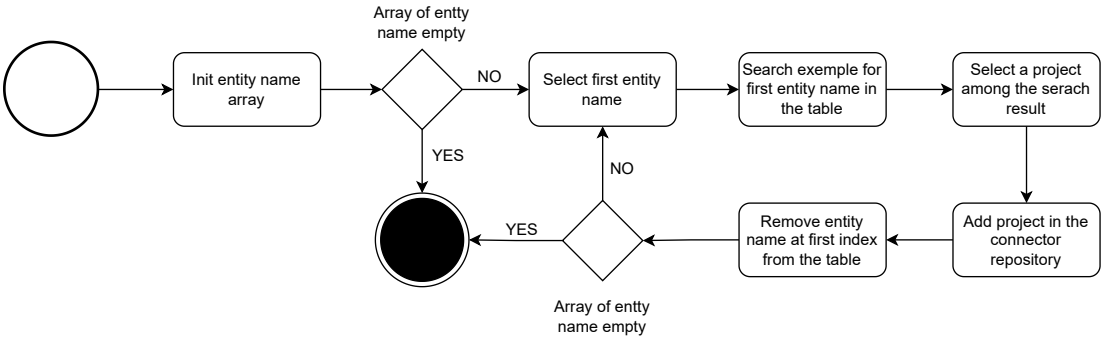


Figure 4.1: Connector repository creation process for ensuring the representativeness of entities in the reified metamodel

Each query should return an instance of a project containing at least the model corresponding to the metamodel entity under investigation. If the example contains multiple patterns, it is preferable to include them. The primary goal of any query is to identify the most appropriate example for the current pattern search. While including other patterns is not mandatory, their presence may be influential in cases where we must choose among several suitable results. Conversely, selecting multiple examples for a single pattern is possible, especially if they are all relevant. Similarly, a single example may be selected for multiple patterns.

After identifying the project's example that involves the most suitable connectors for a specific model, they are added to the connector repository, and the studied models are removed from the list. However, it is impractical to collect all the connectors in the repository due to time constraints and limited access to them worldwide. Therefore, ensuring complete coverage is challenging. It is crucial to have a diverse range of project patterns. Using a smaller set of varied connectors is more valuable than having many similar ones.

The process aims to create a collection of projects that include connectors covering all patterns. The projects collected through the process outlined in Figure 4.1 are presented in several tables for easy comprehension. Specifically, tables 4.1, 4.2, 4.3, 4.5, 4.4, and 4.6 contain projects resulting from research that considers the list of patterns corresponding to entities such as endpoint type, channel type, router type, message type, transformer type, and system management type.

System management type patterns may be part of one of the possible pattern types but are specific to system management needs. For instance, *Control Bus* is categorized under the channel type but is dedicated to transmitting data relevant to managing components involved in the message flow, such as the monitoring system.s

The tables are read from left to right and contain seven columns. The first column is the *Request Identifier* corresponding to a number that increments with each search for a new pattern based on the process outlined in Figure 4.1. The second column presents the patterns for which we selected the search result. The third column displays the origin of each example, which may be a website, book, or blog. The fourth column specifies the type of resource utilized, such as source code, data flow schemes, or specification documentation. The fifth column indicates the number of languages in which the use case is implemented, referring to cases where an end-to-end example is available in multiple languages rather than a mixture of messages. The sixth column lists the messaging systems used, including Apache Kafka, RabbitMQ, ActiveMQ, JMS, and the OpenMQ platform. The presented information includes all relevant patterns covered by the example. If a use case fits multiple searched patterns, both cases are considered and noted in a single row if the patterns are in the same category. The first column, *Request Identifier*, lists all researched patterns that produced the result, separated by commas. The second column will contain a list of searched patterns separated by commas in the same order to allow identification of which request corresponds to which research pattern. The line is duplicated in the second case, where the searched patterns do not belong to the same category.

The algorithm for selecting the relevant use case, presented in Algorithm 1, will consider this duplication.

Table 4.1 presents the results of the case research examples for the endpoint patterns described in Tables 3.3 and 3.2. The table is well-organized and provides detailed data, indicating appropriate endpoint examples, each accompanied by accessible source code. Two sets of requests share the same optimal example: requests 1 and 5 and requests 2 and 6. Although not always applicable, a Requester usually aligns with a Replier, while an Event consumer corresponds with an Event producer.

Request Id	Searched patterns	Use case origin	Artefact type	Languages	Messaging technology	Includes patterns
1, 5	Requester, Replier	EIP book	Source code	Java, C#	JMS	Requestor, Replier, P-to-P Channel Invalid message, Correlation Id, Return Address, Document Message.
2, 6	Event producer, Event consumer	Partner GitLab	Source code	Java	RabbitMQ	Event producer, Event consumer, Publish-Subscribe Channel.
3	Publisher confirms	RabbitMQ official site	Source code	Java, C#, PHP	RabbitMQ	Publisher confirms, Event consumer, Publish-Subscribe Channel.
4	Transactional client (Producer)	Baeldung	Source code	Java	Apache Kafka	Transactional producer, Transactional consumer
7	Competing consumers	Microsoft Learn	Source code	C#	Azure Service Bus	Event producer, Competing consumer, Publish-Subscribe Channel
8	Selective consumer	Partner GitLab	Source code	Java	RabbitMQ	Event producer, Selective consumer, Publish-Subscribe Channel.
9	Polling consumer	RedHat documentation	Source code	Java, Apache Camel DSL	JMS	Event producer, Publish-Subscribe Channel, Polling consumer.
10	Durable subscriber	Novell documentation	Source code	Java	JMS	Event producer, Durable subscriber.
11	Idempotent receiver	GitHub	Source code	C#	Apache Kafka	Event producer, Publish-Subscribe Channel, Idempotent receiver.
12	Message dispatcher	RedHat documentation	Source code	Apache Camel DSL	ActiveMQ	Dispatcher, Publish-Subscribe Channel.
13	Service Activator	GitHb	Source code	Java, Spring integration	RabbitMQ	Service activator, Event Consumer, Event Producer, P-to-P channel

Table 4.1: Table of results of use case search on endpoint patterns

Table 4.2 presents the research findings for *Message channel* patterns. Like the endpoint patterns, the channel research patterns do not contain any empty

data, and there is at least one example of a use case with available source code. Three requests share the same best results: 14 and 16 and 15 and 21. Additionally, requests 14 and 16 align with requests 1 and 5 in Table 4.1. Likewise, request 19 corresponds to requests 2 and 6 in Table 4.1.

Request Id	Searched patterns	Use case origin	Artefact type	Languages	Messaging technology	Included patterns
14, 16 (same 1, 5)	Point-to-Point channel, Invalid Message	EIP book	Source code	Java, C#	JMS	Requestor, Replier, P-to-P Channel Invalid message, Correlation Id, Return Address, Document Message.
15, 21	DataType channel, Message Bridge	Baeldung	Source code	Java, Spring integration	RabbitMQ	Point-to-Point channel, file sender (Publisher), file handler (Consumer), Document Message, Message bridge, service activator, Channel Adapter, DataType channel, Publish-Subscribe channel.
17	Dead letter	RabbitMQ official site	Source code	Java	RabbitMQ	Dead Letter, Event producer, Publish-Subscribe Channel
18	Guaranteed Delivery	Medium	Source code	Spring Integration	RabbitMQ	Guaranteed Delivery, Message store, Event producer, event consumer
19 (same 2, 6)	Publish-Subscribe Channel	RabbitMQ official site	Source code	List 1	RabbitMQ	Event producer, Event consumer, Publish-Subscribe Channel.
20	Channel adapter	Spring official site	Source code	Java, Spring integration	RabbitMQ	Channel adaptor, service activator, Point-to-Point channel

Table 4.2: Table of results of use case search on channel patterns

Tables 4.3 and 4.4 present research findings on the use cases for router and transformer patterns. Each example is unique, and all entries contain relevant data without any gaps. Comprehensive source codes are also available for each exemplar.

Table 4.5 presents the results of the use case research for message constructs. Requests 35, 36, and 37 share the same best example. Furthermore, some requests share the best results with patterns from other tables, such as request 34 with requests 2 and 6 in Table 4.1, and request 19 in Table 4.2. Furthermore, request 39 corresponds to the best result in Table 4.4. In contrast to the previous tables, Table 4.5 presents patterns with source code implementation examples. Therefore, there is only one request with empty data.

Table 4.6 presents the results for the request on system management patterns. Although system management includes transversal patterns that could be represented in different tables, we chose to search for the use case for its patterns and classified them in a separate table. We encountered missing data regarding

Request Id	Searched patterns	Use case origin	Artefact type	Languages	Messaging technology	Included patterns
22	Content-Based router	Java In Use	Source code	Java, Apache camel DSL	JMS	Content-Based Router, Datatype channel, Splitter, Document message.
23	Dynamic router	Java In Use	Source code	Java, Apache camel DSL	JMS	Content-Based Router, Dynamic router, Datatype channel, Splitter, Document message.
24	Message Filter	GitHub	Source code	Java	RabbitMQ	Event Message, Message filter, P-to-P channel, Publish-Subscribe channel, Splitter, Document message, Event producer, Event consumer
25	Splitter	Java In Use	Source code	Java, Apache camel DSL	JMS	Splitter, P-to-P channel
26	Aggregator	EIP book	Source code	Python	Amazone SQS	Aggregator, Publish-Subscribe channel, P-to-P channel, return address
27	Recipient List	Spring official site	Source code	Java, Spring integration	JMS	Splitter, P-to-P channel, Recipient List
28	Resequencer	Huihoo documentation	Source code	Java	JMS	Point-to-Point channel, Resequencer

Table 4.3: Table of results of use case search on router patterns

Request Id	Searched patterns	Use case origin	Artefact type	Languages	Messaging technology	Included patterns
29	Translator	Java In Use	Source code	Java	ActiveMQ	Translator, Datatype channel, Document message, format indicator.
30	Envelope Wrapper	Codetrics	Source code	C++	Google Protocol Buffers	Event Producer, Document message, P-to-P channel, Envelope Wrapper.
31	Content Enricher	GitHub	Source code	Python	Amazone Bridge pipe	P-to-P channel, Event Producer, event consumer, Content Enricher
32	Content filter	Apache camel page	Source code	Java, Apache camel DSL	JMS	Aggregator, P-to-P channel, Content filter

Table 4.4: Table of results of use case search on transformer patterns

the search for message construct patterns, with two examples lacking implemented source code.

The table above provides insights from a thorough connector analysis based on expert knowledge. We use a well-defined process, illustrated in Figure 4.1, to build the connector repository. This process ensures comprehensive coverage of

Request Id	Searched patterns	Use case origin	Artefact type	Languages	Messaging technology	Included patterns
33	Document Message	Java In Use	Source code	Java, Apache camel DSL	JMS	Content-Based Router, Datatype channel, Splitter, Document message.
34 (same 2, 6, 19)	Event message	RabbitMQ official site	Source code	List 1	RabbitMQ	Event producer, Event consumer, Publish-Subscribe Channel.
35, 36, 37	Command Message, Correlation id, return address	Rabbit MQ official site	Source code	List 2	RabbitMQ	Requestor, Replier, P-to-P channel, Correlation id, return address, Command Message
38	Message Expiration	Rabbit MQ official site	Source code	Java	RabbitMQ	Event Message, Publish-Subscribe channel, Event producer, Message expiration
39 (same 29)	Format indicator	Java In Use	Source code	Java	ActiveMQ	Translator, Datatype channel, Document message, format indicator.
40	Message Sequence	EIP book	Documentation	NC	NC	Message sequence, Point-to-Point channel

Table 4.5: Table of results of use case search on message constructs patterns

Request Id	Searched patterns	Use case origin	Artefact type	Languages	Messaging technology	Included patterns
41	Control bus channel	Spring official site	Source code	Java, Spring integration	RabbitMQ	Control bus channel, P-to-P channel
42	Wire Tap	GitHub	Source code	Java, Apache camel DSL	JMS	Wire Tap, Event producer, Event message, P-to-P channel, Control bus channel
43	Message History	EIP book	documentation	NC	NC	NC
44	Message store	Spring official site	Source code	Java, Spring integration	RabbitMQ	SMessage store, P-to-P channel, Service activator
45	Channel purger	EIP book	documentation	NC	NC	P-to-P channel, Channel Purger
46	Test message	EIP book	Source code	C#	MSMQ	Test message, Content enricher, Recipient list, Aggregator, Translator, Return Address, Format indicator

Table 4.6: Table of results of use case search on system management patterns

all Enterprise Integration Patterns (EIPs). A persistent repository containing the

connector corpus can be accessed at this location[7]. Using the information in the table, we can calculate the probability percentage of the presence of each pattern.

Validation of the metamodel using the established corpus requires demonstrating that all connector models conform to the metamodel. However, due to the large number of use cases, validation will be performed selectively in specific scenarios. The challenge is to identify applicable use cases and quantify their number. To address this challenge, we present an algorithm to evaluate all patterns.

Algorithm for Use Cases Selection Algorithm 1 constitutes the list of use cases for validation. This allows us to have a reduced list of use cases that covers all the patterns at least once.

Algorithm 1 Algorithm for building pertinent use case list for validation

$listAddedUseCase \leftarrow \emptyset$
$listAddedPatterns \leftarrow \emptyset$
$listUseCase \leftarrow useCases$
$listUseCaseSize \leftarrow length(listUseCase)$
while $listUseCaseSize \neq 0$ **do**
  $useCase \leftarrow getFirst(listAddedUseCase)$
  $listCurrentPatterns \leftarrow getPatterns(useCase)$
  if $(listAddedPatterns \cap listCurrentPatterns)$ is $\emptyset$ **then**
    $listAddedPatterns \leftarrow concat(listAddedUseCase, listCurrentPatterns)$
    $listAddedUseCase \leftarrow concat(listAddedUseCase, useCase)$
  else($listCurrentPatterns$ not $\subset listAddedPatterns$) is $true$
    $intersection \leftarrow listAddedPatterns \cap listCurrentPatterns$
    $patternsForAdd \leftarrow listCurrentPatterns$ - $intersection$
    $listAddedPatterns \leftarrow concat(listAddedUseCase, patternsForAdd)$
    $listAddedUseCase \leftarrow concat(listAddedUseCase, useCase)$
  end if
  $listUseCase \leftarrow removeFirst(listUseCase)$
end while

The algorithm will initially eliminate pattern requests that contain missing data. Locating a valuable implementation or specification example that includes the patterns is unnecessary. We will remove duplicate examples across table categories and within the same categories. Following the use case research process

[7] `https://zenodo.org/`

from the patterns list corresponding to the metamodel entities names in Figure 4.1, we have 33 out of 46 examples to consider. Ten are duplicate use cases, and three requests contain empty data. A duplicated example refers to the same use case selected for search from two or more patterns. The remaining use cases are then sorted based on the number of patterns covered. At this stage, we have only non-duplicated use cases classified by pattern size in descending order. Pattern size refers to the number of distinct patterns that appear in the use case being considered. In cases of equality, the order is determined by the chronological number of requests through the 'Request Id', with older requests taking priority. The list of use cases for validation is initialized, and the list of added patterns is currently empty. The first use case that contains the most different patterns or has the smallest request number in case of equality is considered as the element of the shortened list. The iteration on the list of use cases continues until it is empty. The intersection between each use case and the list of added patterns is calculated for each use case.

- If the intersection is equal to the empty set, no patterns of the use case are in the list of added. So, we add all the use case patterns to the list of added patterns. This scenario happens at the first iteration when the list of added patterns is initialized to empty. So, the use case is added to the list of retained use cases for validation.

- If the intersection is not an empty set, some patterns of the use case are already in the list of added patterns. So, we need to evaluate two cases.

  - If the list of patterns of the use case is included in the list of added patterns, i.e., the intersection between the two lists is equal to the list of the use case patterns, We do nothing. The use case is not added to the list of retained use cases in this case.
  - Otherwise, we add the list of patterns formed by the list of use case patterns minus the intersection to the list of added patterns. Note that two extreme cases are included in the scenario. Note that the case where the list of added patterns is included in the list of the patterns of the use case is covered by this scenario. So, the use case is added to the list of remaining patterns for the use case.

  After each iteration, the current use case is removed from the list of use cases. We stop when the list is empty.

Figure 4.2shows a curve that represents the incremental addition of new patterns by iteration across 33 use cases that were sourced from the connector repository, following Algorithm 1. On the x-axis, you can see the identifier of each use case, while on the y-axis, you can see the corresponding number of added patterns.

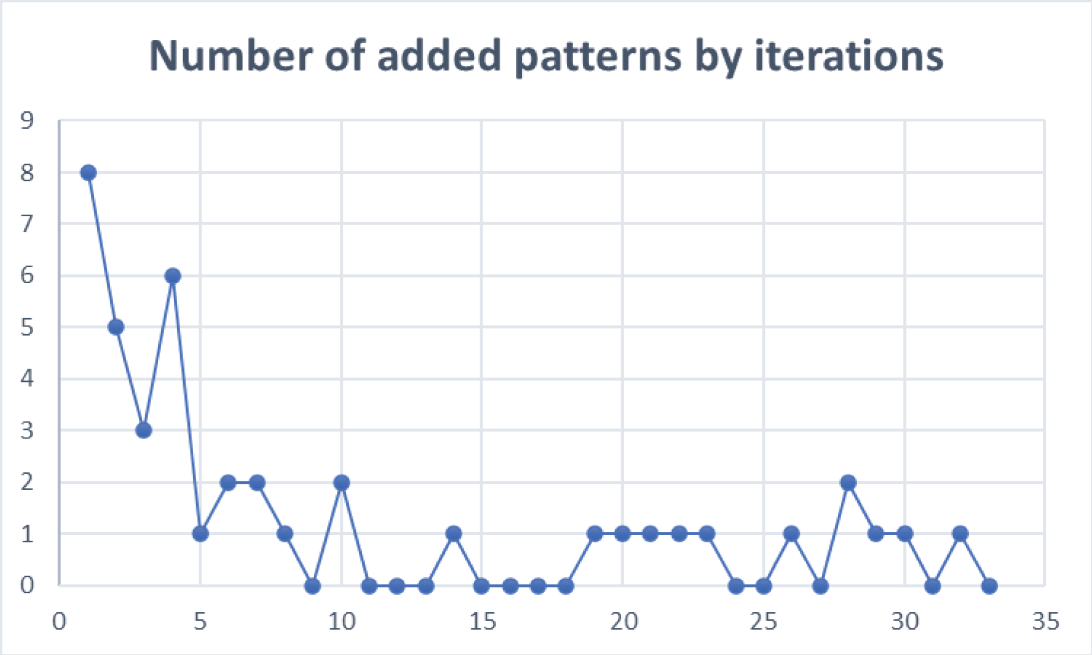


Figure 4.2: Number of new patterns added when transitioning from one use case to another, considering the 33 use cases from the connector repository.

Upon analyzing Figure 4.2, we generated a list of relevant use cases for appropriate validation. The resulting list is provided below. Notably, certain iterations involve the elimination of entire use cases without the addition of new patterns. This implies that all patterns associated with these use cases are already covered. Consequently, this reduction in use cases streamlines the considerations for validating the metamodel's coverage.

4.1.2 Comparison of Compliance with the metamodel through illustrative examples

The previous proposes an algorithm from this corpus to choose the relevant use cases concerning the pattern coverages.

Figure 4.3 shows a list of retained use cases based on their relevance, similar to the curve in Figure 4.2. If a use case is already covered, it is not included in this list. On the x-axis, you can see the identifier of each use case, while on the y-axis, you can see the corresponding number of added patterns.

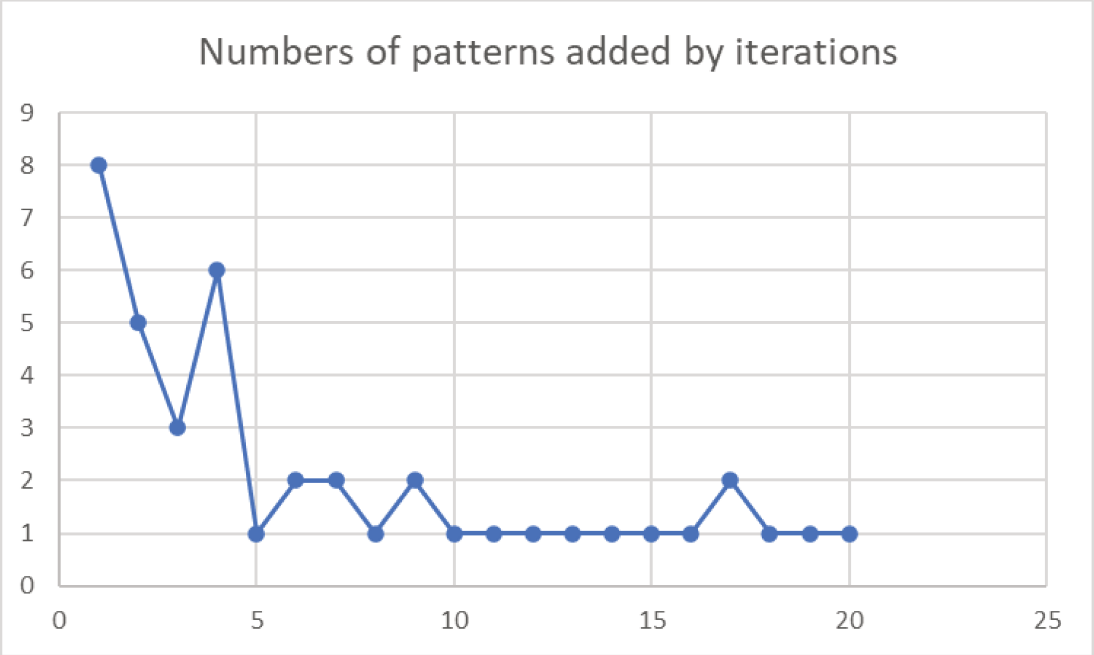


Figure 4.3: Number of added patterns when transitioning between use cases, taking into account only those from the connector repository that have not been covered in the current iteration.

Out of the 33 use cases initially considered, only 20 were selected for thorough examination. To ensure a comprehensive validation of the metamodel, a comparison between the models of these 20 pertinent connectors, covering all integration patterns, is necessary. Each pattern's compliance requires validation at least once. Analyzing the conformity of 20 patterns to our metamodel can be challenging.

Figure 4.3 shows that testing each remaining use case may not be practical. Notably, most patterns appear in the first four iterations, which represent 20% of the entire process. This corresponds to four examples, encompassing 23 out of 33 patterns. After analyzing the first four use cases, we observed that the average number of patterns per use case is generally one.

Based on this observation, we have chosen to validate compliance for only these four examples. This strategic decision allows us to test 70% (23/33) of our relevant use cases instead of attempting to validate all 21 cases. It is important to note that the selected four examples cover 50% (23/46) of the patterns within the metamodel.

Table 4.7 enumerates the four chosen use cases sourced from Tables 4.2, 4.1, 4.3, and 4.6.

Use case Identifier	Use case name
15	MPEG video file transfer
1	JMS Request-Reply
24	The loan broker
46	Order processing

Table 4.7: The four use cases retained for metamodel compliance validation.

It is important to note that all the selected examples belong to different table categories, and this distribution is coincidental, not planned. For each use case, we provide a description, retrieve the UML object diagram through reverse engineering from the source code, and subsequently compare the resulting model with the established metamodel.

For the validation, we will rely on the object model of each connector. Once we have the object models, we must validate that they conform to the designed metamodel. A UML object diagram represents a specific instance of the metamodel. Then, we have concretely the name of the object and the number of instances of each object that compose the connector. The connector object model is an instance of the connector metamodel. For that, we analyzed different objects that the connector has. This information is available by specified constraints in the metamodel. This consists of which required entity is present or not. If an entity is present, does the number respect the minimum and maximum required number? If an element is present, is it similar to one list element? The validation process is manual, and each entity is analyzed manually.

Explanation for table understanding: The table explanations help to understand the comparison result tables that will be presented for each use case. The first line indicates the entity being verified, while the second line lists the constraints the connector must adhere to according to the connector metamodel. The third line shows the status of the object model, including the number of instances corresponding to metamodel entities.

Please refer to the following explanations to clarify the constraints outlined in the second line. The abbreviations NbE, NbC, NbQ, NbT, and NbR refer to the specified numbers of *Endpoint*, *Channel*, *Queue*, *Transformer*, and *Router*, respectively. *Msg* corresponds to *Message*, while *MList* represents a collection of message types, including *Event*, *Document*, *Request*, *Query*, *Reply*, *Seqencer*, *Text*, and *Test*. The notation Ci:nQ indicates that channel C1 has n queues, with C1:1Q indicating one queue for channel C1.

Validation of Use case 1 - MPEG video file transfer: In this section, we scrutinize the validation process for the first use case, focusing on the MPEG video file transfer. The use case involves transferring an MPEG video file from a specified folder to another configured folder using the Spring Integration Framework.

Figure 4.4 visually depicts the message flow for this use case and involves constituents.

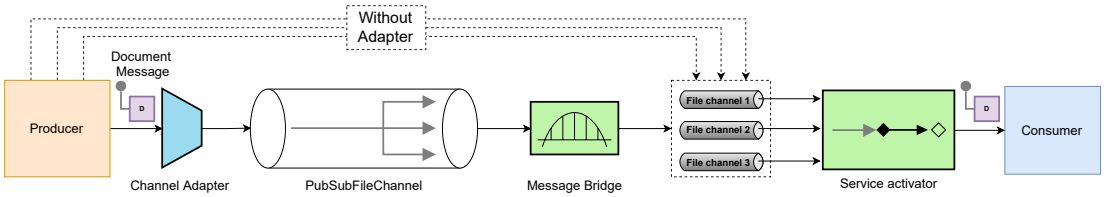


Figure 4.4: Overview of the message flow for the first use case

Message is the data transfer unit; it consists of a header, a container that can be used to transmit metadata, and a body, payload, which is the actual data that is of value to be transferred; in our use-case, the video file. *Channel* is the pipe by which messages are relayed from one system to another. We have three separate channels, all identified by their respective names.

Channel adapter Publish-Subscribe (Pub-Sub) channels establish a one-to-many communication line between systems or components. This will allow us to publish to all three direct channels we created earlier. Then, we want to replace the P2P channel with publish-subscribe to relay the message to the destination.

A channel adapter is used for that. We have now converted the inbound channel adapter to publish to a Pub-Sub channel instead of three separate channels.

Message Bridge The channel adapter allows us to send the files being read from the source folder to multiple destinations. However, to connect two message channels or adapters if they cannot connect directly. We create a bridge from the pubSubFileChannel to fileChannel1, fileChannel2, and fileChannel3 so that messages from pubSubFileChannel can be fed to all three channels simultaneously. Bridge replicate messages from the Publish-Subscribe channel to the Point-to-Point Channel.

Service activator Once the messages are available in channels, the service activator will activate the message that must consume the messages.

From the implemented connector that consists of our first use case, we create the object model of the connector, an instance of the connector metamodel. So, we need to validate that the connector object model respects the connector metamodel. Figure 4.5 shows the object model of the implemented connector.

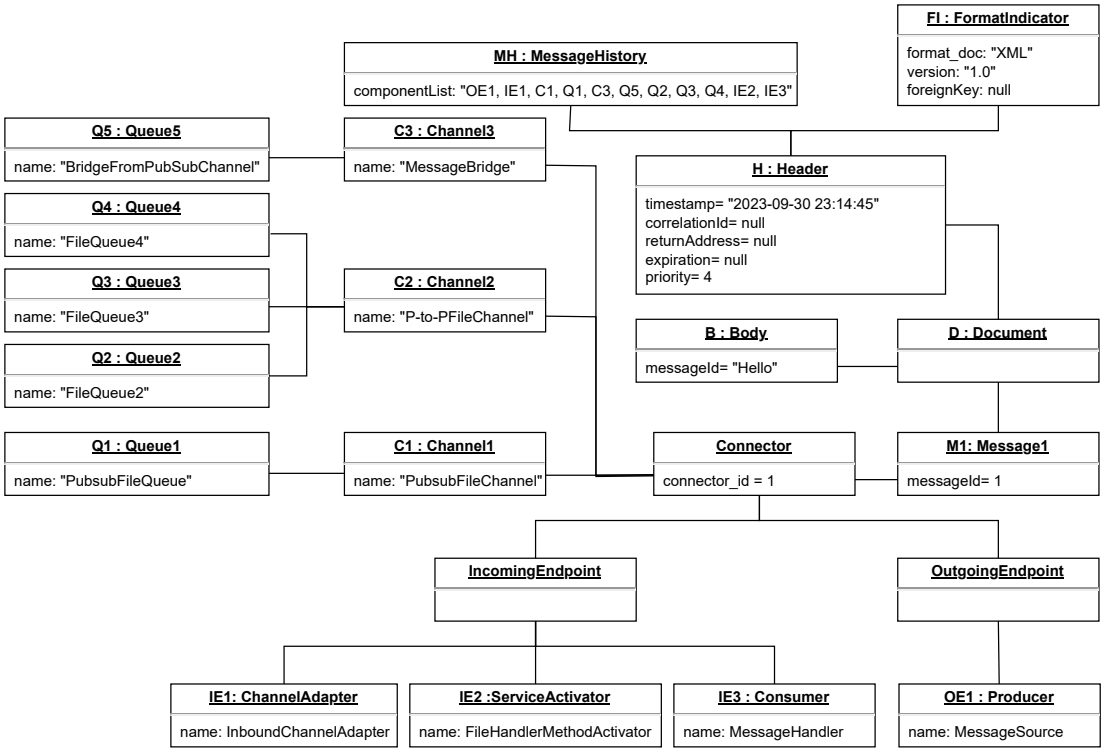


Figure 4.5: Object model of the connector for the first use case

Table 4.8 summarizes the metamodel compliance results for the object model of the first use case and the connector metamodel.

	Endpoint	Channel	Queue	Transformer	Router	Message construct
Metamodel requirement	$NbE \geq 2$	$NbC \geq 1$	$\forall C \exists NbQ / NbQ \geq 1$	$NbT \geq 0$	$NbR \geq 0$	$Msg \in MList$
Object model state	4	2	C1:1Q; C2:3Q	0	0	M1:D

Table 4.8: Metamodel compliance validation for the first use case

The validation results confirm that all connectors adhere to the metamodel requirements. Specifically, the connector metamodel requires a minimum of two message endpoints, and the object model for the first use case exhibits four endpoints. There are two channels, C1 and C2, which align with the metamodel requirement of at least one channel. Channel C1 has one queue, while channel C2 has three queues. Both instances meet the constraints as the metamodel specifies that each channel should have at least one queue ($NbQ \geq 1$).

The metamodel allows for the presence or absence of routers and message transformers; however, the current use case does not include either. The metamodel indicates that the connector manages a specific message list, and the first use case utilizes a *Document* message, which falls within the defined message list.

Considering these observations, we validate that the connector in the first use case conforms to the connector metamodel.

Use Case 2 - JMS Request-Reply: The second use case involves utilizing the Java Messaging System (JMS) to facilitate Request-Reply communication patterns. This example demonstrates the implementation of Request-Reply, where a *Requester* constituent sends a request to a *Replier* constituent. The *Replier* constituent receives the request and returns a reply to the requester. The example demonstrates how an invalid message will be rerouted to a specific channel. Figure 4.6 illustrates the exchange flow that is implemented.

This example includes the main components: the *Requester* and *Replier*, a *Point-to-Point* channel, and an *Invalid message* handling mechanism. The *Requester* acts as a message endpoint responsible for sending a request message and waiting for a corresponding reply message.

The *Replier* is a message endpoint that receives a request message and responds with a reply message. The *Requester* and the *Replier* operate in separate Java virtual machines, facilitating distributed communication. This example assumes that the messaging system has three defined queues: *jms/RequestQueue*,

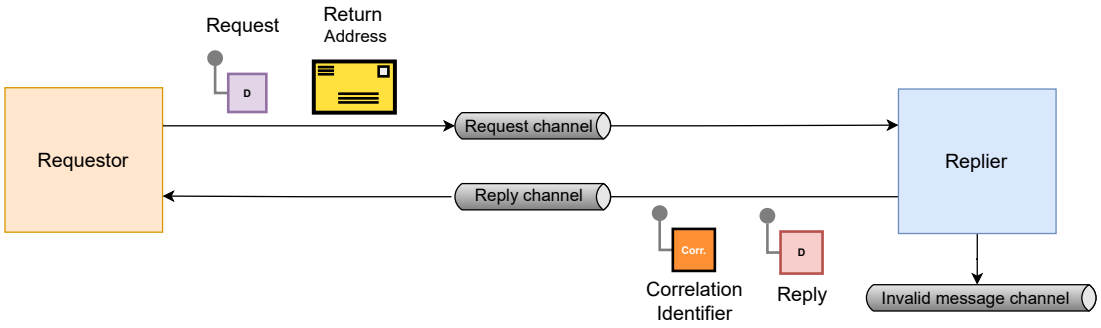


Figure 4.6: Overview of the message flow for the second use case

the queue used by the *Requester* to send the request message to the Replier. The *jms/ReplyQueue* is the queue used by the *Replier* to send the reply message to the *Requester*. The *jms/InvalidMessages* destination channel contains queues where both the *Requester* and *Replier* redirect messages that they cannot interpret.

The sending message includes a specific return queue and provides a *Correlation identifier* to the *Replier*, enabling it to identify which sender requires the answer. This scenario involves a published asynchronous request-response pattern.

The object diagram model for the second use case is extracted and constructed from the specifications and implemented source code, as shown in Figure 4.7.

Table 4.9 summarizes the metamodel compliance results for the object model of the second use case and the connector metamodel.

	Endpoint	Channel	Queue	Transformer	Router	Message construct
Metamodel requirement	$NbE \geq 2$	$NbC \geq 1$	$\forall C \exists NbQ / NbQ \geq 1$	$NbT \geq 0$	$NbR \geq 0$	$Msg \in MList$
Object model state	2	1	C1:2Q	0	0	M1:D

Table 4.9: Metamodel compliance validation for the second use case

Table 4.9shows that the second use case connector meets the metamodel specified requirements. The metamodel requires a minimum of two message endpoints, and the object model for the second use case shows two endpoints. Additionally, one channel, C1, meets the requirement of the metamodel of at least one channel. Channel C1 has two queues: one for requests and one for replies, which

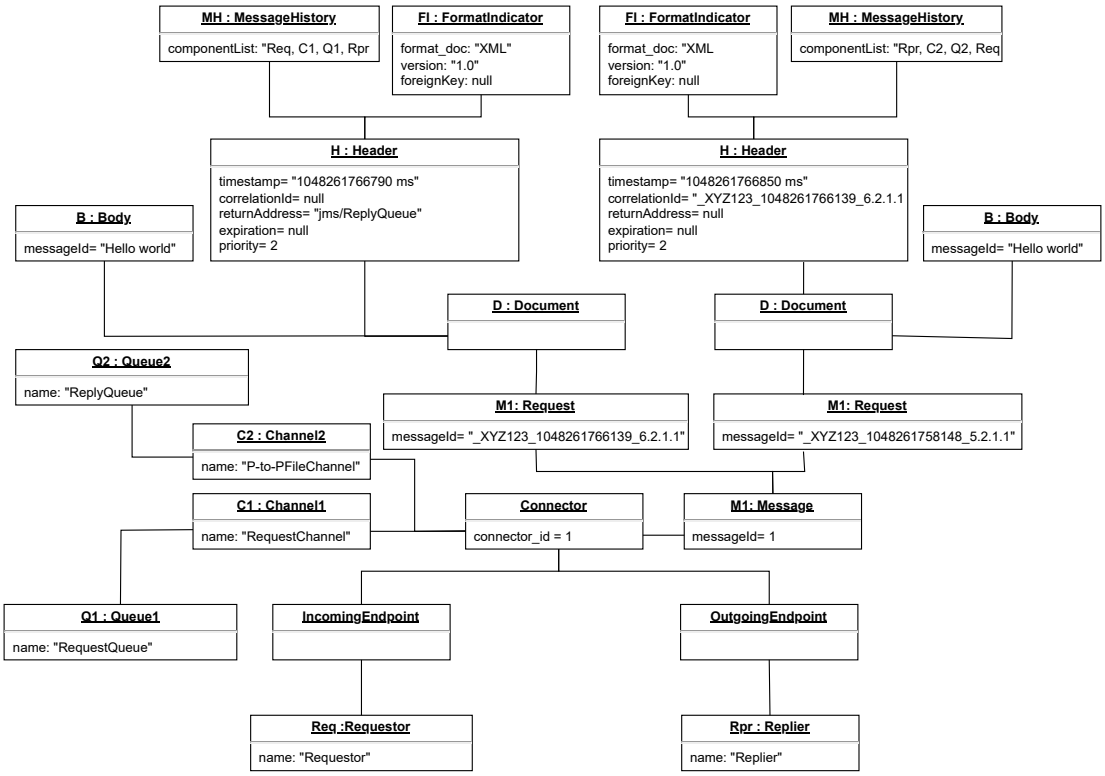


Figure 4.7: Object model of the connector for the second use case

satisfies the requirement imposed by the metamodel that each channel should have at least one queue ($NbQ \geq 1$).

As per the metamodel, the presence or absence of message routers and message transformers is allowed, and the second use case does not include either. The metamodel specifies that the connector manages a specific message list. A *Document* message is used in this use case, aligning with the defined message list.

Upon analyzing the table, we can confirm that the connector in the second use case adheres to the connector metamodel.

Use case 3 - Order processing: The third use case concerns the order processing example. In this example, the company management publishes price changes and promotions to large customers. An event producer is used to print out messages. Whenever the price for an item changes, we send a message notifying the customer. We do the same if we run a special promotion, *e.g.,* all widgets are 10% off in November. Some customers may be interested in receiving price updates

or promotions only related to specific items. If I purchase gadgets primarily, I may not be interested in whether widgets are on sale. The Message Filter That has only a single output channel is used. We have two consumers: a simple Event-Driven Consumer calls whenever a message is received. Figure 4.8 shows the exchange flow that is implemented.

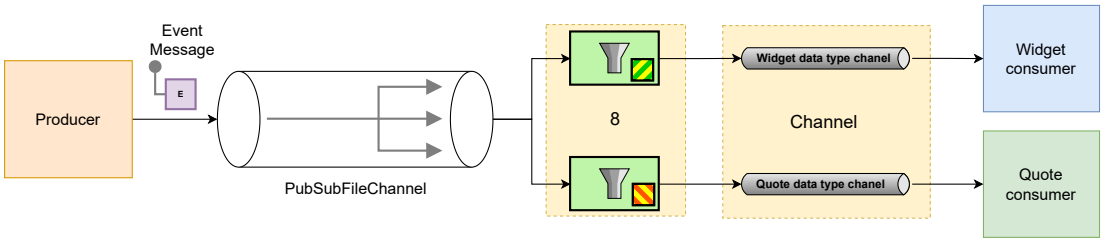


Figure 4.8: Overview of the message flow for the third use case 3

The object diagram model built from the third use case is presented in Figure 4.9.

Table 4.10 summarizes the metamodel compliance results for the object model of the third use case and the connector metamodel.

	Endpoint	Channel	Queue	Transformer	Router	Message construct
Metamodel requirement	$NbE \geq 2$	$NbC \geq 1$	$\forall C \exists NbQ / NbQ \geq 1$	$NbT \geq 0$	$NbR \geq 0$	$Msg \in MList$
Object model state	3	2	C1:1Q; C2:2Q	0	2Filter	M1:E

Table 4.10: Metamodel compliance validation for the first use case

The validation results confirm that all connectors comply with the metamodel requirements. The connector metamodel mandates a minimum of two message endpoints, and the object model for the third use case exhibits two endpoints. There are two channels, C1 and C2, aligning with the metamodel requirement of at least one channel. Channel C1 has one queue, while channel C2 has two queues, meeting the metamodel's assertion that each channel should have at least one queue ($NbQ \geq 1$). Both C1 and C2 meet the constraints.

The use case involves two message routers, more precisely two message filters (R1:2Filter), corresponding to the metamodel requirement that permits zero or more routers.

The metamodel allows for the presence or absence of message transformers, but the current use case does not require one. The metamodel specifies that the

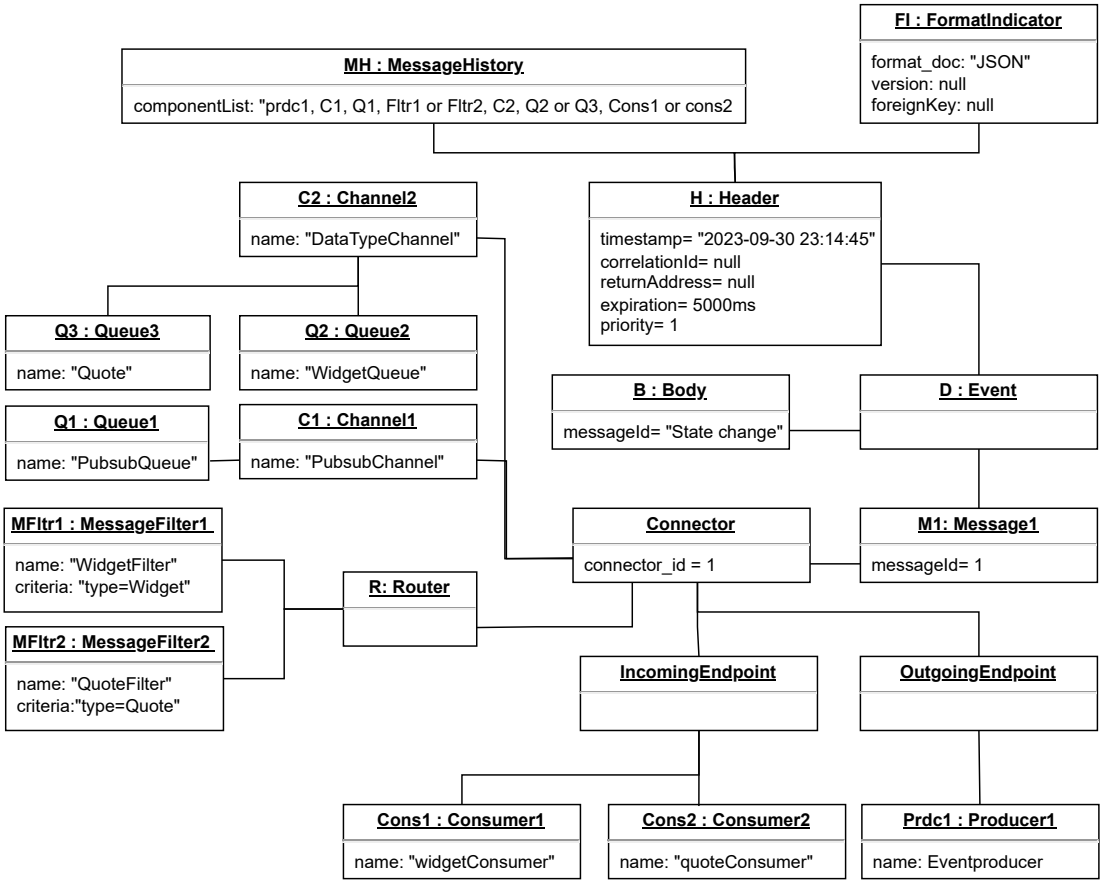


Figure 4.9: Object model of the connector for the first use case

connector manages a specific list of messages, and the first use case utilizes an *Event* message, which is included in the defined message list.

Based on this analysis, we assure that the connector in the first use case adheres to the connector metamodel.

Use case 4 - The loan broker: The fourth use case centers on testing the loan broker system example. This example incorporates routing and transformation patterns into a comprehensive solution, modeling the process of a consumer obtaining loan quotes from multiple banks. Figure 4.10 shows the implemented exchange flow.

The object diagram model from the fourth use case is showcased in Figure 4.11.

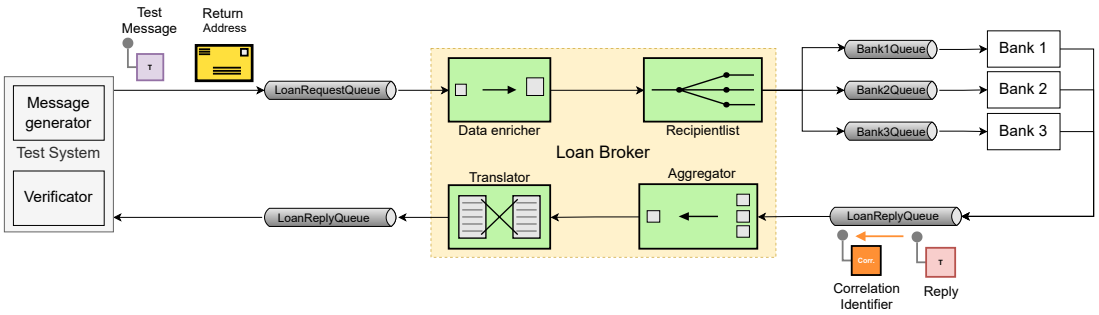


Figure 4.10: Overview of the message flow for the fourth use case

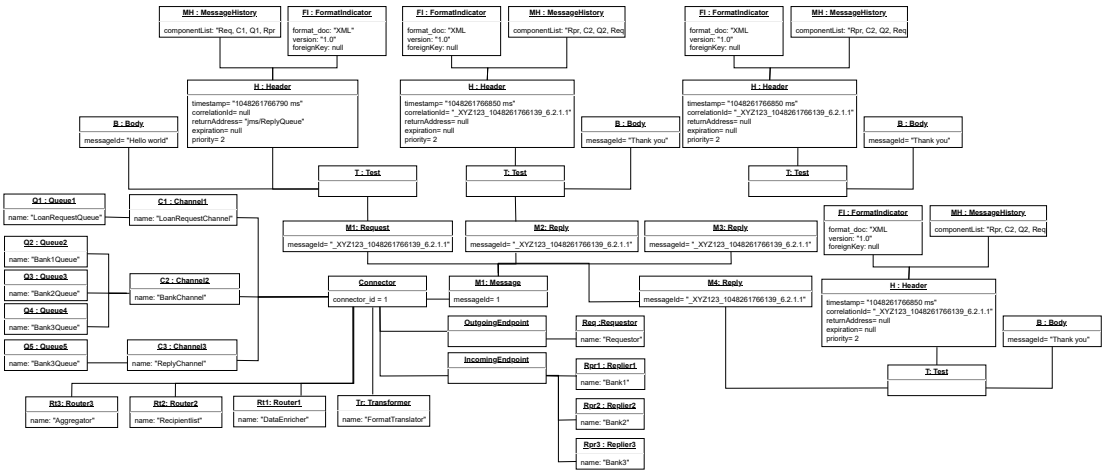


Figure 4.11: Overview of the object diagram for the fourth use case

Table 4.11 summarizes the metamodel compliance results for the object model of the first use case and the connector metamodel.

	Endpoint	Channel	Queue	Transformer	Router	Message construct
Metamodel requirement	$NbE \geq 2$	$NbC \geq 1$	$\forall C \exists NbQ/NbQ \geq 1$	$NbT \geq 0$	$NbR \geq 0$	$Msg \in MList$
Object model state	4	3	C1:1Q;C2:3Q;C3:1Q	T1:DE;T2:Trlt	R1:1RL;R3:1AG	M1:T

Table 4.11: Metamodel compliance validation for the fourth use case

The validation results confirm that all connectors meet the requirements of the metamodel. There are three channels, C1, C2, and C3, which align with the metamodel requirement of at least one channel. Channel C1 has a queue, channel

C2 has three queues, and channel C3 has one queue. Specifically, the connector metamodel requires a minimum of two message endpoints, and the object model for the first use case shows four endpoints. Channel C1 has a queue, channel C2 has three queues, and channel C3 has one queue. Channel C1 has a queue, channel C2 has three queues, and channel C3 has one queue. Each channel meets the constraints, as the metamodel specifies that each channel must have at least one queue ($NbQ \geq 1$).

The metamodel includes two message transformers, a Data Enricher and a Message Translator. Additionally, there are two message routers, namely a Recipient List and an Aggregator. Thus, the requirement for the message transformer and message router is met, as the metamodel allows for zero or multiple instances of these two elements.

The metamodel specifies that the connector manages a specific list of messages, and the fourth use case employs a *Test* message, which is appropriate for the defined message list.

Based on these observations, we confirm that the connector of the first use case is compliant with the connector metamodel.

Results of the Validation of Use Case Compliance with the Metamodel: After validating the four use cases against the connector metamodel, we can confirm that all considered use cases conform to the connector metamodel. Conversely, the connector metamodel covers entities present in all four use cases. Given that the four use cases were selected to cover the most prevalent interoperability scenarios in the repository, we can conclude that the connect metamodel encompasses all connectors present in the connector repository, and that each connector in the connector metamodel conforms to the connector metamodel.

4.1.3 Validation of Metamodel Expandability

To construct the connector metamodel, we use knowledge from Enterprise Integration Patterns, vendor solutions, and in-house industrial solutions. However, we acknowledge that companies may require patterns unknown among the identified Enterprise Integration Patterns. Our metamodel aims to accommodate any connector, whether vendor-specific, vendor-neutral, or an industrial solution. When the metamodel does not cover a specific connector, it is essential to integrate new constituents or properties into the existing metamodel. To accomplish this, we suggest a process starting with an initial metamodel.

The process takes the connector metamodel and the model of an implemented connector as input, requiring reverse engineering from the source code. The entities in the built connector model are classified and categorized into Endpoint, Channel, Router, Transformer, and Message constructs based on expert knowledge related to communication. Each category is assigned a class, and the existing entities in the metamodel are iterated over and compared with each class in the connector. When a metamodel entity corresponds to a connector class, we examine the properties and methods of the metamodel class. If all the properties and methods of the connector model are present in the matched metamodel entity, the two entities are considered identical, or the metamodel entity is deemed more comprehensive than the connector model entity. Conversely, if we find properties or methods in the connector entity but not in the connector metamodel, we integrate the relevant properties or methods. If the connector model does not match any of the metamodel entities in the categories, we add the new entity as a novel addition to the connector metamodel. This indicates the discovery of a new integration pattern.

Figure 4.12 illustrates the process that facilitates the creation of a more complete reified connector, taking into account known connectors and potential future connectors. The model must be extensible to accommodate future connectors.

4.2 Discussion and Conclusion

Although creating a metamodel that encompasses all connectors worldwide is a challenge, our approach offers inclusivity within the boundaries of a specified repository. The comprehensiveness of our metamodel is limited to the boundaries of the connector repository created. To achieve comprehensiveness, we would need a repository that includes all worldwide connectors. However, this is unachievable due to the difficulty of accessing industry-specific interoperability mechanisms implemented without collaboration or anticipating future mechanisms.

To address this limitation, we propose a transparent process to ensure the adaptability of our metamodel, allowing for the addition of new connectors to the repository. In terms of validating the metamodel's conformity with all possible connectors in the repository, compromises were necessary for several reasons. Having a repository containing all connectors is impossible, so claiming absolute coverage is not feasible. Validating conformity for over thirty object diagram classes proved challenging. As a compromise, we selected four examples encompassing various integration models, following a well-thought-out algorithm.

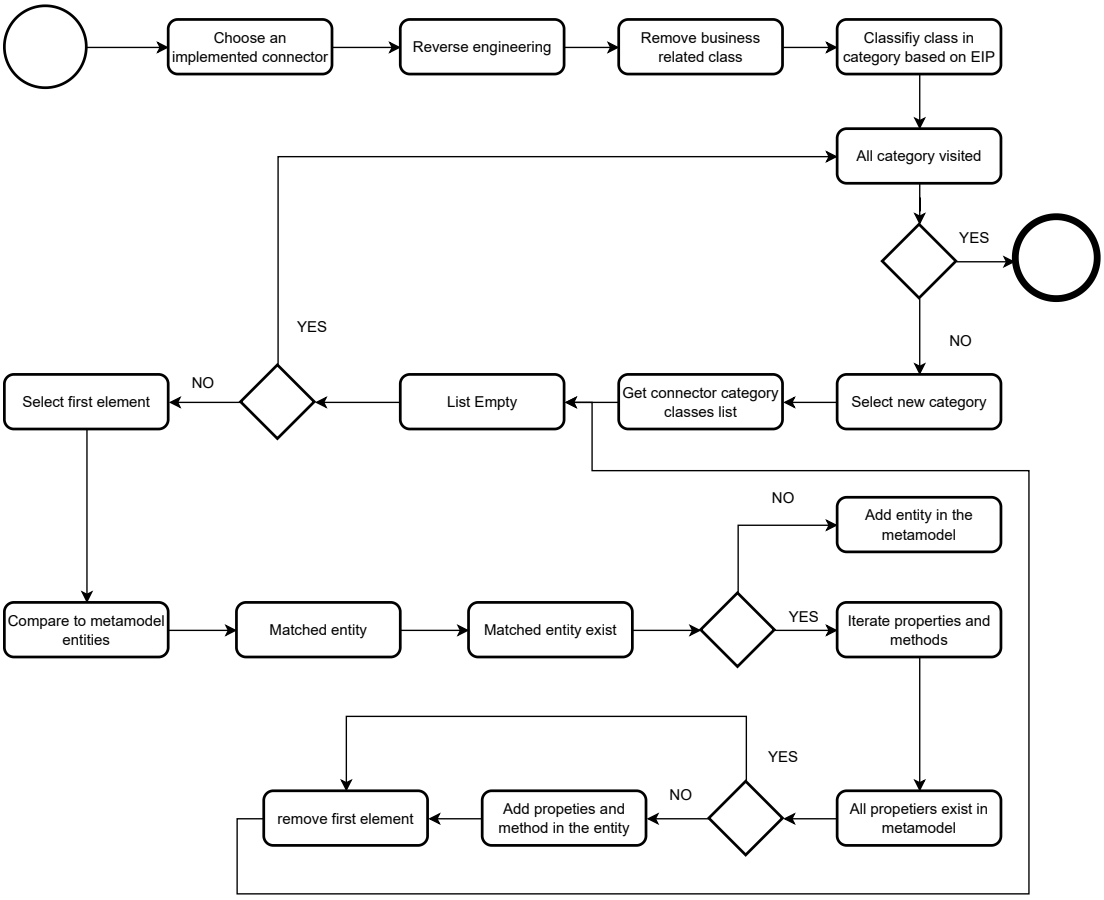


Figure 4.12: The Evolution of the Connector Metamodel: Integrating a New Project with Interoperability Mechanisms

At this stage, expertise in corporate integration is essential to identify integration patterns. Although automation can improve the process, an algorithm for pattern recognition would be particularly useful in managing numerous connectors, especially those written in different languages. Our analysis focused mainly on small-scale projects, which presents challenges for larger projects where a connector may not be a primary entity. Semantic matching during the scalability algorithm can be complex, especially when dealing with new connectors.

Chapter 5. *ConPL*: Unveiling the Connector Product Lines Framework

5.1 Introduction

In the last chapter, we created the *CoReif* metamodel, detailing the specifications for all potential interoperability connectors used in asynchronous communication, also known as messaging. This involved recognizing recurring patterns in various interoperability mechanisms across different solutions. The metamodel helps identify common patterns, guiding connector construction based on these. It adapts to specific communication needs and highlights connectors' common and specific characteristics. This chapter aims to fill the gaps in the literature using the *CoReif* metamodel. Two research questions guide this exploration:

1. How can interoperability patterns be effectively utilized to construct connectors that adapt to system changes, reducing development and maintenance costs?

2. What methods can generate essential connector source code beyond identifying architectural models?

Considering the insights from Chapter 3 and our research questions, this chapter proposes an approach for building interoperability connectors. This approach assumes the connector as an independent component, as the connector metamodel illustrates. To leverage connector commonalities and manage their variability effectively, the set of connectors is viewed as a Software Product Line (SPL) [CN02]. Utilizing Model-Driven Engineering (MDE) techniques [S+06], greater abstraction is achieved, approaching platform independence and enabling the generation of connector source code in targeted programming languages upon demand. This proposal introduces a framework facilitating the automatic configuration and generation of interoperability connectors based on reusable connector artifacts. The goal is to ensure connectors' flexibility, enabling their creation, maintenance, and evolution with minimal impact on business applications, ultimately reducing effort in developing, maintaining, and extending connectors.

5.2 Foundational Concepts

5.2.1 Software Product Line Engineering

A Software Product Line (SPL)[CN02] is a set of software products that share core functionalities and can be customized with additional features. The process used to develop these lines is called Software Product Line Engineering (SPLE)[PBVDL05], which involves two main stages: Domain Engineering (DE) and Application Engineering (AE). DE analyses and models functionalities defines reusable artifacts, and establishes a general SPL structure. AE tailors this structure to create specific, individualized products. For an overview of the SPL process steps, please refer to Figure 5.1.

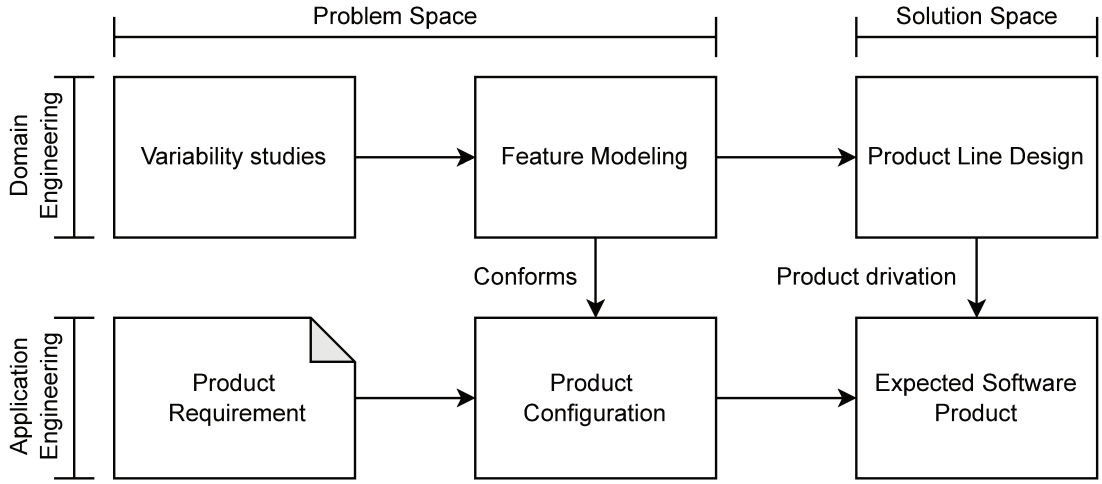


Figure 5.1: Overview of the Software Product Line Engineering process

Figure 5.1 illustrates the broader scope of software product line engineering, accommodating different methodologies and paradigms.

It explores the rationale behind adopting software product lines for interoperability connectors, introducing the Connector Product Line, *ConPL*, framework. This framework, designed for connectors using the DOP paradigm, concludes with a detailed use case requiring the implementation of the *ConPL* framework.

5.2.2 Differentiating Reuse Strategies: Comparative Analysis of Software Product Line (SPL), Component-Based Software Engineering (CBSE), and Software Ecosystem (SECO)

When considering strategies for software reuse, there are various options available, such as Software Product Lines (SPL), Component-Based Software Engineering (CBSE), and Software Ecosystems (SECO), each chosen based on specific needs. To clarify the distinct focuses of these approaches and reinforce the selection of SPL, it is crucial to review them comprehensively.

CBSE , defined by [Crn01], revolves around constructing software systems by integrating reusable components with standardized interfaces. This method emphasizes selecting, integrating, and utilizing pre-existing components, allowing for rapid development, scalability, and maintainability. By assembling software from interchangeable, well-defined components, CBSE enables swifter development cycles and easier maintenance, significantly favoring reuse over building from scratch.

SECO is defined as a network of entities operating collaboratively in a unified software and services market, according to [JBF09]. In SECO, the emphasis extends beyond component reuse to encompass a broader range of resources, including APIs, libraries, knowledge, and best practices. This approach fosters interoperability and collective contributions, thriving within open-source communities, collaborative development platforms, and interconnected software services.

Having clarified these concepts, a summarizing table that delineates these reuse-centric approaches is proposed.

Table 5.1 highlights their objectives, benefits, and challenges.

The table shows that SPL is optimal for managing and systematically reusing commonalities and variabilities across a spectrum of related products. SPL is particularly advantageous when developing a suite of related software products sharing common features and variabilities. CBSE shines when swiftly assembling systems by integrating well-tested, standardized components with specific functionalities or interfaces. Conversely, SECO excels in collaborative environments necessitating the sharing and reusing of diverse resources beyond components, catering to industries leveraging collective contributions across various platforms and communities.

As a result, the Software Product Line has been selected as the chosen reuse strategy for this thesis.

Aspect	Software Product Line (SPL)	Component-Based Software Engineering (CBSE)	Software Ecosystems (SECO)
Primary Focus	Developing a family of related software products	Building software systems by integrating reusable components	Creating an environment for collaboration and resource sharing
Key Objective	Manage commonalities and variabilities across products	Assemble pre-built components to form software applications	Facilitate collaboration, innovation, and resource sharing
Emphasis on Reuse	Systematic reuse of features, and artifacts	Utilization of existing pre-built components with interfaces	Diverse resource sharing beyond components
Approach	Identification and management of reusable elements	Selection, integration, and utilization of pre-built components	Fostering an environment for collaboration and resource sharing
Benefit	Cost reduction, faster time-to-market, consistent quality	Rapid development, maintainability, reuse of proven components	Innovation, interoperability, scalability, diverse contributions
Scope of Reusability	Features, artifacts, configurations within a product line	Pre-built components with standardized interfaces	Wide range of resources including components, APIs, knowledge
Challenges	Managing variability across product variations	Component compatibility, versioning, and selection	Governance, compatibility, evolving standards, diversity
Example Use Case	Developing multiple versions of a software application	Building an application using third-party libraries or APIs	Open-source communities, collaborative development platforms

Table 5.1: Comparative Analysis of Software Reuse Alternative: SPL, CBSE, and SECO

5.2.3 Delta-Oriented Programming principle

Implementing a software product line following the DOP principle implies implementing a core product and set of changes called delta modules that can be activated for a given configuration and applied to create a customized product.

Figure 5.2 gives the overview of the DOP principle.

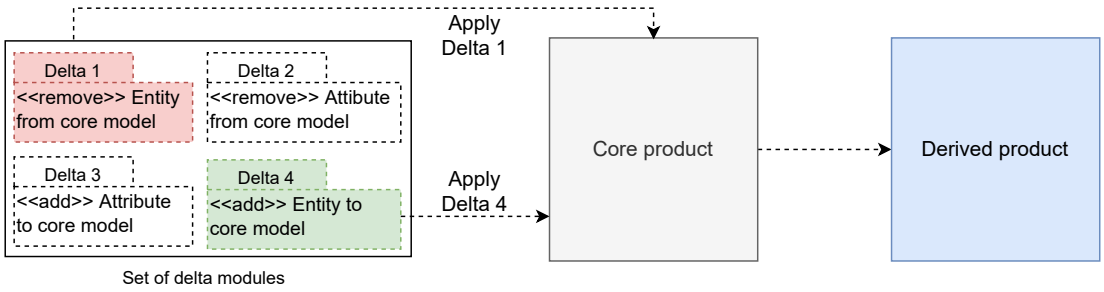


Figure 5.2: Understanding the DOP Principle: A Snapshot

In the context of DOP principles, it is crucial to comprehend several key terms [PKK+15]. The *Delta action* denotes a set of one or more modification actions applicable to an original product, resulting in a modified product. A *Delta module* serves as a container for modifications applicable to a product variant, en-

compassing a set of delta actions. The *Delta set* combines a delta module with a defined application approach to introduce more complex modifications to a product. This methodology enhances the reusability of delta modules, streamlining the development process by eliminating the need to create each potential delta module individually.

5.3 Motivation for Adopting a Software Product Line approach

Software development rarely starts from scratch, as developers naturally reuse existing solutions. Developers typically seek out similar, pre-existing software when creating new software, whether developed in-house or sourced from open-source repositories. This is where the Clown-and-Own (C&O) [RDR03, DRB+13] approach comes into play. This method involves forking and adapting existing software to create the desired product. Initially, the two software instances share common functionality, but the products are gradually differentiated through modification and introducing variability, resulting in a common base with variable components.

Software development often involves reusing existing solutions rather than starting from scratch. Developers commonly search for similar, pre-existing software when creating new software, whether developed in-house or sourced from open-source repositories. The Clown-and-Own (C&O), described in [RDR03, DRB+13], involves forking and adapting existing software to create the desired product. Initially, the two software instances share common functionality. However, the products are gradually differentiated through modification and introducing variability, resulting in a common base with variable components.

The importance of reuse extends beyond legacy systems; modern frameworks often provide generators that allow developers to focus on business logic by leveraging a foundational software base. A prime example is the *Spring Initializr*, a template that facilitates the generation of initial SpringBoot project code [KU22]. This approach allows developers to focus on the business logic, leveraging a foundational software base for time efficiency, although occasional assistance is required for extensions.

Utilizing an existing software base accelerates initial development but leads to managing multiple parallel applications. Transitioning between similar core products demands significant effort for effective management [LDDF11]. Library evolution introduces new cross-functional features and tests to support ongoing development [ZDAT22], yet errors often arise from risks linked to copy-and-paste tasks.

Figure 5.3 illustrates the C&O ad hoc approach, highlighting its potential drawbacks and the complexity of managing multiple derived products.

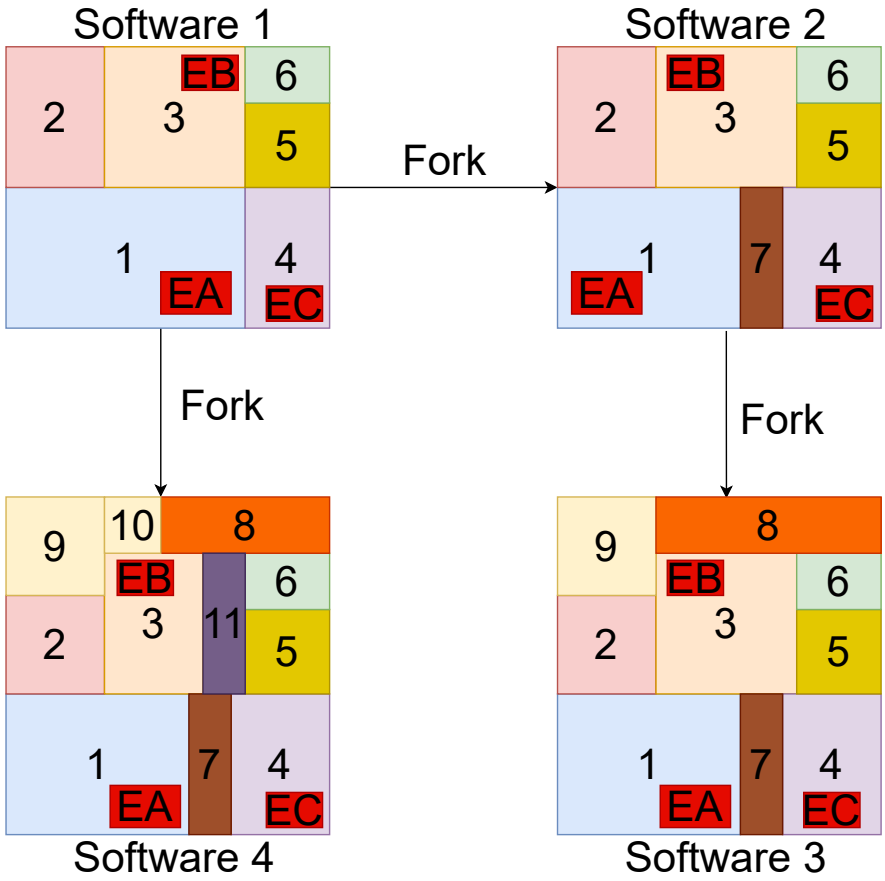


Figure 5.3: Illustration of the ad-hoc, Clone-and-Owns (C&O) approach

Four related software instances are depicted in the illustration (Figure 5.3). The arrow labeled *fork* signifies creating a software version derived from another through the ad-hoc C&O method. Each rectangle is assigned a number, and variations in rectangle sizes with the same number indicate modifications. An extra rectangle represents additional code introducing new functionalities to the software.

In Figure 5.3, we observe four related software instances. The arrow labeled *fork* signifies creating a software version derived from another through the ad-hoc C&O method. Each rectangle is assigned a number, and variations in rect-

	Base software	Bloc list	Update or error fixing
Software 1	Init	1, 2, 3, 4, 5, 6	EA, EB, EC
Software 2	Software 1	1', 2, 3, 4', 5, 6, 7, 8	EA, EB, EC
Software 3	Software 2	1', 2', 3, 4', 5, 6, 7, 8, 9, 10	EA, EB, EC
Software 4	Software 1	1', 2', 3, 4', 5, 6, 7, 8', 9, 10	EA, EB, EC

Table 5.2: Table of illustration of the ad hoc Clone-and-Owns (C&O) approach

angle sizes with the same number indicate modifications. An additional rectangle denotes additional code for new functionalities in the software.

Table 5.2 offers a more comprehensive explanation of these software entities.

Upon analyzing Table 5.2, it is evident that certain areas require enhanced and streamlined maintenance practices. Firstly, the selection of base software appears somewhat arbitrary. The developer may have chosen initial software that is similar but deviates from the desired version. In this case, *Software 4* seems closer to *Software 3* than *Software 1*. After applying the C&O method, the developer reworked the functionalities associated with blocks 1', 2', 4', 7, 8', 9, and 10. If Software 4 had been derived from Software 3, the developer would have only needed to address the corresponding code for blocks 8' and 10.

What is more, there are other areas in need of improvement. Even in the case of arbitrary developer selection, maintenance, and bug-fixing become repetitive tasks that require greater developer involvement. This is underlined by the last column of the table 5.2. The red rectangle in the figure 5.3 means that problems need to be solved or development needs exist, such as library updates, language migrations, and vulnerability fixes. Each software version requires development work, which can result in some software not being updated, making it obsolete or vulnerable. This results in redundant development, correction, and testing efforts.

5.4 Why should connectors be considered as a product line?

Section 5.3 delves into the challenge of unplanned software reuse via the ad-hoc C&O method prevalent in software development. This challenge is notably amplified in software connectors due to their susceptibility to change, influenced by their inherent evolution and the shifts in business software.

Connectors as a Product Line: A Necessity To provide a clearer insight into the challenge of efficiently reusing interoperability mechanisms, we delve into the specific interoperability mechanisms implemented at Berger-Levrault [1].

The Berger-Levrault information system comprises numerous heterogeneous constituents that must work seamlessly between themselves and third-party information systems. However, achieving this is not always straightforward due to constant technical and domain specifications changes.

Developing and maintaining interoperability mechanisms can be very time-consuming. The initial step was to decrease development time using a Message-Oriented Middleware (MOM) [YQC+19] library called BL-MOM. BL-MOM library was referenced in [ALL+20]. BL-MOM library offers low-level features, including a publish/subscribe [Tar12] messaging mechanism via the event broker RabbitMQ [Tos15], following the asynchronous message queuing protocol (AMQP) [Pra21].

Although BL-MOM offers advantages, such as reusing low-level functionalities and enabling focus on high-level connector functionality, a significant portion of the connector development process remains manual. Consequently, despite considerable efforts to boost productivity, only a few connectors rely on the BL-MOM library. Streamlining code reuse is crucial to minimize development time and effort.

Figure 5.4 outlines the current process and the proposed approach to augment reuse.

Table 5.3 shows the metrics extracted from the Berger-Levrault private GitLab projects for seventeen connectors developed using the BL-MOM library, including the actions required for their development and maintenance.

These metrics include the commit count, contributor involvement, duration from project start to the latest activity, the previous iteration commit count, average monthly commits, and the current project maintenance status.

The table displays a selection of connectors developed by Berger-Levrault, as part of a project to facilitate event-driven communication among applications.

[1] `https://www.berger-levrault.com/fr/`

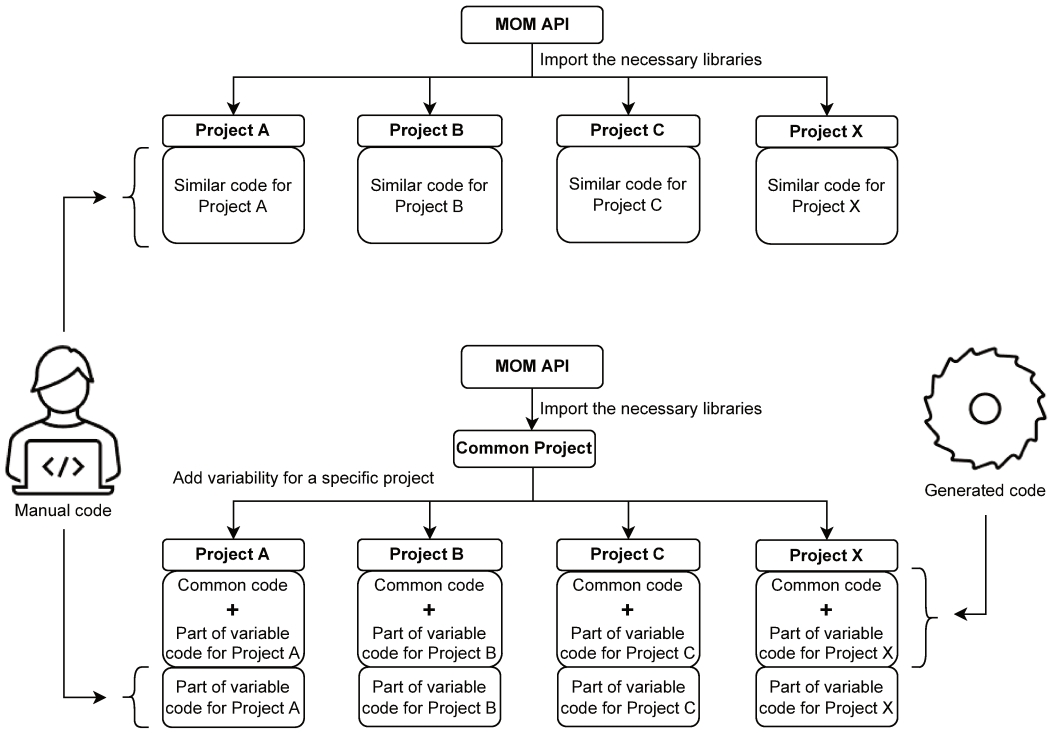


Figure 5.4: Overview of the current industry practices compared to the proposed approach based on SPL. The upper section outlines the process that employs the BL-MOM library without SPL, while the lower section delineates the process that incorporates SPL.

Commit frequencies differ among projects, providing insights into maintenance and evolution needs. However, relying solely on this metric to assess overall effort requirements is insufficient. Key indicators such as contributor counts, last project update dates, and cumulative commit counts are equally essential for a comprehensive evaluation.

The varying average monthly commit rates, ranging from 0.3 to 30.4, highlight different levels of development intensity among the projects. Notably, Project O indicates a surge in early-stage development activity with an average of 30.4. Across the board, the last column confirms that all projects remain active and under maintenance.

At first glance, these measures may seem lower due to Berger-Levrault's efforts to implement BL-MOM-based connectors. However, it is essential to note that the metrics could be higher for projects that have not undertaken this ini-

	Total commits	Contributors	First commit date	Last commit date	Number of commits at last date	Duration (month)	Average commits/-month	Active
Project A	185	3	2019-02-13	2021-12-15	3	34	5.4	YES
Project B	48	1	2019-01-08	2022-03-08	12	38	1.3	YES
Project C	34	1	2021-01-11	2021-12-15	3	11	3.1	YES
Project D	8	1	2019-12-02	2022-07-15	1	31	0.3	YES
Project E	153	2	2018-11-23	2022-03-17	1	42	3.6	YES
Project F	167	2	2018-11-23	2022-04-06	1	40	4.2	YES
Project G	33	1	2020-12-28	2021-12-15	4	12	2.8	YES
Project H	44	2	2018-12-17	2022-09-12	4	45	1	YES
Project I	55	3	2019-05-24	2022-07-07	2	34	2.3	YES
Project J	32	1	2020-06-03	2021-12-15	4	18	1.8	YES
Project K	77	4	2019-10-16	2022-07-07	2	34	2.3	YES
Project L	72	2	2019-12-13	2022-05-25	5	29	2.5	YES
Project M	30	3	2021-10-15	2022-09-29	3	11	2.7	YES
Project N	23	1	2020-12-30	2021-1215	4	11	2.1	YES
Project O	7	2	2022-02-10	2021-02-17	2	0.23	30.4	YES
Project P	65	3	2020-08-28	2022-05-02	4	21	3.1	YES
Project Q	46	3	2019-12-06	2021-12-15	5	24	1.9	YES

Table 5.3: Git metrics analysis for all connector projects of our industrial partner

tiative. Additionally, these indicators can increase significantly during significant transitions, such as the backward compatibility challenges from RabbitMQ to Apache Kafka. Similar issues may arise within Berger-Levrault or other contexts when using traditional ad hoc interoperability methods, resulting in significantly higher connector indicators.

BL-MOM serves as a shared source library, offering functionalities for connectors. Yet, developing and maintaining MOM-based connectors can be time-consuming and lack enthusiasm from other teams.

The software product line approach reduces development efforts by utilizing core functionalities for reuse, simplifying maintenance through collective issue resolution, and enabling variability to customize specific connectors.

A Practical Demonstration: Connectors as Product Lines To argue for considering connectors as a product line, we rely on the connector repository detailed in Chapter 3 and visualized in Figure 3.3. Within this repository, we iden-

tify the primary connectors relevant to specific case studies, which are crucial for validating the metamodel discussed in Chapter 3, as outlined in Subsection 4.1.2.

Figure 4.4, Figure 4.6, Figure 4.8, and Figure 4.10 illustrate these selected use cases. They were deliberately chosen using heuristics to minimize connector numbers while maximizing coverage across various models.

This deliberate focus on particular use cases represents a pessimistic scenario for evaluating commonality within the connectors repository. We have intentionally selected connectors with highly divergent patterns to ensure maximum pattern coverage. Doing so has created a worst-case scenario for commonalities among connectors. If the four most divergent connectors selected share common functionalities, the same will be true for all the other connectors.

Table 5.4 presents lists of connectors and their associated patterns for the use cases.

The table is read as follows:

The first column lists patterns found in at least one connector. The first row lists the considered use cases. Entries in subsequent columns and rows indicate whether a pattern is present or absent for a given use case. The presence of a pattern is denoted by its name, sometimes followed by a numerical identifier. If no number is specified, it implies a single instance. If a specific variant name is unavailable, it is indicated as *Yes* in the table for summary purposes.

Essentially, the data in column two and row two share a commonality, indicating the presence of a pattern. The pattern names denote variability, which is not the main focus of this subsection. The goal is to demonstrate commonalities to support software product lines. Entries marked as $N.C$ indicate no observed commonalities for a pattern within a particular use case.

By applying a basic calculation—dividing the number of cells without $N.C$ by the total number of cells—we find that 26 out of 36 instances indicate a 72% occurrence of commonalities.

5.5 *ConPL*: Model-Based Connector Product Line Framework

The Connector Product Line is focused on interoperability connectors as its software products. It differs inherently from the broader SPLE Framework illustrated in Figure 5.1 by offering a more granular perspective, especially concerning implementation strategies. The product line implementation within the solution space initially adheres to the Delta-Oriented approach. Subsequently, the product line implementation continues within the solution space, using a Model-Driven

Patterns	Use case 1	Use case 2	Use case 3	Use case 3
Incoming endpoint	Channel Adapter	Requestor	Producer	Requestor
Outgoing Endpoint	Service Activator	Replier	Consumer	Replier
Message Channel	Publish-Subscribe, Point-to-Point	Invalid Message	Publish-Subscribe, 2x Point-to-Point	3x Point-to-Point
Message Queue	3x Queue	3x Queue	2x Queue	4x Queue
Message Router	N.C.	N.C	2x Content-Based	Recipient List, aggregator
Message	Document	Document	Event	Test
Correlation Identifier	N.C	Yes	N.C	Yes
Message Transformer	N.C	N.C	N.C	Data Enricher, Translator
Return Adress	N.C	N.C	N.C	Yes

Table 5.4: Analysis of Commonalities within the Highly Diverse Connector Subset from the Connector repository

Architecture. This implementation operates at a Platform-Independent Level, in line with the standard terminology of Model-Driven Architecture (MDA).

Figure 5.5 delineates the customized software product line tailored for the connector product line within the model framework, adhering to the DOP paradigm.

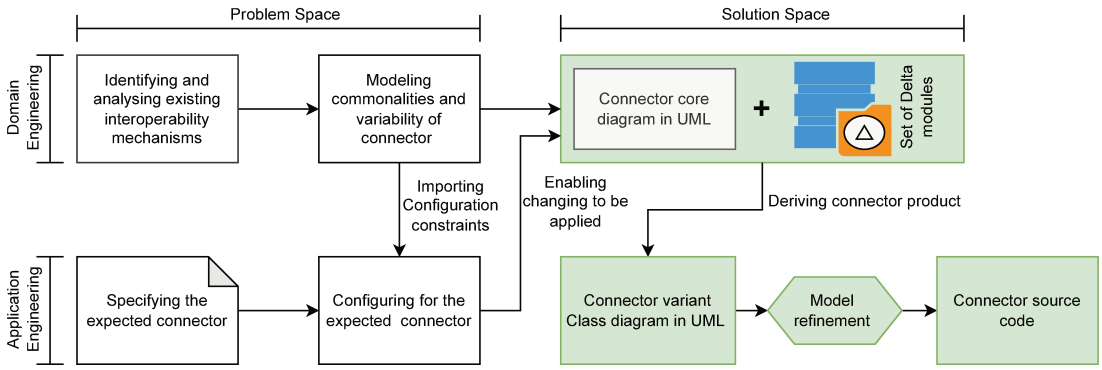


Figure 5.5: Overview of the *ConPL* Framework

This section provides a detailed, step-by-step explanation of the processes involved in the connector product line framework.

5.5.1 Analysis of Commonalities and Variabilities in Connectors

The initial stage of the domain engineering sub-process corresponds to the top left-hand box in Figure 5.5.

This stage involves identifying the functionalities of the connectors and evaluating their common and distinct characteristics. This step is essential as it validates the suitability of an SPL for a set of software applications. The primary focus of the SPL is the reuse of functionalities, where the availability of reusable components is essential. These reusable artifacts can include code snippets containing attributes, functions, classes, and other elements, collectively known as artifacts. Variability studies identify the potential locations of these artifacts based on configurable elements within the product line, which are referred to as features [PKK+15].

SPLE encompasses three primary approaches for analyzing product line characteristics, as delineated in [BKS15]. The proactive approach involves initial product planning and the creation of feature models (FMs) before starting software development. In contrast, the reactive approach involves an iterative analysis and modeling process embedded within the software development stages.

The extractive approach, SPL re-engineering, involves constructing core assets from an existing product not originally developed using SPLE.

Our research focuses on an extensible set of existing connectors from diverse sources. This extensibility implies that the connector repository can accommodate a new connector, as outlined in the process depicted in Figure 4.12.

Hence, our approach involves a combination of extractive and reactive approaches. This combined approach constructs *ConPL* using existing connector products, intending to evolve by seamlessly integrating new connectors into the product line.

While the iterative approach is straightforward—centered around iterative analysis and modeling, the extractive approach involves considering diverse strategies, as outlined by [ALHL+17]. These strategies encompass expert-driven, static, information retrieval, dynamic, and search-based analysis.

The Expert-Driven Strategy [WCC+95] relies on specialists' expertise, such as software engineers, architects, developers, and stakeholders. Static Analysis involves examining structural information from static artifacts without executing them. Dynamic Analysis [CZVD+09] uses tools to gather and analyze execution-related data, often focusing on low-level abstractions like source code. Information Retrieval leverages identifiers and comments to capture domain knowledge, often considering textual similarity. Search-based approaches [HMZ09] apply optimization algorithms, like Genetic Algorithms, derived from the optimization field.

This work relies on an Expert-Driven and static analysis strategy. The process leverages the connector repository developed in Chapter 3, a comprehensive repository encompassing connectors from various sources such as Berger-Levrault or industrial partners, software-neutral vendors, and open-source solutions. Noted that the connector repository led to a connector metamodel.

Most connectors within the repository are implicit, meaning their interoperability mechanisms are intertwined with the business logic code. Extracting the connector codes from the closely coupled business logic codes is essential to delineate their distinct characteristics. This challenge was addressed through reification in Chapter 3. Consequently, duplicating this effort is unnecessary. Leveraging the reified metamodel, which already consolidates the outcomes of the reverse engineering process, serves as our starting point. We will identify features that encompass all connector attributes from this metamodel.

Figure 5.6 showcases the connector characteristics captured by the metamodel, offering reusable artifacts through its entities, attributes, and method signatures. The next step is to model the features, which will be the focus of the following subsection.

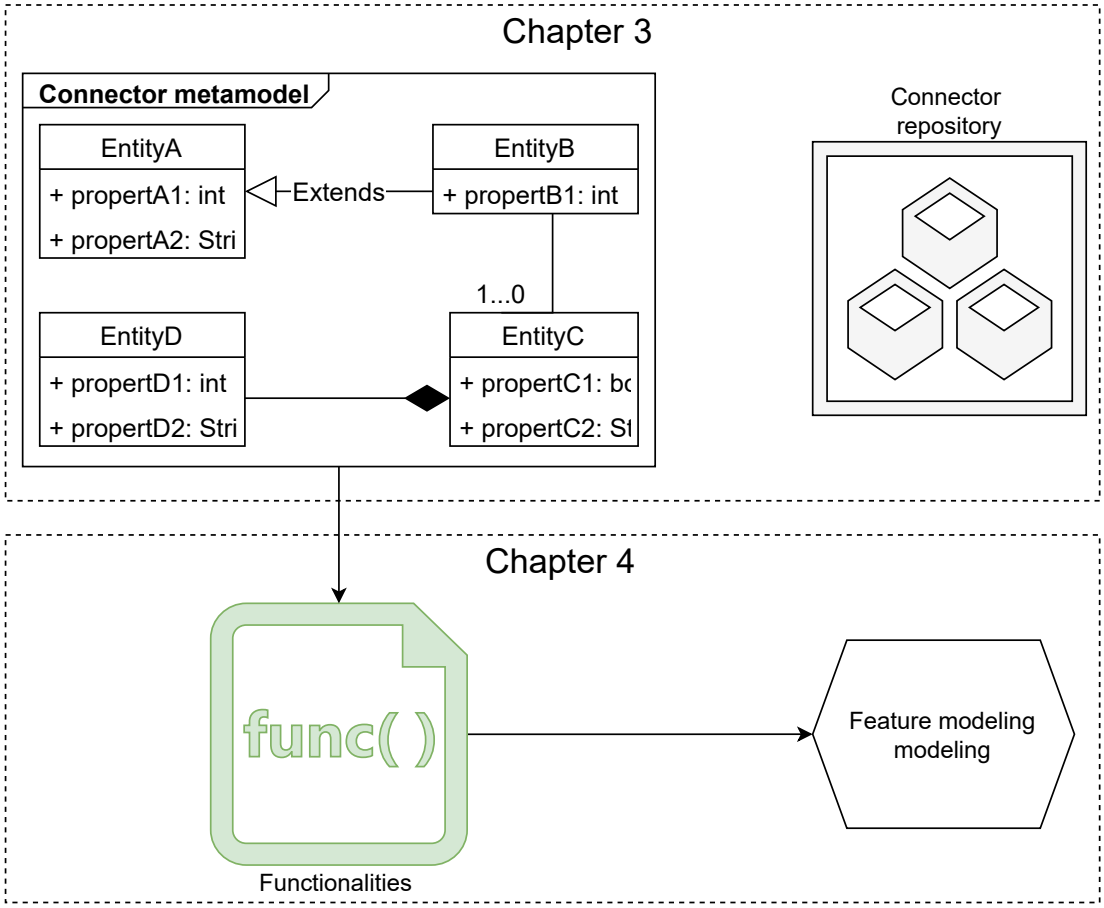


Figure 5.6: Illustrating the process of commonalities and variability analysis based on the connector metamodel

5.5.2 Modeling Variability in Connectors

The second step within the DE sub-process aligns with the second box positioned at the top left in Figure 5.1.

This phase involves exploiting the connectors' characteristics captured by the connector metamodel to organize them to illustrate their common aspects and differences, following the process described in Figure 5.6.

As the metamodel is a structure diagram, it may seem inappropriate to talk about organization. We are talking about organizations showing functionalities commonalities and variability for future configuration, so we need a type of representation conducive to this, hence functionalities modeling.

The literature defines two primary variability modeling approaches: decision models (DM) and feature models (FM). DM encapsulates a set of decisions that aid in distinguishing various members within an application engineering product family [Gom05]. FM specifies valid combinations of features and defines the scope/domain of a software product line [FGFdAM14].

The main distinction between FM and DM is their purpose. FMs aim to facilitate commonality and variability modeling, while DM focuses solely on variability modeling [C+93].

We have chosen feature modeling to exhaustively encompass all possible characteristics, commonalities, and variability and represent them graphically through a feature diagram. The connector feature model is constructed from the identified characteristics in the feature identification process. It represents all connectors based on the built connector repository.

Figure 5.7 illustrates the feature model showcasing all possible functionalities for the connectors within the connector repository.

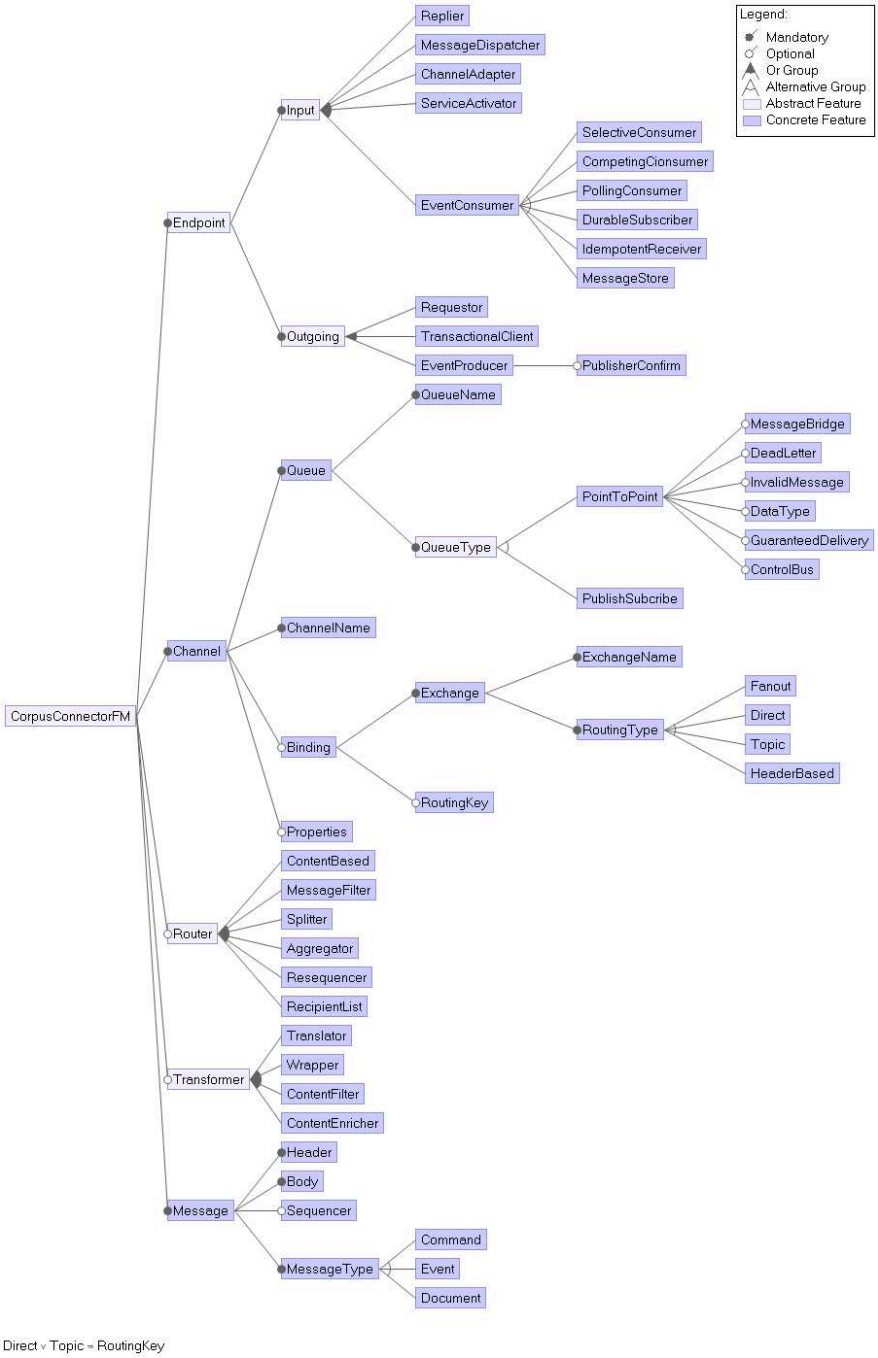


Figure 5.7: Feature Model Encompassing All Potential Connectors Emanated from the Established Connector repository

5.5.3 Implementing the Connector Product Line within the Solution Space

The final step of the Domain Engineering sub-process, located in the top left box of Figure 5.5, focuses on enabling reusability and subsequent product derivation. Several decisions are made in this stage, including choosing the implementation paradigm, defining the product's form and abstraction level, and organizing the reusable artifacts.

Choosing the Implementation Paradigm The literature categorizes software product line implementation methods into two main approaches: annotative and compositional. Annotative approaches use conditional compilation [LAL+10], which is based on the pre-processor technique, and represents one of the earliest approaches. On the other hand, compositional methods encompass Feature-Oriented Programming (FOP) [FGFdAM14], Aspect-Oriented Programming (AOP) [KLM+97], and Delta-Oriented Programming (DOP) [SBB+10], with the latest approach termed Trait-Oriented Approach (TOP) [BD17].

Though prevalent, the emergence of TOP prompts the recognition of a third approach—a transformation-based method encompassing DOP and TOP. Hence, our study identifies three categories of implementation approaches: annotative (conditional compilation), compositional (FOP, AOP), and transformative (DOP, TOP).

This thesis advocates implementing the software product line using the Delta-Oriented Programming (DOP) paradigm. While each paradigm possesses its unique set of advantages and drawbacks, selecting a paradigm hinges upon the specific requirements of the software product line. The rationale behind choosing DOP, supported by a comparative study, is outlined in Table 5.5.

In the comparative analysis presented in the table, the DOP paradigm stands out for its substantial support for incremental, modular modifications within software systems, allowing modifications to existing products by expressing changes as deltas. While FOP, AOP, TOP, and conditional compilation have their respective merits, they lack inherent support for the incremental and modular modifications critical to our software product line implementation.

The context outlined in this thesis revolves around a dynamic environment that requires adaptability to changing interoperability specifications. Given this context, the choice paradigm is needed to facilitate transformative approaches. When considering DOP vs. TOP, the decision was tilted towards the DOP

Paradigm	Approach	Focus	Description	Drawbacks
Conditional computation	Anotative	Preprocessor	Utilizes conditional compilation to include/exclude code fragments based on pre-defined conditions.	Limited flexibility for extensive software product line use.¡br¿-Can lead to codebase complexity and errors.
FOP	Compositional	Feature organization	Organizes code around features using feature models, representing desired software features.	Restricts modification of existing products. Not suitable for extensive product modifications.
AOP	Compositional	Separation of concerns	Enhances modularity by extracting common behavior into reusable aspects, though not inherently designed for SPL implementation.	Lacks explicit focus on software product lines. Limited capability for SPL implementation.
DOP	Transformative	Incremental modular modification	Facilitates incremental, modular software modification through expressing changes in the form of deltas. Considered an extension of FOP.	Complexity in managing a multitude of deltas. Learning curve associated with delta-based modifications.
TOP	Transformative	Methods independent of class hierarchy	Utilizes methods independent of class hierarchy to create specific products.	Less focus on software product line implementation. May not explicitly support SPL concepts

Table 5.5: Summarized software product line implementation paradigms

paradigm due to its unique ability to support incremental modifications, which aligns well with the dynamic nature of our environment.

Now that the DOP paradigm has been selected, let's present its principles and discuss our strategies to overcome its limitations.

Determining Product Line Abstraction Level The analysis presented in Table 5.5 highlighted the advantages of DOP paradigms in incrementally modifying existing software products. This conclusion led us to choose DOP paradigms for our approach.

Despite being a suitable SPL paradigm for interoperability contexts, DOP, like other paradigms, has certain drawbacks. Managing multiple deltas and the associated learning curve for delta-based modifications are key challenges.

Our solution is implementing the connector product line at the model level to address these issues. This abstraction allows for consistent modeling of the core delta module and facilitates product creation through refinement, such as moving from a high-level model to generated code. By using Model-Driven Engineering (MDE), we aim to improve the evolution and maintainability of the product line.

Architecture of the Connector Product Line The article explores the solution space for domain engineering by following DOP principles. The Connector Product Line integrates a core connector model and a repository of reusable

artifacts. The core connector model is depicted by the connector metamodel, structured as a class diagram showcased in Figure 3.4, which was developed in Chapter 3.

Although this metamodel effectively captures the structural information of the connector, it is limited to class, properties, and method signatures. Relying solely on this information would restrict the potential of our approach. To fully achieve our aim of reducing the volume of code developed by programmers, we must leverage a repository of source code from diverse connector projects within our connector repository. This will provide a significant advantage.

To adhere to this perspective, the implementation of the software product line includes three main components: the connector metamodel, which captures structural information; a repository of reusable artifacts, which includes method-level behavior elements; and a collection of delta modules that contain modifications applicable to the central delta module.

Figure 5.8 presents a detailed illustration of the domain engineering implementation. It shows the main components of the solution space in the domain engineering sub-process. The delta module repository describes potential modifications that can be applied to one or more model versions. Reusable artifacts are organized within delta modules, creating a many-to-many relationship: each delta module contains multiple artifacts, and conversely, artifacts can belong to multiple delta modules. This connection aligns entities, properties, and method-level artifacts, ensuring each entity is associated with a reusable delta module.

Introducing a Metamodel for Managing Repositories of Reusable Artifacts The method-level reusable artifacts are crucial for creating coherence within entities and delta modules. However, managing these artifacts can become cumbersome without a specified organizational structure.

To streamline this process, we propose storing these artifacts in a separate repository structured as a JSON file containing an array of JSON objects. Our proposed metamodel is designed to address method source code, facilitating the integration and organization of these reusable artifacts.

Figure 5.9 illustrates the proposed metamodel for organizing method-level reusable artifacts.

A JSON structure representing a delta module has a unique identifier called the *name* and a field called *predecessors*. The *name* acts as a distinctive tag for the delta module, while the *predecessors* field contains a list of other delta module names that must be applied before the current one, separated by commas.

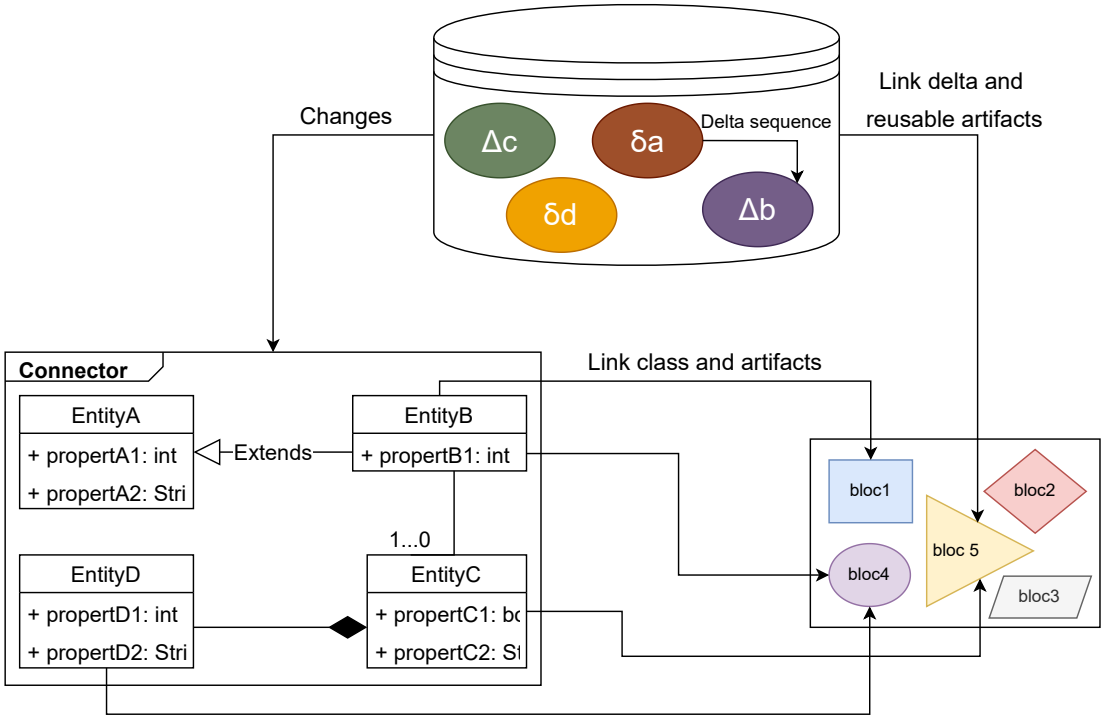


Figure 5.8: Comprehensive Overview of the Solution Space for the Connector Product Line in Domain Engineering: Leveraging DOP Paradigms

This list defines the sequence in which these modules should be applied. Each delta module can have none or multiple predecessor delta modules.

A delta module contains zero or several *classes* corresponding to the core delta module entity. Each class has a name and includes multiple *methods*. The methods within a class have their name and a *source anchor* that represents the source code of the methods and pertains to the reusable artifacts.

The delta module exclusively contains methods not present in any of its predecessor modules. If a delta module depends on another through the precedence relationship, it inherits methods from the *predecessors* delta modules.

5.5.4 Mapping Guidelines: Feature Model to Model-Level Product Line Architecture

The connector product line architecture, represented by the connector metamodel, was established through a thorough reification process outlined in

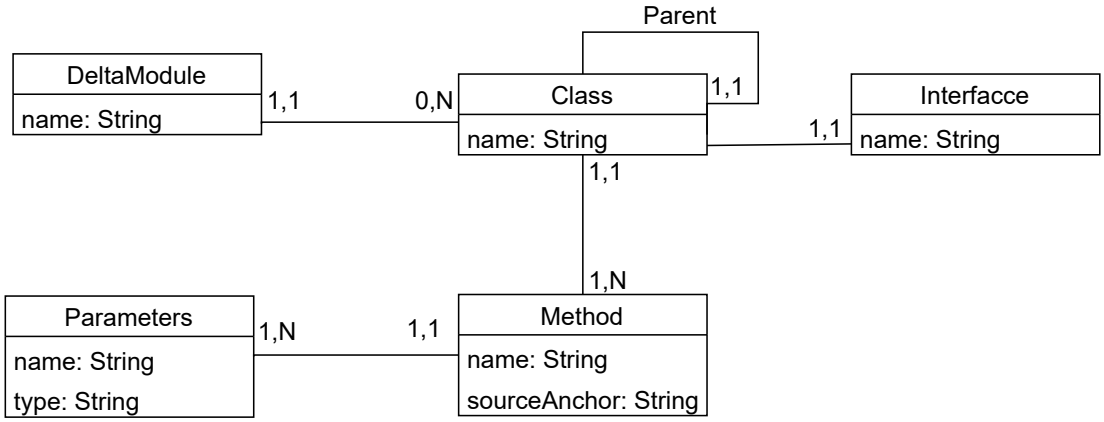


Figure 5.9: Metamodel organizing the Repository of Reusable Artifacts

Chapter 3. This foundational step aligns with the ConPL framework, simplifying the transition from problem to solution space.

The feature model usually comes before the product line architecture. However, bridging the gap between the domain engineering problem and its solution can be intricate, specifically transitioning from the feature model to the product line implementation.

Expanding beyond the variability model, implementing a product line often stalls without leveraging its full potential. To address this, we propose guidelines for a Model-to-Model (M2M) transformation [LWK10]. This transformation bridges the gap, converting a feature model into a product line architecture represented explicitly by a class diagram.

M2M transformations are foundational in model-driven engineering and can be approached through various methods, including operational and declarative. Declarative approaches articulate the relationship between source and target models, while operational approaches delineate the execution steps in generating the target from the source model.

To streamline the transformation process, we categorize features into three distinct groups: the root feature, the internal node feature, and the leaf feature. The root feature denotes the primary feature devoid of any parent association. Internal node features, while not the root, contain at least one child feature. Leaf features, on the other hand, lack child features and can be categorized further as those with and without attributes.

We have developed twelve transformation rules categorized into three sections: feature transformation, relationship modifications, and constraint handling.

N°	Category	Name	Description
1	Feature	Root Feature	Transformed into the root class of the class diagram, establishing a composition relationship with related classes, indicating their dependence.
2	Feature	Internal node Features	Converted into classes; their attributes become the class's properties.
3	Feature	Leaf Features (with Attributes)	Transformed into separate classes
4	Feature	Leaf Features (without Attributes)	Become operations of the parent feature's class
5	Relationship	Mandatory	Transformed into a composition relationship where the parent becomes the composite class, and the child becomes the part class, enforcing the child's presence if the parent exists.
6	Relationship	Optional	Similar to mandatory but with a maximum cardinality of 1 and allowing the child's absence if the parent exists
7	Relationship	OR group	Enforces a composite relationship, allowing for multiple composite classes (one per child feature)
8	Relationship	Alternative	Requires the presence of only one child element, becoming a composite class with one part class (child feature). To ensure mutual exclusion, constraints are expressed in Object Constraint Language (OCL) [RG02].
9	Constraint	Implies	Represents the relationship between Feature A and Feature B where A can exist if B exists but not vice versa. This constraint prevents the existence of the class from Feature A without the class from Feature B
10	Constraints	If and only if (IFF)	Indicates that Features A and B can exist or not but simultaneously. This constraint ensures that a class from Feature A or B does not exist without the other.

Table 5.6: Rule for transforming the feature model into a class diagram

These rules systematically guide converting features, associations, and constraints from a feature model into a structured class diagram.

Refer to Table 5.6 for a detailed breakdown of these transformation rules.

The table presents transformation rules for converting a feature model into a class diagram, categorized into three sections: features, relationships, and constraints. This structured guideline accurately represents the transformed model's feature attributes, relationships, and constraints in the resulting class diagram.

Figure 5.10 shows an illustrative example involving transforming a feature model to a metamodel.

The proposed rules can be used manually or automatically. Automatic transformation can be accomplished using some template-based transformation engine such as the Atlas transformation model (ATL) [JABK08] or Apache velocity [CH07].

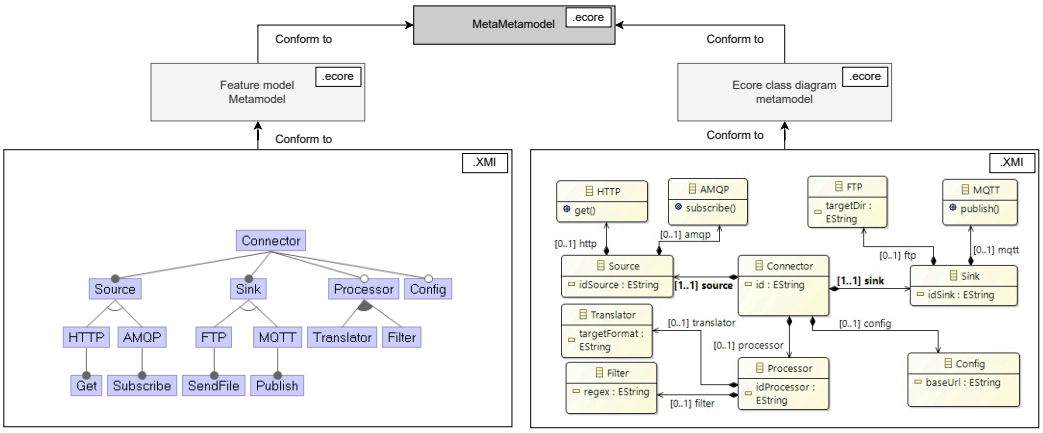


Figure 5.10: Transformation Process: Feature Model to Model-Level Core Product

5.5.5 Application Engineering through Model-Driven Engineering

In software product line engineering, application engineering is pivotal for crafting connector products that meet specific criteria. This process unfolds in three key stages: defining precise product specifications, selecting suitable configurations, and actualizing products aligned with the chosen setup. Employing a Model-Driven Engineering (MDE) approach within this context streamlines the transformation of resulting product models into executable code customized to suit the distinct requirements of the software product line.

Utilizing Domain Engineering for connector creation Within the application engineering sub-process, domain engineering is vital in crafting connectors, progressing through four structured steps. Firstly, the process initiates by precisely outlining the desired attributes of the connector. Following this, stakeholders configure the domain feature model, meticulously selecting the necessary features for the connector's specification.

The subsequent step involves synthesizing the anticipated connector model. This synthesis draws from various sources, such as the connector product line architecture, delta modules, constraints originating from the domain engineering sub-process, and the specific configuration desired for the connector. In the domain engineering phase, delta actions are tailored to reflect diverse constraints pertinent to each configuration.

This meticulous process results in the automatic proposition of all delta actions corresponding to the feature model configuration, respecting the estab-

lished constraints. Ultimately, this effort yields the expected connector model, which is subsequently used to generate the source code of the connector in the desired programming language.

Application Engineering and connector model derivation This section primarily delves into the domain of application engineering, building upon the groundwork laid by the domain engineering sub-process to generate the intended connector. The sequential steps involve meticulously specifying the connector attributes, configuring the domain feature model, and applying delta modules from the domain engineering sub-process to derive the expected connector model.

The resulting target connector model embodies a specific variant derived from the applied delta actions, a pivotal aspect of this modeling process. Finally, the ultimate stage culminates transforming the resultant connector model into executable code. The elucidation of this intricate process utilizes concrete examples within scenarios detailed in the forthcoming implementation use case in Section 5.6. This practical illustration illuminates the complexities and practical applications of this engineered approach to connector development.

5.6 Practical Application Scenario

In Chapter 3, we detailed the rationale behind analyzing a multitude of connectors during the creation of the connector repository and metamodel. These connectors encompassed a range of sources, including those from industrial partners, vendor-specific, and vendor-independent solutions. Each case study revealed diverse patterns that contributed to defining the metamodel's dimensions. To ensure a well-rounded experimentation framework, we selected a connector from each source to create a concise yet representative product line.

Our selection criteria prioritized shared commonalities among the three chosen connectors to streamline the selection process. However, during the metamodel validation in Chapter 3, our focus shifted towards connectors covering a broader spectrum of patterns, even if they shared fewer commonalities. This shift aimed to embrace more diverse patterns, resulting in a comprehensive and detailed product line tailored for efficient product code generation. This emphasis underscores the significance of our work in establishing a versatile connector repository, offering flexibility in selecting connectors based on our specific experimental requirements.

Within the experimental product line, six primary message connectors have been derived from industry expertise—Berger-Levrault, RabbitMQ GitHub

page [2], and EIP book example based on Java Messaging Service [3]. These connectors, based on Berger-Levrault's BL-MOM library, unify various use cases sourced from multiple origins, incorporating RabbitMQ and Java Messaging Service examples. Let's delve into the exploration of these connectors.

Connector 1 , depicted in Figure 5.11, entails a message producer responsible for message publication and a consumer handling message reception and printing.

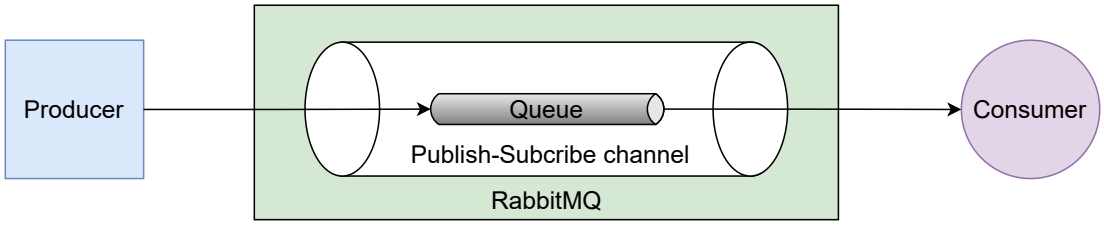


Figure 5.11: Enabling fundamental *Publish-Subscribe* exchange with RabbitMQ

Connector 2 , depicted in Figure 5.12, introduces an intermediary step using an *Exchange* for message publication. This approach enables producers to send messages without specifying a particular *Queue*, facilitating adaptable message retrieval and processing by recipients. The Exchange receives messages from producers and distributes them across queues. This use case focuses on *fan-out*, sending messages to all connected queues, thus enhancing system flexibility and decoupling. For further details on AMQP Exchange, refer to the RabbitMQ documentation [4].

[2] `https://github.com/rabbitmq`

[3] `https://www.enterpriseintegrationpatterns.com/patterns/messaging/RequestReplyJmsExample.html`

[4] `https://www.rabbitmq.com/tutorials/amqp-concepts.html`

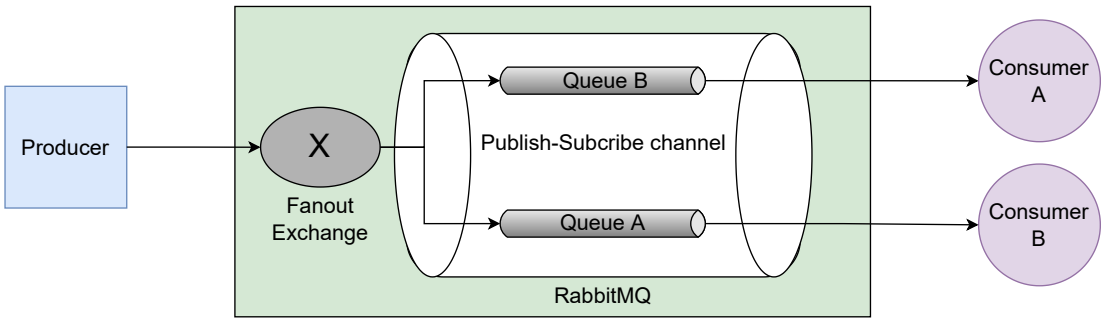


Figure 5.12: Enabling *Publish-Subscribe Fanout Exchange* Pattern with RabbitMQ

Connector 3 (Figure 5.13) employs a *direct* type of exchange to *Connector 2* but introduces message filtering based on routing keys. In this setup, consumers intending to receive the message must be subscribed to a specific queue where the name matches the routing key, for instance, "prescription.hospital.france" or "prefecture.lyon". Compliance with this format is crucial for the consumer.

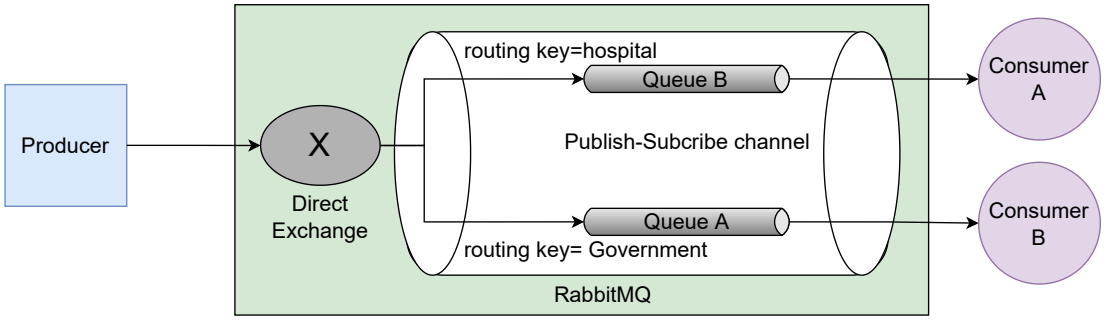


Figure 5.13: Enabling *Publish-Subscribe Direct Exchange* Pattern with RabbitMQ

Connector 4 (Figure 5.14) improves routing flexibility using *topic* exchanges, directing messages with specific routing keys to corresponding queues. Unlike *Connector 3*, consumers interested in receiving the message need not subscribe to a queue that precisely matches the routing key; instead, they adhere to a pattern such as *hospital.*.france*, *prescription.#*, or *city-hall*. Here, "*" signifies any word, while "#" denotes one or more words within the routing key. In *hospital.*.france* example "*" can be replaced by *public* or *private*.

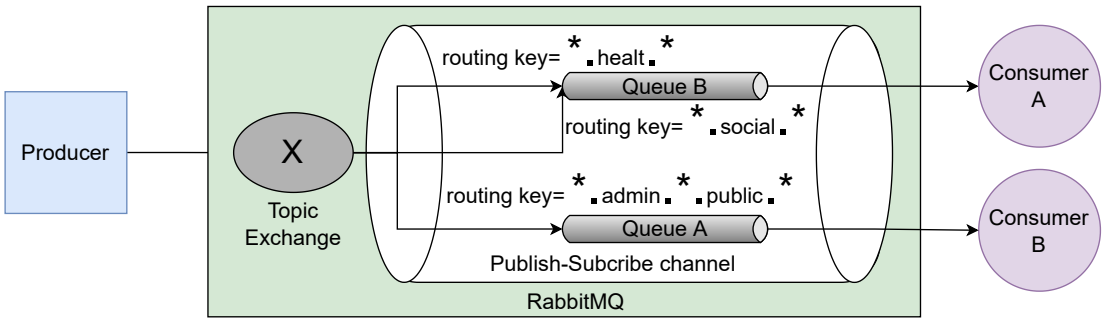


Figure 5.14: Enabling *Publish-Subscribe Topic Exchange* Pattern with RabbitMQ

Connector 5 (Figure 5.15) diverges from the initial four examples by employing *Command* messages rather than *Event* messages and operates distinctively from the *fire-and-forget* principle. It initiates requests and expects server responses.

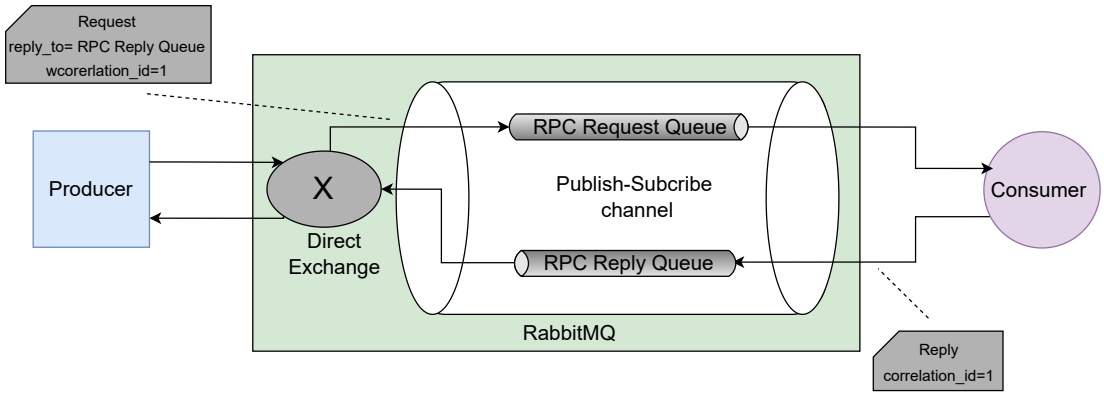


Figure 5.15: Enabling publish-subscribe *RPC* communication with RabbitMQ

Connector 6 demonstrates asynchronous request-response using JMS (Figure 5.16). It comprises a *Requester*, a *Replier*, and dedicated handling of invalid messages in a separate *Invalid channel*. Unlike the previous five use cases, this connector operates on JMS instead of RabbitMQ. Furthermore, unlike the first four connectors, like Connector 5, it does not rely on the *fire-and-forget* principle. Although a response similar to Connector 5 is required, the difference lies in using a *Command* message for Connector 5 and a *Text* message for Connector 6. This distinction extends to using *Event* messages in the initial four connectors.

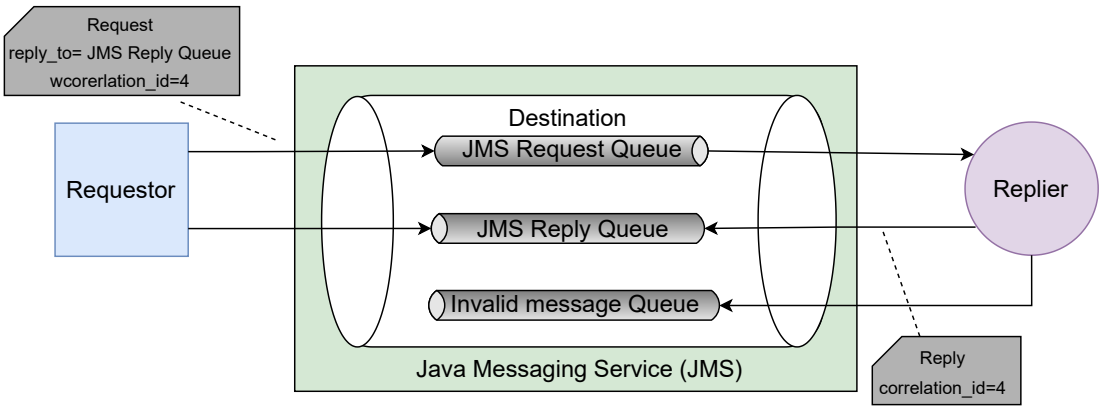


Figure 5.16: Enabling asynchronous *Request-Reply* Exchange via Java Messaging Service (JMS)

The descriptions of these connectors highlight their functionalities and capabilities, both common and unique. Future enhancements may be necessary to tailor the connectors to specific requirements. This exploration provides a comprehensive overview of the connectors' essential features, commonalities, and distinctive functionalities and suggests potential improvements for more advanced connectors.

5.7 Summary

This chapter introduces an innovative approach to generate interoperability connectors, employing a methodology rooted in Software Product Line Engineering (SPLE) specifically tailored for implementing these connectors. Throughout the Domain Engineering phase, we harness a connector repository to construct a comprehensive feature model encompassing all potential connectors for messaging interactions. Simultaneously, we utilize the connector metamodel as a vital component within the connector product line. This metamodel, integrated with the system metamodel, is complemented by a repository of delta modules and a collection of reusable artifacts crucial for illustrating the architecture of the connector software product line.

Our proposed approach features explicit rules and automated transformations, enabling the seamless conversion of a feature model into a class diagram. Additionally, we introduce a meticulously structured metamodel for reusable artifacts at a fine-grained code level, optimizing their practical application.

The subsequent chapter will delve into application engineering, where we will demonstrate the feasibility of this approach through real-world industrial case studies. This section will intricately detail the configuration and derivation of an expected connector model from the connector architecture, employing delta modules that contain essential delta actions. Furthermore, we'll provide insights into generating connector source code from the model.

Despite our adoption of model-driven engineering to achieve platform independence, the current model for code generation is tailored to specific communication patterns. Therefore, future iterations will develop a more inclusive model encompassing diverse interoperability platforms. Additionally, our approach aims to incorporate comprehensive support for connector lifecycle management.

Chapter 6. Tooling Support for Implementing Software Product Lines: The *PhaDOP* Framework

In Chapter 5, the *ConPL* framework is introduced, which is used for building a Connector Product Line (SPL) by combining Delta-Oriented Programming (DOP) and Model-Driven Engineering (MDE). The chapter provides an overview of the basic principles of DOP. However, to practically implement a software product line for product creation, it is necessary to address the solution space within the domain engineering (DE) and application engineering (AE) sub-processes. This section involves preparing all reusable artifacts and applying them for product derivation. Figure 5.8 illustrates the steps in implementing the software product line. The chapter presents the *PhaDOP* framework, which provides technological support for implementing a software product line (SPL) while focusing on the solution space. It includes a fundamental use case to demonstrate the practicality of the *PhaDOP* framework within the *ConPL* framework.

6.1 Surveying Tools for Software Product Line landscape

Software Product Lines (SPL) implementation encompasses a range of tools, but only a few can handle the solution space. Due to this lack of technological support, SPL has not been widely adopted yet [SRA19]. As a result, most projects only focus on product line modeling, where experts use features to create specifications and configuration guides. Stakeholders usually do not move beyond the domain engineering stage. Although DOP is one of the latest approaches to SPL, it also faces the same limitation of technological support. This section provides an overview of well-known tools organized by their primary focus.

Tool for Analyzing, Modeling, and Testing in the Problem Space: Software product line building often takes an extractive approach, where product lines are derived from existing products. Tools such as BUT4Reuse [MZB+17] play a crucial role in extracting variants of software artifacts, addressing commonality, variability, and feature identification.

For functionality modeling and configuration generation in large systems, feature modeling tools such as FeatureIDE [KTS+09] and Familiar [ACLF13] are commonly used. These tools are essential for effective feature representation.

ECCO [FLLHE15] streamlines the product line engineering process by automating various steps, including characterization, variability modeling, and code generation. However, it is limited to handling source code.

IsiSPL [Hla22] covers the entire software product line (SPL) lifecycle, emphasizing feature localization, product generation, and incremental product lines specifically for Java programs. Although it is based on the Abstract Syntax Tree of Java programs and supports the Java language, it follows an annotative approach rather than the Delta-Oriented Programming (DOP) paradigm and lacks support for model-level product line implementation.

LEADT[1] is a Java-based tool designed to help developers identify functionality within Java code and translate it into a software product line. TypeChef [KKHL10] focuses on verifying SPL properties without addressing product generation.

Solution Space Implementation Tool: For FOP paradigms, AHEAD [Bat04], a tool that supports stepwise refinement, allowing incremental feature addition. FeatureHouse [AKL09] provides a language-independent framework for composing software artifacts, such as Java source code.

For AOP paradigms, AspectJ [KHH+01], an Aspect-Oriented Programming (AOP) extension to Java, facilitates modular implementation of cross-functional features. CaesarJ [AGMO06] unifies aspects, classes, and packages, solving problems in AOP and component-oriented programming. FeatureC++ [ALRS05] extends C++ for aspect-oriented and feature-oriented programming.

For conditional compilation, Colligens [MLD+13], a tool for pre-processor-based SPLs in C that integrates TypeChef and FeatureIDE on the Eclipse platform [2]. Munge [3] is a straightforward Java preprocessor designed for simplicity. It exclusively supports conditional source inclusion based on specified string patterns such as "if[tag]", "ifnot[tag]", "else[tag], and "end[tag]". Unlike conventional preprocessors, Munge retains all comments and formatting in its output, ensuring human-readable source code distribution. CIDE [KA09] focuses on preprocessor-based SPL with a variability analyzer. VariantSync [PTS+16] automates synchronization between variant products.

[1] `https://github.com/ckaestne/LEADT`

[2] `https://projects.eclipse.org/projects/eclipse.platform`

[3] `https://github.com/sonatype/munge-maven-plugin`

For the TOP paradigm, TraitRecordJ [BDSS13]programming language, which is a Java dialect that uses traits and records as composable behavior and state reuse units, respectively, and aims for interface-based polymorphism. Xtraitj [BD17] is in continuity TraitRecordJ. It is a language for pure trait-based programming interoperable with the Java-type system without reducing the flexibility of traits.

For the DOP paradigm, DeltaJ [KHS+14], which is a pioneering tool for DOP, offers operations for adding, modifying, and deleting methods and class fields in Java programs. ParametricDeltaj [WKS+16] extends DeltaJ to consider attributes as parameters. Delta-MontiArc [HRR14] is a delta-oriented variability modeling language for architectural variability. SiPL [PKK+15] is a tool suite supporting DOP at the model level, enabling software product line implementation. It allows the construction of SPLs over Python code through model-to-text transformation. PYDOP [Lie23] is a recent Python library implementing DOP for building SPLs over transformable Python artifacts.

Discussion Tools such as ECCO and IsSPL cover various steps in Software Product Line (SPL) development but do not embrace Delta-Oriented Programming (DOP) paradigms. AHEAD and FeatureC++ excel in compositional SPL but lack transformation capabilities. DeltaJ and Delta-MontiArc are focused on code-level implementation within the DOP context. SiPL, which focuses on DOP at the model level, is the closest approach to meeting the specified objectives. However, it requires manually creating model versions to calculate Delta Modules.

To address these gaps, we introduce the PhaDOP framework, an SPL implementation approach that aligns with the DOP paradigm and leverages model-driven engineering. This framework guarantees platform independence and includes features such as GUIs for Delta Project management, Delta Module creation and application, model-to-Java code transformation using the Pharo and Moose platforms, and visualization of Delta Modules. Section 6.2.1 explains the framework's internal functionalities and implementation details and provides guidelines for usage.

6.2 PhaDOP: A Pharo Framework for Implementing Software Product Lines using Delta-Oriented Programming and Model-Based Engineering

PhaDOP is a framework for implementing software product lines using DOP and MDE. The framework is designed to overcome technological barriers, facilitating the implementation of the industrial software use cases presented in Chapter 5, Section 5.6, Figure 5.11, 5.12, 5.13, 5.14, 5.15, 5.16.

6.2.1 The *PhaDOP* Framework: Overview and Internal Mechanism

This section introduces PhaDOP, a model-driven delta programming framework. Figure 6.1 gives an overview of the main components and steps. The implementation is based on the Pharo language and the Moose platform. Pharo [4] is a pure object-oriented, open source, dynamically typed programming language inspired by Smalltalk [BNDP10]. It also provides a powerful and user-friendly environment with a focus on simplicity. Moose [AEH+20] is an open-source platform for software and data analysis in Pharo.

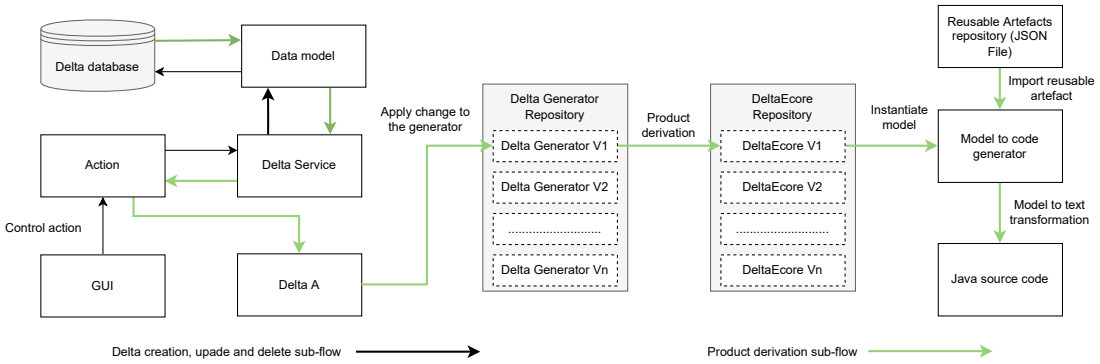


Figure 6.1: Overview of the PhaDOP Framework: Main Components and Steps

The framework comprises a set of components that we will describe in detail.

The PhaDOP Graphical User Interfaces (GUIs): Managing Delta Projects and Delta Modules can be complex when dealing with large amounts of source

[4] `https://pharo.org/`

code or many Delta Modules. The main challenge is to maintain usability when working with Delta Modules. It is essential to hide potentially tedious tasks such as program execution or source code manipulation. This is one of the limitations we have pointed out with DeltaJ [KHS+14]. To overcome this challenge, the *PhaDOP* framework introduces a set of graphical user interfaces (GUIs) that allow users to interact with Delta Projects and modules. These interactions include creating, updating, and deleting Delta Modules. Developed using Spec2 [DVD19], a framework designed for building GUIs in Pharo, the *PhaDOP* framework provides six GUIs, called*Presenters*, that users can use for their interactions. The first presenter, named *SpDopToolStartPresenter*, serves as the entry point of the framework. Figure 6.2 illustrates the starting presenter of the PhaDOP framework.

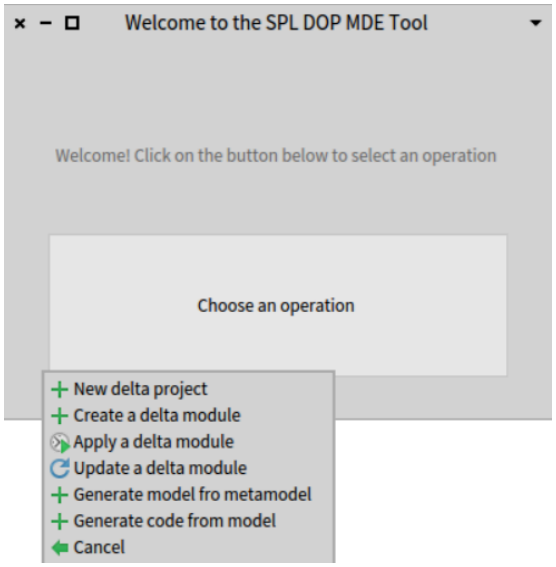


Figure 6.2: Overview of the PhaDOP starting interface

This starting presenter provides a sub-menu of options for users to navigate to their desired presenter. The sub-menu offers choices for seven GUIs, each of which is explained in the following sections: new Delta Project for project initialization, create a delta, apply a set of Delta Modules, update a Delta Module, generate a metamodel from a generator, generate Java code from a metamodel variant, and cancellation. To launch the starting GUIs, which serve as the entry point for the framework, execute the code snippet provided in Listing 6.1 in the Pharo playground.

Listing 6.1: Pharo code for launching the starting presenter

```
SpDopToolStartPresenter new start
```

Listing 6.2 shows the code responsible for creating the sub-menu in the starting presenter.

Listing 6.2: Pharo code for launching the main presenter

```
subMenu
    ^ self newMenu
        addItem: [ :item |
            item
                name: 'New Delta Project';
                icon: (self iconNamed: #add);
                action: [ SpCreateDeltaProjectPresenter new start ] ];

        addItem: [ :item |
            item
                name: 'Create a Delta Module';
                icon: (self iconNamed: #add);
                action: [ SpCreateDeltaModulePresenter new start ] ];

        addItem: [ :item |
            item
                name: 'Apply a Delta Module';
                icon: (self iconNamed: #smallProfile);
                action: [ SpApplyDeltaModulePresenter new start ] ];

        addItem: [ :item |
            item
                name: 'Update a Delta Module';
                icon: (self iconNamed: #smallUpdate);
                action: [ SpUpdateDeltaModulePresenter new start ] ];

        addItem: [ :item |
            item
                name: 'Generate model fro metamodel';
                icon: (self iconNamed: #add);
                action: [ SpModelGeneratorPresenter new start ] ];


        addItem: [ :item |
            item
                name: 'Generate code from model';
                icon: (self iconNamed: #add);
                action: [ SpModelToCodePresenter new start ] ];

        addItem: [ :item |
            item
                name: 'Cancel';
                icon: (self iconNamed: #back);
                action: [ self inform: 'Cancelled back to main menu' ] ];

        yourself
```

Control actions: is activated upon clicking and directs the model toward specific tasks. As per our design, these tasks are assigned to other components, namely *Delta service* for functions related to database access and *Delta action*, which handles code modifications.

Delta services: are called by control action components to handle data access. Consolidating all the code within the control action poses code readability and maintenance risks. The control action module uses a data transfer object to insert

data into the internal database [Mon03]. It also retrieves data from the internal database and passes it to the action module for logical processing requirements.

The data model: encompasses all the properties that define the data being handled, such as the Delta Core, Delta Module, entities, and generators. It is a blueprint for the required data structure for the framework's operations. These data elements are mapped to the internal database to ensure persistence within the system.

Delta database: The Delta database is the repository for essential data within the framework, mirroring the data model. Figure 6.3 illustrates the database structure, highlighting various entities and their interconnections. Information is stored or retrieved for creation, update, deletion, and application.

The Delta Core table stores initial project information, such as a unique identifier, project name, and a feature list that defines possible configurations. The table gets populated when initializing a Delta Project using the *SpCreateDeltaProjectPresenter* interface, which is a GUI for initializing a new Delta Project.

The Delta Module Table contains information on each Delta Module, such as a unique identifier, module name, application condition, predecessors (an ordered list of preceding Delta Modules), the associated Delta Core, and addon/removable entities. This information is recorded using the GUI *SpCreateDeltaModulePresenter*.

The Model Generator table contains details of all generators, each with a unique identifier, name, target package, prefix, and suffix. Generators are copies of the Delta Core metamodel modified by Delta Modules. This table is populated when registering a Delta Module through the *SpCreateDeltaModulePresenter* GUI.

The Entity Table stores entities involved in delta operations, ensuring uniqueness based on entity names. The link between entities and Delta Modules is maintained in the *delta-entity-link* table. Entries are added when registering a Delta Module through the *SpCreateDeltaModulePresenter* GUI.

The *Delta-Generator-Link* Table tracks the link between Delta Modules and model generators, providing insights into the impact of Delta Modules on generators and vice versa.

The database structure enables a comprehensive understanding of the inter-dependencies of project information. It delineates the scope of a Delta Module to a single Delta Project, influencing various entities. Conversely, an entity

can be impacted by one or more Delta Modules, resulting in the addition or deletion of operations. Multiple Delta Modules may collectively affect a generator, while a single Delta Module can contribute to different model variants through zero or more variants.

The database structure enables a comprehensive understanding of the inter-dependencies of project information. It delineates the scope of a Delta Module to a single Delta Project, influencing various entities. Conversely, an entity can be affected by one or more Delta Modules, resulting in the addition or deletion of operations. Multiple Delta Modules may collectively impact a generator, while a single Delta Module can contribute to different model variants through zero or more variants.

Figure 6.3 presents the structure of the PhaDOP database structure. It highlights each table with the concerted action that is initializing the Delta Project, creating the Delta Module, and applying Delta Modules

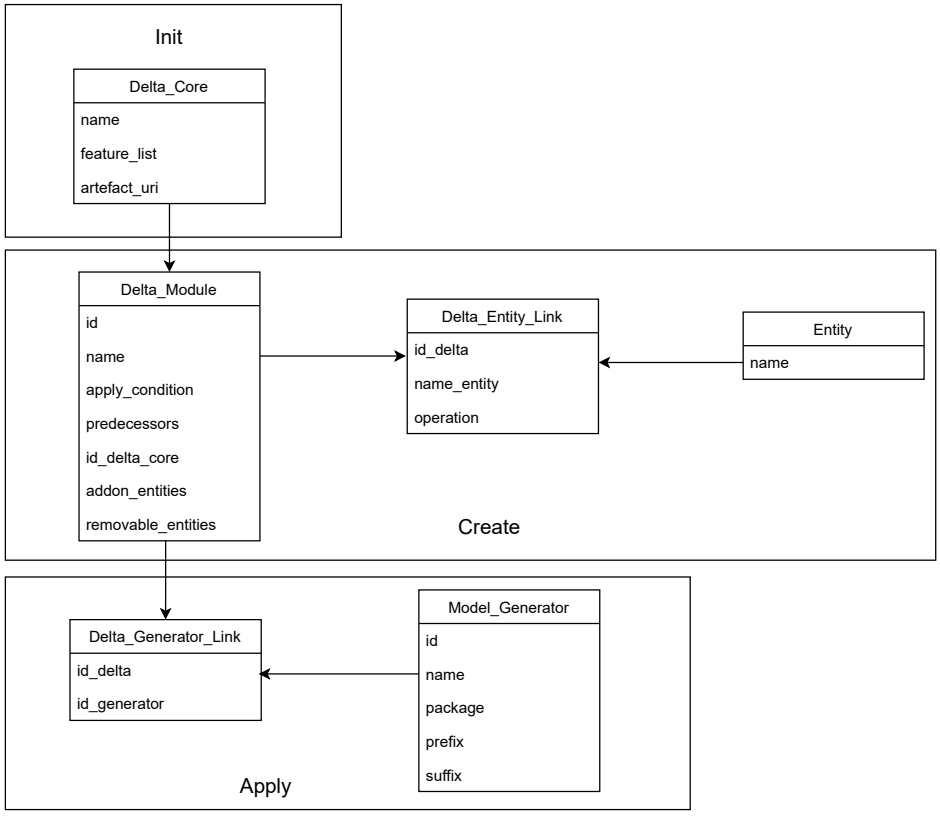


Figure 6.3: Overview of the PhaDOP Framework Database Structure

Delta action manager: manipulates the Delta Module created and applies it for the product derivation. In effect, the user action in the GUI that requests to use a Delta Module retrieves the corresponding Delta Modules and applies them to the target Delta Core generator. The delta manager acts on a copy of the original generator, as we do not want to overwrite the original database.

Delta generator repository: contains each generator version created through the Delta Modules application, i.e., product derivation. Indeed, we do not overwrite. The source and the target version of the generation are tracked in the delta database. This lets us know which Delta Module will enable us to get which generator version and to which version. If necessary, we generate the corresponding models.

Delta Core repository: contains generated metamodels variants from generators; they are UML class diagram models [vdML02]. Indeed, we decided to act on the generator instead of the model itself. Thus, we can generate the corresponding model if we want. The generators have the same structure, including packages, classes, properties, and relations methods. So we can create and apply a common method. Instead, we must use the created function for delta management case by case. For example, we must iterate on a model before removing an entity or its properties.

Reusable artifacts repository a JSON file containing reusable artifacts with method-level granularity. It contains links between model entities and methods with their reusable source code. The class diagram is a structural UML diagram in which the system's behavior, such as the body of methods, cannot be modeled. This means the Delta Core module, the metamodel, cannot capture the product's structure. It, therefore, reduces reusability to the level of architectural commonalities. As a result, the derived product can only generate source code but only the project's structure, such as classes, attributes, inheritance, interfaces, and method signature. We then lose interest in behavioral commonalities. This is the most important thing because the developer must manually rewrite the code without it. JSON links the core and reusable classes to the source code of reusable artifacts. In this way, the source code of reusable methods is stored in one place and can be used by any product.

The Model of the code generator allows the instantiation of the product variant, representing a model obtained through a generator resulting after deriva-

tion. The instantiating consists of giving value to the model properties. The method core is extracted from the Reusable artifact JSON. We must also store the properties' value in the JSON file. Reusable artifacts organization is presented by the metamodel proposed in Chapter 5, Figure 5.9. Once the connector model is instantiated, we can generate the software product's source code. the code generation from the instantiated model rely on the Famix2Java project [5] for the code generation.

6.3 Experimentation and Evaluation

This section provides an in-depth exploration of each component's functionality after introducing the tool's main elements. Practical illustrations are continuously used to enhance understanding and clarity.

To test the effectiveness of the PhaDOP framework, it is being applied to a familiar use case known as the Expression Product Line (EPL). This use case has been previously examined in related research, particularly in the works of DeltaJ and SiPL. DeltaJava implements a Software Product Line (SPL) using the DOP paradigm for Java, while SiPL follows a model-driven approach.

Figure 6.4 depicts the feature model for the EPL, consisting of eight features: three abstract and five concrete. The root feature, *EPL* is an abstract feature at the top level. It has two mandatory abstract child features, *Data* and *Operations*, organizing the features into two distinct categories. Within the *Data* abstract features, three children exist: *Lit*, *Add*, and *Neg*. *Lit* is a mandatory child feature of *Data*, while *Add* and *Neg* are optional. On the other hand, the *Operations* abstract feature includes two children: *Print*, which is mandatory, and *Eval*, which is optional.

In this scenario, the EPL is implemented using a reactive approach. The product line is built from the existing EPL legacy system implemented in Java. Excerpts of this implementation are showcased in Listings 6.3, 6.4, 6.5, and 6.6.

Listing 6.3: Legacy Java Code for the Exp Class

```
48 public class Exp {
49     void print(){ }
50     int eval(){
51         return 0;
52     }
53 }
```

Listing 6.4: Legacy Java Code for the Lit Class

[5] https://github.com/moosetechnology/FAMIX2Java

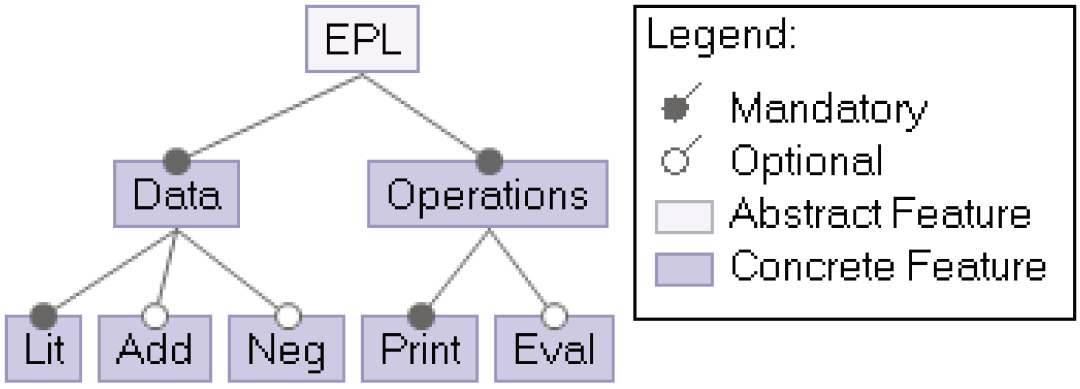


Figure 6.4: Feature model of the Expression Product Line (EPL)

```
54 public class Lit extends Exp {
55     int value;
56     Lit(int n){
57         this.value = n;
58     }
59     void print(){
60         System.out.println(this.value);
61     }
62     int eval(){
63         return this.value;
64     }
65 }
```

Listing 6.5: Legacy Java Code for the Add Class

```
66 public class Add extends Exp {
67     Exp expr1;
68     Exp expr2;
69     Add(Exp expr1 , Exp expr2){
70         this.expr1 = expr1;
71         this.expr2 = expr2;
72     }
73     void print(){
74         this.expr1.print();
75         System.out.print( " + " );
76         this.expr2.print();
77     }
78     int eval(){
79         return this.expr1.eval() + this.expr2.eval();
80     }
81 }
```

Listing 6.6: Legacy Java Code for the Neg Class

```
82 public class Neg extends Exp {
83     Exp expr;
84     Neg(Exp expr){
85         this.expr=expr;
86     }
87     void print(){
88         System.out.print ( " ( - " );
89         this.expr.print();
90         System.out.println ( " ) " );
91     }
92     int eval(){
```

```
93          return (-1) * this.expr.eval();
94      }
95  }
```

Now that we have the legacy system, we can proceed with the PhaDOP Framework.

Creating the Delta Core Two strategies are applicable for implementing a Delta Core [SBB+10]: starting from a *Complex Core* and starting from a *Simple Core*. The *Complex Core* strategy involves creating product variants from complete products, primarily by removing features. The *Simple Core* strategy involves creating product variants from the most basic products, with only mandatory features present.

We have chosen a *Complex Core* strategy to implement the SPL. This means that we create product variants from the most complete products valid. The most significant aspect of this strategy is the removal operations.

The PhaDOP framework utilizes model-based engineering to abstract the Delta Core on the model level. In simpler terms, the EPL metamodel represents the Delta Core as a UML class diagram created from the legacy source code. The Delta Core is produced by using the Moose platform in the Pharo environment and language. We are not going to build the metamodel directly. We are currently focusing on developing a generator to generate a metamodel representing Delta Core. This is a capability that is enabled by the Moose platform.

The Moose platform uses a Pharo class to represent a model generator, which contains methods for creating the metamodel. Like all others, the generator class is created in a Pharo package. In this example, the package name is called *EPL-model-generator*. The generator class is named *EPLModelGeneratorCore* and both are located in the *EPL-model-generator* package. Each class name is an instance variable of the model generator class. You can find the source code of the generator in Listing 6.7.

Listing 6.7: Declaration of the EPL Core Model Generator Pharo Class generator

```
96  FamixMetamodelGenerator subclass: #EPLModelGeneratorCoreOrgn
97      instanceVariableNames: 'exp lit add neg'
98      classVariableNames: ''
99      package: 'EPL-model-generator
```

By default, the Moose metamodel class generator provides two class-side methods and four instance-side methods. The class-side methods include *packageName* for returning the package name in which the model will be generated and *prefix* for providing a string value used for the generated model to prevent ambiguity in entity names.

Listing 6.8 and Listing 6.9 show the source code of the two class-side methods.

Listing 6.8: EPL package name method

```
100 packageName
101     ^ #'EPL-model-generator'
```

Listing 6.9: EPL prefix method

```
102 prefix
103     ^ #'EPL-Core-Orgn'
```

On the instance side, the *defineClasses* method allows the declaration of entities in the Delta Core. For the EPL use case, the required entities include *Exp*, *Lit*, *Add*, and *Neg*. The *defineHierarchy* method establishes inheritance relationships, where *Lit*, *Add*, and *Neg* are designated to have *Exp* as their superclass. The *defineProperties* method specifies the necessary properties for each entity; for instance, *Lit* has *value*, and *Add* entity has *expr1* and *expr2*. The *defineRelations* method indicates relations between entities; however, in the EPL system, entities like *Exp*, *Add*, *Lit*, and *Neg* do not have multiplicity relations.

Listings 6.10, 6.11, 6.12, 6.13 present the methods Pharo code for the four instance-side methids.

Listing 6.10: EPL defineClasses method

```
104 defineClasses
105     super defineClasses.
106     exp := builder newClassNamed: #Exp.
107     lit := builder newClassNamed: #Lit.
108     add := builder newClassNamed: #Add.
109      neg := builder newClassNamed: #Neg.
```

Listing 6.11: EPL defineHierarchy method

```
110 defineHierarchy
111     super defineHierarchy.
112     lit --|> exp.
113     add  --|> exp.
114     neg  --|> exp.
```

Listing 6.12: EPL defineProperties method

```
115 defineProperties
116     super defineProperties.
117     lit property: #value type: #String.
118
119       add property: #expr1 type: #String.
120       add property: #expr2 type: #String.
121
122       neg property: #expr type: #String.
```

Listing 6.13: EPL defineRelations method

```
123 defineRelations
124   "No relation yet"
```

Now that we have the metamodel generator, we can create a Delta Module and begin our project.

6.3.1 Initializing the Delta Project

In the Delta Project initiation stage, the first step is to gather all the necessary prerequisites for the Delta Core. This includes creating the delta database and setting up the required tables based on the architecture shown in Figure 6.3. During this phase, the details of the Delta Core are completed, such as the name of the initial core model, the specifications for the Delta Core module, the list of features to be included or excluded during configuration, and the location of reusable artifacts essential to the project.

To provide the necessary information, we utilize a GUI to create the Delta Project. To begin, we launch the tool through the home presenter and select *New Delta Project* from the submenu. This takes us to the project initialization via the *SpCreateDeltaProjectPresenter* GUI. Within this interface, there are input text fields where the user can enter the required information for persisting in the delta database. The user fills in the database name, Delta Core name, and feature list. A comma should separate each string value.

Figure 6.5 provides a snapshot of the GUI and an illustrative input text example.

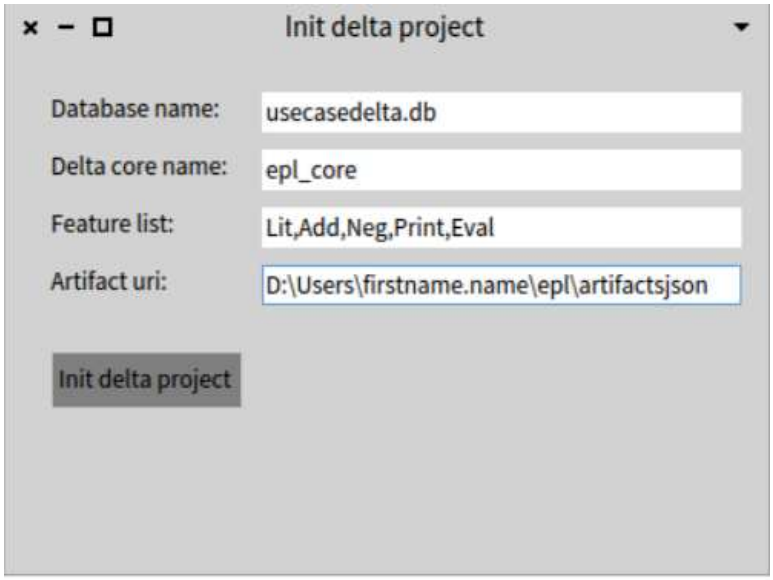


Figure 6.5: Initializing the Delta Project with User-Provided Data

During the initialization stage, the Delta project database is created with all the necessary tables, as shown in Figure 6.3. SQLite is used as the local relational database for this purpose. Various implemented functions enable the storage of relevant information on Delta Core in the corresponding table. This functionality is made possible by the action implemented and the service it invokes. The code for the GUI presenter is shown in Listing 6.14, which invokes the action *initDeltaProject*. The code for the GUI presenter is shown in Listing 6.14, which invokes the action *initDeltaProject*. Additionally, Listing 6.15 contains the code for the *initDeltaProject* action.

Listing 6.14: Code of the init Delta Project action GUI

```
125 initializePresenters
126     fieldDBName := self newTextInput
127         placeholder: 'Name of the delta database ';
128         yourself.
129
130     fieldDeltaCoreName := self newTextInput
131         placeholder: 'Name of the version of the Delta Core ';
132         yourself.
133
134     fieldFeatureList := self newTextInput
135         placeholder: 'List of features ';
136         yourself.
137
138     fieldArtifactUri := self newTextInput
139         placeholder: 'Artifacts location ';
140         yourself.
141
142     buttonInitDeltaProject := self newButton
143       label: 'Init Delta Project';
144        color: Color gray;
145         action:  [ self initDeltaProject]
146       yourself.
```

Listing 6.15: Code of the "initDeltaProject" called when posting the data for Delta Project initialization

```
initDeltaProject

    | connection dbName deltaCoreName featureList artifactUri |
    "Field values"
    dbName := fieldDBName text.
    deltaCoreName := fieldDeltaCoreName text.
    featureList := fieldFeatureList text.
    artifactUri := fieldArtifactUri text.

    "database"
    connection := SQLite3Connection memory.
    connection := SQLite3Connection on:
                      (Smalltalk imageDirectory / dbName) fullName.

    connection open.
    connection
        execute:
        '
        CREATE TABLE "delta_core" (
            "id"    INTEGER NOT NULL,
            "name"  TEXT,
            "feature_list"  TEXT,
            "artifact_uri"  TEXT,
            PRIMARY KEY("id" AUTOINCREMENT)
        );
        '.
    connection
        execute:
        '
        CREATE TABLE "delta_module" (
            "id"    INTEGER NOT NULL,
            "name"  TEXT UNIQUE,
            "apply_condition"   TEXT,
            "predecessors"  TEXT,
            "addon_entities"   TEXT,
          "removable_entities"   TEXT,
            "id_delta_core" INTEGER,
            FOREIGN KEY("id_delta_core") REFERENCES "delta_core"("id"),
            PRIMARY KEY("id" AUTOINCREMENT)
        );
        '.
    connection
        execute:
        '
        CREATE TABLE "variant_generator" (
            "id"    INTEGER NOT NULL,
            "name"  TEXT UNIQUE,
            "package"   TEXT UNIQUE,
            "prefix"    TEXT UNIQUE,
            "sufixe"    TEXT UNIQUE,
            PRIMARY KEY("id" AUTOINCREMENT)
        );
        '.
    connection
        execute:
        '
        CREATE TABLE "delta_variant_link" (
            "id_delta"  INTEGER,
            "id_variant"    INTEGER,
            FOREIGN KEY("id_delta") REFERENCES "delta_module"("id"),
            FOREIGN KEY("id_variant") REFERENCES "variant_generator"("id")
        );
        '.
    connection
        execute:
        '
        CREATE TABLE "entity" (
            "id_entity" INTEGER NOT NULL,
            "name_entity"   TEXT,
            PRIMARY KEY("id_entity" AUTOINCREMENT)
        );
        '.
    connection
        execute:
        '
```

```
223         CREATE TABLE "delta_entity_link" (
224             "id_delta"  INTEGER,
225             "name_entity"   TEXT,
226             "operation" TEXT,
227             FOREIGN KEY("id_delta") REFERENCES "delta_module"("id")
228         );
229         '.
230     "Init Delta Core table"
231     connection
232         execute:
233         'INSERT INTO delta_core(name, feature_list, artifact_uri) VALUES
234         (?1, ?2, ?3);'
235         with: {
236               deltaCoreName .
237                 featureList .
238                 artifactUri}.
239
240     connection close.
241
242     self inform: 'Delta Project succefully initialized'.
243     self inform: 'Delta Core ', deltaCoreName, ' succefully initialized'.
```

Figure 6.6 illustrates the interaction among various tool components during the initiation of a Delta Project.

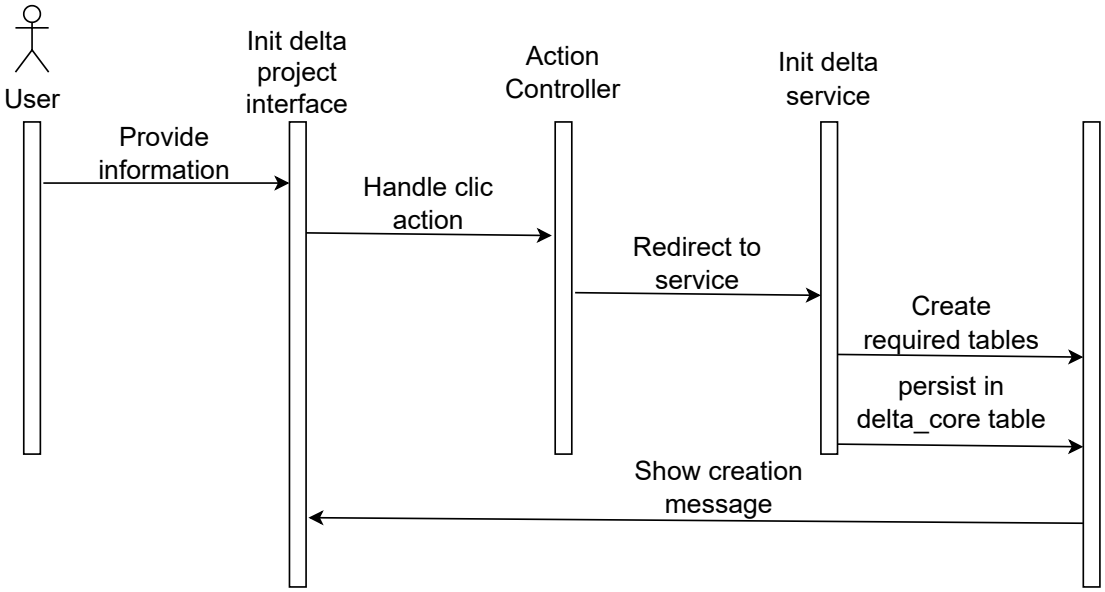


Figure 6.6: Sequence Diagram: Initialization Process of the Delta Project

6.3.2 A Truth Table-Based Methodology to Identifying Entity and Method-Level Granularity Delta Modules

Delta Modules at the entity and method-level granularity refer to changes made at the class level. After initializing the Delta Project with a Delta Core, a list of features, and reusable artifacts, the next step is to prepare Delta Modules

that apply to the Delta Core. Anticipating all potential Delta Modules in advance requires effort, as predicting future additions, such as new entities or methods, can be challenging. Identifying foreseeable changes can enhance the overall process, especially when using model-based engineering. This can increase productivity and make the task more accessible to non-experts.

We propose a methodology for identifying Delta Modules applicable to a Delta Core according to the feature model. The process relies on a known truth table. Indeed, each feature has two possible states for a valid target software product, present or absent. In other words, each feature presence test has a true or false value. The presence of a feature is materialized by a Delta Module that accomplishes an "Add" operation if the corresponding artifact does not exist yet. Inversely, the not present features correspond to the Delta Module that realizes the remove operation if the entity exists. This truth table proposition does not consider the granularity of features at the properties level. Note that the metamodel representing the Delta Core is a reusable entity and attribute-level artifact. Methods are all directly managed from the reusable artifacts repository.

For a truth table with n inputs, where n is the number, there are 2^n possible outputs. We can adapt this to our context by designating features as inputs and valid combinations to create products as outputs.

Table 6.1 presents the truth table corresponding to the EPL. The "+" sign means the feature is present for a given combination materialized by the table truth inputs, and the "-" sign feature is absent. The table shows all the possible configurations at the entity level and method level. Even if the functions are not present in the metamodel, entity and function are linked to the function artifacts reusable located in the reusable artifacts repository.

It is important to note that there are mandatory features that are always present and do not change. These features are not considered as variable inputs in our methodology. Therefore, the total number of inputs equals the number of optional features. The optional features comprise *optional features*, *group features*, and *alternative group features* since each element is occasionally optional. This means that for b, which represents the number of optional features, we have a total of 2^b possible delta modules. The set of possible delta modules has different granularities, namely entity and function. In this context, we focus on the entity granularity.

Table 6.2 displays the truth table of the EPL, which focuses on variable features. Each row in the truth table represents a valid configuration to create an EPL variant. This enables us to identify Delta Modules.

Lit	Print	Add	Neg	Eval
+	+	+	+	+
+	+	+	+	-
+	+	+	-	+
+	+	+	-	-
+	+	-	+	+
+	+	-	+	-
+	+	-	-	+
+	+	-	-	-

Table 6.1: Truth Table for the complete EPL

We have chosen a *Complex Core DOP*, focusing primarily on removal operations. This simplifies both the number and size of the Delta Modules. The table below illustrates the list of delta actions as shown in Table 6.3.

We note a simplification using the *Complex Core*. Indeed, the Delta Module size is reduced, **i.e.,** the number of possible Delta Action, because we do not have operations related to mandatory features. For example, Delta has just one operation, removing the *Eval* method instead of three, as presented in Table 6.3. Once identified, all we have to do is implement the Delta Modules we want.

Table 6.2 is insufficient to show the impact of operation embedded in delta action and difficulties caused by the number of combinations related to the delta module. Indeed, when a configuration outpost gives several Delta Modules with zero or more Delta Action, this does not mean all these Delta Action operations must be applied. Remove actions are accomplished if the entity or method does not already exist. Converting the Add operations is performed if the feature is absent. This means each delta module is itself a set of Delta Action combinations. We can see that the Delta Module itself is a truth table where the Delta Action is input. So, a Delta Module was evaluated thanks to the possible entity-level add and remove Delta Module truth table, Table 6.4.

Print	Add	Neg	Eval
Delta 1	+	+	+
Delta 2	+	+	-
Delta 3	+	-	+
Delta 4	+	-	-
Delta 5	-	+	+
Delta 6	-	+	-
Delta 7	-	-	+
Delta 8	-	-	-

Table 6.2: Possible Delta Modules for Adding and Removing Entities and Methods in the EPL

If we zoom in on a Delta Module in Table 6.4, for example, *Delta 2*, the possible Delta Actions in the second column give $2^n - 1$ versions of the delta module, where n is the number of delta actions. For *Delta 2*, $n = 3$ because we have three delta actions. The -1 is included because a combination remains unused. For example, the composition of Delta Actions*add Add entity, add Neg entity, and remove Eval method* corresponds to the configuration of the feature model in where entities *Add* and *Neg* are present, and method *Eval* is absent.

6.3.3 Delta Module Implementation

Delta Modules are a set of modifications that can be applied to a Core Delta Module. This section aims to create some Delta Modules that can generate valid products. The Core Delta Module is the initial product without any modification. In this case, the Core Delta Module is represented by a metamodel that contains entities extracted from existing source code. Following the *Complex Core* strategy, each entity is initially present in the Core Delta Module. The cardinality between entities is defined using the EPL feature model. Entities of

Delta Module	Possible Delta Action	Possible Delta Action for *Complex Core* strategy
Delta 1	add *Add* entity, add *Neg* entity, add *Eval* method	N.C.
Delta 2	add *Add* entity, add *Neg* entity, remove *Eval* method	remove *Eval* method
Delta 3	add *Add* entity, remove *Neg* entity, add *Eval* method	remove *Neg* entity
Delta 4	add *Add* entity, remove *Neg* entity, remove *Eval* method	remove *Neg* entity, remove *Eval* method
Delta 5	remove *Add* entity, add *Neg* entity, add *Eval* method	remove *Add* entity
Delta 6	remove *Add* entity, add *Neg* entity, remove*Eval* method	remove *Add* entity, remove *Eval* method
Delta 7	remove *Add* entity, remove *Neg* entity, add *Eval* method	remove *Add* entity, remove Neg entity
Delta 8	remove *Add* entity, remove *Neg* entity, Remove *Eval* method	remove *Add* entity, remove *Neg* entity, remove *Eval* method

Table 6.3: Possible entity-level add and remove Delta Module for the EPL

a mandatory feature have a minimum cardinality of 1, while the minimum cardinality for an entity of a mandatory feature is zero. The EPL core module is presented in Figures 6.7.

The Delta Module organizes reusable artifacts for entities using JSON objects. Method implementations are linked to corresponding classes when applying the Delta Module. This allows for the addition and removal of methods as specified. Class source codes are grouped as key-value pairs. The provided example in Listing 6.16 demonstrates reusable artifacts extracted from a Delta Module.

Listing 6.16: Example of delta repository stored in JSON file

Delta Module	Possible Delta Action
Delta 2-1	add *Add* entity
Delta 2-2	add *Neg* entity
Delta 2-3	remove *Eval* method
Delta 2-4	add *Add* entity, add *Neg* entity
Delta 2-5	add *Add* entity, remove *Eval* method
Delta 2-6	add *Neg* entity, remove *Eval* method
Delta 2-7	add *Add* entity, add *Neg* entity, remove *Eval* method
Delta 2-8	N.C

Table 6.4: Possible entity-level add and remove Delta Module for the EPL

```
{
    "Exp": {
        "targetSourceLocation":"D:\\Users\bouboutHiam.niang\\workspace\\
             epl_legacy_dop_tool_demo\\ExpressionProductLineGeneratedNew\\src",
        "methods": [
            {
            "name": "print",
            "sourceEnchor": "{ //Comment:nothing yet }",
            "parameters": [
                {}
                ]
            }
            ],
        "parent": {},
        "interface": [
            {}
        ]
    },
    "Lit": {
        "targetSourceLocation":"D:\\Users\bouboutHiam.niang\\workspace\\
             epl_legacy_dop_tool_demo\\ExpressionProductLineGeneratedNew\\src",
        "methods": [
            {
            "name": "Lit",
            "sourceEnchor": "{ this.value=n; }",
            "parameters": [
                {
                    "name": "value",
```

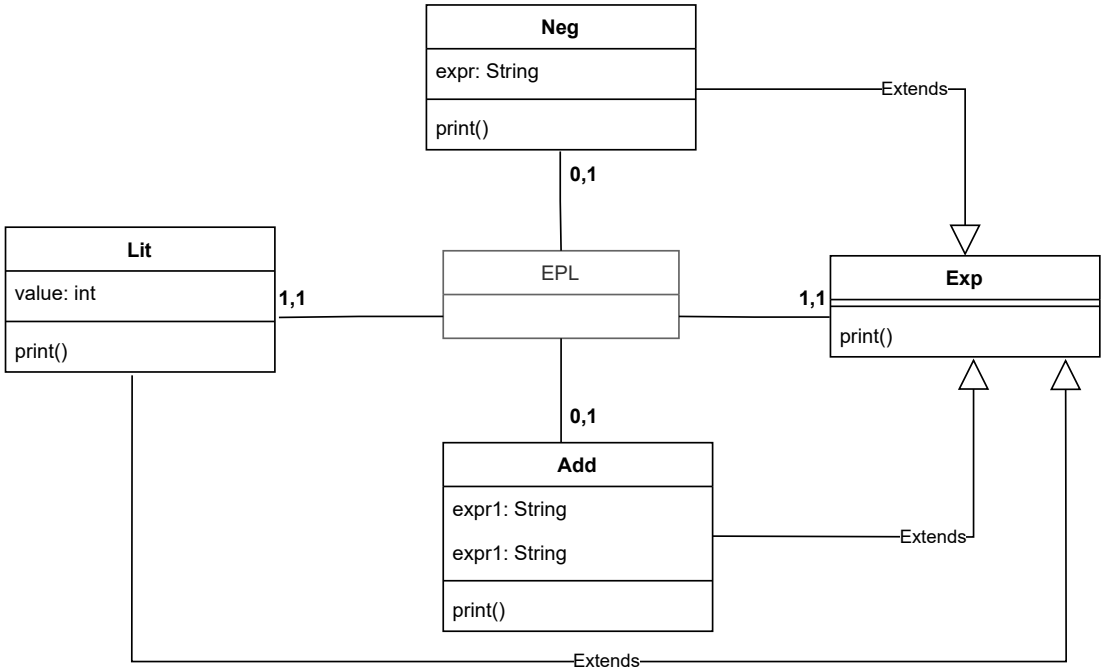


Figure 6.7: Core model of the EPL

```
270                     "type":"int"
271                 }
272                 ]
273             },
274             {
275             "name": "print",
276             "sourceEnchor": "{ System.out.println(this.value); }",
277             "parameters": [
278                 {}
279                 ]
280             }
281         ],
282         "parent": {
283             "name":"Exp"
284         },
285         "interface": [
286             {}
287         ]
288     },
289     "Add": {
290         "targetSourceLocation":"D:\\Users\bouboutHiam.niang\\workspace\\
              epl_legacy_dop_tool_demo\\ExpressionProductLineGeneratedNew\\src",
291         "methods":
292             [
293             {
294             "name": "Add",
295             "sourceEnchor": "{ this.expr1 = a ; this.expr2 = b ; }",
296             "parameters": [
297                 {
298                     "name": "expr1",
299                     "type":"Exp"
300                 },
301                 {
302                     "name": "expr2",
303                     "type":"Exp"
```

```
                    }
                    ]
                },
                {
                "name": "print",
                "sourceEnchor": "{  this.expr1.print();System.out.print( \" + \" ); this.expr2.
                    print(); }",
                "parameters": [
                    {}
                    ]
                }
                ],
            "parent": {
                "name":"Exp"
            },
            "interface": [
                {}
            ]
        }
}
```

We begin by setting up the Delta Project. We do this by initializing the Delta Core Module and reusable artifacts. After that, we create a Delta Module to modify the Delta Core Module version. This is done to create derived products. To accomplish this, we use a scenario that specifies which Delta Module we want based on the application condition and order. The process involves several steps.

The first step in the scenario is with a product with a *Lit* and *Add* entities, each with the *Print* method. We create a delta *DLitAdd* module to modify the core module. Knowing we adopted a *Complex Core* strategy, this Delta Module has evolved to remove optional *Neg* features.

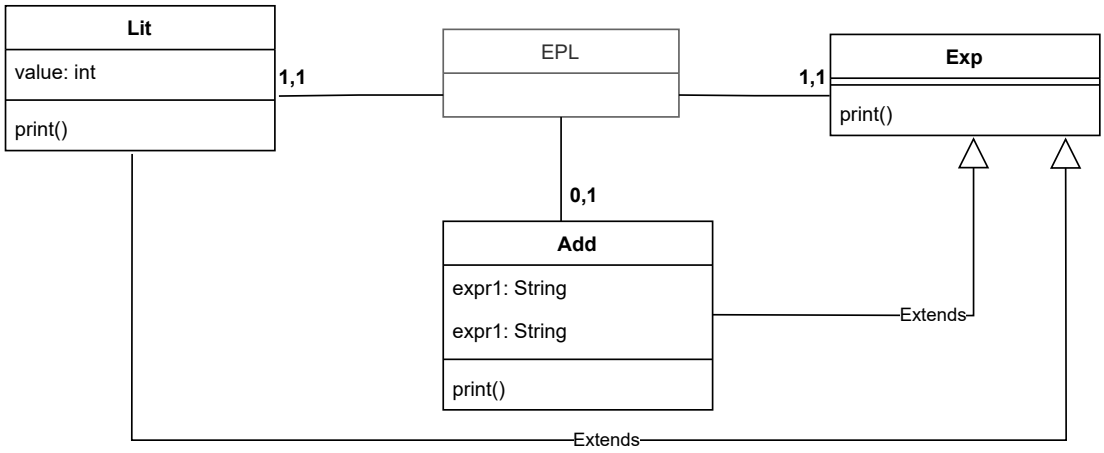


Figure 6.8: Target variant metamodel of when applying the DLitAdd delta model on the EPL core module

The artifacts that can be reused for this delta module are also stored in the artifacts repository, which can be referred to using the reference number 6.16.

Another product version is desired in the second step, building upon the modifications introduced by the *DLitAdd* Delta Module. This modification involves adding the Eval method to both Lit and Add entities. A new Delta Module, *DEvalLitAdd*, is created, with a dependency on the *DLitAdd* module. Notably, applying *DEvalLitAdd* before *DLitAdd* would revert to the original core model. Figure 6.9 showcases the resulting model.

Furthermore, it has been demonstrated that Delta Modules are reusable, as shown by their application order and dependencies. Each class in these modules implements a variant of the *Eval* method, and the associated method-level reusable artifacts are stored in JSON files. To manage dependencies, the JSON file includes fields that specify the order of Delta Modules, separated by commas when applicable.

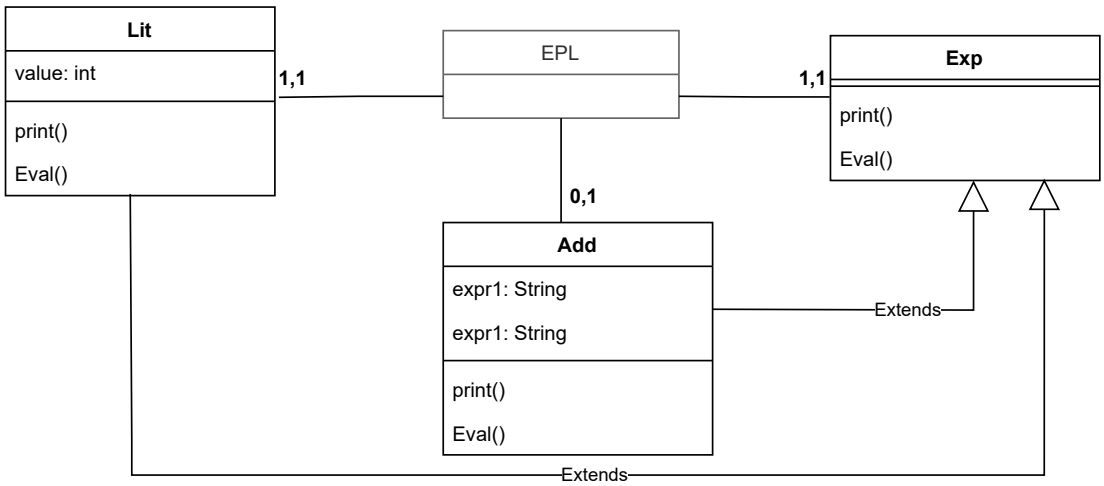


Figure 6.9: The EPL Delta Module: DEvalLitAdd

The JSON file that contains reusable artifacts is structured following the metamodel proposed in Chapter5, Figure 5.9.

Regarding the initial Delta Modules, the reusable artifacts linked to the DEvalLitAdd Delta Module are also archived in the artifacts repository, as depicted in 6.16.

However, for having the wanted product, we could also use it. Instead, it has two dependent Delta Modules. It is possible to use a single Delta Module to go from the original Delta Core and remove one of the Neg entities to have only the Lit, Add with print method for each class, and at the same time introduce the "Eval" method in both Lit and Add class. Let us call this Delta

Module DBigEvalLitAdd. The "DBigEvalLitAdd" will contain the same changes as the first Delta Module, DLitAdd, plus adding a new method Eval. Thus, this is a significant Delta Module with already existing modifications. This needs to emphasize the reusability of the Delta Module itself. The Delta Module management could be more convenient with the reuse of Delta. This is our criticism of DeltaJ afterward, not that Delta Modules can not be reused but that code-level management without MDE could be time-consuming.

To summarize, to have the wanted product, we can choose the second Delta Core and use a big Delta Module, *DBigEvalLitAdd*, or reuse a first Delta Module, *DLitAdd*, composing with a small Delta Module, *DEvalLitAdd*. So specify delta relation, *DLitAdd* after *DEval*, even if we prefer the later choice. We present the Delta Module for the first case. Having an initialized Delta Project with the Delta Core and reusable artifacts, we create a Delta Module to modify the core model version for derived product creation. We use the scenario describing which Delta Module we want following an application condition and their application order. Here are the different steps of the scenario:

The reusable artifacts for the DBigEvalLitAdd Delta Module are stored in the artifacts repository, as referenced in 6.16. Users can choose between a single large Delta Module or a sequence of smaller Delta Modules.

The objective of the third scenario is to create a product that includes the *Neg* entity with the *Print* and *Eval* methods but without the *Add* entity. The *Lit* entity should maintain both the *Print* and *Eval* methods. To accomplish this, we introduce the *DLitNeg* Delta Module, which excludes the 'Add' entity from the Delta Core.

The original Delta Core Module already includes the *Neg* entity, but additional modifications are required to achieve the desired product. Removing the *Add* entity from the original core produces a model variant with the *Neg* entity but without the *Add* entity. However, this action results in the loss of the *Eval* method in the *Lit* entity. To address this issue, we propose applying the Delta Module *DEvalLitAdd* before *DEvalLitNeg*. The original core model is modified by introducing the *Eval* method in both the *Add* and *Lit* entities, followed by the removal of the *Add* entity.

The challenge arises from the fact that the *DEvalLitNeg* module does not have a *Neg* entity, as it relies on the *DLitAdd* module, which removes the *Neg* entity. Therefore, the only viable solution is to transition from the *DEvalLitAdd* module by adding the *Neg* entity. We prefer a complex core strategy where removal takes precedence over addition, so we choose a Delta Module that removes the *Add* entity from the original core module while adding evaluation methods

to both the *Lit* and *Neg* entities. This module is called *DEvalLitNeg* and is similar to *DBigEvalLitAdd* but with *Add* replaced by *Neg*. Another approach is to first implement a Delta Module that removes *Add* (*DLitNeg*) and then introduce *DEvalLitNeg*, which depends on *DLitNeg* and adds evaluation methods to the *Lit* and *Neg* entities.

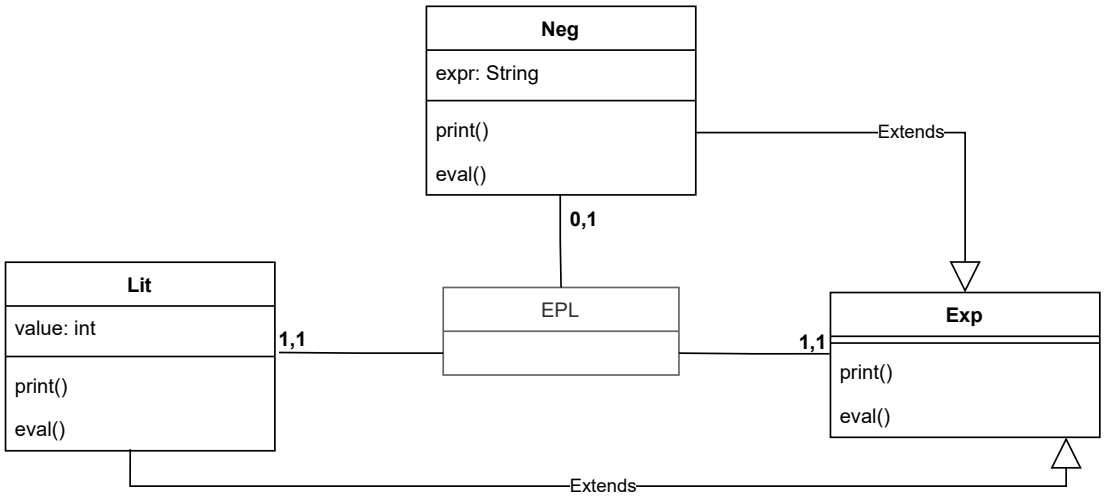


Figure 6.10: The EPL Delta Module: DEvalLitNeg

Similarly to the previous Delta Module, the repository includes reusable source code relevant to this Delta Module, as referenced in 6.16.

Delta Module is created using the *SpCreateDeltaModulePresenter* started by selecting the *Create new Delta Module* option in the submenu available from the main present. Users input details regarding the relevant database and the name of the Delta Module while specifying its application condition. This condition is based on the feature list within the Delta Core table. Additionally, users specify the list of entities for addition or removal. The Delta Core ID precisely designates the involved Delta Project. Refer to Figure 6.11 for a visual representation of the *DEvalLitAdd* Delta Module creation interface.

The Delta Module creation process involves adding user-provided information to the corresponding table in the embedded delta database, as shown in Figure 6.3. The code for the *SaveDeltaModule* action is provided in Listing 6.17.

Listing 6.17: Code of the "SaveDeltaModule" called when posting the data for creating a Delta Module

```
323 saveDeltaModule
```

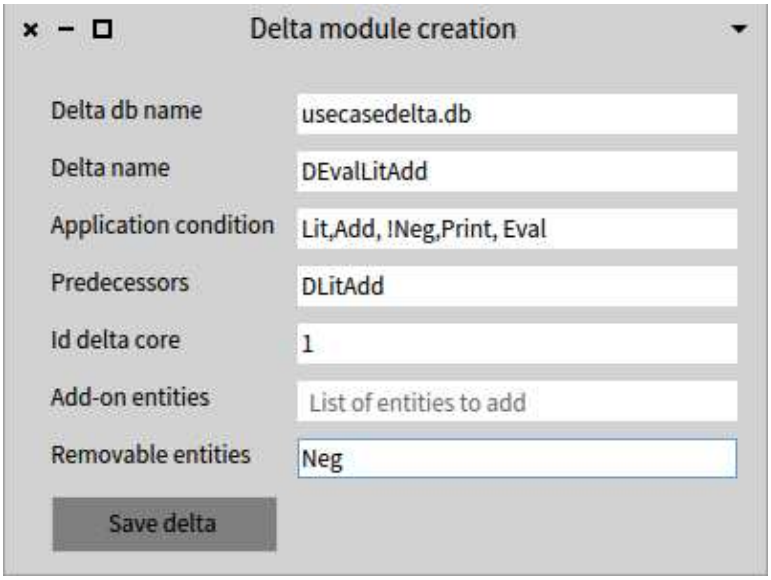


Figure 6.11: Reusable Artifacts of the Delta Module *DEvalLitNeg* in the EPL Artifacts

```
324
325     | connection dbName name applyCondition predecessors idDeltaCore addonEntities removableEntities
            idDeltaModule tabAddonEntities tabRemovableEntities|
326     "Field values will become dto"
327     dbName := fieldDbName text.
328     name := fieldDeltaName text.
329     applyCondition := fieldApplyCondition text.
330     predecessors := fieldPredecessors text.
331     idDeltaCore := fieldIdDeltaCore text.
332     addonEntities := fieldAddonEntities text.
333     removableEntities := fieldRemovableEntities text.
334
335     "database"
336     connection := SQLite3Connection memory.
337     connection := SQLite3Connection on:
338                       (Smalltalk imageDirectory / dbName) fullName.
339
340     connection open.
341
342     connection
343         execute:
344         'INSERT INTO delta_module(name, apply_condition, predecessors, id_delta_core, addon_entities,
                removable_entities) VALUES (?1, ?2, ?3, ?4, ?5, ?6);'
345         with: {
346               name.
347                 applyCondition.
348                 predecessors.
349                 idDeltaCore.
350                 addonEntities.
351                 removableEntities
352                 }.
353
354     "Create entity and delta link"
355
356     idDeltaModule := ((connection execute: 'Select id from delta_module where name=?' with: {name}) next) at
            :'id'.
357    "Todo create function for the two case duplication"
358     tabAddonEntities := addonEntities splitOn: ','.
359     tabAddonEntities do: [ :each |
```

```
360         |entity|
361
362     entity := (connection execute: 'Select name_entity from entity where name_entity=?' with: {each}) next.
363     "if entity not alredy exist"
364      entity ifNil: [
365         each ifNotEmpty: [
366             connection
367                 execute:
368                     'INSERT INTO entity(name_entity) VALUES (?1);'
369                         with: {
370                                 each
371                                 }.
372                                 ]
373
374         ].
375        "Create link between Delta Module and entity"
376     each ifNotEmpty: [
377             connection
378                 execute:
379                     'INSERT INTO delta_entity_link(id_delta, name_entity, operation) VALUES (?1, ?2, ?3);'
380                         with: {
381                                 idDeltaModule.
382                                 each.
383                                 'ADD'
384                                 }.
385                                 ]
386 ].
387
388 "Todo create function for the two case duplication"
389     tabRemovableEntities := removableEntities splitOn: ','.
390     tabRemovableEntities do: [ :each |
391         |entity|
392     entity := (connection execute: 'Select name_entity from entity where name_entity=?' with: {each}) next.
393     "if entity not alredy exist"
394      entity ifNil:  [
395         each ifNotEmpty: [
396             connection
397                 execute:
398                     'INSERT INTO entity(name_entity) VALUES (?1);'
399                         with: {
400                                 each
401                                 }.
402                                 ]
403
404         ].
405
406        "Create link between Delta Module and entity"
407        each ifNotEmpty: [
408             connection
409                 execute:
410                     'INSERT INTO delta_entity_link(id_delta, name_entity, operation) VALUES (?1, ?2, ?3);'
411                         with: {
412                                 idDeltaModule.
413                                 each.
414                                 'REMOVE'
415                                 }.
416                                 ]
417 ].
418
419     connection close.
420
421     self inform: 'Delta Module ', name, ' succefully created'
```

Figure 6.6 illustrates the sequence diagram that delineates the interactions among the involved components during creating a Delta Module.

6.3.4 Visualize Delta Modules

In a comprehensive system, the demand for Delta Modules could reach hundreds, as outlined in Table 6.2. While model-based engineering simplifies the handling of Delta Modules compared to working at the code level, effective man-

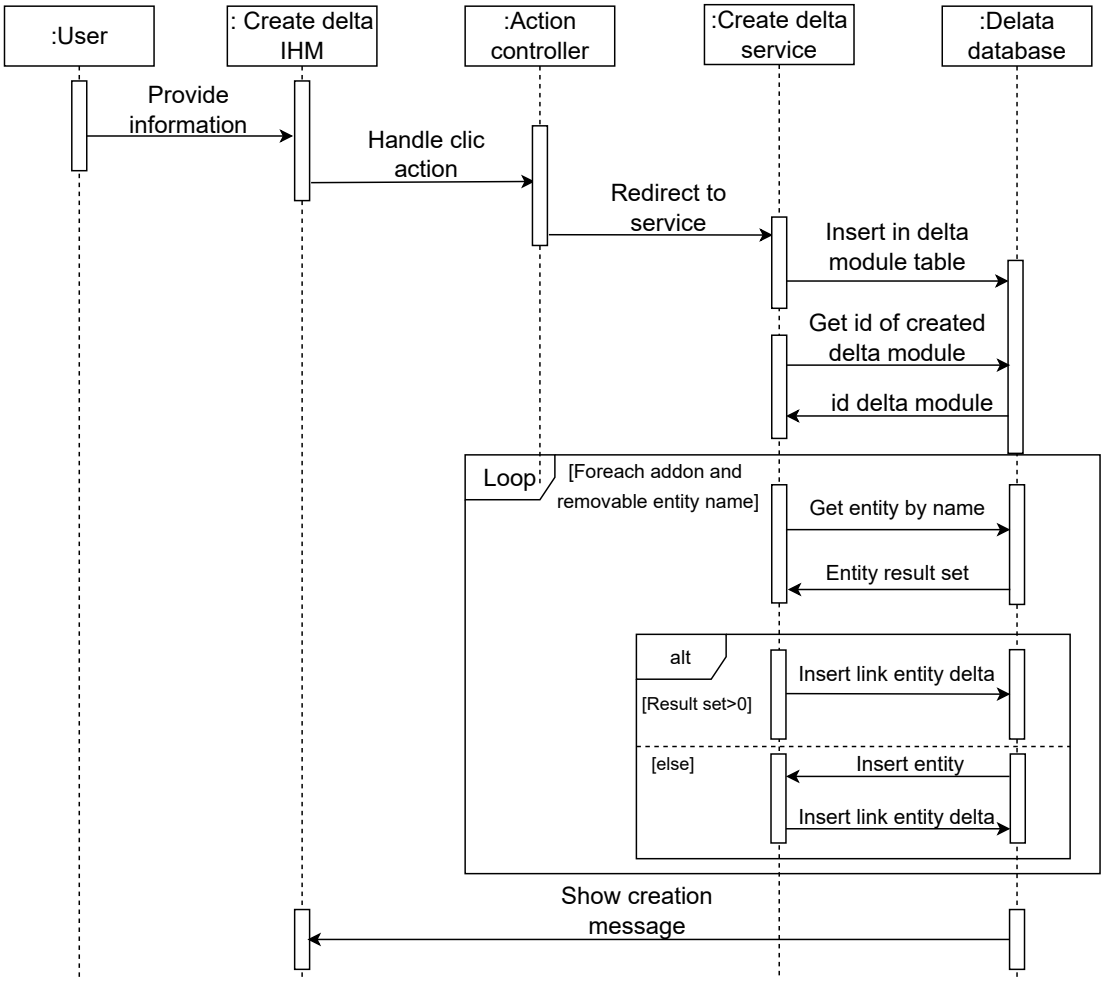


Figure 6.12: Sequence Diagram for Delta Module Creation

agement on such a large scale might require advanced functionalities, especially for individuals not well-versed in the field.

This section underscores the functionality within our framework for visualizing Delta Modules, allowing us to showcase their interdependencies. The visualization feature leverages Roassal [Ber22], a Pharo library. Figure 6.13 provides an overview of the visualization of the Delta Module table.

In a comprehensive system, the demand for Delta Modules could reach hundreds, as outlined in Table 6.2. While model-based engineering simplifies the handling of Delta Modules compared to working at the code level, effective man-

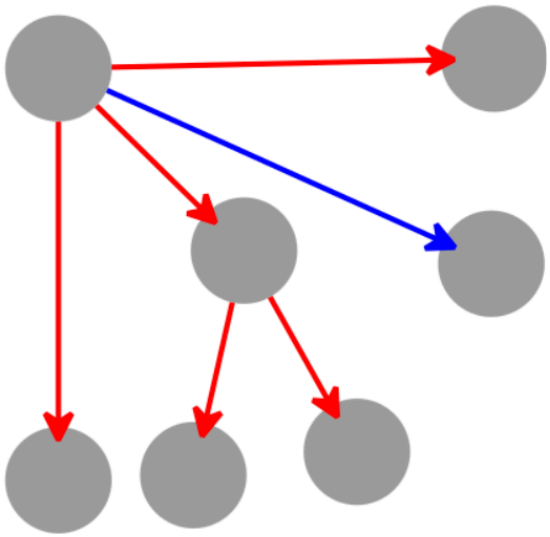


Figure 6.13: Delta Modules dependencies visualization

agement on such a large scale might require advanced functionalities, especially for individuals not well-versed in the field.

This section underscores the functionality within our framework for visualizing Delta Modules, allowing us to showcase their interdependencies. The visualization feature leverages Roassal [Ber22], a Pharo library. Figure 6.13 provides an overview of the visualization of the Delta Module table.

6.3.5 Apply Delta Modules - Product derivation

This section explains how the PhaDOP applies a Delta Module for product derivation. We consider the Delta Module DEvalLitEval introduced in Section 6.3.3. It is about a Delta Module that removes the *Neg* from the Delta Core module and adds the *Eval* method to the "Lit" and "Add" entities. This Delta Module is created using the following configuration: *Lit, Add,!Neg, Print, Eval*. The symbol *!* means the entity or method is absent for the chosen configuration. This corresponds to the Delta Module presented in the database. A single Delta Module accomplishes the modification. Adding Eval to "Lit" and "Add" will be in the artifact repository JSON file.

To apply a Delta Module, we open the interface *SpApplyDeltaModulePresenter* after choosing the option "Apply Delta Module" from the sub-menu of the tool home interface. Then, the user enters the required information: the database

name, the Delta Module name, the generator that must be modified, the package name of the original generator, the prefix for differentiating entity names, and the suffix that will be appended to the original package name. Figure 6.14 give an overview of the delta application user interface.

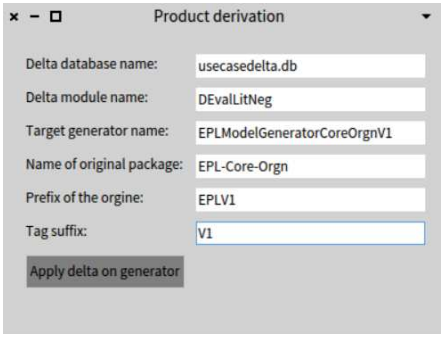


Figure 6.14: Delta Module application GUI

During validation, the system will extract information from the database and subsequently apply these changes to the target model generator. The validation action code is presented in Listing 6.18.

Listing 6.18: Code of the "SaveDeltaModule" called when posting the data for creating a Delta Module

```
applyDeltaModule

     | connection  dbName deltaName generatorName generatorClassName generatorClass entitiesOperationsLink
          packageName prefix suffix deltaModule deltaActionManager sourceNameDico retrievedGenerator|

    dbName := fieldDbName text.
    deltaName := fieldDeltaName text.
    generatorName := fieldGeneratorName text.
    packageName := fieldPackageName text.
    prefix := fieldPrefix text.
    suffix := fieldSuffix text.

    connection  := SQLite3Connection memory.
    connection := SQLite3Connection on:
                    (Smalltalk imageDirectory / dbName) fullName.

    connection open.

    deltaModule := (connection execute: 'Select id from delta_module where name=?' with: {deltaName}) next.
    entitiesOperationsLink := (connection execute: 'Select name_entity, operation from delta_entity_link
         where id_delta=?' with: {(deltaModule at:'id')})"next".

    "generatorClassName := deltaModule at:'generator'."
    generatorClass := Smalltalk classNamed: generatorName.
    "entityName := deltaModule at:'entity'."
    "packageName := deltaModule at:'package'."
    "prefix := deltaModule at:'prefix'."
    "suffix := deltaModule at:'suffix'."

    deltaActionManager := DeltaActionManager new.

    "Each entity delta link"
    (entitiesOperationsLink rows) do:[:row |
        deltaActionManager modifyGeneratorInstanceSideForDelta: generatorClass entitiesOperationsLink: row.
```

```
        ].

    sourceNameDico := Dictionary new.
    sourceNameDico at: 'packageName' put: packageName.
    sourceNameDico at: 'prefix' put: prefix.

    deltaActionManager modifyGeneratorClassSideForDelta: generatorClass sourceNameDico: sourceNameDico
         varianteSufixe:  suffix.

    "After applying we deltas modules we save link with generator in db"
    connection
        execute:
        'INSERT INTO variant_generator(name, package, prefix, sufixe) VALUES (?1, ?2, ?3, ?4);'
        with: {
              generatorName.
                packageName.
                prefix.
                suffix
                }.

   "Create link betweeen generator and Delta Module"
   retrievedGenerator := (connection execute: 'Select id from variant_generator where name=?' with: {
         generatorName}) next.
    connection
        execute:
        'INSERT INTO delta_variant_link(id_delta, id_variant) VALUES (?1, ?2);'
        with: {
              (deltaModule at:'id').
                (retrievedGenerator at:'id')
                }.

    connection close.


    self inform: 'Delta Module', deltaName, ' succefully applied'.
```

The program navigates through several framework components, as depicted in the sequence diagram in Figure 6.15.

The Delta Module is applied by modifying the model generator's methods on both the class and instance sides. The updated generator is then stored in the database, and a relationship between the Delta Module and the generator is established in the corresponding table. This allows tracking which Delta Modules have been applied to the generator and which generators are impacted by a particular Delta Module. This knowledge is essential for effective system management, particularly when assessing the impact of removing a Delta Module.

Once the Delta Module has been applied, the next step is to generate the derived variant using the dedicated GUI provided by the framework, called the *SpModelGeneratorPresenter*. The GUI for generating a model from a metamodel is shown in Figure 6.16.

Listing 6.19 the code corresponding to the action component called when posting the model generation request.

Listing 6.19: Code of "generateModel" for Generate the product model from variant metamodel

```
generateModel: metamodelName
   |class|
    class := (Smalltalk classNamed: metamodelName text).
    class generate.
    self inform: 'Model successfully generated'.
```

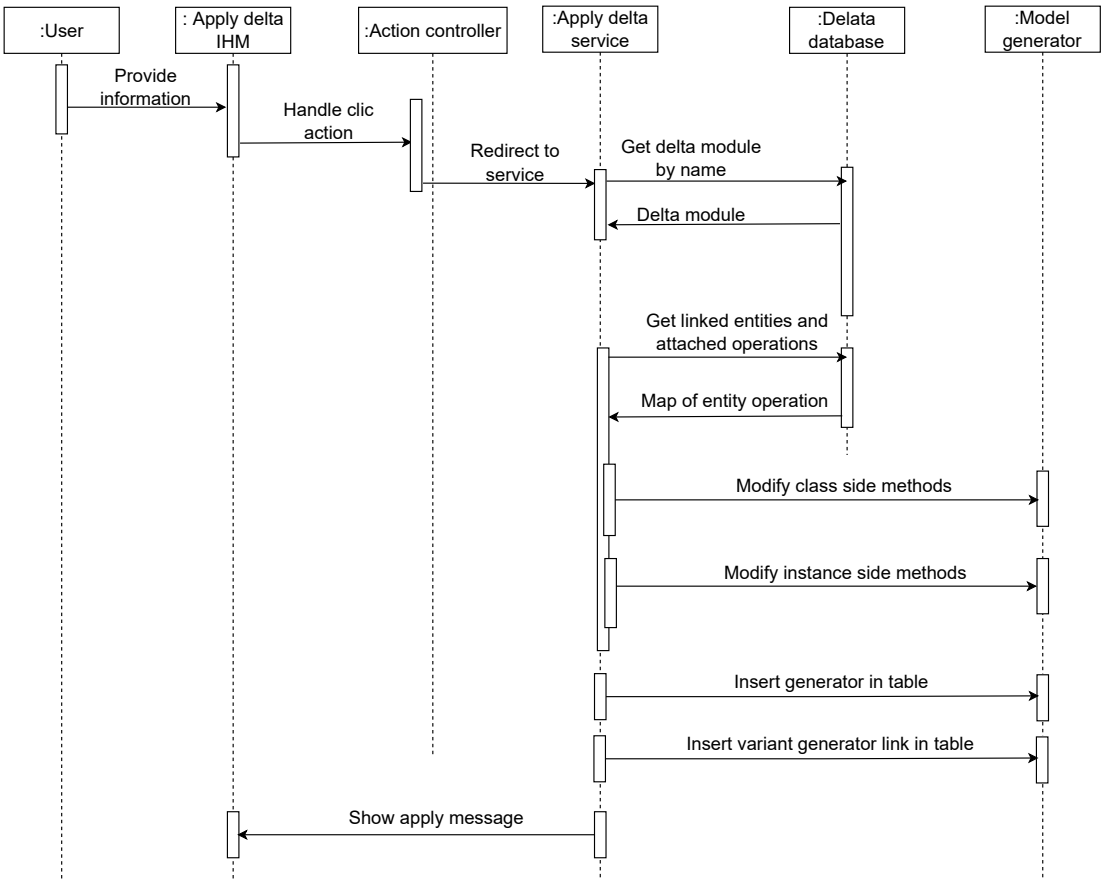


Figure 6.15: Sequence diagram for Delta Module creation

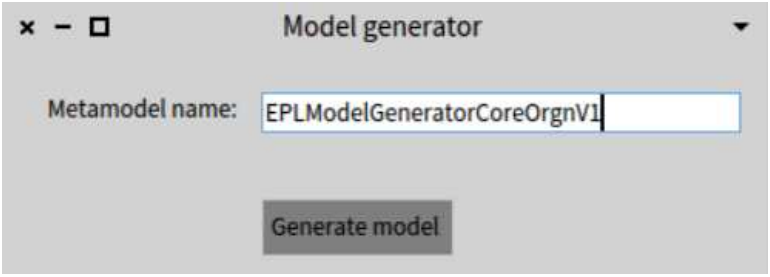


Figure 6.16: Graphical User Interface (GUI) for Model Generation

The subsequent step involves model-to-code transformation after the generated model has been obtained.

6.3.6 Generation of Product Source Code - Application Engineering

This section is dedicated to generating code from the obtained model. PhaDOP provides functionalities for generating Java code, which can be accessed through a dedicated GUI called the 'SpModelToCodePresenter' interface. This interface can be accessed after a selection in the initial interface. Figure 6.17 provides an overview of the tool and the necessary information.

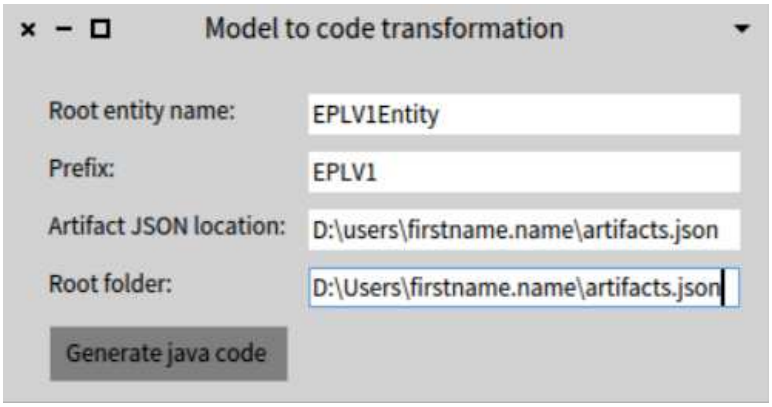


Figure 6.17: Transformation from Model to Code - EPL Variant Code Generation from the Core Delta Module

The program utilizes user-provided data, such as the entity name, prefix (which is removed for processing), location of reusable artifacts, and root folder for source code generation. It then iterates through model entities and creates a class for each entity based on the provided information. However, as the class diagram only captures the static model and excludes behavioral aspects, the program utilizes reusable artifacts stored in a JSON file. A dictionary establishes connections between classes and their corresponding artifacts to bridge the gap. The metamodel shown in Figure 5.9 defines the structure of reusable artifacts. An iterative process is used to extract methods for each class.

We designate the root location for generating the respective class for every class. The program iterates through the methods, setting the method parameters by further iterating through each method's parameters. Each method contains a source code segment that encapsulates the method's body.

Listing 6.20 provides the action code for generating Java source code based on the root model entity name.

Listing 6.20: Source code of the "generateJavaFromDeltaCore" action that export the model to java source code

```
generateJavaFromDeltaCore
    | rootEntityName prefix artifactsLocaton rootFolder entityList artefactsDictionary visitor|

    rootEntityName := fieldRootEntityName text.
    prefix := fieldPrefix text.
    artifactsLocaton := fieldArtifactsLocaton text.
    rootFolder := fieldRootFolder text.

    "Todo funtion: prefixe, class name, Model Name location"
entityList := (Smalltalk classNamed: rootEntityName) allSubclasses.

"targetSourceLocation:= 'D:\Users\boubouthiam.niang\workspace\epl_legacy_dop_tool_demo\
     ExpressionProductLineGeneratedNew\src'."

artefactsDictionary:= artifactsLocaton asFileReference
    readStreamDo: [ :readStream |
        (NeoJSONReader on: readStream) next ].

"Attribut"
entityList do: [ :class |
    |st c m package componentAnnotation componentAnnotationInstance getAnnotation getAnnotationInstance
         parentClass methodArray targetSourceLocation|

    st := FamixJavaClass new.
    st name: 'String'.
    c := FamixJavaClass new.
    c name: (class name copyFrom:prefix size + 1 to:class name size).

    class instVarNames do: [ :var |
        |currentAttribut|
        currentAttribut := FamixJavaAttribute new.
        currentAttribut name: var.
        currentAttribut declaredType: st.

        c addAttribute: currentAttribut.
        ].

    "Method"
    m := FamixJavaMethod new.
    "Get linked artefacts for current class"
    methodArray := (artefactsDictionary includesKey: c name) ifTrue: [ (artefactsDictionary at: c name) at:'
         methods']
    ifFalse: [ OrderedCollection  new.].
    methodArray do:[ :method |
              |arrayParam paramTmp|

               m := FamixJavaMethod new.
                m name: (method at:'name').
                m sourceAnchor:
                  (FamixJavaSourceTextAnchor new source:
                       (method at: 'sourceEnchor')).
                "1 halt."
                m parentType: c.
                m declaredType: st.
                arrayParam := OrderedCollection  new.
                (method at:'parameters') do: [ :p |
                    |param paramType|
                    "1 halt."
                    p ifNotEmpty: [
                        param := FamixJavaParameter new.
                        paramType := FamixJavaClass new.
                     paramType name: (p at:'name').
                        param declaredType: paramType.
                        arrayParam add:param.
                    ]
                ].

                "Multi parameter sort problem in Famix2Java (why param sourceAnchor?)"
                "arrayParam ifNotEmpty: [ m parameters: arrayParam]."

                "Annotation methhod (not key value yet)"
              getAnnotation := FamixJavaAnnotationType new name: 'MethodAnnotation'.
              getAnnotationInstance := FamixJavaAnnotationInstance new annotationType: getAnnotation.

              m annotationInstances add: getAnnotationInstance.
```

```
              "1 halt."
               c addMethod:  m.

            ].



    componentAnnotation := FamixJavaAnnotationType new name: 'ComponentAnnotation'.
    componentAnnotationInstance := FamixJavaAnnotationInstance new annotationType: componentAnnotation.
    c annotationInstances add: componentAnnotationInstance.
    "Inheritance (real parentclass?)"
    parentClass := FamixJavaClass new
       name: 'ParentClass';
        parentPackage: package;
        yourself.

    c
        addSuperInheritance:
            (FamixJavaInheritance new
                subclass: c;
                superclass: parentClass).

    targetSourceLocation := (artefactsDictionary at:c name) at:'targetSourceLocation'.


    visitor := FAMIX2JavaVisitor new.
    "visitor rootFolder: targetSourceLocation asFileReference."
    visitor rootFolder:'D:\Users\boubouthiam.niang\workspace\epl_legacy_dop_tool_demo\
         ExpressionProductLineGeneratedNew\src' asFileReference.
    c accept: visitor.
        ].
```

The code generation action facilitates obtaining the source code classes for the anticipated variant of the EPL system. To achieve this, we partially rely on Famix2Java [6], a visitor designed for exporting FamixJava [Tic99] models to Java code. Specifically, it manages the transition from constructed classes to Java files. Listings 6.21,6.22, and6.23 showcase the generated classes.

Listing 6.21: Generated class Exp

```
@ComponentAnnotation
public class Exp extends ParentClass {
    @MethodAnnotation
    String print() {
     //Comment:nothing yet
    }
}
```

Listing 6.22: Generated class Lit

```
@ComponentAnnotation
public class Add extends ParentClass {
    String expr1;
    String expr2;

    @MethodAnnotation
    String Add() {
     this.expr1 = a ; this.expr2 = b ;
    }

    @MethodAnnotation
    String print() {
    {  this.expr1.print();System.out.print( " + " ); this.expr2.print();
    }

}
```

[6] https://github.com/moosetechnology/FAMIX2Java

	generated	Manual completion
Class	x	-
Method signature	x	-
Method parameters	-	x
Method body	x	-
Class annotation	x	-
Method annotation	x	-
Package import	-	x
Implement interface	-	x

Table 6.5: Table showing what can be generated and what is done manually

Listing 6.23: Generated class Add

```
@ComponentAnnotation
public class Add extends ParentClass {
    String expr1;
    String expr2;

    @MethodAnnotation
    String Add() {
     this.expr1 = a ; this.expr2 = b ;
    }

    @MethodAnnotation
    String print() {
    {  this.expr1.print();System.out.print( " + " ); this.expr2.print();
    }

}
```

The code generated meets our expectations by successfully creating classes, attributes, methods, constructors, and annotations compared to existing systems. Although the model-to-code engine requires improvement in parameter generation, importing classes remains a pending task. However, this issue is not critical as the remaining code to be completed is minimal. Table 6.5 summarizes the artifacts that can be generated or completed manually using the current version of the tool.

6.4 Discussion

The proposed tool takes advantage of working at the model level. This simplifies the delta management compared to code. However, the passage from the model to code after derivation is sometimes challenging for the model-based approach. To this day, we can generate Java code. After instantiating our resulting model, we use a project called Famix2Java to parse the code to Java. The transformation in another language requires another parser. Concerning the derivation process, we use a core complex strategy and focus on removing instead of adding. This can be a limitation for evolving software product lines because we must create a delta operation concerning the added class when the product line evolves by adding a new model. It is necessary to improve the tool to take entity removal into account. In this line, we must accomplish experimentation that implies changes in attributes. We focused on entity and method levels. The current version of the tool does not take charge of the sequential application of the dependent Delta Module. Even if we present how it must work, we need to improve the code for that and realize more experimentation. The product configuration must be more constrained. Today, we indicate a configuration that can activate the Delta Module, but we must do more control when applying the Delta Module. As we move forward, the chosen configuration has little impact on the derivation, so we indicate the metamodel to be applied. It would be interesting if, knowing that a configuration is entered for each Delta Module, it was possible to leave the choice of Delta Modules to the configuration. In this way, a configuration can automatically design all applicable Delta Modules. The visualization of the Delta Module must be improved. It will be interesting to go beyond showing simple Delta Module dependencies, the link with the impacted model generator, and the relation between the Delta Module and entity. This will help with the maintenance of the Delta Project. Thus, the impacted Delta Module could be refactored by removing an obsolete entity when we evolve the product line. However, this tool tackles scientific lock about implemented DOP at the model level because none do the same today.

6.5 Threats to Validity

This chapter introduces tools that facilitate the implementation of Software Product Lines (SPL) based on Delta-Oriented Programming (DOP) paradigms and Model-Driven Engineering (MDE). However, it is essential to acknowledge that these tools have certain limitations.

Specifically, the primary emphasis of the tool is on handling complex core strategies, often neglecting support for adding entities. Consequently, each Delta Module primarily focuses on removing entities, which can make it challenging to add new ones seamlessly. While this approach is suitable for large-scale use cases, such as our ongoing example, it may pose challenges in situations that require complex and multiple operations, especially those involving entity removal and reuse. This limitation could become more pronounced in large-scale systems where detecting all possible Delta Modules may be impractical.

Secondly, the current management of reusable artifacts relies on JSON files. Compared to tools like DeltaJ, manual management of files can become cumbersome at a larger scale, impacting product line maintenance. Exploring more efficient ways to manage JSON files, such as utilizing a JSON database like MongoDB, could significantly enhance this aspect.

Thirdly, we may need to fine-tune the code generated by our tools to improve its overall readability and quality.

Fourthly, concerning the generation of source code for the product, the current process involves converting a model into code using Famix2Java, a visitor [EKLG+03] designed to export FamixJava [Tic99] models into Java code. However, expanding our capabilities to generate code in other programming languages would necessitate the development of a new engine tailored for model-to-code transformations in those specific languages.

6.6 Conclusions

The *PhaDOP* framework is a transformation-centric approach to Software Product Lines. It is specifically designed for implementing SPLs using the Delta-Oriented Programming (DOP) paradigm and leveraging Model-Driven Engineering. Currently, it focuses on generating Object-Oriented code emphasizing entity removal. Ongoing efforts aim to enhance its capabilities to encompass a broader range of operations, including entity addition. The presented end-to-end process validates the framework's functionality through a straightforward use case. We actively pursue continuous improvements and enhancements.

Chapter 7. Experimentation on Software Connector Generation from the Connector Product Line

This Chapter validates the practicality of the proposed framework for creating interoperability connectors using *ConPL*, a software product line approach. The framework consists of domain and application engineering sub-processes, each further divided into problem and solution spaces, as introduced in Chapter 5 and illustrated in Figure 5.8.

This section validates the practicability of the proposed framework for creating interoperability connectors using *ConPL*, a software product line approach. This framework was introduced in Chapter 5, as illustrated in Figure 5.8.

The key steps involved in this practicability demonstration are as follows: Within the problem space of domain engineering, we address feature identification and feature modeling. This entails recognizing common architectural and behavioral features by analyzing interoperability patterns in existing connectors, summarized in a connector corpus. Following this, we construct a feature model representing all potential features for building a messaging connector. We establish relationships between the connector product line architecture materialized by the metamodel presented in Chapter 3, Figure 3.4, and a reusable artifact. This constitutes the solution space of application engineering, specifically for implementing the connector product line.

The primary focus of this section is to demonstrate that the software product line enables the creation of interoperability connectors in practice. The SPL expert creates the product line, and engineers can use it to produce products. This use case emphasizes application engineering and involves:

- Receiving a connector specification and configuring the product to generate a product variant of the established product line. Leveraging model-driven engineering, the expected outcome of the derivation process is a variant of the connector metamodel, potentially with additional or reduced entities and linked reusable artifacts.

- Generating the source code for the resulting product model after instantiation.

For clarity, we will work with a streamlined version of the established product line, considering a simplified metamodel. The process will rely on the *PhaDOP* framework developed in Chapter 6, Figure 6.1. This section will first present a reduced product line, personal product derivation for this product line based on a product specification and configuration, and generation of the interoperability connector.

7.1 Incremental Feature Analysis and Identification

This subsection aims to identify features and reusable artifacts for constructing an experimentation product line. The approach involves meticulously analyzing legacy class source code files, explicitly focusing on Java. In some instances, there might be a need to restructure a class to encapsulate shared features better or enhance usability within our experimentation framework.

This experiment is limited to analyzing Java code. We analyze two groups to ensure meaningful comparisons: publisher code and code consumer code.For each component serving as either a producer or a consumer, we select the first one, identify its features, and insert reusable artifacts into the artifact repository. If necessary, we may restructure without altering the component's behavior. For example, consolidating multiple lines of code into a Java method can create a reusable function or feature. The following sections will explain the proposed process in detail, starting with producers and moving on to consumers.

Incremental analysis for producers For the incremental analysis, we consider two connectors due to time and resource constraints. The connectors used for the experimentation correspond to **Connector 1**, **Connector 2**, where the exchange flow are respectively depicted in order in Figures 5.11, 5.12 in Chapter 5, Section 5.6.

New or variant features are framed in red. So, the code not framed in red corresponds to a feature already identified in a previous analysis. The feature analysis and localization process is manual.

Producer of connector 1: feature location and feature analysis: We start by analyzing the producer of the first connector, who is responsible for publishing a text message to a channel. In Figure 7.1, the producer's code is presented with highlighted features.

The producer of the first connector exhibits seven features, with the red highlights indicating corresponding reusable artifacts. This initial iteration results

Figure 7.1: Identified Features for Basic Publish-Subscribe Producer

in the presentation of the first version of the producer feature table, as shown in Table 7.1. In the table, the symbol *"+"* indicates the presence of a feature, while *"-"* signifies its absence.

From Table 7.1, we obtain the feature model of the connector product line after the first iteration. Figure 7.2 depicts the corresponding feature model.

This initial iteration process aligns with the thesis positioning, which involves constructing a software product line from existing products using static analysis and *Expert-Driven* strategies. Some features in Table 7.1 result from static code analysis, while others are derived from expert knowledge. For example, the documentation extracts features such as *AutoDelete*, *Exclusive*, *Durable*, and *Argument*, which provide possible properties for a message *Queue*. Although not explicitly evident in the code, we encounter values such as *true* and *false*. Some advanced techniques, like feature mining, can potentially complement or enhance this process [SHU+13]. However, this specific technique is not utilized in the scope of this work.

Since they are present, we consider all features mandatory for this initial iteration of the connector. However, exceptions may be based on the information extracted from the code. For example, the Argument feature is marked as null, indicating that it may be present or absent. Meanwhile, features such as *AutoDelete*, *Exclusive*, and *Durable* have optional groups.

Feature	Basic producer
Connection	+
Host	+
Channel	+
Queue	+
Queue name	+
Message	+
basicPublish	+

Table 7.1: First Iteration of the Incremental Feature Table

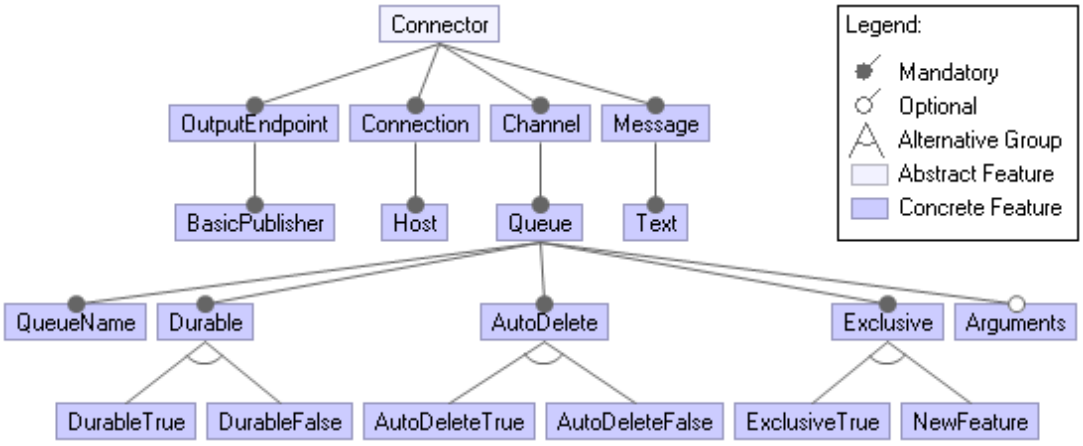
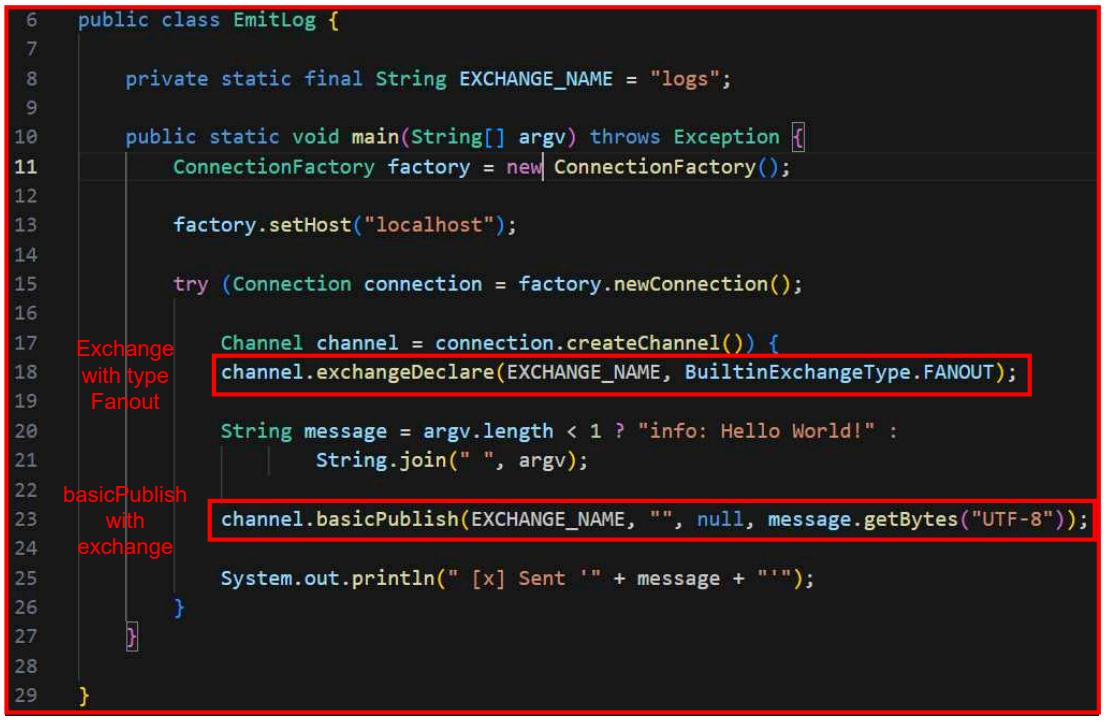


Figure 7.2: Feature Model of the Experimental Connector Following the Initial Iteration

Producer of connector 2: feature location and feature analysis: The second producer is similar to the first, but with a key distinction: it includes an exchange where the producer publishes. The producer publishes directly to a queue in the first connector (Figure 5.11). In the second connector (Figure 5.12),

the exchange enables the producer to publish without specifying a particular queue, broadcasting the message to all queues. The product line's variability is limited to two features. The distinguishing factor is the ability to use an exchange for publishing to all queues. Figure 7.3 displays the identified features in the producer code. Table 7.2 presents the second iteration of features.




```
6   public class EmitLog {
7
8       private static final String EXCHANGE_NAME = "logs";
9
10      public static void main(String[] argv) throws Exception {
11          ConnectionFactory factory = new ConnectionFactory();
12
13          factory.setHost("localhost");
14
15          try (Connection connection = factory.newConnection();
16
17              Channel channel = connection.createChannel()) {
18              channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.FANOUT);
19
20              String message = argv.length < 1 ? "info: Hello World!" :
21                      String.join(" ", argv);
22
23              channel.basicPublish(EXCHANGE_NAME, "", null, message.getBytes("UTF-8"));
24
25              System.out.println(" [x] Sent '" + message + "'");
26          }
27      }
28
29  }
```

Figure 7.3: Identified Features for Fanout through Exchange Publish-Subscribe Producer

Compared to the preceding connector, the second connector introduces two new features. Table 7.2 displays the features of the second producer in a distinct column.

From Table 7.2 resulting from the second iteration evolves the feature model of the connector. The resulting feature model is shown in Figure 7.4.

The second feature model introduces two new features: *ExchangePublisher* and *Exchange*. Additionally, new constraints are evident. Specifically, *BasicPublisher* implies the presence of the *Queue* feature, while *ExchangePublisher* implies the presence of the *Exchange* feature. As a reminder of the *Implies* truth table, for features A and B, A implies B is false only when A is present and B is ab-

Feature	Basic producer	Fanout exchange produce
Connection	+	+
Host	+	+
Channel	+	+
Queue	+	-
Queue name	+	-
Message	+	+
Queue basicPublish	+	-
Fanout exchange	-	+
Exchange basicPublish	-	+

Table 7.2: Second Iteration of the Incremental Feature Table

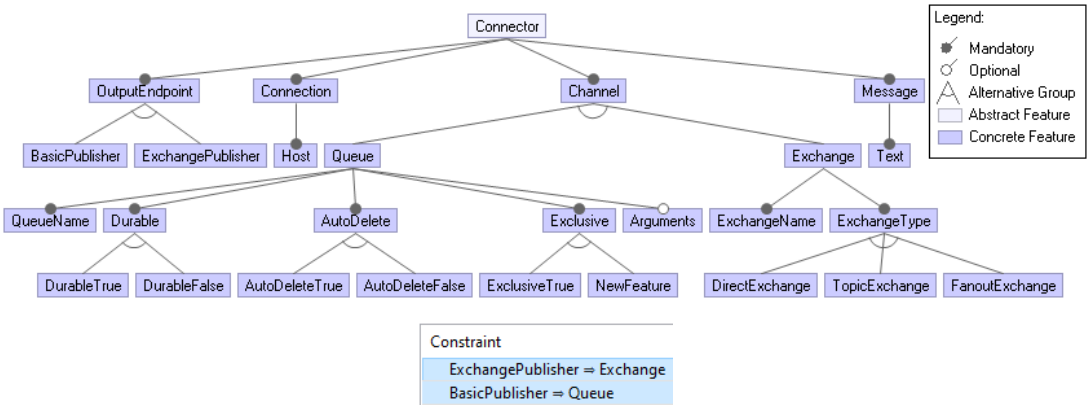


Figure 7.4: Feature Model of the Experimental Connector Following the Second Iteration

sent, and true in all other cases. Therefore, these constraints signify that a valid

configuration cannot have *BasicPublisher* present without *Queue*, and similarly, *ExchangePublisher* cannot be present without *Exchange*.

We must follow a similar process for each producer and consumer to obtain the complete feature model construction. In this demonstration, we will conclude with the producer. It is now time to transition to the solution space.

7.2 Implementation Connector Product Line

Given time and resource constraints, the demonstration primarily focuses on the two producers of the connector. However, it's important to note that the process applies to other constituents of the connector. Based on the feature model, we present the experimental connector's metamodel, a streamlined version of the connector metamodel introduced in Chapter 3, focusing solely on producers.

7.2.1 Metamodel of the Reduced Experimental Connector

Using the Model-Driven Engineering technique, we implemented the Core Delta Module at the model level and aligned it with the prospective model of the reduced connector, as shown in Figure 7.4. The conversion from the feature model to the UML class diagram, which is the reduced metamodel of the connector represents the Core Delta Module, followed the transformation rule articulated in Chapter 5, Section 5.5.4. The metamodel for the experimental connector is depicted in Figure 7.5.

Creating the Metamodel Generator for the Reduced Connector in Pharo Following the **PhaDOP** framework, Chapitre 6, that we follow for the solution space implementation, we create the metamodel generator that will make it possible to generate a metamodel for variant connectors.

Listing 7.1 specifies the prefix used to show the distinction between generated metamodel variants as *CC* for *Core connector*.

Listing 7.1: Method that precise Prefix making a distinction between metamodel variants

```
633 prefix
634     ^ #'CC'
```

Listing 7.2 method specifies the package in which the metamodel will be generated.

Listing 7.2: Method that specify the package where the metamodel will be generated

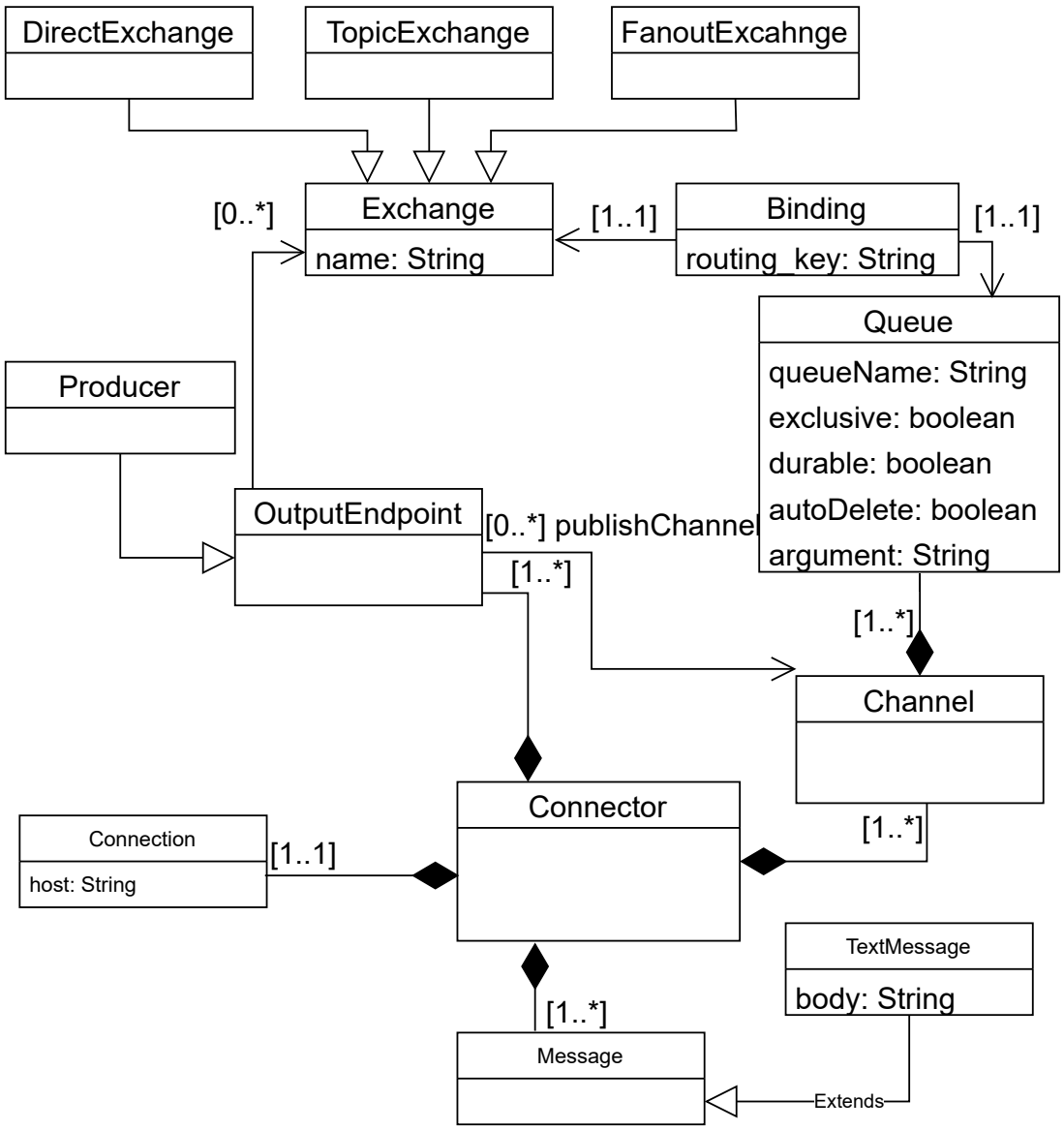


Figure 7.5: Reduced Metamodel of the Connector, Focusing on the Producer Constituent

```
635 packageName
636     ^ #'Connector-Model'
```

Listing 7.3 presents the code for the declaration of the metamodel class generator.

Listing 7.3: Declaration of the connector metamodel generator class

```
FamixMetamodelGenerator subclass: #ConnectorMetamodelGenerator
    instanceVariableNames: 'connector connection channel queue exchange binding directExchange
        topicExchange fanoutExchange message textMessage producer outputendpoint'
    classVariableNames: ''
    package: 'Connector-Model-Generator'
```

Listing 7.4 is the method that specifies creating an instance for each entity that must generate the metamodel

Listing 7.4: Method that creates an instance for each entity of the metamodel

```
defineClasses
    super defineClasses.
    connector := builder newClassNamed: #Connector.

     outputEndpoint := builder newClassNamed: #OutputEndpoint.
     producer := builder newClassNamed: #producer.

    connection := builder newClassNamed: #Connection.
     message := builder newClassNamed: #Message.

    textMessage := builder newClassNamed: #TextMessage.

    channel := builder newClassNamed: #Channel.
    queue := builder newClassNamed: #Queue.
     binding := builder newClassNamed: #Binding.

     exchange := builder newClassNamed: #Exchange.
     directExchange := builder newClassNamed: #DirectExchange.
     topicExchange := builder newClassNamed: #TopicExchange.
     fanoutExchange := builder newClassNamed: #FanoutExchange.
```

Listing 7.5 displays the hierarchy between different classes. The code shows that the class named *Producer* is inherited from the class named *OutputEndpoint*. Additionally, the class named *Exchange* has three child classes: *DirectExchange*, *TopicExchange*, and *FanoutExchange*. In this code, we also use the *TextMessage* class, inherited from the *Message* entity.

Listing 7.5: Method that specify eventual inheritance between entities

```
defineHierarchy
    super defineHierarchy.
    producer --|> outputEndpoint.
    directExchange --|> exchange.
    topicExchange --|> exchange.
    fanoutExchange --|> exchange.
    textMessage --|> message.
```

In Listing 7.6, we present the properties associated with each entity in the proposed metamodel classes. The *Producer* class inherits from the *OutputEndpoint* class as shown in the code. The *Exchange* class has three child classes: *DirectExchange*, *TopicExchange*, and *FanoutExchange*. Additionally, the code demonstrates the use of the *TextMessage* entity, which inherits from the *Message* entity. Properties have been specified for entities such as *Exchange*, *Queue*, *TextMessage*, Binding, and *Connection*.

Listing 7.6: Method that specify properties for each entity that has at lest one

```
defineProperties
    super defineProperties.

    queue property: #queueName type: #String.
    queue property: #durable type: #String.
    queue property: #exclusive type: #String.
     queue property: #autoDelete type: #String.
     queue property: #argunment type: #String.

    exchange property: #exchangeName type: #String.
    exchange property: #exchangeType type: #String.

    binding property: #routingKey type: #String.

     textMessage property: #body type: #String.

     connection property: #host type: #String.
```

In Listing 7.7, we can see the relationships between different entities. One interesting observation is that the entity *Connector* can be associated with multiple instances of *Channels* and *Messages*. This entity has a single *Connection* and one *OutputEndpoint*, each associated to a singular connector. On the other hand, the *OutputEndpoint* class can publish in multiple *Channels* and interact with several possible *Exchanges*. Additionally, a *Binding* is linked to one *Exchange* and one *Queue*.

Listing 7.7: Method that specifies cardinalities between classes

```
defineRelations
   (connector property: #channel) <>-* (channel property: #connector).
    (connector property: #message) <>-* (message property: #connector).
    (connector property: #connection) <>- (connection property: #connector).
    (connector property: #outputEnpoint) <>- (outputEndpoint property: #connector).


    (outputEndpoint property: #channel) <>- (channel property: #outputEndpoint).
    (outputEndpoint property: #exchange) <>- (exchange property: #outputEndpoint).

    (binding property: #exchange) <>- (exchange property: #binding).
    (binding property: #queue) <>- (exchange property: #queue).
```

After obtaining all methods of the metamodel generator, we use the model generator GUIs or PhaDOP to generate the metamodel as represented in Figure 7.5.

Creating Possible Connector Variants and the Associated Delta Module
Once we have the metamodel, we must create a metamodel variant for each connector possible connector. We need to know how many delta modules are required for each producer. Based on Table 7.3, which is the amelioration of Table 7.2, we highlight the possible delta module.

The two variants are obtained as follows:

- *DeltaQueue:* ***DeltaQueue*** corresponds to the initially reduced metamodel when everything related to *Exchange* is removed.

Feature	Basic producer	Fanout exchange produce	Delta Module for Basic producer	Delta Module for Exchangeproducer
Connection	+	+		
Host	+	+		
Channel	+	+		
Queue	+	-		remove Queue
Message	+	+		
Queue basicPublish	+	-		remove Queue Basic publish
Fanout exchange	-	+	remove Queue	
Exchange basicPublish	-	+	remove Exchange Basic publish	

Table 7.3: Second Iteration of the Incremental Feature Table

- *DeltaFanoutExchange:*corresponds to the initially reduced metamodel when both *Queue* and *Exchange* are included, specifically limited to *FanoutExchange*.

Figure7.6 and Figure 7.7 and **??** illustrate the two representations of the resulting metamodel variants obtained after the derivation facilitated by the *PhaDOP* framework.

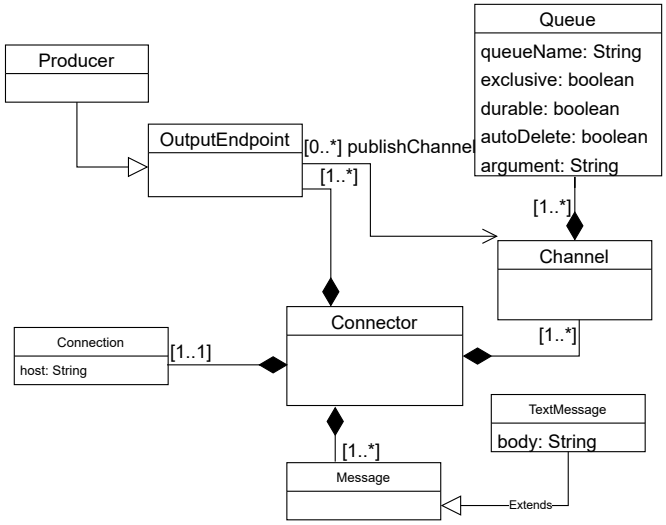


Figure 7.6: Metamodel Variant for the Basic Publish Producer Utilizing a Queue

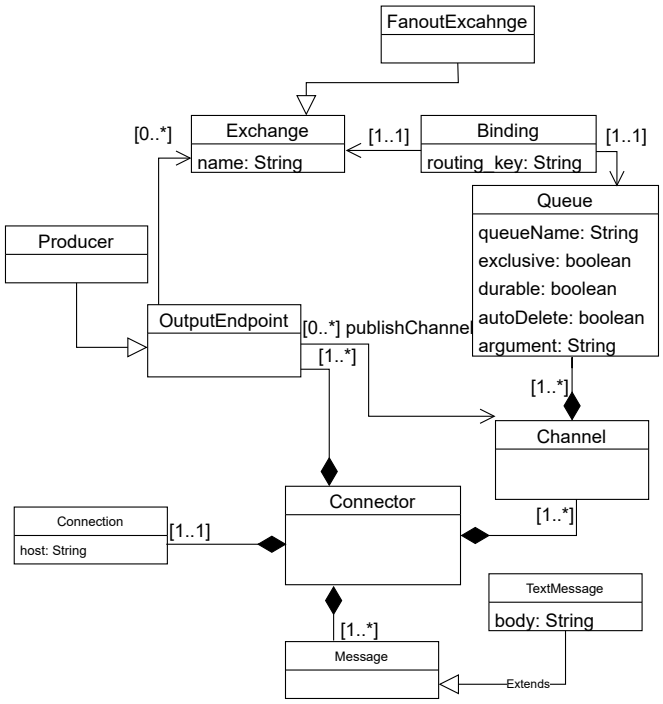


Figure 7.7: Metamodel Variant for the Basic Publish Producer Utilizing a Queue Through a Fanout Exchange

7.2.2 Reusable artifact at the method-level granularity

The Core Delta Module, implemented through the metamodel, emphasizes granularity at both the entity and attribute levels, as explained in Chapter 5, Section 5.5.3, Paragraph 5.5.3. As a structural diagram, it is essential to note that the metamodel cannot contain information about the connector's behavior. To overcome this limitation, in Chapter 5, we introduced a JSON-structured repository to capture reusable artifacts at the method level. This approach exclusively facilitates the generation of the class structure. Furthermore, we proposed a metamodel to capture source code information related to methods.

Refactoring of the artifact The significance of this industrial use case, in contrast to the foundational implementation in Chapter 6 via the EPL use case, lies in the heterogeneous nature of the artifact. The metamodel effectively captures details about attributes and entities. So we can represent architectural commonalities and variabilities [SSS17]. However, we introduce a JSON-based repository

to encapsulate behavior-related information within methods. The challenge with the *PhaDOP* framework is its inability to represent artifacts beyond attributes, methods, and entities. Other code blocks cannot be adequately captured. We propose refactoring the code by consolidating all non-attribute and non-class blocks within methods to address this limitation.

Listing 7.8 and Listing 7.9 represent the refactored version of the source code of the concerned use case for implementation presented in Figure 7.1 and Figure 7.3.

Listing 7.8: Identified Features for Basic Publish-Subscribe Producer after refactoring

```
699 public class SendRefactored {
700
701     private final static String QUEUE_NAME = "hello";
702     Connection connection;
703     Channel channel;
704     String message="Hello World!";
705
706     public static void main(String[] argv) throws Exception {
707         try (Connection connection = getConnection();
708              channel = getChannel(Connection connection)) {
709              queueDeclareChannel(channel, QUEUE_NAME, false, false, false, null);
710              basicPublish("", QUEUE_NAME, null, message.getBytes(StandardCharsets.UTF_8));
711              printMessage();
712         }
713     }
714     private static Connection getConnection() {
715         ConnectionFactory factory = new ConnectionFactory();
716         factory.setHost("localhost");
717         return factory.newConnection();
718     }
719     private static Channel getChannel(Connection connection) {
720         return connection.createChannel();
721     }
722     private static void queueDeclareChannel(Channel channel,
723         String QUEUE_NAME, boolean durable, boolean exclusive,
724         boolean autoDelete, String argument) {
725              channel.queueDeclare(QUEUE_NAME, false, false, false, null);
726     }
727     private static void basicPublish(Channel channel, String str,
728         messageByte) {
729              channel.basicPublish(str, QUEUE_NAME, null,
730              messageByte);
731     }
732     private static void printMessage(String message) {
733         System.out.println(" [x] Sent '" + message + "'");
734     }
735 }
```

Listing 7.9: Identified Features for Fanout through Exchange Publish-Subscribe Producer after refactoring

```
736 public class EmitLogRefactored {
737
738     private static final String EXCHANGE_NAME = "logs";
739     String message = "info: Hello World!";
740
741     public static void main(String[] argv) throws Exception {
742         try (Connection connection = getConnection();
743              channel = getChannel(Connection connection)) {
744              exchangeDeclareChannel(channel, EXCHANGE_NAME, BuiltinExchangeType.FANOUT)
745              publishExchangeFanout(channel, EXCHANGE_NAME, BuiltinExchangeType.FANOUT)
746              printMessage();
747         }
748     }
749
```

```
750     private static Connection getConnection() {
751         ConnectionFactory factory = new ConnectionFactory();
752         factory.setHost("localhost");
753         return factory.newConnection();
754     }
755     private static Channel getChannel(Connection connection) {
756         return connection.createChannel();
757     }
758     private static void exchangeDeclareChannel(Channel channel,
759         String EXCHANGE_NAME, BuiltinExchangeType.FANOUT) {
760             channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.FANOUT);
761     }
762     private static void publishExchangeFanout(Channel channel,
763         String EXCHANGE_NAME, BuiltinExchangeType.FANOUT) {
764             channel.basicPublish(Channel channel, EXCHANGE_NAME,
765             "", null, message.getBytes("UTF-8"));
766     }
767     private static void publishExchangeFanout(Channel channel,
768         String EXCHANGE_NAME, BuiltinExchangeType.FANOUT) {
769             channel.basicPublish(Channel channel, EXCHANGE_NAME,
770             "", null, message.getBytes("UTF-8"));
771     }
772     private static void print(String message) {
773         System.out.println(" [x] Sent '" + message + "'");
774     }
775 }
```

7.2.2.0.1 JSON representation of the model-level reusable artifact:

After the refactoring, we can represent all the reusable artifacts at model-level granularity. This is done in conformity with the reusable artifact repository presented in Figure 5.9. For simplification purposes, we represent reusable artifacts for the producer with publication without *Exchange*, called *SenderRefactored* in Listing 7.8.

Listing 7.10 shows the JSON representation of the reusable artifact repository focusing on the *SenderRefactored* class.

Listing 7.10: Reusable method-level artifact focusing of the basic producer class

```
776 {
777     "producer": {
778         "targetSourceLocation": "D:\\Users\bouboutiam.niang\\workspace\\manuscript\\
                industst_use_case\\src",
779         "methods": [
780             {
781                 "name": "main",
782                 "sourceEnchor": "{ try (Connection connection = getConnection(); channel =
                        getChannel(Connection connection)) {queueDeclareChannel(channel,
                        QUEUE_NAME, false, false, false, null);basicPublish(\"\", QUEUE_NAME,
                        null, message.getBytes(StandardCharsets.UTF_8));printMessage();}",
783                 "parameters": [
784                     {
785                         "name": "argv",
786                         "type": "String[]"
787                     }
788                 ]
789             },
790             {
791                 "name": "getConnection",
792                 "sourceEnchor": "{ConnectionFactory factory = new ConnectionFactory(); factory
                        .setHost(\"localhost\"); return factory.newConnection();}",
793                 "parameters": [
794                     {}
795                 ]
796             },
797             {
798                 "name": "queueDeclareChannel",
```

```
                "sourceEnchor": "{String QUEUE_NAME, boolean durable, boolean exclusive,
                    boolean autoDelete, String argument) {channel.queueDeclare(QUEUE_NAME,
                    false, false, false, null);}",
                "parameters": [
                    {
                        "name": "channel",
                        "type": "Channel"
                    },
                    {
                        "name": "exclusive",
                        "type": "boolean"
                    },
                    {
                        "name": "argument",
                        "type": "String"
                    }
                ]
            },
            {
                "name": "basicPublish",
                "sourceEnchor": "{ channel.basicPublish(str, QUEUE_NAME, null, messageByte);}"
                    ,
                "parameters": [
                    {
                        "name": "channel",
                        "type": "Channel"
                    },
                    {
                        "name": "str",
                        "type": "String"
                    }
                ]
            },
            {
                "name": "printMessage",
                "sourceEnchor": "{  System.out.println(\" [x] Sent '\" + message + \"'\");}",
                "parameters": [
                    {
                        "name": "message",
                        "type": "String"
                    }
                ]
            }
        ],
        "parent": {},
        "interface": [
            {}
        ]
    }
}
```

7.2.3 Product Derivation - Basic Producer Code Generation:

After obtaining the variant of the metamodel and the reusable artifact, the next step is to generate the source code for the Producer class. This can be achieved using the dedicated GUIs provided by the PhaDOP framework. A detailed explanation of this process is presented in Chapter 6, Figure 6.17, and the corresponding code is shown in Chapter 6, Listing 6.20. The repository JSON location and the target folder for the generated source code are the only variables that change. The generated source code is shown in The generated code in Listing 7.11 is quite similar to the initial code that was reverse-engineered. This similarity is discussed in detail in the EPL use case demonstration in Chap-

ter 6. One significant difference is that the generated code contains no method arguments.

Listing 7.11: Basic producer code generated

```
846     @ExampleComponentAnnotation
847 public class SenderRefactored extends ParentExemple {
848
849
850     @ExampleMethodAnnotation
851     String main() {
852      try (Connection connection = getConnection(); channel = getChannel(Connection connection)
             ) {queueDeclareChannel(channel, QUEUE_NAME, false, false, false, null);basicPublish(
             "", QUEUE_NAME, null, message.getBytes(StandardCharsets.UTF_8));printMessage();
853     }
854
855     @ExampleMethodAnnotation
856     String getConnection() {
857     ConnectionFactory factory = new ConnectionFactory(); factory.setHost("localhost"); return
          factory.newConnection();
858     }
859
860     @ExampleMethodAnnotation
861     String queueDeclareChannel() {
862     String QUEUE_NAME, boolean durable, boolean exclusive, boolean autoDelete, String argument
          ) {channel.queueDeclare(QUEUE_NAME, false, false, false, null);
863     }
864
865     @ExampleMethodAnnotation
866     String basicPublish() {
867     { channel.basicPublish(str, QUEUE_NAME, null, messageByte);
868     }
869
870     @ExampleMethodAnnotation
871     String printMessage() {
872       System.out.println(" [x] Sent '" + message + "'");
873     }
874 }
```

7.3 Conclusion

In this chapter, we conducted an industrial use case experiment to validate the feasibility of the proposed software product line approach applied to interoperability connectors. This experiment aimed to validate an end-to-end process for implementing a software product line, starting from the feature location step and culminating in product generation. The Delta-Oriented Programming (DOP) paradigm is used in this process as a transformational approach. Model-Driven Engineering (MDE) techniques are employed to manipulate artifacts at the model level whenever possible.

One challenge encountered was managing product lines at the method-level granularity using MDE and DOP. This challenge was addressed through previous code refactoring. The chapter validated the practicability of the PhaDOP framework in an industrial context. However, there are areas for improvement, such as missing method parameters. Future studies will address formatting issues to make the generated code more user-friendly and visually clear.

Chapter 8. Conclusion

8.1 Summary

The thesis addresses the challenges of achieving interoperability in dynamic and distributed Information Systems that face continuous technological and organizational changes. It introduces a practical metamodel for the reified interoperability mechanisms called *Messaging Connector*, which emphasizes asynchronous communication. The exploration considers interoperability mechanisms as first-class constituents, leading to conceptualizing Information Systems as Systems-of-Systems. This perspective grants technical, geographical, and managerial independence to the information system and its constituents. A transferable heuristic is proposed for the Messaging Connector metamodel to ensure comprehensive coverage. The comparative analysis emphasizes the advantages of messaging-based connectors in industrial settings. Methodologies for Software Product Lines utilizing Delta-Oriented Programming find practical application through the PhaDOP framework. This thesis significantly contributes to interoperability, messaging connectors, and software product lines. The methodologies and frameworks presented offer valuable insights and tools, advancing our understanding and practical application of interoperability mechanisms and software engineering practices in the evolving landscape.

8.2 Contribution

The thesis led to the following contributions:

- Reverse engineering of current systems that incorporate interoperability mechanisms resulted in creating a comprehensive repository of connectors designed for experimentation within the scientific community.
- The reification of the interoperability mechanism, transforming the abstract concept into a tangible notion of the *Messaging Connector*, is accomplished by introducing an extensible model designed to illustrate and define the fundamental aspects of a connector. The proposed metamodel is specifically crafted to highlight asynchronous communication.

- A heuristic for validating and extending a metamodel for *Messaging Connectors*. This contribution outlines methodologies for validating the metamodel to ensure it includes all possible connectors and details the process for accommodating new connectors that the existing metamodel may not cover. This approach is transferable and can be applied to other metamodels.

- Comparative analysis between messaging-based connectors and non-messaging-based connectors. This involved demonstrating, through use cases derived from real-world industrial scenarios, situations where numerous document messages with substantial volumes must be transmitted to an application within a short time frame. The experiment illustrates that messaging solutions exhibit flexibility and systems with resilience are not susceptible to bottleneck issues. Additionally, the experiment explores the impact of the connector's constituents, examining how the speed of interactions varies with increased routers, transformers, etc.

- Proposing methodologies and implementation strategies supported by a dedicated tool for implementing Software Product Lines (SPL). This approach revolves around Delta-Oriented Programming (DOP) using Model-Driven Engineering (MDE) principles.

- Proposing the *PhaDOP* framework and demonstrating its practical application in implementing Software Product Lines. The focus is on employing Delta-Oriented Programming at the model level to execute the *ConPL* framework, illustrated through a specific use case. To facilitate this, a new metamodel has been introduced for organizing and managing the delta module at a large scale.

8.3 Future work

In addition to the contributions detailed in Section 8.2, our research reveals new perspectives for addressing challenges not explicitly covered in this thesis. These perspectives are relevant for both interoperability and software product line engineering.

Using the Connector Metamodel to Analyze Interoperability Mechanisms: We use a manual reverse engineering process to examine existing interoperability mechanisms with expert knowledge when constructing the connector metamodel. Due to the required time and expertise, this method is effective but

challenging for non-experts. The metamodel includes entities relevant to interoperability mechanisms, making it possible for interoperability pattern recognition within projects.

A metamodel is platform-independent, allowing its application across projects that involve diverse programming languages. This versatility is valuable for analyzing interoperability mechanisms and identifying patterns, especially when estimating the impact of changes in interoperability requirements, such as transitioning from HTTP to HTTPS. This is particularly relevant when initiating a project that adopts messaging-style communication for decoupled interoperability. The metamodel helps identify code segments that need to be extracted from business logic and placed in the connector, streamlining the development process.

Leveraging Behavior Models, such as Sequence Diagrams, to Enhance Metamodel Extensibility: Chapter 4 validates the expandability of the metamodel through a defined process illustrated in Figure 4.12. This process may require expertise. The initial step ensures the connector metamodel covers a reified or embedded interoperability mechanism. If not, it should the metamodel must extended by adding the corresponding entity. However, determining whether a particular mechanism corresponds to a specific entity in the metamodel can be challenging. Ideally, the class *DynamicRouter.java* would correspond to the *DynamicRouter* entity in the metamodel 3.4. However, developers are not obligated to use a nomenclature that suits the metamodel. For example, a developer might implement a message producer in a class named *DynamicRouter.java*. Analyzing the behavior of the implemented pattern is essential to determine if it corresponds to an entity in the metamodel. Relying solely on a class diagram to represent the metamodel may be insufficient and time-consuming. Therefore, it may be beneficial to complement the metamodel with a series of sequence diagrams, each involving a pattern presented as an entity in the metamodel. This approach considers class names in the engineering process and examines their interactions with other classes. This aids in the classification of the analyzed pattern within a specific category.

Comprehensive Experimentation: Assessing Performance, Security, Energy Consumption, and Complexity: Chapter 3 provides a comprehensive overview of the reified *Messaging Connector*, highlighting its internal constituents within the system. The initial experimentation primarily focuses on assessing the proposed connector's performance compared to a non-messaging-based counterpart. However, due to the sensitivity of the reified connector to various factors,

more in-depth experimentation is warranted. To evaluate the connector objectively, analyzing its performance concerning complexity is crucial. For instance, it is essential to explore the impact of the number of message routers on interaction speed and identify the key components adversely affecting performance.

Another vital aspect to investigate is the security vulnerabilities of the connector. Conducting ethical hacking tests and benchmarking against other interoperability solutions can provide insights into the connector's robustness. It is also essential to understand the energy consumption of the proposed connector's structure. Identifying constituents contributing to higher or lower energy consumption and comparing the reified connector's energy efficiency with embedded interoperability solutions in business logic is essential. If the reified connector is more energy-efficient, it could reduce the energy consumption associated with bloatware.

Complete Code Generation: Beyond Class Diagrams When summarizing ConPL, the Software Product Line approach described in Chapter 5, Section 5.5, and illustrated in Figure 5.5, the connector metamodel defines the structure of the connector. The ConPL Solution Space in Application engineering includes a set of predefined modifications called delta modules, activated by configuring the metamodel, resulting in a variant of the metamodel. The connector's model is instantiated to create a model of the connector. The connector's source code is generated through a model-to-text transformation.

Class diagrams, being structural diagrams, can only generate the class skeleton, encompassing class, attribute, and method signatures. To address this limitation, we propose storing reusable source code separately. The stored code is combined with the metamodel to generate the methods' body, as Chapter 7 explains.

In the future, exploring the possibility of capturing more than just static information will be necessary. Future work will investigate an approach that integrates class diagrams, sequence diagrams, and statechart diagrams to facilitate the generation of more comprehensive code. The class diagram captures data structure information, while the sequence diagram specifies interdependencies among classes and methods, indicating classes used by others and methods called by others. State machines model the behavior of each entity, enabling the filling of method bodies.

Figure 8.1 shows the process that sequentially uses different models for mode core generation.

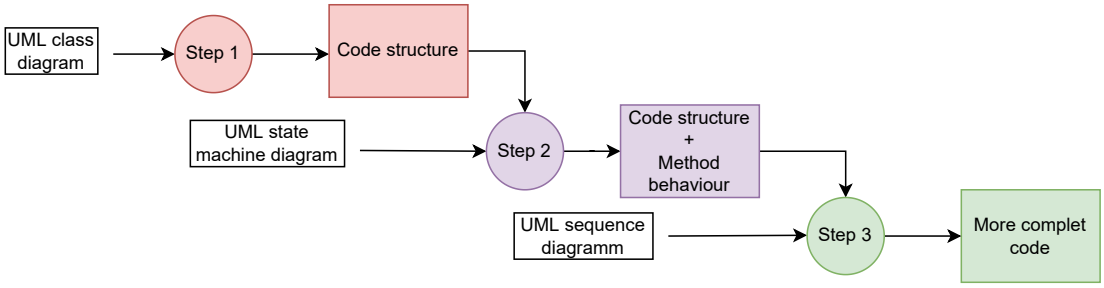


Figure 8.1: Sequential Combination of Class, Sequence, and State Chart Diagrams for Code Generation

Configurable and Comprehensive Code Generation through Generative Artificial Intelligence The process of product derivation, as explained in Chapter 5, Section 5.5, and illustrated in Figure 5.5, utilizes the ConPL approach. This involves applying a delta module driven by the configuration specified in the domain feature model. However, generating code solely from a connector model represented as a class diagram results in structural code through model-to-text transformation. Additional methods, such as reusing source code or exploring diverse model combinations, can enhance code generation.

Generative AI [Win92], including tools like Github Copilot [1] [DMN+23], CodeGeex [2] [ZXZ+23], and OpenAI ChatGPT [3] [ZJYR23], presents a promising solution to overcome the challenges in generating comprehensive code. While the existing code snippets in the connector repository were derived from extensive analysis, discovering new interoperability patterns may necessitate ongoing research, which can be time-consuming, especially for non-experts. Leveraging advanced Generative AI solutions can significantly aid in both configuration tasks and complete code generation.

The manual process of configuring the connector based on a specification involves reading, comprehending, and selecting configurations. An alternative approach involves utilizing an AI tool to autonomously read and synthesize the features specified in the connector documentation, streamlining the configuration process.

Figure 8.2 showcases an example wherein an AI-driven tool interprets a domain feature model outlining all potential valid configurations and a connector

[1] `https://chat.openai.com/`
[2] `https://codegeex.cn/`
[3] `https://chat.openai.com/`

specification. The objective is to furnish a valid configuration aligned with the specified requirements.

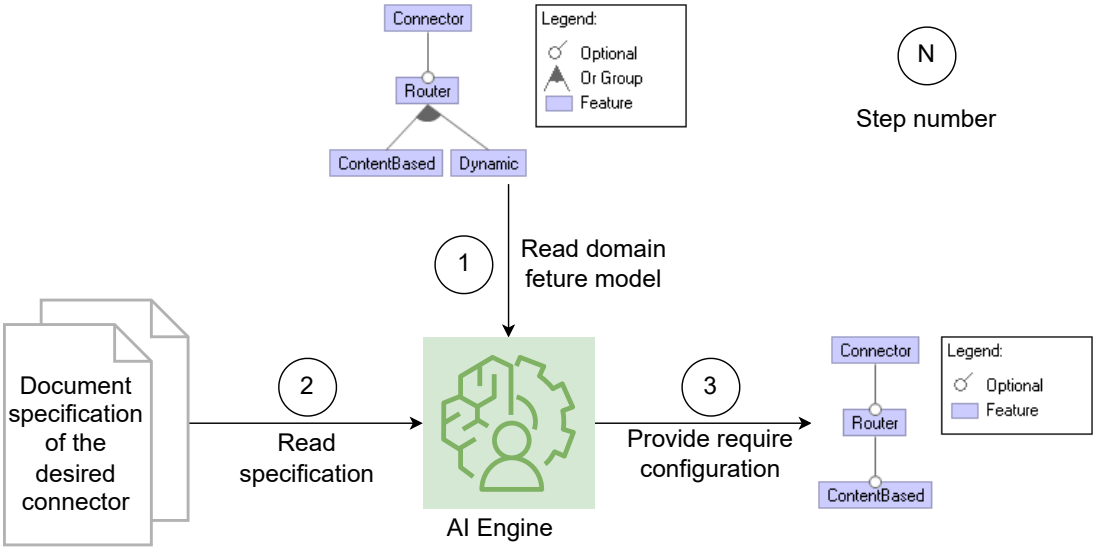


Figure 8.2: The Concept of Utilizing AI for Configuring Software Product Lines

The AI could facilitate the code generation process for the connector. This involves obtaining the metamodel by applying and instantiating the data model and then using the connector model to extract information. From this information, a prompt is constructed and further processed using ChatGPT.

Figure 8.3 illustrates the process of generating source code for connector class diagram fragment.

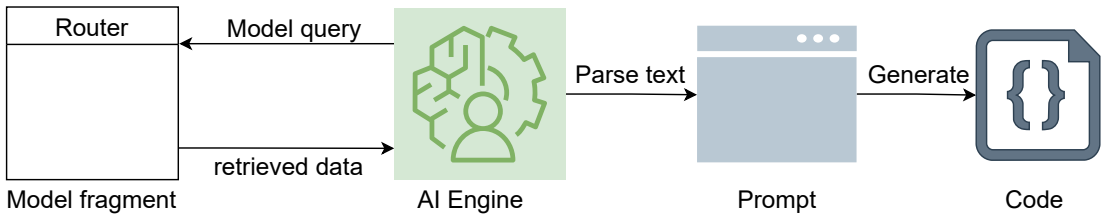


Figure 8.3: The Concept of Utilizing AI for Configuring Software Product Lines

Listing 8.1, Listing 8.2, and Listing 8.3 demonstrate a prompt for creating an event producer and the corresponding source code that it generates.

Listing 8.1: Example promt for generating the Java code of a connector entity

```
Generate in java a producer that publish an event message through RabbitMQ  using Topic\-Type Exchange
```

Listing 8.2: Generated Java code for Connector Router entity

```java
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

public class TopicExchangeProducer {

    private static final String EXCHANGE_NAME = "topic_exchange";
    private static final String ROUTING_KEY = "example.topic";

    public static void main(String[] args) {
        // Set up connection and channel
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost"); // RabbitMQ server address
        try (Connection connection = factory.newConnection(); Channel channel = connection.createChannel())
             {

            // Declare the exchange
            channel.exchangeDeclare(EXCHANGE_NAME, "topic");

            // Message to be published
            String message = "Hello RabbitMQ!";

            // Publish the message with a routing key
            channel.basicPublish(EXCHANGE_NAME, ROUTING_KEY, null, message.getBytes());
            System.out.println(" [x] Sent '" + ROUTING_KEY + "':'" + message + "'");

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Listing 8.3: Generated dependencies required for the generated code

```xml
<dependency>
    <groupId>com.rabbitmq</groupId>
    <artifactId>amqp-client</artifactId>
    <version>5.14.0</version> <!-- Replace with the latest version available -->
</dependency>
```

The objective is to develop a versatile prompt template suitable for various platforms. The interest of the template lies in the deliberate choice of words, which must be used thoughtfully rather than randomly.

Advancing Runtime Interoperability: Leveraging DevOps approach for Continuous Deployment of the Reified Connector One important reason for making connectors tangible components within the system is to achieve system decoupling. This decoupling, as emphasized in the thesis summary (see section 8.1), leads to a system-of-systems where each constituent operates independently. Therefore, the deployment of the connector is crucial. Given the dynamic context of interoperability in information systems and other domains, a potential avenue for future work involves addressing the ongoing generation and deployment of the reified connector. One possible approach is integrating the

connector into a DevOps process, facilitating runtime interoperability through transparent deployment.

LIST OF PUBLICATIONS

The following chronological list showcases the papers published within the context of this thesis:

1. NIANG, Boubou T., KAHN, Giacomo, AMOKRANE, Nawel, et al. Towards the Generation of Interoperability Connectors using Software Product Line Engineering. In : Conférence en IngénieriE du Logiciel. 2021. URL `https://hal.science/hal-03274478/`

2. NIANG, Boubou, KAHN, Giacomo, AMOKRANE, Nawel, et al. Automatic Generation of Interoperability Connectors using Software Product Lines Engineering. In : ICSOFT. 2022. URL `https://hal.science/hal-03673588/`

3. NIANG, Boubou T., KAHN, Giacomo, AMOKRANE, Nawel, et al. Using Moose platform for the implementation of a Software Product Line according to model-based Delta-Oriented Programming. In : IWST22—International Workshop on Smalltalk Technologies. 2022. `https://hal.science/hal-03816240/`

4. COLA journal under review

5. IST journal submit JSS rejected paper before defense

6. Metamodel and validation valorization submit before defense

Throughout the thesis, we conducted research on various other topics. Although these topics are not directly related to the main objective of this thesis, we have compiled a list of corresponding papers for reference:

1. LAVAL, Jannik, NIANG, Boubou Thiam, GHZAIEL, Imene, et al. Le projet Pulse: vers la supervision des échanges dans un système IoT. In : CONGRES INFORSID-Atelier 2: Évolution des SI: vers des SI pervasifs?. 2021. `https://hal.science/hal-03250112/`

2. LAVAL, Jannik, AMOKRANE, Nawel, THIAM NIANG, Boubou, et al. Data interoperability assessment, case of messaging-based data exchanges. Journal of Software: Evolution and Process, 2023, p. e2538. https://hal.science/hal-03250112/

References

[ABG+19] Ermyas Abebe, Dushyant Behl, Chander Govindarajan, Yining Hu, Dileban Karunamoorthy, Petr Novotny, Vinayaka Pandit, Venkatraman Ramakrishna, and Christian Vecchiola. Enabling enterprise blockchain interoperability with trusted data transfer (industry track). In *Proceedings of the 20th international middleware conference industrial track*, pages 29–35, 2019.

[ABKM06] Alexander Arlt, Andreas Brunnert, Robert Kühn, and Matthias Meisdrock. Open message queue. In *Informatiktage*, pages 41–44, 2006.

[ACLF13] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B France. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming*, 78(6):657–681, 2013.

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 187–197. IEEE, 2002.

[ADSG+18a] Marco Autili, Amleto Di Salle, Francesco Gallo, Claudio Pompilio, and Massimo Tivoli. Model-driven adaptation of service choreographies. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1441–1450, 2018.

[ADSG+18b] Marco Autili, Amleto Di Salle, Francesco Gallo, Claudio Pompilio, and Massimo Tivoli. On the model-driven synthesis of evolvable service choreographies. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, pages 1–6, 2018.

[ADSG+18c] Marco Autili, Amleto Di Salle, Francesco Gallo, Claudio Pompilio, and Massimo Tivoli. On the model-driven synthesis of evolvable service choreographies. In *Proceedings of the 12th European Con-*

ference on Software Architecture: Companion Proceedings, pages 1–6, 2018.

[AEH+20] Nicolas Anquetil, Anne Etien, Mahugnon Honoré Houekpetodji, Benoît Verhaeghe, Stéphane Ducasse, Clotilde Toullec, Fatija Djareddir, Jèrome Sudich, and Mustapha Derras. Modular moose: A new generation of software reengineering platform. In *International Conference on Software and Systems Reuse (ICSR'20)*, number 12541, 2020.

[AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of caesarj. *Transactions on Aspect-Oriented Software Development I*, pages 135–173, 2006.

[AIS+19] Marco Autili, Paola Inverardi, Romina Spalazzese, Massimo Tivoli, and Filippo Mignosi. Automated synthesis of application-layer connectors from automata-based specifications. *Journal of Computer and System Sciences*, 104:17–40, 2019.

[AIT18] Marco Autili, Paola Inverardi, and Massimo Tivoli. Choreography realizability enforcement through the automatic synthesis of distributed coordination delegates. *Science of Computer Programming*, 160:3–29, 2018.

[AKL09] Sven Apel, Christian Kastner, and Christian Lengauer. Featurehouse: Language-independent, automated software composition. In *2009 IEEE 31st International Conference on Software Engineering*, pages 221–231. IEEE, 2009.

[AL17] Damian Arellanes and Kung-Kiu Lau. Exogenous connectors for hierarchical service composition. *2017 IEEE 10th Conference on Service-Oriented Computing and Applications (SOCA)*, pages 125–132, 2017.

[ALHL+17] Wesley KG Assunção, Roberto E Lopez-Herrejon, Lukas Linsbauer, Silvia R Vergilio, and Alexander Egyed. Reengineering legacy applications into software product lines: a systematic mapping. *Empirical Software Engineering*, 22(6):2972–3016, 2017.

[ALL+20] Nawel Amokrane, Jannik Laval, Philippe Lanco, Mustapha Derras, and Nejib Moala. Analysis of data exchanges, towards a tooled approach for data interoperability assessment. *Intelligent Systems: Theory, Research and Innovation in Applications*, pages 345–363, 2020.

[ALRS05] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. Featurec++: feature-oriented and aspect-oriented programming in c++. Technical report, Technical report, Department of Computer Science, Otto-von-Guericke . . . , 2005.

[AMS22] Arul Christhuraj Alphonse, Alexandra Martinez, and Akshata Sawant. *MuleSoft for Salesforce Developers: A practitioner's guide to deploying MuleSoft APIs and integrations for Salesforce enterprise solutions*. Packt Publishing Ltd, 2022.

[Bat04] Don Batory. Feature-oriented programming and the ahead tool suite. In *Proceedings. 26th International Conference on Software Engineering*, pages 702–703. IEEE, 2004.

[BB03] Lubomir Bulej and Tomas Bures. A connector model suitable for automatic generation of connectors. Technical report, Citeseer, 2003.

[BBG+13] Nelly Bencomo, Amel Bennaceur, Paul Grace, Gordon Blair, and Valérie Issarny. The role of models@ run. time in supporting on-the-fly interoperability. *Computing*, 95:167–190, 2013.

[BC10] Lamia Berkani and Azeddine Chikh. A process for knowledge reuse in communities of practice of e-learning. *Procedia-Social and Behavioral Sciences*, 2(2):4436–4443, 2010.

[BC19] Oscar Borgogno and Giuseppe Colangelo. Data sharing and interoperability: Fostering innovation and competition through apis. *Computer Law & Security Review*, 35(5):105314, 2019.

[BCD+09] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. Gcm: a grid extension to fractal for autonomous distributed components.

Annals of Telecommunications-annales des télécommunications, 64:5–24, 2009.

[BCK15] Sebastian K Boell and Dubravka Cecez-Kecmanovic. What is an information system? In *2015 48th Hawaii International Conference on System Sciences*, pages 4959–4968. IEEE, 2015.

[BCS12] Carsten Bormann, Angelo P Castellani, and Zach Shelby. Coap: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 16(2):62–67, 2012.

[BD17] Lorenzo Bettini and Ferruccio Damiani. Xtraitj: Traits for the java platform. *Journal of Systems and Software*, 131:419–441, 2017.

[BDSS13] Lorenzo Bettini, Ferruccio Damiani, Ina Schaefer, and Fabio Strocco. Traitrecordj: A programming language with traits and records. *Science of Computer Programming*, 78(5):521–541, 2013.

[Bec95] Kent Beck. Design patterns: Elements of reusable object-oriented software. *IBM Systems Journal*, 34(3):544, 1995.

[BEF+07] A J Berre, Brian Elvesæter, Nicolas Figay, Claudia Guglielmina, Svein G Johnsen, Dag Karlsen, Thomas Knothe, and Sonia Lippe. The athena interoperability framework. In *Enterprise interoperability II: new challenges and approaches*, pages 569–580. Springer, 2007.

[Ben13] Amel Bennaceur. *Synthèse dynamique de médiateurs dans les environnements ubiquitaires*. PhD thesis, Paris 6, 2013.

[Ber22] Alexandre Bergel. *Agile Visualization with Pharo: Crafting Interactive Visual Support Using Roassal*. Springer, 2022.

[BGNI19] Georgios Bouloukakis, Nikolaos Georgantas, Patient Ntumba, and Valérie Issarny. Automated synthesis of mediators for middleware-layer protocol interoperability in the iot. *Future Generation Computer Systems*, 101:1271–1294, 2019.

[BHR15] Françoise Baude, Ludovic Henrio, and Cristian Ruz. Programming distributed and adaptable autonomous components—the

gcm/proactive framework. *Software: Practice and Experience*, 45(9):1189–1227, 2015.

[BI14] Amel Bennaceur and Valérie Issarny. Automated synthesis of mediators to support component interoperability. *IEEE Transactions on Software Engineering*, 41(3):221–240, 2014.

[BKS15] Noor Hasrina Bakar, Zarinah M Kasirun, and Norsaremah Salleh. Feature extraction approaches from natural language requirements for reuse in software product lines: A systematic literature review. *Journal of Systems and Software*, 106:132–149, 2015.

[BN17] Amel Bennaceur and Bashar Nuseibeh. The many facets of mediation: A requirements-driven approach for trading off mediation solutions. In *Managing trade-offs in adaptable software architectures*, pages 299–322. Elsevier, 2017.

[BNDP10] Andrew P Black, Oscar Nierstrasz, Stéphane Ducasse, and Damien Pollet. *Pharo by example*. Lulu. com, 2010.

[Boo06] Grady Booch. The accidental architecture. *IEEE software*, 23(3):9–11, 2006.

[BS06] John Boardman and Brian Sauser. System of systems-the meaning of of. In *2006 IEEE/SMC international conference on system of systems engineering*, pages 6–pp. IEEE, 2006.

[BVT22] Farouk Belkadi, J Vieille, and B Tanous. Towards a smart connector for dynamic interoperability in agile enterprises. *IFAC-PapersOnLine*, 55(10):2342–2347, 2022.

[C+93] Software Productivity Consortium et al. Reuse adoption guidebook, version 02.00. 05. Technical report, Technical report No. SPC-92051-CMC, Software Productivity Consortium ..., 1993.

[Cam21] Guilherme Camposo. *Cloud Native Integration with Apache Camel*. Springer, 2021.

[CBB+00] Ngom Cheng, Valdis Berzins, Swapan Bhattacharya, et al. Automated generation of wrappers for interoperability/june 2000.

American Journal of Orthodontics and Dentofacial Orthopedics, 2000.

[CC19] Binildas Christudas and Binildas Christudas. Activemq. *Practical Microservices Architectural Patterns: Event-Based Java Microservices with Spring Boot and Spring Cloud*, pages 861–867, 2019.

[CD06] David Chen and Nicolas Daclin. Framework for enterprise interoperability. In *Interoperability for Enterprise Software and Applications: Proceedings of the Workshops and the Doctorial Symposium of the Second IFAC/IFIP I-ESA International Conference: EI2N, WSI, IS-TSPQ 2006*, pages 77–88. Wiley Online Library, 2006.

[CFP+01] Carlos Canal, Lidia Fuentes, Ernesto Pimentel, José M Troya, and Antonio Vallecillo. Extending corba interfaces with protocols. *The Computer Journal*, 44(5):448–462, 2001.

[CH07] John Carnell and Rob Harrop. Velocity template engine. *Pro Apache Struts with Ajax*, pages 359–389, 2007.

[Cha04] David A Chappell. *Enterprise service bus: Theory in practice*. " O'Reilly Media, Inc.", 2004.

[CMO+18] Giovanni Ciatto, Stefano Mariani, Andrea Omicini, et al. Respectx: Programming interaction made easy. *Computer Science and Information Systems*, 15(3):655–682, 2018.

[CN02] Paul Clements and Linda Northrop. *Software product lines*. Addison-Wesley Boston, 2002.

[Coh02] Gary Cohen. securing ftp. *login Usenix Mag.*, 27(4), 2002.

[Crn01] Ivica Crnkovic. Component-based software engineering—new challenges in software development. *Software focus*, 2(4):127–133, 2001.

[CZ13] Zhuo Cai and XQ Zhang. Overview of sca 4.0 specification. *J. Commun. Technol*, 46(7):126–128, 2013.

[CZVD+09] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehen-

sion through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[DAB+01] Robert H Dolin, Liora Alschuler, Calvin Beebe, Paul V Biron, Sandra Lee Boyer, Daniel Essin, Elliot Kimber, Tom Lincoln, and John E Mattison. The hl7 clinical document architecture. *Journal of the American Medical Informatics Association*, 8(6):552–569, 2001.

[Dan09] George Danezis. Traffic analysis of the http protocol over tls, 2009.

[Dau22] Bekim Dauti. *Windows Server 2022 Administration Fundamentals: A beginner's guide to managing and administering Windows Server environments*. Packt Publishing Ltd, 2022.

[DC92] Richard De Courcy. Les systèmes d'information en réadaptation. *Québec, Réseau international CIDIH et facteurs environnementaux*, 5(1-2):7–10, 1992.

[DMN+23] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jack Jiang. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software*, 203:111734, 2023.

[DNO98] Enrico Denti, Antonio Natali, and Andrea Omicini. On the expressive power of a language for programming coordination media. In *Proceedings of the 1998 ACM symposium on Applied Computing*, pages 169–177, 1998.

[DRB+13] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An exploratory study of cloning in industrial software product lines. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 25–34. IEEE, 2013.

[DVD19] Clement Dutriez, Benoît Verhaeghe, and Mustapha Derras. Switching of gui framework: the case from spec to spec 2. 2019.

[EKLG+03] Florida Estrella, Zsolt Kovacs, Jean-Marie Le Goff, Richard McClatchey, Tony Solomonides, and Norbert Toth. Pattern reification

as the basis for description-driven systems. *Software & Systems Modeling*, 2:108–119, 2003.

[FGFdAM14] Gabriel Coutinho Sousa Ferreira, Felipe Nunes Gaia, Eduardo Figueiredo, and Marcelo de Almeida Maia. On the use of feature-oriented programming for evolving software product lines—a comparative study. *Science of Computer programming*, 93:65–85, 2014.

[FGM+97] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, and Tim Berners-Lee. Rfc2068: Hypertext transfer protocol–http/1.1, 1997.

[FLLHE15] Stefan Fischer, Lukas Linsbauer, Roberto E Lopez-Herrejon, and Alexander Egyed. The ecco tool: Extraction and composition for clone-and-own. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 665–668. IEEE, 2015.

[GA18] Didem Gürdür and Fredrik Asplund. A systematic review to merge discourses: Interoperability, integration and cyber-physical systems. *Journal of Industrial information integration*, 9:14–23, 2018.

[Gar13] Nishant Garg. *Apache kafka*. Packt Publishing Birmingham, UK, 2013.

[GG21] Felipe Gutierrez and Felipe Gutierrez. Spring cloud data flow: Introduction and installation. *Spring Cloud Data Flow: Native Cloud Orchestration Services for Microservice Applications on Modern Runtimes*, pages 209–262, 2021.

[Gio12] W Gio. Mediators, concepts and practice to appear in studies information reuse and integration in academia and industry, 2012.

[GMFFGS09] Iván García-Magariño, Rubén Fuentes-Fernández, and Jorge J Gómez-Sanz. Guideline for the definition of emf metamodels using an entity-relationship approach. *Information and Software Technology*, 51(8):1217–1230, 2009.

[Gom05] Hassan Gomaa. Designing software product lines with uml. In *29th Annual IEEE/NASA Software Engineering Workshop-Tutorial Notes (SEW'05)*, pages 160–216. IEEE, 2005.

[GON19] Lina Garcés, Flavio Oquendo, and Elisa Yumi Nakagawa. Software mediators as first-class entities of systems-of-systems software architectures. *Journal of the Brazilian Computer Society*, 25(1):1–23, 2019.

[HBS+02] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. Java message service. *Sun Microsystems Inc., Santa Clara, CA*, 9, 2002.

[Hla22] Nicolas Hlad. *IsiSPL: an automated process to facilitate the engineering of software product lines according to a reactive or extractive industrial adoption strategy.* PhD thesis, Université de Montpellier, 2022.

[HMZ09] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. 2009.

[Hoh06] Gregor Hohpe. Conversation patterns: Workshop report. In *Dagstuhl Seminar*, volume 7, 2006.

[HRR14] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. Montiarc-architectural modeling of interactive distributed and cyber-physical systems. *arXiv preprint arXiv:1409.6578*, 2014.

[HTSC08] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. Mqtt-s—a publish/subscribe protocol for wireless sensor networks. In *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08)*, pages 791–798. IEEE, 2008.

[HW04] Gregor Hohpe and Bobby Woolf. Enterprise integration patterns: Designing, building, and deploying messaging solutions. 2004.

[IRHBJ16] Matías Ibáñez, Cristian Ruz, Ludovic Henrio, and Javier Bustos-Jiménez. Reconfigurable applications using gcmscript. *IEEE cloud computing*, 3(3):30–39, 2016.

[IT13] Paola Inverardi and Massimo Tivoli. Automatic synthesis of modular connectors via composition of protocol mediation patterns.

In *2013 35th International Conference on Software Engineering (ICSE)*, pages 3–12. IEEE, 2013.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of computer programming*, 72(1-2):31–39, 2008.

[JBF09] Slinger Jansen, Sjaak Brinkkemper, and Anthony Finkelstein. Business network management as a survival strategy: A tale of two software ecosystems. *Iwseco@ Icsr*, 2009, 2009.

[JSS+12] Sung-Shik TQ Jongmans, Francesco Santini, Mahdi Sargolzaei, Farhad Arbab, and Hamideh Afsarmanesh. Automatic code generation for the orchestration of web services with reo. In *Service-Oriented and Cloud Computing: First European Conference, ESOCC 2012, Bertinoro, Italy, September 19-21, 2012. Proceedings 1*, pages 1–16. Springer, 2012.

[JSS+14] Sung-Shik TQ Jongmans, Francesco Santini, Mahdi Sargolzaei, Farhad Arbab, and Hamideh Afsarmanesh. Orchestrating web services using reo: from circuits and behaviors to automatically generated code. *Service Oriented Computing and Applications*, 8:277–297, 2014.

[KA09] Christian Kästner and Sven Apel. Virtual separation of concerns-a second chance for preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of aspectj. In *ECOOP 2001—Object-Oriented Programming: 15th European Conference Budapest, Hungary, June 18–22, 2001 Proceedings 15*, pages 327–354. Springer, 2001.

[KHS+14] Jonathan Koscielny, Sönke Holthusen, Ina Schaefer, Sandro Schulze, Lorenzo Bettini, and Ferruccio Damiani. Deltaj 1.5: delta-oriented programming for java 1.5. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming*

on the Java platform: Virtual machines, Languages, and Tools, pages 63–74, 2014.

[KKHL10] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. Typechef: Toward type checking# ifdef variability in c. In *Proceedings of the 2nd international workshop on feature-oriented software development*, pages 25–32, 2010.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

[KRU+03] Charles Keating, Ralph Rogers, Resit Unal, David Dryer, Andres Sousa-Poza, Robert Safford, William Peterson, and Ghaith Rabadi. System of systems engineering. *Engineering Management Journal*, 15(3), 2003.

[KTS+09] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. Featureide: A tool framework for feature-oriented software development. In *2009 ieee 31st international conference on software engineering*, pages 611–614. IEEE, 2009.

[KU22] Siva Prasad Reddy Katamreddy and Sai Subramanyam Upadhyayula. Getting started with spring boot. In *Beginning Spring Boot 3: Build Dynamic Cloud-Native Java Applications and Microservices*, pages 29–45. Springer, 2022.

[LAL+10] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 105–114, Cape Town, South Africa, 2010. ACM Press.

[LATN+23] Jannik Laval, Nawel Amokrane, Boubou Thiam Niang, Mustapha Derras, and Néjib Moalla. Data interoperability assessment, case of

messaging-based data exchanges. *Journal of Software: Evolution and Process*, page e2538, 2023.

[LDDF11] Jannik Laval, Simon Denier, Stéphane Ducasse, and Jean-Rémy Falleri. Supporting simultaneous versions for software evolution assessment. *Science of Computer Programming*, 76(12):1177–1193, 2011.

[LEW05] Kung-Kiu Lau, Perla Velasco Elizondo, and Zheng Wang. Exogenous connectors for software components. In *International Symposium on Component-Based Software Engineering*, pages 90–106. Springer, 2005.

[Lie23] Michael Lienhardt. Pydop: A generic python library for delta-oriented programming. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference-Volume B*, pages 30–33, 2023.

[LL09] Kathryn B Laskey and Kenneth Laskey. Service oriented architecture. *Wiley Interdisciplinary Reviews: Computational Statistics*, 1(1):101–105, 2009.

[LLA+20] Leo Liu, Weizi Li, Naif R Aljohani, Miltiadis D Lytras, Saeed-Ul Hassan, and Raheel Nawaz. A framework to evaluate the interoperability of information systems–measuring the maturity of the business process alignment. *International Journal of Information Management*, 54:102153, 2020.

[LWK10] Philip Langer, Manuel Wimmer, and Gerti Kappel. Model-to-model transformations by demonstration. In *International Conference on Theory and Practice of Model Transformations*, pages 153–167. Springer, 2010.

[LY02] Kalle Lyytinen and Youngjin Yoo. Ubiquitous computing. *Communications of the ACM*, 45(12):63–96, 2002.

[MDCB17] Rita Suzana P Maciel, José Maria N David, Daniela Claro, and Regina Braga. Full interoperability: Challenges and opportuni-

ties for future information systems. *Sociedade Brasileira de Computação*, 2017.

[MDG08] Mario Mustra, Kresimir Delac, and Mislav Grgic. Overview of the dicom standard. In *2008 50th International Symposium ELMAR*, volume 1, pages 39–44. IEEE, 2008.

[Men07] Falko Menge. Enterprise service bus. In *Free and open source software conference*, volume 2, pages 1–6, 2007.

[MG00] Pierre-Alain Muller and Nathalie Gaertner. *Modélisation objet avec UML*, volume 514. Eyrolles Paris, 2000.

[MLD+13] Flávio Medeiros, Thiago Lima, Francisco Dalton, Márcio Ribeiro, Rohit Gheyi, and B ANDFONSECA. Colligens: A tool to support the development of preprocessor-based software product lines in c. In *Proc. Brazilian Conf. Software: Theory and Practice (CBSoft)*, 2013.

[MM20] Tarnia Major and Joseph Mangano. Modernising payments messaging: The iso 20022 standard. *1. 1 Managing the Risks of Holding Self-securitisations as Collateral 2. 11 Government Bond Market Functioning and COVID-19 3. The Economic Effects of Low Interest Rates and Unconventional 21 Monetary Policy 4. Retail Central Bank Digital Currency: Design Considerations, Rationales*, page 66, 2020.

[MMHA15] Saleh Majd, Abel Marie-Hélène, and Mishra Alok. An architectural model for system of information systems. In *On the Move to Meaningful Internet Systems: OTM 2015 Workshops: Confederated International Workshops: OTM Academy, OTM Industry Case Studies Program, EI2N, FBM, INBAST, ISDE, META4eS, and MSC 2015, Rhodes, Greece, October 26-30, 2015. Proceedings*, pages 411–420. Springer, 2015.

[MMP00] Nikunj R Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd international conference on Software engineering*, pages 178–187, 2000.

[MMS19] Agustín Mántaras, Mántaras, and Srivastava. *BizTalk Server 2016*. Springer, 2019.

[Mon03] Paul B Monday. Implementing the data transfer object pattern. In *Web Services Patterns: Java™ Platform Edition*, pages 279–295. Springer, 2003.

[MZB+17] Jabier Martinez, Tewfik Ziadi, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Bottom-up technologies for reuse: automated extractive adoption of software product lines. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 67–70. IEEE, 2017.

[NAG19] Mahda Noura, Mohammed Atiquzzaman, and Martin Gaedke. Interoperability in internet of things: Taxonomies and open challenges. *Mobile networks and applications*, 24:796–809, 2019.

[Pan15] Chandan Pandey. *Spring integration essentials*. Packt Publishing Ltd, 2015.

[Pat17] Sanjay Patni. *Pro RESTful APIs*. Springer, 2017.

[PBVDL05] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. *Software product line engineering: foundations, principles, and techniques*, volume 1. Springer, 2005.

[PED19] Cristina Paniagua, Jens Eliasson, and Jerker Delsing. Interoperability mismatch challenges in heterogeneous soa-based systems. In *2019 IEEE International Conference on Industrial Technology (ICIT)*, pages 788–793. IEEE, 2019.

[Pel03] Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[Pir21] Vartan Piroumian. Digital twins: Universal interoperability for the digital age. *Computer*, 54(1):61–69, 2021.

[PKK+15] Christopher Pietsch, Timo Kehrer, Udo Kelter, Dennis Reuling, and Manuel Ohrndorf. Sipl–a delta-based modeling framework for software product line engineering. In *2015 30th IEEE/ACM Inter-*

national Conference on Automated Software Engineering (ASE), pages 852–857. IEEE, 2015.

[PR22] William Penberthy and Steve Roberts. Microsoft sql server. In *Pro. NET on Amazon Web Services: Guidance and Best Practices for Building and Deployment*, pages 303–329. Springer, 2022.

[Pra21] Ambar Prajapati. Amqp and beyond. In *2021 International Conference on Smart Applications, Communications and Networking (SmartNets)*, pages 1–6. IEEE, 2021.

[PRS+16] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of json schema. In *Proceedings of the 25th international conference on World Wide Web*, pages 263–273, 2016.

[PTD+19] Stefan Profanter, Ayhun Tekat, Kirill Dorofeev, Markus Rickert, and Alois Knoll. Opc ua versus ros, dds, and mqtt: Performance evaluation of industry 4.0 protocols. In *2019 IEEE International Conference on Industrial Technology (ICIT)*, pages 955–962. IEEE, 2019.

[PTS+16] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. Synchronizing software variants with variantsync. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 329–332, 2016.

[RBVL18] Felix Maximilian Roth, Christian Becker, Germán Vega, and Philippe Lalanda. Xware—a customizable interoperability framework for pervasive computing systems. *Pervasive and mobile computing*, 47:13–30, 2018.

[RDR03] Claudio Riva and Christian Del Rosso. Experiences with software product family evolution. In *Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings.*, pages 161–169. IEEE, 2003.

[RG02] Mark Richters and Martin Gogolla. Ocl: Syntax, semantics, and tools. In *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, pages 42–68. Springer, 2002.

[Ris07] Dejan Risimić. An integration strategy for large enterprises. *Yugoslav Journal of Operations Research*, 17(2):209–222, 2007.

[S+06] Douglas C Schmidt et al. Model-driven engineering. *Computer-IEEE Computer Society-*, 39(2):25, 2006.

[S+15] Martin Sústrik et al. Zeromq. *Introduction Amy Brown and Greg Wilson*, page 16, 2015.

[Sar19] Rishi Kanth Saripalle. Fast health interoperability resources (fhir): current status in the healthcare system. *International Journal of E-Health and Medical Communications (IJEHMC)*, 10(1):76–93, 2019.

[SBB+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Software Product Lines: Going Beyond: 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings 14*, pages 77–91. Springer, 2010.

[Sch05] Hans Jochen Scholl. Interoperability in e-government: More than just smart middleware. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, pages 123–123. IEEE, 2005.

[SCT13] Andy Stanford-Clark and Hong Linh Truong. Mqtt for sensor networks (mqtt-sn) protocol specification. *International business machines (IBM) Corporation version*, 1(2):1–28, 2013.

[SHU+13] A-D Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, et al. Mining features from the object-oriented source code of software variants by combining lexical and structural similarity. In *2013 IEEE 14th International Conference on Information Reuse & Integration (IRI)*, pages 586–593. IEEE, 2013.

[SI10] Romina Spalazzese and Paola Inverardi. Mediating connector patterns for components interoperability. In *Software Architecture: 4th European Conference, ECSA 2010, Copenhagen, Denmark, August 23-26, 2010. Proceedings 4*, pages 335–343. Springer, 2010.

[SL99] Matti Sinko and Erno Lehtinen. *The challenges of ICT*. Citeseer, 1999.

[SL+16] Ashutosh Satapathy, Jenila Livingston, et al. A comprehensive survey on ssl/tls and their vulnerabilities. *International Journal of Computer Applications*, 153(5):31–38, 2016.

[SM17] Dipa Soni and Ashwin Makwana. A survey on mqtt: a protocol of internet of things (iot). In *International conference on telecommunication, power analysis and computing techniques (ICTPACT-2017)*, volume 20, pages 173–177, 2017.

[SMR+12] Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, and Jean-Bernard Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. *Software: Practice and Experience*, 42(5):559–583, 2012.

[SPCF04] Joseph M Schlesselman, Gerardo Pardo-Castellote, and Bert Farabaugh. Omg data-distribution service (dds): architectural update. In *IEEE MILCOM 2004. Military Communications Conference, 2004.*, volume 2, pages 961–967. IEEE, 2004.

[SPE17] Romina Spalazzese, Patrizio Pelliccione, and Ulrik Eklund. Intero: an interoperability model for large systems. *IEEE Software*, 37(3):38–45, 2017.

[SRA19] Maya RA Setyautami, Rafiano R Rubiantoro, and Ade Azurat. Model-driven engineering for delta-oriented software product lines. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 371–377. IEEE, 2019.

[SSS17] Anas Shatnawi, Abdelhak-Djamel Seriai, and Houari Sahraoui. Recovering software product line architecture of a family of object-

oriented product variants. *Journal of Systems and Software*, 131:325–346, 2017.

[Tar12] Sasu Tarkoma. *Publish/subscribe systems: design and principles*. John Wiley & Sons, 2012.

[Tic99] Sander Tichelaar. Famix java language plug-in 1.0. *Technical Report*, 1999.

[Tos15] Martin Toshev. *Learning RabbitMQ*. Packt Publishing Ltd, 2015.

[Van01] Frédéric Vandenberghe. Reification: History of the concept. *International Encyclopedia of the Social and Behavioral Sciences*, 19:12993–12996, 2001.

[vdML02] Thomas von der Maßen and Horst Lichter. Modeling variability by uml use case diagrams. In *Proceedings of the International Workshop on Requirements Engineering for product lines*, pages 19–25. Citeseer, 2002.

[WCC+95] Brian A Wichmann, AA Canning, DL Clutterbuck, LA Winsborrow, NJ Ward, and D William R Marsh. Industrial perspective on static analysis. *Software Engineering Journal*, 10(2):69, 1995.

[Weg96] Peter Wegner. Interoperability. *ACM Computing Surveys (CSUR)*, 28(1):285–287, 1996.

[Wie92] Gio Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(3):38–49, 1992.

[Win92] Patrick Henry Winston. *Artificial intelligence*. Addison-Wesley Longman Publishing Co., Inc., 1992.

[WKS+16] Tim Winkelmann, Jonathan Koscielny, Christoph Seidl, Sven Schuster, Ferruccio Damiani, Ina Schaefer, et al. Parametric deltaj 1.5: propagating feature attributes into implementation artifacts. In *CEUR WORKSHOP PROCEEDINGS*, volume 1559, pages 40–54. CEUR-WS, 2016.

[YQC+19] Jiang Yongguo, Liu Qiang, Qin Changshuai, Su Jian, and Liu Qianqian. Message-oriented middleware: A review. In *2019 5th International Conference on Big Data Computing and Communications (BIGCOM)*, pages 88–97. IEEE, 2019.

[YS94] Daniel M Yellin and Robert E Strom. Interfaces, protocols, and the semi-automatic construction of software adaptors. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 176–190, 1994.

[ZDAT22] Oleksandr Zaitsev, Stéphane Ducasse, Nicolas Anquetil, and Arnaud Thiefaine. How libraries evolve: A survey of two industrial companies and an open-source community. In *2022 29th Asia-Pacific Software Engineering Conference (APSEC)*, pages 309–317. IEEE, 2022.

[ZJYR23] Jun-Jie Zhu, Jinyue Jiang, Meiqi Yang, and Zhiyong Jason Ren. Chatgpt and environmental research. *Environmental Science & Technology*, 2023.

[ZXZ+23] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*, 2023.