Dissertation Thesis

Towards a Normalized Systems Gateway Ontology for Conceptual Models

by

Marek Suchánek

Supervised by *Robert Pergl* and *Herwig Mannaert*

Thesis submitted to the Faculty of Information Technology, Czech Technical University in Prague, in partial fulfilment of the requirements for the degree of *Doctor* (in Czech: *Doktor*).

Doctoral study programme: *Informatics*

Department of Software Engineering
Faculty of Information Technology
Czech Technical University in Prague

Prague, November 2023

Thesis submitted for the degree of *Doctor of Applied Economics* (in Dutch: *Doctor in de Toegepaste Economische Wetenschappen*) at the University of Antwerp to be defended by Marek Suchánek.

Doctoral study programme: *Applied Economics*

Department of Management Information Systems
Faculty of Business and Economics
University of Antwerp

Antwerp, November 2023

Suchánek, M. (2023). *Towards a Normalized Systems Gateway Ontology for Conceptual Models* [Doctoral thesis, Czech Technical University in Prague & University of Antwerp]. ISBN: 978-90-5728-828-9. `http://purl.org/nsgo4cm`

Deposit number: `D/2023/12.293/47`

Supervisor (Czech Technical University in Prague):

doc. Robert Pergl, Ph.D.
Department of Software Engineering
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Prague 6
Czech Republic

Supervisor (University of Antwerp):

prof. Herwig Mannaert
Department of Management Information Systems
Faculty of Business and Economics
University of Antwerp
Prinsstraat 13
2000 Antwerp
Belgium

Abstract

In software engineering, conceptual modelling is a technique for describing a problem domain related to a software system. Although its primary purpose is to promote human understanding and communication, conceptual models can be (re)used to generate the desired software systems or their fragments. Normalized Systems (NS) describe how to create enterprise software systems by utilising code generation for evolvability and sustainability. It uses modelling of the so-called Elements – building blocks ensuring evolvability. This dissertation thesis designs a solution to transform conceptual models into Normalized Systems (and vice versa) while maintaining mutual consistency. The Gateway Ontology designed on the basis of RDF technologies provides means to define the mappings between arbitrary conceptual modelling languages and Normalized Systems. It supports different types of modelling and enables semantic integration between various models. The thesis also covers the design and implementation of transformation execution that allows one to transform models of selected modelling languages. The overall solution considers evolvability on multiple levels – adopting changes in tools, models, metamodels, and mappings. According to the Design Science Research methodology, the research artefacts were refined and evaluated in design cycle iterations while confronted with the knowledge base and the NS environment. As part of the evaluation, we demonstrate several use cases with different conceptual models, their integration, and transformation into NS.

In particular, the main contributions of the dissertation thesis are as follows:

(i) the design of model-to-model transformation based on RDF/OWL,

(ii) the evolvable transformation between NS elements and RDF/OWL,

(iii) ontologies for conceptual modelling using RDF,

(iv) the SPARQL query generation method for RDF transformations,

(v) the framework for transformations between NS and conceptual models.

Title: Towards a Normalized Systems Gateway Ontology for Conceptual Models
Keywords: Normalized Systems • Ontology Engineering • Model-Driven Development • Semantic Integration • Model Transformation • Metamodelling

Abstrakt

Konceptuální modelování slouží v oblasti softwarového inženýrství k popisu problémové domény související se softwarovým systémem. Ačkoliv primárním účelem je zlepšení lidského porozumění a komunikace, konceptuální modely mohou být (znovu)použity pro generování softwarových systémů nebo jejich částí. Normalizované systémy (NS) popisují, jak vytvářet podnikové informační systémy pomocí generování kódu za účelem evolvability a udržitelnosti. NS používá modelování takzvaných Elementů – stavebních bloků garantujících evolvabilitu. Tato dizertační práce navrhuje řešení pro transformaci konceptuálních modelů na Normalizované systémy (a zpět) se zachováním vzájemné konzistence. Bránová ontologie založená na RDF technologiích umožňuje definovat mapování mezi libovolnými konceptuálními modelovacími jazyky a Normalizovanými systémy. Tento návrh podporuje různé typy modelování a umožňuje také sémantickou integraci mezi modely. Celkové řešení zohledňuje evolvabilitu na více úrovních – změny v nástrojích, modelech, metamodelech i mapováních. V souladu s metodologií Design Science Research byly navržené artefakty postupně vylepšovány a vyhodnocovány v návrhových cyklech a současně konfrontovány se znalostní bází i prostředím NS. V rámci vyhodnocení, demonstrujeme několik případů užití s různými konceptuálními modely, jejich integraci a transformaci do NS.

Hlavními přínosy této dizertační práce jsou zejména:

(i) návrh transformací model-model založený na RDF/OWL,

(ii) evolvabilní transformace mezi NS Elementy a RDF/OWL,

(iii) ontologie pro konceptuální modelování v RDF,

(iv) metoda generování SPARQL dotazů pro RDF transformace,

(v) framework pro transformace mezi NS a konceptuálními modely.

Název práce: Budování bránové ontologie mezi Normalizovanými systémy a konceptuálními modely
Klíčová slova: Normalized Systems • Inženýrství ontologií • Modelem řízený vývoj • Sémantická integrace • Transformace modelů • Metamodelování

Abstract

In het kader van software engineering is conceptuele modellering een techniek om een bepaald domein te beschrijven vanuit het standpunt van een informatiesysteem. Hoewel het primaire doel ervan is het bevorderen van menselijk begrip en communicatie, kunnen dergelijke conceptuele modellen worden gebruikt om de gewenste informatiesystemen geheel of gedeeltelijk te genereren. Normalized Systems (NS) beschrijft hoe informatiesystemen automatisch kunnen worden gecreëerd door gebruik te maken van codegeneratie vertrekkende van de modellering van zogenaamde Elementen – bouwstenen die garanties bieden op het vlak van evolueerbaarheid. Dit proefschrift ontwerpt een oplossing om conceptuele modellen om te zetten in genormaliseerde systemen (en vice versa) met behoud van onderlinge consistentie. De ontworpen Gateway Ontologie is gebaseerd op RDF technologie en biedt voorzieningen om koppelingen te definiëren tussen diverse talen voor conceptuele modellering enerzijds, en genormaliseerde systemen anderzijds. Het ondersteunt verschillende soorten modellering en maakt semantische integratie tussen verschillende modellen mogelijk. Het proefschrift behandelt ook het ontwerp en de implementatie van de transformatie van modellen voor een aantal geselecteerde modelleringstalen. De algemene oplossing houdt rekening met evolueerbaarheid op meerdere niveaus, namelijk het aannemen van veranderingen in tools, modellen, metamodellen en mappings. Volgens de Design Science Research methodologie werden de onderzoeksartifacten verfijnd en geëvalueerd in ontwerpcyclus iteraties terwijl ze werden geconfronteerd met de kennisbasis en de NS omgeving. Als onderdeel van de evaluatie worden verscheidene use cases gedemonstreerd voor de integratie met en transformatie naar NS van conceptuele modellen.

In het bijzonder zijn de belangrijkste bijdragen van het proefschrift als volgt:

(i) het ontwerp van model-naar-model transformatie op basis van RDF/OWL,

(ii) een evolueerbare transformatie tussen NS-elementen en RDF/OWL,

(iii) de uitwerking van ontologieën voor conceptuele modellering met RDF,

(iv) creatie van een SPARQL-query-generatiemethode voor RDF-transformaties,

(v) een raamwerk voor transformaties tussen NS en conceptuele modellen.

Proefschrift Titel: Ontwikkeling van een Normalized Systems Gateway-Ontologie voor conceptuele modellen
Trefwoorden: Normalized Systems • Ontologie Engineering • Modelgestuurde Ontwikkeling • Semantische Integratie • Modeltransformatie • Metamodellering

Acknowledgements

I would like to begin by expressing my appreciation to my dedicated supervisors, Robert Pergl and Herwig Mannaert. Their support, vast expertise, and insightful suggestions were instrumental in guiding me through every stage of this research journey. Their supervision helped me overcome any obstacles that came my way and reach the successful completion of my work.

I extend my sincere gratitude to NSX bvba and its employees for their technical consultations and welcoming approach during my visits. I also wish to thank professor Jan Verelst for organising the Normalized Systems Summer Schools and facilitating the joint double-degree PhD agreement between CTU and UAntwerpen. I would like to express my huge appreciation to Lenka Fryčová and Nele Gernaey for the invaluable assistance with the administrative tasks related to the double-degree program and PhD study.

I would like to give special thanks to the staff at the Department of Software Engineering (FIT CTU in Prague), particularly the department secretary, Adéla Svítková, the head of the department, Michal Valenta, as well as the former secretary, Alena Libánská. Their friendly and flexible approach created a pleasant atmosphere for my research. I would also like to thank my colleagues from the Centre for Conceptual Modelling and Implementation (CCMi) research group, namely Jan Slifka, Vojtěch Knaisl, and David Šenkýř, for their constructive feedback, stimulating discussions, and research collaboration. I am grateful for the support of the people around me who kept me focused on my research and allowed me to push my work forward.

I am also grateful for the practical experience gained through my involvement in projects such as Data Stewardship Wizard (Horizon 2020 grant No. 871075, MEYS grants No. LM2018131 and No. LM2023055) and Smart City Compass (TACR No. TJ02000344). My research has also been partially supported by the Grant Agency of the Czech Technical University in Prague, specifically through grants “Advanced Research in Software Engineering” (No. SGS17/211/OHK3/3T/18 and No. SGS20/209/OHK3/3T/18).

Last but not least, I want to express my deepest gratitude to my family for their unwavering love and support. They have provided me with the best environment to pursue my studies, and I am incredibly grateful for their presence in my life.

Contents

List of Figures

List of Listings

List of Algorithms

List of Tables

List of Acronyms

A | B | C | D | E | F | G | I | J | M | N | O | P | Q | R | S | T | U | V | W | X

A

AI artificial intelligence

ALF Action Language for Foundational UML

API Application Programming Interface

ATL ATL Transformation Language

B

BA Business Architecture

BCD Bank Contents Table

BORM Business Object Relationship Modelling

BPDM Business Process Definition Metamodel

BPEL Business Process Execution Language

BPMN Business Process Model and Notation

C

CABE computer-aided business engineering

CASE computer-aided software engineering

CASL Common Algebraic Specification Language

CbyC Correctness by Construction

CIM Computation-independent model

CMP Conceptual Model Programming

CRUD Create, Read, Update, Delete

D

DEMO Design & Engineering Methodology for Organizations

DL description logic

DOAP Description of a Project

DOGMA Developing Ontology-Grounded Methods and Applications

DRY Don't Repeat Yourself

DSR Design Science Research

E

EER Enhanced Entity–Relationship

EMF Eclipse Modeling Framework

ER Entity-Relationship

F

FAIR findable, accessible, interoperable, and re-usable

FOAF Friend of a Friend

fUML Foundational UML

G

GCL Guarded Command Language

I

IDE Integrated Development Environment

IFML Interaction Flow Modeling Language

IRI Internationalized Resource Identifier

J

JPEG Joint Photographic Experts Group

JSON JavaScript Object Notation

JSON-LD JSON for Linking Data

JVM Java Virtual Machine

M

MDA Model Driven Architecture

MDD Model-Driven Development

MDE Model-Driven Engineering

MDT Model Development Tools

MOF Meta-Object Facility

MOFM2T MOF Model to Text Transformation Language

MVC Model-View-Controller

MVP Model-View-Presenter

N

NEMO Ontology & Conceptual Modeling Research Group

NLP natural language processing

NS Normalized Systems

NST Normalized Systems Theory

O

OASIS Open and Active Specification of Information Systems

OCD Organisation Construction Diagram

OCL Object Constraint Language

OCMI Ontology for Conceptual Models Integration

OCML Operational Conceptual Modelling Language

OCR Optical Character Recognition

OMG Object Management Group

OR Objects Relations

ORM	Object-Role Modeling
OWL	Web Ontology Language
P	
PDF	Portable Document Format
PIM	Platform-independent model
PNG	Portable Network Graphics
POJO	Plain Old Java Object
PSD	Process Structure Diagram
PSM	Platform-specific model
Q	
QVT	Query/View/Transformation
R	
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
RML	RDF Mapping Language
S	
SBMO	SPARQL-Based Mapping Ontology (SBMO)
SHACL	Shapes Constraint Language
ShEx	Shape Expressions
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
SVG	Scalable Vector Graphics
T	
TEMOS	Textual Modelling System
TPT	Transaction Product Table

U

UFO Unified Foundational Ontology

UI User Interface

UML Unified Modeling Language

URI Uniform Resource Identifier

URL Uniform Resource Locator

UUID Universally Unique Identifier

UWE UML-based Web Engineering

V

VCS Version Control System

VDM Vienna Development Method

W

W3C World Wide Web Consortium

WIDOCO WIzard for DOCumenting Ontologies

X

XMI XML Metadata Interchange

XML Extensible Markup Language

XSD XML Schema Definition

XSL Extensible Stylesheet Language

XSLT Extensible Stylesheet Language Transformations

xtUML Executable Translatable UML

Chapter 1

Introduction

"The precise statement of any problem is the most important step in its solution."

Edwin Bliss

The introductory chapter provides a comprehensive overview of the motivations behind the research, identifies the associated problems, and presents the goals of the dissertation thesis. It introduces the research problem, along with our research hypothesis and related research objectives. To ensure a systematic and rigorous approach, we adopt the Design Science Research (DSR) methodology, which is concisely explained within this chapter, thereby establishing a robust framework for our research process.

Furthermore, this chapter includes a succinct yet informative summary of the related work and previous results that are pertinent to our study. These findings are then explored in greater depth in Chapter 2, allowing for a comprehensive understanding of the existing body of knowledge in the field.

Lastly, the chapter highlights the significant contributions made by this dissertation thesis, showcasing its originality and potential impact. Additionally, it provides a clear outline of the thesis structure, offering a roadmap to navigate through the content and facilitating a coherent reading experience.

1.1 Motivation

Conceptual modelling is an activity of describing a particular world using concepts, i.e. abstractions that allow omitting details unnecessary for the model. It has solid roots in philosophy and cognitive sciences. As such, conceptual models are being used in various forms since time immemorial. One can even see cave paintings as conceptual models of how people can hunt mammoths (or other animals based on the level of abstraction). Using abstract thinking and building worlds of concepts is something natural for human beings. Therefore, conceptual modelling promotes understanding and communication between people, i.e. capturing and sharing knowledge about something.

In software engineering, conceptual modelling is a significant part of the analysis that is essential to build information systems that serve its purpose well. To support an organisation with a software system, one must understand its needs, processes, structure, rules, and other aspects. It is only possible to design the system to meet expectations and support the organisation efficiently. There are many methodologies and modelling languages for capturing those various aspects. Some try to cover everything from a high-level perspective, others are oriented on a specific aspect (e.g. processes), but in detail, then there are some oriented on ontological meanings, and others on a technical solution. The software analyst must be skilled and experienced to pick the suitable "tools" for a given case and use them well.

Conceptual models in the software development process serve as a source of knowledge in the analysis phase. However, the models can be then used for proceeding to implementation efficiently. Yet again, there are many methods, tools, and frameworks for generating software source code from models. Some can produce skeletons for a whole simple application; some generate only the data layer entities, and so on. As it helps with the development at the beginning, it still has issues maintainability and hence sustainability. It is caused by the fact that the generated software from models usually needs adjustments in additional source code written by programmers. It results in breaking the consistency with the conceptual model and additional complexity.

Every software information system needs to be updated over time as the organisation needs are changing and technologies move forward. The more complex the system, the more resources are necessary for each next change to be added. According to best practises and internal conventions, maintaining the system sustainable, with good documentation, and still supporting the organisation is a challenge. The system often reaches the level of complexity when it is more efficient (in terms of resources) to implement the system again from scratch. Those problems are addressed by Normalized Systems Theory (NST) [1].

Normalized Systems (NS) describe how to build (software) systems that are evolvable, i.e. changes can be easily adapted without adverse effects on the whole system. It does so by using various principles, creating a fine-grained modular structure, and generating the system using templates called *Expanders* from models of *Elements*. In a sense, those models are also conceptual models as a software analyst tries to describe the domain (e.g. an organisation or its part) using the elements. Still, it requires additional and precise knowledge related only to the NS as the modelling approach is significantly different from the ontological view. In addition, models contain technical details that are usually irrelevant during analysis. Finally, some common construct from conceptual modelling that is natural for people (e.g. inheritance or meronymy relations) are not present in NS for evolvability reasons.

In this dissertation thesis, we are dealing with connecting the world of conceptual modelling with its many modelling languages and methodologies with the world of evolvable software build using Normalized Systems. Rather than covering a subset of the modelling languages, we design a framework that supports transformation between NS models and various other conceptual models based on ontological descriptions of those transformations. There are several partial issues addressed in our work related to transformation descriptions, diversity of conceptual modelling languages, knowledge integration, and maintaining consistency between models. The core of our solution, as well as this thesis, is called *Normalized Systems Gateway Ontology for Conceptual Models* because it is an ontology opening the doors to conceptual models into the world of NS while maintaining the consistency with the original, i.e. there is also a way back possible.

1.2 Problem Statement

The problem addressed by this thesis is overcoming the barrier between conceptual modelling and Normalized Systems as a way of model-driven development targeting evolvable information systems. Figure 1.1 shows the barrier together with its related issues that are our research objectives RO1–RO4. First, at the conceptual model level, many modelling languages can be used to describe a problem domain. Different modelling languages are suitable for different cases, and usually, even a combination of multiple is desired. Any information from conceptual models should not be lost during the transformation to NS models. Although expressiveness may differ in various models, the models on both sides of the abyss should remain consistent. Finally, as the transformation is tied to Normalized Systems that stress the evolution of systems, the transformation needs to be also evolvable, designed as a fine-grained modular structure, and easily adapt to changes.

RH: Research Hypothesis *Normalized Systems models capture domain-related knowledge that can also be represented in various conceptual models using well-established modelling languages such as Unified Modeling Language (UML), Business Process Model and Notation (BPMN), Object-Role Modeling (ORM), or Business Object Relationship Modelling (BORM). Therefore, it is possible to design evolvable and configurable transformations between conceptual and NS models.*

The transformation would benefit from the expressiveness, tooling, and community support around those well-established modelling languages on the one side, and evolvable information systems generation on the other side. For our contribution, we set the following research questions and related objectives coupled with the hypothesis: RO1 – Semantic Integration, RO2 – Transformation, RO3 – Evolvability, and RO4 – Consistency.

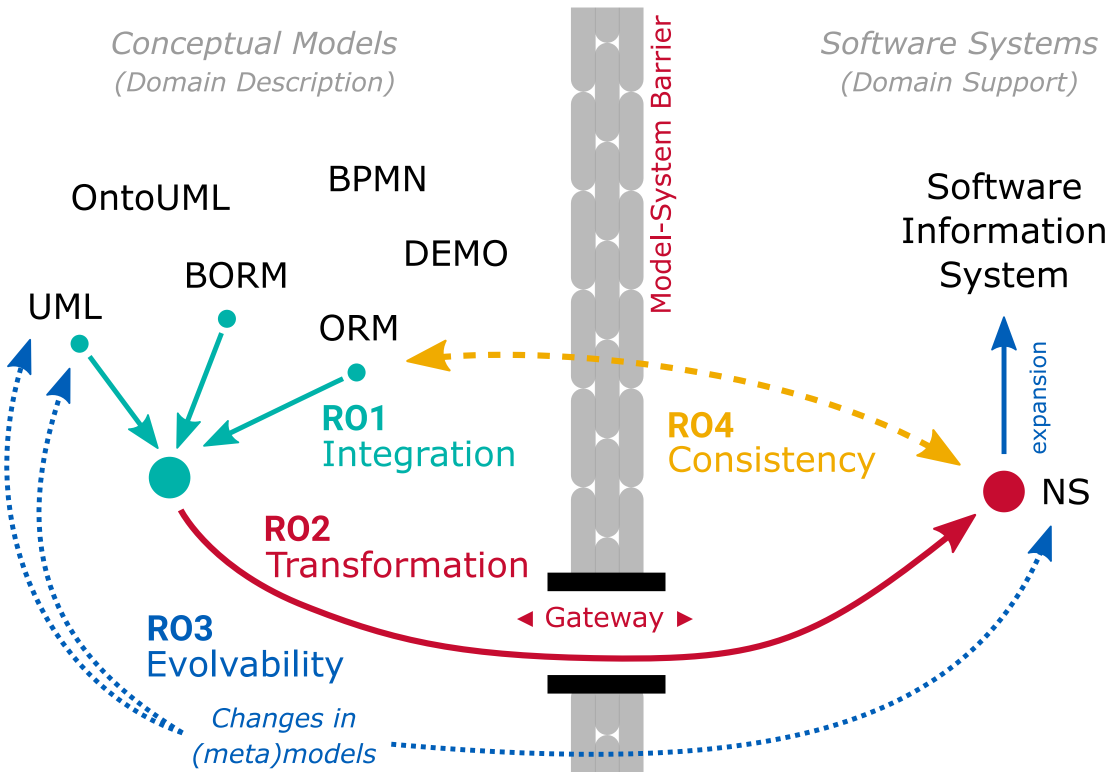

Figure 1.1: Overview of the gateway and our research objectives

RO1: Research Objective 1 – Semantic Integration *How can we interconnect various conceptual models that focus different aspects, e.g. processes, structure, or facts, in order to enable holistic view over specific problem domain?*

The modelling in Normalized Systems covers various aspects of a problem domain. However, many conceptual modelling languages and frameworks are focused on a particular aspect in greater detail. For example, OntoUML considers only the structure of a domain, ORM works only with facts, and BPMN is focused on processes. Therefore, we first need to allow the semantic integration of the knowledge captured in different conceptual models. With the holistic view – in the ideal case – on the domain can be used to produce a matching NS model and information system.

RO2: Research Objective 2 – Transformation *How can be conceptual models capturing various aspects transformed into NS models (and vice versa) with minimal information loss?*

Research is mainly focused on the transformation of conceptual models through NS models into evolvable information systems. A set of semantically integrated conceptual models is the input and the corresponding NS model is the direct output that allows the generation of a software system. The transformation must minimise information loss as much as possible using the current version of the NS metamodel. If information loss is prevented, then it should be possible to use the reverse direction of the transformation to create a conceptual model from an NS model. The problem related to this question is about allowing input models in different conceptual modelling languages and possibly additional integration information according to RO1.

RO3: Research Objective 3 – Evolvability *Is it possible to achieve evolvability, i.e. reduce the negative impact of future changes, of the transformation with respect to the metamodels and their independent changes?*

The transformation between various conceptual models and Normalized Systems will be dependent on the metamodels (e.g. definition of `Class` in UML or `DataElement` in the NS metamodel). The metamodels can change over time, and the transformation should ensure that the change in one of the metamodels will not affect the whole transformation framework. For example, when UML updates its specification, the changes should affect first only the UML-related part or module of the transformation, whereas all others are unchanged. Similarly, for the change in the NS metamodel. Last but not least, changes in some of the metamodels can open a possibility to enhance the transformation elsewhere. In the case of enhancement of the NS metamodel, new constructs may be available, and transformations from some conceptual modelling languages may take advantage of it.

RO4: Research Objective 4 – Consistency *Can we design the transformation in a way that allows maintaining or at least check consistency between conceptual and NS models?*

Not only can the metamodel change over time, but also can the underlying models. When there are models of the same domain at the conceptual level and the NS level, it is essential to consider their consistency. If something in the domain changes, the conceptual models are adjusted and the transformation should project it into the NS model. Vice versa, if the NS model needs to be adjusted, it should be possible to project the changes to a conceptual model or indicate where inconsistency occurs. Without such a mechanism, the relation between conceptual models and NS models would be broken after the first change.

1.3 Related Work & Previous Results

The topic of transformations between conceptual models and Normalized Systems is novel; therefore, there is no directly related work that presents such automatised transformation with Normalized Systems. The leading cause is that NS can still be considered as a novel approach in software engineering. The only work done (except our contributions) in terms of relating NS with conceptual modelling is related to Design & Engineering Methodology for Organizations (DEMO) [2, 3]. It focuses on linking DEMO and NS on a theoretical level using practical use cases to achieve enterprise agility.

However, if we broaden the scope, there are already previous results for transformations between conceptual modelling languages (modelling in NS can also be considered conceptual modelling). Naturally, more work has already been done on commonly used languages. For UML Class Diagram, transformations to Web Ontology Language (OWL) [4–8], Ecore [9], or Alloy [10–12] are already designed. Similarly, from BPMN to UML (and vice versa) [13,14], or from OntoUML to Alloy [15] and OWL [16]. Comparison between UML and ORM has been done in the past by Halpin [17–19] and can also serve as a mapping definition. The related work can help us design the transformation using the same source or destination modelling language as we need.

Model transformations are an essential aspect of model-driven engineering, enabling the automated manipulation of models to achieve desired outcomes. Several transformation techniques and tools have emerged to support this process. One popular approach is the use of the Query/View/Transformation (QVT) standard [20], which provides a language for specifying transformations between different models. Another widely used approach in model transformations is ATL Transformation Language (ATL) that operates on models represented in various formats such as Eclipse Modeling Framework (EMF) models, UML diagrams, and Extensible Markup Language (XML) documents [21]. Model Driven Architecture (MDA) approach is related also to MOFLON [22] and MOF Model to Text Transformation Language (MOFM2T) [23] approaches. Also, XML transformations, e.g., using Extensible Stylesheet Language Transformations (XSLT) play a crucial role in manipulating and converting XML documents which are often used for models representation or serialisation [24].

When we consider Normalized Systems as a technique for Model-Driven Development (MDD), previous work has also been done. The well-known MDA [25] defines steps for transforming conceptual model (computation-independent) to platform-independent and then to platform-specific models that can be used directly for code generation. There are many alternative and complementary approaches to MDA. The "model as a code" approach described in [26] proposes the use of a holistic model of a system that directly describes all aspects of a software application and is used instead of a high-level programming language, e.g. Java. Another important work by Rybola has been done on transformations from OntoUML to relational databases [27]. Various other methods, such as OO-Method [28] and scaffolding (e.g. in Ruby on Rails [29]) are further discussed in Chapter 2. Nevertheless, Normalized Systems are the only method so far that considers the evolvability and sustainability as a primary objective.

In terms of NS, we can consider as related work the NST [1] itself together with a number of articles related to modelling and metamodelling with NS as well as existing practical documentation and tooling [30]. NST provides us with essential principles and approach to evolvability that we want to adopt in our work as well. Moreover, NS is the target environment of the designed artefacts and as such the compliance with theoretical as well as technical foundations must be taken into considerations and the design itself.

Finally, our goal is to present a solution in the form of an ontology. Ontologies in computer science and software engineering are unquestionably related to Resource Description Framework (RDF) and OWL technologies. As these technologies are widespread, the need for ontology matching, mapping, and RDF transformations is already addressed in various related works [31–36].

1.4 Methodology

In the discipline of information technology and especially software engineering, the design science methodology provides a structured approach for conducting research and ensuring the quality of software solutions. Unlike other methodologies such as empirical research, action research, or explanatory science research commonly used in various scientific domains (e.g., life sciences, physics, or social sciences), the design science methodology is specifically tailored to the unique challenges of designing and building software solutions. Design Science Research (DSR), also known as constructive research, emphasizes the development and evaluation of human-made artefacts in the context of information technology. This methodology is particularly applicable to categories of artefacts such as algorithms, interfaces, languages, software applications, systems, and models, including ontologies. [37,38]

The methodology promotes a systematic and rigorous approach to problem-solving and solution development. It encourages the formulation of clear objectives, the identification of design principles, and the iterative refinement of artefacts based on feedback and evaluation. One of the key advantages of DSR is its emphasis on practical relevance. By focusing on the creation of usable artefacts, the methodology ensures that the research outcomes have direct applicability in real-world scenarios. This aspect is crucial in software engineering, where the ultimate goal is to produce effective and efficient solutions to address specific needs or problems. [37–39]

Furthermore, the design science methodology facilitates knowledge creation and accumulation within the field of software engineering. By providing a framework for documenting and sharing design theories, the methodology contributes to the collective understanding of effective software design practices. This shared knowledge can serve as a foundation for future research and development efforts, fostering continuous improvement in the field.

Peffers et al. [40] presents and demonstrates the use of DSR as a methodology for information systems research. Although we do not strive to design an information system but a transformation framework, their process model is well applicable, as shown in Figure 1.2. The problem and motivation have been set in this chapter. We also defined the *objectives of a solution*, i.e. our research objectives. The design and development of artefacts is the integral part, where results fulfilling the objectives are constructed to be then demonstrated and evaluated. Concerning the connection between research objectives, design, and evaluation, Braun et al. [41] propose Requirements-Driven DSR where various types of requirements are captured from problem analysis.

Moreover, the DSR process includes a crucial feedback loop that enables the refinement of objectives and artefacts based on evaluation. This iterative nature ensures that the transformation outcomes align with the desired goals. Additionally, as a part of effective communication and dissemination of results, this dissertation thesis incorporates publications that serve as references. While previous publications refer to specific partial topics or intermediate stages of our work, this thesis provides a comprehensive summary of the final results obtained through iterations of the feedback loop. It presents the state where both objectives and artefacts have been refined, highlighting significant improvements achieved over time in dedicated parts of the thesis.

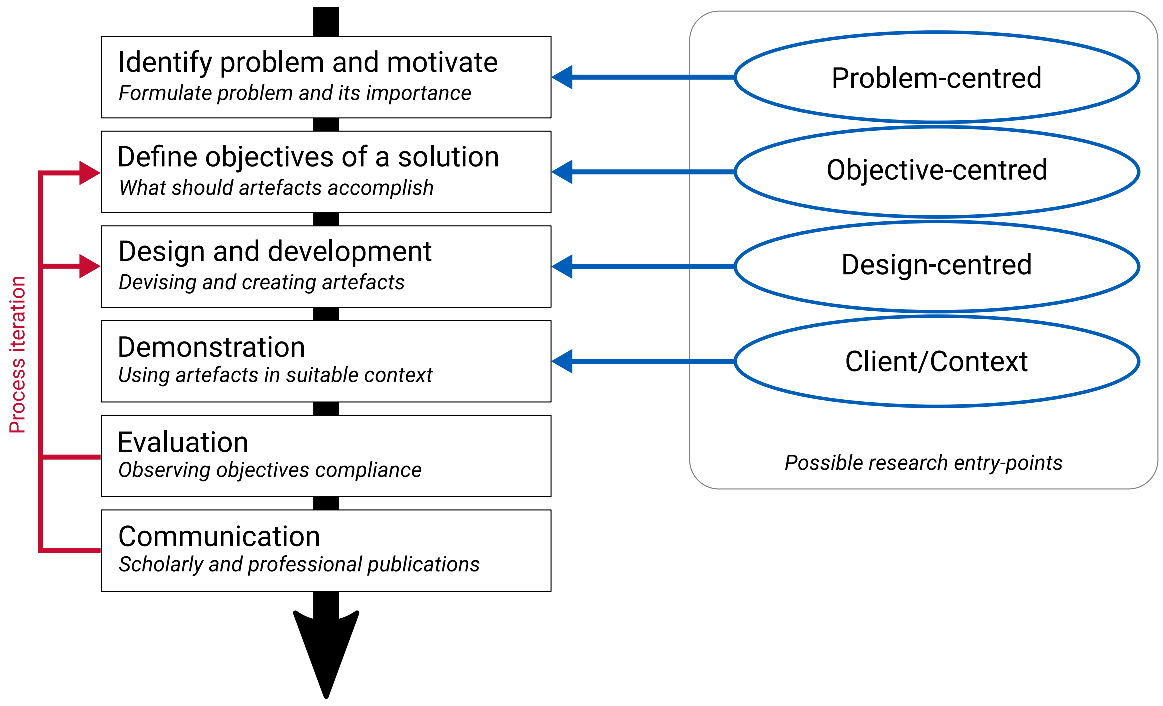
Identify problem and motivate
Formulate problem and its importance
Define objectives of a solution
What should artefacts accomplish
Design and development
Devising and creating artefacts
Demonstration
Using artefacts in suitable context
Evaluation
Observing objectives compliance
Communication
Scholarly and professional publications
Process iteration
Problem-centred
Objective-centred
Design-centred
Client/Context
Possible research entry-points

Figure 1.2: The DSR Process Model (according to Peffers et al. [40])

The operation in DSR with relation to external environment and previous work can be described using three related cycles of activities as depicted in Figure 1.3 [42]:

- The *relevance cycle* links research with the environment and its requirements as inputs. Hand-in-hand with the requirements, it defines the acceptance criteria for the evaluation of the research results. This cycle ensures the overall fulfilment of the requirements identified in the environment.
- The *rigour cycle* provides access to the existing knowledge base and thus ensures innovation. It serves to guarantee the quality of the research and prevent doing a routine design with well-known processes.
- The *design cycle* iterates between two core activities – building artefacts and their evaluation. It guides to iteratively build the ultimate research results from parts that are evaluated and potentially improved over several iterations.

The environment for our research is the domain of Normalized Systems development where there is the need for (re-)using conceptual models. The requirements are specified through the set in Section 1.2 in the form of research questions. Furthermore, the relevant aspects of the domain are described in Section 2.3. Evaluation of the research results, i.e. "field testing", is executed according to the needs and in collaboration with NSX bvba, a spin-off company developing Normalized Systems.

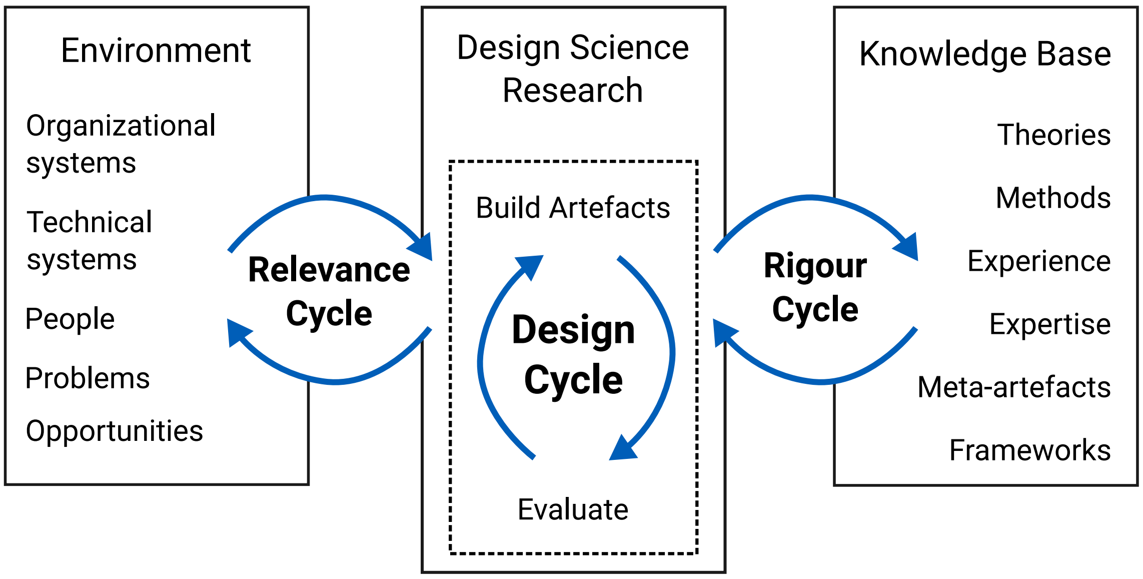

Figure 1.3: The Three Cycle View of DSR (according to Hevner [42])

The knowledge base that serves as the foundation for our research is described in Chapter 2. It encompasses a diverse range of disciplines, including conceptual modelling, model-driven development, formal specification, ontology engineering, Normalized Systems, and (bi-directional) transformations of knowledge representations. These components collectively contribute to the breadth and depth of our understanding. Furthermore, as our research progresses, there is potential for new additions to the knowledge base. This can occur when we introduce innovative design or gain unique insights. By continually expanding the knowledge base, we strive to contribute to the evolving landscape.

Finally, the artefacts we design, build, evaluate, and iteratively enhance throughout the design cycle are categorized into three primary components. First, we create an RDF/OWL representation for Normalized Systems, ensuring a comprehensive understanding of their structure and behaviour. Second, we develop a gateway ontology that serves as an encapsulation of NS, facilitating seamless integration with other systems and domains. Our approach aligns with the principles advocated by the NST, emphasizing the importance of a fine-grained modular structure that promotes evolvability. By incorporating this modular approach, we strive to enhance the efficacy and longevity of the artefacts we create, supporting the ongoing evolution of our research.

This section provides a brief overview of the methodology used in this thesis. The particular steps are explained in chapters corresponding to the developed artefacts. Chapter 3 describes the requirements, overall architecture, and decomposition to partial artefacts that are then presented in Chapter 4, Chapter 5, and Chapter 6. The demonstration as part of the “field-testing” is in Chapter 7. The summary of artefacts and other results is provided in Chapter 8.

1.5 Contributions of the Dissertation Thesis

The main contributions of this thesis can be summarised as follows:

1. *Design of model-to-model transformations based on RDF/OWL* – a design introducing gateway ontology and modular mappings and utilising RDF technologies as the medium that allows transferring knowledge between models based on different metamodels. Despite our focus on transformations between traditional conceptual modelling languages and NS, the design and related insights can also be re-used for other model-to-model transformations.

2. *Evolvable transformation between NS elements and RDF/OWL* – a method and related tool implementation for bi-directional transformation between models of NS elements and RDF/OWL. The evolvability lies in the design based on NS principles as well as using NS expanders for developing the tool. In our work, the transformation is used to have the NS metamodel in RDF/OWL and to be able to transform the underlying models in both directions.

3. *Ontologies for conceptual modelling using RDF* – a set of OWL ontologies with examples and other documentation for representing various conceptual models in RDF. For our transformation, we need to have the conceptual models encoded in RDF; therefore, we had to re-use or create new ontologies to support it. These ontologies can also be used for other use cases outside of our scope related to transformations with NS as keys to interoperability for conceptual modelling.

4. *SPARQL-based mappings for RDF transformations* – a method and implementation to encode mappings between two (or more) ontologies using RDF and then execute a transformation based on that by translating it into SPARQL `CONSTRUCT` queries. It allows metadata specification about mapping and creates links between them, thus enabling modularity and promoting evolvability.

5. *Transformations between NS and conceptual models* – a transformation framework based on our model-to-model transformation design that enhances possibilities of mapping different conceptual modelling languages that have metamodels captures as OWL (or RDFS) ontologies. Its modular design using layers and mappers for specific modelling languages follows the separation of concerns principle for evolvability and extensibility reasons. It utilises the contributions listed above to streamline (possibly bi-directional) transformations between NS and conceptual models.

This dissertation thesis covers both designs of artefacts but also its reference implementation that we needed for evaluation and demonstration following DSR. It also helps verify the relevance of the environment using field tests (as DSR suggests). The reference implementation of partial artefacts is ready to be used separately or together as a pipeline for transformations between conceptual models and NS. Furthermore, due to the focus on evolvability and extensibility, it is ready for future evolution.

1.6 Structure of the Dissertation Thesis

The thesis is organized into nine chapters as follows:

1. *Introduction* describes the motivation behind our efforts together with our goals. It briefly presents related work and the methodology used for conducting this research work. Finally, it summarises the main contributions of the thesis and outlines its structure.

2. *Background and State-of-the-Art* introduces the reader to the necessary theoretical background and surveys the current state-of-the-art relevant to our topic as the knowledge base for our DSR-based research. It summarises and explains the variety of possibilities in terms of conceptual modelling and the flexibility of ontologies. Finally, it provides an overview of related work in the areas of semantic integration, ontology transformations, model-driven development, and NS.

3. *Overview of Our Approach* explains the overall design of the Normalized Systems Gateway Ontology for Conceptual Models, its modules, and critical properties.

4. *Transformation between NS Elements and RDF/OWL* describes the transformation of NS Elements models into RDF representation and vice versa. It also explains additional steps for producing OWL from NS models, which is vital for the Gateway Ontology. The implementation of a prototype and expanded tool for transformation is also presented in this chapter.

5. *Using RDF/OWL to Represent and Integrate Conceptual Models* explains how we utilise RDF and OWL to represent conceptual models made in several well-known and widely-used modelling languages (namely: UML, OntoUML, BPMN, ORM, and BORM). The reasons for doing so are discussed. We use such representations both for semantic integration and for transformations with the Gateway Ontology.

6. *Transforming between NS Elements and Conceptual Models using Gateway Ontology* describes the Gateway Ontology architecture in greater detail, including its three layers. It further explains how transformation of a conceptual modelling language to (and from) NS can be defined and executed.

7. *Demonstration Use Cases* shows how Gateway Ontology can be used in several practical use cases and how it handles changes in both the Normalized Systems metamodel and the conceptual modelling language specification.

8. *Main Results* recapitulates the contributions of the dissertation thesis, their design, relations, and possible use. It also clarifies the use of DSR to develop them together with a related retrospection.

9. *Conclusions* summarises the results of our research, suggests possible topics for further research, and concludes the thesis.

Chapter **2**

Background and State-of-the-Art

"Today knowledge has power. It controls access to opportunity and advancement."

Peter Drucker

This chapter aims to cover the necessary knowledge required for understanding and following research of our topic. Because the domains of model-driven development and conceptual modelling are vast, for the sake of brevity, we focus on the core of theoretical background and the main approaches. Next, we mention previous results done in the area of our research. As there is no previous research directly on transformations between Normalized Systems (NS) and conceptual models, we investigate NS-related work that is related to conceptual modelling or business analysis in other ways. Moreover, transformations between conceptual models, in general, using various technologies, such as Extensible Stylesheet Language Transformations (XSLT) or Query/View/Transformation (QVT), are also part of our review. The last part of the chapter is then dedicated to Normalized Systems that are the key to our solution proposed later on in this dissertation thesis.

This chapter describes the knowledge base and existing solutions used for our research through the rigour cycle in terms of the Design Science Research (DSR) methodology. The chapter primarily provides the grounding to our research. Then, our additions to the knowledge bases regarding references to our contributions are also part of the chapter, for example, our review of UML-to-OWL transformations [A.8], mapping of UFO-B to process modelling languages [A.6], or transformation of textual requirements to NS [A.14].

2.1 Theoretical Background

In this section, we clarify the terms and methodologies of conceptual modelling, formal specifications, and ontologies that we then use in our research and corresponding parts of this dissertation thesis. We also briefly summarise the theory behind various approaches to implementation models and their transformations into the software.

2.1.1 Conceptual Modelling

The primary purpose of conceptual modelling is to formally describe some aspects of the physical and social world in order to improve understanding and communication between people [43]. As a conceptual model is a formal description of a problem domain, it is possible to use it for other purposes as well. One of such challenging purposes is a transformation into implementation in software. The main motivation is the efficiency of software development and thus swiftness in adapting to changes in the domain and its needs. As information systems as software applications are complex systems capturing many aspects of the problem domain and other technical details, the used conceptual models need to describe those aspects in a precise, complete, and unambiguous way.

Conceptual models are used to describe various aspects of the problem domain. Several modelling categories include structural, process, fact modelling, or even business rules definitions as parts of conceptual modelling. Various modelling languages and methodologies can be used for creating a conceptual model. Each of such a language has its advantages and disadvantages related to target use cases. In some cases, a simple modelling language for the description of domain structure is better to use than a complex one. In other cases, one needs to model different aspects of the domain, such as communication or process flows. When picking a suitable modelling language for our case, we should consider the following properties of the language:

- language expressiveness,
- tooling support,
- community and popularity,
- scalability and customizability.

Another essential property of a conceptual model is its quality. There is already work that clarifies aspects of quality in conceptual models [44] and others that discuss the issues of its evaluation and future steps [45]. Quality of models has more dimensions, including modularity, cognitive clarity, or correctness with respect to the modelled domain [46]. The evaluation of the quality can be then partially subjective, which causes evident problems.

In the subsequent subsections, we briefly introduce the well-known languages used for conceptual modelling that will be further observed and their transformation into implementation researched during our research. The languages are selected intentionally to capture various aspects of the problem domain and to be overlapping in some parts but focus on different properties or scope.

2.1.1.1 Entity-Relationship

The Entity-Relationship (ER) modelling language is one of the first developed for structural conceptual modelling in the software engineering domain; it was published in 1976 by Peter Pin-Shan Chen in *The Entity-Relationship Model: Toward a Unified View of Data* [47]. ER models are similar to simpler semantic nets, and they describe entities together with relations between them and attributes that characterise them. The primary use case of ER is to model a part of the real world that is necessary to capture some (business) case needs [48].

ER captures attributes that can be key, multivalued, composite, or derived. Key attributes serve as an identifier of an entity instance, for example, a primary key in the case of databases. Naturally, the entity can be free of key attributes; in such case, we call it a *weak entity* [47]. ER allows relations of generally any arity and are captured as "hub" connected to all participating entities; each connection is described by a maximal number of entity instances that can participate in single relation. Even relations can have attributes that can be used for identification. [47]

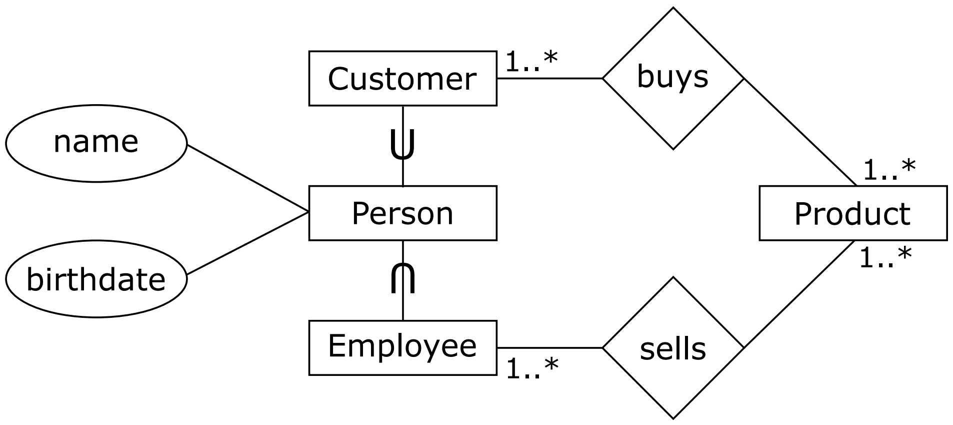

Figure 2.1: Example of Enhanced Entity-Relationship model

Since the publication of the ER, several alternative conventions of modelling with ER has been introduced to capture some aspect more straightforwardly, e.g. relations [49]. By adding new constructs, mainly notable *is-a* relation to incorporate generalisations and specialisations of entities, or enhancing relations, as shown in Figure 2.1, the Enhanced Entity–Relationship (EER), also known as E2R, has been developed later on. Although ER modelling is here since the 70s, it is still widespread, for example, for a design of relational database schema [48]. These days, even solutions for big data are being connected to ER modelling as discussed and described in [50].

2.1.1.2 UML

Unified Modeling Language (UML) is a widely-used complex modelling language developed and standardised by Object Management Group (OMG) to visualise, specification, construct, and document software applications (mainly object-oriented). As the name states, it was developed as a unification of many different modelling languages in the 1990s [48]. As a result, UML provides a standardised and well-known way of modelling through the software development cycle from an analysis (e.g. domain model, use cases, requirements, activity) via design (e.g. components, package, class, state machine) to development and deployment (e.g. database, deployment, timing) [48]. The latest version of UML specification is 2.5.1, but ISO recognises older version 2.4.1 as standard ISO/IEC 19505 [51].

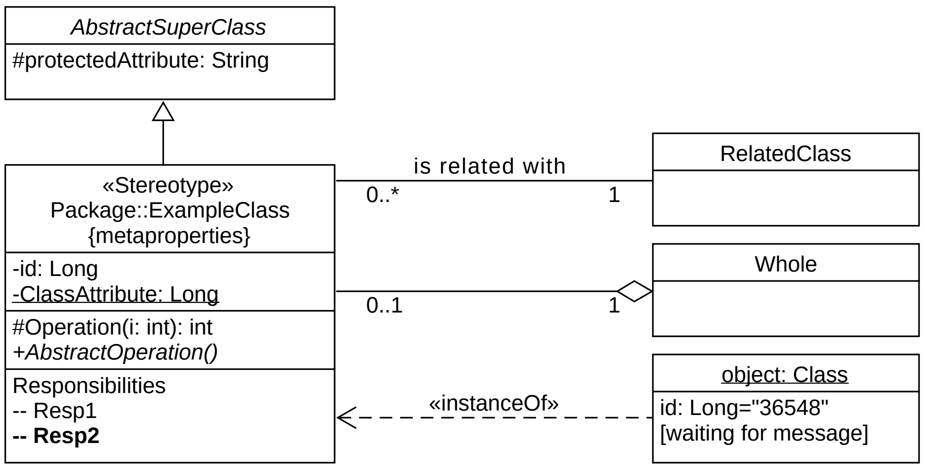

Figure 2.2: Capabilities of UML class diagrams

An important aspect is that UML is only a language, and it does not state how it should be used or how to do the analysis of a system; it is not a methodology. Nevertheless, it is suitable for the object-oriented approach to analyse, design, and implement software systems [48]. As being said, UML can be used during the whole software development cycle, but since it is more implementation-specific, for purposes of conceptual modelling are the domain mains ontologically too vague. Similarly, it is not suitable for other areas of analysis and design. Fortunately, UML profiles enable extending the language, and there are profiles like OntoUML for conceptual modelling or SysML for general systems engineering. [52]

UML [52] divides diagrams into categories: structural, behavioural, and interaction (part of behavioural). Thus, two essential aspects – structure and processes – are covered. Probably the most used and known way of modelling with UML is the class diagram that has a lot of features possible (see Figure 2.2). It is clearly visible that it has a lot in common with ER models, and it is also the reason for using it in conceptual modelling. In terms of processes, the most universal and suitable for domain-oriented models is the activity diagram that represents traditional flow charts [48]. All types of diagrams and their constructs are described in UML metamodel (can be extended with UML profiles). The metametamodel is called Meta-Object Facility (MOF) [53] that defines an M3-model, which conforms to itself [52].

The UML specification [52] is rather complex, and tools usually support only part of it. Some even try to simplify the metamodel and use their own XML Metadata Interchange (XMI) profiles for serialisation, which then causes inconsistencies and is a blocker for interoperability. Ecore is a metamodel defined in terms of itself as part of the Eclipse Modeling Framework (EMF) [54]. It is often understood as a UML subset focused on structural modelling to use model-driven development techniques or create a domain-specific language. Having a MOF-compliant modelling language useful for transformations between modelling languages, typically using of QVT [55].

2.1.1.3 OntoUML

OntoUML is a UML profile for ontology-driven conceptual modelling based on the structural aspects from the Unified Foundational Ontology (UFO) introduced by Giancarlo Guizzardi in his dissertation [56]. The language and support in the form of tools are mainly done by research groups NEMO and Menthor; however, standard UML modelling tools can be used as well. New features and changes are introduced usually using scientific articles done by members of these groups. For example, in [57] transformation to Alloy is described, and in [58] many internal OntoUML proposals has been polished and merged into “OntoUML 2.0”. Important mathematical foundations of OntoUML lie in modal logic that is used for rules and laws of UFO terms and their relationships [56].

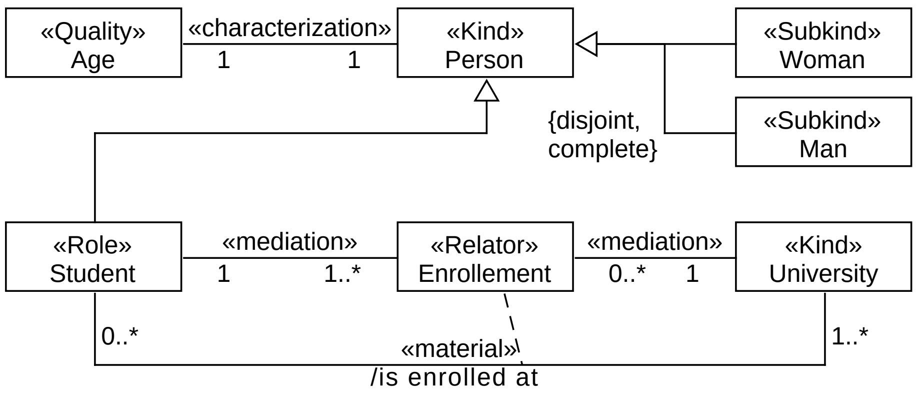

Figure 2.3: Example of a simple OntoUML model

As UML profile, OntoUML uses so-called stereotypes demonstrated in Figure 2.3. There are certain stereotypes for classes (such as kind, role, mixin, or relator), other for associations (for instance mediation, formal, or material) and special for part-whole relations (memberOf, componentOf, subCollectionOf, etc.) that are often denoted just by one letter inside or nearby the diamond symbol. Each stereotype – or term from UFO – carries specific constraints and ontological meaning based on cognitive and philosophical science. Class stereotypes are grouped according to the relationship to an identity principle to sortals and non-sortals and then by the ability to change: rigid, semi-rigid, and anti-rigid. Some types of entities are so-called identity providers, and each object (entity instance) must have exactly one identity of one identity principle. [56]

The empirical study [59] shows that ontology-driven conceptual modelling with OntoUML is higher-quality in many aspects when compared to traditional conceptual modelling, for example, with UML or ER. Although OntoUML is more discussed and used in the academic environment rather than in the enterprise, more and more fields identify the need of having solid-based and ontologically well-founded conceptual model for their efficient work [60, 61].

2.1.1.4 DEMO

Design & Engineering Methodology for Organizations (DEMO) is an enterprise modelling methodology developed since the 1980s by Jan Dietz and his research team. It consists of various theories named after Greek letters (ALPHA, TAO, DELTA, PSI, and others). It is highly affected by philosophy and the language/action perspective. DEMO uses as the core construct of a transaction and describes it as a connection of coordination and production between initiator and executor. The whole process for each transaction, called a complete transaction pattern, describes the typical flow between acts (activities) and facts (states) with possible loops and revokes. [62]

DEMO is focused on separating the modelled organisation and transactions into three levels: ontological, infological, and datalogical; where the ontological is describing the true essence of the organisation and the other two are implementation details. The organisation can be described in DEMO by multiple models that are again split into three levels (sometimes called the hamburger model). The highest level is for construction models that are Transaction Product Table (TPT), Organisation Construction Diagram (OCD), and Bank Contents Table (BCD). Then in the middle are process model, such as Process Structure Diagram (PSD), and fact models similar to other process models and structural models. Furthermore, in the lowest level, action models are used to describe conditions and the flow using structured scenarios with decisions and claims. [62]

A significant advantage of the DEMO is that it provides mathematically-based axioms and interconnected models to create an organisation's complete view. It describes the structure of an organisation and its products, the process that flows inside and between transactions, and facts and business action rules. Enterprise Institute also provides professional certification in this methodology, and tooling support is good because it is rather academic-based work. Research on relating DEMO to other modelling languages has been already done, for example, its combination with Normalized Systems [63] or its conversion to BPMN [64].

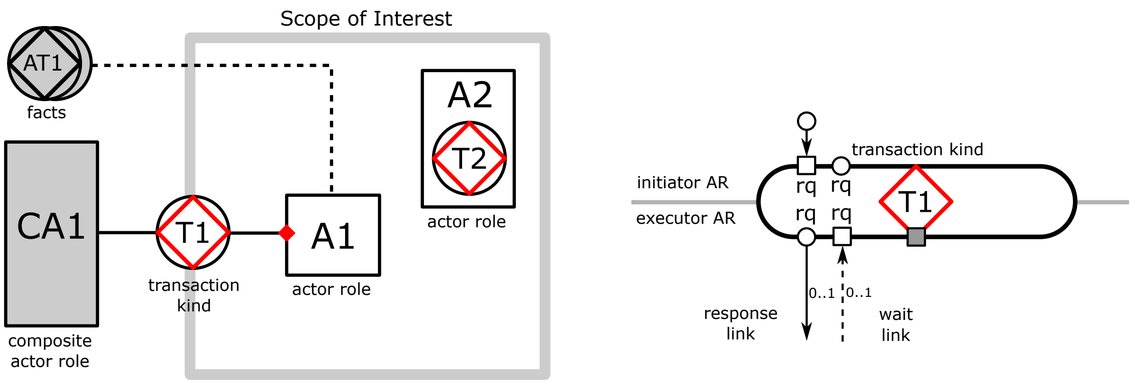

Figure 2.4: Elements of OCD and PSD diagrams

2.1.1.5 ORM

Object-Role Modeling (ORM) [65] is similarly to DEMO designed for conceptual modelling with domain facts. In contrast to ER and UML, ORM treats all elementary facts as relationships and then groups facts into structures. It results in improved semantic stability and even simplifies model verbalization into natural language. Still, a considerable visual concordance with ER/EER models is observable. Relations in ORM are modelled as predicates of any arity which is shown in Figure 2.5. Facts are captured by entities and their participations in various relations. We refer to ORM 2 released in 2005 which added several new features and simplified the diagrams [66].

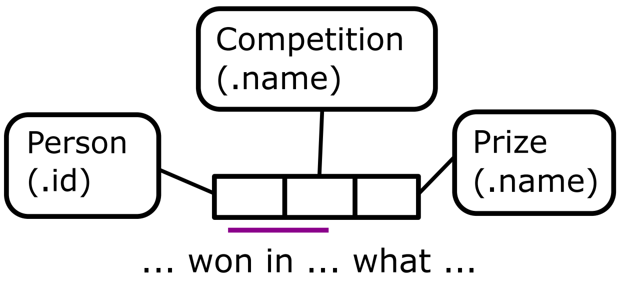

Figure 2.5: ORM relation of Person winning a prize in competition

The most significant difference and advantage of ORM, when compared to UML or EER, is the ability to specify a wide variety of constraints defined by the language. Nevertheless, the language is very easy to read and use. It allows to simply express uniqueness, mandatory relationship, multiplicity, combinations of relations with operators (such as *exclusive or*), relational properties, enumerations, and many others [65]. Automated formal reasoning on ORM conceptual schemes is possible thanks to its well-formalised semantics based on first order predicate logic and set theory [67].

2.1.1.6 BPMN

Business Process Model and Notation (BPMN) is a set of principles and rules, including graphical language, to express business processes. Due to the OMG standardisation, existing tooling, and ability to orchestrate processes via Business Process Execution Language (BPEL), it is widely used [68]. It can be seen as a more complex flow chart diagram. Aside from swimlanes and activities with transitions, it also has several types that can form a subprocess (modularity of process). There are also multiple types of events and gateways. In BPMN, it is possible to capture conditional events, timed events, errors, and signals. For gateways, instead of traditional decision branching and parallel fork, there are parallel, inclusive, exclusive, complex gateways and then also event-based gateways (incorporating an event in the process). [69]

Associations and message flow further increase the expressiveness of this notation (as a complement to sequence flow) and three types of artefacts: data objects, groups, and annotations. Still, one of the issues with BPMN is the mix of domain-specific and orchestration-related information in a model without a clear separation that is provided, for example, in DEMO. BPMN 2.0 uses solid Business Process Definition Metamodel (BPDM) to create a single consistent language, and there are Extensible Markup Language (XML) schemas available to transform BPMN models to support decision processes and applications within organisations [69]. From the ontological point of view, there is vagueness similarly to UML and in [70] are proposed improvements for so-called “Onto-BPMN” as a counterpart to OntoUML.

2.1.1.7 BORM

Business Object Relationship Modelling (BORM) is a method designed for conceptual modelling of process-intensive organisations. It provides an effortless but conceptually sufficient way how to describe processes in a system. As all process models in graphical notation, even BORM has some similarities to flowcharts in its Objects Relations (OR) diagrams. Instead of swimlanes, participant blocks are used with a distinction between a person, organisation, and technology, e.g. software. Each participant has their own process where are states and between two states as a transition is always activity. A state can contain another process. There can be communication with data flows between participants, more specifically, between their activities. Branching is done simply from a state. [71]

Besides, Business Architecture (BA) diagrams are covered in BORM as well. They serve to capture the functional blocks in a system and its scenarios to which are related OR diagrams linked. Scenarios can be connected with *uses* and *extends* relations [71]. Model Driven Architecture (MDA) is well supported in BORM, and it uses strong formal background – communicating finite state machines. Other interesting aspects of BORM are its modularity possibilities discussed in [72] and straightforward interconnection with object-oriented software. We investigated the relations between process modelling with UML Activity Diagram, BPMN, BORM, and UFO-B [A.6].

2.1.2 Formal Specifications

Conceptual modelling languages are designed to promote communication and understanding between people, but logical or mathematical reasoning becomes desirable when it comes to validating a model. Conceptual models often tightly coupled with a graphical representation (but some also use textual, e.g. actions in DEMO). Although some conceptual modelling languages provide reasoning by leveraging their mathematical foundations, formal specifications are specifically designed for verification, reasoning, and easy transformations into the software [73].

Some conceptual modelling languages allow transformation to so-called formal specifications, e.g. mentioned OntoUML to Alloy [57]. There are many languages for a description of a system that can be then used for simulations and validation. Such methods are often based on set theory and algebra. Historically, many development methods of mission-critical systems were tied with formal specifications and mathematical proofs of correctness [73]. This section only refers to the recently used and the most interesting for our future research and focuses on generic approaches.

2.1.2.1 OCL

The Object Constraint Language (OCL) is a declarative language for capturing constraints and rules to connected UML model(s), but it can be used with any MOF-compliant model [74]. Thus, for example, OntoUML can be further described by OCL as well. It is very close to simpler object-oriented programming languages. A description is split into packages, and then rules (sets of declarations) are tied to a certain context that can be a class, a method, or an attribute. Many standard types, collections, and operations (including operators) are available [75]; still, more can be defined manually or imported from additional prepared packages. The rules are of multiple types:

- `inv` = invariant, a condition that must always hold,
- `pre` = pre-condition, a condition that should be true before calling a method,
- `post` = post-condition, a condition that should be true after calling a method,
- `init` = assertion for initial values of attributes,
- `body` = assertion for an expected result of the associated operation at a certain point in time,
- `derive` = declaration of a expected property which is derived (for example, age from birthdate).

OCL has excellent tooling support thanks to its relation to UML and the OMG specification. It provides a robust means to validate instances of a model. Using OCL can significantly help with resulting software quality as shown in [76].

2.1.2.2 Alloy

Alloy [77] is (similarly to OCL) a declarative language for formal specification of structures based on first-order logic. It is developed together with the related tool Alloy Analyzer on the Massachusetts Institute of Technology. The language is profoundly affected by Z notation, a mathematical encoding of a system, but the syntax is closer to OCL. A model in Alloy is a description of structures grouped into modules. Although the syntax is simple and there is a minimal number of constructs, it allows expressing everything needed. Thanks to the declarative nature of the language, same as in OCL, the order of statements does not matter.

An Alloy model consists of:

- signatures of defined sets,
- facts representing permanent constraints,
- predicates that can be understood as parametric facts,
- functions that return a value for some parameters,
- boolean statements about the model.

The Alloy Analyzer can generate for such specification instances and try to find a counterexample that violates a specific statement, i.e. a problem in the model definition [77]. In this case, it is again similar to UML with OCL, and [78] shows how translation between UML with OCL and Alloy is possible.

2.1.2.3 OASIS

Open and Active Specification of Information Systems (OASIS) is a formal specification language using an object-oriented paradigm and first-order logic [79]. It has been developed in the 1990s and further updated with new versions [80, 81]. The importance lies in use within OO-Method [28] that we describe briefly in Section 2.2. It provides the ability to formally describe all functionality, including UI using four views: static, dynamic, functional, and presentation [79].

Thanks to formalisms behind OASIS, specification of classes, behaviour, object constraints, extra functionality and presentation of object-oriented systems, it allows extensive verifications, reasoning, and transformations [28]. Unfortunately, there is no recent work on OASIS and resources are not easily accessible, which is an evident obstacle in language adoption and broader use.

2.1.2.4 Correctness by Construction

Correctness by Construction (CbyC) is a way of using formal specification rather than a particular language for building the specifications. The core idea lies in using formal specification during the construction of software using annotations or as transformed fragments in the code so that correctness according to the formally proven specification is inevitable. In the book “The Correctness-by-Construction Approach to Programming” [82], the Guarded Command Language (GCL) is used to specify invariants, preconditions, and postconditions.

A similar approach uses the TestEra tool for Java [83]. It also allows to specify invariants, preconditions, and postconditions in annotations of methods and classes but with the Alloy specification language. Such a direct connection to implementation leads to several advantages when compared to separated formal specifications. It can validate the implementation, but it also is easier to read when only fragments are connected to a place where they belong. On the other hand, if someone wants a complete formal specification, parsing of source codes must be done and for another way, know what declaration belongs where is essential. Finally, it is more efficient to track changes and search for violation causes in the source code.

2.1.2.5 Algebraic Specifications

An *algebraic specification* is a technique used in software engineering to specify the behaviour and structure of a system using formalism from the areas of mathematics called algebra. An algebra consists of axioms and groups of object sets and related operations in this set with their functionality for formal specifications. Vital aspects of using algebraic specifications in addition to automated reasoning and proving correctness are an abstraction from implementation details and distinctions between data element constructor function and additional functions, i.e. operations. [84]

There are many languages for writing algebraic specifications with various advantages and disadvantages. Like Maude or Z notation, some are not very used these days regarding research and usage in practice. Nevertheless, there are others like Common Algebraic Specification Language (CASL), Vienna Development Method (VDM), B-method, Onion, or PRISM that are used [85]. Lastly, Onion and PRISM are examples of languages (and tools) for process algebra that focus on flow specifications using mathematical constructs such as communicating finite automata, Markov chains, and others. Algebraic specifications are relevant and valuable; however, not very widely used in practice recently [86]. The specifications are replaced by various tests frameworks that specify the rules and in both human-readable and programmer-convenient ways. Still, it can be used for the specification of missing critical software or its most essential parts.

2.1.3 Ontologies

Ontology, which means (from Greek) *study of being* or *science of being*, is a philosophical discipline that studies concepts of existence, being, becoming, and reality. In computer science, software engineering, and data science, the ontologies and conceptual models are very close and have common goals. Both are used to capture some knowledge, to describe a part of reality. Ontologies (and ontology-driven conceptual models) usually focus more on the philosophical essence of things and relations, whereas conceptual models might be more implementation-oriented [59]. The divergence is related to the difference of open-world assumptions related to ontologies, i.e. what is not captured in the ontology may still exist. On the other hand, conceptual models and traditional formal specifications use closed-world assumptions.

Ontologies can be used for storing any kind of knowledge, including integrations of knowledge from various ontologies [87]. There are also many upper ontologies defined with higher-level terms that can be reused or also integrated to connect some external knowledge [88]. Another group of ontologies are used to describe metadata about other projects and ontologies, such as Dublin Core ontology [89]. Finally, there are also defined ontologies for conceptual and business modelling, for example, REA (Resources, Events, Agents) ontology [90]. Formal specifications are used to represent knowledge in a well-defined syntax and semantics. Another way how to address this with a common goal is called description logic (DL). Problem domains can be described using DL languages, which can use various description logics: general, spatial, temporal, spatiotemporal, and fuzzy [91].

An ontology in computer science is a representation, formal naming and specification of the categories, together with their properties and relations between the concepts. Ontologies are being intensively used in the fields of artificial intelligence (AI), conceptual modelling, semantic web, and to promote data interoperability in general. Several languages and methods can be used in conceptual modelling, e.g. DOGMA, F-Logic, OCML, or KL-ONE. However, the most widely used due to its good tooling support and versatility is the Resource Description Framework (RDF) and the Web Ontology Language (OWL). [92]

2.1.3.1 RDF and OWL

The core idea of RDF lies in the simplicity and versatility of triples: subject, predicate, object. Each of these three components can be an entity identified by Uniform Resource Identifier (URI) or even its internationalized generalisation Internationalized Resource Identifier (IRI). Objects can also be literals such as strings, numbers, dates, and others. A set of such triples can be formed to describe practically anything, i.e. capture any knowledge. That is similar to natural language but with improved machine readability and understandability. To actually give meaning to the triples and bring structure into otherwise unstructured triples, an ontology or a schema can be created or (re)used. The giving meaning to concepts (referred by URIs) is done again by adding triples. For example, if we want to state that a person has a name, the predicate *hasName* should be defined with relation to a class *Person* and expectation to have a string literal as an object. [88, 93]

Resource Description Framework Schema (RDFS) is a set of classes with specific properties that can be used to describe ontologies. RDFS defines, for instance, what is a class, datatype, literal, label, sub-class, or domain and range of a property. It is sufficient for basic vocabularies and simple ontologies where particular constraints and metamodelling are not necessary. For example, the well-known Friend of a Friend (FOAF) or Description of a Project (DOAP) ontologies are defined using RDFS. [88, 92]

When the expressiveness of RDFS is not sufficient, the Web Ontology Language (OWL) comes to the rescue. It enhances some of the concepts from RDFS and adds new such as constraints, special relations between classes, or means for metamodelling, e.g. punning. OWL is actually a family of knowledge representation languages for creating ontologies. There are three sublanguages:

- OWL Lite – a bare minimum of additional constructs to support easier adoption within tools,
- OWL DL – maximum expressiveness possible while keeping computational completeness and decidability,
- OWL Full – uses different semantics than the two above, allows metamodelling but is undecidable [88].

In OWL2 [94], the separations into sublanguages are done slightly differently – to overlapping profiles. The essential property is that OWL (and OWL2) are RDF-based, i.e. use triples to define ontologies and compatible with RDFS. It is, therefore, possible to start creating an ontology with RDFS and, when needed, start using OWL without the need to rework everything from scratch. Finally, it is even possible to keep track of changes with triples and refer to older versions efficiently [92].

2.1.3.2 RDF Formats and Tooling

RDF itself does not specify a format for representation; the formats are independent of the core ideas in the framework. There are multiple formats focused on different use-cases. For example, for interoperability and the use of standard parsing tools, RDF/XML is suitable. For human readability and compactness, there are formats such as Turtle, TRiG, or N-Triples. Finally, to support linked data and semantic web, RDFa (RDF in attributes) and JSON for Linking Data (JSON-LD) formats can be used. Usually, each of the formats has its own specification and is evolved independently of others. [92, 93]

Hand in hand with the wide use in the world of semantic web and linked data, the tooling support of RDF and OWL, is rather good. Most modern text editors and Integrated Development Environments (IDEs) support composing RDFs by syntax highlighting, prefix resolution, or autocompletion. Then, there are specialised tools for composing ontologies such as Protégé [95], virtualising them (e.g. WebVOWL [96]), or documenting them (e.g. WIDOCO [97]). Libraries for development with RDF in main programming languages are also available. It is always essential to pick the tool or library that is maintained and stable, which is not always the case as many of the RDF-related tools are developed in academics and can be identified as *professor-ware*, proof-of-concept, or just a prototype. [93]

The RDF triples can be stored in text files, but graph databases (e.g. Neo4J) or triple stores (e.g. GraphDB or AllegroGraph) can be used. Triple stores often provide additional features when compared to graph databases such as SPARQL Protocol and RDF Query Language (SPARQL) endpoints or inference mechanisms. On the other hand, graph databases are usually faster and optimised for big data. SPARQL is yet another technology in the Semantic Web stack that serves to query, change, or transform RDF data. It can be seen as an RDF counterpart to Structured Query Language (SQL) from the world of relational databases. [98]

2.1.3.3 Ontology Mapping

With the development of both upper and domain ontologies, overlaps are naturally emerging. For example, many ontologies describe the concept of a *Person*. If the relations between the concepts of multiple ontologies are found and described, it allows the semantic integration of the underlying data. For example, we can relate multiple statements about the same person created using different ontologies. *Ontology alignment* (sometimes also called *ontology matching*) is the process of determining correspondences between concepts in different ontologies. A set of such correspondences is also called an *alignment*. A broader term *ontology mapping* encompasses the process to define similarities among the concepts belonging to separate source ontologies (not limited to correspondences). [99,100]

There are various ways how to find and define the relations between ontologies. The most basic way is provided directly in OWL through predicates *sameAs*, *equivalentClass*, or *equivalentProperty* [94]. Nevertheless, there are also specialised mapping languages and vocabularies. For automation of the matching, AI and natural language processing (NLP) techniques are being used, which is vital for large ontologies [100]. Mapping the ontologies further promotes the key ideas of linked data as it helps to link different datasets directly. We proposed a high-level ontology that generalizes the terms used by various conceptual modelling languages to support their semantic integration [A.5].

2.1.4 Model-Driven Engineering

Model-Driven Engineering (MDE) is a software development methodology that leverages the use of conceptual domain models to increase productivity. There are various branched of MDE targetting difference use-cases where the productivity can be increased using the conceptual models:

- maximization of compatibility between systems by adopting standard or reference models,
- optimization of a system through its model representation (e.g. process re-engineering),
- simplifying the process of design by using standardized patterns,
- promoting communication between people by using standardized terminology and best practices,

- simplifying the process of development (and maintenance) of software systems [101, 102].

The lastly listed is often being called Model-Driven Development (MDD). MDD uses conceptual models to produce software systems or its parts. Such a use can be realised by transforming the model(s) into source code. There are, again, many techniques and methods for MDD. In the following subsection, we mention key aspects related to MDD and then, in Section 2.2 we describe the most widely used approaches.

2.1.4.1 Multi-View Models

In this dissertation thesis, we use the term *holistic model* or *holistic view* of a system to describe a model that describes a system completely in all aspects needed. With the use of MDD, it is the ideal case to have complete knowledge of a domain in the model, so the generated software system is also complete, i.e. without any need of custom programming [103, 104]. Here we mention and refer to already existing effective approaches that use multiple views on a system and integrate them to create a complete description.

We already mentioned UML and DEMO in Section 2.1.1, and both of them use different models to capture various aspects of a system. In both cases, the models are integrated. For UML [52], a class from a class diagram can be used in a sequence diagram or for creating objects in an object diagram, and it can have its own state machine diagram. As is already said, other languages than UML can be more suitable for conceptual modelling [59]. In DEMO [62], again, all models are integrated via a transaction that is the crucial concept in this methodology. On the other hand, views only in transactions, facts, rules, and their relations, can be too simplistic in specific cases [105].

Multiple views are also used on a higher level of abstraction when describing the overall architecture of a software or system in general [103]. One of the approaches is called *4+1 architectural view model* [106]. As shown in Figure 2.6, there are four views: logical, development, process, and physical, and then the one stands for scenarios. Although this architecture design is interesting, it is intended mainly for software and not conceptual-level. Also, the integrations of views and scenarios are not completely clear, and from Figure 2.6 is visible that connection is not bidirectional, which can cause consistency issues.

2.1.4.2 Implementation Models and Reverse Engineering

In model-driven development, the software is developed using models from the conceptual level to programming code. The conceptual model is turned into some intermediary implementation models that contain details related to computational model, software architecture, and platform or programming language capabilities. As an unwanted effect, degradation of the conceptual model in terms of losing important details due to lower expressiveness and so-called construct overloading is often tied to these transformations. Due to that, backward consistency is hard to maintain. For such implementation models, the widely used language is UML or customised EER-like models that restrain modelling

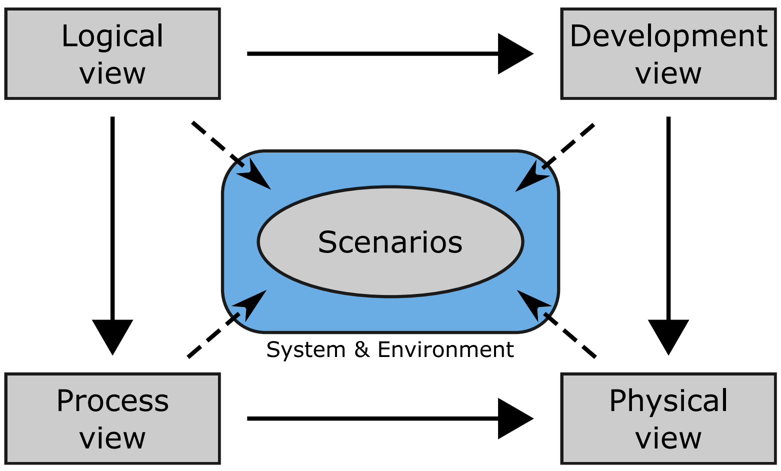

Figure 2.6: The 4+1 architectural view model (according to Kruchten [106])

enough and are close to the software way of working. The main approaches of this topic are further explained in Section 2.2. [107–109]

For the other side, i.e. a backward transformation from implementation into a model, the method is called *reverse engineering*. Usually, some details from programming languages might be lost if the model is not expressive enough. There are again various solutions for different modelling and programming languages, such as CPP2UML for C++ into UML in the XMI format [110] or from a database into ER diagrams [111]. Both code generation and reverse engineering are also well-supported in commercial solutions, for example, Enterprise Architect [112].

2.1.5 Model-to-Model Transformations

MDD is utilizing various kinds of transformations at multiple stages. Some of the methods use several levels of models for gradual change of input domain knowledge into source code. It omits domain details and adds technical information on each level. Code generation from domain knowledge itself can be seen as a kind of transformation (usually encoded using mapping and code templates). Finally, transformations can also help exchange domain knowledge between different modelling languages. For example, one might need to transform a BPMN model into UML Activity Diagram to relate it to the corresponding Class Diagram easily. [101, 107]

The approaches for such transformations vary, but there are two main aspects that should be taken into consideration. First, there is the directionality of the transformation. Bi-directional transformations (sometimes denoted as *bx*) can help maintain consistency by allowing transformations back and forth. Still, the transformations are usually not lossless. Second, some of the transformations are designed purely on the level of serialised information (syntax). Model serialisations are widely done through XML, and for that

XSLT transformations are usable. On the other hand, context-aware or semantic-level transformation can provide better results by utilising metamodel knowledge. An example is the QVT designed for transformation between MOF-compliant metamodels.

2.1.5.1 Query, View, Transformation (QVT)

Query/View/Transformation (QVT) is defined by OMG as a standard set of languages for model transformation [20]. As such, it is tied to other OMG specifications, such as MOF or OCL. The standard defines three languages for model transformation, and all of them work with models conforming to MOF. The OCL is integrated and extended to support the transformation through imperative features (on top of existing declarative). The three languages of QVT as shown in Figure 2.7 are:

- QVTr (relations language) enables declarative specification of relationships between MOF models using complex object pattern matching, trace classes and instances. It has both a textual and a graphical concrete syntax. Consistency can be checked during transformation execution.
- QVTc (core language) is a simple and small declarative language. It serves as a target for relations-to-core transformation that does not preserve all semantics as QVTc is not as expressive as QVTr. The specification [20] uses analogy to JVM where QVTr is like Java and QVTc is Java Byte Code.
- QVTo (operational mapping language) is an imperative language for writing uni-directional transformations. It provides OCL extensions to support the procedural programming style. The mappings can be used to implement relations from specification done using QVTr as it may be difficult or impossible to do so only in declarative style.

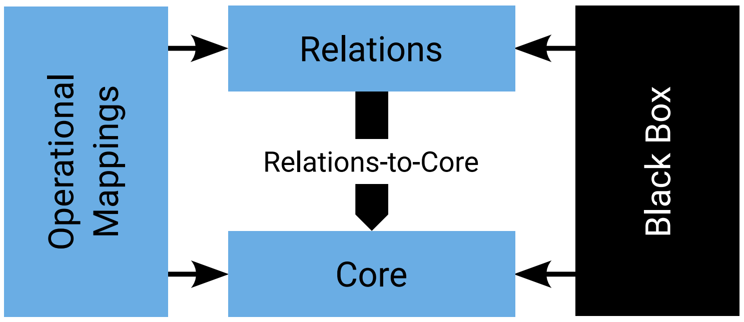

Figure 2.7: Relationships between QVT metamodels (according to [20])

As also captured in Figure 2.7, there are so-called Black-Box implementations that encompass various plugins and extensions. Again, the analogy with Java Virtual Machine (JVM) describes this part as a counterpart to an external component called the Java Native Interface. Although, with this approach any additional functionality or transformation details can be captured, it is also marked as dangerous as there is no control what the black-box can perform.

There are various implementations of QVT languages that follow the standard to a certain level (supporting a subset of it). Moreover, several transformation languages, such as ATL Transformation Language (ATL), are highly inspired by QVT and even sometimes called QVT-Like. The limitation of requiring MOF-compliant models is solved by additional and custom tools that allow, for example, MOF-to-Text transformations (MOFM2T).

2.1.5.2 ATLAS Transformation Language

ATL [21] is a model transformation language and toolkit, part of the Eclipse MMT (Model-to-Model Transformation) project that also supports QVT. It allows for the production of a set of target models from a set of source models. The language to create transformation specifications (sometimes called the ATL programme) is basically a set of helpers and rules to execute. The rules have input and output patterns, and helpers serve to prepare input data from a source model (for instance, using if-conditions or aggregations). The requirement is the common metametamodel (M3) which is in the Eclipse environment usually Ecore.

The advantage of ATL is the tooling support, the documentation with examples, and the connection with EMF. Also, unlike others, this language is still being maintained and updated together with the tooling. However, complex ATL transformations are more similar to regular programming in a new language (similar to OCL) rather than just mapping specification.

2.1.5.3 XML Transformations

XML transformation languages such as XSLT, XQuery, XProc, or XQuery are programming-like languages designed specifically to transform an input XML document into an output XML (or other) document [24]. Therefore, we can distinguish XML-to-XML and XML-to-Data transformation where the second one can target in the most generic case to a byte stream. XML-to-XML transformations can be seen as Model-to-Model transformations because XML structure is specified by a schema (i.e. definition of tags, their attributes, and parent-child relations). The schemas can be models or metamodels; for example, in the case of XMI that captures the UML model, the schema for UML serves as a metamodel, and the actual XMI file contains an UML model.

2.1.5.4 MOF Model to Text Transformation Language

MOF Model to Text Transformation Language (MOFM2T) [23] is another OMG-specified transformation language. More specifically, it can be used to define transformations that transform a MOF-compliant model into text. The target text is usually meant as documentation or source code, but practically it can also be other model representation than the supported XMI. For example, one can use MOF Model to Text Transformation Language

(MOFM2T) to transform a MOF-compliant model into RDF in Turtle format. Acceleo[1] is an implementation of the standard within Eclipse toolset.

2.1.5.5 MOFLON

MOFLON [22] is a "standard-compliant metamodeling framework with graph transformations" that implements MDA, and uses MOF together with OCL. The graph transformations are handled using the Fujaba tool. As a result, a Java representation is generated from XSLT transformations, OCL constraints, and graph transformations. The resulting application (interfaces and implementation) can be used for model analysis, integration, and transformation execution. This extra step of generating an application/classes for transformation has interesting aspects in terms of potential extensions and re-use.

2.1.5.6 Bi-Directional Object-Oriented Transformation Language

The Bi-Directional Object-Oriented Transformation language (BOTL) [113] provides interesting insights and concepts despite it being not widely adopted and used. As the name suggests, it focuses on object-oriented models (with a focus on UML) and their transformations. It is defined on the basis of set theory and provides formal definitions and proofs for both transformation rules and verification mechanisms of BOTL specifications. Then, the transformations can be captured using a comprehensible graphical notation. Finally, the specifications are designed to support bijection (and thus bi-directionality of transformations).

2.2 Previous Results and Related Work

This section describes the previous research and practical work related to our topic and which we use as references for our approach. Unlike the broader and generic previous section, we briefly introduce specific solutions, proofs of concepts, propositions, and other exciting research from which we can benefit while designing the Normalized Systems Gateway Ontology for Conceptual Models. Each of the presented works is commented in the context of this dissertation thesis topic.

2.2.1 Model Driven Architecture

Model Driven Architecture (MDA) is an application of MDD by OMG in the ecosystem of model-based engineering [107]. It applies various standards and other work of OMG, for instance, UML, MOF, or QVT. MDA was firstly published in 2001 and then revisited with newer versions of related standards and other updates in the domain [26]. The core idea is to allow transformation between models from more domain-oriented into implementation-oriented. There are three levels of models (listed in order from the conceptual level to implementation specification):

[1]https://www.eclipse.org/acceleo/

1. Computation-independent model (CIM),
2. Platform-independent model (PIM),
3. Platform-specific model (PSM) [108].

The three levels of models allow adjusting a domain model for the technical realisation gradually. However, the model may also gradually degrade in terms of losing domain-relevant information. It becomes challenging to maintain consistency between the related models, resulting in immediate issues when re-generation is needed (e.g. after a change in the domain model). The problem is amplified by additional custom source code in the generated application. As a result, the approach may be useful for initial software development but then turns into traditional maintenance and sustainability issues. [114]

Model Driven Architecture (MDA) is well supported by commercial tools such as Enterprise Architect, which supports all types of UML diagrams and several UML profiles. Moreover, it provides a mechanism of extensions so more UML profiles or other modelling languages and code templates can be added [112]. One of the other approaches to MDA is a method and also language called Executable Translatable UML (xtUML) that contains a subset of UML extended by execution semantics and rules. Models done in xtUML can be executed, tested and complied with lower-level programming languages. This language is supported by commercial and even non-commercial tools [115].

A similar approach is observable in Foundational UML (fUML) and Action Language for Foundational UML (ALF) that are defined and standardised directly by the OMG. The language fUML is again a subset of UML and is used to model the structure and behaviour of a system. ALF serves to behaviour specification via programming-like syntax. An advantage aside from OMG support is the close relation of fUML, ALF, and UML [116]. Another example of using MDA levels of models is the transformation of OntoUML models (CIM) into relational databases (PSM) via UML (PIM) [27].

Another aspect that is observable empirically in the use of MDA in practice is the reusability and streamlining of software production. For cheaper and more quickly delivered software, not just code generation is needed. However, reuse of existing components also because human beings still do their design, verification, improvements, and other maintenance tasks. There are, for example, attempts of reusing PIM models using ideas from the already mentioned 4+1 model [117].

2.2.2 Model as a Code

The article *Conceptual Model Programming: A Manifesto* [118] describes a way of programming using models instead of traditional writing source code. This idea is very ambitious, but a significant requirement is not fulfilled. A holistic conceptual model that would cover structure, processes, but also security, user interface, connections of external resources (e.g. web Application Programming Interfaces (APIs), sensors, or webhooks) is vital. Moreover, a serious question remains – Is such an approach even appropriate? In conceptual models, we want to export and capture the essence of the problem domain and abstract from implementation details such as UI or communication protocols.

According to the manifest, (*holistic*) conceptual model replaces completely source code of software application and is itself and alone used to produce the application. It removes all the problems with inconsistency between model and implementation. Another advantage of a runnable conceptual model is the evolvability of the resulting system that is ensured on the conceptual level and intricate source code patterns, or other low-level improvements are not needed anymore.

Conceptual Model Programming (CMP) is in its principle similar to MDA without seeing any source code and ability to edit it [108]. Nevertheless, oppositely to MDA, CMP is still further theoretical and necessary tooling, and modelling languages are missing. Nevertheless, ideas of CMP are not only for the academic field and are in some way implemented in the commercial area. Various "compilers" of conceptual models are developed, for example, IBM Rational Rhapsody or WebRatio [118].

2.2.3 OO-Method

The OO-Method is an implementation of the Automated Programming Paradigm that started in 1992, and the following work was published up to 2007. Sadly, most of the work is more academic-oriented and tooling for applying the method in practice is not openly available. The core ideas are powerful and inspirational – it covers software production processes and the environment by combining conventional and formal methods leading into object-oriented implementation interconnected with models. [28, 119, 120]

When compared to MDA, it covers modelling on the PIM level. For the CIM level, communication analysis to describe requirements and processes in the domain is based on the research of other main modelling languages. The core is built on a formal object-oriented model called OASIS. It combines multiple types of models (functional, interaction, state transition, object, and others), where some are textual, and some are graphical. As a result, it together composes a complete specification to create the implementation. [119]

A recent work by Martins [121] proposes an ontology-driven approach for evolving OO-Method into the so-called OntoOO-Method. It explains how the works related to OntoUML, "Model as a Code" and model-driven development represented by OO-Method can be harmonically integrated to support ontology-driven software development. An issue with respect to code generation from the conceptual model is the specification of non-functional software requirements. For example, MDA partially deals with that by levels of models and selection of code template. A relation between such requirements specification thought *i** framework and OO-Method is analysed in [122]. It follows the older work [123] that deals with relation to requirements engineering in a broader scope. It explains that all of the requirements are not usually part of the domain models; therefore, they must be incorporated in another way.

2.2.4 Scaffolding

With the growing popularity of web frameworks, specific frequently repeated patterns and architectures are identified and well-described. As for the architecture of web applications, multi-tier (usually three-tier) architecture combined with Model-View-Controller (MVC) or Model-View-Presenter (MVP) is commonly used. In each layer, a developer does many repeated steps related to the data and flow models that come from a conceptual model describing a problem domain that the application is supposed to support: data entity for a database, form to create or edit instances of the entity, frontend and backend data validation, controller to process-related user interaction, a template to display detail of an instance, and so on.

Many web frameworks of various programming languages (such as Symfony, Nette, Ruby on Rails, Django, Spring, Node.js, and many others) as well as external tools provide significant helper in this task called *scaffolding* [29, 124]. This approach is purely practical than academic as the three previously described but still crucial for our research. A developer provides a description of some entity, and all necessary source code is generated using templates. The entered information is sufficient to produce a very simple working Create, Read, Update, Delete (CRUD) web application that can be then easily enhanced. Provided information typically covers just the name, attributes, and relationships of data classes. Additionally, constraints about attributes and relationships can be supplied to guard the integrity and validate user forms.

2.2.5 Eclipse Modeling Framework and Ecore

The EMF is a platform for (meta)modelling and model-driven development that enables to build applications and tools on top of structured data models. EMF is Java-based and tightly bound to the Eclipse development environment. The core of EMF consists of:

- *Ecore* – metamodel for describing (meta)models together with runtime support such as XMI serialization and API for efficient manipulation with EMF objects,
- *EMF.Edit* – framework for building editors of EMF models,
- *EMF.Codegen* – code generation facility for generating editors for an EMF model. [125]

In summary, it allows to design of a custom modelling language in Ecore, create an editor for it, and streamline code generation. Moreover, it is possible to (re-)use parts of EMF for other purposes, e.g. to load Ecore models from XMI and work with them in custom Java applications. There are multiple related projects to the core EMF, such as EMF.cloud [126] that supports web-based modelling tools or EMFStore [127] for collaboration and versioning. The robustness of the whole EMF is also a source of over-complexity that complicates its use in a non-standard way.

The related Eclipse Model Development Tools (MDT) support other well-known modelling languages aside from Ecore. Eclipse UML2 is an implementation of the UML 2.x

OMG metamodel based on EMF for use within the Eclipse platform. The implementation is supported by a tool called Papyrus [128]. It can be easily used with EMF and provide enhanced expressiveness compared to Ecore (e.g. by allowing to model behaviour). Similarly, there is an implementation of OCL to specify constraints for EMF-based models. [125]

The Ecore metamodel instance of itself, i.e. Ecore is defined in terms of Ecore. The striking resemblance with the well-known UML Class Diagram is not accidental. As (complete) UML is defined in terms of OMG's Complete MOF (CMOF), Ecore is, according to Ed Merks [129], a de-facto reference implementation of Essential MOF. It also works with concepts of classes, attributes, references (relationships), operations, or packages. Ecore is well-known as a subset of UML with good tooling support even beyond EMF. Moreover, it is also possible to use other Eclipse metamodels such as UML2 (implementation of OMG's UML 2.x metamodel) or OCL metamodel. Finally, Ecore as a whole EMF is well-documented [125, 130] which is also one of the conditions for its wide adoption and growing community around it.

2.2.6 Conceptual Model Transformations and Ontologies

As explained, various modelling languages have different advantages in specific use cases (e.g. level of detail, granularity, focused aspects, or tooling support). It may be desired to have a model in multiple languages and keep them consistent to grasp the advantages of multiple modelling languages. There are many transformations proposed, defined, and prototyped to support these needs. Work by Cibrán [14] allows to translate of BPMN models into UML Activity diagrams with the use of ATL. Khlif, Ayed, and Ben-Abdallah [13] propose a set of transformation rules from BPMN to various UML diagrams.

We already reviewed in detail several UML-to-OWL (and vice versa) transformations and presented our case-study-based review [A.8]. A proposed QVT transformation has been evaluated as the best among the selection. The XSLT transformations lost many details and did not support the direction from OWL back to UML. There is also described one transformation from UML to OWL described using a custom algorithm working with tabular serialization of a UML model. Other modelling languages are also being investigated in terms of transformation or relating with OWL ontologies, for example, BPMN-to-OWL [131] or ORM-to-OWL [132]. For OntoUML, there is the so-called gUFO [133], a lightweight implementation of the UFO in OWL that allows direct encoding of OntoUML models in RDF.

2.2.7 Ontology Mapping and RDF Transformations

As we are about to design a gateway ontology, ontology mapping and RDF/OWL transformations are essential topics. There are many tools for ontology mapping; an overview is provided in [134]. An approach based on stable matching is proposed in [135] which seems to result in more efficient (both in quality and speed) ontology matching. A bit older work by Euzenat and Valtchev [136] uses similarity to match terms from ontologies. However, the method is limited only to the OWL Lite sublanguage.

It is better to use only RDF transformations directly without a specification of the alignment between ontologies for some use cases. Again, there are multiple options. STTL [35] utilizes SPARQL with a new extension for templating that allows turning queried triples into any textural fragment (which can be again RDF). RDF Mapping Language (RML) [33] works in the opposite direction than STTL. It allows specifying a mapping from textual formats (e.g. JSON or XML) to RDF. As RDF can be represented as XML, it can serve as RDF transformation language. Finally, TRIPLE [36] is a language for specifying rewrite rules in RDF. The TRIPLE is a bit older work, and most of its functionality can be done simply by creating SPARQL `CONSTRUCT` queries. The approach of using SPARQL is also suggested in our work where we compose the queries from pattern-based mappings [A.11].

2.2.8 Ontology-Based Software

From the perspective of using OWL ontologies in software development, i.e. to create ontology-based software, there are also several interesting works on transformations to various programming languages. For instance, [137] describes a generation of data models in Go from OWL ontology (oriented on structural aspects), or [138] focuses property axioms and their representation in Groovy (oriented on rules of relationships). Ontologies can be used even further in the software development process as a basis for testing [139]. Next, there are mapping frameworks for aligning object-oriented programming with ontologies [140]. For example, JOPA [141] enables efficient ontology-based information system design in Java (and compatible JVM languages, e.g. Kotlin).

2.3 Normalized Systems

Normalized Systems (NS) [1], a theory developed and used in practice at the University of Antwerp and its NSX spin-off company, deals with the evolvability of systems in general by the application of elementary principles ensuring fine-grained modularity. The primary purpose and application in practice are to develop evolvable enterprise information systems in the form of web applications. By evolvable, it is meant without combinatorial effects that are obstacles to change.

Software systems need to be changed over time, and their complexity grows. If there are ripple or combinatorial effects, each change takes more and more effort to increase the system's complexity. It is more efficient to develop an entirely new system and abandon the old over-complex one at some point in the system's lifetime. Normalized Systems are evolvable and do not suffer these effects, allowing them to be changed over time easily.

2.3.1 Normalized Systems Theory

The Normalized Systems Theory (NST) [1] clarifies how it should be system structured to achieve evolvable modularity, i.e. have a fine-grained modular structure with combinatorial effects eliminated or encapsulated and "under control". It does so by using principles from system design, software engineering, but also thermodynamics. The core principles are:

- *Separation of Concerns* – It is a well-known design principle for separating a computer program into distinct sections, such that each section addresses a separate concern. Every part of the application should be focused on its single purpose and goal. Nothing should do multiple, even slightly different, things.
- *Data Version Transparency* – Data entities used as input or output in actions must be updated without impact (combinatorial effect) on actions.
- *Action Version Transparency* – Actions that are called by other actions must be able to be updated without impact (combinatorial effect) on the calling actions.
- *Separation of States* – The calling of an action by another action must have state separated (no combinatorial effects of the specific call to the actions).

In contrast to other software development methodologies that target modularity, NST proves that applying the principles avoids combinatorial effects. Therefore, there is no vagueness or ambiguity as in traditional methodologies that propose several patterns but do not precisely specify how to use them or maintain the resulting system. It also explains how to design and develop such software systems. The key to evolvability is, next to the fine-grained modular structure, also a code generation technique. Although primarily intended and applied in software engineering, the theory is applicable in any domain where systems are being designed. [1, 142]

2.3.2 Metamodel and Meta-Circularity

Normalized Systems in software use the core principles to create modular applications with so-called *Elements*. They represent different aspects of the problem domain to be implemented in the NS application with evolvable modularity. The problem domain and requirements are captured using a model of these elements. For example, Figure 2.8 shows how structural and behavioural aspects are modelled in an application for booking flights. The implementation modelling method can be considered a precise technical specification since the implementation is created from such model and supplied technical details (e.g. what framework should be used for data persistence). The Elements metamodel is improved over time to cover more cases and avoid more custom code in the applications. There are five types of elements:

- *Data Element* – Description of data structures or classes including its attributes as data or link fields, connected data children, finders, and various metadata as options. There are specific data elements to express their usage, e.g. primary, taxonomy, or history.

- *Task Element* – A task models an execution of some actions on a data element. The result of such execution can be success or failure. Although tasks are used within workflows, NS needs to keep them independent.

- *Flow Element* – Connection of tasks into a process is expressed via a flow element containing information about a workflow for a single data element.

- *Connector* – Allows interaction with external systems and users in a stateful way.

- *Trigger* – Serves for scheduling or other planning procedures of (semi-)automatic workflows [1].

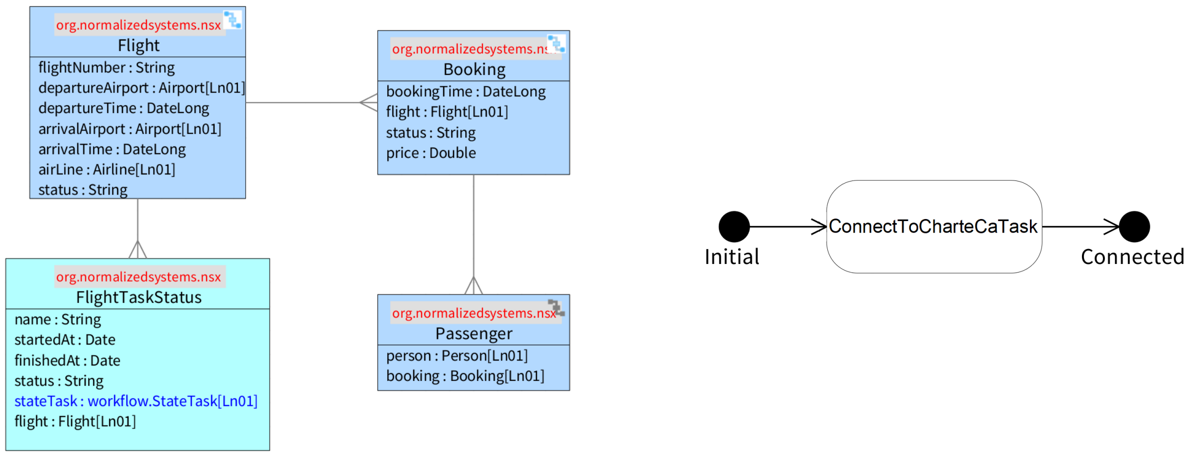

Figure 2.8: Example of structural and behavioural model in Normalized Systems

A model of Elements forms a Component that is re-usable in many applications. For example, a flight booking application may use the same components for user authentication as an application for solar panels management. The generic components that are widely used (e.g. for user management, workflow scheduling, or assets) are called *base components*. The NS Elements metamodel is an instance of itself, e.g. Data Element and Task Element are modelled in the metamodel as data elements as shown in Figure 2.9. It enables the Meta-Circle, which helps the NS metamodel and tooling also evolve easily. [30, 143]

The modelling with NS metamodel is oriented on evolvable implementation. Therefore, it avoids constructs that cause ripple effects. For example, there is no specific construct for enumerations as is known in both conceptual modelling languages and object-oriented programming languages. It must be modelled as a data element with a set of pre-defined instances. Similarly, there is no inheritance, i.e. generalisation-specialisation relation. It can be simulated only using link fields that represent associations. We proposed and investigated several patterns on how to transform inheritance [A.3, A.9].

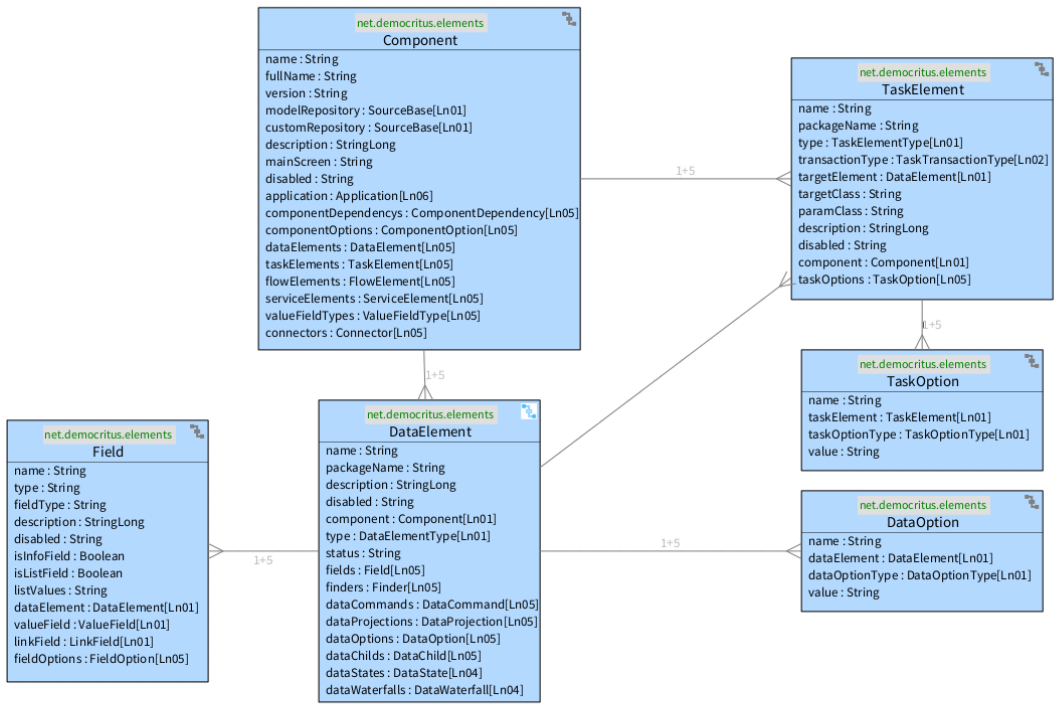
net.democritus.elements
Component
name : String
fullName : String
version : String
modelRepository : SourceBase[Ln01]
customRepository : SourceBase[Ln01]
description : StringLong
mainScreen : String
disabled : String
application : Application[Ln06]
componentDependencys : ComponentDependency[Ln05]
componentOptions : ComponentOption[Ln05]
dataElements : DataElement[Ln05]
taskElements : TaskElement[Ln05]
flowElements : FlowElement[Ln05]
serviceElements : ServiceElement[Ln05]
valueFieldTypes : ValueFieldType[Ln05]
connectors : Connector[Ln05]
net.democritus.elements
TaskElement
name : String
packageName : String
type : TaskElementType[Ln01]
transactionType : TaskTransactionType[Ln02]
targetElement : DataElement[Ln01]
targetClass : String
paramClass : String
description : StringLong
disabled : String
component : Component[Ln01]
taskOptions : TaskOption[Ln05]
1+5
net.democritus.elements
TaskOption
name : String
taskElement : TaskElement[Ln01]
taskOptionType : TaskOptionType[Ln01]
value : String
1+5
net.democritus.elements
Field
name : String
type : String
fieldType : String
description : StringLong
disabled : String
isInfoField : Boolean
isListField : Boolean
listValues : String
dataElement : DataElement[Ln01]
valueField : ValueField[Ln01]
linkField : LinkField[Ln01]
fieldOptions : FieldOption[Ln05]
1+5
net.democritus.elements
DataElement
name : String
packageName : String
description : StringLong
disabled : String
component : Component[Ln01]
type : DataElementType[Ln01]
status : String
fields : Field[Ln05]
finders : Finder[Ln05]
dataCommands : DataCommand[Ln05]
dataProjections : DataProjection[Ln05]
dataOptions : DataOption[Ln05]
dataChilds : DataChild[Ln05]
dataStates : DataState[Ln04]
dataWaterfalls : DataWaterfall[Ln04]
1+5
1+5
net.democritus.elements
DataOption
name : String
dataElement : DataElement[Ln01]
dataOptionType : DataOptionType[Ln01]
value : String

Figure 2.9: Fragment of NS Elements metamodel [30]

2.3.3 Expansion and Craftings

The code templates, together with mapping from NS models, are called Expanders. From a technical perspective, those are Java String Templates and XML mappings. It allows the generation of any textual files from NS models. There is no limitation in terms of output from the expansion; it can be any textual file (e.g. Java source code, XML configuration, documentation or SVG file). The expanded code base is expected to be enhanced by adding custom code fragments to implement functionality that cannot capture using elements. For example, a field of a data element can be modelled as *calculated*, but the calculation must be implemented as a Java code fragment. [30]

In traditional MDD methods, such enhancements in the code would become a source of inconsistency and blocker of future re-generations. However, NS provides ways of handling custom code. There are two types of such craftings:

- Insertions – The generated files contain anchors between which a custom code fragment can be inserted. The anchors are specified directly by the used expanders (code templates). The generated files are typically part of `gen` directory.

- Extensions – Custom source code files or whole packages created in `ext` directory. For example, a package implementing an API client of an external system would be an extension. Then, it can be used in the application via insertions, e.g. call external API when a task is executed.

To avoid overwriting the craftings upon re-generation (or so-called rejuvenation) of a system, they should be harvested. The harvesting procedure basically goes through the code base and stores both insertions and extensions in the designed location. Then, when a system is rejuvenated, it takes the selected expanders, NS model of Elements, technical details and harvested craftings to generate the codebase. There might be several reasons for re-generation: a change in the model, updated expanders, or a different underlying framework (cross-cutting concern). [1, 30, 142]

Finally, the expanders can also be variable using *features*. It aims to include pieces of code to multiple expanders. Then, a feature can be enabled by an option from the NS model (e.g. by specifying a particular data option for a data element) or another condition. It typically adds some logic to multiple artefacts at once to extract cross-cutting concerns from the expanders (decoupling). As expanders are very variable and can become complex, there is also a prepared way to test them easily. Testing is done by specifying a fragment of input model using *onion specs* and then matching the generated result with expectation. [30]

2.3.4 Prime Radiant and NS Modeller

Appropriate tooling support is needed to work with NS models and expanders effectively and to create and run the resulting enterprise application. In the past, expanders were started by command line scripts together with a basic XML description of the model with elements. As this was hard to do for non-technical people, a tool called Prime Radiant [30] (the name comes from Isaac Asimov's Foundation series) provides a user interface in a web application with various views and forms. It is simple to use, and it is itself an evolvable NS application. Prime Radiant covers the whole process from designing NS elements, through specifying flows and runtime details, up to deployment technologies and running the instances of applications.

The Prime Radiant [30] helps with design, but it is still a text entered with some non-trivial forms and displayed in tables. That is not easy to grasp by stakeholders from a business who are used to the graphical representation in diagrams. For that purpose, the NS Modeller has been developed based on the OpenPonk modelling platform. It easily allows creating a visually close model to ER or UML for data elements with flows and to specify many details like finders or data children. Furthermore, model validations are possible, which is a totally new feature that is missing in Prime Radiant and can lead to build errors of the NS application. Later during our work, the Prime Radiant become superseded by Micro Radiant (µRadiant) [144].

Moreover, another tooling for NS is also generated from the NS metamodel. For example, the Java library for importing and exporting NS models through their XML serialization. It uses so-called *Tree* projections to store elements in Java and to allow manipulation of them. Similarly, there are *Composite* projections that have resolved relations (forming a graph instead of a tree). Due to the meta-circularity, it is possible to simply re-generate such libraries whenever the metamodel is updated. Other tools and libraries that enable work with NS can also take such an advantage if they are implemented (at least partially) using expanders. [1, 30]

2.3.5 Normalized Systems in Other Domains

Although NST is currently intensively used in the domain of software engineering, it is applicable to other domains, and we say that to any domain of human activities. Whenever some structured system is designed (including buildings, organisations, documents, software and hardware), principles of NST can be used to create a fine-grained modular structure and enable evolvability. For example, having modular evolvable highways made of connected building blocks (with all pipelines, lights, signalisation wiring, and so on) would allow faster road repairs as discussed in [1].

There is extensive research of using it for working with study plans and documents at the university [145], [146], and [147]. We also contributed to this work with our expertise in conceptual modelling [A.1] and in the extended version, we created a prototype of a documentation system for OntoUML [A.2]. The modularisation of documents with the use of ontologies is also further proposed in [148]. A promising work described in [149] applies NS in the field of user interface design. We also applied NS principles for message formatting in service-oriented architecture [A.4]. Finally, we also pursued a way of generating NS models directly from textual specification in natural language (English) using the tool Textual Modelling System (TEMOS) [A.14]. These contributions to the NS knowledge base are crucial for grasping the true power of evolvability and the possibility to apply it in various use cases.

Chapter 3

Overview of Our Approach

"It is not the strongest of the species that survives, nor the most intelligent; it is the one most adaptable to change."

Charles Darwin

This chapter describes the overall architecture of our work that we developed while following the Design Science Research (DSR) methodology. As such, we also explain how it was improved over time during the design cycles. First, we summarise the functional and non-functional requirements together with the means of verification according to the relevance cycle of the DSR. Then, we describe the architecture, i.e. types of our designed artefacts and how they together compose the whole supporting transformations between Normalized Systems and conceptual models. We aim to theoretically describe the integrations and transformations and provide a practical means for realising these theoretical research results in the form of a software tool.

With the architecture described, we can discuss its essential aspects and properties that are crucial for fulfilling the requirements and the operations in practical use cases or future development. The discussion is also related to both the evaluation of artefacts in the design cycle and confronting them to the requirements. Finally, we provide a brief overview of alternative approaches that were not used but provided essential lessons learnt. The chapter is based on our previous architecture description [A.18], but it was further refined during other works [A.10, A.11, A.12] according to the DSR design cycle into the current state that we present here.

3.1 Design Requirements

The relevance cycle of DSR describes the requirements that come from the environment (our target domain) – the needs of Normalized Systems (NS) developers and in the domain of software engineering in general. The high-level objectives are clarified in Section 1.2; however, to design the architecture and partial artefacts, more granular requirements are needed. Although the ultimate goal is to design a *Normalized Systems Gateway Ontology for Conceptual Models*, the requirements are also defined with respect to the components around the gateway ontology that form a whole framework for transformations between conceptual models and NS. We can further refer to these requirements, as various parts of the work contribute to their successful fulfilment.

To formulate the requirements, we follow the Requirements-Driven DSR proposed by Braun et al. [41] which clearly explains the need for a more detailed requirements specification in DSR. The problem analysis and statement sets the research problem and objectives with respect to a certain context (domains) with stakeholders. From these various types of requirements emerge, as shown in Figure 3.1, for example, context causes contextual requirements (i.e. constraints), the research problem has problem features that cause feature-related and functional requirements, as well as theory-based requirements.

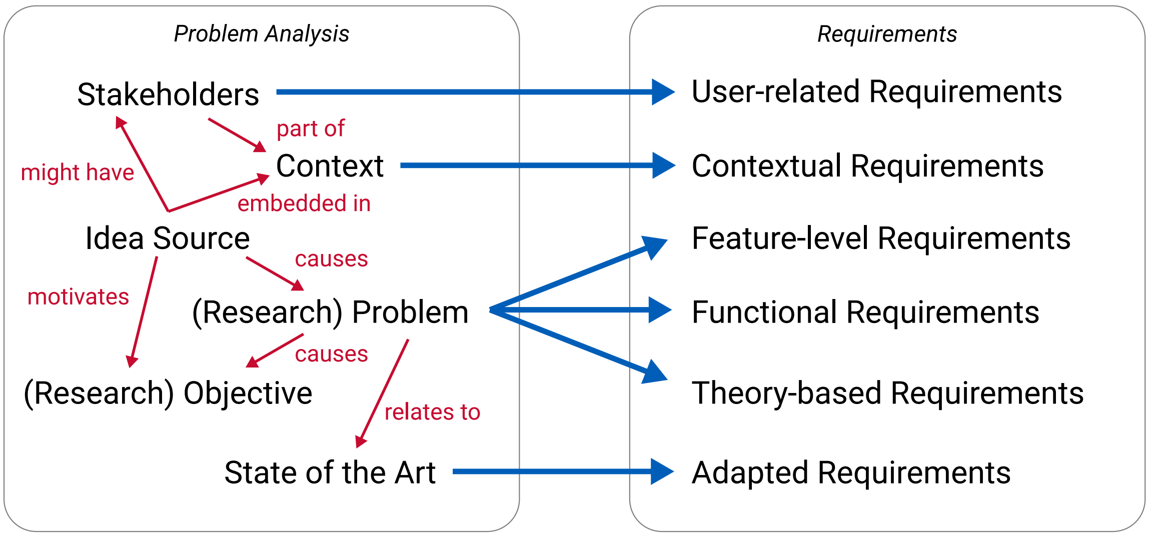

Figure 3.1: Excerpt of Requirements-Driven DSR (according to Braun et al. [41])

Formulating these requirements is crucial as it sets what must the design support and what will be evaluated – requirements serve as evaluation criteria. Moreover, we want to demonstrate the proposed design on a practical level (not just theoretical) and create "executable artefacts", so the DSR requirements specified in this section are also the basis for the software requirements used to create prototype or reference implementation of the designed artefacts.

3.1.1 Feature-Level Requirements

Based on the research problem and its objectives stated in Section 1.2, we specify the following requirements classified as feature-level. We intentionally do not specify the details (atomic functional requirements [41]) at this point; these are further discussed in the corresponding chapter where the partial design artefact for the specific features is described.

FR1 **Transform a structural conceptual model to an NS Elements model.**
The framework must allow the transformation of structural information about a domain from a conceptual model into NS Elements. It will be demonstrated using Unified Modeling Language (UML) Class Diagram as the widely used mean of structural modelling. To also show it can handle conceptually richer models, OntoUML models encoded with the gUFO will be supported as well.

FR2 **Transform a behavioural conceptual model to an NS Elements model.**
The framework must allow the transformation of behavioural information about a domain from a conceptual model into NS Elements. It will be demonstrated on UML Activity Diagram, Business Process Model and Notation (BPMN), and Business Object Relationship Modelling (BORM) models.

FR3 **Transform a fact-based conceptual model to an NS Elements model.**
The framework must allow the transformation of behavioural information about a domain from a conceptual model into NS Elements. It will be demonstrated on Object-Role Modeling (ORM) models.

FR4 **Enable semantic integration of models prior to transformation.**
The framework must enable to integrate various information from different conceptual models. It will provide a complex domain description which is the input for transformation to NS. This feature must not lay any obstacles in the combination of conceptual modelling languages. For example, UML Class Diagram models must be possible to integrate with BORM models in the same way as for ORM or UML Activity Diagram models.

FR5 **Allow to change the specification of transformations.**
The framework must not be limited to initially supported conceptual modelling languages, and it must allow to change them out-of-the-box, i.e. without (re-)programming the framework. First, it must allow to further extension or adjust the mapping and transformations as metamodels of both conceptual modelling languages, and NS Elements may change over time. Second, new conceptual modelling languages may need to be supported. Finally, alternative transformations of the same language may also be needed for specific use cases.

FR6 **Maintaining consistency through back-transformation and checks.**
The framework must allow transforming from the NS Elements model back to conceptual models for keeping or checking consistency. It gives meaning to the word “gateway” in its name as it should also provide a way back.

The first three requirements (FR1, FR2, and FR3) are related to the second research question RO2 as they target to transform different kinds of conceptual models to NS models. In all cases, the target is to avoid or minimize information loss. The mentioned kinds of conceptual models were presented in Section 2.1.1. The requirement FR4 is defined to realise RO1. FR5 is a predisposition for dealing with evolvability issues stated in RO3. Finally, FR6 supports RO4.

3.1.2 Theory-Based and Contextual Requirements

Aside from fulfilling the feature-level requirements in terms of models integration and transformations with keeping consistency described in Section 1.2, there are theory-based and contextual requirements on our solution that are desirable for successful contribution applicable in practice and extensible for the future. Concerning the Requirements-Driven DSR [41] where theory-based requirements are also called non-functional and contextual requirements are constraints, we also include the user-level requirements as part of contextual, i.e. as constraints coming from the domain. Each of the following criteria must be taken into account when developing the framework.

NR1 **Evolvability** – Our solution must be well designed according to the core principles of Normalized Systems theory [1]. A fine-grained modular structure will allow good maintainability and easier development in the future, i.e. sustainability. It is directly related to RO3.

NR2 **Extensibility** – Extensibility together with versatility are highly related to evolvability. It should be possible to easily extend the framework with a new modelling language or notation, update the existing one, and introduce new transformations without any negative effects on the existing functionality and usability.

NR3 **FAIRness** – The four basic principles of making (scientific) data findable, accessible, interoperable, and re-usable (FAIR) were described in [150]; nevertheless, it is usable loosely on any project. Being FAIR leads to successful and easy-to-use data, software, or any research results.

- NR3-F **Findability** – Having a useful solution without anybody knows is losing its sense. Publishing it in visible places and using registers where the target audience is focused is necessary.
- NR3-A **Accessibility** – Our solution should be published in standard way for its specific parts. For example, the framework as software could be published via standard channels and package index for the selected language. Also, a clear and standard license must be used to define who and under what terms can use, distribute, and edit the work.
- NR3-I **Interoperability** – Standard or widely used technologies, procedures, and conventions should be used. First, it helps with the design of the solution since those

are well documented and supported. Secondly, it assures that the solution is easy to adopt and used together with others.

NR3-R **Re-usability** – The partial building blocks in the overall solution should be possible to share internally and, if applicable, even used externally. It allows to minimize repetitions and "reinventing the wheel". For example, others may take advantage of the existing parts of our framework and build their own for a different use case.

NR4 **DRYness** – By having architecture based on a fine-grained modular structure, we should also comply with the well-known "Do not repeat yourself" principle. The repeated things should be a source of re-use. That can be done by using generalisations, compositions, or code generation. The generated parts will, of course, manifest into repeating patterns. There should always be only one way of accomplishing a particular goal.

NR5 **Development Stack** – To ease up the adoption and allow future development together with other NS tooling, it should comply with the development stack used (Java programming language, Groovy for scripting, Git Version Control System (VCS), and others). The parts of the framework directly related to NS models need to be created through expansion, i.e. custom expanders must be designed to generate code directly related to the NS (meta)model, which will also help in accomplishing NR1. It will also enable the use of other existing artefacts in the code base of NS tools.

NR6 **Documentation** – All parts and procedures must be well documented to enhance its adoption. Although the use of the framework (running transformations, checking consistency, and more) should be intuitive and straightforward for the users, the documentation should cover all functionality of the framework. Moreover, the ways of extending the framework, e.g. by supporting a new modelling language, must be documented as well. The documentation is expected to significantly overlap with the following chapters of this dissertation thesis.

NR7 **Multi-Platform Solution** – The framework must be possible to use on any standard computer platform supported by other NS tools (Windows, Linux, macOS).

3.1.3 Adapted Requirements

Finally, the Requirements-Driven DSR [41] specifies adapted requirements caused by State-of-the-Art, i.e. originated from existing solutions and knowledge base. It encompasses any requirements adapted or reused from previous comparable and reliable works. We already presented our knowledge base and related work in Chapter 2 where such requirements are also outlined. In our case, these requirements directly influence the formulation of our feature-level and theory-based requirements – Figure 3.1 as well as the DSR three cycle view explains that the research problem is connected to State-of-the-Art.

To summarise the adapted requirement, we use the following list. However, we do not further refer to these requirements as those are covered in the other requirements as further described for each individually:

AR1 **Conceptual Heterogeneity** – There is a vast number of modelling languages focusing on different aspects for a good reason; there are various aspects in the reality that may be significant for the domain, use case, or software system, i.e. the reason of modelling. The mentioned MDD method use a specific language, subset of languages, or devise its own modelling language. This requirement states that the conceptual heterogeneity in modelling must not be limited. The design must not lay any obstacles in incorporating any existing (and future) conceptual modelling language. For example, MDA is using UML (or other MOF-compliant languages) or EMF allows to use Ecore-compliant languages and models. Our design should provide a more flexible way than transforming a modelling language for custom metamodel compliance. This requirement is directly related to NR2 and NR3i. Moreover, it is manifested in FR1, FR2, and FR3 where different aspects are to be supported and demonstrated.

AR2 **Multiple Views on Domain** – In Chapter 2, we described several modelling languages and methods that allow us to combine models of different aspects to create a more detailed view on a domain. For example, UML allows combining class, activity, or state machine diagrams to describe both structural and behavioural aspects. Similarly, the OO-Method or Model-as-a-Code use different types of models combined to create software systems. In the ideal case, it would create a holistic view encompassing every information about the domain, and a complete system could be generated from it. Our design must support a combination of different models concerning AR1. It is reflected in FR4 that requires semantic integration of knowledge across models prior to transformation to NS.

AR3 **Transformations with Semantics** – There are existing ways of model transformations, e.g. Extensible Stylesheet Language Transformations (XSLT) or Query/View/-Transformation (QVT). Our design must enable performing based on semantics contained in models and not the syntax used to represent it. For example, XSLT is basically transformation of XML nodes that can represent the model according to its metamodel, but not necessarily. QVT provides semantic querying; however, is again limited to MOF-compliant languages. The requirement is loosely related to FR1–FR5 as semantics-based transformations are key for supporting conceptual heterogeneity as well as semantic integration and straightforward transformation specifications.

AR4 **Consistency between Transformed Models** – A known issue of the current model transformation and the MDD methods in general is keeping consistency across models on various levels. For example, MDA uses transformations from CIM to PIM and from PIM to Platform-specific model (PSM), but does not solve how concurrent changes in these models (or the generated implementation) should be handled for keeping everything consistent. NS uses the concept of craftings (insertions and extensions) that can be harvested and injected after rejuvenation. Our design must provide a way of consistency which is also required in FR6.

3.1.4 Verification and Evaluation

All requirements must be possible to evaluate during the design cycle of DSR and possible to verify within the target environment (field testing). It is essential to evaluate the framework artefacts in the design cycle iterations to ensure the quality and benefits of the results. For the functional requirements, FR1, FR2, and FR3 will be evaluated by performing transformation of various input conceptual models and checking the output NS models for correspondence, completeness, and information loss. Similarly, FR4 will be tested using multiple integrated input models and observing the results. FR5 will be evaluated together with NR1 and NR2 by creating actual mappings for conceptual modelling languages in several stages and improving them in design cycle iterations. The consistency mechanism of FR6 can be evaluated on the models used to test the first four functional requirements by checking consistency.

Some of the non-functional requirements can be implemented by design or by selecting underlying technologies. Then, the verification will be straightforward to state that the realisation (still) fulfils these requirements. For example, NR5 and NR7 will be taken into account when selecting the technologies and implementing the framework. The technologies will also highly affect NR3 as it requires standard and widely used technologies, procedures, and conventions. NR4 can be evaluated together with NR6 if there is always one way of doing things and no unnecessary duplication is present. The documentation will be verified with the collaboration of potential future users.

3.2 Framework Architecture

After setting the requirements, an overall architecture of the framework for transformations between conceptual models and Normalized Systems can be designed. It specifies the partial artefacts that can be developed according to DSR. The architecture itself is also an artefact in terms of DSR. In this section, we present the final architecture; however, the process of evolution due to design cycle is described in the next chapter.

3.2.1 Design Modularisation

The design of transformation between conceptual models and Normalized Systems through the gateway ontology is split into loosely coupled modules according to the separations of concerns principle. The concerns that are separated were identified upon design cycle iterations (listed in the direction from conceptual models to NS):

- representation of conceptual models,
- mapping of a conceptual modelling language using our gateway ontology,

- definition of NS Gateway Ontology for Conceptual Models,
- transformation of conceptual models according to the mapping,
- transformation between NS and mapped conceptual models.

By incorporating this fine-grained modularisation into our design, we promote both extensibility and evolvability. It is more efficient to extend the transformation by changing or creating new small modules. It also removes the burden of maintaining the product of all version variants (e.g. different conceptual modelling languages and their versions). For example, if we have n conceptual modelling languages mappings and support m versions of NS metamodel, it is required to maintain only $n+m$ artefacts instead of $n \cdot m$. It becomes even more significant when we consider versions of the gateway ontology, transformation tools, and others.

To describe the modularisation, we first split the logical and technical view where each approaches the modules in different ways:

- **Logical View** – explains the structure of the whole framework around Normalized Systems Gateway Ontology for Conceptual Models, what artefacts are there, and how are they interconnected. It is further separated into three planes: conceptual modelling, gateway, Normalized Systems (as described further in the subsequent subsections).
- **Technical View** – outlines the implementation and workflow of transformations between conceptual models and Normalized Systems. It is a high-level blueprint for the technical realisation of the proposed framework.

For the common representation of conceptual models, the mappings, and the gateway ontology, we use Resource Description Framework (RDF) and Web Ontology Language (OWL). It has been identified as the most suitable technology based on the set requirements, mainly for its versatility and wide use. However, we also considered other technologies as described in Section 3.6.

3.2.2 Grounding of our Design

The design is highly affected by previous knowledge, experience, and identified similarities that provide a touch-base with existing artefacts in the real world. Such analogies play a significant role in problem-solving and science. First, analogies are helpful for design – reusing a specific aspect or aspects of an existing and verified occurrence. Such things that we use for inspiration can be even from a different domain, often from nature or life sciences. Then, it also helps us to easily grasp new ideas and communicate complex solutions by relating them to something widely used and known. It has been argued that analogy is “the core of cognition”. [151, 152]

"Science is nothing but the finding of analogy, identity, in the most remote parts."

Ralph Waldo Emerson

In this subsection, we briefly provide analogies to our gateway ontology design that helped us devise and name the solution. The grounding using analogies provides intuitive but yet solid reasoning for certain design ideas and decisions. It should also help the reader to understand the various aspects properly and design decisions that we took, e.g. why we have layers and planes, or what the role of RDF is in our work.

3.2.2.1 Gateway Converting Information

According to Gartner[1], a gateway in computer networking "gateway converts information, data, or other communications from one protocol or format to another". Because the protocols for local networks and the Internet differ, a gateway often serves as a protocol converter allowing one to send and receive communications over the Internet. Generally, a gateway is described as "a product or feature that uses proprietary techniques to link heterogeneous systems". It must allow communication in both directions and maintain the connections. A gateway can also be seen as a "supervised entry point" through which messages (carries of information/knowledge) can enter a particular environment under given conditions and rules applied.

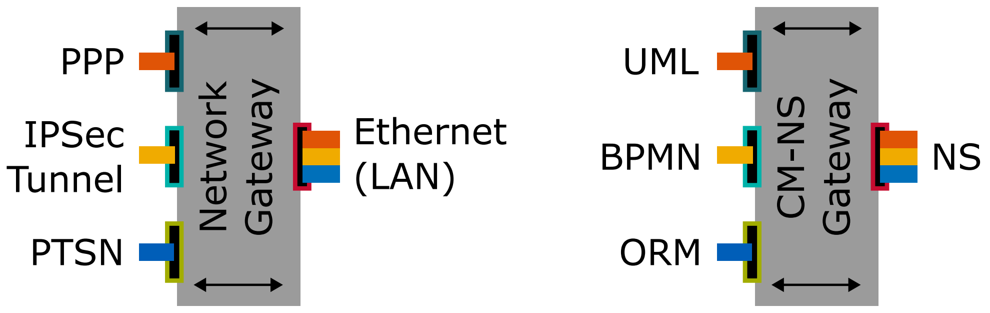

Figure 3.2: Gateway ontology analogy with network gateway

We aim to create a gateway for conceptual models to the environment of NS as shown in Figure 3.2. This gateway must receive compliant conceptual models of supported types (modelling languages) and pass the knowledge while maintaining the connection with the source. Moreover, it should also allow "communication" in the opposite direction and allow knowledge from NS be converted to "protocols" of conceptual modelling languages. The ontology serves as a way to specify the gateway and its operations and provides a way to define its modules – converters for modelling languages, i.e. mappings for transformations.

[1]https://www.gartner.com/en/information-technology/glossary/gateway

3.2.2.2 Lingua Franca for Knowledge Representation

Lingua Franca is "a language adopted as a common language between speakers whose native languages are different" [153]. In research on the international level, we can say that English is used as a lingua franca. For example, international conferences and journals use English, and similarly, this dissertation thesis is written in English, not Czech nor Dutch. When the Czech and Dutch natives meet, they are not going to learn one language from each other; they will most likely speak English. The reason is obvious; people would need to know all languages in the worst case. In this sense, English as a lingua franca has been a "medium" for transferring knowledge in a real-world scenario for many decades.

We have different conceptual modelling languages and the NS modelling language (defined by its metamodel). Our goal is to transfer knowledge between these languages. First, we will not create a particular translator between each modelling language and NS separately (for a person who knows all languages). Second, we will not invent another *Esperanto* for knowledge representation; however, re-use of existing is desired, and there are two possible options – RDF and XML Metadata Interchange (XMI).

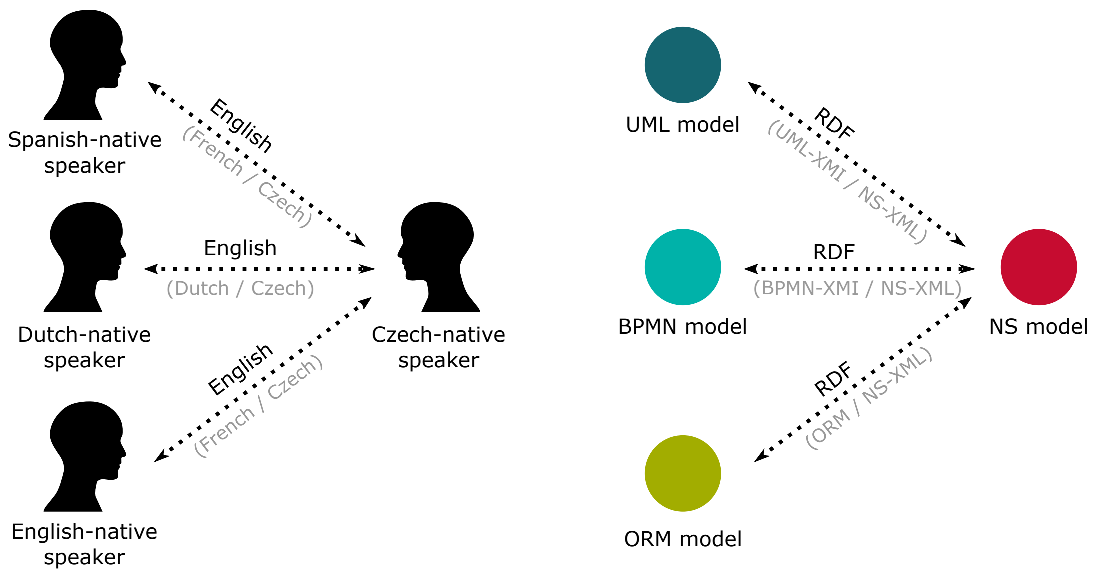

Figure 3.3: Gateway ontology analogy with lingua franca

In this work, RDF is used as a lingua franca for all representations of knowledge (both models and mappings) as depicted in Figure 3.3. However, it is not just representation, but a key to transformation due to its "grammar" and solid foundations in first-order logic. RDF is versatile, flexible, well-maintained, widely used, and mature related technologies are used mainly in Semantic Web and Linked Data domains. It enables machine-actionability and rich semantics through vocabularies and ontologies, e.g. in RDFS or OWL. In this sense, we see XMI as *Communicationssprache* (international auxiliary languages based on French) – it is closely related to modelling, and related technologies such as QVT are not used, so widely and cross-domain as RDF. Moreover, XMI has certain limitations, for instance, it is coupled with Extensible Markup Language (XML) whereas RDF has no such issues.

3.2.2.3 Vector Space, Dimensions, and Linear Transformation

A vector (or linear) space is characterised by its dimension, which basically specifies the number of independent directions in the space. A vector space is a set of vectors, and a subset of it is called a *basis* if its elements are linearly independent and span the vector space. Every vector space always has at least one such basis. For example, for a vector space $\mathbb{R}^3$ (3-space, a set of all triples made up of real numbers), the standard basis is a set $\{(1,0,0),(0,1,0),(0,0,1)\}$. Any other vector of the vector space can be expressed as a linear combination of these vectors of basis (sometimes called base factors). Then, a linear transformation (or linear mapping) is a mapping between two vector spaces that preserves the operations of vector addition and scalar multiplication.

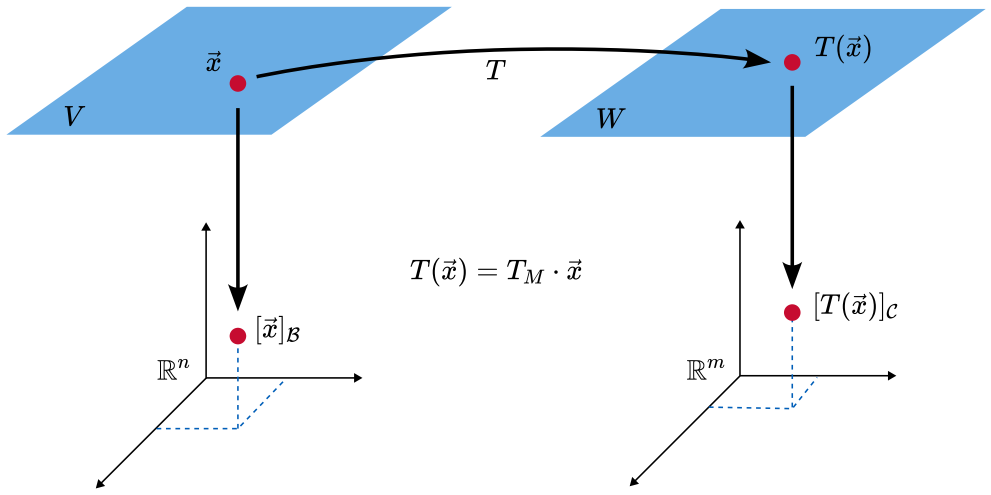
$\vec{x}$
V
T
$T(\vec{x})$
W
$T(\vec{x}) = T_M \cdot \vec{x}$
$[\vec{x}]_{\mathcal{B}}$
$\mathbb{R}^n$
$[T(\vec{x})]_{\mathcal{C}}$
$\mathbb{R}^m$

Figure 3.4: Gateway ontology analogy with vector spaces and transformation

Figure 3.4 shows a linear transformation T of a vector $\vec{x}$ from n-dimensional vector space V to m-dimensional vector space W. To express T as a matrix T_M, we need to choose bases $\mathcal{B}$ (in space V) and $\mathcal{C}$ (in space W). As T is a transformation from $\mathbb{R}^n$ to $\mathbb{R}^m$, the transformation matrix $T_M \in \mathbb{R}^{m\times n}$. T_M captures the relation between V and W using the bases $\mathcal{B}$ and $\mathcal{C}$ respectively.

For our problem of transformation between conceptual models and NS, we can see multiple dimensions as in vector spaces and we need to specify bases and base factors to be able to unify different vectors and capture their relations (i.e. specify transformation). Various modelling languages (metamodels) can be likened to vector spaces, with dimensions related to properties of those languages, e.g. aspects that they are focusing on and can express. A “standard basis” is needed in order to be able to specify the transformations, in terms of this analogy – to create transformation matrices, for transforming the underlying models (aka vectors). In our case, yet again, it is RDF, that is well-grounded, based on first-order logic.

3.2.3 Logical View

The logical view, as captured in Figure 3.5, captures the separation of different representations of models and their corresponding metamodels. According to our transformation needs, there are three planes. Two are related to the inputs and outputs of the transformation: *Conceptual Models* plane and *Normalized Systems* plane. Instead of directly designing transformations between models (and metamodels) on these planes, an intermediary *Gateway* plane is added. Its goal is to de-couple, allow re-use, and enable semantic integration of conceptual models prior to the transformation.

Each plane contains a set of artefacts and related concerns that may be further divided into sub-modules. Then, artefacts may be transformed between the planes – from conceptual models via the gateway to NS and vice versa. There may also be a specific transformation for metamodels between the planes, e.g. of NS metamodel to gateway plane. Such transformations support the primary transformation between conceptual models and NS. The logical view corresponds to our problem statement and objectives (from Section 1.2) where we visualised the gateway in Figure 1.1. The transformation (RO2) and consistency (RO4) research objectives are directly captured in the figure, while semantic integration (RO1) is related to the conceptual models plane and evolvability (RO3) will be described for each plane separately.

The goal is to transform conceptual models (level M1) that are expressed through specific modelling languages (i.e. its metamodel, level M2) – dashed arrows depict instance-of relationships. Of course, a model may have instances (if possible, level M0) and a metamodel should be formally expressed through metametamodel, which is not always the case. To allow the transformation to NS, we first need to shift the models (and metamodels) from a language/tool-specific format to our “lingua franca” and “common denominator” – RDF. There a well-defined metametamodel is guaranteed as well as the possibility to create instances. The *Normalized Systems* plane almost mirrors the *Conceptual Models* plane, just the metamodel is meta-circular; thus, serves also as the metametamodel. The *gateway* then contains the *gateway ontology* as an extension to NS metamodel turned to an ontology that allows and supports the creation of mappings between a specific conceptual modelling language (its metamodel) and NS metamodel as depicted by dotted arrows. The mapping then enables both transformation and consistency checks for the models.

3.2.3.1 Conceptual Models Plane

The conceptual models plane encompasses all the conceptual models, modelling languages, and their metamodels. It is very heterogeneous as each language focuses specific aspects and is related to different formats for representing the models. Moreover, even a single

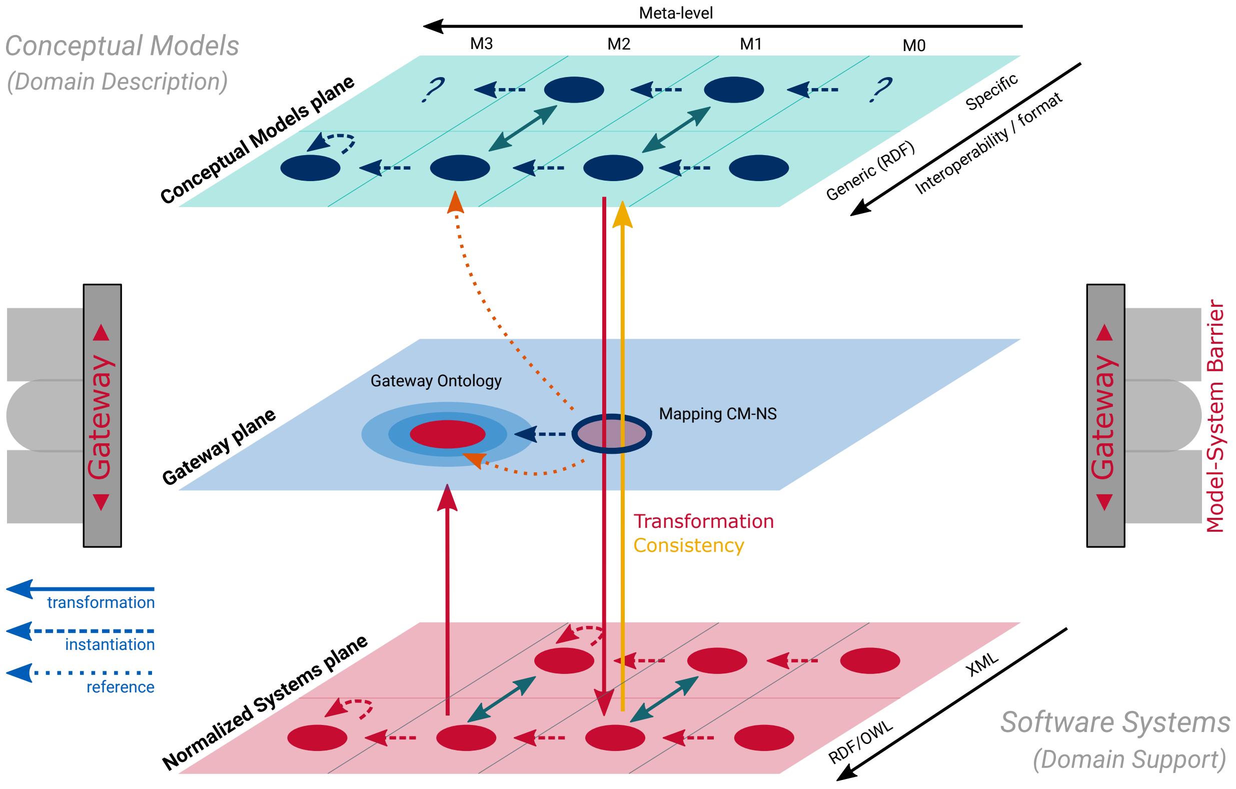

Figure 3.5: Logical view on the NS Gateway for Conceptual Models

conceptual model captures using a language of a certain specification version (e.g. UML 2.5.1) can be represented using several formats based on a modelling tool. The plane encapsulates for each language and format the concern of representation and the actual modelling of a domain using the language. Each single language specification as well as each single format for model representation. Although it separates the conceptual models representations in fine-grained modules, it also allows re-use if necessary. For example, a format used by various versions of a modelling language can be re-used (for instance, XMI for both UML 2.4 and UML 2.5).

Figure 3.6 shows that the plane covers different modelling languages and underlying models (depicted by different colours). We do not distinguish various language/tool-specific formats as our approach treats them in the same way and requires the transformation to a common language and format – RDF. On the side of specific formats, there could be for some languages even the metametamodel (M3), e.g. Meta-Object Facility (MOF). However, that would not be the metametamodel for all metamodels. When metamodels (M2) are transformed to Resource Description Framework Schema (RDFS)/OWL ontologies, it allows creation of instances (models, M1) as well as provides well-defined and common metametamodel. The models (M1) can be also transformed in RDFS/OWL to further allow instances (M0).

The use of conceptual models (and metamodels) transformed to RDF allows semantic integration as shown in Figure 3.7. The figure shows an integration of two models

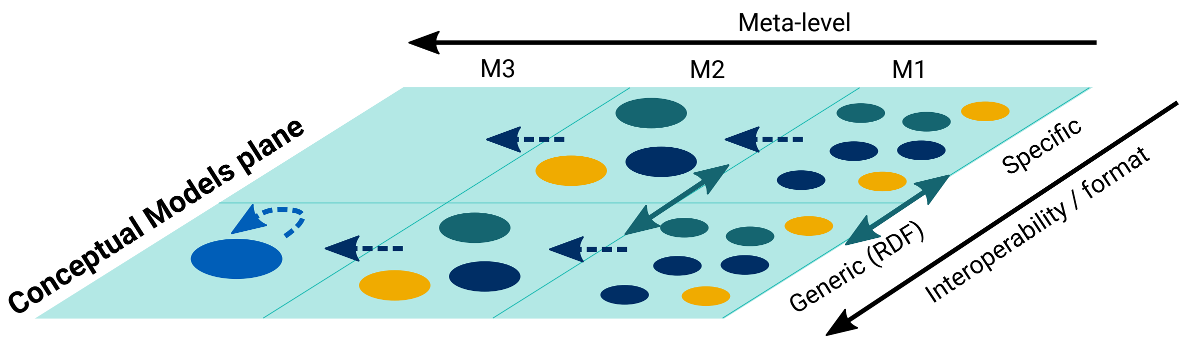

Figure 3.6: Conceptual models plane with various metamodels and models

created using two different modelling languages (metamodels). The metamodel for the semantically integrated model is a combination of the two original models with the possibility to specify additional constructs for binding them together. Then the rest of our architecture works in the same way as if the integrated models and metamodel were any other model and metamodel – specified mapping (created by re-using those from separate original metamodels) and transformation to/from NS.

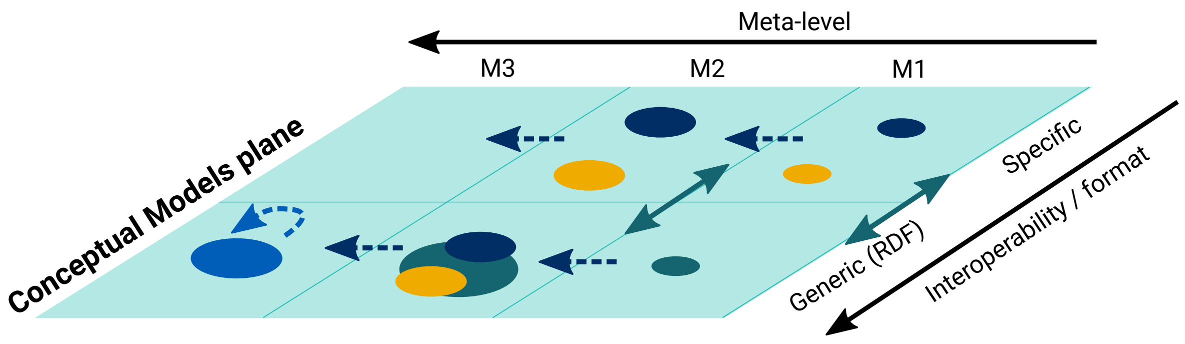

Figure 3.7: Conceptual models plane with semantic integration

3.2.3.2 Gateway Plane

The central gateway plane supports the transformation by providing means for its specification. As shown in Figure 3.8, the mappings on this plane represent the relationship between a conceptual modelling language (its metamodel) and the NS metamodel via the gateway ontology. The mappings are prescriptions for executing transformations between conceptual models and NS models and can be specified so it is possible to check consistency and/or transform in both directions.

The core part is the gateway ontology which creates the link between conceptual models and NS as well as the metamodels. It provides constructs for specifying the conceptual

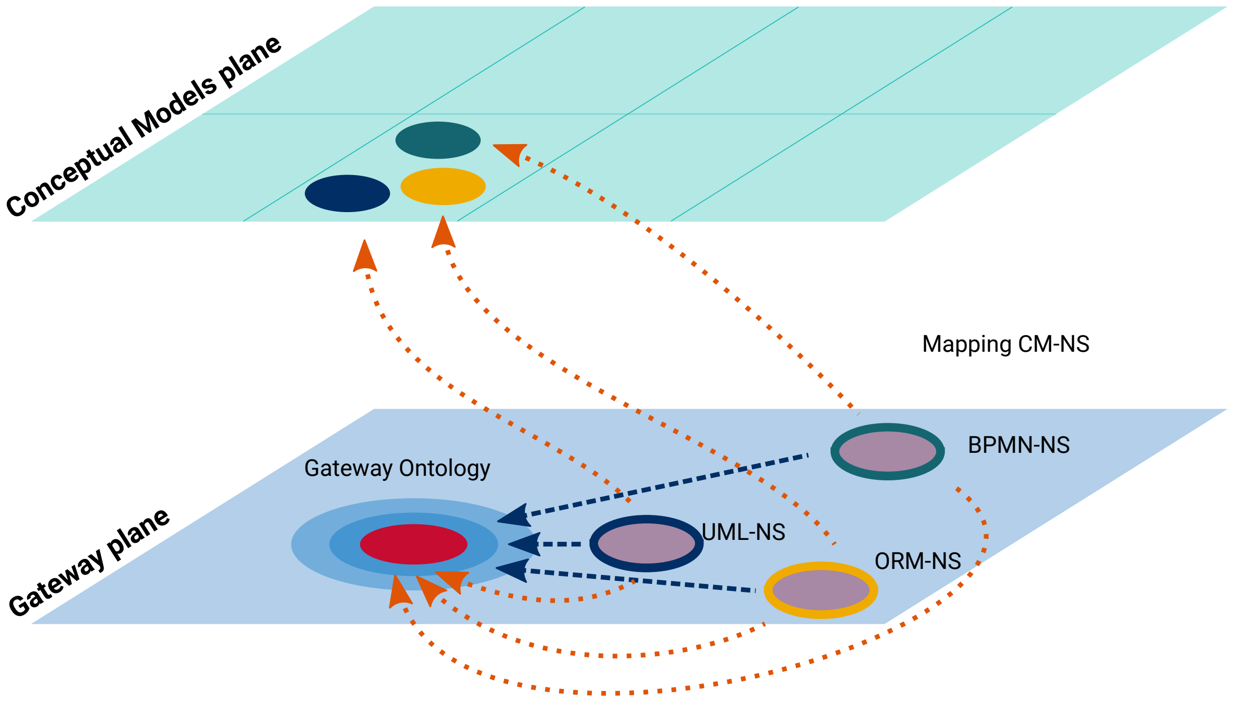

Figure 3.8: Gateway plane with gateways and links to modelling languages

modelling language mappings to NS. For example, how UML Class Diagram is related to NS Elements defines the relation between UML Class and NS Data Element. This knowledge can be then used to transform conceptual models to NS and vice versa. As the gateway ontology covers several key concerns (NS metamodel in RDF, re-usable extensions, and transformation specifications), it has been further divided into submodules called layers.

3.2.3.3 Normalized Systems Plane

The Normalized Systems plane represents the modelling part of NS using its tooling and specific formats as captured in Figure 3.9. The models from this plane can be expanded into artefacts (e.g. source codes of information systems) or further enhanced by technical details. In terms of modelling, it may be seen as a specialisation of a conceptual modelling plane with just a single metamodel and a particular purpose – expanding evolvable systems.

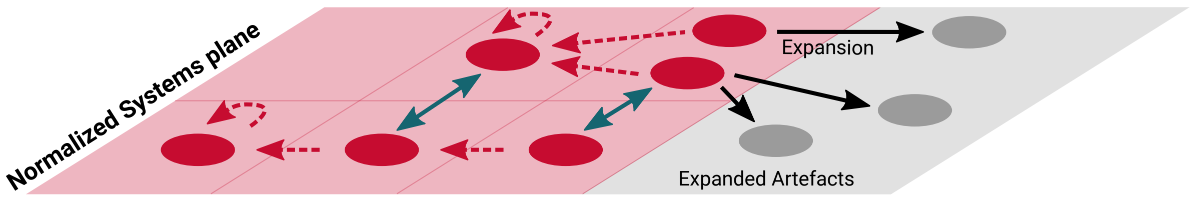

Figure 3.9: Normalized Systems plane with models and artefacts

We separate this plane as it has a different concern than the conceptual models plane. In terms of the primary purpose (FR1–FR3), the conceptual models plane contains inputs that are through the gateway plane transformed towards the NS plane. Then, it can be even reversed for checking the consistency (FR6).

3.2.4 Technical View

In the technical view, the artefacts introduced in the logical view are related to a transformation tool(s) for actually transforming conceptual models (as instances of the conceptual modelling language metamodels) to NS models. It describes the pipeline through which an input conceptual model is turned into an NS model (and then into an enterprise information system). It also considers the opposite direction of the transformation with respect to the consistency requirement.

Figure 3.10 illustrates the transformation pipeline based on the gateway ontology. First, the input conceptual model in a tool-specific file format is transformed into RDF based on the OWL ontology for the metamodel of the modelling language. The metamodel represented as OWL is mapped to NS using the constructs of the gateway ontology. The mapping together with the conceptual model in RDF are inputs for the transformation. It transforms the model based on the mappings and according to the rules of the gateway ontology and produces NS model representation in RDF. The final step is to transform the NS model from RDF to XML serialisation, which can be imported and used in NS tooling to further process or expand the information system. The transformation tool does not lie any obstacles in terms of the type of language or the use of the gateway ontology (FR1–FR3).

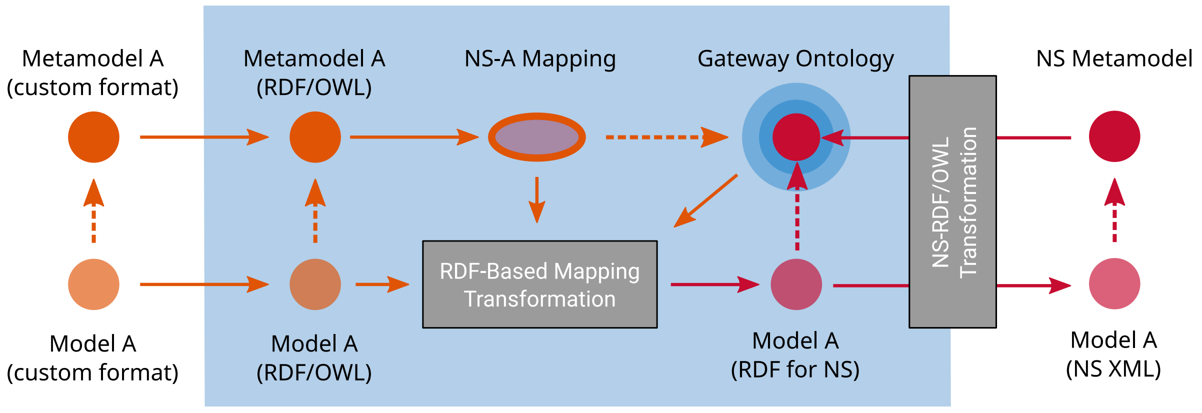

Figure 3.10: Technical view on the Gateway plane

The transformation procedure can easily handle semantic integration FR4 of input conceptual models. As shown in Figure 3.11, there may be more input (integrated) conceptual models. In that case, the transformation uses multiple corresponding metamodel mappings. The output NS model in RDF is then produced for the combined input. The semantic integration can be done traditionally in input RDFs, e.g. using `owl:sameAs` or simply by using the corresponding identifiers across models. The input models can be all together with the semantic integration specification defined in a single RDF file or multiple files – it is an insignificant implementation detail, the tool should support both ways.

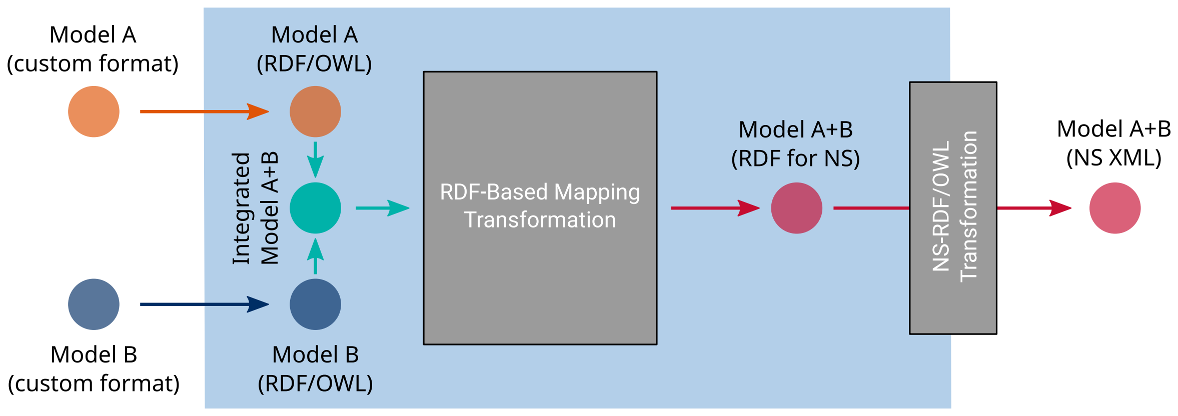

Figure 3.11: Transformation for semantically integrated models

The transformation can be reversed to verify consistency and transform the NS models into conceptual models with mapped metamodels. Figure 3.12 shows that the reverse transformation takes as input an NS model and conceptual modelling language mapping(s). First, the models are transformed into RDF representation. Then inverse transformation rules from the gateway ontology are used based on the mapping to produce conceptual models in RDF. To overcome the loss of information due to metamodels differences, additional information for backward re-construction of conceptual models must be encoded into so-called Options of NS Elements.

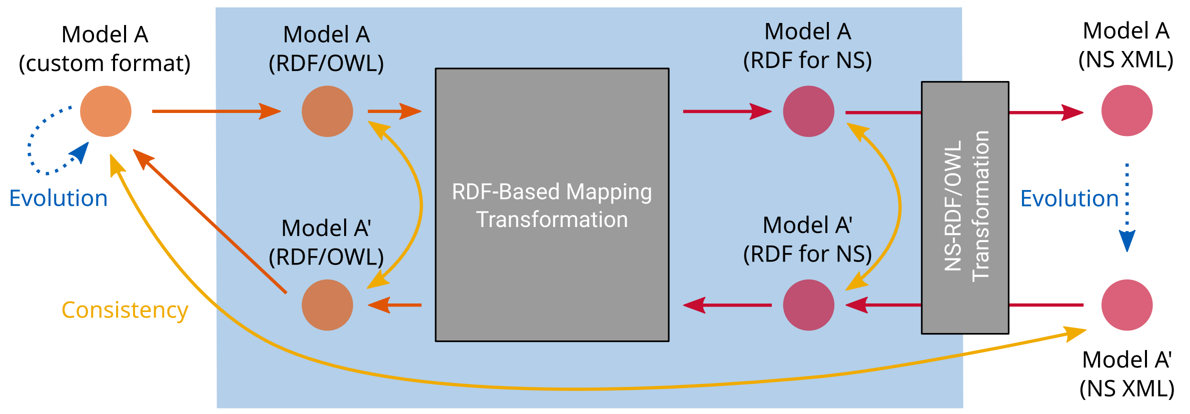

Figure 3.12: Reversed transformation for maintaining consistency

3.2.5 NS Gateway Ontology for Conceptual Models

The gateway ontology is the key part of our transformation between conceptual models and Normalized Systems. Its design is affected by related work (e.g. Model Driven Architecture (MDA) or OO-Method) and the requirements for extensibility (FR5, NR2) and evolvability (NR1). As explained, the gateway ontology covers several concerns; therefore, it is split into modules called layers. The primary purpose of the gateway ontology is to capture and provide knowledge for efficient transformation using RDF and OWL.

3.2.5.1 Core Layer

The core layer encapsulates the relationship with the NS metamodel. It consists only of the RDF/OWL representation of the NS metamodel. The version of this layer is affected only by the metamodel version, as well as the transformation procedure between NS and RDF/OWL. The constructs contained in the core always have a direct counterpart and can be transformed to NS XML representation. For example, for the Data Element there is `nsgo4cmCore:DataElement` for which the transformation is straightforward.

3.2.5.2 Extensions Layer

The extension layer provides additional constructs related to the NS metamodel. The primary objective is to avoid duplications NR4 of complex transformation patterns for constructs that appear often in conceptual modelling but are not supported by NS. An example of this may be inheritance, where the extension layer can define a new relation between data elements `nsgo4cmExt:specializes`. Then, for the transformation, this well-defined additional relation represents a certain pattern in terms of core layer constructs. Similarly, it would work also for the mentioned enumerations that can be realised using a taxonomy data element with name field.

It allows to propose constructs and patterns to make NS modelling "closer" to specific conceptual modelling languages for easier transformation without changes in the NS metamodel. However, after refinements and justification of the benefits, such constructs can be proposed for inclusion in the metamodel. When this occurs and a new version of the metamodel is released, it will become part of the core layer, and it can be linked directly from the extension layer.

3.2.5.3 Transformations Layer

Finally, the transformations layer defines the means for actual transformation specification, i.e. mapping of a conceptual modelling language metamodel to the NS metamodel (or constructs from the extensions layer). A mapping uses these definitions to describe the relations. For example, a mapping for UML Class Diagram could state that `uml:Class` is mapped to `nsgo4cmCore:DataElement`, a class name becomes a data element name, attributes of a class become fields of a data element, and so on.

All the knowledge about relations between metamodels must be captured using the vocabulary given by this layer, and as targets for the mapping are used the two underlying layers. Then, a transformation tool must be able to read and act according to it, for example, transform each class to a corresponding data element with its name and fields.

3.3 Formal Specification of Transformations

We use mathematical notation combining first-order logic, set theory, and algebra to formally describe our architecture, its components, transformations, inputs, outputs, and others. It provides essential formal grounding to our work and precisely expresses the design. We also considered other formal specification approaches mentioned in Chapter 2; however, those are software-oriented and unnecessarily complex for our case. This section defines our overall design using mathematical equations and serves as a foundation for the subsequent chapters, where introduced concepts and definitions are further elaborated.

3.3.1 Conceptual Models

The goal is to support any kind of (formal) conceptual models with a specified metamodel. Therefore, we must not make any assumptions about the constructs in such (meta)models. A conceptual model is a set of concepts M. The form of concept, i.e. its representation, is based on the metamodel. For example, if the metamodel is UML, then the concept c_i can be represented as a class with its properties and relations. However, with other metamodels, the representation of concepts may be completely different. Thus, a concept c_i is for us a tuple further described by the metamodel M'. A model (or metamodel) M describes a set of conforming instances. In case of a metamodel, the instances are models; for a metametamodel, those are metamodels, and so on.

$$M = \{c_i\} \in \mathfrak{inst}\,(M') \tag{3.1}$$

A concept c_i in a model M is expressed using a subset of concepts its metamodel M'. For instance, a concept can be expressed as a UML class with name, stereotype, and attributes.

$$c_i \in \mathfrak{inst}\,(C'_i \subset M') \tag{3.2}$$

Then, we use the term semantic integration with conceptual models to define relations between concepts of two (or more models). It can be understood as a union of the integrated models (conforming to union of metamodels):

$$M_A \cup M_B \in \mathfrak{inst}\,(M'_A \cup M'_B) \tag{3.3}$$

3.3.2 NS Models and Expansion

The notation used for expressing NS metamodel, models, elements, or expansion comes primarily from the Normalized Systems Theory (NST) [1] but also the fact that we can consider NS models as conceptual models.

$$M_{NS} = \{c_i\} \in \mathfrak{inst}\,(M'_{NS}) \tag{3.4}$$

The NS metamodel is a an instance of a metametamodel; however, as already described, metametamodel in NS is in fact a subset of the metamodel (meta-circularity). Therefore, if we support an function, or more specifically a transformation, supporting NS models, it will support also the NS metamodel (and metametamodel).

$$M'_{NS} \in \mathfrak{inst}\,(M''_{NS} \in M'_{NS}) \tag{3.5}$$

The expansion notation described in the NST [1] does not directly use NS Elements, but the requirements formulated through them. The NS model captures functional data requirements D (e.g. data elements, their fields, data projections, or states) and functional processing requirements P (e.g. task and flow elements, transitions, or triggers). There are also additional non-functional requirements captured technology settings T. Then, the expansion produces data classes S and action classes F based on the requirements:

$$\mathcal{I}(D_m, T_\alpha) = \{S_{m,k}\}_{k=1,\ldots,K} \cup \{F_{m,l}\}_{l=1,\ldots,L} \tag{3.6}$$

$$\mathcal{I}(P_n, T_\alpha) = \{S_{n,k}\}_{k=1,\ldots,K} \cup \{F_{n,l}\}_{l=1,\ldots,L} \tag{3.7}$$

We also do not want to make assumptions about specific constructs defined in the NS metamodel (M'_{NS}), such as data element and task elements. There are two reasons; first, the metamodel may evolve and the constructs may change which could require us to update our artefacts; second, new and alternative NS metamodels with different constructs can appear in the future. Furthermore, we can generalise the statements above related to the expansion process. D can be understood as structural specifications, P behavioural specification, and the resulting S and F can be unified as artefacts (without limitation to source code or classes).

$$\mathcal{I}(D_m \cup P_n, T_\alpha) = \{A_k\}_{k=1,\ldots,m+n} \tag{3.8}$$

3.3.3 RDF and OWL

RDF is based on triples t that consists of a subject s, predicate p, and object o. Eventually, quads q may be used when additional context c is necessary. When we talk about a dataset or graph, we mean a set of triples D (often called RDF dataset). A set of resources $\mathbb{R}$ encompasses all resources with URIs $\mathbb{R}_U$ but also anonymous, i.e. blank nodes $\mathbb{R}_B$. Then, we use $\mathbb{P}$ to capture a set of predicates and $\mathbb{L}$ a set of literals. To denote literals of a special type, e.g. $\mathbb{L}_{\text{string}} \subset \mathbb{L}$.

$$t = (s, p, o) \in D \subseteq \mathbb{R} \times \mathbb{P} \times (\mathbb{R} \cup \mathbb{L}) \tag{3.9}$$

$$q = (s, p, o, c) \in D' \subseteq \mathbb{R} \times \mathbb{P} \times (\mathbb{R} \cup \mathbb{L}) \times \mathbb{C} \tag{3.10}$$

A directed graph G_D can be constructed vertices V_D and edges E_D based on the set of triples (or quads) D as follows:

$$\begin{gathered} G_D = (V_D, E_D) \\ \forall (s_i, p_i, o_i) \in T : s_i \in V_D \land o_i \in V_D \land (s_i, o_i) \in E_D \end{gathered} \tag{3.11}$$

Although OWL ontology is captures also using a set of triples, it provides the concept of instantiation as conceptual models. Basically, it limits the allowed predicates p and objects o for individual s that is of type r. For example, it states that p cannot be used with s or that p may use only a certain subset of $\mathbb{L}$ as o. To express precisely OWL, it is essential to use the well-known functional syntax [154].

Both RDFS and OWL allow us to specify models and metamodels. Then, we may use the same notation as we did for conceptual and NS models. For example, an ontology for UML is de-facto a metamodel $M'_{UML-OWL}$ that allows to create instances – capture UML models $M_{UML-OWL}$ in RDF. OWL serves in this case as a metametamodel M''_{OWL}.

$$M_{UML-OWL} = \{c_i\} \in \mathfrak{inst}\left(M'_{UML-OWL}\right) \tag{3.12}$$

$$M'_{UML-OWL} \in \mathfrak{inst}\left(M''_{OWL}\right) \tag{3.13}$$

3.3.4 Transformations

A transformation in our scope is a mapping $T_{I \to O}$ between a representation of input knowledge using the metamodel M'_I and a presentation of output knowledge using the metamodel M'_O. For example, we use $T_{NS \to OWL}$ to denote a transformation mapping from the NS model to the OWL ontology.

$$T_{I \to O}\left(M_I \in \mathfrak{inst}\left(M'_I\right)\right) = M_O \in \mathfrak{inst}\left(M'_O\right) \tag{3.14}$$

A mapping specifies how concepts (more precisely instances of a construct from a metamodel) from an input model can be translated into concepts of an output model. For such a mapping, we expect that for a single input model it returns a single output model (and always the same one). Therefore, $T_{I \to O}$ is more specifically a function with domain M'_I and codomain M'_O. This expectation does not interfere with the possibility of transform a single concept $c_{I,i} \in M_I$ into multiple concepts $c_{O,j} \in M_O$, as $T_{I \to O}$ is the mapping of models, not its partial concepts. Due to possible information loss (caused by different levels of expressiveness or focus on different aspects), $T_{I \to O}$ may not be bijective. If $T_{I \to O}$ is bijective, we can find the inverse function $T^{-1}_{I \to O} = T_{I \leftarrow O}$ that allows bidirectional transformation without information loss. In other cases, we can define the opposite mapping $T_{O \to I}$ based on $T_{I \to O}$ to support bi-directionality but with information loss.

We can also observe a mapping of the underlying concepts of the metamodels. For example, that UML class is mapped to a NS data element. As a single concept $c_{I,i} \in M'_I$ from an input metamodel may be assigned to multiple concepts $c_{O,j} \in M'_O$ in an output metamodel, it cannot form a function. Nevertheless, we can define a function $T'_{I \to O}$ that does not map single concepts but sets of concepts (practically patterns of concepts). Such a function can be even bijective and can be used for avoiding information loss, e.g., by using a metamodel-specific concept to denote how it was formed from the input (in NS we can use the so-called options).

3.4 Evolution in Design Cycle

The architecture explained in the previous section results from several iterations of the DSR's design cycle. We used the bottom-up approach, where we first prototyped direct transformations as a proof-of-concept. Then, we focus on generalising and connecting the partial solutions. During prototyping, we identified and dealt with several obstacles related to the set requirements. This section briefly describes the key milestones or partial artefacts that contributed to the final architecture during the design cycles.

3.4.1 Initial Prototype OntoUML-NS

The very first work done was a prototype to transform OntoUML models directly to NS Elements. It revealed several issues, mainly with the loading and manipulating of conceptual models and related tooling. As the input for the transformation has been selected XMI from the Menthor Editor. Although XMI should promote interoperability, due to the custom and undocumented serialisation profile used by Menthor, it was interpreted as any other XML without any efficient querying over XMI. It was implemented in Python using standard libraries.

Another issue was to capture and serialise the export NS models from the tool. Custom data classes to represent entities of the NS metamodel, e.g. data element or field, were implemented together with XML serialisation. The transformation itself was part of the Python code. For example, all OntoUML classes were converted into NS data elements with stereotypes captured using options and attributes with relations as fields. Some of the constraints of OntoUML were lost during the transformation and the ontological meaning of the stereotypes. The most challenging was the transformation of inheritance, which is very common in OntoUML models and is based on a principle of identity.

There were several main lessons learnt from this prototype. First, it is necessary to have a common language for conceptual models to allow efficient extraction and semantic integration of knowledge. Second, instead of re-implementing utilities related to the NS metamodel, it can be re-used if the Java language is selected (it resulted in NR5). Then, the transformation cannot be captured directly in the source code but must be defined as input for the transformation. Finally, there may be several ways of transformation, and it may not be possible to decide which is the best one automatically. We identified this issue with inheritance and addressed it further in related work [A.3].

3.4.2 Ecore-NS Transformation

The problems encountered in the OntoUML-NS transformation led to another stand-alone transformation tool prototype – from Ecore (as UML subset) to NS [A.12]. We used Java which allowed to re-use existing libraries for working with NS models and XML (de-)serialising them efficiently directly in the code. For Ecore, we used Eclipse Modelling Framework also for model manipulation and XMI (de-)serialisation. It allowed us to purely focus on the implementation of the transformation without the need to bother with formats and model querying. That verified the advantages of using the same technologies as are using in NS tooling (NR5).

The transformation between Ecore and NS is more straightforward when compared to the previous one with OntoUML. Ecore metamodel is closer to NS because it is more low-level oriented and does not emphasise ontological clarity of concepts. Moreover, some of the rules are similar to or the same as in the OntoUML-NS transformation. For example, there is the same issue with inheritance and classes with attributes are also being transformed to data elements with fields.

However, there were some differences, for instance, enumerations or annotations. Enumerations are not (just as inheritance) allowed directly in NS as they are obstacles to evolvability. It has to be transformed into a taxonomy-type data element and a list of its instances. Annotations, on the other hand, can be seen as a direct counterpart to options – they are used to further describe an entity of the model in a generic way. We also experimented with backward transformation in order to check consistency.

Together with the OntoUML-NS, the Ecore-NS prototype showed that the re-use of transformation rules even between different conceptual modelling languages is desired. Then, there may be cases where transformation of a single construct may turn into a whole pattern, which contains also instance-level entities. Finally, the options in NS models can be used both as a direct counterpart to constructs from conceptual modelling languages and as a versatile way to keep additional information for consistency checking.

3.4.3 Adding Intermediary Plane

After the experience with the OntoUML-NS and Ecore-NS transformations, we started the modularisation of the transformation architecture by adding an intermediary plane. The primary motivation was to move the definition of transformation rules from a source code to a more flexible format, i.e. different execution and specification of a transformation. First, the separation is a step forward to achieve FR5. Then, the need for re-usability and DRYness (NR4) is also more straightforward within a transformation-oriented specification rather than in source code.

The architecture design has been split into several components or modules in a divide-and-conquer way. This intermediary plane with transformation definitions works with both conceptual and NS models using the exact mechanism and the same format – “common language”. Then there is separated concern for bi-directional transformation between the intermediary representation and NS models in its original representation (XML). Furthermore, the same principle is used for all conceptual modelling languages or events, a variant or format of a language; each has its module for transformation to the “common language”.

Then, we investigated what could be the “common language”. It must be flexible and versatile enough to cover any knowledge, have good support (especially in Java), and allow evolvability for future development. We tried several promising approaches that focus on model transformations; however, we evaluated RDF/OWL as the best way to move forward. The essential part of the intermediary plane started to be called a *gateway ontology* as it practically lets conceptual models enter the world of Normalized Systems but also knows the way back (FR6). The alternative approaches are described in Section 3.6.

3.4.4 NS-OWL Tool Development

With RDF/OWL decided as the core technology for the gateway ontology, the first key artefact to be built based on the bottom-up approach was the NS-OWL transformation. Initially, we designed it as a tool in Java that composes an OWL ontology from an NS (meta)model [A.7]. However, more information needed to be encoded for the backward transformation rather than just classes and properties (from fields). The first extension of the tool within its own design cycle was to utilise RDF to capture everything from an NS model based on the NS metamodel.

The second enhancement was to expand the transformation tool as NS application itself [A.10]. This promoted the evolvability of the solution, as it can be simply re-generated with any change in the NS metamodel. In our architecture, it is used to manage the core layer of the gateway ontology. However, the tool itself can be used to transform any NS model to RDF/OWL and vice versa. It has the potential to be used as an alternative way of serialisation in the future next to XML.

3.4.5 Conceptual Models in RDF and OWL

From the opposite direction, the first step was to convert conceptual models to the “common language” of the intermediate plane. Due to the wide use of RDF and OWL, there are several tools, proposed mappings, or algorithms to transform various conceptual models into RDF and OWL. Some of the approaches have already been mentioned in Chapter 2. We first focus on the transformations for UML and review them [A.8]. Unfortunately, the subset of usable solutions focused only on UML Class Diagrams; therefore, we had to design UML-in-RDF independently. The review provided a valuable knowledge base.

Supporting both UML Class Diagrams and UML Activity Diagrams contributed to FR1 and FR2. For OntoUML (also related to FR1), it can be captured in RDF using our solution for UML as well as using the gUFO. Moreover, both can be combined in a single RDF representing the conceptual model. That is an example of the versatility in which a single conceptual model can be represented in different ways as input for the transformation to NS.

As separate artefacts but designed and developed similarly are targetting BPMN and BORM (FR2), and ORM (FR3). Again, we took advantage of existing proposals for BPMN-to-OWL mappings. In the case of BORM, no previous solution was possible to use and the ontology for representing BORM models in RDF had to be developed without relying on any previous work [A.13]. ORM also had previous mappings that were re-used.

3.4.6 Generalising the Workflow

During the design and development of the individual artefacts (mainly the mappings of conceptual modelling languages and metamodels for transformation to and from NS) the documentation and generalisation of the workflow have been done. To support FR5 and fulfil NR6, the documentation of how to define the transformation rules using our gateway ontology is crucial. It was not done directly during the initial steps while designing the base mappings for the selected languages.

Instead, first, we developed the mappings into a state where it was sufficient to evaluate against requirements based on DSR. Then, we described the procedures taken in a generalised way, i.e. not oriented to a specific part of conceptual modelling (e.g. structural modelling). We created a new artefact that was then evaluated as a recipe for creating another mapping.

3.4.7 Enhancing Layers in Gateway Ontology

With all partial artefacts done in an integrable state, the improvements by design cycle iterations were made on the gateway ontology layers and conceptual modelling language mappings. For each conceptual modelling language, we minimised information loss (for FR6) by adding new constructs to the extension and transformation layers. Moreover, if we identified similar patterns in the mappings, we adjusted the transformation layer accordingly to eliminate complex duplications.

The evaluation of the design cycle at this stage showed how the output NS model from various conceptual models improved (lowering information loss) and how the mapping was simplified (complexity and size of the mapping). In the later phase, we also focused on semantic integration (FR4) – integrated conceptual models and again added suitable new constructs to the extension and transformation layers with the same motivation and evaluation as for individual languages.

3.4.8 Future of Design Cycle

The design cycle of artefacts that we created during this research may be iterated in the future. It is designed to evolve over time. Whenever there is a need to change the mapping or transformation of a particular conceptual modelling language, the corresponding artefact may iterate again, be improved, evaluated, and used in practice. The same goes for other artefacts – NS-OWL transformation, the gateway ontology and its layers, and the transformation tool.

The need for change may come from changes in a modelling language specification, NS metamodel specification, or the adoption of new modelling languages. We also expect enhancements based on long-term use in practice. As already mentioned, the transformation rules may differ based on a use case and the input models. The DSR allowed even further work to be done efficiently.

3.5 Aspects and Benefits

In this section, we undertake a concise overview and evaluation of the fundamental aspects encompassing the proposed design. By providing a summary, we aim to capture the key elements of our design. This allows for a comprehensive understanding of the design's core features and how they align with the identified requirements.

3.5.1 Future of Design Cycle

The architecture, as designed, demonstrates the application of Design Science Research methodology in addressing several of the requirements that were initially set. This achievement is attributed to the iterative nature of the design cycle, as previously explained. By incorporating feedback and insights gained from each iteration, the architecture has been refined to meet the evolving requirements. Additionally, the selection of appropriate technologies and the incorporation of nuanced improvements have contributed to the proposed architecture's ability to maximise the fulfilment of design-level requirements.

Future iterations of DSR hold great potential for advancing further the design in case new requirements or possibilities occur. As technology continues to evolve at a rapid pace, researchers and developers will have access to an expanding range of tools and techniques. These future iterations of DSR will may embrace emerging technologies such as artificial intelligence or machine learning. These technologies have the potential to transform the way artefacts are designed, developed, and evaluated, opening up new possibilities for problem-solving and creating innovative solutions. By continually refining and expanding its scope, future iterations of DSR have the potential to push the boundaries of our design to be up-to-date with the evolving target environment.

3.5.2 Models Integration Support

Due to having input conceptual models in RDF, the semantic integration of various conceptual models (FR4) is enabled in a standard way. RDF datasets representing multiple conceptual models (e.g. using UML Class Diagram and BORM) can be simply merged into a single dataset. Then, it allows to use standard means of semantic integration mentioned in Chapter 2, for example, predicates `owl:sameAs` or `owl:equivalentProperty`.

Preferably, integrated models should use common identifiers of individuals; for example, a single entity `model:Person` can be `uml:Class` and `borm:Participant`.

To fully satisfy FR4, the semantically integrated input models must be supported during the transformation process. For example, the transformation should treat equally the case with `model:Person` and `m1:Person owl:sameAs m2:Person` where `m1:Person` is `uml:Class` and `m2:Person` is `borm:Participant`. Similarly, for other mapping properties and other constructs. The means for semantic integration defined in OWL specification must be supported and documented.

3.5.3 Existing Tooling

Using the widely used technologies of RDF and OWL, the development, documentation, quality control, and adoption are significantly improved. RDF and OWL specifications provide solid grounding and interoperability with tools and existing works. The tooling support is essential, especially in terms of ontology development and RDF processing on a technical level. In terms of NR5 and NR7, there are enterprise-ready RDF frameworks in Java – Apache Jena and Eclipse RDF4J (formerly OpenRDF Sesame). There are also tools supporting NR6, i.e. for documenting ontologies.

Moreover, the tooling support will help us to evaluate the integral artefacts. First, there are best practices for ontology design and FAIRness with RDF. Second, there are also tools for verifying syntactical correctness and evaluating the quality of OWL ontologies. Both are crucial aspects related to the development of artefacts according to DSR.

3.5.4 Interoperability

RDF and OWL undoubtedly promote interoperability (NR3-I) and re-usability (NR3-R) of the artefacts. The gateway ontology, conceptual modelling language mappings, and conceptual (meta)models are being captured in an interoperable and standardised format. It allows one to take advantage of existing tooling and other previous work, as explained in the previous part; furthermore, it allows others to use the artefacts for different purposes.

Due to the modularity, it further simplified the re-use. For example, one may want to use only the core of the gateway ontology to perform an analysis on the NS metamodel. Similarly, one may re-use the definition of transformation rules for a different technical implementation. The other two FAIR requirements (NR3-F and NR3-A) cannot be directly addressed in the design but in the realisation and publishing phases – by providing metadata and licence, using persistent identifiers, and publishing. Finally, for interoperability (NR3-I), the standard conventions must be applied.

3.5.5 Extensibility

RDF and OWL are designed to promote extensibility as part of the Semantic Web framework and key technologies for Linked Data. It enabled the extensibility NR2 of both conceptual modelling language mappings and the gateway ontology. The version management is supported well in RDF and OWL too. Semantic versioning, changelogs, deprecations, and other standard means to keep track of changes and transparently explain compatibility can be used.

The implementation to execute the transformation must also support the extensions added to the mappings and the gateway ontology. As the transformation tools are designed to be (in accordance with NR5) itself an expanded NS application, it supports the extensions through craftings (custom code fragments). Alternatively, the expanders themselves can be extended using features or directly if necessary.

3.5.6 Evolvability and Maintainability

Having all the content kept in RDF and having the fine-grained modular structure as explained above can be considered evolvable. It contains loosely coupled stand-alone artefacts that can evolve independently. The number of variations to maintain is the sum and not the product. For the technical part, the evolvability in terms of NR2 is also considered in the design. The tool for NS-RDF/OWL transformation, which is mainly used to generate the core of the gateway ontology, is designed as an expanded NS application. Similarly, the tool to execute the transformation of conceptual models and NS models is also itself an NS application.

Hand-in-hand with evolvability goes maintainability. Not only that the burden of maintaining the product of all artefact versions is eliminated, but the RDF format significantly simplifies its handling. Standard tools for VCS (NR5) can be used together with CI pipelines that run various tools to check and enforce the quality and correctness of artefacts according to their specification. The same applies to custom expanders and related NS applications.

3.6 Alternative Approaches

During the design and implementation of the integral artefacts, we also pursued several alternative approaches. These approaches did not lead to the fulfilment of our goals or were simply superseded by a more suitable solution. However, they provide important experience and justification for our final design. In this section, we briefly explain the main alternative approaches that we pursued according to DSR. It is also clarified which requirements cause issues with a specific alternative approach.

3.6.1 Per-Language Transformations

One of the options for transforming conceptual models to (and from) Normalized Systems is approaching each modelling language individually. It is also the approach used for prototyping transformations with the bottom-up approach as explained concerning the DSR design cycle. The individual approach without any generalisations needed allows covering all nuances of the conceptual modelling language. This helps to promote consistency and avoid information loss during the transformation (FR6).

The evolvability NR1 and extensibility NR2 would be possible to achieve in the same way as it is for the NS-RDF/OWL transformation tool – implementing it as an NS application. However, to support n conceptual modelling languages of various kinds as desired by FR1–FR3, n tools (or components) would need to be implemented. That would create significant development and maintenance overhead, resulting in the need for generalisations.

Furthermore, the possibility of changing the transformation that would be encoded in the source codes and expanders cannot be considered as FAIR NR3 nor fulfilling the requirement FR5. Finally, for semantic integration of input models FR4, would not be possible to directly integrate the semantics of the input models, and mapping in some suitable language would need to be provided for transformation.

To overcome the disadvantages of this approach, the implemented components would need to be generalised, found a versatile and FAIR way of encoding the transformation rules, and the input models would need a representation that allows semantic integration. It manifested in our final architecture, where the generalisation is the transformation engine and RDF/OWL is selected for representing both transformations and models.

3.6.2 XSLT Transformations

The XSLT is often used for model transformations, as XML serialisation is widely used in conceptual modelling and computer-aided software engineering (CASE) tools. We investigated XSLT for our purpose and also during the review of UML-to-OWL transformations [A.8]. The output XML file is created by using the Extensible Stylesheet Language (XSL) template and filling it with matching data queries from the input XML file. It would be possible to use XSLT as the NS models are XML-serialisable. However, a direct transformation would have the same pitfalls as explained in the previous alternative solutions.

Furthermore, XSLT has its own limitations, and expressiveness is limited when compared to RDF-based solutions. XSLT has limited error handling capabilities, which makes troubleshooting and debugging challenging. Compared to other programming languages or technologies, the tooling ecosystem for XSLT is relatively limited. Moreover, it operates within its own ecosystem and requires a clear understanding of how it fits into the overall architecture. Integrating XSLT with other technologies or frameworks may require additional effort and potentially introduce compatibility issues.

In summary, a generic transformation between conceptual models would quickly become very complex in terms of template size. Such transformations would also result in evolvability issues (NR1) as it mixes syntactic representation and semantics. The re-use of transformations parts is also not possible in an efficient way which results in violation of NR4 or the need for additional mechanisms for composing XSLT programmatically. Our review [A.8] also shows the practical advantages of QVT over XSLT.

3.6.3 QVT Transformations

Query/View/Transformation (QVT) is one of the leading technologies for model transformations related to MOF and MDA. QVT requires the input and output models to conform to some MOF 2.0 metamodel. First, we would need to make NS metamodel compliant with MOF. Then, it would be possible to transform between NS and existing MOF compliant modelling languages (e.g. UML). That is a very limited subset compared to our requirements FR1–FR3. For other modelling languages, again, metamodels would need to be modelled in terms of MOF. Moreover, QVT does not directly address semantic integration (FR4) and consistency (FR6) and additional work would need to be done on building QVT queries or back-transformations.

While creating metamodels with MOF for various conceptual modelling languages and Normalized Systems Elements is feasible, it results in more limitations than with RDF and OWL. The essential advantage of using MOF would be the existing tooling to execute the transformations and support in some of the modelling languages (MOF compatibility and XMI serialisations). However, we identified the issues with versatility and maintainability (NR1 and NR4) of the transformations as too significant and chose RDF/OWL instead.

Although MOF is an industry-standard for modelling, there can still be variations and inconsistencies in its implementation across different tools and platforms. This lack of standardisation can lead to compatibility issues, making it challenging to seamlessly exchange models between different tools or environments. Similarly, QVT has a standardised syntax and semantics, which can limit its flexibility in certain scenarios. However, the transformation capabilities and expressiveness may not cover all possible transformation requirements or complex transformation patterns. This limitation would lead to custom workarounds increasing complexity and reducing interoperability between different QVT implementations.

An important aspect is that OWL itself is in agreement with MOF. Therefore, our solution can be used in the future together with QVT transformations. For example, QVT can be used to transform suitable conceptual models to RDF/OWL representation expected by our mappers and vice versa. Similarly, future work can investigate the possibilities of QVT transformations from NS models in OWL and the relations of MOF to the NS metamodel.

3.6.4 ATL Transformations

Another possible approach is to use ATL Transformation Language (ATL) in a similar way to QVT transformations. The ATL is related to Eclipse Modeling Framework (EMF) and Ecore. First, we would need to make NS metamodel compliant, e.g. define NS metamodel in terms of Ecore. Then, for all modelling languages that we would need to support, its metamodel needs to be specified through supported means (EMOF, Ecore, or KM3). In addition, transformation rules would need to be specified through ATL rules. The advantage would be a large library of existing transformations. Finally, the input model(s) could then be transformed from XMI to target XMI. Therefore, the requirements FR1–FR3 could be done; however, RDF turned to fulfil them in better quality.

The pitfalls of this approach are naturally similar to the issues with QVT, as ATL is highly affected by QVT. Although existing transformations are prepared and it is possible to define its own re-usable transformations, the language itself is not as versatile as our solution with RDF. For example, semantic integration of multiple models (FR4) can be done only at the level of transformation rules. Otherwise, it would need to be done using a new metamodel and then a single input XMI with an integrated model would be used.

Creating a bi-directional transformation for FR6 would be the same as doing two uni-directional transformations between XMIs. FR5 would be fulfilled as ATL allows us to maintain and enhance the transformation rules in a programming-like manner. More severe issues were identified concerning the non-functional requirements (especially with interoperability and evolvability). Another concern is the support for ATL as it does not seem to be widely used, the tools are not actively developed, and the documentation with examples is outdated.

Chapter **4**

Transformation between NS Elements and RDF/OWL

"An algorithm must be seen to be believed."

Donald E. Knuth

The transformation between Normalized Systems and the intermediary Gateway plane representation is a crucial part of our solution. First, it is used to generate the core of the gateway ontology, which is the central part and the target of all other transformations from conceptual models. Then, it allows transformed conceptual models into Normalized Systems (NS) on the intermediary plane to be made compatible with other NS tooling, for example, NS Expanders or NS Modeler, by transformation to the NS plane. Although the transformation is related directly to the Research Objective 1, the design must address RO3 and RO4 as well.

This chapter describes the transformation and its realisation together with the necessary background explained (for instance, the representation of NS models on the NS plane). It captures progress and improvements made during design cycles according to Design Science Research (DSR). The initial transformation design has already been published [A.7] and further improved as a selected paper for extended version [A.10]. The final artefact designed also benefits from our work on NS-Ecore transformation [A.12] and design of inheritance patterns [A.9].

4.1 Design of Bi-directional Transformation

The final design of our bi-directional transformation between Normalized Systems models and RDF/OWL is shown in Figure 4.1. It shows that this designed artefact consists of several sub-modules as it has to handle various concerns. First, it needs to work with source and target representations of models, that is, read and write NS models as well as Resource Description Framework (RDF) knowledge graphs. Then, there must be a well-designed mapping that defines the rules for transformation to minimise information loss and enable bi-directionality. Finally, we also identified the need to transform instances of a model.

The design of this artefact must allow its evolvable implementation. As explained, the goal is to provide a working example of the designed artefacts realisation. It must prove that the design is usable in practice and help evaluate it and enhance it during the design cycles.

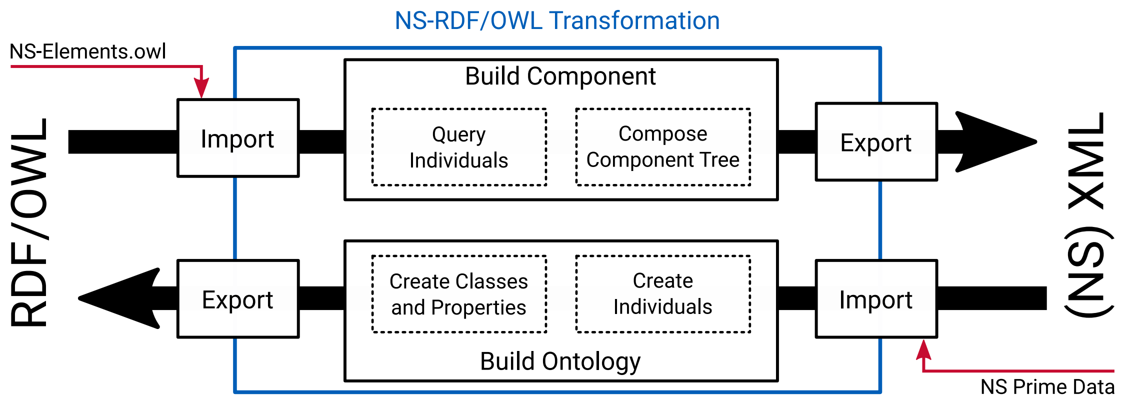

Figure 4.1: Design of NS-RDF/OWL transformation architecture

Using our formal notation (described in Chapter 3), this transformation is defined via the mapping function $T_{NS \rightarrow NS/RDF}$ and its inverse function $T^{-1}_{NS \rightarrow NS/RDF} = T_{NS \leftarrow NS/RDF}$. As we show later, RDF with NS ontology describing the NS metamodel (all constructs) allows us to capture all possible constructs in NS models in RDF. And vice versa, as we have all possible constructs captured in RDF (based on the NS metamodel ontology), we ensure that $T_{NS \rightarrow NS/RDF}$ is bijective. However, for $T_{NS \rightarrow OWL}$ it is not the case. Web Ontology Language (OWL) itself does not allow us to capture all constructs from NS. This is the reason why the NS metamodel ontology is needed. Still, adding triples using this to-OWL transformation is easily possible (and inverse mapping can be created as well) despite the significant information loss.

4.1.1 Motivation for NS in RDF/OWL

As explained in the previous chapter, RDF has been selected as the *common language* of the Gateway plane of our transformation design. The RDF is expected to be flexible and versatile, allowing the encoding of all the necessary information from the NS models, including the NS metamodel. Basically, RDF could be used just as a different serialisation of NS models, e.g. next to existing direct Extensible Markup Language (XML) serialisation.

NS (meta)models form a certain ontology that captures various concepts, properties, and relations. However, the way in which it captures the concepts is technology- or implementation-oriented. Rather than dealing with ontological aspects (such as identity, rigidity, or mereology), it focuses on constructs needed for implementation pragmatically (such as data entities and projections, validation of fields, or if a field should be visible in a list view).

However, some constructs could be captured using OWL or Resource Description Framework Schema (RDFS) to support relating RDF data to the ontology of the NS metamodel. With both RDF and OWL for NS (meta)models and the ability to transform models in both directions, the NS and the Gateway plane transformation can be solved. Based on our goals, the solution should be evolvable by design – quickly adapt to the NS (meta)model changes.

Normalised Systems also support custom metamodels due to the meta-circularity. One can easily define the own metamodel, generate tools and write own NS Expanders for the new metamodel. The design should not be limited to just the main NS metamodel, but should also support custom metamodels. That may also provide RDF support for other use cases in NS and promote the re-usability of our solution.

The metametamodel in NS is not explicitly stated, but is currently rather implicit. It is the meta-circular core of the NS Elements metamodel that allows also specifying other metamodels. For example, metamodels are defined using data elements and their fields, but not using task or flow elements. The subset of constructs from the NS Elements metamodel that can be used to define other metamodels can be called *implicit NS metametamodel.* On this level of abstraction, the implementation details are also omitted. Thus, it is potentially straightforward for transformation and mapping using ontologies.

4.1.2 Mapping NS to OWL

Before mapping NS models to RDF data sets, we need to design the ontology according to which the RDF data will be formed (i.e. defined classes and properties). The first step is to find the mapping between the NS metamodel and the OWL metamodel. It is impossible to use only RDFS as we strive for metamodelling techniques (such as punning) to capture even the homoiconicity of the NS metamodel. That will allow custom NS metamodels to be supported as long as they use the NS Elements model as the implicit metametamodel.

The equivalent concepts for the mapping of NS Elements and OWL metamodels can be found in Table 4.1. As a Component forms an Ontology, the fields of a Component, such as a name, description, or version, should become annotations of the transformed Ontology.

Table 4.1: Concept transformation from NS to OWL

NS Meta-model	**OWL Ontology (Functional Syntax)**
Component	`Ontology(...)`
Component metadata	`Ontology( Annotation(...) )`
Component Dependency	`Import(...)`
Data Element	`Declaration( Class(...) )`
Value Field	`Declaration( DataProperty(...) )`
Calculated Field	`Declaration( DataProperty(...) )`
Link Field	`Declaration( ObjectProperty(...) )`
Link Field (back)	`ObjectInverseOf(...)`
Value Field Type (instance)	`Declaration( Datatype(...) )`

This level of transformation allows instantiation of the model and interrelating with other ontologies by imports or using ontology matching techniques. The Data Element construct can be seen as a counterpart to Class, consequently for Value Field with Data Property and Link Field with Object Property.

For fields (data and object properties), other aspects can be captured. Inversed navigation could be modelled in NS using the opposite link field. For example, Customer has a link field *orders* to Order, and Order has the opposite link field *customer* back to Customer. In that case, it must not be mapped to two declarations of object properties, but just one and one inversion. For all fields, cardinality should be captured (e.g. `ObjectMaxCardinality` or `DataExactCardinality`) based on the type of link field or value field type – if its a list and if it is required). Finally, the range and domain of properties is used to capture source and target data element, or target value field type (for data properties).

When mapping value (and calculated) fields to `DataProperty`, the range must correspond to the related value field type. The Value Field Type is defined as a data element in the NS Elements metamodel and there are several pre-defined instance, for example, `String`, `Long`, or `Iban`. These instances can be mapped to datatype declarations in the resulting ontology (as described in instance-level mapping). However, to promote modelling as well as readability of the ontology and underlying RDF data, our design suggests using a basic mapping between pre-defined value field types and basic XML Schema Definition (XSD) data types (such as `xsd:string`) as shown in Table 4.2. Still, the exact value field type information remains captured in the RDF as the object property of the field.

Our design for mapping NS to OWL allows us to maintain the metamodeling and homoiconicity of the NS metamodel. When transforming a data element called Data Element from the NS Element metamodel, it is of its own type: `ClassAssertion( ns:DataElement ns:DataElement )`, i.e. the data element is an data element (and also an OWL Class). When creating individuals in RDF, they can be directly of type `ns:DataElement` as it is a special type of `owl:Class` (the punning technique is applied).

Table 4.2: Mapping between value field and XSD types

NS ValueFieldType	**XSD DataType**
Boolean	`xsd:boolean`
Date	`xsd:date`
DateLong	`xsd:dateTime`
DateTime	`xsd:dateTime`
Double	`xsd:double`
IntSpinner	`xsd:integer`
Integer	`xsd:integer`
LongSpinner	`xsd:long`
Long	`xsd:long`
Short	`xsd:short`
Time	`xsd:time`
byte[]	`xsd:base64Binary`
storage type = string	`xsd:string`
other	`xsd:base64Binary`

4.1.3 Mapping NS to RDF

Our designed mapping of NS to OWL can be directly used for a transformation of NS (meta)models to OWL ontologies and vice versa. However, there are many unmapped constructs of NS (and OWL), such as Task Element, Data Option, or Data Projection. That would cause a significant loss of information. To capture all parts of NS (meta)models our design uses RDF instead of limiting to OWL constructs. By *every part*, we mean each instance for which there exists a class in the OWL of NS metamodel as already explained. The mapping is related to the NS-OWL mapping and can be summarised in three steps:

- Instances of Data Element (in the NS Elements metamodel) become individuals with appropriate type. For example:
  - `ClassAssertion( ns:DataElement model:Customer )`,
  - `ClassAssertion( ns:TaskElement model:FinalizeOrderTask )`,
  - `ClassAssertion( ns:ValueField model:Customer-firstName )`.
- For instances of Data Element (in the NS Elements metamodel), all value field values are mapped using data property. For example, the name of value field of Customer:
  - `DataPropertyAssertion( ns:ValueField-name model:Customer-firstName "firstName" )`.
- For instances of Data Element (in the NS Elements metamodel), all link field values are mapped using object property. For example, the link between Customer and its

value field:
`ObjectPropertyAssertion( ns:DataElement-field model:Customer model:Customer-firstName ).`

With this approach, it ultimately allows us to encode any NS model (including the NS Elements metamodel). Although it is based on the NS-OWL mapping (Table 4.1) that considers only the key structural constructs and does not map others, for example, `TaskElement`, everything can be captured in the underlying RDF data due to the homoiconicity of the NS Elements metamodel. For example, `TaskElement` is in the metamodel represented as `DataElement`, therefore it is possible to capture it in RDF together with all its properties (fields from NS metamodel).

4.1.4 Recovering NS from RDF

To support bi-directionality of the transformation, it is also required to map RDF back to NS models. The only consideration is to read RDF using the same (or compatible) metamodel as it was initially transformed or created. Here we use the term *NS ontology* for the NS Elements metamodel captured in OWL using our mapping. Then, the mapping is a direct inversion of the three steps described in the previous section:

- Each resource of type from the NS ontology in the RDF graph is the corresponding instance of the NS metamodel in an NS model. For example, `Customer` becomes an instance of `DataElement`.
- All data properties from the NS ontology related to the resource are used to retrieve field values of the corresponding instance in an NS model. For example, the value field name of `firstName` related to the data element `Customer` is retrieved as `"firstName"` (string value from an RDF literal).
- All object properties from the NS ontology related to the resource are used to retrieve links between instances in an NS model. For example, for the data element `Customer` its field `firstName` is retrieved and linked.

The RDF graph can contain additional data (resources and properties) as we describe the mapping from those related to the NS ontology. Such additional assertions can be skipped as there is no added value in keeping them as part of an NS model. For future use, we designed an optional mapping of additional assertions using the so-called Options in NS models. For example, if there is another assertion related to a data element, it can be stored in the data option.

4.1.5 Instance-Level Mapping

The modelling in NS does not allow the enumeration construct for evolvability reasons. On the other hand, there are taxonomy-typed data elements that usually have two fields

– name and value. Together with its instances, it can be seen as an evolvable enumeration (one can add new fields, relate them with other elements, and so on). The instances are defacto also part of a model. For example, if one has a taxonomy type Gender, its instances with names Male and Female should always be present. Similarly, the metamodel has data element types – Primary, Taxonomy, History, and others.

The instances must also be assigned to RDF/OWL to allow them to be relating with other constructs:

- Each instance related to the model (especially instances of taxonomy data elements) must be turned into individual with corresponding properties:
  `ClassAssertion( model:Gender model:female )`
  `DataPropertyAssertion( model:female model:Gender-name "female" )`

A special case appears for the taxonomy data element Value Field Type of the NS metamodel. It has its own instances that should follow the above mapping rule and the one from NS-OWL mapping. The instances created by the rule above are used to match the exact type in NS where it degrades during the NS-OWL mapping for a range of data properties. For example, `String`, `MultilineString`, and `Iban` are mapped to `xsd:string` but by relating the field to the specific value field type, it is possible to recover the information when transforming from RDF to NS.

4.2 NS Elements Models Representation

After designing the mapping for the NS-RDF/OWL transformation, the inputs and outputs must be handled. We need to deal with multiple representations of NS Elements models: XML serialisation, objects and classes, RDF triples, and OWL triples. This section describes the vital aspects of each representation and the related NS concept of projections.

4.2.1 NS Projections

Normalized Systems use the concept of so-called *Projections*. An NS Element is always expanded into several projections. For example, for a data element *Person*, there is no direct class `Person`. However, there are several projections such as `PersonTree`, `PersonComposite`, `PersonBean`, and others. Each projection targets a different concern and provides its own functionality (has its purpose). For example, objects of Tree projections form a tree (as visualised in Figure 4.2, children objects are linked, but links to parent are just to a so-called `DataRef` instead of the object), but Composite projections have all the links in all directions resolved.

Such projections are also expanded from the NS metamodel. Some of them form re-usable Java libraries and provide additional functionality. One of the key libraries is `elements-ioxml`. It has utilities for XML import and export of NS models via the Tree projection. In terms of NS, both XML and Tree representations are data projections, i.e.

different views on an NS model. Tree projections are designed to be Plain Old Java Objects (POJOs) – therefore, it does not bound any special restrictions.

The transformation between NS and OWL is designed as a transformation between Tree and RDF data projections. The RDF data projection can capture all the information from Tree projection according to our mapping. The unique property of the RDF projections is its interoperability. Moreover, it allows re-using the existing library for working with XML-serialized NS models.

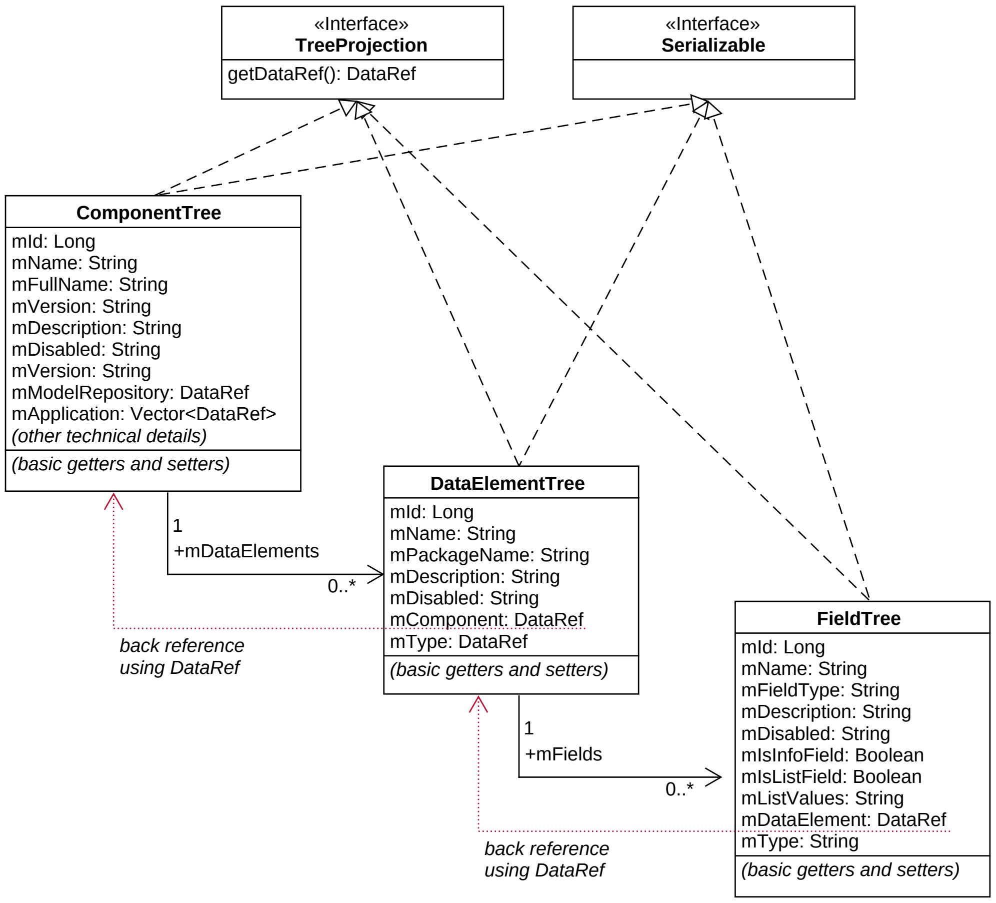

Figure 4.2: Excerpt of NS Elements tree projections class diagram

4.2.2 NS XML Representation

Normalized Systems use XML serialisation of the models and various configurations. The schemas used for serialisation reflect directly the NS metamodel. For example, there is a `<dataElement>` tag for a data element as shown in Listing 4.1. As such, the XML representation of the NS metamodel is possible to read and write using standard XML libraries. However, for Java, there is the mentioned `elements-ioxml` library which is already evolvable and guarantees compatibility with the specific version of the NS metamodel (through Tree projections).

Listing 4.1: Data element XML projection

```xml
<dataElement name="Person">
  <packageName>org.normalizedsystems.example</packageName>
  <description/>
  <dataElementType name="Primary"/>
  <fields>
    <field name="firstName">
      <fieldType>VALUE_FIELD</fieldType>
      <description/>
      <isInfoField value="true"/>
      <isListField value="false"/>
      <valueField name="ValueField:Person_firstName">
        <valueFieldType component="" name="String"/>
      </valueField>
      <fieldOptions/>
    </field>
    <!-- ... other fields -->
    <field name="gender">
      <fieldType>LINK_FIELD</fieldType>
      <description/>
      <isInfoField value="false"/>
      <isListField value="false"/>
      <linkField name="LinkField:Person_gender">
        <linkFieldType name="Ln01"/>
        <targetPackage>org.normalizedsystems.example</targetPackage>
        <targetClass>Gender</targetClass>
      </linkField>
      <fieldOptions/>
    </field>
  </fields>
  <!-- ... finders, dataCommands, dataProjections, dataOptions etc. -->
</dataElement>
```

Although for our artefact, the essential part is the transformation between the RDF and Tree projections, the XML projections are suitable for testing and as input or output. For the NS-to-RDF transformation, the input can be both Tree projections of an NS model (using the API) or XML projections of the same model loaded from the file system. Vice versa for the NS-to-RDF transformation, Tree and XML projections are the output.

The example in Listing 4.1 shows a serialisation of a single data element in XML. It contains all the information related to the data element, from basic attributes, such as name, description, or type, to contained children of different types. For example, fields of a data element are also part of its XML serialisation. Each data element subtree is stored in its own XML node/file. A component serialised into a folder then consists of a single XML file containing information about the component, and then in subfolders are different types of elements (data, task, flow, etc.).

4.2.3 RDF-Triples Representation

The RDF projection uses triples to capture all information from Tree projections, i.e. serialisation of an entire NS model. Content-wise, these two projections are equal. It is essential for the bi-directionality of the transformation without information loss. Although the Tree projections form a tree, RDF projections may form a graph. There is no need to avoid using backlinks in RDF when using Uniform Resource Identifiers (URIs). The resource can be resolved using its identifier only when necessary.

There is an issue related to the RDF representation for the NS metamodel. To create triples for the NS metamodel, it must refer to the OWL triples of itself (e.g. a data element individual, which is also a class, is an instance of class Data Element). Therefore, it is required to have a vocabulary of the NS metamodel elements as URI before the generation of triples. Then, RDF provides several formats to serialise triples into a text.

Figure 4.3 shows a part of RDF graph with a data element `Person` that has a value field `lastName`. The `example` prefix (or namespace) contains the NS model, while the `ns` prefix is used for the NS metamodel. A more complete example in RDF Turtle format is shown in Listing 4.2 which corresponds to the previously presented Listing 4.1. Similarly, Listing 4.3 shows RDF for the value field `lastName`.

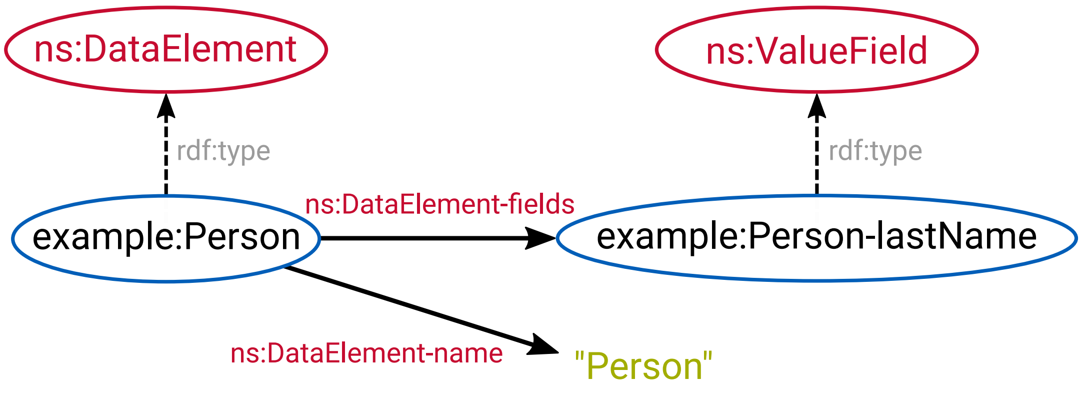

Figure 4.3: Example of RDF graph for NS model

The result of using $T_{NS \to NS/RDF}$ for an input NS model is an RDF dataset, a set of triples, where each (atomic) input concept is represented by a single triple. From the example Listing 4.2, a data element is represented by the triple introducing the resource with a type, all the other properties of that resource are understood as related atomic concepts, e.g. that it has some name, fields, or data options (fields and data options are

Listing 4.2: Data element RDF projection

```
@prefix example: <https://purl.org/nsgo4cm/example#> .
@prefix ns: <https://owl.stars-end.org/test/elements#> .
@prefix nsx: <https://owl.stars-end.org/test/nsx#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

example:Person
  a ns:DataElement , owl:Class ;
  rdfs:subClassOf              owl:Thing ;
  ns:DataElement-component     example:example ;
  ns:DataElement-name          "Person" ;
  ns:DataElement-packageName   "org.normalizedsystems.example" ;
  ns:DataElement-type          ns:Primary ;
  ns:DataElement-dataOptions   example:Person-hasDisplayName ;
  ns:DataElement-fields
    example:Person-firstName ,
    example:Person-gender ,
    example:Person-dateOfBirth ,
    example:Person-lastName ;
  ns:DataElement-finders
    example:Person-findAllPersons ,
    example:Person-findByNameEq ,
    example:Person-findByGenderEq ;
  nsx:dataRef "example::Person" .
```

themselves concepts represented by resources with own properties). When a complete NS metamodel ontology supports all the constructs, the transformation maps every concept from an NS model to RDF triples based on the NS metamodel ontology:

$$\forall c_{NS,i} \in M_{NS} \left(\exists c_{NS/RDF,j} \left(c_{NS/RDF,j} \in M_{NS/RDF} = T_{NS \rightarrow NS/RDF} \left(c_{NS,i} \right) \right) \right) \tag{4.1}$$

As already stated before, this enables bijection and thus bi-directionality:

$$\forall c_{NS,i} \in M_{NS} \left(c_{NS,i} = T_{NS \leftarrow NS/RDF} \left(T_{NS \rightarrow NS/RDF} \left(c_{NS,i} \right) \right) \right) \tag{4.2}$$

4.2.4 OWL-Triples Representation

In addition to RDF triples, we also add OWL triples (as a particular type of RDF triples). As explained, using only OWL for the transformation would suffer enormous information loss if used without additional RDF triples. According to the mapping, these triples use a predicate or object a construct from OWL or RDFS, or additional metadata about the construct. The ontology part of the RDF projection has its own purpose, serving as a description for its individuals. When using the NS metamodel, it creates the NS Elements ontology for the underlying RDF triples.

Figure 4.4 shows the corresponding parts of RDF representation like Figure 4.3; however, it captures how the same constructs (data element `Person` and value field `lastName`) are related to OWL. The mapping can also be observed in examples Listing 4.2 and Listing 4.3. In this case, the use of `xsd:string` as `rdfs:range` for the value field is decided based on the mapping in Table 4.2.

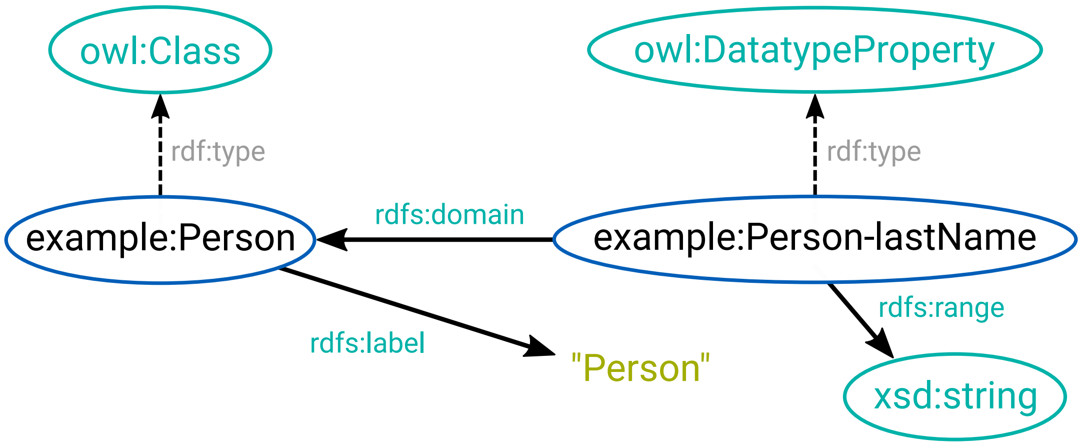

Figure 4.4: Example of OWL graph for NS model

4.3 Transformation Execution

With the defined mapping for the bi-directional transformation between NS and RDF, and handled input and output representation, the central part that actually executes the transformation rules based on the mapping remains. It defines how to turn an input model (RDF or NS) to a corresponding output model (RDF or NS) using the mapping. The described execution may have various practical implementations, as discussed in the next sections.

4.3.1 Building URIs

When creating RDF graphs (and OWL), the nodes that are resources need to have URI assigned. That is, of course, different from the models in NS where there are other means for the identification of the elements and their parts. As visible from Listing 4.1, there is no direct unique identifier of every construct in a component. The constructs are identified by their names, which are unique in a specific scope inside a component. For example, there cannot be two data elements of the same name within a single component or two

Listing 4.3: Value field RDF projection

```
@prefix example: <https://purl.org/nsgo4cm/example#> .
@prefix ns: <https://owl.stars-end.org/nsgo4cm/elements#> .
@prefix nsx: <https://owl.stars-end.org/nsgo4cm/nsx#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ns-example:Person-lastName
  a  ns:Field , ns:ValueField , owl:DatatypeProperty ;
  rdfs:domain           example:Person ;
  rdfs:range            xsd:string ;
  ns:Field-dataElement  example:Person ;
  ns:Field-fieldType    "VALUE_FIELD" ;
  ns:Field-isInfoField  true ;
  ns:Field-isListField  false ;
  ns:Field-name         "lastName" ;
  ns:Field-valueField   example:Person-lastName ;
  nsx:dataRef           "example::Person::lastName" .
```

fields with the same name in a single data element. The use of blank nodes would prevent us from uniform processing of constructs and cross-referencing. Therefore, URI must be composed using the names and context.

The schema for building URIs is explained in Figure 4.5. First, URI of a construct starts with a prefix that can be specified using the component options or entered as a transformation parameter. Then the name of the component is appended, which is itself always URI-safe string (cannot contain illegal characters). We designed the component name to be part of the URL path instead of the fragment because we form a single ontology per component and then it may use its own RDF prefix. Finally, the fragment (after the # symbol) contains the context and name of the construct composed by using the so-called *DataRef* (as shown also in Listing 4.3).

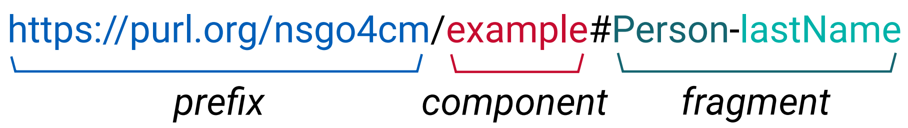

Figure 4.5: URI scheme for NS-RDF transformation

The composition of the construct URIs is only important for the NS to RDF/OWL part of the transformation. For the opposite direction, URIs are just used from the input RDF data to query the graph and compose the corresponding NS model – the schema of

URIs is irrelevant for this part. The use of names and context turned out to be the best possible solution with the current NS metamodel during our design cycle when we tried several other approaches, for instance, generating random identifiers or storing identifiers in component, data, field, and other options.

In an ideal case, URIs should use arbitrary identifiers for the NS constructs (such as UUIDs) that every construct in NS has assigned. The use of names and contexts in URIs causes issues when a construct is being renamed. For example, if we rename *Customer* to *Client* and the semantics of the construct stays the same, its URI should preferably also remain unchanged. That is not currently achievable unless we limit the transformation to enhanced models with pre-generated identifiers, which is too limiting concerning our requirements and objectives. Therefore, we propose such an extension to the NS metamodel as a potential future enhancement.

4.3.2 NS to RDF

With the ability to create a URI for every construct in the NS models, the next step is to provide knowledge about every such resource in the form of constructing triples. As shown in Figure 4.3, the types (e.g. `ns:DataElement`) and corresponding predicates (e.g. `ns:DataElement-name`) used to capture information about a resource are based on the NS metamodel. Thus, in order to transform an NS model to RDF, all its parts must be traversed and, depending on the type, triples must be added. For each type of constructs (represented as a data element in the NS metamodel), the predicates will differ depending on the value and the link fields in the NS metamodel.

The traversal must “visit” each construct in the input NS model exactly once; however, the order is irrelevant (by the nature of RDF triples). The procedure of a single visit is described by Algorithm 4.1. Despite the possibility of arbitrary order, we present the construction in logical order – first introduce a resource with a type and then provide additional statements based on value and link fields (respectively, datatype and object properties). Finally, child entities are traversed recursively. The algorithm should start at the level of a single component. Then, it will traverse different types of elements; for data elements, it will traverse all fields, and so on (as a flooding algorithm).

Algorithm 4.1 is finite as there cannot appear a loop in constructs provided by the *getChildren* function (which would cause infinite recursion over the `visit` procedure) and since all functions and procedures called are finite. Both the memory and time complexity are bound to the size of the input model. Because each construct will be visited exactly once (ensured by the *getChildren* function based on the NS metamodel), the complexity is $\mathcal{O}(N)$ where N is the size of the input model (counting all constructs defined using the NS metamodel). There is no ambiguity, and the algorithm is deterministic.

The function `buildURI` is described in the previous subsection; for a given instance of an NS metamodel construct, it returns a corresponding URI. The insert procedure adds a new triple (in square brackets) to a given RDF model. Finally, we use several “get” predicates that are trivial (usually extraction of attributes and potentially a simple transformation). As the described algorithm is generic (i.e. for any type of construct based on the NS

Algorithm 4.1 NS-RDF transformation traversal

```
 1: procedure VISIT⟨$T$⟩(RDF model $M$, construct $c$)
 2:     $r \leftarrow$ BUILDURI($c$)
 3:     $t \leftarrow$ BUILDURI($T$)
 4:     INSERT($M$, [$r$, rdf:type, $t$])
 5:     for $f \in getValueFields(T)$ do
 6:         $p \leftarrow$ BUILDURI($f$)
 7:         INSERT($M$, [$r, p, getLiteral(c, f)$])
 8:     end for
 9:     for $f \in getLinkFields(T)$ do
10:         $t \leftarrow$ BUILDURI($getTargetType(f)$)
11:         $p \leftarrow$ BUILDURI($f$)
12:         if $isCollection(f)$ then
13:             for $e \in getItems(c, f)$ do
14:                 $o \leftarrow$ BUILDURI($e$)
15:                 INSERT($M$, [$r, p, o$])
16:             end for
17:         else
18:             $e \leftarrow getItem(c, f)$
19:             $o \leftarrow$ BUILDURI($e$)
20:             INSERT($M$, [$r, p, o$])
21:         end if
22:     end for
23:     APPENDIX⟨$T$⟩($M$, $c$)                    ▷ additional steps (e.g. OWL)
24:     for $x \in getChildren(c)$ do
25:         VISIT($M$, $x$)
26:     end for
27: end procedure
```

metamodel), it is expected that implementation of this algorithm will use polymorphism, and the `visit` procedure will be adjusted to a specific type (e.g. for data elements, for value fields, and others).

By applying the `visit` procedure on a whole component, all information will be transformed into RDF – making it a serialisation equivalent to the XML serialisation in terms of stored data. It is ensured by traversing according to the already mentioned (implicit) NS metametamodel. Naturally, the procedure can also be used on the NS metamodel itself (metacircularity).

4.3.3 NS to OWL

In addition to the previously described NS-to-RDF transformation, the mappings with OWL described in Table 4.1 (together with the value types in Table 4.2) should be used for the transformation. We design this as an optional extra step where for some constructs of the NS (meta)metamodel OWL triples are added to the resulting RDF model (or graph).

Algorithm 4.2 shows the additional "visit" step for a data element. First, it states that the resource (identified by URI of the data element) is OWL class with a standard label. Then, based on its value, calculated, and link fields are created corresponding properties related to the class. In all of these properties, as the domain is used again, the URI of the data element.

Algorithm 4.2 NS-RDF additional OWL steps for data element

```
1: procedure createClassDE(RDF model $M$, Data Element $de$)
2:     $r \leftarrow$ buildURI($de$)
3:     insert($M$, [$r$, rdf:type, owl:Class])
4:     insert($M$, [$r$, rdfs:label, $getNameLiteral(de)$])
5:     for $f \in getValueFields(de)$ do
6:         createPropertyVF($M$, $de$, $f$)                ▷ described below
7:     end for
8:     for $f \in getCalculatedFields(de)$ do
9:         createPropertyCF($M$, $de$, $f$)                ▷ similar to VF
10:    end for
11:    for $f \in getLinkFields(de)$ do
12:        createPropertyLF($M$, $de$, $f$)                ▷ similar to VF
13:    end for
14: end procedure
```

Algorithm 4.2 is finite and deterministic as there is always a finite number of all types of fields and as all called functions and procedures are finite. Furthermore, the total complexity is $\mathcal{O}(n)$ where n is the number of fields of the data element. Procedures created for other constructs have the same properties except complexity, for example Algorithm 4.3 has complexity $\mathcal{O}(1)$ (all called procedures and functions also have $\mathcal{O}(1)$).

As an example, we provide Algorithm 4.3 that shows how straightforward it is to transform fields into the OWL representation. For the value and calculated fields, the first step is to recognise the corresponding type based on Table 4.2. Then, a datatype property is created with label, domain, and range. Additional statements can be added in the same way, for instance, to support multiplicity or other details related to properties. For link fields, the target data element URI must be built instead of resolving value type. Moreover, for link fields, the algorithm also tries to look up the opposite link field (forming a bi-directional relationship between two data elements) in order to capture it using `owl:inverseOf`.

A similar algorithm (in principle) is used for the components. However, there it is significantly more straightforward as it implements only the first three rows of the mapping in Table 4.1 (ontology definition, metadata, and imports). The essential part of that mapping is the identification of the component version that becomes a version of the ontology.

Algorithm 4.3 NS-RDF additional OWL steps for value field

```
1: procedure CREATEPROPERTYVF(RDF model $M$, Data Element $de$, Value Field $f$)
2:     $d \leftarrow$ BUILDURI($de$)
3:     $r \leftarrow$ BUILDURI($f$)
4:     $t \leftarrow$ VALUETYPE($getValueFieldType(f)$)                ▷ type mapping
5:     INSERT($M$, [$r$, rdf:type, owl:DatatypeProperty])
6:     INSERT($M$, [$r$, rdfs:label, $getNameLiteral(f)$])
7:     INSERT($M$, [$r$, rdfs:domain, $d$])
8:     INSERT($M$, [$r$, rdfs:range, $t$])
9: end procedure
```

That can be further used to maintain consistency between different representations of the (semantically) same models.

The separation from NS-to-RDF further supports evolvability as it is a manifestation of the separations of concerns principle. Whenever the mapping to OWL constructs is extended (e.g. due to new version of OWL or changes in the NS metamodel), the change impact is limited to the concern and construct-related procedure. Moreover, the procedure can be used independently on the NS-to-RDF transformation if only OWL is the desired output (and not a full RDF representation of an NS model).

4.3.4 RDF to NS

The reverse transformation queries the input RDF model and constructs components with their parts from the retrieved data. It still traverses the component tree recursively. The only difference is that it must use URIs of the constructs and query the details of the provided RDF model instead of direct extraction from the NS model with known and stable structure. The structure in the RDF model is given only by the NS metamodel ontology; however, that is not enforced. By default, the algorithm tries to extract as much as possible based on the ontology. Then, an optional validation step may check whether some information is missing.

Algorithm 4.4 shows the steps in which a construct i of type T is queried and constructed using URI r. Instead of a procedure, we use a function in this case as it must return the newly (re-)constructed construct to be added in its parent. First, it initialises the construct instance (in object-oriented programming, create an object). Then, it sets attribute values from extracted value field literals (e.g. that a data element has name *Person*). Then, relations to other constructs are queried and added to the construct instance. Here, the recursion occurs to traverse and prepare the child constructs.

The complexity of Algorithm 4.4 depends highly on the `query` function (used from the RDF library) which is expected to be $\mathcal{O}(\log N)$. Then, it works recursively in a similar way to Algorithm 4.1 – if there are no loops, the algorithm is finite and the complexity is $\mathcal{O}(N \log N)$.

Algorithm 4.4 RDF-NS backwards transformation traversal

```
1: function VISIT⟨$T$⟩(RDF model $M$, URI $r$)
2:     $i \leftarrow$ CREATENEW($T$)
3:     for $f \in getValueFields(T)$ do
4:         $p \leftarrow$ BUILDURI($f$)
5:         $v \leftarrow$ QUERY($M$, $[r, p, \_]$)
6:         $setValue(i, f, v)$
7:     end for
8:     for $f \in getLinkFields(T)$ do
9:         $p \leftarrow$ BUILDURI($f$)
10:        if $isCollection(f)$ then
11:            $objects \leftarrow$ QUERY($M$, $[r, p, \_]$)
12:            for $o \in objects$ do
13:                $c =$ VISIT($M$, $o$)
14:                $addRelated(i, f, c)$
15:            end for
16:        else
17:            $o \leftarrow$ QUERY($M$, $[r, p, \_]$)
18:            $c \leftarrow$ VISIT($M$, $o$)
19:            $setRelated(i, f, c)$
20:        end if
21:    end for
22:    VALIDATE($i$)                              ▷ optional validation procedure
23:    return $i$
24: end function
```

To initiate the algorithm, all component URIs must be first queried from the input RDF model and then the `visit` function is called on each of them. The complete component tree is then returned as a result. The optional validation procedure is expected to report potential errors or warnings in any way (based on implementation, e.g. log messages or raising exceptions). The final component tree can be then serialised to XML when needed.

4.3.5 OWL to NS

The transformation from OWL to NS is not covered by-design by this partial artefact. Still, it is crucial to mention here that such a transformation is by-design supported by the overall solution. As explained above, the RDF to NS only queries the individuals based on the NS metamodel OWL ontology. Thus, it ignores the additional OWL triples added during the NS to OWL procedure.

Indeed, some knowledge could also be extracted from that, and at least partial NS model could be composed (e.g. using classes, datatype, and object properties). Although that is not within the scope of our designed NS-RDF/OWL bi-directional transformation, it is within the scope of the overall gateway ontology solution. It will be possible to define a mapping between OWL and NS where OWL will be used just as a metamodel of a modelling language. Then, the underlying OWL ontology can be transformed just like any other conceptual model.

This relation also shows the closeness of ontologies and conceptual models as already mentioned in Chapter 2 (our knowledge base). Moreover, since OWL offers only very generic constructs for expressing ontologies (or models), it is just as well possible to use any upper ontology in addition, for instance, the already presented gUFO [133]. Again, a mapping between gUFO and NS would be needed; then the transformation on RDF level would be executed and the NS model would be produced (and vice versa).

4.4 Transformation Tool Implementation

The transformation design, together with its mapping, is applicable for manual transformation, i.e. writing RDF while reading the NS model. However, with the size of the NS models and the expected frequency of transformation execution, a tool to implement the transformation is crucial. It also eliminates potential human-made errors and inconsistencies.

The implementation also evolves over time, since it was first developed as a proof of concept [A.7]. Then, we make the tool evolve by incorporating the NS expansion [A.10]. However, the design cycle for this artefact further iterated to improve its features in relation to the requirements (Section 3.1).

4.4.1 Traditional Prototype

The initial proof of concept, or prototype, of the transformation tool, has been implemented in Java (with respect to NR5 and NR7). This enables the use of the existing libraries for NS models (de-)serialisation and manipulation. On the other hand, to work with RDF and OWL in Java code, Apache Jena [155] has been selected for its broad applicability and features, including ontology modelling support.

It uses the well-known visitor pattern – for each NS metamodel element (as shown in Figure 4.6). Two types of visitors were created, one for each direction. The first type of visitor realises the mapping from NS to RDF with possible additional OWL statements based on Algorithm 4.1. The other one queries RDF to add a specific NS construct in the NS model based on Algorithm 4.4. In both cases, it traversed the whole model; for example, visiting a data element, it subsequently visited all its fields.

The main issue with the prototype turned out to be the evolvability (NR1). Although the designed RDF representation of NS models is direct and only the additional construct (e.g. from OWL) cannot be directly derived from an NS model, it was hard-coded directly in the visitors, as shown in Listing 4.4 (despite the use of inheritance and generics). A change in the NS metamodel would result in updating the used NS libraries. Then, all

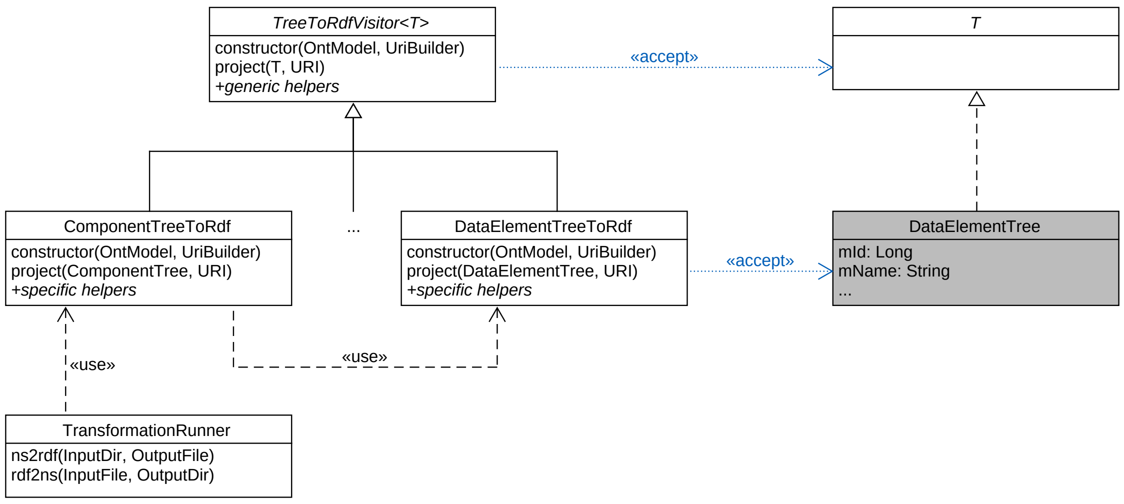

Figure 4.6: Fragment of NS-RDF prototype design

changes would need to be resolved manually in the code of visitors (e.g. a new link between elements or removed deprecated attribute).

Still, the prototype provided the necessary experience with NS to RDF/OWL transformation used for improved implementation. It also allowed creating RDF/OWL representation of the NS metamodel to work on the design of the gateway ontology.

4.4.2 Expanded Transformation Tool

After proving that the design is implementable and leads to the desired bi-directional transformation, we focused on the evolvability of the realisation. In the next design cycle iteration, the tool was re-designed to expand as an NS application. All the parts related to an NS (meta)model can be directly expanded. The NS-RDF mapping is designed as 1:1 with rules to compose identifiers and navigate a model.

We designed three expanders (code templates and mappings) that are used to generate metamodel-related classes for $T_{NS \to RDF}$:

- `VocabularyExpander` = An expander for a vocabulary class per each component. A vocabulary class contains static constants for all URI of data elements (OWL classes) and fields (OWL properties). It separated the concern of building URIs, and all other parts of the tool use it as a single source of truth when a URI is needed. For example, instead of synthesising a string `https://.../elements#DataElement` it uses a reference `Elements.DataElement`.

- `TreeToRdfExpander` = An expander for classes that accepts a tree projection, adds new statements according to the mapping to the RDF graph and handles the propagation to related tree projections as described by Algorithm 4.1. For example, for a

Listing 4.4: Example of data element to ontology model transformation

```java
public class DataElementTreeToRdf implements TreeToRdfVisitor<DataElementTree> {

  public DataElementTreeToRdf(OntModel model, UriResolver uriResolver) {
    // ...
  }

  @Override
  public Individual project(DataElementTree dataElementTree, URI parentURI) {
    URI dataElementURI = uriResolver.uriFor(parentURI, dataElementTree.getName());

    // Individual
    Individual individual = model.createIndividual(dataElementURI, NS.DataElement);
    individual.addProperty(NS.DataElement_name, dataElementTree.getName());
    // ... other properties

    // DataElement-dataOptions link field
    DataOptionTreeToOwl dataOptionTreeToOwl = new DataOptionTreeToOwl(model,
↪   uriResolver);
    ArrayList<RDFNode> dataOptionNodes = new ArrayList<>();
    for (DataOptionTree dataOption : dataElementTree.getDataOptions()) {
      dataOptionNodes.add(dataOptionTreeToOwl.transform(dataOption, dataElementURI));
    }
    individual.addProperty(NS.DataElement_dataOptions,
↪   model.createList(dataOptionNodes));
    // ... other relations

    // NS-OWL
    OwlClass owlClass = model.createClass(dataElementURI);
    owlClass.setSuperClass(OWL2.Thing);
    owlClass.addProperty(RDF.type, NS.DataElement);
    owlClass.addProperty(RDFS.label, dataElementTree.getName());
    // ... fields as properties

    return individual;
  }

}
```

data element, it can accept an instance of `DataElementTree`, add it together with all of its properties to the RDF graph, and propagate the visit to fields, data projections, and others of the data element.

- `RdfToTreeExpander` = An expander for classes that accept a graph and a resource of the entity for which it was expanded. Based on Algorithm 4.4, it queries the RDF graph and extracts all the information to create a tree projection for the entity.

Then it again handles the propagation to related entities (using object relations). For example, for a data element, it can accept a resource that is RDF representation of a data element, queries the given RDF graph for all information, builds a tree projection, and propagates the visiting to fields, data projections, and other resources related to the data element.

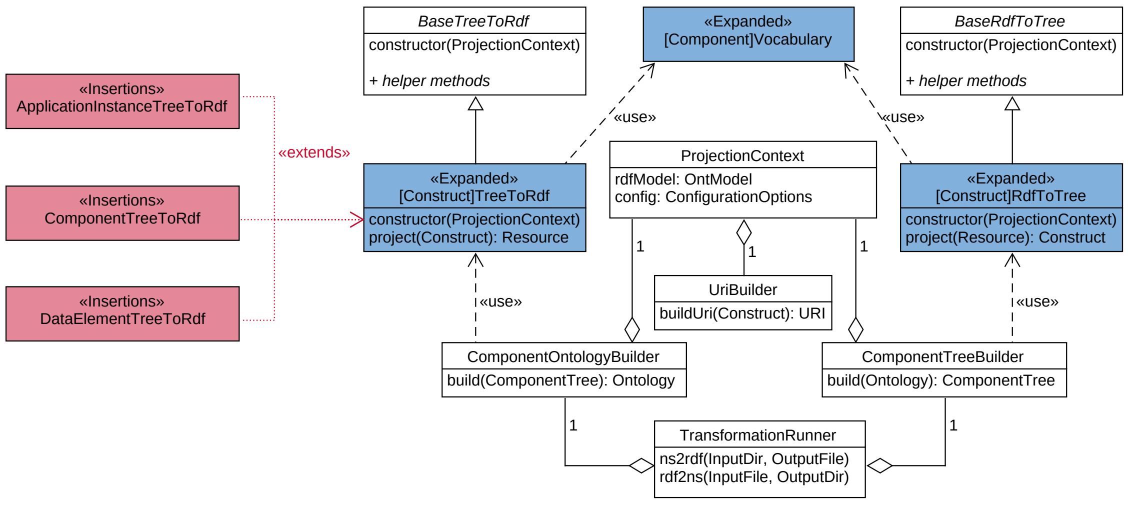
BaseTreeToRdf
constructor(ProjectionContext)
+ helper methods
«Expanded» [Component]Vocabulary
BaseRdfToTree
constructor(ProjectionContext)
+ helper methods
«Insertions» ApplicationInstanceTreeToRdf
«Insertions» ComponentTreeToRdf
«Insertions» DataElementTreeToRdf
«extends»
«use»
«use»
ProjectionContext
rdfModel: OntModel
config: ConfigurationOptions
«Expanded» [Construct]TreeToRdf
constructor(ProjectionContext)
project(Construct): Resource
«Expanded» [Construct]RdfToTree
constructor(ProjectionContext)
project(Resource): Construct
1
1
1
UriBuilder
buildUri(Construct): URI
«use»
«use»
ComponentOntologyBuilder
build(ComponentTree): Ontology
ComponentTreeBuilder
build(Ontology): ComponentTree
1
1
TransformationRunner
ns2rdf(InputDir, OutputFile)
rdf2ns(InputFile, OutputDir)

Figure 4.7: Simplified diagram of classes in expanded NS-RDF tool

Several classes must be maintained separately in the expanded application as there is no reason to expand them (they are not coupled with the metamodel), as shown in Figure 4.7. These are mainly utility classes that handle: (de)serialisation of RDF and Tree projections, hold RDF projection context that is passed upon visiting resources, or constants such as the URI prefix. The function and the design of these classes remained almost the same as in the traditional prototype; the only difference is the access to the generated classes.

The OWL mapping and $T_{NS \rightarrow OWL}$ is not part of the expanded code, but it is designed as a harvested additional code. That promotes the evolvability of the solution as creating OWL is desired for the NS Elements metamodel, but for other custom NS metamodels, this might not be the case. Moreover, in expanders, it would need some additional conditions or change the NS Elements metamodel to capture this particular behaviour using options (e.g. data option for the data element, that it should be turned into OWL class). The OWL mapping can be managed in the expanded application and harvested using standard NS development workflow and tooling.

For both the expanders and the expanded application, we also designed and implemented a set of tests. The expander test consists of input NS models and desired results (classes). Tests for the expanded application use a similar idea of integration testing. There are prepared input NS models and RDF/OWL and expected outputs. The test cases are always focused on a specific construct, for example, that a data element is transformed into an OWL class or that a resource of the value field is transformed into a tree projection of the value field related to the correct data element.

4.4.3 Transformation Verification Procedure

As we designed the transformation as bi-directional and implemented a tool for executing the transformation on input NS models or OWL ontologies (based on the direction), the verification and testing information loss can be done using a cycle as shown in Figure 4.8. The verification approach is similar to the work done by Zedlitz for UML-OWL transformation [156].

As input, we have an NS model m_A confirming to NS Elements metamodel. By transforming it, we get a RDF data set r_A with OWL ontology o_A, that is, $(r_A, o_A) = T_{NS\to RDF/OWL}(m_A)$. By using r_A as an input to reverse the direction of transformation, we get $m_B = T_{RDF\to NS}(r_A)$. We need to evaluate the equivalence of m_A and m_B. The exact same principle can be applied for the opposite direction with an input RDF dataset r_A being first transformed to m_A and then to r_B.

$$m_A \overset{?}{\equiv} T_{RDF\to NS}\left(r\left(T_{NS\to RDF/OWL}(m_A)\right)\right) \tag{4.3}$$

The `semantic equivalence` of two NS models (and two RDF datasets) means that they describe the same (static) semantics – carry the same information but the structure may differ (e.g. names of fields). However, our transformation is designed to follow the structure and even the naming. The models should be equivalent in a way that their serialisation (if formatted and ordered consistently) should be equal on the level of characters:

$$\text{xml}(m_A) \overset{?}{=} \text{xml}\left(T_{RDF\to NS}\left(r\left(T_{NS\to RDF/OWL}(m_A)\right)\right)\right) \tag{4.4}$$

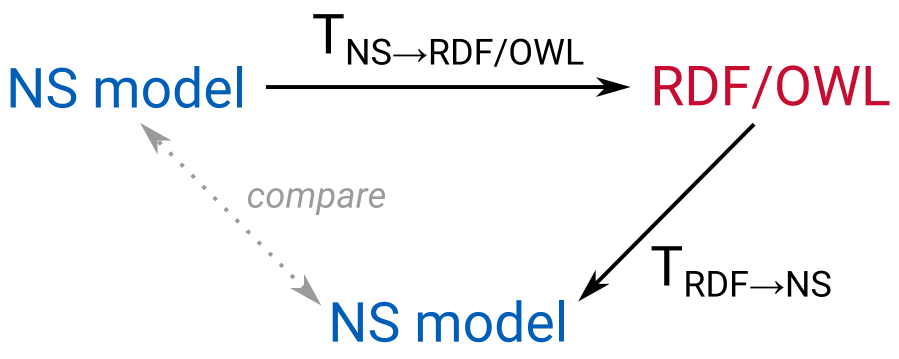

Figure 4.8: Verification of NS-RDF/OWL bi-directional transformation

4.5 Design Cycle of NS-RDF/OWL Transformation

Initially, we designed to capture NS models only as OWL ontologies, i.e. as hierarchies of classes with data and object properties. However, the loss of information was too significant and NS models that would be created purely according to the mapping from OWL ontologies would require additional manual adjustments in NS tooling to expand software systems. In the next iteration, we used the OWL and added the RDF encoding of all information related to the OWL of the metamodel. That design improvement eliminated information loss and also added versatility in terms of custom metamodels.

To verify the mapping with practical use cases in NS (meta)models, we implemented a transformation tool – first as a traditional prototype and then as a semi-expanded NS application. The design of the expanded tool has proven its advantages in terms of evolvability. Still, the bottom-up approach turned also shown its value as the Visitor pattern and general algorithm for traversal and model building used in the traditional prototype has remained unchanged (just mostly moved into expanders).

We also considered designing a direct expansion of RDF/OWL from NS models by having code templates that follow our mappings NS-RDF and NS-OWL. That design would allow only one-way transformation, and we would still need to expand a tool for the other direction. Having a single tool for both directions of the transformation promotes consistency (e.g. the tool is ensured to be expanded using one NS metamodel).

As a final step, we used the bi-directionality of the transformation together with the expanded tool to test and verify the mapping and its implementation. For all tested NS models, the transformation has been lossless. Nevertheless, the expanders and the tool are prepared for future metamodel changes and further improvements when needed.

4.6 Summary of NS-RDF/OWL Transformation

This chapter describes the mapping between Normalized Systems models (including its metamodel) and semantic triples using RDF and OWL. It is used as the essential part of the NS-RDF/OWL bi-directional transformation design (lays foundation for FR1–FR4). We also had to deal with the input and output model representations to design a fully executable transformation. The input model is read recursively during the transformation, and the output model is built based on the mapping according to the algorithms presented.

We verified and evaluated the design of the transformation using implementation. First, we implemented a prototype using the visitor pattern to realise the algorithm in object-oriented programming. Then, we switched the implementation to a partially expanded NS application, where all metamodel-related components are expanded. It directly promotes the evolvability of the transformation tool (FR5, NR1, and NR2). The tool is included in Appendix A. Finally, the verification procedure for bi-directionality of the transformation has been proposed and used. The solution is compliant with the development stack requirement (NR5), is documented (NR6), and multi-platform NR7.

In our overall design, the bi-directional transformation is used to relate the intermediary plane with the NS plane in both directions. It allows us to easily shift the NS metamodel to RDF/OWL for mapping other conceptual modelling languages. The conceptual models are then mapped to RDF/OWL using the NS metamodel OWL constructs. This allows using the transformation in the reversed direction – to transform NS models from RDF to NS compatible format. The bi-directionality of the transformation is essential to check and ensure consistency between models (FR6).

Chapter 5

Using RDF/OWL to Represent and Integrate Conceptual Models

“The purpose of information is not knowledge. It is being able to take the right action.”

Peter Drucker

The previous chapter described the design for the transformation between the Gateway plane and the Normalized Systems (NS) plane. This chapter approaches the Gateway plane from the opposite direction and explains the transformations between conceptual modelling languages and RDF/OWL. The goal is to shift various conceptual models into a *common and machine-understandable language* for mapping and transformation to NS (and vice versa).

We describe the design of capture models made in different conceptual modelling languages (as well as their metamodels) in Resource Description Framework (RDF) and Web Ontology Language (OWL) with a focus on the key aspects and more complex constructs. We also clarify the possibilities allowed in terms of semantic integration (RO1), as well as modularity and evolvability (RO3).

The chapter is based on our previous design of Ontology for Conceptual Model Integration [A.5], mapping of UFO-B to process modelling languages [A.6], review of UML-OWL transformation [A.8], design of OntoBORM [A.13], and other existing work summarised as our knowledge base in Chapter 2.

5.1 Dealing with Heterogeneity in Conceptual Modelling with RDF/OWL

In Chapter 2, we briefly described several conceptual modelling languages of three main kinds: process, structural, and fact. Even methodologies and languages support combinations of those kinds; for example, DEMO combines process and fact modelling. The heterogeneity is a necessity, as it is required to take different aspects into account and capture them in a different level of detail for various purposes.

On the other hand, this heterogeneity raises questions related to knowledge integration between different types of models. For example, how can an ORM fact *Customer* be related to an UML class *Customer* and to a BORM participant *Customer*? One of our objectives (namely RO1) is to allow semantic integration by capturing the relations between modelling languages in a machine-understandable way for automated processing.

The issue of heterogeneity goes beyond the semantics of modelling languages. Despite attempts to unify modelling (UML, UML Profiles, MOF) and formats for their serialisation (XMI), many modelling languages use their own (meta)metamodels and formats. That blocks interoperability and simple processing. By encoding the knowledge of conceptual models using the same mechanism, the interoperability issues caused by the heterogeneity in conceptual modelling can be mitigated.

5.1.1 Syntactic Heterogeneity

Conceptual modelling languages (as other languages) define their syntax and semantics. For example, UML uses a specific rectangular shape for a class in a class diagram with a specific meaning. Naturally, different languages use different symbols to represent concepts, possibly applying cognitive sciences or simply stressing differences or similarities between concepts. The first issue with syntactic heterogeneity appears with the need to integrate models that use different languages, but have the same (or similar) symbols representing different concepts, or vice versa (different shapes for the same concepts).

Another issue is related to a model serialisation, i.e. how a model is stored (using symbols to represent a model). Again, for various purposes, different forms are suitable, e.g. for humans, a diagram, tables, or structured text with highlighting is suitable, whereas, for computer processing, a text or binary file is necessary (when not considering techniques such as OCR). Typically, the form of a model for humans can be created from a file using a CASE tool. However, not all languages have such tooling support. In addition, languages do not have a suitable file format defined as part of the language specification (as a syntax for computer processing).

The absence of a serialisation format for computer use motivates potential language adopters (in terms of tooling) to devise their own serialisation based on a specific need, such as processing efficiency, library support for parsing, or re-use of existing formats. Thus, the tools use different formats to represent the same models, as interoperability is not considered. Even with XMI, tools may use different XMI profile based on their internal needs. Those who then need to process such models need to treat each format (and the internal profile or schema) differently. For example, a thesis on searching in (ontological) Unified Modeling Language (UML) conceptual models [157] clearly demonstrates these issues, as different parsing and processing must be implemented for formats coming from various CASE tools.

5.1.2 Semantic Heterogeneity

The semantic heterogeneity of conceptual modelling languages allows one to focus on different aspects and at a different level of detail. Although it is also related to syntactic heterogeneity, the differences on the semantic level make the languages unique and usable for specific use cases. However, it leads to the need for semantic integration (e.g. our previous example with the concept of *Customer* in different models, thus with different semantics).

Having multiple conceptual models that describe different aspects (through their semantics) gives an opportunity to capture the knowledge more precisely and create a more complete view. The semantic integration is done using the overlaps between the semantics of different modelling languages and capturing the same concept using them. In this sense, semantic heterogeneity is a good feature. On the other hand, it creates a place for the explained syntactic heterogeneity, which prevents capturing and representing the “merged” knowledge efficiently, i.e. relating or merging the symbols of modelling languages.

5.1.3 Modelling Language Specifications

The syntax and semantics of a formal modelling language are defined in its specification. It creates (intentionally or incidentally) a metamodel for the conceptual models. Traditionally, the specification is in the form of a human-readable document, e.g. specification of UML [52] or Business Process Model and Notation (BPMN) [158]. However, some are mainly described in a set of scientific articles, for instance, in the case of Business Object Relationship Modelling (BORM) or OntoUML, which makes it harder to use and refer to a specific language version. In the case of OntoUML, we also contributed to a unified and evolvable specification in the form of web-based documentation driven by machine-actionable descriptors [A.2].

The human-readable specification is important for people (e.g. business or software analysts) to understand the syntax and semantics and use it properly. But for automated processing, such as transformations and mapping specifications. Both the mentioned UML [52] and BPMN [158] are accompanied by a set of machine-readable attachments that contain parts of the specifications in formats such as XMI, XSD, or XSLT. Unfortunately, those are intended for particular use cases and do not encompass the entire specification available in human-readable form.

5.1.4 Absence of Meta-Circularity

A solid metametamodel is needed to support language specification as a language to specify the metamodel. Without such a metametamodel, the specification is not formally supported. The same principle is applicable to the metametamodel, and so on. Thus, meta-circularity is essential to solve this issue. This is the case for NS [143] and also exhibits in Meta-Object Facility (MOF) [53]. Again, both human-readable and machine-actionable forms of the specification are essential to cover both uses by humans and automation using machines.

5.1.5 Conceptual Model as Semantic Web and Linked Data

To comply with our research objectives and architecture set in Chapter 3, we pursue the representation of conceptual models (and their metamodels in RDF/OWL as shown in Figure 5.1). It provides machine-actionability, improves evolvability (by enabling modularity, versioning, and referencing), and allows semantic integration. These properties come from the original intentions of RDF/OWL. As described in Chapter 2, the Semantic Web focuses on a representation of knowledge in a machine-actionable way. The formation of relations between knowledge and creating references across concepts is the domain of Linked Data.

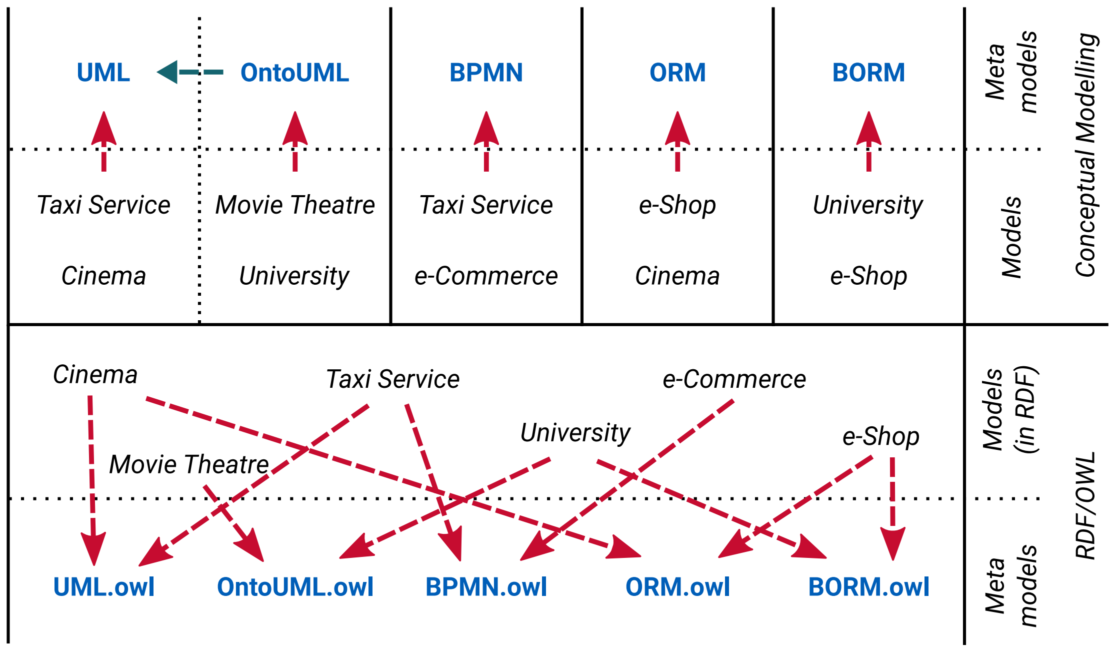

Figure 5.1: Using RDF/OWL to deal with metamodel heterogeneity

The idea of creating OWL ontologies for conceptual modelling languages and representing models in RDF for interoperability and analysis or inference exists as long as OWL itself. There is a variety of existing work for the main conceptual modelling languages.

Our contribution is to revisit the existing work, adjust and update it for our use case, and create the ways of RDF/OWL representation for other modelling languages where no previous work is done.

5.2 Conceptual Modelling Ontologies for NS Gateway

In the following sections of this chapter, we present the ontologies of the selected conceptual modelling language to support RDF/OWL representation of conceptual models. The selected languages reflect our research goals, which are to cover modelling, structural modelling, and fact modelling representatives. I The complete ontologies, including its documentation, are part of Appendix A. To support the explanations, we also provide brief examples. Again, complete examples are included as part of the ontologies documentation.

In the scope of this thesis, manual transformation from various conceptual model representations to RDF/OWL is intended. However, it does not place any obstacles to the implementation of automatic transformation. The main challenge would be to cover different formats of different languages and computer-aided software engineering (CASE) tools as the heterogeneity is significant. We do not expect any new research questions emerging from the implementation of such transformation mechanisms between formats that represent the same knowledge. Moreover, our ontologies are designed based on the language specifications or its defined subsets, so the mappings for transformation will be straightforward – one-to-one mapping. It may even be a bijective mapping, unless the tool adds custom concepts not included in the specification.

5.3 UML Models in RDF/OWL

UML is a widely used and well-known modelling language; there are already existing methods and captured ideas on transforming UML models to OWL ontologies (and back). These methods focus mainly on UML Class Diagrams. We review the main approaches to UML-OWL transformation [A.8]. We identified the QVT-based method by Zedlitz et al. [156] as the most complete and elaborated, including the bi-directionality. Other of the reviewed methods also had exciting ideas, such as using XSLT or mathematically defined transformations. However, none of the methods is ready for use to transform Class (and Activity) Diagrams to RDF/OWL with their metamodels captured in OWL.

5.3.1 Motivation for UML in RDF/OWL

Our need is to be able to capture a UML model as RDF with additional OWL triples in a similar way as we have for NS-OWL described in the previous chapter. For example, a UML class Customer should be captured in RDF as an individual of type `uml:Class` (including its required properties, such as name), and in the additional OWL, it should state that it is also of type `owl:Class`. As a suitable UML ontology that would allow capturing at least UML Class and Activity Diagrams to RDF/OWL does not yet exist, we need to design and create it.

Our goal is not to cover the entire UML metamodel but to allow for the capture of conceptual models made in UML Class and Activity Diagrams (as the principal diagrams used in conceptual modelling) in RDF/OWL for their transformation to and from NS models. UML State Machine Diagram is also widely used as a child diagram for classes from a Class Diagram. Therefore, our ontology of the UML 2.5 metamodel is focused only on the constructs relevant to conceptual modelling. However, the ontology can be extended in the future or even replaced by a complete UML metamodel ontology that will also serve other purposes. During our work, the review [A.8] of existing UML-OWL transformation is a valuable source of information for mappings.

5.3.2 UML Ontology for Transformation Design

To allow UML models to be captured in interoperable RDF, we first design the ontology. As explained, we need to support three types of diagrams from UML that are essential for conceptual modelling. UML is about the notation, i.e. graphical representation, whereas in RDF, only the semantics matter. These two facts are the cornerstones for our UML ontology design. It can be split into three interconnected parts related to Class, State Machine, and Activity Diagrams. It contains only crucial constructs to capture the semantics of conceptual models made in these types of diagrams; other properties related to their representation, implementation, or connection to other UML diagrams are intentionally omitted.

For the ontology, we use UML specification version 2.5.1 (the current version). Having a single ontology for multiple UML diagrams eliminates the need to maintain its metadata and compatibility with versions. It also allows for the integration of the models while encoding them in RDF directly. For example, it is expected that a class will have states defined for its state machine and may participate in some process.

5.3.3 UML Class Diagram Ontology

To capture in OWL the essential parts of the UML metamodel related to the UML Class Diagram, we applied the knowledge from the reviewed existing transformations [A.8], various conceptual models in UML, and also the Ecore metamodel. Class diagrams typically represent a single package with its classes. The associations can be done even between classes of different packages. Instead of creating a top-level class for a diagram, our design uses `uml:Package` directly.

A package may contain instances of `uml:Classifier` which is either `uml:Class` or `uml:DataType`. For a class, it is possible to specify associations and attributes. We intentionally do not include operations at this point, as it is not essential for conceptual modelling, where it is often substituted using derived attributes. A class can be marked as abstract and may have a specified stereotype.

For an instance of `uml:Attribute` it is possible to state the classifier used as a type, lower and upper bound, default value, and a set of constraints. `uml:Association` is more complex, as UML supports N-ary relations, as well as special binary relations, i.e. `uml:BinaryAssociation`, `uml:Composition`, or `uml:Aggregation`. An association always has its name. It may also link an association class using `uml:hasAssociationClass`. An N-ary association must have more than two association ends; binary has always two.

The design of UML ontology defines `uml:AssociationEnd` that captures lower and upper bound together with role name. Moreover, we distinguish navigability by using sub-classes `uml:NavigableAssociationEnd` and `uml:NotNavigableAssociationEnd` which are disjoint. However, their union does not cover `uml:AssociationEnd` as UML allows for unspecified navigability.

The property of relating an end-to-end association to an association is in the most generic form `uml:hasAssociationEnd`. Nevertheless, there is the `uml:hasOwnedEnd` sub-property that may appear just once of an association. For aggregation (shared aggregation) and composition (composite aggregation), there is always a role related using `uml:hasAggregateEnd`.

The `uml:Enumeration` class represents a special kind of `uml:DataType` that contains a set of values represented using `uml:EnumLiteral`. Each literal has just its name and optionally also a description. For data types, we include a set of pre-defined individuals. For example, `umlType:String` for representing string value and listing compatible XSD types.

A generalisation between two classes (also known as inheritance or generalisation-specialisation relation) is in our ontology also represented as a class – `uml:Generalisation`. At always has its `uml:hasSuperClass` and `uml:hasSubClass`. The reason to require individuals to represent inheritance in UML models is the possibility of including them in a generalisation set. The `uml:GeneralisationSet` allows one to group and name a set of generalisations and specify constraints such as complete and disjoint.

To abstract the common properties, we use, according to the UML and Ecore meta-models, the `uml:ModelElement` and `uml:NamedElement` superclasses. All of our defined classes are always subtype of `uml:ModelElement` and those with `uml:hasName` use as its type the `uml:NamedElement` class.

UML uses class and association stereotypes. Similarly, there are constraints for associations, association ends, generalisation sets, and attributes. Those create a hierarchy so, for example, only `uml:ClassStereotype` can be assigned to a class and not to an association. Finally, we pre-define individuals for well-known stereotypes and constraints such as `umlStereotype:Interface` or `umlConstraint:Ordered`. UML profiles that use stereotypes can define individuals in the same way (as will be shown for OntoUML).

5.3.4 UML State Machine Ontology

State Machine Diagrams are typically related to a certain class to express its states and possible transitions between them. However, UML also allows the creation of a stand-alone State Machine Diagram. Therefore, our ontology captures a `uml:StateMachine` that may be related to a `uml:Class`. Then, it is possible to have `uml:State` with special subclasses for different types of states: Initial, Final, History, Synchronisation, Exit, and Entry. Then, there is `uml:Transition` for the process flow.

As State Machine Diagrams also allow branching and parallel flows, the ontology contains `uml:Choice` and `uml:ForkJoin`. Together with `uml:State` they are subclasses of `uml:TransitableElement` – elements that can be a source or target of a transition. Finally, a state machine can be used again in another state machine for modularity. It is directly possible as an inner state machine must again have initial and final states, typically entry and exit states.

5.3.5 UML Activity Diagram Ontology

There are various behavioural modelling diagrams in UML, but as we explained, the Activity Diagram is the most commonly used and suitable for conceptual modelling. In our design, a single diagram represents an instance of `uml:Process` that contains its `uml:ProcessElement` instances. A process may use several partitions representing different participants using `uml:Partition`. Again, a partition may contain `uml:ProcessElement` instances, here is the object property `uml:hasProcessElement` instances transitive.

An instance of `uml:Activity` allows us to contain actions, objects, and control nodes. Moreover, activity may have certain parameters with name and type, similar to the previously explained `uml:Attribute`. For actions, there are various types of actions captures using subclasses of `uml:Action`, e.g. `uml:TimeAction`, that may be used to specify period or wait time. If an activity is `uml:StructuredActivity`, it may again contain elements using the transitive property `uml:hasProcessElement`.

There are also distinguished different types of control nodes: initial, final, decision, merge, and fork-join. Between activities, actions, and control nodes can be established an instance of `uml:ControlFlow` using object properties `uml:flowStartsIn` for the source node and `uml:flowEndsIn` for the target node. It is possible to give a name to a flow and a state condition.

UML Activity Diagram allows `uml:SendEvent` and `uml:Receive` events to be used in a control flow but also using the special `uml:InterruptFlow`. Finally, the designed ontology also supports the use of `uml:Object` in the flow by relating it to `uml:objectFlow`. An object flow must always have an object as a source or target. The object is naturally related to a class and optionally to its state.

5.3.6 UML in RDF Example

The ontology that covers the UML Class, State Machine, and Activity Diagrams enables us to encode the models as individuals in RDF. However, it is expected that at the model level, the individuals will additionally be `owl:Class`. For example, `uml:Customer` will be an instance of `uml:Class` and of `owl:Class`. We present a brief example of individuals linked across the three types of diagrams in Listing 5.3.

Listing 5.1: UML example in RDF

```
@prefix uml: <https://purl.org/nsgo4cm/cm-ontology/uml#> .
@prefix umlConstraint: <https://purl.org/nsgo4cm/cm-ontology/uml/constraints#> .
@prefix umlType: <https://purl.org/nsgo4cm/cm-ontology/uml/types#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix : <http://example.com/model/uml#> .

:Customer a uml:Class, owl:Class ;
          rdfs:label "Customer" ;
          uml:hasAttribute [
            a uml:Attribute ;
            rdfs:label "name" ;
            uml:dataType umlType:String ;
            uml:lowerBound 1 ;
            uml:upperBound 1 ;
          ] , [
            a uml:Attribute ;
            rdfs:label "e-mail address" ;
            uml:dataType umlType:String ;
            uml:lowerBound 1 ;
            uml:upperBound 1 ;
            uml:hasConstraint umlConstraint:unique;
          ] ;
          uml:hasStateMachine :CustomerSTM ;
          uml:participatesIn :CustomerPartition .

:CustomerSTM a uml:StateMachine .  # (...states and transitions)

:CustomerPartition a uml:Partition .  # (...activities and flow)
```

5.3.7 Ecore as UML Subset

Ecore metamodel is de-facto subset of UML [129]. In addition, we have already been working on the mapping between Ecore and NS directly [A.12]. We can define the Ecore ontology as a subset of our UML ontology with different handling of associations. In Ecore ontology, we first map directly matching classes and properties using `owl:equivalentClass` and `owl:equivalentProperty`. For example, there is a statement for `ecore:Classifier` to express its relationship to `uml:Classifier`. It captures the relation between the UML and Ecore ontologies, but does not limit them to having different features.

The main difference is in associations. Ecore deals with them using reference fields. Therefore, instead of having corresponding classes for the `uml:Association` hierarchy, there is `ecore:EReference` that is closer to an end of association but in the other direction. It still points to the target classifier with name, lower, and upper bound, but the opposite direction is just optionally captured using `ecore:inverseReferenceOf`. Ecore also does not have the concepts of stereotypes, constraints, and generalisation sets. Instead, it has the concept of annotations that allow providing additional information to model elements.

By having both ontologies for UML and Ecore, it is possible to create models that combine the two metamodels, as shown in Listing 5.2. This also allows one to start with a simpler Ecore model and transform it to UML later. The only issue that needs a transformation procedure is related to the associations (and references) and annotations.

Listing 5.2: Ecore example with UML in RDF

```
@prefix ecore: <https://purl.org/nsgo4cm/cm-ontology/ecore#> .
@prefix uml: <https://purl.org/nsgo4cm/cm-ontology/uml#> .
@prefix umlConstraint: <https://purl.org/nsgo4cm/cm-ontology/uml/constraints#> .
@prefix umlType: <https://purl.org/nsgo4cm/cm-ontology/uml/types#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix : <http://example.com/model/uml#> .

:Customer a ecore:Class, uml:Class, owl:Class ;
          rdfs:label "Customer" ;
          ecore:hasAttribute [  # Ecore attribute with annotations
            a ecore:Attribute ;
            rdfs:label "name" ;
            ecore:dataType xsd:string ;
            ecore:lowerBound 1 ;
            ecore:upperBound 1 ;
            ecore:hasAnnotation [
              a ecore:Annotation ;
              ecore:annotationSource "Example" ;
              ecore:annotationDetail [
                ecore:annotationKey "hint" ;
                ecore:annotationValue "Enter first and last name" ;
              ] ;
            ] ;
          ] ;
          uml:hasAttribute [  # UML attribute with constraint
            a uml:Attribute ;
            rdfs:label "e-mail address" ;
            uml:dataType umlType:String ;
            uml:lowerBound 1 ;
            uml:upperBound 1 ;
            uml:hasConstraint umlConstraint:unique;
          ] .
```

5.4 OntoUML Models in RDF/OWL

OntoUML as a UML profile that extends the UML Class Diagram with various stereotypes from the Unified Foundational Ontology (UFO) can be captured using our ontology described in the previous section. It can be done using custom stereotypes for classes (e.g. Kind, Relator, or Phase) and associations (e.g. material, mediation, or memberOf). However, to support it directly, we define the UML profile for OntoUML at the OWL level by creating special named individuals from the `uml:Stereotype` subclasses. This approach applies the same principles as the OntoUML UML profile only defines UML stereotypes with corresponding names to UFO.

5.4.1 OntoUML as UML Profile in RDF

As shown in Listing 5.3, UML models based on class diagrams enhanced with OntoUML stereotypes. The actual semantics of the stereotypes are captured only as a description of the individuals defined in our `ontouml:` ontology. Therefore, there are no restrictions directly for the RDF data; for example, a class with the *Category* (non-sortal) stereotype can be captured as a subtype of a class with the *Kind* (sortal) stereotype, which is against the rules of UFO. However, the RDF/OWL representation of OntoUML as a UML profile enables one to define such constraints using Shapes Constraint Language (SHACL) shapes or validate through SPARQL Protocol and RDF Query Language (SPARQL) queries.

Listing 5.3: Example of OntoUML as UML profile in RDF

```
@prefix uml: <https://purl.org/nsgo4cm/cm-ontology/uml#> .
@prefix ontouml: <https://purl.org/nsgo4cm/cm-ontology/uml/ontouml#> .
@prefix umlType: <https://purl.org/nsgo4cm/cm-ontology/uml/types#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix : <http://example.com/model/ontouml#> .

:Customer a uml:Class, owl:Class ;
         rdfs:label "Customer" ;
         uml:hasAttribute :hasCustomerName ;
         uml:hasClassStereotype ontouml:KindStereotype .

:hasCustomerName a uml:Attribute, owl:DatatypeProperty ;
                 rdfs:label "name" ;
                 rdfs:domain :Customer ;  # can be derived from uml:hasAttribute
                 uml:dataType umlType:String ;
                 rdfs:range xsd:string ;  # can be derived from uml:dataType
                 uml:lowerBound 1 ;
                 uml:upperBound 1 .
```

The advantage of this OntoUML RDF representation is its simplicity and the direct use of UML constructs. It allows straightforward semantic integration with State Machine Diagrams and Activity Diagrams captured using our OWL ontology for UML. Moreover, it can further be used as a serialisation format with better interoperability than a customised XMI as some tools (e.g. Menthor Editor).

5.4.2 OntoUML in RDF using gUFO

The other option to capture OntoUML models in RDF is the use of the existing OWL ontology called gUFO [133]. In Listing 5.4, the same example is shown as for the previous method. Instead of creating individuals based on the UML ontology and assigning the defined stereotypes, individuals use UFO entity types directly. For example, to create an entity of type Kind, instead of using `uml:Class` with the stereotype `ontouml:Kind`, `gufo:Kind` is used.

Listing 5.4: Example of OntoUML in RDF using gUFO

```
@prefix gufo: <http://purl.org/nemo/gufo#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix : <http://example.com/model/ontouml-gufo#> .

:Customer a owl:Class, gufo:Kind ;
          rdfs:subClassOf gufo:Object ;
          rdfs:label "Customer" .

:hasCustomerName a owl:DatatypeProperty ;
                 rdfs:domain :Customer ;
                 rdfs:range xsd:string .
```

The gUFO ontology is maintained by the Nemo research group that is behind the development of UFO and OntoUML. Thus, the main advantage of this approach when compared to the previous one is the sustainability and re-use of existing ontology. The ontology also deals with the hierarchy of entity types of UFO, such as sortality and rigidity. Finally, as it is a "lightweight implementation of UFO suitable for Semantic Web OWL 2 DL applications", it also includes additional types from UFO-B and UFO-C. We further investigated the practical possibilities of OntoUML in RDF with gUFO in [A.44].

5.4.3 Integrating OntoUML in RDF

The two approaches presented to capture OntoUML models are not mutually exclusive. In use cases where both advantages of the approaches are required, RDF for an OntoUML model can combine them. An example of such a combination is shown in Listing 5.5. Whereas the gUFO is minimalistic and based on existing OWL constructs, our UML ontology allows capturing various (Onto)UML-specific. A single UML class can be directly associated with entity type from the gUFO. In our example, the Customer class is simultaneously a UML class (with Kind stereotype) and Kind (from gUFO). Similarly, it would be done for other constructs captured in both ontologies, for example, associations. However, attributes are not subjected to the gUFO; those are captured as in Listing 5.3.

Listing 5.5: OntoUML in RDF as UML profile combined with gUFO

```
@prefix gufo: <http://purl.org/nemo/gufo#> .
@prefix uml: <https://purl.org/nsgo4cm/cm-ontology/uml#> .
@prefix ontouml: <https://purl.org/nsgo4cm/cm-ontology/uml/ontouml#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix : <http://example.com/model/ontouml-combined#> .

:Customer a uml:Class, gufo:Kind, owl:Class ;
          rdfs:subClassOf gufo:Object ;
          rdfs:label "Customer" ;
          uml:hasAttribute :hasCustomerName ;
          uml:hasClassStereotype ontouml:KindStereotype .
```

5.5 BPMN Models in RDF/OWL

BPMN is widely used to model processes in the enterprise environment. The notation is highly affected by business analysis and also the orchestration aspects. Regarding the three levels of modelling in BPMN, conceptual modelling focuses on the first two (descriptive and analytical). The third (executable) level is related to transformation to BPEL and orchestration, e.g. tool-specific gateways. Our ontology needs to focus on the essential constructs of conceptual modelling and selected universal constructs to describe the concepts for execution.

As BPMN is well-known and widely used with good tooling support, we used previous works as a knowledge base for our design. Similarly to the reviewed methods for UML-to-OWL, there is a method by Kchaou et al. [131] that proposes BPMN-to-OWL transformation. Sanfilippo, Borgo, and Masolo [159] provide an ontological analysis of BPMN in which the semantics of events and activities are clarified.

Finally, Natschläger [160] proposes OWL ontology for the full BPMN 2.0 with 260 classes, 178 object properties, and 59 data properties. Unfortunately, some of the constructions from that ontology were too complex for our use cases. Nevertheless, it is still possible to use this ontology and even create a mapping with the one designed by us. We present an example of such mapping together with the primary differences.

5.5.1 Conceptual Part of BPMN

Our BPMN ontology focuses only on the Collaboration Diagram of BPMN, as it is the main diagram used for process description. Both Conversation and Choreography Diagrams represent interactions between participants and pools. We also do not include transactions and call activities, as those are related to the technical realisation. The reusability of call activity can be achieved quickly without having a particular type for it.

We need to support all types of tasks, markers, events, flows, and gateways from the notation constructs for the Collaboration Diagram. Moreover, we also need data elements that represent the crucial concept of data objects passed during the flow. Groups and text annotations are omitted from our ontology, as such documentation-related constructs are handled differently in RDF.

5.5.2 BPMN Ontology

The top-level class in our BPMN ontology is the `bpmn:Collaboration` with name, description, and pools. There is a difference between `bpmn:WhiteBoxPool` that must have swimlanes and `bpmn:BlackBoxPool` that must not have swimlanes but allows message exchange directly. Each `bpmn:Swimlane` then contains model elements that are activities, events, gateways, data elements, and flows. The principle of transitivity for subprocesses is identical to the one explained for the UML Activity Diagram.

All activities (instances of `bpmn:Activity`) may have associated some of the predefined `bpmn:Marker` individuals. The use of a subprocess marker requires one to attach a process flow. Our ontology uses only one type of activity, which is `bpmn:Task`. However, for each type of task, there is a subclass, for instance `bpmn:SendTask` or `bpmn:ServiceTask`. Similarly, subclasses are used to distinguish different types of `bpmn:Gateway`.

A more complex hierarchy is designed for `bpmn:Event` where there are three orthogonal trees of subclasses. The first is to deal with specialised types, for example `bpmn:ErrorEvent` or `bpmn:MessageEvent`. The second is about its use, for example `bpmn:StartEvent`. The third is to express the behaviour, for example `bpmn:NonInteruptingEvent`. An event instance may then use a combination of these subclasses. The minimal one uses just `bpmn:StartEvent`, `bpmn:IntermediateEvent`, or `bpmn:EndEvent`. In that case, it is an untyped event with standard behaviour – in the BPMN diagram expressed as an empty circle.

For data elements, we use `bpmn:DataObject` with subclasses `bpmn:DataCollection`, `bpmn:DataInput`, and `bpmn:DataOutput`. Then, there is also `bpmn:DataStore`. The instances of these classes can be associated with activities and processes via an instance of `bpmn:DataAssociation`. Similarly, `bpmn:MessageFlow` is also designed as a class in our ontology and related flow with message events, message tasks, or black box pools used as sources and targ. A black box pool that can be used for both.

The last thing to define is `bpmn:SequenceFlow`, again captured as a class. BPMN allows default and conditional flows, but these are related to gateways. For example, an exclusive gateway may use an outgoing sequence flow by default by using the specialised property `bpmn:hasDefaultOutgoingFlow` instead of the generic property `bpmn:hasOutgoingFlow`. However, a condition must be attached to a `bpmn:SequenceFlow` directly for conditional flows.

5.5.3 BPMN Models in RDF

With our BPMN, we can encode collaboration diagrams from the BPMN models in RDF. Listing 5.6 shows a part of such a model. As our ontology is limited to the essential conceptual constructs, additional information from BPMN is omitted. Compared to the use of the complete BPMN 2.0 Ontology [160], the RDF representation is more straightforward and shorter.

Listing 5.6 shows a simple example of part of a BPMN model captured in RDF. There is a single collaboration where a customer places an order validated and then approved or denied by the system. It shows how pools and swimlanes are encoded, how events and activities are related to a swimlane, and how sequence and message flows are used. When the gateway and subprocesses are incorporated into a model, the use of flows is identical (`bpmn:elementNonInteruptingEventOf` does not bind elements to a swimlane but to an activity, and flows are from or to gateways).

5.5.4 Relating to BPMN 2.0 Ontology

In Listing 5.7, we demonstrate how our ontology is assigned to the complete BPMN 2.0 ontology presented by Natschläger [160]. Mappings that use marking-equivalent classes and properties are added directly to our BPMN ontology. As such, it allows for the extension and simple shifting of data (BPMN models) from one ontology to another or for the definition using both ontologies simultaneously.

5.6 BORM Models in RDF/OWL

Unlike the previously mentioned languages in the previous three sections, there was no previous related work for the BORM – process modelling method. Although the BORM method is not widely used, it has its advantages (as described in Section 2.1.1.7), and we observed several similarities with modelling using flow and task elements in Normalized Systems.

Another motivation to work on the BORM method is the absence of a unified model serialisation format. The tools for BORM modelling (Craft.CASE, OpenPonk, and legacy OpenCABE) use their own formats for storing whole projects. The only way how to switch between tools is to model everything again. Custom analytics over models (e.g. evaluating the difficulty of a process for a particular participant) are also hindered by non-understandable and un-documented formats.

The goal was to design and demonstrate the ontology for BORM to represent the Business Architecture (BA) and Objects Relations (OR) models in RDF. The work also

Listing 5.6: BPMN model fragment in RDF

```
@prefix bpmn: <https://purl.org/nsgo4cm/cm-ontology/bpmn#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix : <http://example.com/model/bpmn#> .

:c1 a bpmn:Collaboration ;
  rdfs:label "Customer places order" .

:Customer a bpmn:Participant, owl:Class ;
        rdfs:label "Customer" .

:CustomerLane a bpmn:BlackBoxPool ;
            bpmn:participant :Customer .

:EShop a bpmn:WhiteBoxPool ;
       rdfs:label "E-shop system" .

:Website a bpmn:Swimlane ;
       rdfs:label "E-shop website" ;

:orderStart a bpmn:StartEvent, bpmn:MessageEvent ;
            bpmn:elementOf :Website .

:flowReceiveOrder a bpmn:MessageFlow ;
                  bpmn:flowSource :Customer ;
                  bpmn:flowTarget :orderStart .

:validateOrder a bpmn:Activity ;
               rdfs:label "Validate order" ;
               bpmn:elementOf :Website;

:flowStart a bpmn:SequenceFlow ;
       bpmn:flowSource :orderStart ;
       bpmn:flowTarget :validateOrder .
```

includes examples for semantic integration as well as for using other technologies related to RDF such as SPARQL or SHACL. [A.13]

5.6.1 OntoBORM – Ontology for BORM

The BORM Ontology is designed based on BORM metamodel for the OR and BA diagrams. It is a single ontology (as shown in Figure 5.2) that describes both metamodels for OR and BA diagrams, as those are closely related and represent the core part of modelling in BORM. It cannot use only Resource Description Framework Schema (RDFS) as OWL

Listing 5.7: Example of mapping to BPMN 2.0 Ontology

```
@prefix bpmn: <https://purl.org/nsgo4cm/cm-ontology/bpmn#> .
@prefix bpmn20base: <https://www.scch.at/ontologies/bpmn20base.owl#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .

bpmn20base:Activity owl:equivalentClass bpmn:Activity .

bpmn20base:SignalEventNonInterrupting owl:equivalentClass [ owl:unionOf
                                                               bpmn:IntermediateEvent,
                                                               bpmn:SignalEvent,
                                                               bpmn:NonInteruptingEvent
                                                          ] .
```

is used for metamodelling support. The suggested prefix for BORM ontology is `borm:`. The naming is taken directly from BORM metamodel; changes and additional elements are explicitly mentioned further.

The ontology is annotated with its own metadata for usability and documentation purposes. Aside from the basic OWL metadata (`owl:versionInfo`, `dct:contributor`, or `dct:modified`), all of the classes and properties specified by the ontology have its own `rdfs:label`, `rdfs:comment`, and `skos:definition`. These textual literal are currently in English; however, more languages can be added at any time. The metadata are aligned for the tool WIDOCO [97].

Some of the classes are designed to be instantiated by domain-specific entities. For example, it allows `org:SalesPerson` to be of type `borm:Person`. Then, an instance of a salesperson can be a particular human being, for example, John to be of type `org:SalesPerson` (which is also `owl:Class`). The same principles may be applied to all of our classes. There are three abstraction levels:

1. Metamodel level = BORM ontology in OWL that defines how to define processes.

2. Model level = BORM model in RDF+OWL that defines processes in an organisation in accordance with BORM ontology.

3. Instance level = BORM model in RDF that captures instances of processes (e.g. who and when participated in the process).

Our target in this work is the first two levels. However, the third level is fully supported and can help organisations track processes, simulate, and verify them.

5.6.2 Representing BORM BA in RDF

The business architecture part of the ontology contains four core classes:

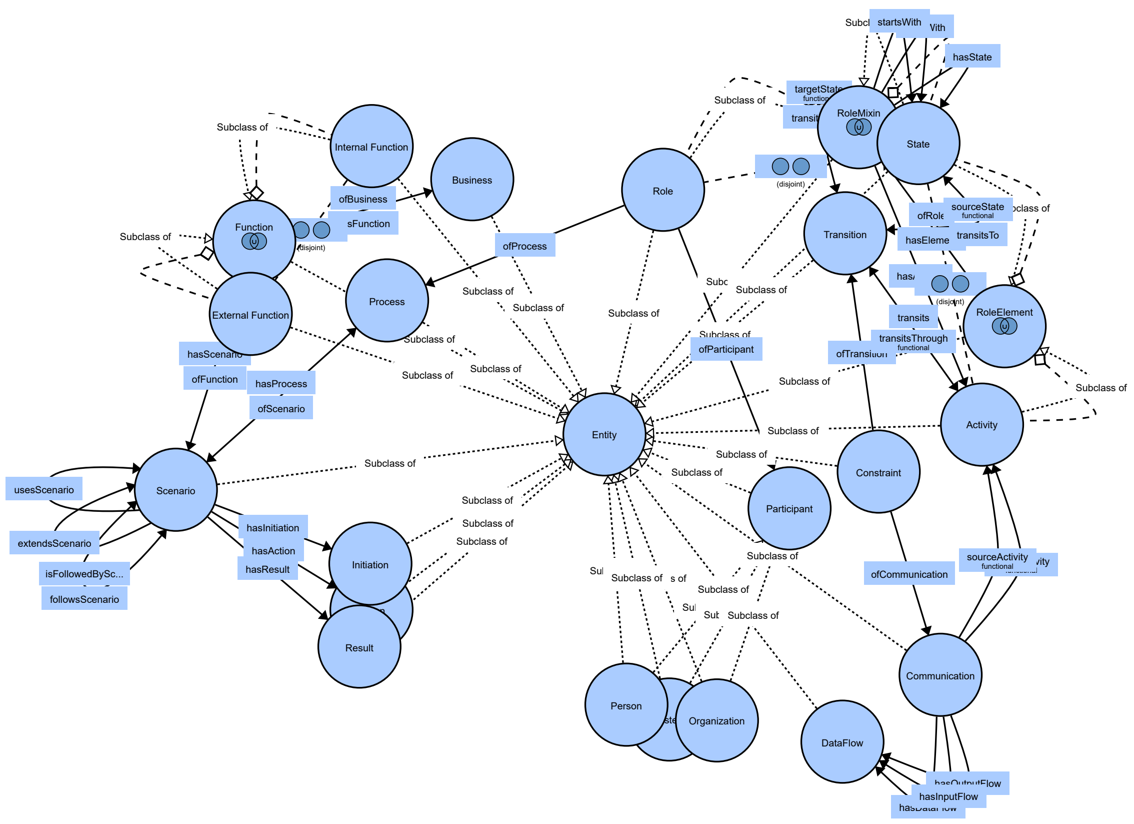

Figure 5.2: Visualisation of OntoBORM (using WebVOWL)

- ◦ `borm:Business` is not directly defined in the BORM method. However, we need a single top-level entity that bounds others. It represents a modelled business organisation.
- ◦ `borm:Function` represents a business function. It has two subclasses based on classification of functions in BORM: `borm:InternalFunction` and `borm:ExternalFunction` (disjoint, union).
- ◦ `borm:Scenario` supports a certain function a business and serves as a container for related processes.
- ◦ `borm:Process` represents a single object-relation diagram, i.e. description of a business process that is part of a certain scenario.

All of these classes require standard `rdfs:label` for a human-readable name. Use of `rdfs:comment` is recommended for a brief explanation. There are three object properties for relating entities `borm:hasFunction`, `borm:hasScenario`, and `borm:hasProcess` (in the ontology with the corresponding domain and range). As inverse relations, `borm:ofBusiness`, `borm:ofFunction`, and `borm:ofScenario` are defined. The semantics of the properties of

BORM are different from generic part-of taxonomies, which is the reason for not using existing as it is done for labels. For expressing the relations between scenarios, the corresponding object properties are included: `borm:usesScenario`, `borm:extendsScenario`, and `borm:followsScenario`.

We demonstrate the use of OntoBORM and model serialisation using the example E-Shop case from the Craft.CASE tool [161]. Listing 5.8 shows a fragment of RDF for the BA model.

Listing 5.8: BORM BA example in RDF

```
@prefix borm: <https://purl.org/nsgo4cm/cm-ontology/borm#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix : <http://example.com/model/borm#> .

:myEShop a borm:Business ;
         rdfs:label "My E-Shop Example" ;
         borm:hasFunction :f1, :f2, :f3, :f4 ;

:f1 a borm:InternalFunction ;
    rdfs:label "Sell Food" ;
    rdfs:comment "..." .

:sce2 a borm:Scenario ;
      borm:usesScenario :sce1 ;
      borm:followedByScenario :sce3 ;
      borm:ofFunction :fun1 ;
      borm:hasIntitiation :init21 ;
      rdfs:label "schedule delivery" ;
      rdfs:comment "..." .
```

5.6.3 Representing BORM OR in RDF

The OR part of the ontology is more complex than the business architecture. First, there are five classes to express participants in processes and their roles (flows):

- `borm:Participant` represents a type of stakeholder that may participate in processes. There are three subclasses (disjoint, non-union) inspired by OpenPonk and OpenCABE: `borm:Person`, `borm:System`, `borm:Organization`.
- `borm:Role` is used to relate a participant with a process, i.e. a participant has a role in a process. This could be seen as an object property; however, we need to specify additional details and have a reference to distinguish the different roles of the same participant in multiple processes.

- `borm:State` represents a state within a role. There are two special subclasses for the start and end state.

- `borm:Activity` represents an activity within a role that serves for the transition between two different states (which are within the same role).

- `borm:Transition` is used to express the transition between two states via activity. It is not done through an object relation for the same reasons as role – there is additional information required for its instances.

Again, well-known `rdfs:label` is required and `rdfs:comment` are recommended for human-readability. The union of classes `borm:State` and `borm:Activity` forms the class `borm:RoleElement` which has the property `borm:ofRole`. A role uses object properties `borm:startsWith` and `borm:endsWith` with an corresponding state subclass instances. A transition can be related to states using `borm:sourceState` and `borm:targetState`, and with an activity by `borm:transitsThrough`. Then, a participant is related to its role by `borm:hasRole`, and to its role in the process by `borm:ofProcess`. The last part is to capture relations between roles and related constraints:

- `borm:Communication` represents a links between activities of different roles within the same process. It has two subclasses (union and disjoint) for synchronous and asynchronous communication.

- `borm:DataFlow` can be attached to a communication. It is intended to be used by domain-specific class; e.g. `Order` can be `borm:DataFlow`.

- `borm:Constraint` can be used for both communication and transition and specifies a condition as in BORM that results in two subclasses (disjoint, union). It should be described by text, but it can be related to domain-specific entity similarly to data flow.

A communication is related to activities by object properties `borm:sourceActivity` and `borm:targetActivity`. To distinguish the directionality of data flows, there are two sub-properties of `borm:hasDataFlow` – `borm:hasInputFlow` and `borm:hasOutputFlow`. The object property `borm:ofCommunication` forms a link between a communication constraint and a communication, and correspondingly for transitions (`borm:ofTransition`). To support sub-process (more specifically a process flow within a state), a `borm:State` is a subclass of that abstracts the common properties, i.e. the ability to contain a process flow (having role elements, starting and ending states).

Again, we demonstrate the use of OntoBORM and model serialisation using a selected OR from the example E-Shop case from the Craft.CASE; fragment of the example is shown in Listing 5.8 shows a fragment of RDF for the BA model.

Listing 5.9: BORM OR example in RDF

```
@prefix borm: <https://purl.org/nsgo4cm/cm-ontology/borm#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix : <http://example.com/model/borm#> .

:ro1 a borm:Role ;
     borm:startsWith :st1 ;
     borm:endsWith :st4 .

:st1 a borm:StartState ;
     borm:ofRole :ro1 .

:act1 a borm:Activity ;
      borm:ofRole :ro1 ;
      rdfs:label "opens website" .

:tr1 a borm:Transition ;
     borm:sourceState :st1 ;
     borm:targetState :st2 ;
     borm:transitsThrough :act1 .

:df1 a borm:DataFlow ;
     rdfs:label "Login data" .

:co1 a borm:SynchronousCommunication ;
     borm:sourceActivity :act1 ;
     borm:targetActivity :act4 ;
     borm:hasInputFlow :df1 .
```

5.6.4 BORM Designed for Semantic Integration

We tried to transform a UML Class Diagram to OWL according to [6] and link it with an overlapping BORM model in RDF/OWL. An example of this approach is shown in Listing 5.10. Then, we also experimented with additional ontologies. In the integration, some of the classes from UML were identical to the participants and entities in the data flow in BORM. This mapping could be done even semi-automatically with the use of natural language processing (NLP) methods.

The ultimate goal of promoting interoperability for BORM models has been achieved and demonstrated in the previous section. All entities in the RDF representation of a BORM model may have identifiers through which can be referenced. These references are internal; for instance, a participant in multiple processes can be identified as one entity. However, it can also be external; one may add more statements about an entity, as shown for participants and particular people. Finally, having BORM in RDF allows various tooling designs for RDF and OWL.

The semantic interoperability allows creating a complete description of a domain (e.g.

Listing 5.10: Linked BORM model

```
@prefix borm: <https://purl.org/nsgo4cm/cm-ontology/borm#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix m2: <http://example.com/model/uml#> .
@prefix : <http://example.com/model/borm#> .

:SalesPerson a borm:Person, owl:Class .

<https://orcid.org/0000-0001-7525-9218> a borm:Person, m2:Human, foaf:Person ;
                                        rdfs:label "John Walker" ;
                                        foaf:name "John Walker" ;
                                        m2:firstName "John" ;
                                        m2:lastName "John" ;
                                        borm:hasRole :role01 .
```

a business organisation) by integrating knowledge captured using different methods and focusing on various aspects. In this sense, the BORM models can be integrated with structural information about business and business rules. With the integrated knowledge, the analysis and optimisation will be more efficient when compared to separated and inconsistent models that need to be managed in different tools.

5.6.5 Using SPARQL and SHACL for BORM

Our contribution in the form of OntoBORM allows to use various of RDF technologies for BORM models such as SPARQL or SHACL constraints. SPARQL as part of the Semantic Web stack [162] can be used to efficiently query information from a BORM model in RDF. A business analyst can have several queries prepared and execute them with different models. Helpful queries might be queries that count certain elements or patterns in the model, for example, the number of activities for each participant in all processes, as shown in Listing 5.11. Also interesting are `ASK` queries that yield true or false results, for instance, the number of communications between two participants, as shown in Listing 5.12. It can be even used to validate the model; a more complex query can ask if it is valid (does not violate any of the five constraints stated in this paper). `DESCRIBE` can help to find out more information about a particular entity in a model, especially if it is linked with other RDF data. Finally, it can be used for `CONSTRUCT` new statements from a model.

5.7 ORM Models in RDF/OWL

Object-Role Modeling (ORM), more specifically ORM2, is a representative of fact-based modelling. It focuses on semantics and treats implementation concerns as irrelevant. All elementary facts (facts that cannot be further simplified) are captured using relationships, Listing 5.11: SPARQL query to count activities for each participant

```
# ... PREFIXes
SELECT ?p, ?p_label, (count(distinct ?a) as ?cnt)
WHERE {
  ?a rdf:type borm:Activity .
  ?a borm:ofRole ?r .
  ?r borm:ofParticipant ?p .
  ?p rdfs:label ?p_label .
} ORDER BY DESC(?cnt)
```

Listing 5.12: SPARQL query checking communication from `p1` towards `p2`

```
# ... PREFIXes
ASK {
  :p1 borm:hasRole ?r1 .
  ?r1 borm:hasElement ?e1 .
  :p2 borm:hasRole ?r2 .
  ?r2 borm:hasElement ?e2 .
  ?c a borm:Communication .
  ?c borm:sourceActivity ?e1 .
  ?c borm:targetActivity ?e2 .
}
```

including relations to attributes. For example, "Customer has name" is a fact modelled using entities related by a binary predicate. In this sense, the ORM is already close to RDF.

Although ORM is not as widespread as BPMN or UML, previous works relate ORM with OWL. Hodrob and Jarrar [132] mapped and designed the transformation of ORM models into OWL 2 ontologies. They cover 22 out of 29 ORM constructs using $\mathcal{SHOIN}$ description logic and the DogmaModeler tool. The mapping focuses only on transformation from ORM to OWL, e.g. entities to classes or predicates to properties. It does not maintain any upper ORM-specific ontology that would be used to describe the ORM model in RDF/OWL.

Another work by Franconi, Mosca, and Solomakhin [163] also does not propose ORM2 metamodel ontology and deals with mapping on the level of description logic, in this case, $\mathcal{ALCQI}$ description logic. The main goal seems to be the ability to use OWL2 reasoners (such as HermiT and FaCT++) for ORM2 models. Both works are essential fragments in our knowledge base; however, we need to design an ORM2 ontology for our use case similarly to UML, BPMN, and BORM.

5.7.1 ORM2 Ontology

Due to the similarities between OWL2 and RDF, the ontology for our purpose can be very straightforward. The only complexity can be seen with relation to dealing with predicate arity and related constraints:

- `orm2:EntityType` represents an entity with a name and optional code. We model it directly as a subclass of `owl:Class`. Similarly, the subtype object property `orm2:subtypeOf` is a subproperty of `rdfs:subClassOf`.
- `orm2:ValueType` represents a named value; in other modelling languages, it would be similar to attributes. In our design, we added an optional relation to a data type.
- `orm2:ObjectType` is a union for entity and value types. It abstracts the common properties (e.g. name) and also allows us to capture the independence, i.e. that instances of the type may exist, without being used any roles.
- `orm2:Predicate` represents a relationship between entities and value types with its name. It contains roles based on the arity; we do not distinguish arity by specific subtypes, as it can be derived from a number of attached roles. Finally, a set of constraints may be related to a predicate.
- `orm2:Role` is (optionally named and ordered) a role of an entity or value type in a predicate. It may also be a subject of a constraint. There is a `orm2:MandatoryRole` to specify a simple mandatory role.
- `orm2:Constraint` represents generic constraints and is further specialised in the hierarchy since there are various kinds of constraints in ORM2. For example, some constraints are related only to a predicate as a whole, others to some of the roles for a single predicate, and constraints across more predicates. Moreover, there are even constraints for entity and value types (e.g. to specify enumeration).

An simple example of a particular constraint is `orm2:ValueConstraint` that can be used to specify an enumeration of values (e.g. {`'Male','Female'`}) or ranges of values (e.g. {`0..10`}), there are two subclasses – `orm2:ObjectTypeValueConstraint` for entity and value types, and `orm2:RoleValueConstraint` for roles. Other constraints related to a single role, predicate, value type, or entity type, such as object cardinality, internal frequency, ring and deontic constraints, or internal uniqueness, are solved identically. Another category of constraints are those related to multiple predicates, roles, or subtyping relationships. For instance, there is `orm2:JoinSubsetConstraint` that relates multiple roles of different predicates to another predicate.

The last construct to be covered from ORM 2 is the so-called objectification. Our design allows us to solve this in RDF by having a single resource, that is, `orm2:Predicate` and `orm2:EntityType` at the same time. That exactly matches what objectification from ORM2 means: "The predicate (fact type) is objectified as an entity type whose instances can play a role.". With that, all the constructs of ORM2 are successfully captured in our OWL ontology for ORM models.

5.7.2 ORM2 in RDF Example

Listing 5.13 shows the use of our ORM2 ontology in practice to represent a simple ORM model in RDF. Customer subtypes an independent entity type Person. Then, we have a simple value type for e-mail addresses. Finally, a binary predicate relates customer and e-mail address types. An e-mail must be associated with a customer in this model. Also, the internal uniqueness constraint is used to form a one-to-many pattern so that a single customer may have multiple (unique) e-email addresses.

Listing 5.13: ORM2 example model in RDF

```
@prefix orm2: <https://purl.org/nsgo4cm/cm-ontology/orm2#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix : <http://example.com/model/orm#> .

:Person a orm2:EntityType ;
        rdfs:label "Person" ;
    orm2:isIndependent true .

:Customer a orm2:EntityType ;
          rdfs:label "Customer" ;
          orm2:subtypeOf :Person .

:Email a orm2:ValueType ;
       rdfs:label "E-mail address" .

:customerEmail a orm2:Predicate ;
       rdfs:label "has" .
       orm2:hasRoles [
         a orm2:Role ;
         orm2:roleTarget :Customer ;
       ] , [
         a orm2:MandatoryRole ;
         orm2:roleTarget :Email ;
         orm2:roleConstraint [
           a orm2:InternalUniquenessConstraint;
         ] ;
       ] .
```

5.8 Integrating Knowledge from Conceptual Models using RDF

Having conceptual models represented in RDF by using well-defined OWL ontologies such as those proposed by us (but also other existing, e.g. full BPMN 2.0 ontology [160]) enables semantic integration. There are multiple ways to integrate the information from different conceptual models at the RDF level. Some of them we have already tackled in our examples in the previous sections. One option is to have a mapping done on the level of metamodels and upper ontologies; for example, relations (such as equivalence) are defined between UML and ORM ontology. The mapping between modelling languages is itself a challenging task. We worked on a mapping between behavioural modelling languages (namely UML Activity Diagram, BPMN, and BORM) using upper ontology UFO-B [A.6].

To efficiently create a mapping between languages, using an upper ontology as the "glue" is another option. Based on various conceptual modelling languages, we proposed a minimal ontology with the essential constructs used in conceptual models [A.5]. The Ontology for Conceptual Models Integration (OCMI) is an upper ontology focused on the integration of various conceptual modelling languages, i.e. their metamodels represented as OWL or RDFS ontologies. It allows them to refer to more generic terms from OCMI rather than more specific concepts from language metamodels.

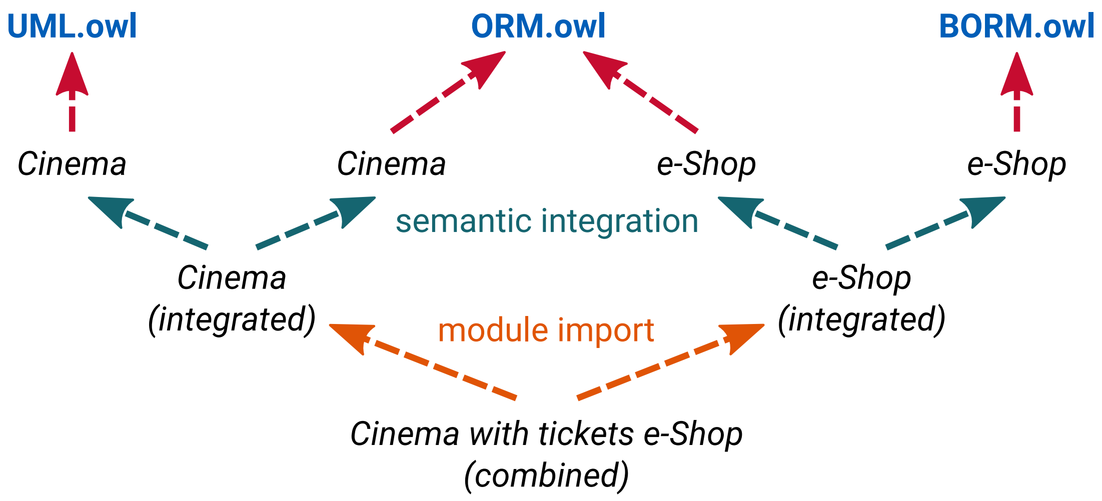

Figure 5.3: Cinema and e-Shop models example integration architecture

The other option is to relate various conceptual modelling languages within individuals representing the combined models. For instance, a customer as an individual in RDF can be of type `uml:Class`, `orm:EntityType`, and `bpmn:Pool`. It does not require defining any additional constructs or incorporating an upper ontology designed for integration. As a result, multiple models may be integrated on some individuals linking the different models, while still keeping knowledge related to a certain modelling language. For example, the customer will have some attributes described using the UML ontology, but this information is not related to the BPMN model. In this way, a representation RDF of integrated models of the same domain may provide a more complete view that is useful also for model-driven development purposes. The example in Figure 5.3 shows the concept of integrating models made using different modelling languages (i.e. ontologies) together with a combination of concepts to create richer models.

5.9 Modularity and Evolvability of Conceptual Models in RDF/OWL

In addition to the semantic integration of models captured using different modelling languages, another advantage is the modularity and evolvability of models in RDF/OWL. A single domain model can be split into multiple files and namespaces, just as various modelling languages allow diagrams and packages to be used. It gives absolute freedom in structuring the model – from having all knowledge about a complex domain in a single namespace to having a namespace for each fragment composed from just a few entities linked using imports. Such an approach is also shown in Figure 5.3 where two models are integrated and combined together to form a new extended model.

The evolvability of conceptual models is also promoted as not all CASE/CABE tools support model versioning and versioning of underlying project serialisation (XML with model data, metadata, data about graphical representation, and others) is not suitable for version control. With RDF/OWL and a well-structured project, versioning is directly supported and verified by practice. When a model is divided into modules by concerns, those can be versioned separately. As a result, similarly to Normalized Systems, instead of maintaining a product of all module versions, it is required to maintain only a sum of versions of the modules.

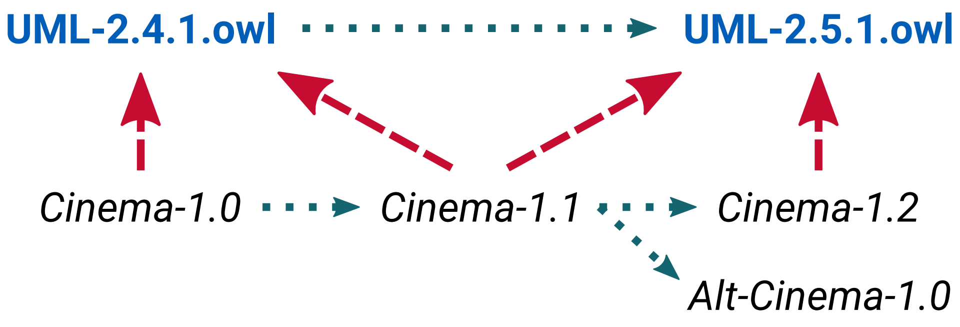

Figure 5.4: Evolution example for UML models

The ontologies for conceptual modelling languages may evolve over time, just as the specifications of their metamodels. To avoid unexpected breaking changes, the models encoded in RDF should use a version-specific prefix instead of redirecting to the latest version. Then, the ontologies for conceptual modelling should use the well-known semantic versioning, deprecations, and changelogs, to promote simple adoption of updates. Such an update of a model to a newer version of conceptual modelling ontology (e.g. a newer language metamodel) means changing the prefixed Uniform Resource Identifier (URI) for the ontology and making the necessary changes. With this approach, it is also possible to have a single RDF with a conceptual model that complies with multiple versions of the ontology. The example in Figure 5.4 shows the concept of evolvability of models and metamodels captured in RDF/OWL, its versioning, and branching.

5.10 Design Cycle of Representing Conceptual Models in RDF/OWL

In this chapter, we describe several ontologies that we designed and created. Each is itself a designed artefact according to Design Science Research (DSR) and has been done using design cycle iterations. For evaluation purposes, we used existing models and encoded them in RDF using ontologies. Then, we minimised the loss of conceptually necessary knowledge from the models. It would not be possible to assess appropriately without actually creating the OWL ontologies based on design and having models RDF. Again, the design implementation (in this case, RDF/OWL files – part of the Appendix) turned out to be valuable for applying DSR. During the cycles, we greatly benefited from the evolvability aspects mentioned in the previous section.

When the design cycle for individual ontologies stopped, we started to evaluate them together as an artefact collection to design enhancements for the semantic integration. Due to the properties of the RDF and OWL technologies for linked data and semantic integration, we practically just verified that our ontologies allow integration in a straightforward and standard way. Moreover, we also worked on mapping of behavioural modelling languages [A.6] and integration using a more generic ontology for conceptual modelling languages [A.5].

5.11 Summary of Representing Conceptual Models in RDF/OWL

In this chapter, we presented how to capture conceptual models in RDF and OWL to promote their interoperability and transform them using our NS Gateway Ontology for Conceptual Models. Although we had to develop new custom ontologies for most of the modelling languages, some were designed according to related work (e.g. UML and BPMN). We also demonstrated how to incorporate well-maintained ontologies representing metamodels for our purpose, such as gUFO. Based on our research goals, we prepared the ontology and way of models capturing in RDF for the following types of conceptual modelling and languages:

- structural modelling – UML (Class Diagram), OntoUML/gUFO (UFO-A);
- behavioural modelling – UML (Activity Diagram, State Machine Diagram), BPMN, BORM, gUFO (UFO-B);
- fact modelling – ORM.

With various conceptual models captured in RDF, we could also explain how to integrate them semantically. That fulfils FR4 of our design and thus also the objective RO1. The possibility of having different types of modelling "shifted" to RDF/OWL (as listed above) contributes to FR1, FR2, and FR3. We presented our OCMI that defines very generic concepts shared across different modelling languages in terms of semantic integration. Finally, we described the modularity and evolvability of the conceptual models captured in RDF. The designed artefacts are ready for use according to our general architecture described in Chapter 3.

Chapter 6

Transforming between Models using Gateway Ontology

"The art of programming is the art of organizing complexity."

Edsger W. Dijkstra

After introducing the transformation between Normalized Systems (NS) and RDF/OWL and explaining how to encode conceptual models from various modelling languages in RDF/OWL, the final part, according to our designed architecture, is the transformation in the Gateway plane. This chapter first clarifies the general principles and describes the internal layers of the Gateway Ontology design. Then, it explains the adapters for specific modelling languages by providing examples concerning the previous chapter.

Finally, it describes the transformation procedures based on the ontological specification. We created an implementation for verification purposes of our algorithms, which is also briefly introduced in this chapter. As several of our designed artefacts have been again described, their evolution during the design cycles (according to the DSR) is briefly summarised. The transformation using SPARQL Protocol and RDF Query Language (SPARQL) is based on our previous work on pattern-based SPARQL transformations [A.11] and SPARQL mappings in Resource Description Framework (RDF) [A.15].

6.1 Relating NS Elements and Conceptual Models

With the NS-RDF/OWL transformation presented in Chapter 4 and describing how to represent conceptual models using RDF in Chapter 5, both NS models and conceptual models can "speak" the same language – RDF. With that, we can proceed to the specification of mapping to transform knowledge between conceptual models and NS without dealing with incompatible formats or other syntax-related issues.

A conceptual modelling language is a broad term, and as explained in Chapter 2 and Chapter 3 modelling in NS can be itself considered as conceptual modelling. The mapping will state how the constructs of two modelling language metamodels M'_X are related, while we take the NS metamodel M'_{NS} as one of the mapped. The expected key difference is the focus on implementation in NS and a variety of aspects close to the real-world in other conceptual models. We relate the metamodels M'_X and M'_{NS} through a mapping $T_{X \to NS}$ that allows us then to transform or translate the underlying models $M_X \in \mathfrak{inst} M'_X$ to the corresponding NS models $M_{NS} \in \mathfrak{inst} M'_{NS}$. Due to the use of RDF for representations on M_X, M'_X, M_{NS}, and M'_{NS} (thus also M''_{NS}), it does not need to deal with a syntactic transformation, but rather purely semantic. As explained in Chapter 3, there can be $T^{-1}_{X \to NS}$ defined to enable bi-directional transformation (from $M_{NS} \in \mathfrak{inst} M'_{NS}$ to $M_X \in \mathfrak{inst} M'_X$).

Our way of defining the mappings, i.e. the "glue" between different metamodels and their compliant models, must be itself evolvable, FAIR, and mainly very flexible. It must allow specifying simple mappings such as a class in UML corresponds to a data element in NS, as well as complex mappings of nested patterns and relations. In this section, we briefly overview how the selected conceptual modelling languages are related to the NS metamodel – yet on the theoretical level without encoding in our solution. We focus on the key constructs in the description; complete mappings can be found in Appendix A. It will serve as a reference for specification in our framework using the Gateway Ontology. In addition, it provides essential knowledge of the types of mappings needed using the bottom-up approach.

6.1.1 UML Class Diagram Mapping

As a Unified Modeling Language (UML) class diagram is suitable and widely used for structural conceptual modelling, the mapping is expected to cover the branch of the NS metamodel related to data elements. The mapping will also be very similar to our previous work on mapping and transformations between NS and Ecore [A.12] because the Ecore is a de-facto UML subset related to the class diagram constructs.

Table 6.1 presents the core of the mapping. The main and most straightforward match is identified with UML Package and NS Component and UML Class and NS Data Element (including its basic properties, such as name or description). Despite the fact that UML uses stereotypes for classes that cannot be directly mapped to data element types. There are two pre-defined stereotypes, namely enumeration and datatype, that create exceptions in the mapping of classes. The Enumeration construct is mapped to the Taxonomy Data Element with name and value fields, as is common in NS models. The Datatype construct

Table 6.1: Mapping between UML Class Diagram and NS

UML (Class Diagram)	**NS Elements**
Package	Component
Class	Data Element
Enumeration (Class)	Taxonomy Data Element + Value Fields[4]
Datatype (Class)[1]	Value Field Type[6]
Attribute	Value Field
Derived Attribute	Calculated Field
Operation[1]	Calculated Field
Association	Link Field(s)[2]
Aggregation	Link Field(s)[2] + Data Child
Composition	Link Field(s)[2] + Data Child
Generalisation	Link Fields[5]
Dependency[1]	Link Field
Implementation[1]	Link Field
Association Class	Data Element + Link Field(s)[3]
N-ary Association	Data Element + Link Field(s)[3]
* (all used constructs)	Component, Data, or Field Options

[1] rare in conceptual models
[2] using multiplicity, direction, and role names
[3] with every participating data element, "star topology"
[4] traditional *name* and *value* fields
[5] always bi-directional, link field type based on abstract superclass
[6] with mapping to common types in NS

is a counterpart to Value Field Type, and we create an extra mapping for a common datatype with a fallback to string.

For a class, attributes are assigned to value fields, and associations are assigned to link fields. If an attribute is derived, then it is actually a calculated field. A calculated field is also used to represent operations; however, the use of operations in UML conceptual models is relatively rare. The multiplicities of both attributes and associations affect the type of the mapped field (e.g. if the link field will be many-to-many). For association, direction and role names also affect if the link field will be on both sides of the relationship and the field name. The particular types of associations – aggregations and compositions – use the exact mapping as associations; moreover, data child is added to represent the part-whole relationship. Similarly, other special relationships, such as generalisation, dependency, or implementation, are mapped as associations, but with additional field options to keep the semantics. Suppose that there are more complex associations, e.g. n-ary or association class. In that case, the association is itself a data element forming a "star topology" since there are only binary relations in NS.

Finally, the mapping shows the use of a component, a field, and data options to capture additional semantics that currently do not have a direct counterpart in NS. Due to the use of keeping consistency through RDF, we could omit that, but the options can be used for the expansions of NS. For example, with a field option that states that a link field realises a generalisation relation, an expander can be prototyped to generate the link differently, according to our inheritance implementation patterns [A.3]. This way enables such prototyping and eventual adoption of certain constructs from conceptual modelling languages in NS. Options capture origin of all of the mapped constructs even with a direct match, e.g. a data element will have a data option that it was originally a UML class. The same principle is also applied to the other mappings described in this chapter.

The mapping with the UML class diagram allows the transformation of most of the commonly used constructs in conceptual modelling. For classes and inheritance, we also considered mapping to a single data element for a hierarchy with data projections per each class in the hierarchy; however, that would not be desirable in all cases. The user could then decide on the transformation (if such a user-intervention mechanism is implemented). The only construct that is often used in conceptual modelling but is not covered are constraints in general. UML allows to specify constraints (and stereotypes) almost for all constructs and it can be in form of Object Constraint Language (OCL) expression but also natural text. The only way to map the constraints to NS is again via options to the affected mapped construct, e.g. keep a class stereotype as a data option related to the corresponding data element.

6.1.2 UML Activity Diagram Mapping

UML activity diagrams contain knowledge about processes in a certain domain. When compared to the possibilities of capturing behaviour in NS, NS is simpler and focuses on flows of a single data element (with a possible reference to remote tasks). The identification of relations between process flows in UML and flow elements related to data elements in NS is essential.

As captured in Table 6.2, each identified process flow (usually an activity diagram has one flow) is mapped to a flow element; alternatively, multiple flow elements per each participating entity that is mapped to a data element. The relation to an entity is derived from swim-lanes (or so-called partitions) and attached objects, possibly with specified states, using an object flow. Then, all the constructs connectable using the control flow are mapped to related tasks and states of NS as parts of the corresponding flow element.

Because UML activity diagrams do not have a direct concept of a state that is, on the other hand, necessary in NS, both a control flow construct is always mapped to both task and state after execution of the task. The only differences are start/end events and the decision and fork constructs that are mapped to branching tasks, possibly branching to more states. However, the actual branching mechanism is performed in implementation and is not further specified in NS models. Therefore, again the options are used to capture such additional semantics usable by expanders and implementation (source code) of the resulting software system.

Table 6.2: Mapping between UML Activity Diagram and NS

UML (Activity Diagram)	**NS Elements**
Partition / Swimline	Flow Element[1]
Start Event	State
End Event	State
Activity / Action	Task + State[2]
Send	Task + State[2]
Receive	Task + State[2]
Decision	Branching Task + State(s)
Merge	Task + State[2]
Fork	Branching Task + State(s)
Join	Task + State[2]
Object (with State)	Flow Element[1]
Structured Activity / Subprocess	*decompose*
Control Flow	Transition
Object Flow	-[1]
Interrupt Flow	Failed Transition
* (all used constructs)	Options

[1] process flow split to NS flows based on the participating entity (data element)
[2] subsequent state after executing the task (including intermediary and fail state if necessary)

The control flow itself is mapped to transitions between task-state patterns in NS. For interrupt flow, we identified a direct NS counterpart – failed transition that branches from a failed task to another state than on success. Finally, UML activity diagrams allow composition (sub-processes) using structured activities. Such composition is not possible within NS; therefore, such nested processes must be decomposed into the top-level flow prior to transformation.

6.1.3 UML State Machine Diagram Mapping

When compared to the previous UML activity diagram mapping, the match between the state machine diagram constructs and NS is significantly better. Just as NS uses flow for a data element, there might be a state machine for a certain class in UML, i.e. the flow is bound to its entity exclusively and entirely. That avoids the complexity of transformation related to checking of to flow (and data element) is a process construct related. Furthermore, even the flow constructs are semantically closer to NS than we have seen in the case of the activity diagram.

Table 6.3 clearly shows that the mapping is rather straightforward. Each state machine becomes a flow element. Then every state of a state machine diagram is mapped to a state

Table 6.3: Mapping between UML State Machine Diagram and NS

UML (State Machine)	**NS Elements**
State Machine	Flow Element
State	State
State Composition	*decompose*
Initial (pseudostate)	State
Final (pseudostate)	State[1]
Terminate (pseudostate)	Task + State
Fork (pseudostate)	Task + State
Choice (pseudostate)	Task + State
Join (pseudostate)	Task + State
Entry Point (pseudostate)	*decompose*
Exit Point (pseudostate)	*decompose*
Junction (pseudostate)	Task + State
History (pseudostate)	Task + State + History Data Element[2]
Transition	Task
* (all used constructs)	Options

[1] implicit if follows directly other pseudostate
[2] additional history data element and relate using a link field

of the corresponding flow. The transitions between states are mapped to tasks (as that is the only way to relate two states in NS flows). As with activity diagrams, any potential composition done using the state composition and pseudostates "extry point" and "exit point" must be decomposed into the top-level flow. Then, there are more pseudostates that are similarly mapped again as for the activity diagram – fork, choice, and join – that are related to branching tasks. Similarly, as pseudostate "terminate" and "junction" are expected to have additional implementation (execute some action), those are mapped to task with state. Finally, the pseudostate "history" is also mapped to task with state, but also creates a history data element attached to the primary data element of the flow. That history data element is the container for storing the captured information or snapshot when the primary data element is in the pseudostate (implemented in the mapped task).

As the mapping for state machine diagram has better coverage than for activity diagram (due to UML-NS alignment), we can recommend to prefer this type of diagram (if choice is possible). On the other hand, the alignment is caused by the fact that the state machine diagram is closer to implementation than the activity diagram in general terms. It is often used to design a software system, whereas activity diagram is traditionally used for domain analysis. However, state machine diagram is also an important source of domain knowledge, e.g. state of order in an e-Commerce domain.

Table 6.4: Mapping between UML Class Diagram and NS

OntoUML (UML Profile)	**NS Elements**
Class: Powertype / HOU	Taxonomy Data Element
Phase Partition	Data State(s) + Value Field
Class: Mode	History Data Element + Link Field
Class: Quality	Value Field
Association: Formal	*(nothing)*[1]
Association: Material	*(nothing)*[1]
Generalization	Data Element + Data Projections[2]
* (UML Class diagram mapping)	*

[1] derived associations should not form link fields
[2] merge hierarchy "up" to identity provider and add data projections for all sortals

6.1.4 OntoUML Mapping

We map OntoUML as a UML profile (not a fragment of the domain ontology specified using Unified Foundational Ontology (UFO)). The main reason is that our goal is conceptual modelling languages. Furthermore, this allows us to show reusability across mappings that is natural for such cases as are UML profiles. Still, it can further serve as a foundation to define the mapping between UFO and NS. The OntoUML mapping itself is then just a simple extension to the one we presented for UML class diagram.

The additional mapping and differences (or "overrides") are captured in Table 6.4). Besides keeping stereotypes of classes and associations encoded in the data and field options, we adjust the mapping based on these stereotypes from the OntoUML model. For example, a power-type or "HOU" (higher-order universal) is mapped to a taxonomy-typed data element, or phase partitions are mapped to data states. The special constraints defined by OntoUML, such as *essential* and *inseparable* part-whole relationships, are captured through field options.

Finally, the different notion of generalisation relations in OntoUML is captured by the use of data projections. Generalisations with non-sortals are just as for UML; however, we merge a hierarchy of sortals to the single data element (corresponding to the identity provider class of the sortals hierarchy; there is always exactly one). Then, each subclass has a corresponding data projection with its fields (through reference fields). The separation of sortals and non-sortals together with the identity provider enables to use this way of mapping inheritance that is not possible for plain UML class diagrams.

As explained, the OntoUML mapping provides additional improvements to UML class diagram mapping. However, most of the stereotypes and constraints given by OntoUML constructs are mapped to options – would need extensions to expanders for automated implementation in the corresponding software system. Eventually, when the NS metamodel is extended to increase the adaptability of such constraints, the mapping could be extended based on the previous work of Rybola [27]. Alternatively, it can serve as a basis for the work on expanders, i.e. generating code based on stereotypes and their mapping to source code.

6.1.5 BPMN Mapping

The mapping of Business Process Model and Notation (BPMN) is again related to tasks and flows in NS as it is a process modelling language. We can observe certain similarities with the UML activity diagram mapping as we focus here on the BPMN collaboration diagram that captures the process flows of potentially multiple participants with possible branching, related to data objects, or nested process flows. Although conversation and choreography diagrams may provide additional domain knowledge, we do not cover them in the mapping, as they are not in our BPMN ontology nor have direct counterparts in the NS metamodel.

In a collaboration diagram, one or more flow elements may be found based on related entities (that are mapped to data elements). The relation to an entity is done using white-box pools and its lanes. Black-box pools, similar to data objects and data stores, are mapped to just related entities; therefore, a link field should be linked with the corresponding data element(s). As described in Table 6.5, tasks, transactions, and activities are (as in UML activity diagram) mapped to task and subsequent state. BPMN allows specifying a type or marker for activities and tasks, e.g. manual task, which is kept as option for consistency as well as manual task type in NS.

Events (e.g. start timer event or intermediary message event) are also mapped to task and state; however, the mapping is affected by the event types. For example, a “plain” event is mapped just to a state, but any message event is mapped to a task (sending or receiving the message) and subsequent state. If it is also a start message event, then the mapping also adds the state before the task (as the flow must start with the state in NS). Gateways, as in UML mapping, are branching tasks with subsequent states. That also covers the mapping of default and conditional flows related to branching. Both types of gateways and events are captured as options for possible customisation in code expanders.

The transitions are mapped from normal sequence flows. The subprocesses must be again decomposed into the top-level flow. If the subprocess is allowed to be executed multiple times (sequentially or in parallel), a branching pattern with a possible loop must be added. Finally, the message flow represents communication between tasks of different data elements (implementation in the code of the affected tasks). Similarly, the sequence flow crossing boundary of lanes is mapped as communication because each lane is mapped to a separate data element.

6.1.6 BORM Mapping

Business Object Relationship Modelling (BORM) just as the previous BPMN is a representative of behavioural or process modelling and as such we can observe certain similarities between these two. However, the modelling approach is a bit different and, because

Table 6.5: Mapping between BPMN and NS

BPMN (Collaboration Diagram)	**NS Elements**
Pool (Black Box)	-
Pool (White Box)	Flow Element(s)[1]
Lane	Flow Element(s)[1]
Task / Transaction / Activity	Task + State
Task Type / Activity Marker	Task Type
Gateway	Branching Task + State(s)
Event	(Task +) state(s)
Sequence Flow[3]	Transition
Message Flow	Link Field[4]
Data Store / Object / Association	-[1]
Subprocess	*(decompose)*
* (all used constructs)	Options

[1] each data element per lane in pool with corresponding flow element
[2] based on event type
[3] including conditional and default flows
[4] ensuring relation for communication

BORM is based on the communicating state machine, it also shares some concepts with UML state machine diagram with respect to mapping towards NS. Aside from previously explained Business Architecture (BA) and Objects Relations (OR) models, we included an extension to BORM that some computer-aided software engineering (CASE) tools (e.g. Craft.CASE) provide – modelling of classes and attributes. Again, some concepts are matching with UML class diagrams; however, the BORM extension provides significantly simplified structural modelling.

The core of the BORM-NS mapping in Table 6.6 shows that just as in the BPMN and UML activity diagram participants (roles) within a process are mapped to data elements to which the flow is attached. However, in the case of BORM, there might be system-like roles that only provide or expose certain functionality as activities without flows. The data element or the flow element should not be created for that case. The states from BORM are mapped directly to the states in NS, including the starting and ending states. Activities between states are called tasks in NS while transitions are also identical. The exception transition from BORM is mapped to the failed transition. The nested flow that may be "inside" a state must be decomposed again into the top-level flow.

In BORM, the branching may appear after a state; therefore, it must be mapped to a branching task with states (again, notice the similarity with previous mappings of process modelling languages). The actual communication must be solved in the implementation, as there is no corresponding construct in NS. Still, a relation between data elements representing communicating participants must be established. A potential data flow attached

Table 6.6: Mapping between BORM and NS

BORM	NS Elements
Participant (Role)	Flow Element[1]
Activity	Task
State	State
Subprocess (in State)	*(decompose)*
State (with branching)	Branching Task + State(s)[2]
Transition Start	Transition
Transition End	Transition
Exception	Failed Transition
Communication	Link Field[3]
Data Flow	Data Element[4]
Class	Data Element[4]
Variable (in Class)	Value/Link Field
* (all used constructs)	Options

[1] flow element related to corresponding data element, not for system-like participants
[2] unlike BORM, NS allows branching from tasks (activities), not from states
[3] ensuring relation for communication
[4] ensuring existence of container for data passed in communications

to communication also results in a data element. Finally, the appendix contains classes (data elements) with variables (value and link fields).

After designing this mapping, we can state that BORM is the closest conceptual-oriented process modelling language to flows and tasks in NS. The main difference lies in the possibility of communication between roles through their activities (tasks). Also, the mapping of classes and their variables is more straightforward than UML due to its simplicity.

6.1.7 ORM Mapping

Object-Role Modeling (ORM) represents fact-based conceptual modelling in our mapping scenario. Despite the conceptually significant differences between facts and structural concepts such as classes, we found out that for mapping with NS these notions disappear, and the mapping is relatively close to mapping with the UML class diagram. If we would select Entity-Relationship (ER) or Enhanced Entity–Relationship (EER), the mapping would be identical in the main concepts, which is also given by the close relation of ORM and ER. Thus, the ORM constructs are mapped to the data element part of the NS metamodel as shown in Table 6.7.

The entity types of ORM are mapped to the data elements, which is the highest level of mapped constructs; therefore, a single component must be created for a whole ORM

Table 6.7: Mapping between ORM and NS

ORM	**NS Elements**
Entity Type	Data Element
Value Type	Value Field Type[1]
Reference Mode (of Entity Type)	Value Field
Independent Object Type	Data Option
External Object Type	Data Option
Unary Fact Type	Value Field (boolean flag)
Binary Fact Type (to Value Type)	Value Field
Binary Fact Type (to Entity Type)	Link Field
N-ary Fact Type (N ¿ 2)	Data Element + Link/Value Fields[2]
Objectification	Data Element + Value Field[3]
Internal Frequency Constraint	Link Field Type (of Link Field)
Subtyping	Link Field
Subtyping Constraints	Link Field Type + Data/Field Options
* (all used constructs)	Options

[1] with mapping to common types in NS
[2] using the same rules as binary fact types for each link in "star topology" (similar to the UML mapping) [3] source fact must be data element even if binary or unary

model (there are no packages or modules). Then, value types are mapped to value field types similarly to datatypes in UML – we also provide a mapping of commonly used types. Although ORM allows distinguishing independent and external types, it is just informative in the scope of NS and captures in data options.

The core of ORM are fact types. Unary fact types and binary fact types to a value type are mapped to value fields where unary facts form a boolean flag. Then, binary fact types to another entity type are link fields where internal frequency constraint affects its type (e.g. one-to-many or many-to-many). For other n-ary fact types, a data element with link fields to all joined types is necessary – forming a "star topology" similarly to n-ary associations in UML. Whenever a fact type is used with objectification, it must be turned into a data element as well, even if it is just a binary or unary fact type. The subtyping relation between types is just as a generalisation in UML mapped to the link field with type and additional options based on related ORM constraints. The complex constraints of ORM (except multiplicity and uniqueness) do not have direct counterparts among the NS constructs and are captured using the corresponding data and field options.

6.1.8 Unmatched Constructs and Consistency

As explained for each language mapping, there are certain gaps for both sides, i.e. there is no direct counterpart in the output metamodel that would allow capturing of specific knowledge from the input. We used options for the directions towards NS where possible; however, it is not complete and does not work for the other direction. For example, if we have a complex NS model with constructs such as data children, trigger or connector elements, or UI-related attributes of a value field.

The question remains how to capture these unmatched constructs to maintain consistency. On the general level of mapping, we could specify an appendix to the output model that would encode all the additional knowledge linked to the construct. When using NS, it is a trivial task since the input model, the output model, and the potential appendix may be considered as a single set of triples $\{t_i\}$.

$$T_{X \to NS}\left(\{t_{X,i}\}\right) = \{t_{X,i}\} \cup \{t_{NS,j}\} \tag{6.1}$$

The set of triples can be extended according to the mappings above for an input model. All input triples $\{t_{X,i}\}$ will remain the same; thus, $\{t_{NS,j}\}$ serves as the complete appendix to maintain consistency. For example, if a dataset contains UML and transformed NS models, the NS part can be turned in NS-in-XML, edited, and then plugged back next to the related UML. To allow full linkage even to the XML representation of an NS model, the resource URI must be attached to the NS constructs as its option. Then it can be used to update or check the consistency between the original input model and the NS model (and vice versa).

$$T^{-1}_{X \to NS}\left(\{t_{NS,i}\}\right) \subseteq \{t_{X,i}\} \tag{6.2}$$

$$\{t_{X,i}\} \cup T^{-1}_{X \to NS}\left(\{t_{NS,i}\}\right) = \{t_{X,i}\} \cup T_{X \leftarrow NS}\left(\{t_{NS,i}\}\right) \overset{?}{=} T_{X \to NS}\left(\{t_{X,i}\}\right) \tag{6.3}$$

To promote transparency of the operations over triples, a context may be provided by turning them into quads (named subgraphs). The metadata about a subgraph may then capture what triples were generated, when, by whom, or with what version of a transformation tool. As will be explained in more detail and shown with the design of actual transformation execution, the use of RDF allows maintaining consistency while allowing evolvability.

The primary part of this dissertation thesis is the design of the Gateway Ontology for transformations between NS and conceptual models. As outlined in Chapter 3, the term “Gateway Ontology” indicates that it is a set of concepts to define how conceptual models can be transformed into NS while keeping track of a possible way back. This section describes the design of the Gateway Ontology and related features. The actual concepts of the ontology are further described concerning the transformation execution and mapping of conceptual languages.

6.1.9 Design and Features

The Gateway Ontology that we design as part of our solution according to the decisions clarified in Chapter 3 using RDF and Web Ontology Language (OWL). However, the design is not technology-specific and could be implemented with other technologies that would also allow the representation of NS and various conceptual models. However, the use of RDF and OWL is sufficient for fulfilling the requirement NR7. The ontology itself encompasses several concerns:

1. manage interface to the NS modelling via its metamodel,
2. allow to extend the metamodel with concepts from modelling languages to promote DRYness (NR2 and NR4),
3. provide vocabulary for mapping conceptual modelling languages (and their metamodels) to the interfaced NS metamodel,
4. enable realisation of mappings based on Section 6.1 without any restriction to modelling type and integration of models (FR1–FR4).

Therefore, we divide these into layers while following the NS design principles to achieve evolvability (FR5 and NR1). The layered approach for the Gateway Ontology is depicted in Figure 6.1. Each layer is described in more detail in the subsequent parts of this section.

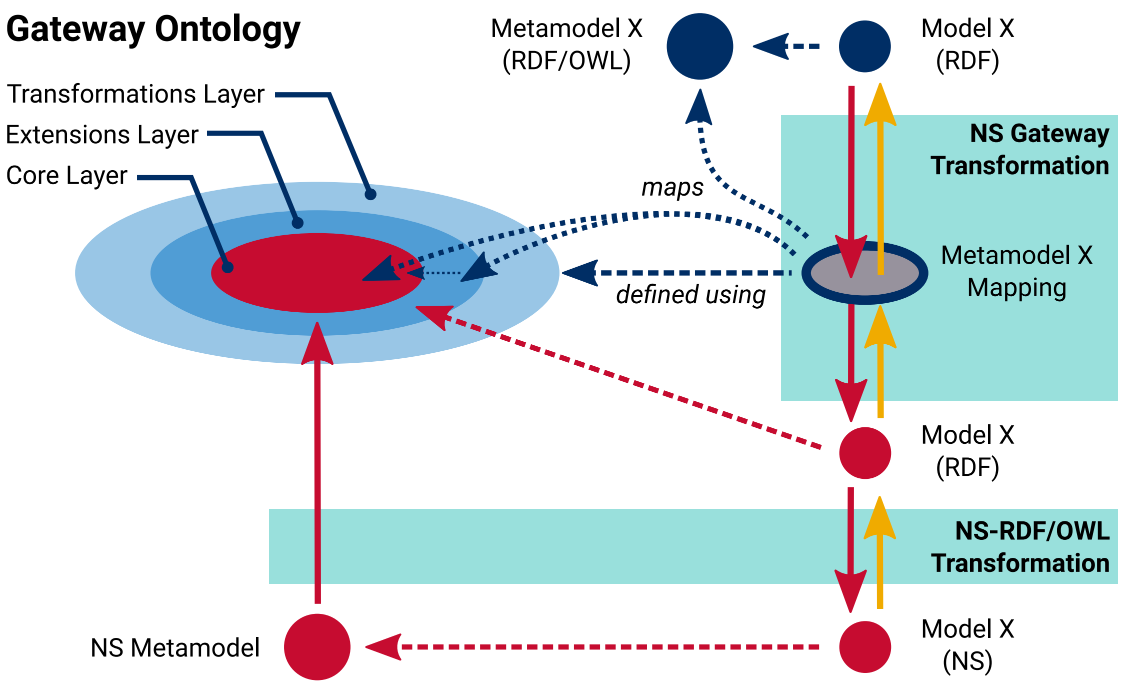

Figure 6.1: Design of the Gateway Ontology layers and transformation

The essential aspect of the layered Gateway Ontology is the strict separation of layers and relating them using the exposed "interface". In terms of RDF, each layer forms its own dataset with a namespace and is able to version independently of the others. Then, a layer may import the inner layer(s) to use the provided constructs for defining new. Finally, the most outer layer where separate mappings of conceptual modelling languages reside uses the constructs to encode the mapping (e.g. as presented in Section 6.1).

Similarly to splitting the solution into more parts, we applied modularisation yet again. Just as software systems can be composed of components and then modules and submodules, our framework has module gateway ontology layers. With that, we again achieve the advantage of *sum vs product* – each layer is maintained and versioned independently, and a composition is formed by imports (dependencies). A layer can specify which versions of the used layers are compatible if needed.

6.2 Gateway Ontology

The entire Gateway Ontology (all its layers) is designed by applying best practices from ontology engineering, especially with a focus on FAIRness (NR3). Appropriate metadata are always present in RDF directly, and the design allows the use of persistent unique identifiers such as pURL or W3ID. The documentation (NR6) can be generated as follows best practices, new concepts are described directly in RDF in a standard way, e.g. with `rdfs:label` or `rdfs:comment`.

6.2.1 Core Layer

The core layer of the Gateway Ontology is nothing else than RDF/OWL representation of the NS Elements metamodel created using our NS-RDF/OWL tool introduced in Chapter 4. It provides reference to all the constructs in the NS metamodel both for models construction (e.g. that construct `Person` is of type `DataElement`) and for specification of mapping with other metamodels (e.g. that `Class` from UML is mapped to `DataElement`). This is the only concern of the core layer; it does not add any additional constructs or other content to the RDF data.

The version of this layer is always identical to the version of the NS metamodel, i.e. version of the *Elements* component. As its name suggests, the core layer does not have any dependencies or other relations with different layers and parts of the Gateway Ontology. The only references are well-known vocabularies and ontologies, such as RDF, RDFS, XSD, or OWL – used via imports in the RDF representation. These are considered static; however, they can be updated in this layer if necessary without causing any combinational effect.

Whenever a new version of the NS metamodel, i.e. the Elements component, it released, a corresponding version of the Gateway Ontology core layer can be directly generated by the NS-RDF/OWL tool. The version of a component is directly transformed into the version of the projected ontology and is also used as a version indicator of the core layer. The ontology must not be changed once generated to maintain consistency. If changes are needed, those must be done either directly in the Elements component resulting in a new version (of both the component and the core layer) or in the extensions layer (where it would also cause a new version).

The core layer does not raise any obstacles in the transformation of different types of conceptual models or their integration (FR1–FR4). As explained, the extensibility (NR2) of this layer cannot be done directly, as it must remain a projection of the NS metamodel. Nevertheless, there is a next layer designed for this very purpose.

6.2.2 Extensions Layer

The extensions layer serves for the definition of additional constructs to the core layer to reduce the gap between the modelling in NS and the traditional conceptual modelling languages. With the bottom-up approach, repetitive patterns used in mappings of conceptual modelling languages will result in the addition of the corresponding abstraction to the extensions layer.

For example, a concept of inheritance (generalisation or specialisation relation) is widespread in structural modelling languages, but also in some other modelling languages. There is no direct construct for inheritance in NS due to its evolvability issues. The mapping of each modelling language that supports inheritance would need to specify how it should be represented in NS – most typically by composition. However, if we define the concept of inheritance as a part of the extensions layer, the mappings of such languages can be simplified. Moreover, we can specify more ways in which inheritance can be represented, e.g. by following our previous work on inheritance patterns [A.3]. Another candidate for inclusion in the extensions layer is the list of enumerations that are expected to be mapped regularly to taxonomy-type data elements.

To be able to relate the additional construct definitions to the NS metamodel, the core layer must be imported, i.e. used as a dependency, and declare the compatibility. The transformation layer cannot be imported as that would cause a dependency loop or so-called circular imports. The problem is then how to specify the mapping of newly defined constructs to NS. As a solution, our design iterated to the state of equilibrium where the transformation engine directly supports the constructs in a specific version of the extensions layer.

We can specify to what constructs it is mapped without the transformations layer but we can not state how. That is a concern of the tool that executed the transformations. The same concept is applied to the transformations layer itself. It can define different means to capture mapping between conceptual modelling languages and NS, but that is just a description of the execution of the transformation. Moreover, the tool (not necessarily our reference implementation) can specify which subset of extensions and transformations it supports. The details of the operations are explained further in the corresponding sections.

As the extensions layer may define new constructs in addition to the imported core layer, each change in the extensions layer, as well as update of import causing incompatibility, requires a new version to be created. With that mechanism, evolvability is kept just as in software systems. The constructs should not be removed, but marked as deprecated to avoid breaking changes. Similarly, significant changes of existing constructs should be reconsidered as new constructs. Those are just requirements, as there is no usable mechanism to enforce it by design in RDF. Finally, the design allows modularising the layer further if needed. When clusters start appearing in the layer, they may be extracted and form sublayers that can be used separately and evolve independently. For example, a subset of constructs is often used for structural modelling languages, but not for others.

The main goal of the extensions layer is to promote DRYness (NR4) and re-usability (NR3-R) for mappings of conceptual modelling language by enabling extensibility (NR2) to the core layer. However, the mappings do not have to use any constructs from this layer; therefore, it also does not limit the types or aspects captured in the conceptual models (concerning FR1–FR4). It can be changed (FR5) over time without causing combinatorial effects in the Gateway Ontology.

6.2.3 Transformations Layer

The transformations layer provides a vocabulary for capturing the mappings of conceptual modelling languages to NS in RDF. It has its foundations in the SPARQL-Based Mapping Ontology (SBMO) presented in our related work [A.15]. It provides extended expressiveness compared to predicated mappings of OWL such as `owl:sameAs` or `owl:equivalentClass`. It is also designed for execution that utilises SPARQL by composing `CONSTRUCT` queries from the mapping and input data captured in RDF. The constructs in the layer can be extended again using the bottom-up approach. We expect that to fulfil requirements FR1–FR4, it will also become sufficient for other modelling languages. Still, it is ready to evolve over time as all part of our solution.

Despite the focus on composing SPARQL queries to execute the transformation, the overall design can also support other ways of transforming the models. If someone devises a different method, the transformations layer can be either extended with required constructs specific to the method, or an alternative transformations layer can be introduced. Nevertheless, the preferred way would be to create an alternative transformation layer (or sublayer) that contains such method-specific constructs.

This layer is dependent on both extensions and core layers, as it may refer to constructs defined there for direct mapping to them. It can be seen as a facade to NS exposed to mappings. Although it has two dependencies that can evolve, the version used on the extensions layer (eventually its sublayers in the future) limits the versions of the core layer that can be supported.

The constructs that are defined in the transformations layer may evolve very similarly to those in the extensions layer. The only difference is the purpose of the content in the respective layer. Therefore, the same principles for versioning, deprecations, and even sublayering apply in this layer too. The version is again crucial for reference (by conceptual modelling mappers), as well as the tool implementing the transformation based on the mapping specification in RDF.

The requirements FR1–FR4 are directly addressed by this layer – it enabled one to capture the mapping for executing the transformation. Furthermore, the constructs should allow bi-directional transformation for maintaining the consistency (FR5). As the extensions layer, it is expected to be changed over time (FR5 and NR2) and evolvability (NR1) is still assured by following the NS principles.

6.2.4 Conceptual Modelling Language Mappers

The Gateway Ontology, with its three layers presented above, serves as a vocabulary for defining a mapping between any metamodel captured as RDFS or OWL ontology (more generally, it allows us to define a mapping for any patterns in RDF) and the NS metamodel, i.e. the core layer. We call such a mapping specification defined in terms of the Gateway Ontology a *mapper*. When mapping a metamodel of a conceptual modelling language, it is a conceptual modelling language mapper. In terms of RDF, mapper is a set of quads (triples with context) that use resources (as subjects, predicates, or objects) from the Gateway Ontology.

Listing 6.1: Mapping example for a UML classes

```
@prefix rdfs: <https://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <https://www.w3.org/2002/07/owl#> .
@prefix ns: <https://purl.org/nsgo4cm/gateway-ontology/core#> .
@prefix nsgo-e: <https://purl.org/nsgo4cm/gateway-ontology/extensions#> .
@prefix nsgo-t: <https://purl.org/nsgo4cm/gateway-ontology/transformations#> .
@prefix uml: <https://example.com/uml> .
@prefix m0: <https://example.com/ns-mapping/owl/m-class-basic> .
@prefix : <https://example.com/ns-mapping/owl/m-class-basic> .

{
  :mapping a nsgo-t:Mapping .
  :mapping rdfs:label "Basic UML Class mapping to Data Element" .
  :mapping nsgo-t:hasGraphPattern :input .
  :mapping nsgo-t:hasConstructTemplate :output .
  :mapping nsgo-t:isPartOf m0:mapping .
  :cls a nsgo-t:Variable .
  :cls nsgo-t:preferredName "cls" .
  # ... other variables and mapper metadata
}

:input {
  :cls a uml:Class .
  :cls uml:Class-name :cls-name .
  :input nsgo-t:hasFilter "STRLEN(?cls-name) > 0" .
}

:output {
  :cls a ns:DataElement .
  :cls ns:DataElement-name :cls-name .
}
```

As the transformations layer is based on our SPARQL-Based Mapping Ontology [A.15], the mappers follow the same principles as shown in Listing 6.1. A mapper is composed of multiple mapping definitions that may be related to each other (dependency, part-whole). The modularity promotes evolvability as the mappings can be versioned and changed independently, separating concerns of potentially complex mappings into smaller and easily manageable modules as is the one in the example. Moreover, each of these mappings should have its own identifier and metadata; thus, promoting FAIRness.

Except the metadata, each mapping consists of SPARQL `CONSTRUCT` fragments – variables, (input) graph pattern, and (output) construct template. The graph pattern is used to search the input data, binding the specified variables in the pattern (e.g. class resource as URI and class name in Listing 6.1). Then the bound variables are used in the construct template that specifies the output triples. Based on the used constructs in the patterns, the mapping can be made bi-directional directly or by specifying the inversion explicitly.

Listing 6.1 refers directly to the core layer constructs (data element and its name) through generic SBMO terms. Nevertheless, it is intended to use terms from the extensions and transformations layer that provide common patterns for the mapping specification for complex mappings. As explained more in the following section, these layers are designed to be continually enriched bottom-up. Still, it is always possible to specify the mapping without these two additional layers.

The dependency between mappings may occur if one mapping relies on the triples resulting in another mapping. This relation may cause a combinatorial effect; thus, it should always refer to an immutable version of the other mapping. On the other hand, the part-whole relation enabling the tree-like composition of mappings does not cause such issues as it does not specify the order of mappings. For both types of relations, it must be checked that there is no loop formed, i.e. both compositions and dependencies form acyclic graphs.

6.3 Building Gateway Ontology Bottom-Up

As already explained, the extensions and transformations layers are designed to be continually enriched with reuseable constructs. Although it is possible to avoid the use of these two layers and specify the mappings directly with the NS Elements ontology (the Gateway Ontology core), the expectation is to encounter several repeated patterns in the mappings of various conceptual modelling languages. The reason for this assumption is the same target of the mapping and observed similarities of the mappings specified at the beginning of this chapter.

The Gateway Ontology enrichment process is based on Design Science Research (DSR) and mapper refinements. We consider each mapper as a designed artefact and it can be refined within the design cycle. The other mappers and the Gateway Ontology provide the grounding (as a knowledge base). The environment requires mapping for a specific conceptual modelling language, e.g. UML; thus, it defines the scope and requirements together with evaluation criteria, for instance, in the form of a set of models to transform and expected results. As the mapper is being designed, evaluated, and refined, a repeated pattern may occur, as well as similarity with other mappers in the knowledge base. Then, the corresponding abstraction is created, e.g. a new construct as an extension to the NS metamodel with pre-defined mapping to the core layer. All affected mappers can be refined if desired with a new construct in the extensions layer (alternatively, the transformations layer).

The bottom-up approach avoids the definition of redundant, ill-designed, or unused constructs in the extensions and transformations layers. With a top-down approach, the knowledge base would be overwhelmed with abstractions from conceptual modelling languages, causing mappers to inconsistently use these abstractions or intentionally avoid them for their complexity. That is not the case for bottom-up, as only the abstractions needed are added, eventually enhanced, replaced, or removed according to the cycles of DSR.

6.3.1 Structural-Based Gateway Ontology Abstractions

Most of the abstractions included in the extensions layer are related to structural constructs. One of the most common patterns is a primary data element with a name and an optional description. Another complexity that was common in different structural mappings is related to the value field types. The common pattern is matching a string literal describing the type to existing value field type in NS with a fallback to the string type.

Then, there is another specific group of structural abstractions – constructs that are common in various modelling languages; however, not supported in NS. Such an example is an enumeration with literals which can be expressed in NS using a type data element with a name or value field. Then, instances can be generated for each enumeration literal, e.g. to be imported in the expanded application. Another example is the inheritance relation between two entities which can be transformed to link fields with a special field option.

6.3.2 Behavioural-Based Gateway Ontology Abstractions

Similarly to the structural one, there are also several behavioural abstractions that are developed using the bottom-up approach. We designed simplified workflow modelling constructs in the extensions layer that avoid the use of some NS-specific aspects and complex nested structures or duplications. Instead of creating tasks and data flow tasks, it is enough to specify a task for a data element. Then, it is mapped to both, and additionally it may produce names for all states required if not stated explicitly. For example, if the task is named *CommitBooking*, then the related failed state would be *CommitBookingFailed*.

In this way, we were able to reduce the complexity and repetition in mappings for BPMN, BORM, and UML activity diagram. All these languages share common constructs, such as states, activities (or tasks and events), or branching. With our extensions that are closer to the constructs in those languages, the mapping is more straightforward, and thus is also maintainable.

6.3.3 General Gateway Ontology Abstractions

In addition to the previously mentioned groups of abstractions in the gateway ontology, there are also abstraction components that are not tied exclusively to a specific type of conceptual modelling language. These components serve to simplify the mapping process for any conceptual modelling language, without making assumptions related to covered aspects. A notable example of such an abstraction is the specification of a more complex component required to create a valid NS model.

By incorporating these versatile abstraction components into the gateway ontology, we provide a more inclusive and adaptable framework for mapping various conceptual modelling languages. This flexibility enables researchers, analysts, and practitioners to leverage the ontology in a broader range of contexts, accommodating different modelling requirements and capturing essential details of the original conceptual models. Through these diverse abstraction components, we enhance the ontology's effectiveness in supporting mapping and translation processes, fostering interoperability and facilitating the integration of different modelling perspectives.

6.3.4 DSR-Based Extensibility

The layered approach of our design offers both a bottom-up enrichment of layers through repeated patterns and specific use cases, as well as the flexibility to add new layers or replace existing ones as needed. This design principle ensures the extensibility of our artefact. For instance, suppose there is a requirement for compliance and integration with another model-transformation framework. In that case, we can develop an additional transformations layer that facilitates more specific model manipulation, further enhancing the core ontology. Similarly, the extensions layer can be subdivided into multiple parts based on different types of conceptual modelling. By allowing such modifications, our design accommodates diverse needs and promotes scalability and adaptability.

These possibilities align perfectly with the iterative nature of DSR and can be viewed as subsequent iterations within the design cycle. As new circumstances and requirements arise, these further iterations enable our design to evolve and improve continuously. By embracing this iterative approach, we ensure that our design remains flexible, responsive, and capable of incorporating advancements and enhancements in the future. The design science research methodology serves as the guiding framework for these iterations, facilitating the integration of new knowledge and insights into the design process, and ultimately leading to a more robust and adaptable artefact.

6.4 Performing Transformation with Mapping

By encoding the mappings between conceptual modelling languages and the NS metamodel from Section 6.1 as mappers in RDF as shown in Section 6.2.4, we enabled machine-actionable and thus possibility to perform the transformation for underlying models based on the mapping. While having the mapping specification in tables is suitable for human readers and for the initial design of the mapping, it is not directly executable. Even if such a table is in a machine-readable format, a tool to execute the transformation would not understand the content (e.g. how should be the link fields created for associations according to Table 6.1). In this sense, the initial mappings presented in Section 6.1 are not machine-actionable; however, they are suitable for capturing the general matching of constructs between the metamodels.

Turning the mappings into RDF mappers allows us to precisely specify the relations between the metamodels as recipes for performing the transformation. It links directly to a conceptual modelling language metamodel (as ontology) on one side and the Gateway Ontology constructs (consequently NS metamodel as ontology) on the other. Furthermore, it enables modularisation of the mappings and creation of re-usable abstractions to make the mappers maintainable by applying linked data and semantic web techniques consistently to what we are using in the other artefacts design in this dissertation thesis.

6.4.1 Pattern-Based SPARQL Queries

Mappers composed of partial (potentially interconnected) mapping modules specify fragments of SPARQL `CONSTRUCT` queries, as shown in Figure 6.2. The mappings can be interlinked by dependency and composition, and possibly specified as bi-directional. Then, those mappings (e.g. a set of mappings for a single conceptual modelling language) are input for executing the transformation, more specifically generating the corresponding SPARQL queries.

With SPARQL queries prepared based on the mappings, the transformation of input RDF dataset with a conceptual model can be executed. New triples created by SPARQL queries are added to the dataset and may be used by subsequent queries. As part of the finalisation phase, the dataset can be cleaned up, i.e. unnecessary triples can be removed. If there are bi-directional mappings, it is also possible to execute the transformation in opposite direction for which different SPARQL queries are generated from the mappings.

6.4.2 Mapping Specification in RDF

The mappings in RDF serve as a description for creation of SPARQL `CONSTRUCT` queries; thus, must enable encoding of the patterns according to the World Wide Web Consortium (W3C) specification [164] for such queries. A SPARQL `CONSTRUCT` query has two main parts. The first is a *construct template* that contains a set of triples. The second part, which can be more complex, is a `WHERE` clause that contains *group graph pattern*. The triple patterns may contain variables in both parts, the pattern in `WHERE` clause is “filled” with matching data and values are bound to variables filling the *construct template*.

Except variables and triples (subject, predicate, object), a *group graph pattern* may contain additional constructs: subgroups, `OPTIONAL`, `FILTER`, `UNION`, `MINUS`, sub-`SELECT`, or other less-commonly used constructs and functions. We created SPARQL-Based Mapping Ontology (SBMO) (SBMO) to define the constructs for expressing the mapping in RDF;

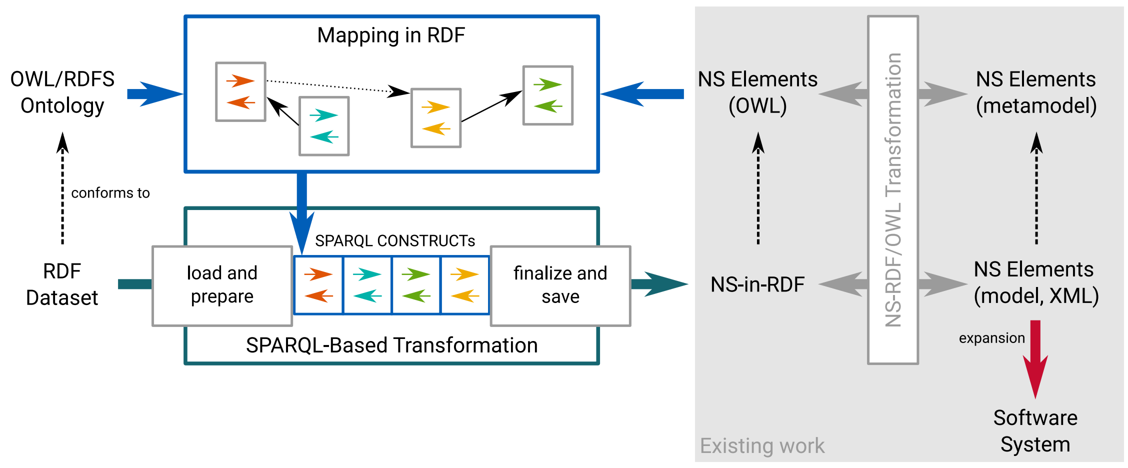

Figure 6.2: Design of SPARQL-based transformation from RDF mapping

moreover, it is designed for further enhancements and extensions in the same way as other designed artefacts if this dissertation thesis.

To specify the mapping in RDF, our design uses named graphs and recommends using the TRiG format to keep separate mappings (as modules) as already presented in Listing 6.1. The default graph of the mapping in RDF contains its metadata such as name, the definition of used variables, or relation to graph pattern and construct template—these two form separate graphs where are the input and output triples. The input pattern may contain the additional constructs; the example shows the use of a basic filter – the constraint is defined as a string value for higher flexibility.

The mappings can be defined without any notion of NS or the Gateway Ontology. Nevertheless, the composability of the mappings enabled by our design allows us to introduce prepared fragments, re-usable helpers, or basically any additional constructs to simplify other mappings. That is the content of our extensions and transformations layers to reduce the complexity of mappings between conceptual modelling languages and NS. For each of these constructs, a mapping must be supplied or handled differently during the transformation.

For example, suppose we have mappings of two constructs A and B from different conceptual modelling languages towards the same NS construct C. The mappings are sets of triples with measurable complexity (e.g. the number of statements). Both mappings have certain similarities in the patterns that can be abstracted. Thus, we introduce extension E mapping to C to represent the repeated non-trivial pattern. Then, A and B can be mapped to E instead of C, which reduces the complexity of the two mappings. Finally, for transformation from A to C, the two mappings must be executed (A to E and E to C).

6.4.3 Transformation Execution

With the SPARQL-based transformation design and mappings defined in RDF, we can proceed to address the actual execution of transformation based on a mapping. The execution algorithm can be split into two stages. The first stage prepares the transformation based on mappings, i.e. creates SPARQL queries from the input RDF mappings. It consists of the following steps:

A1. Load mappings (starting with input top-level mapping, recursively follow composition links).

A2. For loaded mappings, find order of mappings conform to dependency links (if such order does not exist, algorithm fails).

A3. Check name clashes for variables and resolve them (using numbering to ensure uniqueness) if necessary.

A4. Rewrite each mapping to its SPARQL `CONSTRUCT` equivalent (according to definitions in SBMO).

Practically, step A1 is expected to be implemented to load multiple RDF TRiG files together and extract mappings from them after providing the URI of the top-level one. The result is a list of SPARQL queries that can be eventually exposed but is not intended for external use or manual edits (as it should remain consistent with the mapping). Then, when the SPARQL queries are prepared, the actual transformation of the RDF dataset according to the mapping can take place using the following steps:

B1. Load input RDF dataset D and create its immutable copy D'.

B2. Execute each SPARQL query over D in the list (in the order). For each, insert the constructed triples by SPARQL query add D (one-by-one, if not already in D).

B3. (Optional cleaning step) Remove all statements in D that are in D'.

B4. Save D in desired format.

The steps B1–B4 are simply executing the SPARQL queries and adding the constructed triples to the result dataset. We propose to add the triples to the input dataset for two reasons. First, the dependent mappings may already use triples generated according to the previous mappings and avoid duplication. For example, again, it is necessary to check if a class forms a data element when dealing with object properties to link fields mapping. The second reason is that the desired output in most cases is an enriched RDF dataset rather than a dataset with only the new triples; however, that is possible with optional step B3. Another essential aspect is that after the successful first stage (steps A1–A4), the second stage may be repeated for various inputs until the mapping changes.

Both stages are finite as there the size of inputs (mapping files, statements in mappings, and statements in input dataset) is also finite. It may fail only if invalid mapping is provided, for example there is a linked mapping is missing or invalid triple is included. The overall complexity highly depends on execution of the underlying engine executing the generated SPARQL queries (typically polynomial complexity [165]).

6.4.4 Bi-directionality and Consistency

The last goal (**G3**) requires a definition of a procedure to verify (or maintain) consistency. Our framework and way of mapping specification using SBMO allows bi-directional transformation. The identifiers (Uniform Resource Identifiers (URIs)) are always kept based on the mapping, there is no non-determinism, and RDF is declarative. Therefore, it is possible to keep the RDF datasets consistent using the reverse direction of transformation and a simple merging of changes. The recommended way is to keep the data in a single RDF dataset that is continuously updated rather than keep two separately.

The opposite direction for the transformation can be issued simply by switching the input and output patterns. The only issue is with additional constructs of the `WHERE` clause. There are two possibilities with SBMO. First, both input and output patterns can contain such constructs, and only one is used, based on the direction of the transformation. Another possibility is to use directionality of mapping and create an inverse one using `sbmo:inverseMapping`.

6.5 Design Cycle of Gateway Ontology Development

As for the previously presented designed artefacts, DSR guided the construction of the Gateway Ontology design and even its materialisation in the RDF/OWL form. We further separated the design into partial sub-artefacts – layers and mappers according to the separation of concerns principle. Then, we also designed the actual transformation based on mappings captured using the ontology. The existing work and common approaches from the knowledge base, as well as the NS context (environment), also provided solid foundation for our design.

The Gateway Ontology, its layers, and transformation procedures were iteratively validated and refined based on the design cycle. The straightforward incorporation of needed enhancements was accomplished by design that enables evolution and bottom-up extensions. The core layer served as the basis for the work. Then, we designed the transformations layer using the SPARQL-based mappings [A.15]. While mapping each of the selected conceptual modelling languages (both defining the mapping and encoding it as mappers), we enriched the transformation layer when necessary. We designed the extensions layer based on common patterns in different mappers to avoid complex repetitions.

The design cycle for the transformation procedure iteratively increased coverage of the definitions in mappers to full coverage, i.e. everything that can be specified will be transformed. The validation and evaluation have been done by executing the transformation for a set of different conceptual models prepared in the mapped languages. Whenever the result did not correspond to the mapping, its RDF form was updated or the transformation extended. Finally, the same procedure was used for semantically integrated conceptual models and reverse transformation. A part of the evaluation is presented in Chapter 7.

The final artefact presented in this thesis is ready to be (re-)used and further extended according to DSR. Although we provide a reference implementation for the transformation procedure using the Gateway Ontology, the design allows alternative implementations as well as an exchange of underlying technologies. For example, it is possible to use a different transformation mechanism than SPARQL by adjusting the transformations layer and mappers.

6.6 Summary of Gateway Ontology

This chapter captures our design of the Gateway Ontology with its layers, mapping of different conceptual modelling languages, and the transformation execution using RDF technologies. It is the final piece (partial artefact) of our architecture presented in Chapter 3. However, it is, as the name indicates, the key that addresses the actual transformation between conceptual models and NS, i.e. requirements FR1–FR4. The other artefacts supported the use of the ontology by allowing knowledge representation from conceptual models and NS in RDF.

Both evolvability (FR5 and NR1) and extensibility (NR2) are enabled by-design and helped to refine the artefact according to DSR as discussed in the previous section. The design enables bi-directional transformations to maintain consistency (FR6). Also, the used RDF contributes to this requirement fulfilment as there are existing techniques and tools for comparing, merging, or other manipulation of RDF datasets.

The transformation design utilises SPARQL but does not provide any other limitations with respect to the technologies or platform (NR5 and NR7). The provided reference implementation as a materialisation of our designed artefact was used for evaluation. Nevertheless, it can be used for real-world transformation scenarios between conceptual models and NS. All of the presented designed artefacts can be directly used, further extended, or taken as a blueprint for the specification of mapping and transformation between different kinds of knowledge representations.

Chapter 7

Demonstration Use Cases

"No research without action,
no action without research"

Kurt Lewin

After presenting all designed partial artefacts in the previous chapters, we can use them together based on our overall architecture as explained in Chapter 3. We created reference realisations and prototypes based on our design to evaluate, verify, and subsequently refine the design. Due to the presence of these realisations of the artefacts, we can use them all together to evaluate the overall architecture design.

This chapter demonstrates the practical application of our Normalized Systems Gateway Ontology for Conceptual Models design with presented partial realisations, namely developed tools and created ontologies and mappings. We composed a set of inter-related conceptual models for an e-commerce system based on literature and technical solutions review. Then, several use cases are presented – starting with a simple one-way transformation from a single conceptual model, followed by semantic integration and dealing with changes on various levels, to reverse-engineering, where a conceptual model is created from an Normalized Systems (NS) application. As we include only relevant fragments in the text, the complete demonstration case is included in Appendix A.

7.1 Using NS Gateway Ontology for Conceptual Models

The architecture proposed in Chapter 3 and related designed artefacts together allow transformation between conceptual models and NS. According to our research objectives formulated in Section 1.2, there are different ways in which it can be used and what are the related requirements or prerequisites, inputs, and resulting outputs. Although our solution is quite flexible, in this dissertation thesis we focus on our research objectives and work with the following use cases (which may be potentially combined):

1. The semantic integration of various conceptual models (RO1) can be done when the models are in an interoperable format – in our case in Resource Description Framework (RDF), preferably structured according to a well-defined ontology using Resource Description Framework Schema (RDFS) or Web Ontology Language (OWL). The transformation from original "source" to RDF is not within the scope of this thesis, as there are limitless ways to capture a conceptual model. RDF technologies directly provide ways of combining multiple knowledge graphs, conceptual models, relating concepts, and providing a more detailed view on a problem domain by combining different aspects or viewpoints. The main goals of this step or use case is to prepare an input for the transformation and by having such a more detailed view, we will get potentially more precise corresponding software system (also based on the used mapping). The semantic integration and related mapping needs affect also the Gateway Ontology as commonly used constructs can be abstracted to the extensions and transformations layers for reuse (and thus Don't Repeat Yourself (DRY)ness of mappings).

2. Transformation of a conceptual model to NS (RO2), or alternatively of semantically integrated conceptual models to NS, can be done directly using our designed artefacts. As a prerequisite, in addition to the input conceptual model in RDF, a mapping must be provided for the conceptual modelling language(s) used. Then, the input model is transformed using the SPARQL-based mapping defined using the Gateway Ontology. The resulting RDF represents an NS model; thus, the next step is to use the NS-RDF/OWL transformation that creates corresponding NS components usable with other NS tooling, e.g. for further editing or expanding a software system. The transformation also allows the opposite direction, i.e. creating RDF representation of NS models and also their transformation to conceptual models (in RDF).

3. Adapting to various changes (RO3) is also significantly simplified using RDF to represent all (meta)models, the Gateway Ontology as well as mappings defined in terms of the Gateway Ontology. The most direct case that may occur is a change in the input conceptual model (that was previously used in the transformation); then, the changed model can be transformed again and the result must be compared and merged with the previous result (in case there were changes made using the NS

tooling). As long as the mapping is the same, the unchanged parts of the model will result in the same NS counterparts. Another change may occur in the mapping used for transformation (e.g. some coverage improvement), in that the same procedure must be used including the merging. Finally, the most difficult changes in terms of their adoption are related to metamodels of modelling languages. When a new version of conceptual modelling language specification occurs, the ontology for that new version must be created, the input model must be adapted to that new version, and thus also the mapping. If a change occurs in the NS metamodel, the procedure is similar; however, newer versions of the NS metamodel are typically backward compatible.

4. The consistency between the input and output models (RO4) is done by keeping links between concepts using the identifiers from RDF – Uniform Resource Identifiers (URIs). With that, it is possible to track which part of input resulted in what output without investigating the mapping. Therefore, it also allows to simplify check for differences. We also propose using a single RDF dataset for both input and output models together. The transformation just "enrich" the dataset by missing constructs, keeping all the information together.

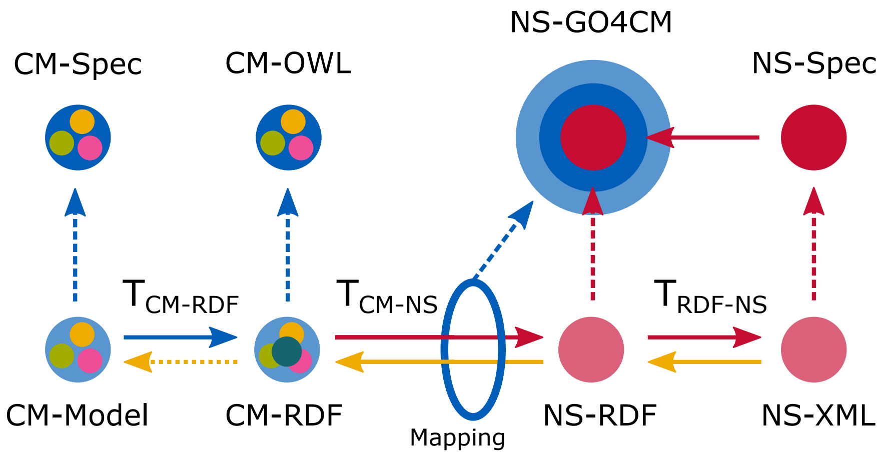

Figure 7.1: The Gateway Ontology and its various use cases

These use cases and their relations are also visualised in Figure 7.1. As a first step, we need to transform given conceptual models (defined using a specific conceptual modelling language, that is, metamodels) into RDF representation (Section 7.2). There, the models can be semantically integrated where necessary (RO1, Section 7.4). However, the primary purpose is a transformation of a conceptual model to NS even without such integration. The transformation can be performed for the modelling languages supported by mapping defined using the Gateway Ontology, then the two-step transformation with intermediary NS model in RDF is possible (RO2, Section 7.3). With transformed models, there are various change-drivers as different artefacts and models can evolve and the transformations support the propagation of the change (RO3 and RO4, Section 7.5). Finally, the transformation is possible in the reverse direction, that is, from NS to conceptual models, if a mapping supports it (RO1 and RO4, Section 7.6).

7.2 Conceptual Models for e-Commerce System

To evaluate the design of partial artefacts, various models were used – both created directly for testing (e.g. to cover specific constructs) or from existing work (e.g. domain or software documentation, or relevant supervised bachelor and master theses). However, to provide a homogeneous but complex demonstration, we selected the e-Commerce domain for which we searched for existing suitable conceptual models. The models described in this section serve as input for our further demonstration of various use cases of what can be done with the solution based on the Gateway Ontology.

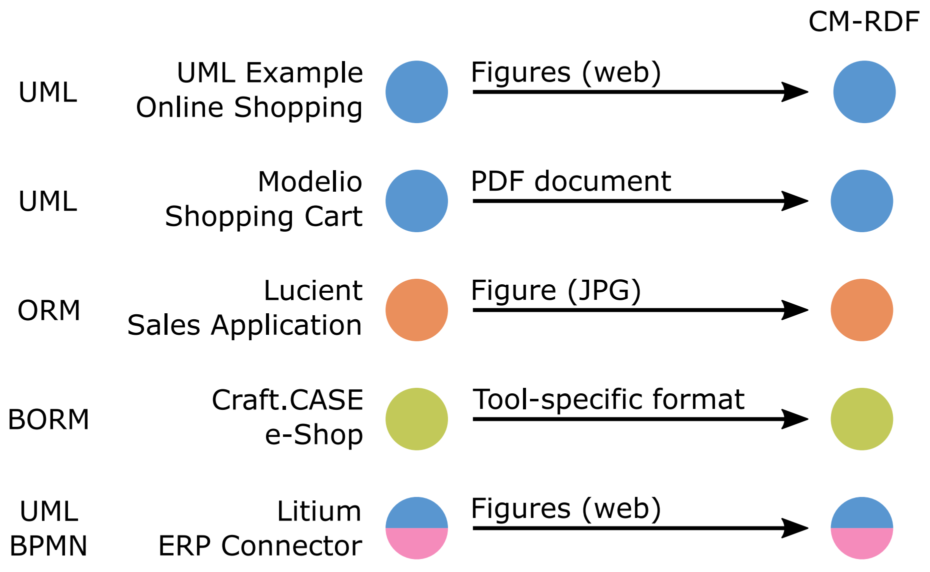

Figure 7.2: Transformation of conceptual models to RDF

Although the selected models for the demonstration are related to a single domain and capture overlapping concepts, they have certain differences. The disparity of models in size, original purpose, modelling language used, and captured form represents the real-world encounter of conceptual models. The selection has been made on the basis of these different properties of the models to present the versatility of our artefacts.

Figure 7.2 (as similar figures for the following parts of this demonstration) visualise the inputs and desired outputs described in this section. We selected five input models from various setups: first two are Unified Modeling Language (UML) models where one is presented on a website and the other one as Portable Document Format (PDF) documentation for software development, next one is an Object-Role Modeling (ORM) model captured just as a single diagram in Joint Photographic Experts Group (JPEG) file, the penultimate model is in Business Object Relationship Modelling (BORM) in a tool-specific format (Craft.CASE project), and the last one is combining two modelling languages – UML for capturing structure and Business Process Model and Notation (BPMN) for behaviour. As none of them are in suitable and interoperable format, we manually transform them to RDF using the ontologies described in Chapter 5.

7.2.1 UML Diagrams: Online Shopping

UML-diagrams.org is a comprehensive guide to UML including its various versions and all diagrams. It also contains several examples, one of which is "Online Shopping" [166]. The example consists of different related diagrams. We use its class, activity, and state machine diagrams for this demonstration.

The diagrams are accompanied by textual descriptions and clarification of used constructs. Other advantages of this example are that it provides multiple interconnected diagrams, uses different and even rare constructs, and is still concise, so we can focus on details when examining the transformations. Finally, there are some possible improvements, so it is also suitable to demonstrate the evolution of the conceptual model and its impact.

The class diagram (as shown in Figure 7.3) captures 8 classes and 2 enumerations with different types of relations, constraints, and attributes. The class account is related to the state machine diagram that captures its 4 states (New, Active, Suspended, and Closed) and transitions between them. The activity diagram describes the process of shopping: managing items in the shopping cart, checking the cart, and checkout.

All the diagrams are available only as figures in PNG format. Thus, we manually encode the information from the diagrams to RDF using our ontology presented in Chapter 5. It is often the case that conceptual models are kept only as figures, possibly as part of more extensive documentation. That is not interoperable, and information must be extracted manually or semi-automatically for further processing.

7.2.2 Craft.CASE e-Shop Example

To represent BORM models in the demonstration, we use the "e-Shop" example from tool Craft.CASE that is already encoded in RDF as part of our work on OntoBORM [A.13]. As BORM is not widely used and models in BORM are rarely published, it is not surprising that no other e-Commerce related models were found on public websites or scientific articles, except for a single smaller Objects Relations (OR) model (as figure) [167].

The advantage of the "e-Shop" example from Craft.CASE [161] is that it covers the entire modelling process according to BORM where Business Architecture (BA) and OR are just partial results. As noted in Chapter 5, it also contains a description of classes and their attributes that are not captured in the two main diagrams of BORM (more specifically 1 BA shown in Figure 7.4, and related 4 OR).

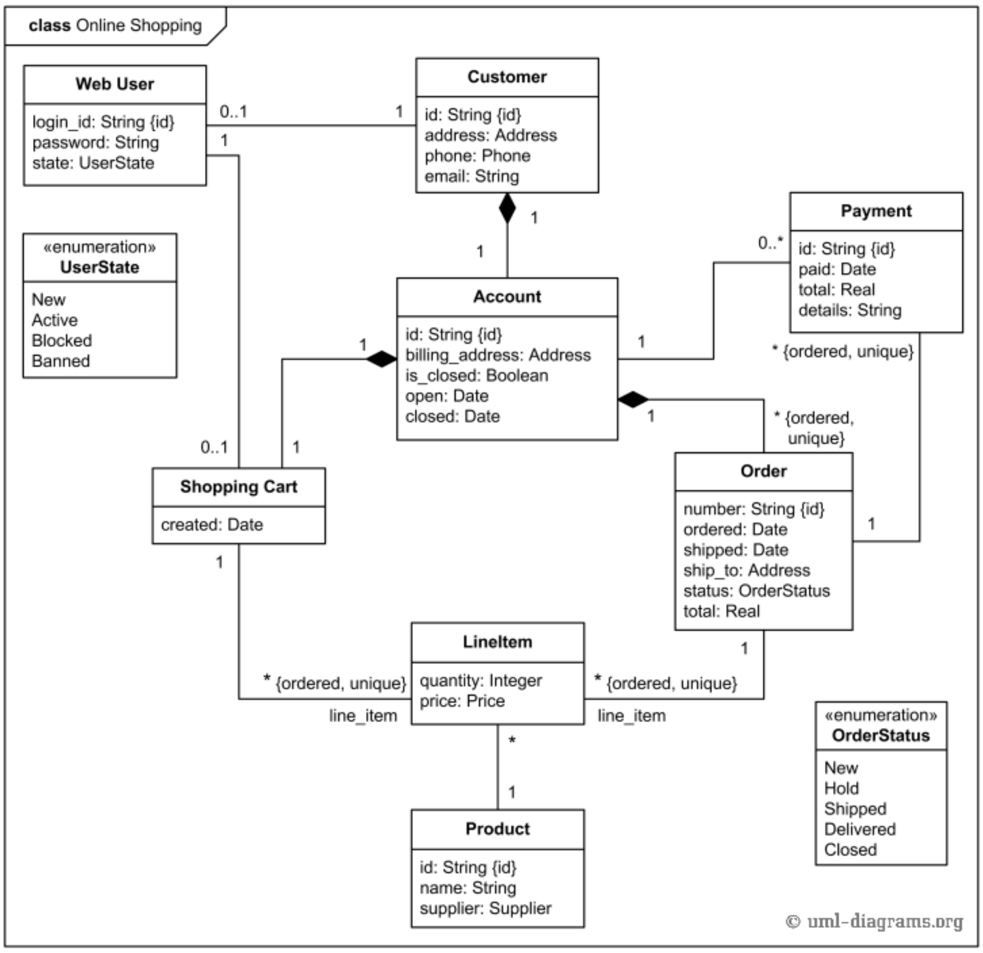

Figure 7.3: UML class diagram of the Online Shopping case [166]

The original model is part of the Craft.CASE tool installation, i.e. it is a reference project in tool-specific format distributed together with the tool. There was no way to get or refer to the model. Moreover, while working on this demonstration part of the dissertation thesis, the tool and its website changed owners and it is no longer possible to download the tool directly. That shows another common issue with current models – persistency and accessibility.

7.2.3 Modelio's Shopping Cart

Modelio[1] is a company with over 30 years of object-modelling experience that provides training, consultancy, and modelling platform. It also publishes various examples of models, including real-world example documentation of complex systems analysis and design. One of such examples is the "Shopping Cart" case [168]. In its 47 pages, the document describes various use cases and packages related to an e-commerce software system.

[1]https://www.modeliosoft.com/en/

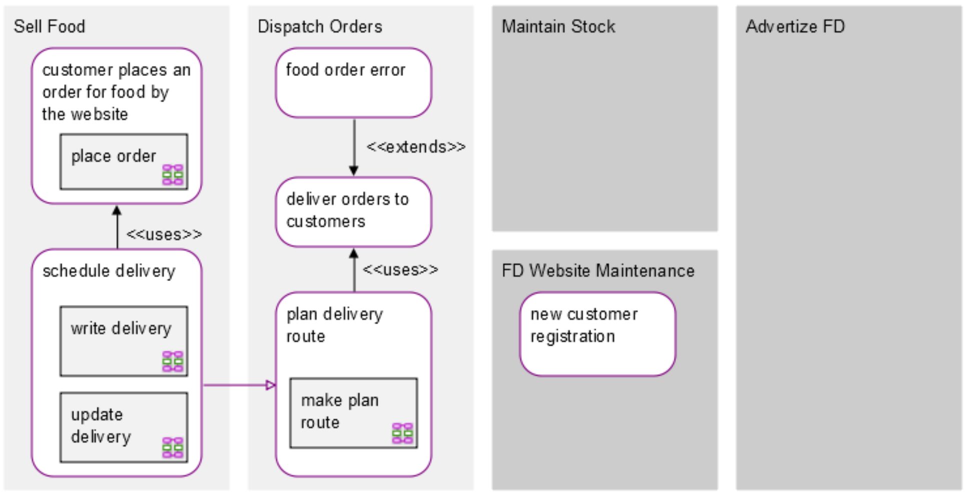

Figure 7.4: e-Shop BA in BORM example of Craft.CASE [161]

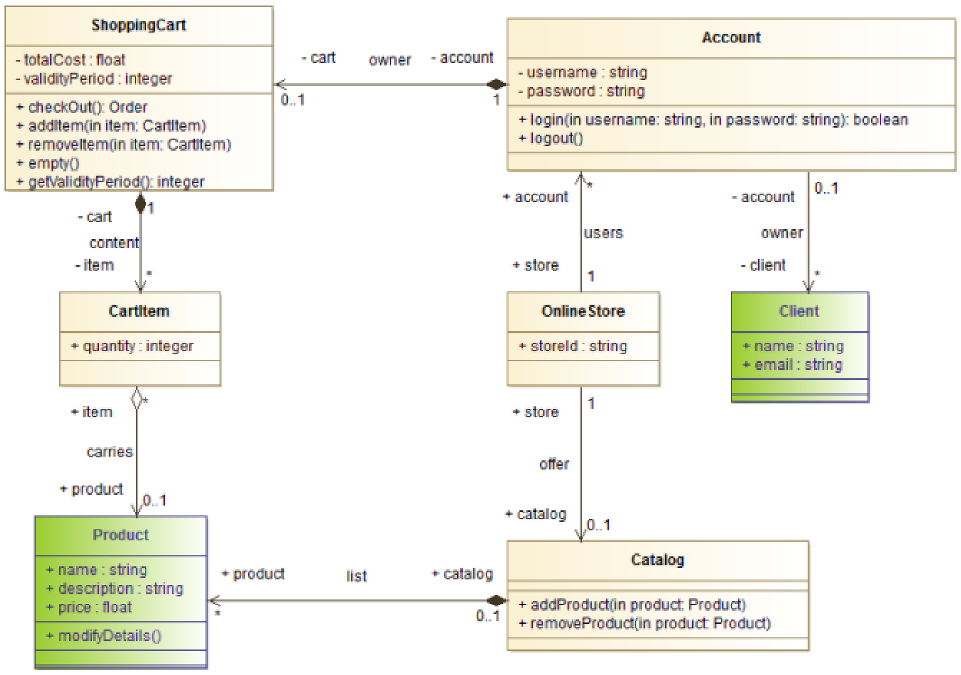

Figure 7.5: UML class diagram of OnlineStore package [168]

The essential part of conceptual modelling is “domain” packages: online store (in Figure 7.5), bank, client, products, and types. Except for the bank package (with single class Payment), all have UML class diagrams. Moreover, classes Order and Account (in Figure 7.6) have their own state machine diagrams. All classes are also described in the text, including tables with attributes and operations as usual in such documentation. The remaining part of the documentation contains a software system design that is not relevant for our demonstration.

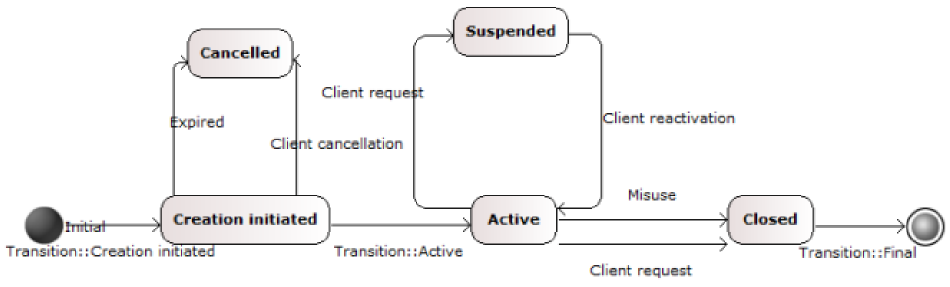

Figure 7.6: UML state machine diagram of Account [168]

Although it could be possible to extract the information from PDF file and figures using text mining and the OCR technique, we again encode it to RDF manually due to efficiency reasons. It furthermore shows the lack of interoperability in the world of conceptual modelling.

7.2.4 Litium Connector Models

Litium[2] is a scalable e-commerce platform that unlike others publishes its documentation including conceptual models [169]. The models are part of the Litium Connect component that supports integrations with third-party systems. The documentation is in the form of a website with models captured in text and figures. As in other cases, it is necessary to encode the information into RDF manually.

There are two domain class models. The Prices model is simple, with just two classes. However, the Orders model has 15 classes and 7 enumerations. There are also many various attributes as the model is related to technical implementation (importing and storing relevant data). We also use the related BPMN model of the checkout flow.

7.2.5 Lucient's ORM Model for Sales Application

Global data consulting firm Lucient[3] publishes various experiences and examples on its blog. One of the older posts by consultant Dejan Sarka presents an ORM model to support an imaginary Sales application [170] as shown in Figure 7.7. Again, the model is not in interoperable format (which is not even specified for ORM) – this time in JPEG format which is not suitable for diagrams.

The model is relatively simple in terms of ORM constraint capabilities, but it also uses rare constructs such as objectification. On the other hand, there are only binary predicates. The potential issue may arise from nine value types that are very domain-related and vague; e.g., for a discount, it is not clear if it is a percentage of the price, fixed

[2]https://www.litium.com
[3]https://lucient.com

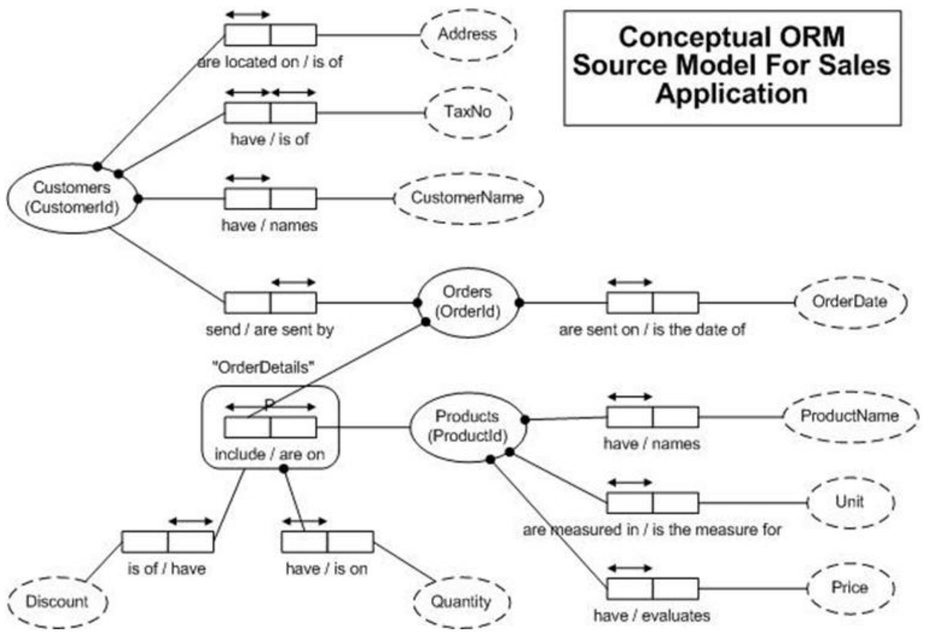

Figure 7.7: ORM model for Sales Application [170]

amount, or possibly a combination. Thus, the model may also be used to demonstrate the evolution – improvements in the conceptual model and reflecting it in NS.

7.3 From Conceptual Models to Normalized System

This first scenario demonstrates the primary use of our transformation framework – one-way transformation from an existing conceptual model to the corresponding NS model. It tests our mappings and transformation procedures in the direction toward NS. We use all previously introduced input conceptual models, more specifically their interoperable RDF representation. The process is explained thoroughly for the Modelio case, and then for others, we highlight differences and specifics to the input modelling languages and models.

With respect to RO2, we demonstrate the transformation of conceptual models that capture various aspects (structural, behavioural, and facts) into NS models. The minimal loss of information is related purely to the mappings; only the things omitted in the mapping are lost. Thus, the framework itself does not directly limit any information transfer.

With the conceptual models captured in RDF using the ontologies of Chapter 5, we can proceed with the two-step transformation to NS using the Gateway Ontology and related tooling (designed artefacts) as shown in Figure 7.8. For each of the models, we proceed individually. First, we execute the transformation using our mappings (described in Chapter 6) and NS-RDF/OWL transformation (described in Chapter 4). Then, we observe the results, confront them with original models, and identify information loss. The outputs for the scenario are the resulting NS models in Extensible Markup Language

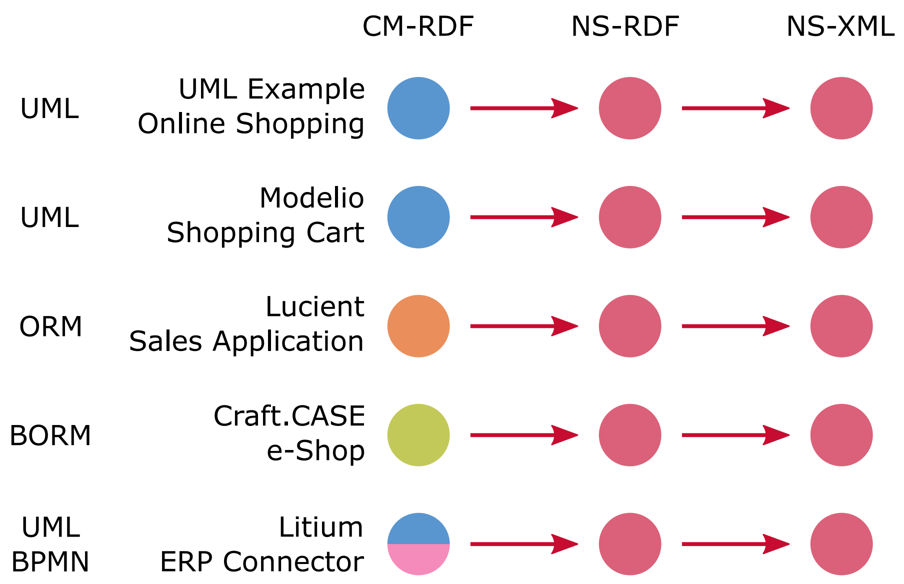

Figure 7.8: Conceptual models to RDF transformation

(XML) (which can be used in existing NS tooling, e.g. to expand applications), as well as the intermediary NS/RDF representation.

7.3.1 UML Diagram Examples to NS

When transforming the class diagram of the UML Examples Online Shopping case, all eight classes and two enumerations are turned into data elements as shown in Figure 7.9. More specifically, the `OrderStatus` and `UserState` enumerations become taxonomy data elements. For attributes (mapped to value fields), we had to improve the RDF-NS transformation to deal with non-compliant naming – in this case, snake case in attribute names, whereas camel case is allowed. The `id` constraint is captured using the field option. Some attributes use a data type `Address` that is not defined; thus, it uses the default mapping to `String`, which we found acceptable for this case. The link fields are based on associations and aggregations in the class diagram – link field types mapped based on cardinality and constraints again kept as field options. As a refinement, we added data children to express composition, e.g. `Order` is a data child of `Account`. Here, we identified an issue with the input model and its `LineItem` class that should be linked via composition and not to both `Order` and `ShoppingCart` at the same time (preferably to a superclass of these two classes).

The state machine diagram is clearly linked to the user account (after our fix of the inconsistency – `UserAccount` and `Account`). The four states (New, Active, Suspended, and Closed) were transformed into data states of the Account data element. Moreover, the Initial state was created as there is a named transition between Initial and New states in the diagram. On the other hand, the Closed account is just marked as final (the transition is not named) – this was part of our refinements as we initially had an unnamed transition

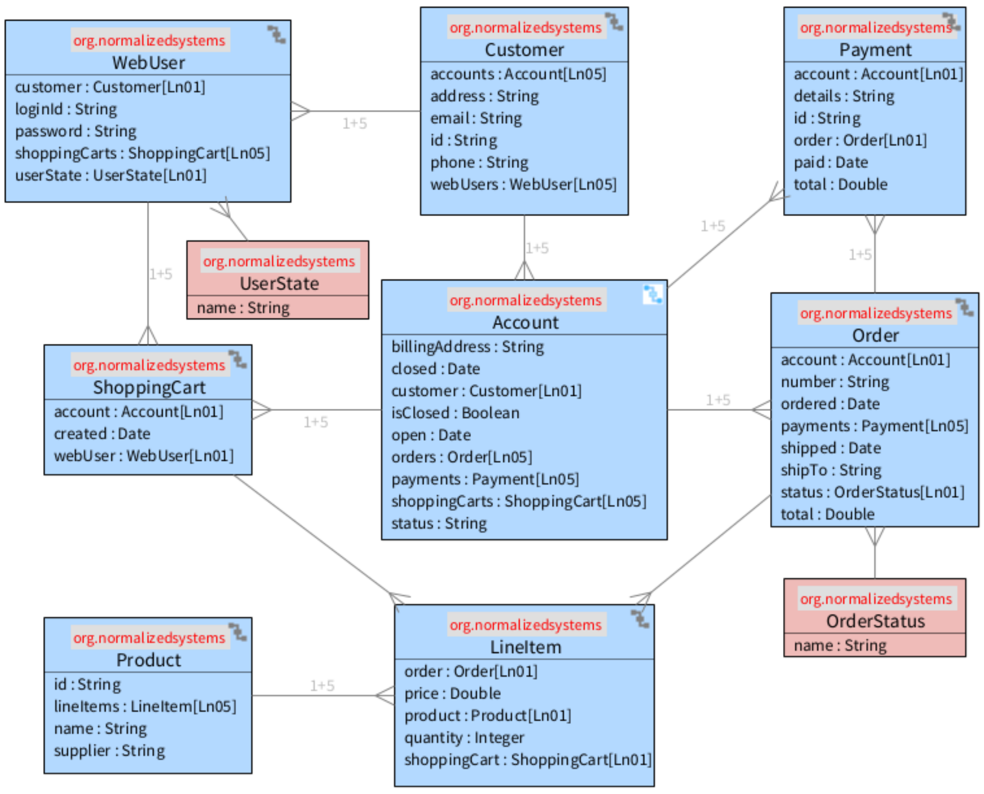

Figure 7.9: UML Online Shopping example in NS

between Closed and Final states. The transitions are mapped to task elements. As the names of transitions are not unique, we had to refine the mapping to comprise the name together with the name of the data element and source/target states. Still, there were duplicities, e.g. `AccountNewSuspendSuspendedTask`; however, this was solved by merging the overlapping transitions together (it can be one with logical OR applied constraints). Finally, data flow tasks are created with failed and interim states named according to the task element. The constraints are not mapped and are captured as task options.

The activity diagram did not provide any useful result when transformed to NS. The leading cause is that it does not capture any relation to class nor state machine diagrams of the Online Shopping example. It would need to enrich the diagram with semantic links, e.g., a "View Item" activity is somehow related to the "LineItem" class. Again, there was an issue with branching for the state machine. Another problem was creating a flow element according to the mapping, since the diagram is not related to a single data element or class. In summary, the activity diagram, in this case, is not suitable for software generation; it can be helpful for people only to "read" how online shopping roughly works. The result cannot be improved just by refining the mapping and tools; it would require incorporating some AI-related features that can result in non-deterministic transformation and are out of our scope.

7.3.2 Modelio: Shopping Cart Model to NS

Unlike the previous Online Shopping example, the Modelio case is structured into multiple packages. In addition, various constructs (packages, classes, and attributes) are provided with textual descriptions. In total, there are five components: `bank`, `client`, `onlineStore`, `products`, and `types`. Both the components `bank` and `types` are quite trivial, contain three data elements with only value fields (`Payment`, `Address`, `CreditCard`). The `Address` data element is, for example, then used from the component `client` as the link field of the data element `Client` (mapped from the attribute with a custom datatype). The `client` component also contains linked `ApplicationForm` and its data child `ApplicationField` data elements.

The component `products` contains four data elements transformed from simple UML classes with its attributes (value fields), associations (link fields, including many-to-many), and a composition between `Order` and `OrderLineItem`. The operation `modifyDetails` of `Product` was mapped to a data option, as there is no other suitable counterpart in NS. In the same way, operations of the `onlineStore` package are just data options. The only important thing in this package and transformed component are the link fields to other components (relations to `Product` and `Client`) as visible in Figure 7.10. To add more complexity, we improved the input model by making `totalCost` a derived attribute that resulted according to expectations in a calculated field (the calculation based on the price and quantity of the product needs to be part of the custom code).

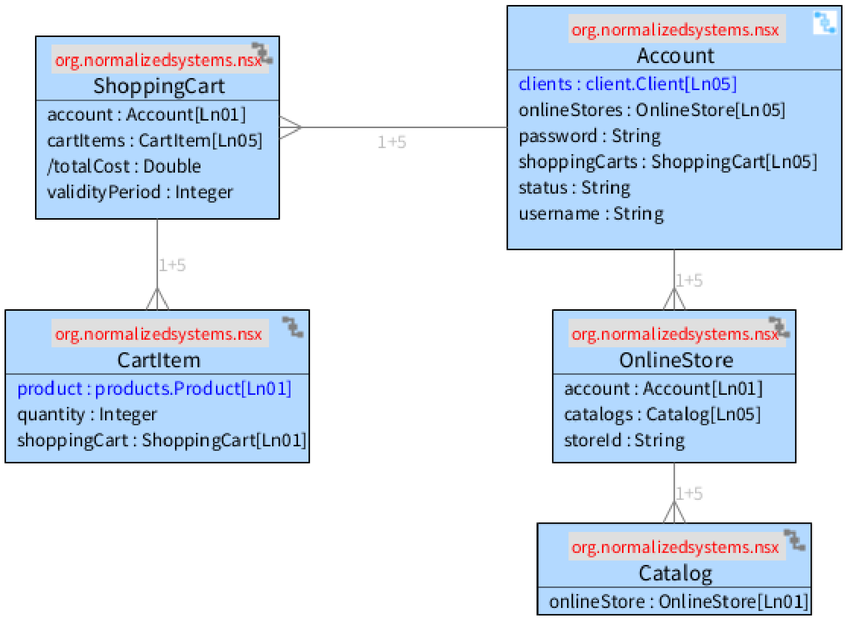
org.normalizedsystems.nsx
ShoppingCart
account : Account[Ln01]
cartItems : CartItem[Ln05]
/totalCost : Double
validityPeriod : Integer
org.normalizedsystems.nsx
Account
clients : client.Client[Ln05]
onlineStores : OnlineStore[Ln05]
password : String
shoppingCarts : ShoppingCart[Ln05]
status : String
username : String
1+5
1+5
org.normalizedsystems.nsx
CartItem
product : products.Product[Ln01]
quantity : Integer
shoppingCart : ShoppingCart[Ln01]
1+5
org.normalizedsystems.nsx
OnlineStore
account : Account[Ln01]
catalogs : Catalog[Ln05]
storeId : String
1+5
org.normalizedsystems.nsx
Catalog
onlineStore : OnlineStore[Ln01]

Figure 7.10: Component `onlineStore` for the Modelio case in NS

Two state machines, for `Account` and `Order`, are transformed into corresponding flow elements. None of the flows had any new mapping-related issues compared to the previous UML Online Shopping Example. States are transformed into data states, transitions into task elements with appropriate interim and failed states, and finally, data flow tasks are created to link them together. We had to improve the naming-fixing feature of RDF-NS transformation as the transition is named with inconsistent capitalisation and while using spaces and colons. The resulting `OrderFlow` is quite complex, with 38 data states and 14 tasks (and data flow tasks).

7.3.3 Lucient: Sales App to NS

The transformed NS model from the input conceptual model for Sales Application in ORM by Lucient is quite simple but shows how the fact modelling is handled. For entity types, `Customers`, `Orders`, `Products`, and for objectification `OrderDetails`, a corresponding data element is created. We had to handle the pluralised form in the name, as in NS the names should be in the singular form. Also, value fields for identifiers as part of entity type definitions are included. We enhanced the initial mapping based on this model to use names of value types as the name for generated value fields instead of predicate naming. Then, all the value fields used fallback `String` value field type. Therefore, an optional name-based value field type resolution has been added to the mapping for ORM resulting in types assigned as in Figure 7.11; for example, value fields that have a name ending with *Date* result in having `Date` type, or names such as *Quantity* or *Count* will be related with type `Integer`.

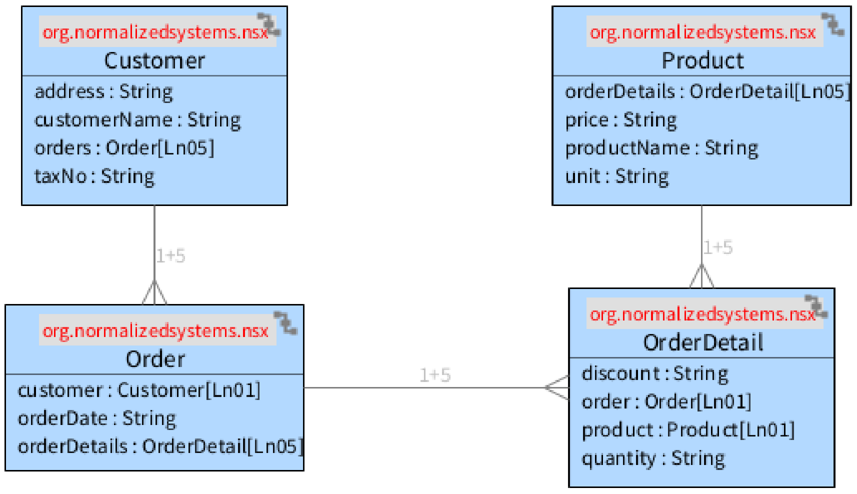

Figure 7.11: Resulting data elements for the Lucient case in NS

According to the mapping, binary facts between entity types (and objectification) were transformed into link fields (in both directions of the fact). There are no other types of facts (e.g. ternary that would result in an additional data element). Again, instead of using a predicate name that forms a simple sentence, the name of an opposite entity is used as common for NS modelling. For instance, the `Customer` data element has the `orders` link field and not `send`. The constraints of predicates resulted in a type of link field (e.g.

many-to-many) and field options, e.g. whether it is a required field. Despite the relatively small size, the core constructs of ORM were covered and successfully transformed into NS.

7.3.4 Craft.CASE: e-Shop to NS

The Craft.CASE example with the e-Shop resulted in a relatively complex NS component. It contains 10 data elements transformed from class information (non-diagram modelling in Craft.CASE) and OR diagrams. The structural part is naturally less rich, as BORM is focused on business process modelling. Thus, the behavioural part captured in NS is more interesting in this sense. There are 13 flow elements, one for each role in every OR diagram. The name is composed using the role name and the OR process name, e.g. `CustomerPlaceOrder`. Similarly, status fields are named according to the process name since one data element may have multiple flows, for instance `Customer` has three such value fields: `statusPlaceOrder`, `statusUpdateDelivery`, and `statusWriteDelivery` (as also shown in Figure 7.12). Consequently, an instance of such a data element cannot simultaneously participate in multiple flow instances of the same flow element.

Figure 7.12 also clearly shows that the input model has several flaws in its structural part. Although BORM in Craft.CASE provides a limited class modelling (e.g. both relations and attributes are captured as "variables"), and some information is plainly missing in the input model. Mainly, there are no relations between `Customer`, `VanDriver`, `Supplier` and the rest of the classes. Then, `WebsiteOfFD`, `FDDatabase`, and `LogisticsManager` do not even have class specified, and the data elements are created just through roles in processes (capture in OR).

Another naming-related issue arises with task elements and data states. First, some of the names are very long, for instance, 61 characters for a task resulting from *Update Delivery* OR. Second, the names of states and activities are not unique in different OR (and, in fact, do not have to be unique even in a single OR). For instance, the activity `is started` and the state `started` are in both *update delivery* and *write delivery* ORs. To overcome this issue, we further refined the naming fixing procedure in RDF-NS transformation ensuring uniqueness where necessary by adding a numeric suffix to the transformed name fragment (before adding other suffixes, e.g. `Task`). This results in (based on the configuration) `WebsiteOfFDIsStarting01Task`. Still, the original name is kept in options, e.g. in this case, as a task option.

7.3.5 Litium ERP Connector to NS

The Litium case has two UML class diagrams, one related to pricing is trivial, whereas the other one is relatively complex and covers structured around orders. As mentioned in the case description, the models are pretty technical and oriented toward data integration, still with some missing details; thus, it provided a slightly different use case than for UML Examples or Modelio cases. The Pricing Entity Model has just two classes (transformed to data elements) related via aggregation (link fields and data child). The Order Entity Model resulted in a component with 22 data elements where 7 are taxonomy data types

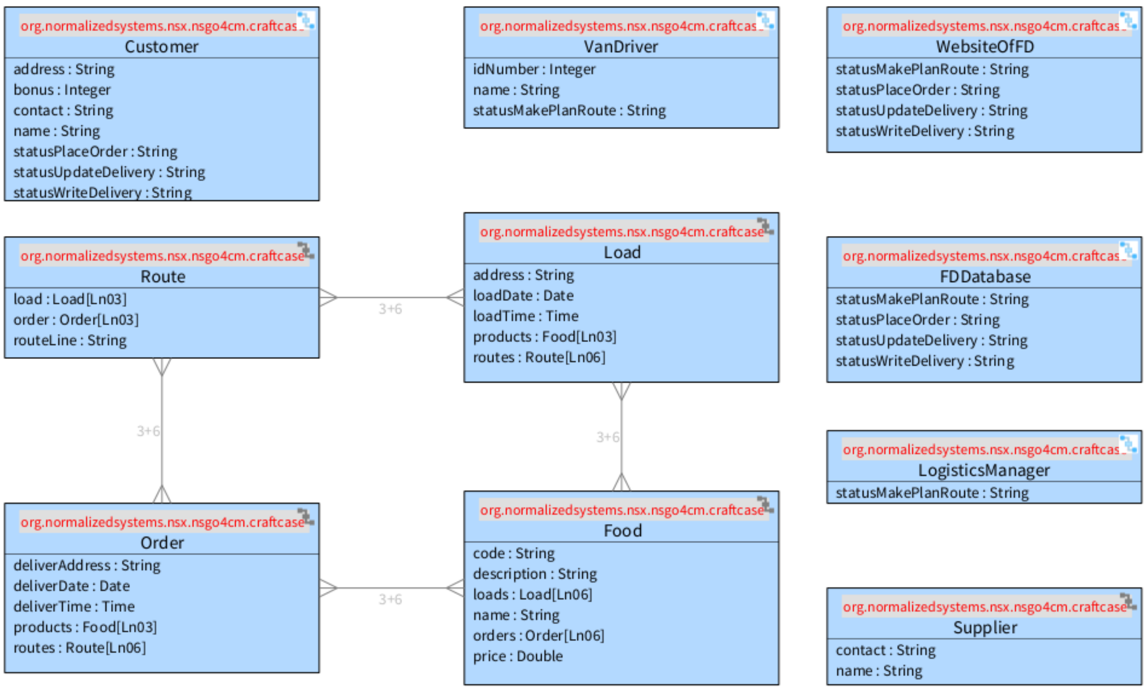
org.normalizedsystems.nsx.nsgo4cm.craftcas
Customer
address : String
bonus : Integer
contact : String
name : String
statusPlaceOrder : String
statusUpdateDelivery : String
statusWriteDelivery : String
org.normalizedsystems.nsx.nsgo4cm.craftcas
VanDriver
idNumber : Integer
name : String
statusMakePlanRoute : String
org.normalizedsystems.nsx.nsgo4cm.craftcas
WebsiteOfFD
statusMakePlanRoute : String
statusPlaceOrder : String
statusUpdateDelivery : String
statusWriteDelivery : String
org.normalizedsystems.nsx.nsgo4cm.craftcase
Route
load : Load[Ln03]
order : Order[Ln03]
routeLine : String
org.normalizedsystems.nsx.nsgo4cm.craftcase
Load
address : String
loadDate : Date
loadTime : Time
products : Food[Ln03]
routes : Route[Ln06]
org.normalizedsystems.nsx.nsgo4cm.craftcas
FDDatabase
statusMakePlanRoute : String
statusPlaceOrder : String
statusUpdateDelivery : String
statusWriteDelivery : String
org.normalizedsystems.nsx.nsgo4cm.craftcas
LogisticsManager
statusMakePlanRoute : String
3+6
3+6
3+6
3+6
org.normalizedsystems.nsx.nsgo4cm.craftcase
Order
deliverAddress : String
deliverDate : Date
deliverTime : Time
products : Food[Ln03]
routes : Route[Ln06]
org.normalizedsystems.nsx.nsgo4cm.craftcase
Food
code : String
description : String
loads : Load[Ln06]
name : String
orders : Order[Ln06]
price : Double
org.normalizedsystems.nsx.nsgo4cm.craftcase
Supplier
contact : String
name : String

Figure 7.12: Part of the Craft.CASE e-Shop from BORM in NS

representing enumerations. Most of the relations are compositions that, as in other cases, were transformed into link fields and data child. The resulting components can be used for data integration purposes, as is also the intention of the models.

We encountered several issues related to the input model. Some of the data types use collections such as List or Dictionary (almost every class has an attribute `AdditionalInfo` of type dictionary, where the key is a string and the value is an arbitrary object). For these types, the fallback to string type is acceptable as the collections may be serialised into a string for data integration purposes. We also had to extend the types to support the question mark for an optional attribute (non-standard). There are non-standard links between some entities, probably relating using an attribute that stores the ID of the other entity; however, it is not modelled as an association. The transformation omitted these links. Finally, there is a note attached to `ShippingInfo` class describing what can appear in the aforementioned `AdditionalInfo` attribute; we refined the mapping to include class-related notes as data options.

As the BPMN model is not clearly linked to the UML class diagrams, it rather created another component with its data elements, but mainly flow and task elements. More specifically, there are three data elements that represent the pools with related flow elements: `Buyer`, `Litium`, and `ERPConnectorApplication`. According to the mapping, tasks, gateways, and events are task elements. The naming is based on the task name, e.g. `SendOrderConfirmationEmailTask`; here, we had to improve the name refining step again to convert it properly to *PascalCase*. The communication and other behaviour of created task elements (for instance, sending/receiving a message, parallel branching, or waiting) must be implemented in custom code. The name of states is based on the names of the related tasks by adding the appropriate suffix `Done`, `Failed`, and `Working` as shown in Figure 7.13.

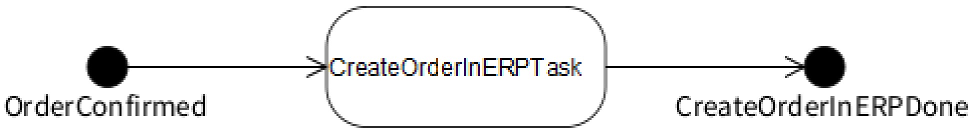

Figure 7.13: Simple `ERPConnectorApplication` flow transformed from BPMN to NS

7.3.6 Results and Limitations

This section briefly discusses the transformation of the selected input models to NS with a focus on the identified challenges, issues, and refinements performed throughout the design cycle. When using structural input models (e.g. UML class diagrams), the data elements with related value fields, calculated fields, and link fields were created by transformation up to the expectations. For aggregation, the use of data children resulted in an excellent part-whole result in generated systems (e.g. waterfall screens in User Interface (UI)). The mapping of data types to value field types was refined on the selected models and can be refined in the same way in the future if needed; fallback to the `String` value field type is identified as suitable. Additional information for constructs without directly mapped counterparts in NS is captured through options; thus, it may be used in expanders. For example, to generate specific source code fragments implementing constructs not directly supported in the NS modelling (yet).

The results with fact modelling in ORM are very similar to structural modelling. Limitations are related mainly to complex constraints that are captured as options. Behavioural models (in our case UML activity diagram, BORM and BPMN models) resulted in flow and task elements, and associated data states and data flow tasks. One issue is that the input behavioural models must be clearly related to (and consistent with) the associated structural model; otherwise, the flow elements are attached to the data elements without any other than `status` fields. The actual behaviour of tasks must be implemented as custom code, including waiting, communication, or branching – that may be a potential future work on working with options and expanders as mentioned above. It is related to currently limited flow modelling in NS, where it is not possible to express a connection of different flows (e.g. passing data between them or synchronising).

The limitations caused by the naming restrictions and conventions in NS that are not enforced in conceptual models (e.g. *PascalCase* for naming elements or uniqueness requirements in certain contexts) were solved by adding a step (and in the design cycle refining) to rectify names during RDF-NS transformation. As the RDF-NS transformation creates components, those can be added directly to the NS applications, potentially refined and expanded. As expected, we expanded the applications for our demonstration examples, resulting in typical Create, Read, Update, Delete (CRUD)/workflow applications.

One of the essential properties for this use case is the absence of input size limitations, i.e. the transformation works with any number of constructs (for instance, hundreds of classes in UML and thousands of attributes) without any changes needed in the mapping or tools. However, it is typical that such huge models are modularised into parts, e.g. packages, resulting in multiple smaller components in NS, which promotes reusability. The size does not matter, but compliance with the modelling language specification (i.e. metamodel) does. If the model contains such constructs, they will naturally be omitted, as they are not described in the ontology based on the metamodel; therefore, they are not assigned to NS. A similar issue is with the unexpected types such as we had in the Litium example (e.g. `List<String>` instead of `String` with cardinality); the use of well-known types is needed. Alternatively, the mapping of types must be enriched before the transformation, and that requires investigating the input model, e.g. manually or via SPARQL Protocol and RDF Query Language (SPARQL) when the model is in RDF.

7.4 Semantic Integration with BORM Model

The second scenario still focuses on the transformation from conceptual models to NS but together with the semantic integration of the input models. We have different models related to the same domain (e-Commerce); thus, by relating the models, a more detailed description of the problem domain can be used to create a software system. We first show how the integration of models in RDF is carried out together with its implications for mappings. Then, we do the transformation and evaluate the resulting model in NS.

The semantic integration of models prior to their transformation to NS is related to our objective RO1. Using RDF as *lingua franca* to capture knowledge from conceptual models allows the integration of concepts, e.g. relating them or specifying matches. Then, it is possible to use the Gateway Ontology with adequate mappings to transform this (in an ideal case) holistic view to NS.

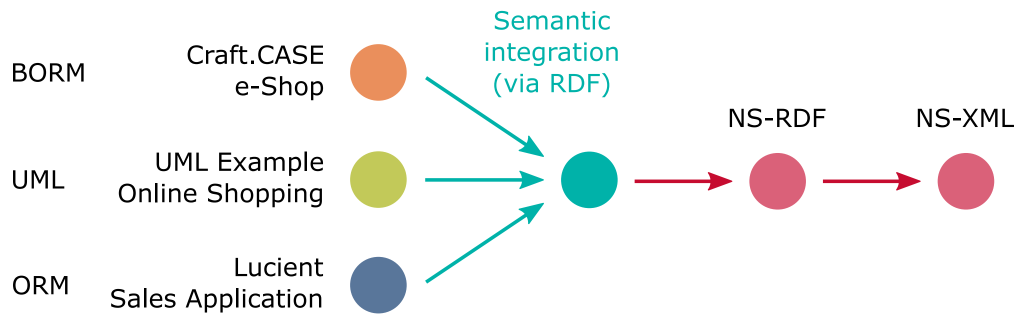
BORM
Craft.CASE e-Shop
UML
UML Example Online Shopping
ORM
Lucient Sales Application
Semantic integration (via RDF)
NS-RDF
NS-XML

Figure 7.14: Conceptual models semantic integration and transformation

In terms of inputs and outputs, we take three different models and first semantically integrate them on the RDF level; then we proceed as in the previous scenario and as shown in Figure 7.14. The selected input models use different modelling languages, but have overlapping concepts. In addition to the NS model in XML and the intermediary NS /RDF, the RDF data set containing the integrated input model is also output for this scenario.

7.4.1 Integrated Input Conceptual Models

To demonstrate integration, we selected the Craft.CASE e-Shop in BORM (rich behavioural model) to integrate with UML class diagram based on UML-Examples and classes from the Craft.CASE e-Shop. To further enrich the model, we also integrated the ORM model for the Sales Application. These modelling languages do not use common metamodel or compatible formats; thus, integration is not possible in their original form. Here, we take advantage of our design and integrate the models in RDF, which is suitable for such use cases (that are common in area of linked data).

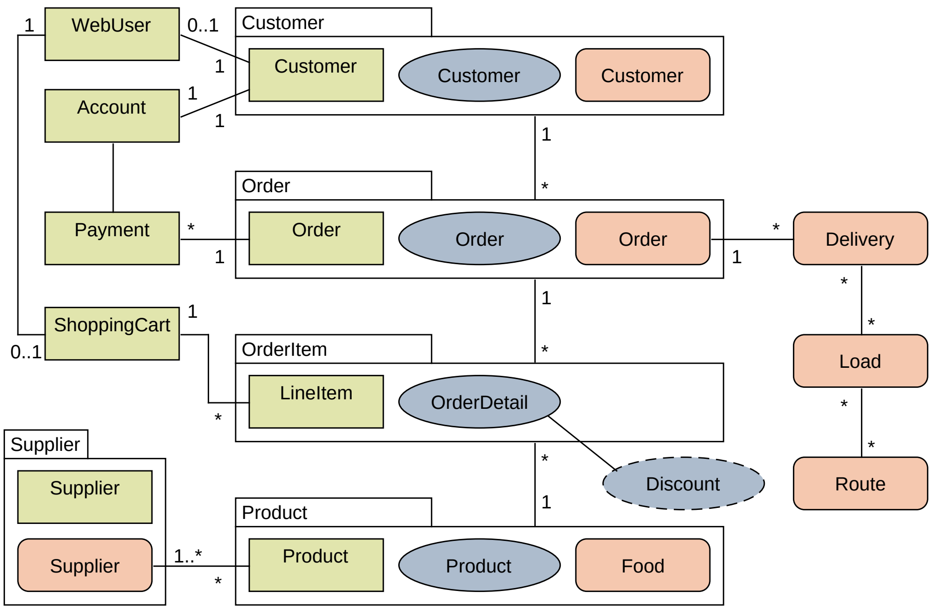

Figure 7.15: Visualisation of integrated concepts from three different conceptual models

We used manual semantic integration, where we identified the relations between concepts originating from different models. For example, *Customers* from the ORM model represents the same concept as *Customer* from the UML model and *Customer* from the BORM model. The identified related concepts that become part of the new integrated model are visualised in Figure 7.15. For this relatively small case, manual integration is feasible and ensures well-defined relations between models. Another possibility is to use (semi)automatic integration that utilises natural language processing (NLP) techniques such as searching for synonyms or guessing relations using corpus or thesaurus. However, since our goal here is to have well-integrated models rather than to evaluate the possibilities of semantic integration itself, we had to do this manually.

7.4.2 Mapping and Transforming Integrated Models

For the first transformation, we simply used all three mappings together. There was no need to specify any additional mappings, as semantic integration did not introduce any "new" constructs or properties that would unite the concepts. We produced both an enriched RDF dataset that contains input triples and output/NS triples and an NS-only output dataset.

The second transformation was used in the same way as for the first scenario (with non-integrated input models). That again shows the separation of concerns and reasons for separating the two transformations (and their implementations). As expected, the resulting NS models for both types of input (outputs of the first transformation) were identical to the NS triple in the matching of the data sets.

7.4.3 NS from Integrated Models

The resulting NS model in XML from the two sets of transformations described above is a match to a union of the three models transformed separately in the first scenario (Section 7.3). It is because we used the merged mappings and semantically integrated the input models without adding new constructs. The issue is with the different naming of concepts resulting, for instance, in a data element name or a field name. However, this information (or rather preference) is captured by the order of the mappings merged together; for example, that class name from UML has a higher priority than the fact name from ORM. In this way, all overlapping properties are resolved.

The result shown in Figure 7.16 met the expectations. It covers the integrated concepts based on Figure 7.15 and adds the extra concepts from the original models. Although integration can combine different concepts together into a single application, it can provide a way to easily combine behavioural and structural modelling, as visible from this case – BORM provided processes (flows and tasks) and UML together with ORM contributed to the structural description. In conclusion, an analyst is needed to either properly define the semantic integration while omitting unwanted but related concepts from the original models or manually adjust the resulting model based on final needs.

7.5 Evolvability and Consistency

The third scenario demonstrates how the solution provides support to adopt different changes, i.e. cope with various change drivers. When we specify the transformation between a conceptual modelling language (and its metamodel) and NS (and its metamodel), the mapping is inevitably dependent on the metamodels. Moreover, it also depends on the mapping specification, in our case, on the Gateway Ontology. We first deal with a more natural evolution of the input conceptual model; we have the conceptual model and transform it into NS, but then we need to make changes in the conceptual model again. Then we show the ability to adopt changes in mappings and metamodels.

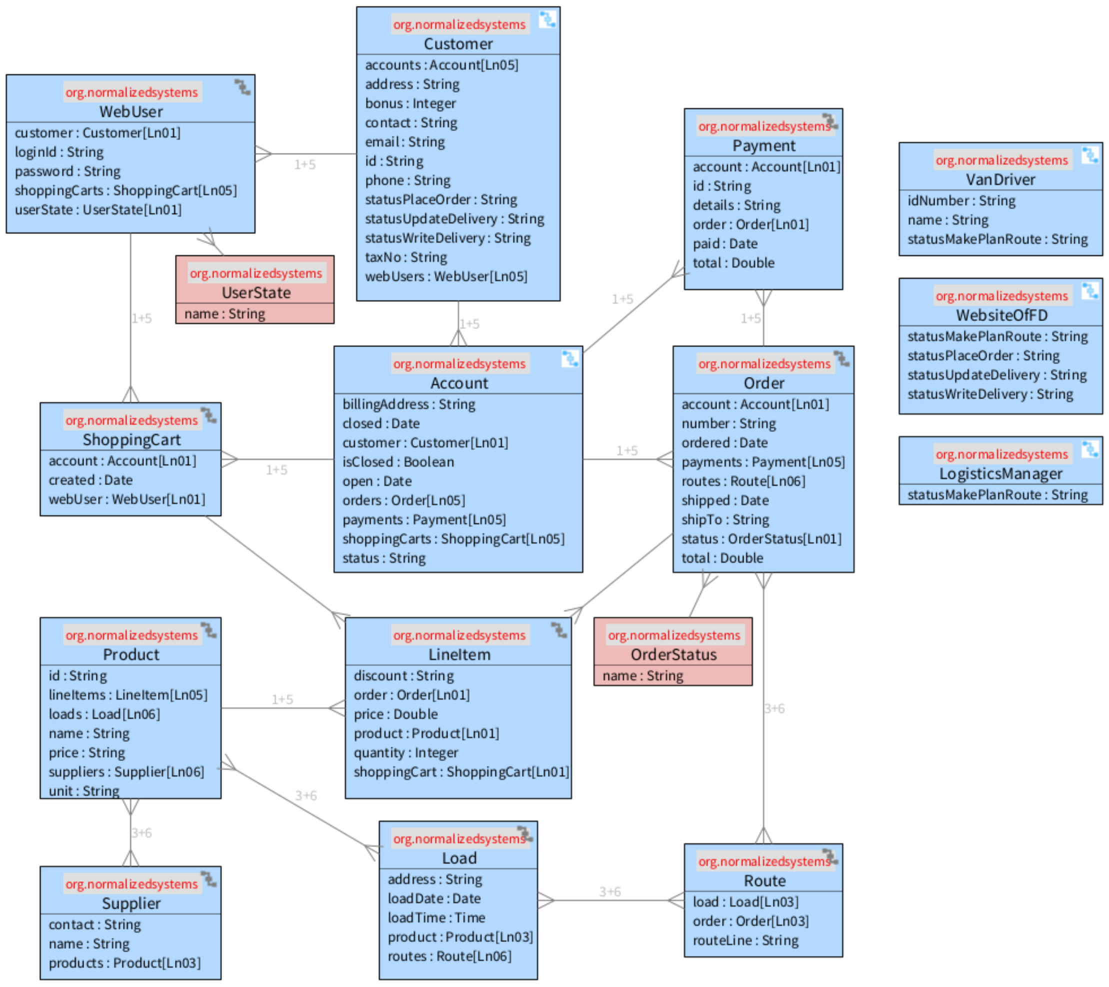

Figure 7.16: Data elements transformed from integrated conceptual models

This scenario explains how our solution addresses RO3 and RO4. The mitigation of change impact and actual adaptability to different types of changes allows evolvability as described in RO3. Closely related to evolvability is the ability to maintain or verify consistency according to RO4. We show how the links between models can be kept and used to check and maintain consistency.

In this scenario, we use the outputs of the first scenario as inputs and trigger various change drivers as depicted in Figure 7.17. For example, we have an already transformed conceptual model to NS in XML (with intermediary NS NS in RDF) and we make an additional change in the original conceptual model – then, we observe how the change affects the other parts and how it can be propagated and reflected in the resulting NS model in XML. The output of this scenario is the resulting state of all transformed models after accommodating the changes.

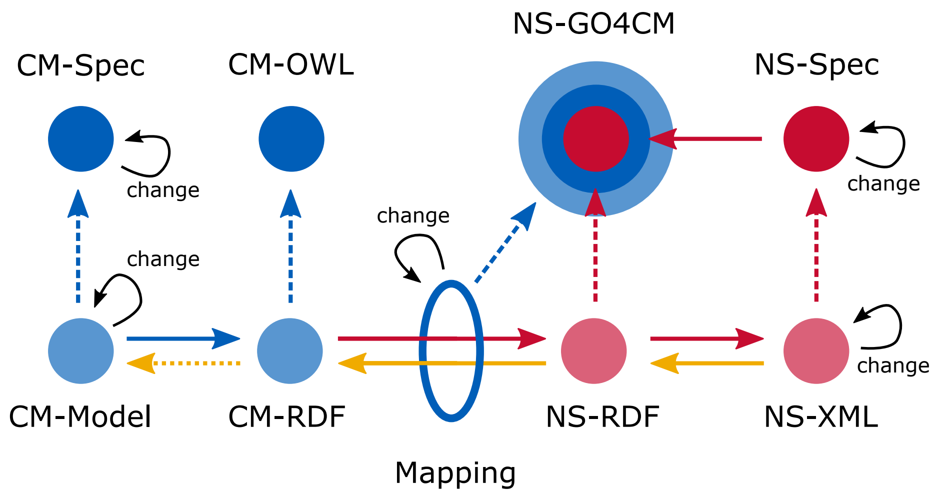

Figure 7.17: Change drivers and evolution within transformations

7.5.1 Adopt Changes in Conceptual Model

The most direct change that may occur is an update in a conceptual model that was used as input for the transformation. A real-world domain evolves, or the model becomes more precise, resulting in such changes. The new model can be used to re-generate the NS counterpart; possibly, a merge will be needed with existing changes made on the NS side (e.g. technical details or custom code craftings).

We demonstrate this type of change using the previous UML class diagram example for Online Shopping domain (Figure 7.3). We first transformed the model to NS and made a few changes there: added craftings, application, and component technical information, and manually added a simple flow to the `Payment` data element, including a `status` field. There are several things to improve the model:

- added super class `ItemSelection` abstracting the common relation to `LineItem` and turning it into composition,
- added `unit_price` to `Product`,
- added new class `Supplier` (related to `Product`),
- added custom datatype `Address`,
- changed `total` of `Order`, `price` of `LineItem`, and `is_closed` of `Account` to be a calculated field,
- removed `phone` attribute of `Customer`.

These changes represent various typical evolutions that may occur: simple additions, complex refactoring using abstraction, changes in types, or removal of some concepts. After incorporating the changes in the input model, more specifically its RDF representation, we transformed it to NS in the same way as in the first scenario. We used two possibilities to merge the changes with the previously transformed model. First, we merged the models using well-known VCS Git – created a branch at the point of the previous transformed version and added the new one as changes, then merged with the parallel changes made by other NS tooling (technical details and harvested custom code). Second, we use the reverse transformation from NS to RDF and then enrich it with the new input model. The first way was, in our case, without issue; in the second, we identified a problem with removing information from the resulting RDF (e.g. `phone` field) remained. We improved the NS-RDF transformation to detect such changes using the stored options in the previous transformations. For both ways, we achieved a component as shown in Figure 7.18.

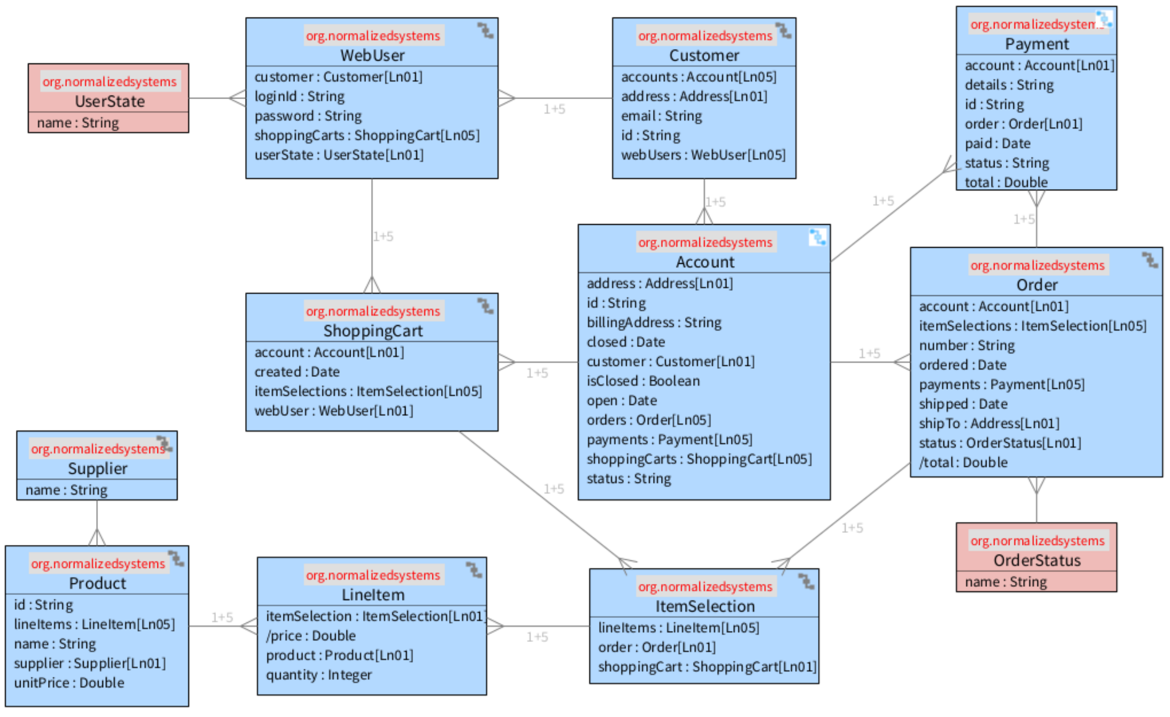

Figure 7.18: Data elements transformed for evolved Online Shopping UML class diagram

There was no issue with harvested custom code craftings, as we did not change (remove or rename) the related elements nor change package configuration. In cases where a change that affects craftings naming or location occurs, manual intervention is required – moving and renaming the file accordingly. The reason is that the craftings are not coupled with the model in the RDF representation, which would automatically allow such rectification.

7.5.2 Adopt Changes in a Modelling Language

A modelling language specification may evolve over time. Some languages use versioning of their specifications (and metamodels), e.g. UML or BPMN; however, some do not version explicitly. If a change in a specification occurs, it may trigger a change in our mappings (if the construct is used). Preferably, it should result in a new OWL ontology, and thus a new mapping as a released language specification (version) should be immutable. Still, the previous version of both ontology and mapping can be re-used. It consequently results in an easier "upgrade" of the underlying conceptual models.

To demonstrate how our solution can adopt this type of change, we use BPMN versions 1.2 [171] and 2.0 [158]. It allows us to demonstrate additions, deletions, and edits in the specification. More specifically, we will focus on the following changes between the BPMN versions:

- added non-interrupting events for a process and event sub-processes for a process,
- removed reference tasks with suggested replacement by using a (newly added) call activity and a global task.

To support BPMN 2.0 while already having BPMN 1.2 supported, the first step is to create a new OWL ontology for the new version. As the previous version is "frozen", it can be copied and adjusted as the new version, i.e. with its new base URI (changed version fragment) and its own ontology metadata. Then, in the new (not-yet-released) ontology, the changes of the new specification can be mirrored. In our case, we add the new types of events, event sub-process class, and remove the reference task construct together with its properties. When everything is updated, the ontology can be released and "frozen".

In our case, we also wanted to update the mapping and a simple BPMN model. The procedures were identical to the updates in the ontology, creating new RDF datasets based on the previous and changing the used BPMN ontology. This demonstration use case helped us refine the versioning of the ontologies for conceptual modelling languages. The ontology must have its own version and be related to a version of the language specification rather than having just one version of the specification. For example, when we released and adopted the ontology for BPMN 2.0, then identified an error (such as typing error or incorrectly captured construct), we need to update the ontology while still referring to BPMN 2.0. Then, the mapping must be tied to the version of a mapping that supports a particular language version and not directly to the language version.

7.5.3 Adopt Changes in a Mapping

As mentioned in the previous case for ontologies of the conceptual modelling language, a mapping also has its own versioning independent of the versioning of both supported modelling language(s) and the Gateway Ontology (refer to them instead). There may be external and internal reasons to introduce changes to a mapping that result in its new version. For external, a change in an ontology for conceptual modelling language or in the Gateway Ontology may occur and require adoption in mappings; for example, the extensions layer is enriched with additional constructs to promote DRYness or a new version of UML specification is available as the OWL ontology. On the other hand, an internal change is related to improving how the mapping is captured, e.g., increasing coverage of input/output constructs while keeping the same version of both conceptual modelling ontology and Gateway Ontology.

This type of change frequently occurred during our design cycle iterations. For the demonstration, we selected the improvement of our UML class diagram mapping, where we added the aforementioned aggregation relation with the data child construct of NS. The mapping has been extended by adding a sub-mapping that creates a data child for each aggregation (and composition) on the side of a whole. There was no change impact on existing parts of the UML-NS mapping. After creating this new version of a mapping that includes the new feature, the input UML models were transformed (again), and the result was observed. If a change also impacts other existing parts of the mapping, the appropriate change would also be needed there. However, mapping modularisation helps to reduce such a negative impact.

7.5.4 Adopt Changes in the NS Metamodel

Whenever a change occurs in the NS metamodel and its new version is released, the change affects the Gateway Ontology directly and via that also the mappings. To simulate this, we created an adjusted metamodel with removed data child construct, added a flag to the link field to indicate aggregation, and changed link field types. If the new version is to be adopted by our solution; the steps were used as follows:

1. The NS-RDF/OWL tool was rejuvenated with the new NS metamodel, which resulted in its new version.

2. The new NS-RDF/OWL tool was used to create RDF/OWL representation of the new NS metamodel (as the core layer of the Gateway Ontology).

3. With the updated core layer, both extensions and transformations layers were also updated, resulting in a new version of the Gateway Ontology as a whole.

4. After having the new version of the Gateway Ontology, it was possible to update the mappings in the same way as when a conceptual modelling language is updated.

As a result, we were able to propagate the change from NS metamodel to mappings and transform the input models into NS using this new version. Due to the modularisation on all levels and mainly in mappings, the changes were atomic. For instance, in mapping, we just changed the sub-mapping related to the data child construct and to the link fields. The changes we used in the demonstration are relatively radical in the sense that NS metamodel is not expected to remove existing constructs as we did simply. Nevertheless, it clearly shows the impact on our solution.

7.6 Reverse Engineering Normalized System

The fourth and final scenario shows the opposite direction of transformation, i.e. how a model in NS (e.g., an existing information system) can be transformed into a conceptual model. We use UML as the target modelling language and the custom e-Commerce Normalized Systems component to demonstrate this feature. After transforming the model via reverse mapping, we discuss the resulting UML models. Also, we show how consistency is maintained with respect to the previous scenario.

We address again RO1 in this scenario, more specifically its "and vice versa" part. Furthermore, we can also check the consistency of this reverse transformation, as we have already shown with respect to RO4.

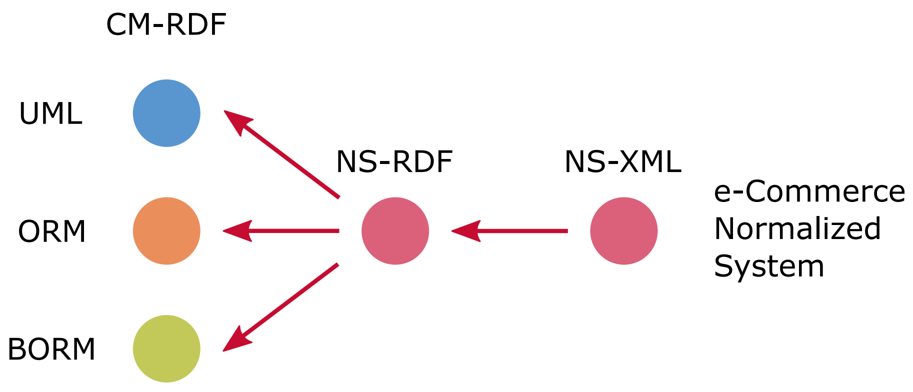

Figure 7.19: Reverse transformation from NS to conceptual models

An input in this scenario is an NS model in XML (created using NS modelling tools). Then the transformation is reversed compared to the first two scenarios. NS-RDF/NS transformation is used to obtain NS in RDF. It is then used for transformation to various conceptual models (in different modelling languages). The second transformation utilises inverted mappings. Both the intermediary NS in RDF and the conceptual models in RDF are outputs of the scenario.

7.6.1 Normalized e-Commerce System

Based on the experience with the input conceptual models in the previous parts, we created an NS application for e-Commerce solution. It covers the key concepts of the conceptual models together with their properties and relations. However, we designed it in a way suitable for NS as if we want to expand it and use the enterprise information system. There are three components: *orders* (Figure 7.20), *products* (Figure 7.21), and *shared*.

The component *orders* covers user accounts, payments, ordering, and delivery. It uses *products* and *shared* as component dependencies to enable reference to `Address` and `Product` data elements. Similarly, the *products* component depends on *shared*. It covers products, their categorisation, parameters, brands, and suppliers. Finally, the *shared* component contains only a definition of the `Address` data element.

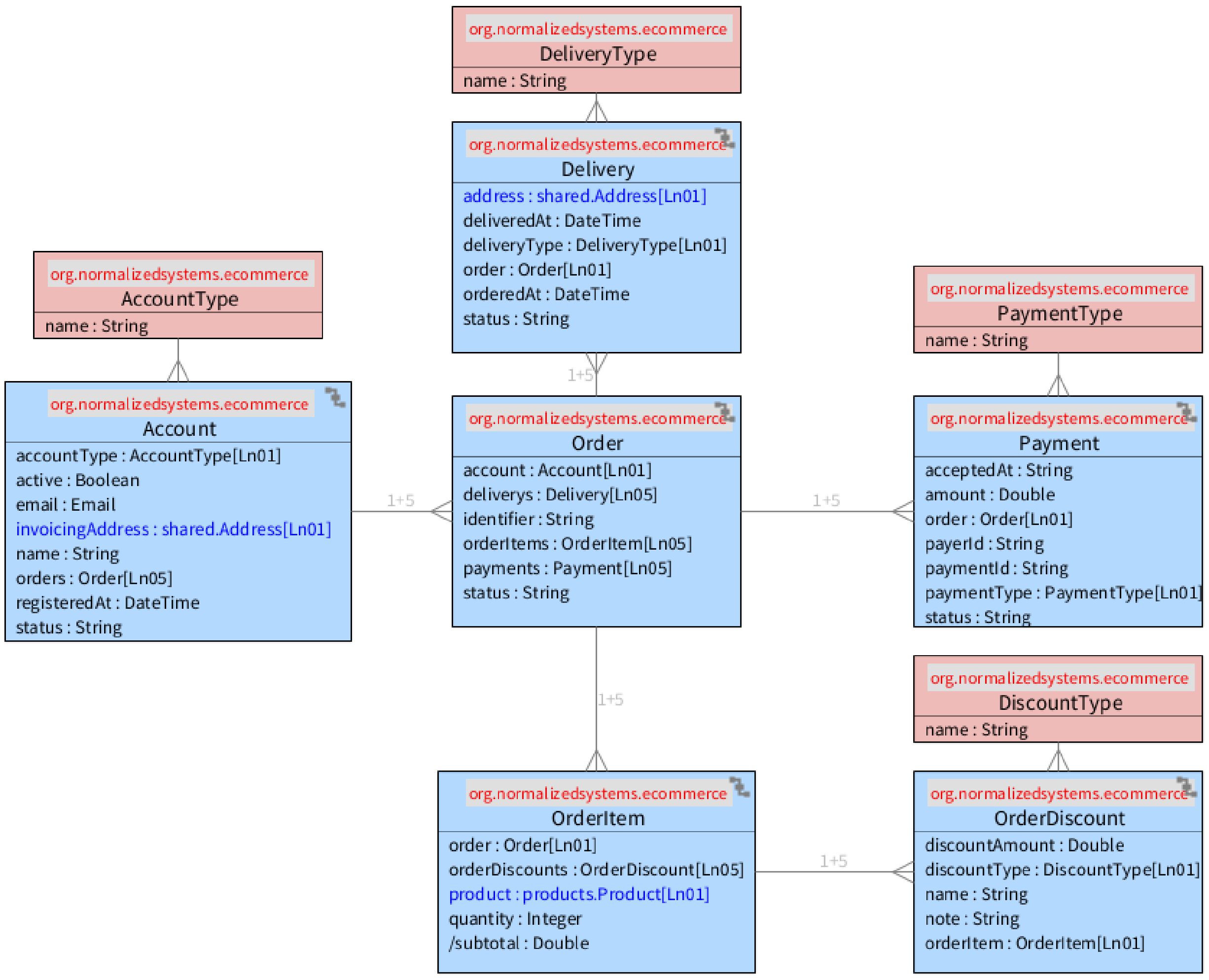

Figure 7.20: Orders component of eCommerce NS Application

There are also various constructs used that are not visible from the provided visualisation; however, they are taken into account in mappings and thus transformations. For example, `Order` has data child `OrderItem`. Another important construct is the field option `isRequired` used for both the value and link fields. Last but not least, there are also flows and tasks related to `status` fields of various data elements.

7.6.2 Reverse Transformation to UML via RDF

The first step of transformation from NS to UML is to use the NS-RDF/OWL tool. The resulting RDF dataset contains the model of e-Commerce solutions (all its components) encoded using the NS metamodel ontology, i.e. core of the Gateway Ontology. To quickly verify how many constructs (e.g. data, task, and flow elements) were transformed, we used

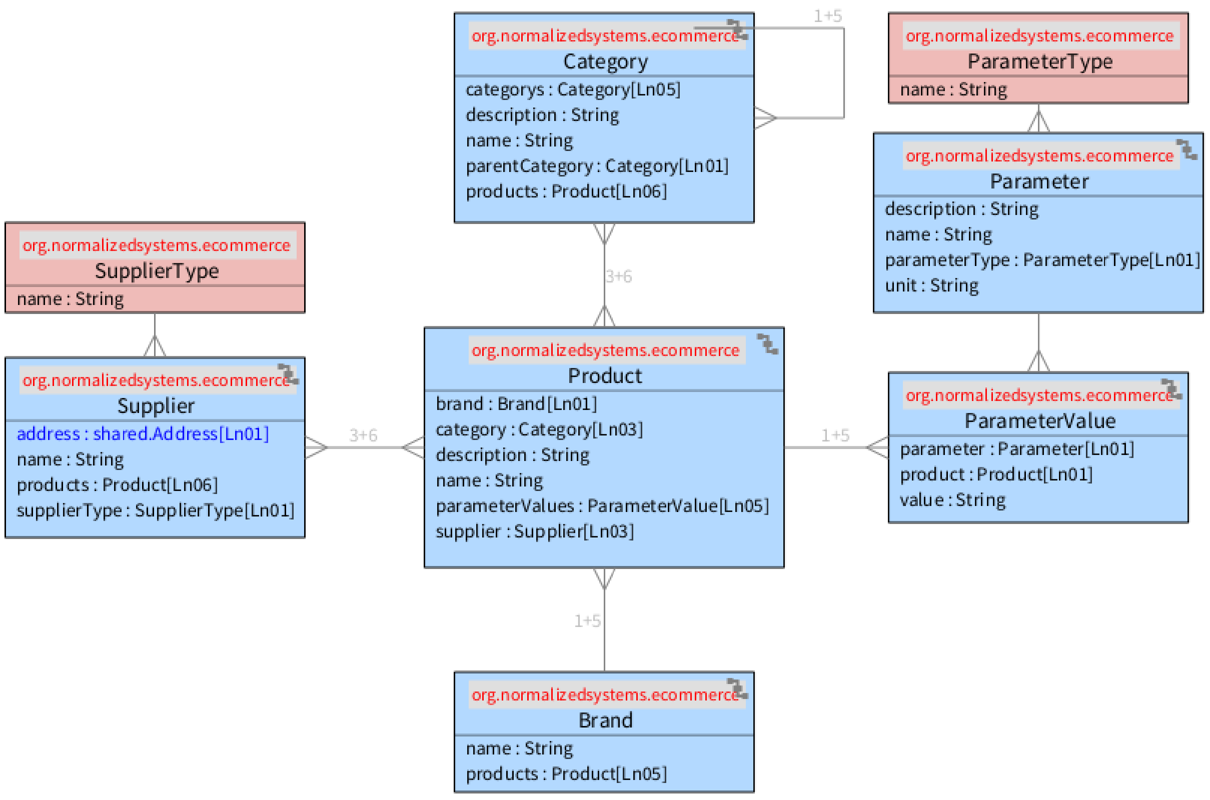
org.normalizedsystems.ecommerce
Category
categorys : Category[Ln05]
description : String
name : String
parentCategory : Category[Ln01]
products : Product[Ln06]
1+5
org.normalizedsystems.ecommerce
ParameterType
name : String
org.normalizedsystems.ecommerce
Parameter
description : String
name : String
parameterType : ParameterType[Ln01]
unit : String
org.normalizedsystems.ecommerce
SupplierType
name : String
3+6
org.normalizedsystems.ecommerce
Supplier
address : shared.Address[Ln01]
name : String
products : Product[Ln06]
supplierType : SupplierType[Ln01]
3+6
org.normalizedsystems.ecommerce
Product
brand : Brand[Ln01]
category : Category[Ln03]
description : String
name : String
parameterValues : ParameterValue[Ln05]
supplier : Supplier[Ln03]
1+5
org.normalizedsystems.ecommerce
ParameterValue
parameter : Parameter[Ln01]
product : Product[Ln01]
value : String
1+5
org.normalizedsystems.ecommerce
Brand
name : String
products : Product[Ln05]

Figure 7.21: Products component of eCommerce NS Application

a simple SPARQL query. It also demonstrates the basics of possibilities to analyse NS models transformed to RDF in terms of reverse engineering.

To allow for the reversed transformation from NS-in-RDF to UML-in-RDF, we had to improve the mapping to support bi-directionality. Instead of creating a totally new mapping, which would be possible but more laborious, we added its inverted variant for each of the sub-mappings. Then, we were able to use the mapping and execute transformation resulting in UML. We tried both pure UML-in-RDF and enriching mode that adds UML-in-RDF to the input NS-in-RDF.

7.6.3 Resulting UML Model from NS

The target was set to have the class and state machine diagrams in UML separated into packages according to the input components. To finalise the transformation from NS to UML, we transformed UML-in-RDF to UML class diagrams and state machine diagrams using computer-aided software engineering (CASE) tool Enterprise Architect. This transformation has been done manually similarly to what we transformed selected e-Commerce UML diagrams, e.g. for the Litium case, to RDF. Although it carries the same information as is encoded in RDF, visualisations are essential in reverse engineering.

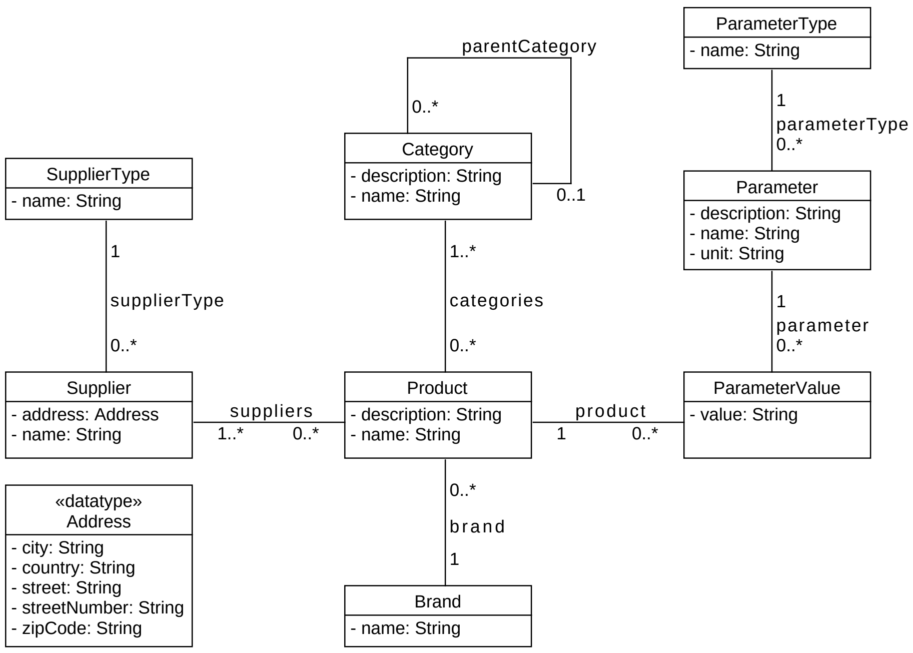

Figure 7.22: Resulting UML class diagram from Products component

7.6.4 Consistency for Reversed Transformation

While working on this case, we iteratively improved the URI composition while transforming from NS to RDF. The main cause is that while transforming an NS model, there are constructs that are not used in the NS metamodel itself, for example, the data child construct. The URIs identifying constructs in RDF are based on context and naming. A potential future improvement would be the identification via Universally Unique Identifier (UUID) that is part of NS model directly.

The consistency can be maintained as long as the URI remains persistent for the construct/concept it identifies. For example, a data element `Order` has its URI which is used in both NS-in-RDF and UML-in-RDF. Then, when we update the counterpart UML or NS, the construct in RDF can be updated via URI.

7.7 Demonstration Summary

In this chapter, we demonstrate how all the designed artefacts could be used with respect to our research objectives formulated in Section 1.2. As shown, the proposed NS Gateway Ontology for Conceptual Models allows transforming between conceptual models and NS models while maintaining consistency and enabling evolution at different levels. The complete input, results, and documentation in Appendix A serve as an example of operations for artefacts and proofs of its capabilities in different use cases.

Concerning Design Science Research (DSR), this part of our work is also essential. First, it helped to evaluate and refine the artefacts during their development (in the design cycle according to Figure 1.3). Second, since it describes real-world scenarios related to our target environment, it also serves as a field testing of the relevance cycle. Finally, documented use cases contribute to the knowledge base via the rigour cycle.

Chapter 8

Main Results

"Accomplishment will prove to be a journey, not a destination."

Dwight D. Eisenhower

This chapter provides a concise discussion of the main results and how they were achieved in terms of Design Science Research (DSR). In the previous chapters, we presented the designed artefacts that together form our solution for the transformation between conceptual models and Normalized Systems (NS). Here, we discuss these by-design reusable artefacts, their usability in our as well as other use cases together with other consequences of their (re-)use in practice and future research.

8.1 Applied Design Science Research

When summarising our results, it is important to retrospectively evaluate how the results were actually achieved by applying to the DSR methodology. It guided our research from the problem statement to the demonstrated and evaluated that meet the research objectives. We followed the process of DSR captured in Figure 1.2 according to Peffers et al. [40] with the problem-centred initiation. The research objectives did not have to be adjusted during the work, although the process allows that. The main reason is that the topic of the dissertation thesis and its ensuing goals was fixed after the initial study based that helped to consolidate the objectives.

In Chapter 1, we formulated the problem statement as well as outlined the research objectives. The design and development of artefacts is the “centre of gravity” of this dissertation thesis. First, Chapter 3 covers the overall design and its decomposition into partial artefacts that together fulfil the research objectives. Then, Chapter 4, Chapter 5, and Chapter 6 describe the design and development of these partial artefacts together with its evaluation and discussion on process iteration related to design refinements. Finally, Chapter 7 demonstrates the use of partial, as well as whole, solution and aggregate evaluation. The last step of the process, the communication, is this dissertation thesis itself and the partial publication that are used as references.

Chapter 3 also specified the requirements according to Figure 3.1 based on the work of Braun et al. [41]. Despite the fact that the requirements may sound tightly related to software engineering and not as part of rigour research, their use significantly helped to modularise the research objectives, evaluate and refine the design artefacts. A vital aspect of the requirements is their relation to the existing knowledge base and target environment (i.e. domain and stakeholders) in the form of theory-based, contextual, and adapted categories of requirements.

In terms of functional requirements, we fulfilled and demonstrated all of them in Chapter 7. The theory-based and context-based requirements affected the design and realisation of the designed artefacts to follow the best practices and provide relevant results for the target environment. Finally, the adapted requirements (which were already mapped to our other requirements) connected our work with the previous solutions and conclusions of similar approaches. When we look back at the particular requirements, we can summarise their fulfilment as follows:

- FR1, FR2, and FR3 are accomplished as we do not place any limitations on the use of the conceptual modelling language use and our solution provides a versatile way of mapping and transformation execution for any knowledge represented in Resource Description Framework (RDF). Still, to proof the possibility for the required branches of conceptual modelling we covered structural (UML and OntoUML), behavioural (UML, BORM, and BPMN), and fact-based (ORM) modelling.
- The versatility and use of RDF also enables semantic integration (FR4), we presented an example where the knowledge from the conceptual model is integrated and then transformed to NS.

- All our artefacts are designed following the DSR methodology, and thus their design cycle may re-initiate (FR5) when new requirements emerge.
- The mapping specification, according to our design, efficiently supports bi-directionality (FR6). The consistency is also managed via our transformation design that enriches the input instead of providing detached output.
- NR1 and NR2 were addressed already in terms of FR5 and was achieved by following DSR.
- The design of our artefacts is based on standard technologies that are well-documented and thus suitable for re-use (NR3). With the publishing of the dissertation thesis the artefacts will be made accessible and findable together with the specific realisations of the design, e.g. OntoBORM ontology or tool for transformation.
- NR4 is addressed by the relevant artefacts. For instance, the NS-RDF/Web Ontology Language (OWL) tool is expanded not only for the evolvability reasons but to promote DRYness. Other examples are the extensions and transformations layers of the Gateway Ontology that encapsulate repeated patterns in the mappings and provide additional re-usable constructs.
- NR5 as well as NR7 were fulfilled by not relying on any particular technologies in the designs except semantic web technologies. Furthermore, the realisation of designed artefacts is also compliant with the requirements.
- The documentation (NR6) is this dissertation thesis itself together with its appendices as it describes completely the overall design and the design of the partial artefacts. The realisations are documented typically for their type (e.g. metadata and generated documentation for ontology or documentation comments in source codes).

As an important insight, it would be better to specify the requirements with measurable "hard" metrics for evaluation purposes. However, that was not achievable for our research objectives and ultimate goals. We could measure, for instance, the execution time needed for the transformation with different sizes of models. Nevertheless, such metrics would be artificial and not actually relevant to our goals. Therefore, our requirements served as something that must be fulfilled in the best possible way. In the minimum of cases where there are certain limitations, we always provide the necessary explanation. As a follow-up to this insight, we plan to work on a proposal of an evaluation framework related to the DSR methodology.

The procedures based on the three-cycle view already shown in Figure 1.3 also positively affected our work. We continually enriched (according to the rigour cycle) the relevant knowledge base represented by Chapter 2 as well as made partial contributions to it through publications of our works [A.1]–[A.18]. We also made other contributions [A.19]–[A.27] that does not directly relate to this dissertation thesis but undoubtedly provided helpful experience. The same applies to supervised theses of bachelor and master students [A.29]–[A.44] that were related to information system development, use or review of Model-Driven Development (MDD) methods, or complex software design. The relevance cycle was used to confront our artefacts with the NS environment, existing applications, models, libraries, and stakeholders. Finally, as discussed for each artefact, the design cycle helped to refine the artefacts, try different approaches, and let the artefacts mature into the final form as shown in Figure 8.1.

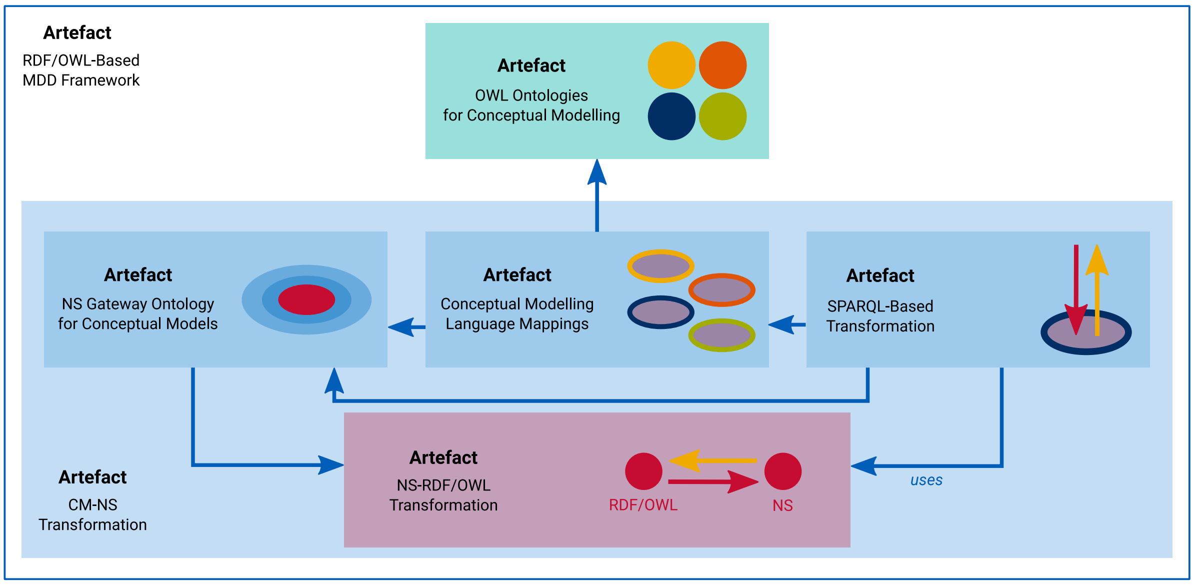
Artefact
RDF/OWL-Based MDD Framework
Artefact
OWL Ontologies for Conceptual Modelling
Artefact
NS Gateway Ontology for Conceptual Models
Artefact
Conceptual Modelling Language Mappings
Artefact
SPARQL-Based Transformation
Artefact
NS-RDF/OWL Transformation
RDF/OWL
NS
uses
Artefact
CM-NS Transformation

Figure 8.1: Designed artefacts and their dependencies

The guidance provided by the DSR methodology through our entire work on the dissertation thesis significantly contributed to the quality and relevance of the final results further presented in this chapter. Moreover, it helped to learn from existing previous work and let us contribute to the knowledge base for others. In this way, our results are also by-design re-usable for future work.

8.2 NS-RDF/OWL Transformation

The designed partial artefact of our framework that solves the transformation between NS and RDF as described in Chapter 4 is a treasure of this dissertation thesis on its own. It clearly demonstrates the power of NS theory, as well as its implementation in software. The design takes advantage of the provided NS infrastructure (which is itself expanded from the NS metamodel) and the meta-circularity. We generically specified the design with descriptions of algorithms used so it can also be used with other metamodels compliant to the NS metametamodel.

After the experience gained with the traditionally implemented tool, we shifted to incorporating the expanders to generate source code that actually performs the transformation. The manifestation of meta-circularity in the design is showing how NS enables evolvability. The NS metamodel is used to generate the transformation classes, but can also be used as an input to the tool that is generated from it. Moreover, it is feasible to develop other metamodels based on NS metametamodel and do the same for those. The expanders themselves are not even complex in the number of source code lines.

The design following the set requirements allows extensions and simplifies potential future development. The expanders can use so-called features, and in the tool, custom code craftings can be added to extend the base functionality. We also took advantage of this technique, and the OWL part is implemented using such craftings. That results in a clear separation of concerns and thus in evolvability of the solution.

The design of the transformation tool and its reference implementation utilising expanders can be used outside our framework and concerns related to conceptual modelling. It can plainly serve to provide RDF serialisation (or projection) of NS models for other purposes such as analysis, semantic integration, or data transfer. The tool with its design can also serve as an example for other similar projects, where a different type of NS model manipulation is desired.

8.3 OWL Ontologies for Conceptual Modelling

As part of this dissertation thesis, we developed several OWL ontologies based on conceptual modelling language specifications and metamodels. It was a reaction to the lack of interoperability in conceptual modelling. The practical side of this issue is emphasised also in our demonstration where most of the models are distributed as figures of diagrams, in a better case using a tool-specific format. Although we reviewed existing methods and related work, we identified only BPMN 2.0 Ontology [160] as usable with respect to our goals. In the end, we designed and created ontologies for well-known conceptual languages: UML (supporting Class Diagram, Activity Diagram, and State Machine Diagram) with profiles support, OntoUML, BPMN (Collaboration Diagram), BORM, and ORM. The detailed description is provided in Chapter 5. The ontologies are designed with respect to our requirements as well as current best practices. Moreover, all the ontologies are provided with documentation and examples.

The ontologies can be easily reused in other scenarios due to following the requirements related to FAIRness. The ontologies have open licenses and are prepared for further development and community contributions when needed. That said, the clear versions and transparent track of changes allow referring to versions related to this dissertation thesis. Also, with proper versioning, the ontologies may evolve independently on our gateway ontology and mappers. The mappers may be updated when necessary to another specific version of the ontology describing the language metamodel. This approach does not limit future use or further development in any way.

In the scope of the dissertation thesis, we designed the ontologies as a way to have well-defined conceptual models in RDF. We do not provide any universal tool that automates the transformation of conceptual models from various standard or non-standard (i.e. tool-specific or language-specific) formats to RDF. Nevertheless, for other use cases, it may be necessary to support such a transformation in addition to our contributions. Another aspect of why we did not tackle these transformations is that (in contrast to NS-RDF/OWL transformations), the work on translating formats with both specified syntax and semantics is not research-related but falls into straightforward software development. Still, with the number of formats in the current world of conceptual modelling, it can be long-term work. Therefore, we foresee implementing such a transformation on a "when-needed" basis.

8.4 SPARQL-Based Mappings in RDF for Versatile RDF Transformations

The key partial contribution of the dissertation thesis is the method for defining the mappings between different RDF datasets directly in RDF. It does so by providing an OWL ontology with constructs usable in SPARQL Protocol and RDF Query Language (SPARQL) `CONSTRUCT` query, e.g. use of a variable in input/output triple patterns. Moreover, by applying NS principles, the method allows modularity of the mappings that leads to improved evolvability and potential re-use of mapping modules. The execution of the transformation then involves translating the mappings in RDF according to the attached metadata to SPARQL `CONSTRUCT` queries that are then sequentially applied to transform RDF dataset.

Both the method and the created transformation tool can be used for any scenario where RDF dataset need to be enriched or transformed using defined mapping patterns using any OWL or Resource Description Framework Schema (RDFS) ontologies. We used the method for transformation between our ontologies for the conceptual model and NS metamodel as an ontology in OWL. Furthermore, we created the Gateway Ontology to simplify the specification of the mapping when relating different ontologies to the NS ontology. Our extensions and transformations layers provide additional constructs that can be reused and seen as shortcuts when using the SPARQL-based mappings together with NS-in-RDF/OWL.

The way we used the SPARQL-based mappings, the transformation tool, and created the Gateway Ontology, as well as the mappings for conceptual modelling languages, are themselves a contribution that can serve as an inspiration for related further research and work on the mappings. By having the mappings in RDF, metacircularity is enabled, and, potentially, a mapping that creates other mappings can be introduced.

8.5 CM-NS Transformations

The primary goal set by the research hypothesis (and subsequent objectives) is achieved successfully by completing all the partial artefacts that make up the framework of transformation between the conceptual models and NS. The requirements set according to the DSR methodology in Chapter 3 were fulfilled and the partial artefacts, as well as the overall solution, have been described and demonstrated.

Regarding the demonstration of the use of our transformation framework, since we did not create just a theoretical design, but materialised it for evaluation purposes, we explained and showed several use cases in Chapter 7 that clearly reveal the capabilities of our work. The reference implementation served for evaluation and continuous improvement according to the design cycle. It also proves the possibility of transformations between various conceptual models (eventually semantically integrated) to NS and vice versa.

Despite the readiness of the reference implementation to be directly used for such transformations, it is also prepared for future evolution as external dependencies are also expected to evolve. The evolvability lies in the architecture and application of NS principles as well as other contextual and theory-based requirements. The essence of evolvability lies in the independent evolvability of the partial artefacts.

8.6 RDF/OWL-Based MDD Framework

The overall generic design of ontology-based model-to-model transformation utilising RDF technologies and NS principles is itself a re-usable artefact. For different types of input and output models, one can follow our steps and the architecture presented with a gateway ontology as the central part. The framework is conceptually capable of supporting not only new conceptual models, but also other implementation ontologies that might follow NS. Therefore, it can be considered as a kind of bus-like hinge between conceptual models and implementation ontologies. In this way, the architecture can be applied for a different implementation-oriented language, e.g. mentioned Executable Translatable UML (xtUML) or Foundational UML (fUML) with Action Language for Foundational UML (ALF). To support that, a transformation with the corresponding tool would have to be created (e.g. xtUML-RDF/OWL tool). Then the gateway ontology layers would need to be adjusted to the new core layer. Still, our work on the side of conceptual models could be re-used.

On the other hand, the architecture can be altered not to relate with conceptual models in general but with another kind of input, for example, using formal specification methods or reference ontologies (in various ontology languages). This would require creating RDF/OWL representations of those, corresponding mappers for the gateway ontology, and adjustments in the transformations and extensions layers based on the supported input constructs. However, the ability to cope with changes in all planes would be retained. In the end, formal specifications or various ontologies could be transformed into NS.

With both of the cases described, the whole stack could be changed and support totally different inputs and outputs for transformations than we do, for instance, formal specifications in combination with xtUML – xtUML Gateway Ontology for Formal Specifications. Our design is based on specific requirements for conceptual modelling and NS; therefore, minor adjustments and extensions may be needed. Still, the flexible design enables such changes in partial artefacts (or modules).

Chapter 9

Conclusions

"Work. Finish. Publish."

— Michael Faraday

In this final chapter, we conclude the dissertation thesis by summarizing the results achieved in accordance with the research objectives established earlier. We provide a comprehensive overview of the findings together with their implications and recapitulate the main contributions made by this research. We emphasize the original insights gained and the potential impact of our work on the field. Finally, we outline potential prospects for future research.

9.1 Summary

In this dissertation thesis, we present the Normalized Systems Gateway Ontology for Conceptual Models as a framework for transformation between various conceptual modelling languages and Normalized Systems (NS) while addressing semantic integration and model consistency issues. Due to the following the Design Science Research (DSR) methodology, we designed the overall solution from partial artefacts that are fulfilling the set research objective and the environmental requirements. Following the requirements and the analysis performed, we based our work on Resource Description Framework (RDF) and Web Ontology Language (OWL). The framework consists of several transformation mechanisms developed as separate artefacts. First, we worked on transformations between NS and RDF/OWL where we incorporated the NS expanders. Then, we focused on the side of conceptual models and proposed their representation using RDF. With both conceptual models and NS models, we designed the gateway ontology with its layered architecture as the central part for transformations.

The core layer of the gateway ontology is formed by the NS metamodel generated using our NS-RDF/OWL transformation. The extensions and transformations layers provide additional constructs for mapping conceptual modelling languages to the NS metamodel. Finally, each modelling language has its own mapper module that describes how the constructs of the modelling language relate to the NS constructs. With all the semantics needed for the transformation captured, we were able to design the SPARQL-based execution of the transformation. The framework was also successfully demonstrated with models based on real-world e-commerce systems.

Concerning the research hypothesis, we confirmed by the reference implementation of the gateway ontology framework that it is possible to design transformations between conceptual and NS models. The evolvability and configurability is the essence of our solution, as it is based on the versatile and interoperable technologies of RDF and OWL. Furthermore, it utilises the NS principles on the design level together with the NS expanders on the implementation level. The modular architecture and design of the partial artefacts enable extensions, especially in form, adopting additional modelling languages in the future.

The first research objective (RO1) is achieved by basing the design on the technologies of the semantic web and linked data. As shown in Chapter 5, the knowledge in various models can be integrated in a way that is straightforward and verified by practice. It allows one to combine knowledge about a problem domain of different aspects and provide a comprehensive domain description that can be used for Model-Driven Development (MDD). The advantage is apparent – more information in the machine-readable format results in more complete generated software, and thus less custom code needed.

We split the second research objective (RO2) based on DSR into three requirements for our design – to support structural, behavioural, and fact conceptual modelling as the most significant types of modelling used in software engineering. Our design itself does not make any assumptions about the aspects of the model, and theoretically, any formalised modelling language can be used (as described in Chapter 6). We demonstrated the use of our framework for these three different types to show this feature in Chapter 7. The information loss is mitigated by storing additional constructs that are not covered directly in NS within so-called options (e.g. data options). That allows their retrieval for analysis and to keep models consistent.

As explained, the third research objective (RO3) related to the evolvability is addressed by the selected technologies and use of NS theory. The modularisation in the overall design (as presented in Chapter 3) follows the NS principles. The separation of concerns into layers and encapsulation within layers to modules limits the change impact. For example, changes in modelling language specification are encapsulated within its own mapper module. Similarly, a change in the NS metamodel is projected in the core layers of the gateway ontology using expanded NS-RDF/OWL transformation (described in Chapter 4). All parts can be versioned and evolved independently by simply ensuring the conformance of the interfaces between versions – just as in NS software applications.

Finally, the fourth research objective (RO4) on consistency is achieved by avoiding information loss and allowing bi-directional transformation. Both RDF and NS have constructs for keeping additional knowledge that cannot be captured directly using defined constructs from metamodels or ontologies. In RDF, we attach the additional data from the NS models using the constructs specified in the gateway ontology. On the other hand, additional data from conceptual models are stored using options of constructs NS models, as already mentioned. The gateway ontology presented in Chapter 6 enables bi-directional transformation as a mapping specification for a conceptual modelling language is reversible. With that and the elimination of information loss, the transformation can be executed in both directions. Thus, one can bring changes made in a conceptual model can be brought back to the NS model and vice versa.

All research objectives are successfully accomplished, and the resulting artefacts are presented. The designed framework is completed with its reference implementation that served for demonstration and evaluation according to DSR. However, it can be used directly to transform models in real-world scenarios and further evolve with newly emerging needs. The design itself, as well as the partial artefacts, can also be re-used for different use cases in the area of software engineering and MDD as our contribution to the current knowledge base.

9.2 Contributions of the Dissertation Thesis

The main contributions of the dissertation thesis are:

- the design of model-to-model transformation based on RDF/OWL,
- the evolvable transformation between NS elements and RDF/OWL,
- OWL ontologies for conceptual modelling using RDF,
- the SPARQL generation method for RDF transformations,
- the framework for transformations between NS and conceptual models.

Moreover, as discussed in Chapter 8, our architecture as well as its partial artefacts can be re-used to create transformations with different modelling languages and other MDD techniques than NS. These contributions are our extension of the knowledge base according to DSR and its rigour cycle.

9.3 Future Work

Whilst designing the transformation between NS and conceptual models, we identified several related research topics that are beyond our scope. However, it might be beneficial to explore them in the future as a result of subsequent research. Some of these topics are (re-)using our work (the new contributions to the knowledge base) for different or extended use cases, whereas others were identified as gaps in the existing knowledge base. We suggest exploring the following research topics:

- Extending modelling languages support – One of the by-design intended future steps is to support another conceptual modelling language needed for specific use cases. We support the most widely used structural, behavioural, and fact modelling representatives. However, for some non-traditional enterprise systems, it might also be needed to support other languages. For example, a legacy system described in Vienna Development Method (VDM) or Z-Notation, or a low-level system (e.g. for Internet of Things) modelled in SysML. The research work would need to investigate the metamodel of the specific language, specify OWL ontology, and create the corresponding mapping. The recommended way is to follow the same steps as we used the DSR methodology. The field testing with existing models and their analysis can be expected as an evaluation.

- Using conceptual models in RDF/OWL – The interoperability issues in conceptual modelling as one way to abstract and capture domain knowledge need to be addressed and the appropriate solution adopted by the broad community. As is clearly shown in this dissertation thesis, interoperable conceptual models open many new possibilities. RDF together with OWL and related technologies turned out to provide a “lingua franca” and a good environment for working with interoperable conceptual models. As we tackled SPARQL and SHACL, there are other possibilities to be explored to promote the interoperability of conceptual models, but also to enable new ways of using them. For example, SHACL and ShEx could be used to specify and validate constraints on models similarly to OCL or RML to obtain semantic RDF data from other formats of conceptual models such as XMI. The use of inference engines and mechanisms for RDF could also show its value for conceptual modelling. Finally, conceptual models are often tied to their graphical representation; being able to capture it as an appendix to the semantics of the language metamodel would help address this issue of keeping conceptual models in RDF. A crucial part of this research journey should also be a thorough comparison with other techniques, for instance, Meta-Object Facility (MOF)-based Query/View/Transformation (QVT).

- Incorporating UI modelling languages – The conceptual modelling of a problem domain is not the only modelling of software engineering. Promising work is being done in the field of User Interface (UI) modelling. It can be seen as a natural complement to domain conceptual models that can be used by an MDD technique to generate software. Nevertheless, such a system must use a generic UI. For example, NS have its forms, tables, waterfall views, or component-based menus. By supporting languages and methods of the so-called *Model-Driven UI Development* such as Interaction Flow Modeling Language (IFML) or UML-based Web Engineering (UWE), more information for generating the software system (now including its UI aspect) would be reachable. Our framework provides a good foundation for this effort. The future work could extend UI modelling in NS (through its metamodel and expanders), then update the core of the gateway ontology seamlessly, and finally provide a way for mapping language such as IFML by enhancing the extensions layer of the gateway ontology and creating a corresponding mapper. As an evaluation, the improvement in limiting UI-related custom code craftings could be measured on existing NS applications.

- Comparing methods for systems generation – The ability of transformation between conceptual models and NS can be used as a part of a "testbed" to compare different MDD methods. For example, we proposed a solution for expanding NS directly from textual requirements using the Textual Modelling System (TEMOS) tool [A.14]. The tool also supports the creation of UML models from the text while dealing with issues of ambiguity, incompleteness, and inconsistency [172]. It could be interesting to observe the results for text-to-NS and text-to-UML-to-NS. A similar approach could be applied even to other tools and methods. With adding support to generate another type of implementation-related models than NS it could even be possible to compare the ability of such MDD methods to adopt constructs from different conceptual modelling languages.

- Generating craftings from conceptual models – The scope of our work was focused on the transformation between conceptual models and NS models; however, a helpful addition would be a support of transforming not just to NS elements but also code fragments. The additional information from conceptual models is currently encoded using options constructs, e.g., data options or field options. Future work could use this "dangling" information to generate code fragments and limit the number of manually-maintained craftings. Several approaches might be explored to achieve this goal, e.g., generating additional harvest files or designing the so-called expander features. A profound impact analysis would be necessary to evaluate if maintaining such mappings and generation is not causing higher overhead than maintenance of the manually-coded craftings that the technique can generate.

- Refining models and mappings with AI methods – The techniques and methods of artificial intelligence (AI) as well as natural language processing (NLP) are often used with RDF datasets. With our solution, the conceptual models, the gateway ontology, mappers for modelling languages, and NS are in RDF. Therefore, such methods could be used to analyse the models and find potential refinements. For instance, similar patterns might be found across different language mappers to propose some new abstractions to be incorporated into the extensions layer of the gateway ontology. With such an approach, mapper maintenance could become more efficient. Such further research work could also result in the proposal of such bottom-up emerged extensions to become a part of the NS metamodel.

- Analysing existing systems and their evolution – Our work (as a side effect) allows us to transform existing NS models into RDF. This feature can be used to analyse existing systems, find similarities, and uncover essential facts that would be difficult to discover at the NS level. Such analysis can utilise the entire arsenal of RDF-compatible techniques including inference, NLP or AI methods. There are existing works on the NS level, e.g. counting and comparing the custom code craftings across different NS applications. A potential first step could be to rework them with the use of RDF, enhance them, and compare them with the previous solution.

Bibliography

[1] Mannaert, H.; Verelst, J.; et al. *Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design.* Kermt, Belgium: Koppa, 2016, ISBN 978-90-77160-091.

[2] Krouwel, M. *Towards the Agile Enterprise: A Method to Come from a DEMO Model to a Normalized System, Applied to Government Subsidy Schemes.* Master's thesis, TU Delft, 2010. Available from: `http://resolver.tudelft.nl/uuid:a170e23f-9fee-45fd-b99b-3a85e4d551cf`

[3] Krouwel, M. R.; Land, M. O. Combining DEMO and Normalized Systems for Developing Agile Enterprise Information Systems. In *Advances in Enterprise Engineering V – First Enterprise Engineering Working Conference, EEWC 2011, Antwerp, Belgium, May 16-17, 2011. Proceedings*, *Lecture Notes in Business Information Processing*, volume 79, edited by A. Albani; J. L. G. Dietz; J. Verelst, Springer, 2011, pp. 31–45, doi:10.1007/978-3-642-21058-7_3.

[4] Vo, H. L. M.; Hoang, Q. Transformation of UML Class Diagram into OWL Ontology. *Journal of Information and Telecommunication*, volume 4, no. 1, 2019: pp. 1–16, doi:10.1080/24751839.2019.1686681.

[5] Sadowska, M.; Huzar, Z. Representation of UML Class Diagrams in OWL 2 on the Background of Domain Ontologies. *Software Engineering Journal*, volume 13, no. 1, 2019: pp. 63–103, doi:10.5277/e-Inf190103.

[6] Zedlitz, J.; Jörke, J.; et al. From UML to OWL 2. In *Knowledge Technology – Third Knowledge Technology Week, KTW 2011, Kajang, Malaysia, July 18-22, 2011. Revised Selected Papers*, *Communications in Computer and Information Science*, volume 295, edited by D. Lukose; A. R. Ahmad; A. Suliman, Springer, 2011, pp. 154–163, doi:10.1007/978-3-642-32826-8_16.

[7] Belghiat, A.; Bourahla, M. From UML Class Diagrams to OWL Ontologies: A Graph Transformation Based Approach. In *Proceedings of the 4th International conference on Web and Information Technologies, ICWIT 2012, Sidi Bel Abbes, Algeria, April 29-30, 2012*, *CEUR Workshop Proceedings*, volume 867, edited by M. Malki; S. Benbernou; S. M. Benslimane; A. Lehireche, CEUR-WS.org, 2012, pp. 330–335. Available from: `http://ceur-ws.org/Vol-867/Paper38.pdf`

[8] Gasevic, D.; Djuric, D.; et al. Converting UML to OWL Ontologies. In *Proceedings of the 13th international conference on World Wide Web – Alternate Track Papers & Posters, WWW 2004, New York, NY, USA, May 17-20, 2004*, edited by S. I. Feldman; M. Uretsky; M. Najork; C. E. Wills, ACM, 2004, pp. 488–489, doi:10.1145/1013367.1013539.

[9] Memon, M. A.; Hassan, Z.; et al. Aspect Oriented UML to ECORE Model Transformation. *ISC International Journal of Information Security*, volume 11, no. 3, 2019: pp. 97–103, doi:10.22042/isecure.2019.11.0.13.

[10] Cunha, A.; Garis, A. G.; et al. Translating between Alloy Specifications and UML Class Diagrams Annotated with OCL. *Software and Systems Modeling*, volume 14, no. 1, 2015: pp. 5–25, doi:10.1007/s10270-013-0353-5.

[11] Shah, S. M. A.; Anastasakis, K.; et al. From UML to Alloy and Back Again. In *Models in Software Engineering, Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers*, *Lecture Notes in Computer Science*, volume 6002, edited by S. Ghosh, Springer, 2009, pp. 158–171, doi:10.1007/978-3-642-12261-3_16.

[12] Anastasakis, K.; Bordbar, B.; et al. On Challenges of Model Transformation from UML to Alloy. *Software and Systems Modeling*, volume 9, no. 1, 2010: pp. 69–86, doi:10.1007/s10270-008-0110-3.

[13] Khlif, W.; Ayed, N. E. B.; et al. From a BPMN Model to an Aligned UML Analysis Model. In *Proceedings of the 13th International Conference on Software Technologies, ICSOFT 2018, Porto, Portugal, July 26-28, 2018*, edited by L. A. Maciaszek; M. van Sinderen, SciTePress, 2018, pp. 657–665, doi:10.5220/0006866606570665.

[14] Cibrán, M. A. Translating BPMN Models into UML Activities. In *Business Process Management Workshops, BPM 2008 International Workshops, Milano, Italy, September 1-4, 2008. Revised Papers*, *Lecture Notes in Business Information Processing*, volume 17, edited by D. Ardagna; M. Mecella; J. Yang, Springer, 2008, pp. 236–247, doi:10.1007/978-3-642-00328-8_23.

[15] Braga, B. F. B.; Almeida, J. P. A.; et al. Transforming OntoUML into Alloy: Towards Conceptual Model Validation using a Lightweight Formal Method. *Innovations in Systems and Software Engineering*, volume 6, no. 1-2, 2010: pp. 55–63, doi:10.1007/s11334-009-0120-5.

[16] Barcelos, P. P. F.; dos Santos, V. A.; et al. An Automated Transformation from OntoUML to OWL and SWRL. In *Proceedings of the 6th Seminar on Ontology Research in Brazil, Belo Horizonte, Brazil, September 23, 2013*, *CEUR Workshop Proceedings*, volume 1041, edited by M. P. Bax; M. B. Almeida; R. Wassermann, CEUR-WS.org, 2013, pp. 130–141. Available from: `http://ceur-ws.org/Vol-1041/ontobras-2013_paper44.pdf`

[17] Halpin, T. A. Information Analysis in UML and ORM: A Comparison. In *Advanced Topics in Database Research*, volume 1, edited by K. Siau, IGI Global, 2002, pp. 307–323, doi:10.4018/978-1-930708-41-9.ch016.

[18] Halpin, T. A. Comparing Metamodels for ER, ORM and UML Data Models. In *Advanced Topics in Database Research*, volume 3, edited by K. Siau, IGI Global, 2004, pp. 23–44, doi:10.4018/978-1-59140-255-8.ch002.

[19] Halpin, T. A. Information Modeling in UML and ORM. In *Encyclopedia of Information Science and Technology*, edited by M. Khosrow-Pour, IGI Global, first edition, 2005, pp. 1471–1475, doi:10.4018/978-1-59140-553-5.ch258.

[20] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Technical report, Object Management Group (OMG), June 2016, [Accessed 24 May 2022]. Available from: `https://www.omg.org/spec/QVT/1.3/PDF`

[21] Eclipse Foundation, Inc. ATL Transformation Language. [online], 2022, [Accessed 22 March 2022]. Available from: `https://www.eclipse.org/atl/`

[22] Amelunxen, C.; Königs, A.; et al. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *Model Driven Architecture – Foundations and Applications, 2nd European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings*, *Lecture Notes in Computer Science*, volume 4066, edited by A. Rensink; J. Warmer, Springer, 2006, pp. 361–375.

[23] Object Management Group. MOF Model to Text Transformation Language, v1.0. Technical report, Object Management Group (OMG), January 2008, [Accessed 9 June 2022]. Available from: `https://www.omg.org/spec/MOFM2T/1.0/PDF`

[24] Williams, I. *Beginning XSLT and XPath: Transforming XML Documents and Data*. Wrox beginning guides, Wiley, 2009, ISBN 978-0-470-56746-3.

[25] Soley, R.; et al. Model Driven Architecture. *OMG White Paper*, 2000, [Accessed 3 March 2022]. Available from: `https://www.omg.org/~soley/mda.html`

[26] Liddle, S. W. Model-Driven Software Development. In *Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges*, Berlin: Springer, 2011, ISBN 978-3-642-15865-0, pp. 17–56, doi:10.1007/978-3-642-15865-0_2.

[27] Rybola, Z. *Towards OntoUML for Software Engineering: Transformation of OntoUML into Relational Databases*. Ph.D. thesis, Czech Technical University in Prague, Prague, Czech Republic, August 2017, [Accessed 1 April 2019]. Available from: `https://www.fit.cvut.cz/sites/default/files/PhDThesis-Rybola.pdf`

[28] Pastor, O.; Insfrán, E.; et al. OO-METHOD: An OO Software Production Environment Combining Conventional and Formal Methods. In *Advanced Information Systems Engineering, 9th International Conference CAiSE'97, Barcelona, Catalonia, Spain, June 16-20, 1997, Proceedings*, *Lecture Notes in Computer Science*, volume 1250, edited by A. Olivé; J. A. Pastor, Springer, 1997, pp. 145–158, doi: 10.1007/3-540-63107-0_11.

[29] Hartl, M. *Ruby on Rails Tutorial: Learn Web Development with Rails*. Addison-Wesley Professional, third edition, 2015, ISBN 978-0-13-407770-3.

[30] NSX. Prime Radiant Online. [online], 2022, [Accessed 14 November 2022]. Available from: `http://primeradiant.stars-end.net/foundation/`

[31] Shvaiko, P.; Euzenat, J. Ontology Matching: State of the Art and Future Challenges. *IEEE Transactions on Knowledge and Data Engineering*, volume 25, no. 1, 2013: pp. 158–176, doi:10.1109/TKDE.2011.253.

[32] Choi, N.; Song, I.; et al. A Survey on Ontology Mapping. *SIGMOD Record*, volume 35, no. 3, 2006: pp. 34–41, doi:10.1145/1168092.1168097.

[33] Dimou, A.; Sande, M. V.; et al. RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In *Proceedings of the Workshop on Linked Data on the Web co-located with the 23rd International World Wide Web Conference (WWW 2014), Seoul, Korea, April 8, 2014*, *CEUR Workshop Proceedings*, volume 1184, edited by C. Bizer; T. Heath; S. Auer; T. Berners-Lee, CEUR-WS.org, 2014, p. 1. Available from: `http://ceur-ws.org/Vol-1184/ldow2014_paper_01.pdf`

[34] Heyvaert, P.; Dimou, A.; et al. Towards Approaches for Generating RDF Mapping Definitions. In *Proceedings of the ISWC 2015 Posters & Demonstrations Track co-located with the 14th International Semantic Web Conference (ISWC-2015), Bethlehem, PA, USA, October 11, 2015*, *CEUR Workshop Proceedings*, volume 1486, edited by S. Villata; J. Z. Pan; M. Dragoni, CEUR-WS.org, 2015, p. 70. Available from: `http://ceur-ws.org/Vol-1486/paper_70.pdf`

[35] Corby, O.; Faron-Zucker, C. STTL – a SPARQL-based Transformation Language for RDF. In *WEBIST 2015 – Proceedings of the 11th International Conference on Web Information Systems and Technologies, Lisbon, Portugal, 20-22 May, 2015*, edited by V. Monfort; K. Krempels; T. A. Majchrzak; Z. Turk, SciTePress, 2015, pp. 466–476, doi:10.5220/0005450604660476.

[36] Decker, S.; Sintek, M.; et al. TRIPLE – an RDF Rule Language with Context and Use Cases. In *W3C Workshop on Rule Languages for Interoperability, 27-28 April 2005, Washington, DC, USA*, W3C, 2005, p. 98. Available from: `http://www.w3.org/2004/12/rules-ws/paper/98`

[37] Hevner, A. R.; March, S. T.; et al. Design Science in Information Systems Research. *MIS Q.*, volume 28, no. 1, 2004: pp. 75–105. Available from: `http://misq.org/design-science-in-information-systems-research.html`

[38] Hevner, A. R. Design Science Research. In *Computing Handbook: Information Systems and Information Technology*, volume 22, edited by H. Topi; A. Tucker, CRC Press, third edition, 2014, pp. 1–23.

[39] Kuechler Jr., W. L.; Vaishnavi, V. K. On Theory Development in Design Science Research: Anatomy of a Research Project. *European Journal of Information Systems*, volume 17, no. 5, 2008: pp. 489–504, doi:10.1057/ejis.2008.40.

[40] Peffers, K.; Tuunanen, T.; et al. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, volume 24, no. 3, 2007: pp. 45–78, doi:10.2753/MIS0742-1222240302. Available from: `http://www.jmis-web.org/articles/765`

[41] Braun, R.; Benedict, M.; et al. Proposal for Requirements Driven Design Science Research. In *New Horizons in Design Science: Broadening the Research Agenda – 10th International Conference, DESRIST 2015, Dublin, Ireland, May 20-22, 2015, Proceedings*, *Lecture Notes in Computer Science*, volume 9073, edited by B. Donnellan; M. Helfert; J. Kenneally; D. E. VanderMeer; M. A. Rothenberger; R. Winter, Springer, 2015, pp. 135–151, doi:10.1007/978-3-319-18714-3_9.

[42] Hevner, A. R. The Three Cycle View of Design Science. *Scandinavian Journal of Information Systems*, volume 19, no. 2, 2007. Available from: `http://aisel.aisnet.org/sjis/vol19/iss2/4`

[43] Mylopoulos, J. Conceptual Modelling and Telos. *Conceptual Modelling Databases and CASE: An Integrated View of Information Systems Development*, 1992: pp. 49–68, University of Toronto. Available from: `http://www.cs.toronto.edu/~jm/2507S/Readings/CM+Telos.pdf`

[44] Lindland, O. I.; Sindre, G.; et al. Understanding Quality in Conceptual Modeling. *IEEE Software*, volume 11, no. 2, 1994: pp. 42–49, doi:10.1109/52.268955.

[45] Moody, D. L. Theoretical and Practical Issues in Evaluating the Quality of Conceptual Models: Current State and Future Directions. *Data Knowledge Engineering*, volume 55, no. 3, Dec. 2005: pp. 243–276, ISSN 0169-023X, doi:10.1016/j.datak.2004.12.005.

[46] Miao, H.; Sun, J.; et al. A Literature Review on Conceptual Model Quality of Iinformation Systems. *International Journal of Digital Content Technology and its Applications*, volume 7, no. 5, 2013: p. 574.

[47] Chen, P. P.-S. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems (TODS)*, volume 1, no. 1, Mar. 1976: pp. 9–36, ISSN 0362-5915, doi:10.1145/320434.320440.

[48] Elmasri, R.; Navathe, S. B. *Fundamentals of Database Systems*. Pearson, 7th edition, 2015, ISBN 978-0-13-397077-7.

[49] Mylopoulos, J. Information Modeling in the Time of the Revolution. *Information Systems*, volume 23, 1998: pp. 127–155, ISSN 0306-4379.

[50] Villari, M.; Celesti, A.; et al. Enriched ER Model to Design Hybrid Database for Big Data Solutions. In *Computers and Communication (ISCC), 2016 IEEE Symposium on*, IEEE, 2016, pp. 163–166.

[51] Object Management Group. Information Technology – Object Management Group Unified Modeling Language (OMG UML), Infrastructure, v. 2.4.1. Technical report, Object Management Group (OMG), April 2012, [Accessed 2 March 2022]. Available from: `http://www.omg.org/spec/UML/ISO/19505-1/PDF`

[52] Object Management Group. OMG Unified Modeling Language, v. 2.5. Technical report, Object Management Group (OMG), March 2015, [Accessed 10 March 2022]. Available from: `http://www.omg.org/spec/UML/2.5/PDF`

[53] Object Management Group. OMG Meta Object Facility (MOF) Core Specification. Technical report, Object Management Group (OMG), October 2019, [Accessed 5 March 2022]. Available from: `https://www.omg.org/spec/MOF/2.5.1/PDF`

[54] Steinberg, D.; Budinsky, F.; et al. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008, ISBN 978-0-32-133188-5.

[55] Kurtev, I. State of the Art of QVT: A Model Transformation Language Standard. In *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers*, *Lecture Notes in Computer Science*, volume 5088, edited by A. Schürr; M. Nagl; A. Zündorf, Springer, 2007, pp. 377–393, doi: 10.1007/978-3-540-89020-1_26.

[56] Guizzardi, G. *Ontological Foundations for Structural Conceptual Models*. Enschede (The Netherlands): Centre for Telematics and Information Technology, Telematica Instituut, University of Twente, 2005, ISBN 978-9-07517-681-0. Available from: `http://doc.utwente.nl/50826/1/thesis_Guizzardi.pdf`

[57] Braga, B. F.; Almeida, J. P. A.; et al. Transforming OntoUML into Alloy: Towards Conceptual Model Validation Using a Lightweight Formal Method. *Innovations in Systems and Software Engineering*, volume 6, no. 1-2, 2010: pp. 55–63, doi:10.1007/s11334-009-0120-5.

[58] Guizzardi, G.; Fonseca, C. M.; et al. Endurant Types in Ontology-Driven Conceptual Modeling: Towards OntoUML 2.0. In *Conceptual Modeling – 37th International Conference, ER 2018, Xi'an, China, October 22-25, 2018, Proceedings*, *Lecture Notes in Computer Science*, volume 11157, edited by J. Trujillo; K. C. Davis; X. Du; Z. Li; T. W. Ling; G. Li; M. Lee, Springer, 2018, pp. 136–150, doi: 10.1007/978-3-030-00847-5_12.

[59] Verdonck, M.; Gailly, F.; et al. Comparing Traditional Conceptual Modeling with Ontology-Driven Conceptual Modeling: An Empirical Study. *Information Systems*, 2018, ISSN 0306-4379, doi:10.1016/j.is.2018.11.009.

[60] Sales, T. P.; Guarino, N.; et al. An Ontological Analysis of Value Propositions. In *21st IEEE International Enterprise Distributed Object Computing Conference, EDOC 2017, Quebec City, QC, Canada, October 10-13, 2017*, edited by S. Hallé; R. Villemaire; R. Lagerström, IEEE Computer Society, 2017, pp. 184–193, doi: 10.1109/EDOC.2017.32.

[61] Ferreira, M. I. G. B.; Moreira, J. L. R.; et al. OntoEmergePlan: Variability of Emergency Plans Supported by a Domain Ontology. In *12th Proceedings of the International Conference on Information Systems for Crisis Response and Management, Krystiansand, Norway, May 24-27, 2015*, edited by L. Palen; M. Büscher; T. Comes; A. L. Hughes, ISCRAM Association, 2015. Available from: `http://idl.iscram.org/files/mariaigbferreira/2015/1184_MariaI.G.B.Ferreira_etal2015.pdf`

[62] Dietz, J. L. Towards a Discipline of Organisation Engineering. *European Journal of Operational Research*, volume 128, no. 2, 2001: pp. 351–363, doi:10.1016/S0377-2217(00)00077-1.

[63] Krouwel, M. R.; Land, M. O. Combining DEMO and Normalized Systems for Developing Agile Enterprise Information Systems. In *Advances in Enterprise Engineering V – First Enterprise Engineering Working Conference, EEWC 2011, Antwerp, Belgium, May 16-17, 2011. Proceedings*, *Lecture Notes in Business Information Processing*, volume 79, edited by A. Albani; J. L. G. Dietz; J. Verelst, Springer, 2011, pp. 31–45, doi:10.1007/978-3-642-21058-7_3.

[64] Mráz, O.; Náplava, P.; et al. Converting DEMO PSI Transaction Pattern into BPMN: A Complete Method. In *Advances in Enterprise Engineering XI – 7th Enterprise Engineering Working Conference, EEWC 2017, Antwerp, Belgium, May 8-12, 2017, Proceedings*, *Lecture Notes in Business Information Processing*, volume 284, edited by D. Aveiro; R. Pergl; G. Guizzardi; J. P. A. Almeida; R. Magalhães; H. Lekkerkerk, Springer, 2017, pp. 85–98, doi:10.1007/978-3-319-57955-9_7.

[65] Halpin, T. *Object-Role Modeling Fundamentals: A Practical Guide to Data Modeling with ORM.* Basking Ridge, NJ, USA: Technics Publications, LLC, 2015, ISBN 978-1-63462-074-1.

[66] Halpin, T. ORM 2. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, Springer, 2005, pp. 676–687, doi:10.1007/11575863_87.

[67] Jarrar, M. Towards Automated Reasoning on ORM Schemes. In *Conceptual Modeling – ER 2007, 26th International Conference on Conceptual Modeling, Auckland, New Zealand, November 5-9, 2007, Proceedings*, *Lecture Notes in Computer Science*, volume 4801, edited by C. Parent; K. Schewe; V. C. Storey; B. Thalheim, Springer, 2007, pp. 181–197, doi:10.1007/978-3-540-75563-0_14.

[68] Völzer, H. An Overview of BPMN 2.0 and Its Potential Use. In *Business Process Modeling Notation – Second International Workshop, BPMN 2010, Potsdam, Germany, October 13-14, 2010. Proceedings*, *Lecture Notes in Business Information Processing*, volume 67, edited by J. Mendling; M. Weidlich; M. Weske, Springer, 2010, pp. 14–15, doi:10.1007/978-3-642-16298-5_3.

[69] Allweyer, T. *BPMN 2.0: Introduction to the Standard for Business Process Modeling.* Books on Demand, 2016, ISBN 978-3-83-709331-5.

[70] Guizzardi, G.; Wagner, G. Conceptual Simulation Modeling with OntoUML. In *Proceedings of the Winter Simulation Conference*, WSC '12, Winter Simulation Conference, 2012, pp. 5:1–5:15.

[71] Knott, R. P.; Merunka, V.; et al. The BORM Methodology: A Third-Generation Fully Object-Oriented Methodology. *Knowledge-Based Systems*, volume 16, no. 2, 2003: pp. 77–89, doi:10.1016/S0950-7051(02)00075-8.

[72] Podloucký, M.; Pergl, R.; et al. Revisiting the BORM OR Diagram Composition Pattern. In *Enterprise and Organizational Modeling and Simulation*, *Lecture Notes in Business Information Processing*, volume 231, Stockholm: Springer, 2015, pp. 102–113, doi:10.1007/2F978-3-319-24626-0_8.

[73] van Lamsweerde, A. Formal Specification: A Roadmap. In *22nd International Conference on on Software Engineering, Future of Software Engineering Track, ICSE 2000, Limerick Ireland, June 4-11, 2000*, edited by A. Finkelstein, ACM, 2000, pp. 147–159, doi:10.1145/336512.336546.

[74] Object Management Group. Object Constraint Language, v. 2.4. Technical report, Object Management Group (OMG), February 2014, [Accessed 23 March 2022]. Available from: `http://www.omg.org/spec/OCL/2.4/PDF`

[75] Gogolla, M. UML and OCL in Conceptual Modeling. In *Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges*, Berlin: Springer, 2011, ISBN 978-3-642-15865-0, pp. 85–122, doi:10.1007/978-3-642-15865-0_4.

[76] Gogolla, M.; Doan, K. Quality Improvement of Conceptual UML and OCL Schemata Through Model Validation and Verification. In *Conceptual Modeling Perspectives*, edited by J. Cabot; C. Gómez; O. Pastor; M. Sancho; E. Teniente, Springer, 2017, pp. 155–168, doi:10.1007/978-3-319-67271-7_11.

[77] Jackson, D. *Software Abstractions: Logic, Language, and Analysis.* Cambridge, MA, USA: MIT Press, 2006, ISBN 978-0-262-10114-1.

[78] Cunha, A.; Garis, A. G.; et al. Translating Between Alloy Specifications and UML Class Diagrams Annotated with OCL. *Software & Systems Modeling*, volume 14, no. 1, 2015: pp. 5–25, doi:10.1007/s10270-013-0353-5.

[79] Pastor, O.; Hayes, F.; et al. OASIS: An Object-Oriented Specification Language. In *Advanced Information Systems Engineering, CAiSE'92, Manchester, UK, May 12-15, 1992, Proceedings*, *Lecture Notes in Computer Science*, volume 593, edited by P. Loucopoulos, Springer, 1992, pp. 348–363, doi:10.1007/BFb0035141.

[80] Pastor, O.; Ramos, I. OASIS 2.1: A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach. *Servicio de Publicaciones Univ. Politécnica de Valencia,*, 1995.

[81] Letelier Torres, P.; Sánchez Palma, P.; et al. OASIS Versión 3.0: Un Enfoque Formal Para el Modelado Conceptual Orientado a Objeto. Technical report, Universidad Politécnica de Valencia. Servicio de Publicaciones, 1998, [Accessed 5 March 2022]. Available from: `https://repositorio.upct.es/bitstream/handle/10317/733/oef.pdf`

[82] Kourie, D. G.; Watson, B. W. *The Correctness-by-Construction Approach to Programming.* Springer, 2012, ISBN 978-3-642-27918-8, doi:10.1007/978-3-642-27919-5.

[83] Khalek, S. A.; Yang, G.; et al. TestEra: A Tool for Testing Java Programs using Alloy Specifications. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, edited by P. Alexander; C. S. Pasareanu; J. G. Hosking, IEEE Computer Society, 2011, ISBN 978-1-4577-1638-6, pp. 608–611, doi:10.1109/ASE.2011.6100137. Available from: `http://mir.cs.illinois.edu/marinov/publications/KhalekETAL11TestEraDemo.pdf`

[84] Ehrig, H.; Mahr, B. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, *EATCS Monographs on Theoretical Computer Science*, volume 6. Springer, 1985, ISBN 3-540-13718-1, doi:10.1007/978-3-642-69962-7.

[85] Wirsing, M. Algebraic Specification Languages: An Overview. In *Recent Trends in Data Type Specification, 10th Workshop on Specification of Abstract Data Types Joint with the 5th COMPASS Workshop, S. Margherita, Italy, May 30 – June 3, 1994, Selected Papers*, *Lecture Notes in Computer Science*, volume 906, edited by E. Astesiano; G. Reggio; A. Tarlecki, Springer, 1994, pp. 81–115, doi:10.1007/BFb0014423.

[86] James, P.; Roggenbach, M. *Recent Trends in Algebraic Development Techniques.* Springer, 2017, ISBN 978-3-31-972043-2.

[87] Haase, P.; Motik, B. A Mapping System for the Integration of OWL-DL Ontologies. In *Proceedings of the first international ACM workshop on Interoperability of Heterogeneous Information Systems (IHIS'05), CIKM Conference, Bremen, Germany, November 4, 2005*, edited by A. Hahn; S. Abels; L. Haak, ACM, 2005, pp. 9–16, doi:10.1145/1096967.1096970.

[88] McGuinness, D. L.; Van Harmelen, F.; et al. OWL Web Ontology Language Overview. *W3C Recommendation*, volume 10, no. 10, 2004, [Accessed 5 April 2022]. Available from: `https://www.w3.org/TR/owl-features/`

[89] Coyle, K.; Baker, T.; et al. Guidelines for Dublin Core Application Profiles. [online], 2009, [Accessed 14 July 2022]. Available from: `http://dublincore.org/documents/profile-guidelines/`

[90] Gailly, F.; Poels, G. Ontology-Driven Business Modelling: Improving the Conceptual Representation of the REA Ontology. In *Conceptual Modeling – ER 2007, 26th International Conference on Conceptual Modeling, Auckland, New Zealand, November 5-9, 2007, Proceedings*, *Lecture Notes in Computer Science*, volume 4801, edited by C. Parent; K. Schewe; V. C. Storey; B. Thalheim, Springer, 2007, pp. 407–422, doi:10.1007/978-3-540-75563-0_28.

[91] Baader, F.; Horrocks, I.; et al. *An Introduction to Description Logic.* Cambridge University Press, 2017, ISBN 978-0-521-69542-8.

[92] Rudolph, K. *The Use of Ontologies in Practice.* GRIN Verlag, 2015, ISBN 978-3-65-697610-3.

[93] Powers, S. *Practical RDF.* O'Reilly Media, 2003, ISBN 978-0-59-651561-4.

[94] OWL Working Group. OWL 2 Web Ontology Language Document Overview. [online], 2012, second edition, [Accessed 9 April 2022]. Available from: `https://www.w3.org/TR/owl2-overview/`

[95] Musen, M. A. The Protégé Project: A Look Back and a Look Forward. *AI Matters*, volume 1, no. 4, 2015: pp. 4–12, doi:10.1145/2757001.2757003.

[96] Lohmann, S.; Link, V.; et al. WebVOWL: Web-based Visualization of Ontologies. In *Knowledge Engineering and Knowledge Management – EKAW 2014 Satellite Events, VISUAL, EKM1, and ARCOE-Logic, Linköping, Sweden, November 24-28, 2014. Revised Selected Papers*, *Lecture Notes in Computer Science*, volume 8982, edited by P. Lambrix; E. Hyvönen; E. Blomqvist; V. Presutti; G. Qi; U. Sattler; Y. Ding; C. Ghidini, Springer, 2014, pp. 154–158, doi:10.1007/978-3-319-17966-7_21.

[97] Garijo, D. WIDOCO: A Wizard for Documenting Ontologies. In *International Semantic Web Conference*, Springer, Cham, 2017, pp. 94–102, doi:10.1007/978-3-319-68204-4_9. Available from: `http://dgarijo.com/papers/widoco-iswc2017.pdf`

[98] Robinson, I.; Webber, J.; et al. *Graph Databases: New Opportunities for Connected Data*. O'Reilly Media, 2015, ISBN 978-1-49-193084-7.

[99] Ehrig, M. *Ontology Alignment: Bridging the Semantic Gap*. Semantic Web and Beyond, Springer, 2006, ISBN 978-0-38-736501-5.

[100] Euzenat, J.; Shvaiko, P. *Ontology Matching*. Springer Berlin Heidelberg, 2013, ISBN 978-3-64-238721-0.

[101] Brambilla, M.; Cabot, J.; et al. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers, second edition, 2017, ISBN 978-1-62-705695-3.

[102] Gaševic, D.; Selic, B.; et al. *Model Driven Engineering and Ontology Development*. Springer, 2009, ISBN 978-3-64-200282-3.

[103] Hilliard, R. Views and Viewpoints in Software Systems Architecture. In *First Working IFIP Conference on Software Architecture, San Antonio*, 1999.

[104] Baldonado, M. Q. W.; Woodruff, A.; et al. Guidelines for Using Multiple Views in Information Visualization. In *Proceedings of the working conference on Advanced visual interfaces, AVI 2000, Palermo, Italy, May 23-26, 2000*, edited by V. D. Gesù; S. Levialdi; L. Tarantino, ACM Press, 2000, pp. 110–119, doi:10.1145/345513.345271.

[105] Dumay, M. Demo or Practice: Critical Analysis of the Language/Action Perspective. *CoRR*, 2004, doi:10.48550/arXiv.cs/0410006, [Accessed: 8 May 2022]. Available from: `http://arxiv.org/abs/cs.OH/0410006`

[106] Kruchten, P. Architectural Blueprints: The 4+1 View Model of Software Architecture. *IEEE Software, Vol. 12, Number 6*, 1995: pp. 42–50, ISSN 1937-4194, doi: 10.1109/52.469759.

[107] Wymore, A. W. *Model-Based Systems Engineering*. Boca Raton, FL, USA: CRC Press, 1993, ISBN 978-0-8493-8012-9.

[108] Truyen, F. The Fast Guide to Architecture – The Basics of Model Driven Architecture. [online], 2006, [Accessed 18 April 2022]. Available from: `http://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf`

[109] Silva, A. R. D. Model-Driven Engineering: A Survey Supported by the Unified Conceptual Model. *Computer Languages, Systems & Structures*, volume 43, 2015: pp. 139–155, doi:10.1016/j.cl.2015.06.001.

[110] Korshunova, E.; Petkovic, M.; et al. CPP2XMI: Reverse Engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code. In *13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy*, IEEE Computer Society, 2006, pp. 297–298, doi:10.1109/WCRE.2006.21.

[111] Springsteel, F.; Kou, C. Reverse Data Engineering of ER-Designed Relational Schemas. In *PARBASE-90: International Conference on Databases, Parallel Architectures, and Their Applications*, IEEE, 1990, pp. 438–440.

[112] Sparx Systems. Enterprise Architect 15.2 User Guide. [online], 2021, [Accessed 19 February 2022]. Available from: `https://sparxsystems.com/enterprise_architect_user_guide/15.2/index/index.html`

[113] Braun, P.; Marschall, F. BOTL – The Bidirectional Object Oriented Transformation Language. Technical report, TU München, 2003, [Accessed 8 August 2022]. Available from: `https://wwwbroy.in.tum.de/publ/papers/TUM-I0307.pdf`

[114] Noureen, A.; Amjad, A.; et al. Model Driven Architecture – Issues, Challenges and Future Directions. *Journal of Software*, volume 11, no. 9, 2016: pp. 924–933, doi: 10.17706/jsw.11.9.924-933.

[115] Starrett, C. xtUML: Current and Next State of a Modeling Dialect. In *Proceedings of the 2nd International Workshop on Executable Modeling co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), Saint-Malo, France, October 3, 2016*, *CEUR Workshop Proceedings*, volume 1760, edited by T. Mayerhofer; P. Langer; E. Seidewitz; J. Gray, CEUR-WS.org, 2016, pp. 33–37. Available from: `http://ceur-ws.org/Vol-1760/paper5.pdf`

[116] Guermazi, S.; Tatibouet, J.; et al. Executable Modeling with fUML and Alf in Papyrus: Tooling and Experiments. In *Proceedings of the 1st International Workshop on Executable Modeling co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015), Ottawa, Canada, September 27, 2015*, *CEUR Workshop Proceedings*, volume 1560, edited by T. Mayerhofer; P. Langer; E. Seidewitz; J. Gray, CEUR-WS.org, 2015, pp. 3–8. Available from: `http://ceur-ws.org/Vol-1560/paper1.pdf`

[117] Verdier, F.; Seriai, A.; et al. Reusing Platform-Specific Models in Model-Driven Architecture for Software Product Lines. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira – Portugal, January 22-24, 2018*, edited by S. Hammoudi; L. F. Pires; B. Selic, SciTePress, 2018, pp. 106–116, doi:10.5220/0006582601060116.

[118] Embley, D. W.; Liddle, S. W.; et al. *Conceptual-Model Programming: A Manifesto*, chapter Programming with Conceptual Models. Berlin: Springer, 2011, ISBN 978-3-642-15865-0, pp. 3–16, doi:10.1007/978-3-642-15865-0_1.

[119] Pastor, O.; Gómez, J.; et al. The OO-Method Approach for Information Systems Modeling: From Object-Oriented Conceptual Modeling to Automated Programming. *Inf. Syst.*, volume 26, no. 7, 2001: pp. 507–534, doi:10.1016/S0306-4379(01)00035-7.

[120] Giachetti, G.; Marín, B.; et al. Updating OO-Method Function Points. In *Quality of Information and Communications Technology, 6th International Conference on the Quality of Information and Communications Technology, QUATIC 2007, Lisbon, Portugal, September 12-14, 2007, Proceedings*, edited by R. J. Machado; F. B. e Abreu; P. R. da Cunha, IEEE Computer Society, 2007, pp. 55–64, doi: 10.1109/QUATIC.2007.20.

[121] Martins, B. F. The OntoOO-Method: An Ontology-Driven Conceptual Modeling Approach for Evolving the OO-Method. In *Advances in Conceptual Modeling – ER 2019 Workshops FAIR, MREBA, EmpER, MoBiD, OntoCom, and ER Doctoral Symposium Papers, Salvador, Brazil, November 4-7, 2019, Proceedings*, *Lecture Notes in Computer Science*, volume 11787, edited by G. Guizzardi; F. Gailly; R. S. P. Maciel, Springer, 2019, pp. 247–254, doi:10.1007/978-3-030-34146-6_23.

[122] Buarque, A.; Castro, J.; et al. The Role of NFRs When Transforming i* Requirements Models into OO-Method Models. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, edited by S. Y. Shin; J. C. Maldonado, ACM, 2013, pp. 1305–1306, doi: 10.1145/2480362.2480605.

[123] Alencar, F. M. R.; Marín, B.; et al. From *i** to OO-Method: Problems and Solutions. In *Proceedings of the 4th International* i* *Workshop, Hammamet, Tunisia, June 07-08, 2010*, *CEUR Workshop Proceedings*, volume 586, edited by J. B. de Castro; X. Franch; J. Mylopoulos; E. S. K. Yu, CEUR-WS.org, 2010, pp. 9–14. Available from: `http://ceur-ws.org/Vol-586/iStar10-paper02.pdf`

[124] Zaninotto, F.; Potencier, F. *The Definitive Guide to Symfony.* Apress, 2007, ISBN 978-1-59-059786-6.

[125] Steinberg, D.; Budinsky, F.; et al. *EMF: Eclipse Modeling Framework.* Addison-Wesley Professional., second edition, 2008, ISBN 978-0-321-33188-5. Available from: `https://www.informit.com/store/emf-eclipse-modeling-framework-9780321331885`

[126] Doschek, N. Eclipse EMF.cloud. [online], 2020, [Accessed 11 February 2022]. Available from: `https://projects.eclipse.org/projects/ecd.emfcloud`

[127] Koegel, M.; Helming, J. EMFStore: A Model Repository for EMF Models. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering – Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, edited by J. Kramer; J. Bishop; P. T. Devanbu; S. Uchitel, ACM, 2010, pp. 307–308, doi: 10.1145/1810295.1810364.

[128] Gérard, S.; Dumoulin, C.; et al. Papyrus: A UML2 Tool for Domain-Specific Language Modeling. In *Model-Based Engineering of Embedded Real-Time Systems – International Dagstuhl Workshop, Dagstuhl Castle, Germany, November 4-9, 2007. Revised Selected Papers*, *Lecture Notes in Computer Science*, volume 6100, edited by H. Giese; G. Karsai; E. Lee; B. Rumpe; B. Schätz, Springer, 2007, pp. 361–368, doi:10.1007/978-3-642-16277-0_19.

[129] JAXenter. Eclipse Modeling Framework – Interview with Ed Merks. [online], 2010, [Accessed 21 February 2022]. Available from: `https://jaxenter.com/eclipse-modeling-framework-interview-with-ed-merks-100007.html`

[130] Eclipse Foundation, Inc. EMF Javadoc. [online], 2020, [Accessed 12 February 2022]. Available from: `https://www.eclipse.org/modeling/emf/javadoc/`

[131] Kchaou, M.; Khlif, W.; et al. Transformation of BPMN Model into an OWL2 Ontology. In *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2021, Online Streaming, April 26-27, 2021*, edited by R. Ali; H. Kaindl; L. A. Maciaszek, SCITEPRESS, 2021, pp. 380–388, doi:10.5220/0010479603800388.

[132] Hodrob, R.; Jarrar, M. Mapping ORM into OWL 2. In *Proceedings of the 1st International Conference on Intelligent Semantic Web-Services and Applications, ISWSA 2010, Amman, Jordan, June 14-16, 2010*, edited by A. J. Alnsour; S. A. Aljawarneh, ACM, 2010, p. 9, doi:10.1145/1874590.1874599.

[133] Almeida, J. P. A.; Guizzardi, G.; et al. gUFO: A Lightweight Implementation of the Unified Foundational Ontology (UFO). [online], 2019, [Accessed 9 June 2022]. Available from: `http://purl.org/nemo/doc/gufo`

[134] Bergman, M. 50 Ontology Mapping and Alignment Tools. [online], 2014, [Accessed 10 July 2022]. Available from: `https://www.mkbergman.com/1769/50-ontology-mapping-and-alignment-tools/`

[135] Ouali, I.; Ghozzi, F.; et al. Ontology Alignment using Stable Matching. In *Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 23rd International Conference KES-2019, Budapest, Hungary, 4-6 September 2019*, *Procedia Computer Science*, volume 159, edited by I. J. Rudas; J. Csirik; C. Toro; J. Botzheim; R. J. Howlett; L. C. Jain, Elsevier, 2019, pp. 746–755, doi:10.1016/j.procs.2019.09.230.

[136] Euzenat, J.; Valtchev, P. Similarity-Based Ontology Alignment in OWL-Lite. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, edited by R. L. de Mántaras; L. Saitta, IOS Press, 2004, pp. 333–337.

[137] Dähling, S.; Razik, L.; et al. OWL2Go: Auto-Generation of Go Data Models for OWL Ontologies with Integrated Serialization and Deserialization Functionality. *SoftwareX*, volume 12, 2020: p. 100571, doi:10.1016/j.softx.2020.100571.

[138] Hnatkowska, B.; Woroniecki, P. Transformation of OWL2 Property Axioms to Groovy. In *SOFSEM 2018: Theory and Practice of Computer Science – 44th International Conference on Current Trends in Theory and Practice of Computer Science, Krems, Austria, January 29 – February 2, 2018, Proceedings*, *Lecture Notes in Computer Science*, volume 10706, edited by A. M. Tjoa; L. Bellatreche; S. Biffl; J. van Leeuwen; J. Wiedermann, Springer, 2018, pp. 269–282, doi:10.1007/978-3-319-73117-9_19.

[139] Wotawa, F.; Bozic, J.; et al. Ontology-Based Testing: An Emerging Paradigm for Modeling and Testing Systems and Software. In *13th IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2020, Porto, Portugal, October 24-28, 2020*, IEEE, 2020, pp. 14–17, doi:10.1109/ICSTW50294.2020.00020.

[140] Ledvinka, M.; Kremen, P. A Comparison of Object-Triple Mapping Libraries. *Semantic Web*, volume 11, no. 3, 2020: pp. 483–524, doi:10.3233/SW-190345.

[141] Ledvinka, M.; Kostov, B.; et al. JOPA: Efficient Ontology-Based Information System Design. In *The Semantic Web – ESWC 2016 Satellite Events, Heraklion, Crete, Greece, May 29 – June 2, 2016, Revised Selected Papers*, *Lecture Notes in Computer Science*, volume 9989, edited by H. Sack; G. Rizzo; N. Steinmetz; D. Mladenic; S. Auer; C. Lange, 2016, pp. 156–160, doi:10.1007/978-3-319-47602-5_31.

[142] Mannaert, H.; De Bruyn, P.; et al. On the Interconnection of Cross-cutting Concerns Within Hierarchical Modular Architectures. *IEEE Transactions on Engineering Management*, 2020: pp. 1–16, ISSN 1558-0040, doi:10.1109/TEM.2020.3040227.

[143] Mannaert, H.; De Cock, K.; et al. On the Realization of Meta-Circular Code Generation: The Case of the Normalized Systems Expanders. *ICSEA 2019*, 2019: pp. 171–176, ISSN 2308-4235.

[144] NSX. µRadiant. [online], 2023, [Accessed 27 April 2023]. Available from: `https://foundation.stars-end.net/docs/tools/micro-radiant/`

[145] Oorts, G.; Mannaert, H.; et al. On the Evolvable and Traceable Design of (Under)graduate Education Programs. In *Advances in Enterprise Engineering X – 6th Enterprise Engineering Working Conference, EEWC 2016, Funchal, Madeira Island, Portugal, May 30 – June 3, 2016, Proceedings*, *Lecture Notes in Business Information Processing*, volume 252, edited by D. Aveiro; R. Pergl; D. Gouveia, Springer, 2016, pp. 86–100, doi:10.1007/978-3-319-39567-8_6.

[146] Oorts, G.; Mannaert, H.; et al. Exploring Design Aspects of Modular and Evolvable Document Management. In *Advances in Enterprise Engineering XI – 7th Enterprise Engineering Working Conference, EEWC 2017, Antwerp, Belgium, May 8-12, 2017, Proceedings*, *Lecture Notes in Business Information Processing*, volume 284, edited by D. Aveiro; R. Pergl; G. Guizzardi; J. P. A. Almeida; R. Magalhães; H. Lekkerkerk, Springer, 2017, pp. 126–140, doi:10.1007/978-3-319-57955-9_10.

[147] Oorts, G.; Mannaert, H.; et al. Toward Evolvable Document Management for Study Programs Based on Modular Aggregation Patterns. In *PATTERNS 2017: the Ninth International Conferences on Pervasive Patterns and Applications, February 19-23, 2017, Athens, Greece/Mannaert, Herwig [edit.]; et al.*, 2017, pp. 34–39.

[148] Knaisl, V.; Pergl, R. Proposing Ontology-Driven Content Modularization in Documents Based on the Normalized Systems Theory. In *Trends and Innovations in Information Systems and Technologies – Volume 1, WorldCIST 2020, Budva, Montenegro, 7-10 April 2020*, *Advances in Intelligent Systems and Computing*, volume 1159, edited by Á. Rocha; H. Adeli; L. P. Reis; S. Costanzo; I. Orovic; F. Moreira, Springer, 2020, pp. 45–54, doi:10.1007/978-3-030-45688-7_5.

[149] Slifka, J.; Pergl, R. Laying the Foundation for Design System Ontology. In *Trends and Innovations in Information Systems and Technologies – Volume 1, WorldCIST 2020, Budva, Montenegro, 7-10 April 2020*, *Advances in Intelligent Systems and Computing*, volume 1159, edited by Á. Rocha; H. Adeli; L. P. Reis; S. Costanzo; I. Orovic; F. Moreira, Springer, 2020, pp. 778–787, doi:10.1007/978-3-030-45688-7_76.

[150] Wilkinson, M. D.; Dumontier, M.; et al. The FAIR Guiding Principles for Scientific Data Management and Stewardship. *Scientific Data*, volume 3, 2016, doi:10.1038/sdata.2016.18.

[151] Little, J. Analogy in Science: Where do we go from here? *Rhetoric Society Quarterly*, volume 30, no. 1, 2000: pp. 69–92, doi:10.1080/02773940009391170.

[152] Hofstadter, D. Analogy as the Core of Cognition. In *The Analogical Mind: Perspectives from Cognitive Science*, edited by D. Gentner; K. J. Holyoak; B. N. Kokinov, MIT Press, 2001, pp. 499–538.

[153] Oxford University Press. Definition of Lingua Franca. [online], 2022, [Accessed 9 July 2022]. Available from: `https://www.lexico.com/definition/lingua_franca`

[154] OWL Working Group. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax. [online], 2012, second edition, [Accessed 9 April 2022]. Available from: `https://www.w3.org/TR/owl2-syntax/`

[155] Apache Software Foundation. Apache Jena. [online], 2022, [Accessed 9 May 2022]. Available from: `https://jena.apache.org/`

[156] Zedlitz, J.; Luttenberger, N. Conceptual Modelling in UML and OWL-2. *International Journal on Advances in Software*, volume 7, no. 1, 2014: pp. 182–196, ISSN 1942-2628.

[157] Husár, R. *Searching Inside (Onto)UML Structural Conceptual Models.* Master's thesis, Czech Technical University in Prague, Czech Republic, 2021, [Accessed 9 April 2022]. Available from: `https://dspace.cvut.cz/handle/10467/94498`

[158] Object Management Group. Business Process Model and Notation (BPMN). Technical report, Object Management Group (OMG), January 2011, [Accessed 3 March 2022]. Available from: `https://www.omg.org/spec/BPMN/2.0/PDF`

[159] Sanfilippo, E. M.; Borgo, S.; et al. Towards an Ontological Analysis of BPMN. In *Proceedings of 10th Workshop on Knowledge Engineering and Software Engineering (KESE10) co-located with 21st European Conference on Artificial Intelligence (ECAI 2014), Prague, Czech Republic, August 19 2014*, *CEUR Workshop Proceedings*, volume 1289, edited by G. J. Nalepa; J. Baumeister, CEUR-WS.org, 2014, p. 9. Available from: `http://ceur-ws.org/Vol-1289/kese10-06_submission_9.pdf`

[160] Natschläger, C. Towards a BPMN 2.0 Ontology. In *Business Process Model and Notation – Third International Workshop, BPMN 2011, Lucerne, Switzerland, November 21-22, 2011. Proceedings*, *Lecture Notes in Business Information Processing*, volume 95, edited by R. M. Dijkman; J. Hofstetter; J. Koehler, Springer, 2011, pp. 1–15, doi:10.1007/978-3-642-25160-3_1.

[161] Craft.CASE Ltd. Craft.CASE: Business Process Analysis. [online], 2021, [Accessed 28 December 2021]. Available from: `http://craftcase.com`

[162] Breitman, K.; Casanova, M.; et al. *Semantic Web: Concepts, Technologies and Applications.* NASA Monographs in Systems and Software Engineering, Springer London, 2010, ISBN 978-1-84-996621-4.

[163] Franconi, E.; Mosca, A.; et al. ORM2: Formalisation and Encoding in OWL2. In *On the Move to Meaningful Internet Systems: OTM 2012 Workshops, Confederated International Workshops: OTM Academy, Industry Case Studies Program, EI2N, INBAST, META4eS, OnToContent, ORM, SeDeS, SINCOM, and SOMOCO 2012, Rome, Italy, September 10-14, 2012. Proceedings*, *Lecture Notes in Computer Science*, volume 7567, edited by P. Herrero; H. Panetto; R. Meersman; T. S. Dillon, Springer, 2012, pp. 368–378, doi:10.1007/978-3-642-33618-8_51.

[164] World Wide Web Consortium; et al. SPARQL 1.1 Query Language: W3C Recommendation 21 March 2013. [online], 2013, [Accessed 13 April 2022]. Available from: `https://www.w3.org/TR/2013/REC-sparql11-query-20130321/`

[165] Pérez, J.; Arenas, M.; et al. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.*, volume 34, no. 3, 2009: pp. 16:1–16:45, doi:10.1145/1567274.1567278.

[166] uml-diagrams.org. UML Diagram Examples: Online Shopping. [online], 2022, [Accessed 12 June 2022]. Available from: `https://www.uml-diagrams.org/examples/online-shopping-example.html`

[167] Tuma, J.; Pícka, M.; et al. Generated Report of the ORD BORM Model. *Acta Informatica Pragensia*, volume 4, no. 1, 2015: pp. 30–43, doi:10.18267/j.aip.58.

[168] Smith, J. Modeliosoft: Shopping Cart, version 1.7. [online], 2013, [Accessed 15 June 2022]. Available from: `https://www.modeliosoft.com/example/AnalysisAndDesign.pdf`

[169] Litium AB. Litium Docs. [online], 2022, [Accessed 19 June 2022]. Available from: `https://docs.litium.com`

[170] Sarka, D. Lucient: Object-Role Modeling Part 1. [online], 2007, [Accessed 20 June 2022]. Available from: `https://lucient.com/blog/object-role-modeling-part-1/`

[171] Object Management Group. Business Process Model and Notation (BPMN). Technical report, Object Management Group (OMG), January 2009, [Accessed 12 March 2022]. Available from: `https://www.omg.org/spec/BPMN/1.2/PDF`

[172] Šenkýř, D.; Kroha, P. Patterns in Textual Requirements Specification. In *Proceedings of the 13th International Conference on Software Technologies, ICSOFT 2018, Porto, Portugal, July 26-28, 2018*, edited by L. A. Maciaszek; M. van Sinderen, SciTePress, 2018, pp. 231–238, doi:10.5220/0006827302310238.

Reviewed Publications of the Author Relevant to the Thesis

[A.1] Suchánek, M.; Pergl, R. Evolvable Documents – an Initial Conceptualization. *PATTERNS 2018, The Tenth International Conference on Pervasive Patterns and Applications*. Wilmington: IARIA, 2018. p. 39-44. ISSN 2308-3557. ISBN 978-1-61208-612-5. Available from: `https://www.thinkmind.org/index.php?view=article&articleid=patterns_2018_4_10_78002`

The paper has been cited in:

- Knaisl, V. Proposing an Architecture of an Intelligent Evolvable Document Generation System Based on the Normalized Systems Theory. In: *Enterprise and Organizational Modeling and Simulation*. Springer, Cham, 2019. p. 70-81. 1. ISSN 1865-1348. ISBN 978-3-030-35645-3. Available from: `https://link.springer.com/chapter/10.1007/978-3-030-35646-0_6`
- Knaisl, V.; Pergl, R. Proposing Ontology-Driven Content Modularization in Documents Based on the Normalized Systems Theory. In: *Trends and Innovations in Information Systems and Technologies*. Springer, Cham, 2020. p. 45-54. ISSN 2194-5357. ISBN 978-3-030-45687-0. Available from: `https://link.springer.com/chapter/10.1007/978-3-030-45688-7_5`
- Knaisl, V.; Pergl, R. Improving Document Evolvability based on Normalized Systems Theory. In: *Information Systems and Technologies*. Springer, Cham, 2022. p. 131-140. ISSN 2367-3370. ISBN 978-3-031-04818-0. Available from: `https://link.springer.com/chapter/10.1007/978-3-031-04819-7_14`

[A.2] Suchánek, M.; Pergl, R. Towards Evolvable Documents with a Conceptualization-Based Case Study. In: *International Journal on Advances in Intelligent Systems*. Wilmington: IARIA, 2018, 11(3&4). p. 212–223. ISSN 1942-2679. Available from: `https://www.thinkmind.org/index.php?view=article&articleid=intsys_v11_n34_2018_8`

[A.3] Suchánek, M.; Pergl, R. Evolvability Evaluation of Conceptual-Level Inheritance Implementation Patterns. In: *PATTERNS 2019, The Eleventh International Conference on Pervasive Patterns and Applications.* Wilmington: IARIA, 2019. p. 1–6. ISSN 2308-3557. ISBN 978-1-61208-612-5. Available from: `https://www.thinkmind.org/index.php?view=article&articleid=patterns_2019_1_10_78001`

The paper has been cited in:

- Haerens, G. Ontological Analysis of the Evolvability of the Network Firewall Rule Base. In: *Proceedings of the 20th CIAO! Doctoral Consortium, and Enterprise Engineering Working Conference Forum 2020, co-located with 10th Enterprise Engineering Working Conference (EEWC 2020), Bozen / Bolzano, Italy, September 28th, October 19th and November 9th-10th, 2020.* 2020, p. 1–15. Available from: `https://ceur-ws.org/Vol-2825/paper2.pdf`
- Haerens, G. On the Evolvability of the TCP-IP Based Network Firewall Rule Base. Dissertation Thesis. University of Antwerp, Antwerp (Belgium), 2021. Available from: `https://hdl.handle.net/10067/1834610151162165141`

[A.4] Suchánek, M.; Slifka, J. Evolvable and Machine-Actionable Modular Reports for Service-Oriented Architecture. In: *Enterprise and Organizational Modeling and Simulation.* Springer: Cham, 2019. p. 43–59. 1. ISSN 1865-1348. ISBN 978-3-030-35645-3. Available from: `https://link.springer.com/chapter/10.1007/978-3-030-35646-0_4`

The paper has been cited in:

- Joshi, S.; Rambola, R. Context-Aware Service Oriented Architecture for Secure Data Transmission. In: *Journal of University of Shanghai for Science and Technology.* vol. 22, pp. 257 - 262, ISSN 1007-6735. 2020. Available from: `https://jusst.org/context-aware-service-oriented-architecture-for-secure-data-transmission-2/`
- Bastidas, V. ArchiSmartCity: Modelling the Alignment of Services and Information in Smart City Architectures. Dissertation Thesis. National University of Ireland, Maynooth (Ireland), ProQuest Dissertations Publishing, 2021. Available from: `https://mural.maynoothuniversity.ie/14874/`

[A.5] Suchánek, M. Designing an Ontology for Semantic Integration of Various Conceptual Models. In: *Enterprise and Organizational Modeling and Simulation.* Springer: Cham, 2019. p. 3–17. 1. ISSN 1865-1348. ISBN 978-3-030-35645-3. Available from: `https://link.springer.com/chapter/10.1007/978-3-030-35646-0_1`

[A.6] Suchánek, M.; Pergl, R. Mapping UFO-B to BPMN, BORM, and UML Activity Diagram. In: *Enterprise and Organizational Modeling and Simulation.* Springer: Cham, 2019. p. 82–98. 1. ISSN 1865-1348. ISBN 978-3-030-35645-3. Available from: `https://link.springer.com/chapter/10.1007/978-3-030-35646-0_7`

The paper has been cited in:

- Costa M.Z., Guizzardi G., Almeida J.P.A. On Capturing Legal Knowledge in Ontology and Process Models Combined. In: *Frontiers in Artificial Intelligence and Applications*, Legal Knowledge and Information Systems - JURIX 2022: The Thirty-fifth Annual Conference, Saarbrücken, Germany, 14-16 December 2022. Amsterdam: IOS Press, 2022, 362. p. 267–272. ISBN 978-1-64368-364-5. Available from: `https://ebooks.iospress.nl/doi/10.3233/FAIA220478`

[A.7] Suchánek, M.; Mannaert, H.; Uhnák, P.; Pergl, R. Bi-directional Transformation between Normalized Systems Elements and Domain Ontologies in OWL. In: *15th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE).* Porto: SciTePress – Science and Technology Publications, 2020. p. 74–85. ISSN 2184-4895. ISBN 978-989-758-421-3. Available from: `https://www.scitepress.org/Link.aspx?doi=10.5220/0009356800740085`

The paper has been cited in:

- Mannaert, H.; De Cock, K.; Uhnák, P.; Verelst, J. On the Realization of Meta-Circular Code Generation and Two-Sided Collaborative Metaprogramming. In: *International Journal on Advances in Software.* Wilmington: IARIA, 2020, 13(3&4). p. 149–159. ISSN 1942-2628. Available from: `https://www.thinkmind.org/index.php?view=article&articleid=soft_v13_n34_2020_4`

[A.8] Suchánek, M.; Pergl, R. Case-Study-Based Review of Approaches for Transforming UML Class Diagrams to OWL and Vice Versa. In: *IEEE 22nd Conference on Business Informatics (CBI).* Los Alamitos: IEEE Computer Society, 2020. p. 270–279. vol. 1. ISBN 978-1-7281-9926-9. Available from: `https://ieeexplore.ieee.org/document/9140231`

The paper has been cited in:

- Chiarcos, C.; Gkirtzou, K.; Ionov, M.; Kabashi, B.; Khan, A.F.; Truica, C.O. Modelling Collocations in OntoLex-FrAC. In: *Proceedings of the 2020 Globalex Workshop on Linked Lexicography*, European Language Resources Association, pp. 10–18, 2020. Available from: `https://aclanthology.org/2022.gwll-1.3/`

- Huber, F.; Hagel, G. Work-In-Progress: Converting Textual Software Engineering Class Diagram Exercises to UML Models. In: *Proceedings fo the 2022 IEEE Global Engineering Education Conference (EDUCON 2022)*, pp. 1–3, 2022. ISSN 2165-9567. Available from: `https://ieeexplore.ieee.org/document/9766593`

[A.9] Suchánek, M.; Pergl, R. Evolvability Analysis of Multiple Inheritance and Method Resolution Order in Python. In: *PATTERNS 2020, The Twelfth International Conference on Pervasive Patterns and Applications.* Wilmington: IARIA, 2020. p. 19–24. ISSN 2308-3557. ISBN 978-1-61208-783-2. Available from: `http://www.thinkmind.org/index.php?view=article&articleid=patterns_2020_2_10_79`

[A.10] Suchánek, M.; Mannaert, H.; Uhnák, P.; Pergl, R. Towards Evolvable Ontology-Driven Development with Normalized Systems. In: *Evaluation of Novel Approaches to Software Engineering.* Cham: Springer International Publishing, 2021. p. 208–231. Communications in Computer and Information Science. ISSN 1865-0929. ISBN 978-3-030-70005-8. Available from: `https://link.springer.com/chapter/10.1007/978-3-030-70006-5_9`

The paper has been cited in:

- Slifka, J.; Pergl, R. User Interface Modelling Languages for Normalised Systems: Systematic Literature Review. In: *Information Systems and Technologies.* Springer, Cham, 2022. p. 349-358. ISSN 2367-3370. ISBN 978-3-031-04828-9. Available from: `https://link.springer.com/chapter/10.1007/978-3-031-04829-6_31`

[A.11] Suchánek, M.; Pergl, R. Pattern-Based Ontological Transformations for RDF Data using SPARQL. In: *PATTERNS 2021, The Thirteenth International Conference on Pervasive Patterns and Applications.* Wilmington: IARIA, 2021. p. 11–16. ISSN 2308-3557. ISBN 978-1-61208-850-1. Available from: `https://www.thinkmind.org/index.php?view=article&articleid=patterns_2021_1_20_78002`

[A.12] Suchánek, M.; Mannaert, H.; Uhnák, P.; Pergl, R. Building Normalized Systems from Domain Models in Ecore. In: *SOMET 2021, New Trends in Intelligent Software Methodologies, Tools and Techniques.* Amsterdam: IOS Press, 2021. p. 169–182. Frontiers in Artificial Intelligence and Applications. vol. 337. ISBN 978-1-64368-194-8. Available from: `https://ebooks.iospress.nl/doi/10.3233/FAIA210018`

[A.13] Suchánek, M.; Pergl, R. Representing BORM Process Models using OWL and RDF. In: *Proceedings of the 13th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management - Volume 2: KEOD*. Porto: SciTePress – Science and Technology Publications, 2021. p. 170–177. ISBN 978-989-758-533-3, ISSN 2184-3228. Available from: `https://www.scitepress.org/Link.aspx?doi=10.5220/0010653900003064`

[A.14] Šenkýř, D.; Suchánek, M.; Kroha, P.; Mannaert, H.; Pergl, R. Expanding Normalized Systems from Textual Domain Descriptions using TEMOS. In: *Journal of Intelligent Information Systems*. Springer, 2022. ISSN 1573-7675. Available from: `https://link.springer.com/article/10.1007/s10844-022-00706-8`

[A.15] Suchánek, M.; Mannaert, H.; Pergl, R. Towards Normalized Systems from RDF with SPARQL. In: *SOMET2022, New Trends in Intelligent Software Methodologies, Tools and Techniques*. Amsterdam: IOS Press, 2022. p. 609-620. Frontiers in Artificial Intelligence and Applications. vol. 335. ISBN 978-1-64368-316-4. Available from: `https://ebooks.iospress.nl/doi/10.3233/FAIA220290`

Remaining Publications of the Author Relevant to the Thesis

[A.16] Suchánek, M. *Conceptual Modelling with Respect to Precise Technical Specification.* [Technical Report] 2019. Report no. TR-FIT-19-01. Faculty of Information Technology, CTU in Prague, Prague, Czech Republic, 2019.

[A.17] Suchánek, M. *Conceptual Modelling with Respect to Precise Technical Specification.* Defense date 2019-06-24. Doctoral Minimum. Supervised by R. Pergl and P. Kroha. Faculty of Information Technology, CTU in Prague, Prague, Czech Republic, 2019.

[A.18] Suchánek, M.; Mannaert, H.; Pergl, R. *Designing an Architecture of Normalized Systems Gateway Ontology for Conceptual Models.* Doctoral Days 2019-10-21. Supervised by R. Pergl and H. Mannaert. Faculty of Business & Economics and Antwerp Management School, University of Antwerp, Antwerp, Belgium, 2020.

Remaining Publications of the Author

[A.19] Suchánek, M.; Pergl, R. Data Stewardship Wizard for Open Science. In: *Data a znalosti & WIKT*. Brno: Vysoké učení technické v Brně. Fakulta informačních technologií, 2018. p. 121–125. 1. ISBN 978-80-214-5679-2.

The paper has been cited in 5 publications.[1]

[A.20] Pergl, R.; Hooft, R.; Suchánek, M.; Knaisl, V.; Slifka J. "Data Stewardship Wizard": A Tool Bringing Together Researchers, Data Stewards, and Data Experts around Data Management Planning. In: *Codata Science Journal*. Paris: CODATA – International Council for Science. 2019, 18(1), p. 1–8. ISSN 1683-1470. Available from: `https://datascience.codata.org/articles/10.5334/dsj-2019-059/`

The paper has been cited in 31 publications.[1]

[A.21] Suchánek, M.; Hooft, R.; Bourhy, K. Progress on Data Stewardship Wizard during BioHackathon Europe 2020. [Technical Report] 2020. Available from: `https://biohackrxiv.org/9mnkb/`

The paper has been cited in 1 publication.[1]

[A.22] Vácha, T.; Blizničenko, J.; Barič, K.; Kuzmič, M.; Suchánek, M.; Kandusová, V.; Cabrnochová, K. SMART CITY COMPASS: Software pro podporu implementace a evaluace chytrých opatření ve městech. [Research Report] Praha: CTU. University Centre of Energy Efficient Buildings, 2020. Report no. TJ02000344-V2.

[A.23] Schultes, E.; Magagna, B.; Hettne, K. M.; Pergl, R.; Suchánek, M.; Kuhn, T. Reusable FAIR Implementation Profiles as Accelerators of FAIR Convergence. In: *Advances in Conceptual Modeling - ER 2020 Workshops CMAI, CMLS, CMOMM4FAIR, CoMoNoS, EmpER, Vienna, Austria, November 3-6, 2020, Proceedings*. Wien: Springer. 2020. p. 138–147. Lecture Notes in Computer Science. Available from: `https://link.springer.com/chapter/10.1007/978-3-030-65847-2_13`

The paper has been cited in 27 publications.[1]

[1] Number of citations based on results from Google Scholar and Semantic Scholar.

[A.24] Miksa, T.; Walk, P.; Neish, P.; Oblasser, S.; Murray, H.; Renner, T.; Jacquemot-Perbal, M.-C.; Cardoso, J. et al. Application Profile for Machine-Actionable Data Management Plans. In: *Codata Science Journal*. CODATA. 2021, 20(1), ISSN 1683-1470. Available from: `https://datascience.codata.org/articles/10.5334/dsj-2021-032/`

The paper has been cited in 3 publications.[1]

[A.25] Suchánek, M.; Alper, P.; Slifka, J.; Děd, V.; Barry, N.D.; Lieby, P.; Vidak, M.; Zlender, N. DS Wizard Meets DAISY: A Romance Solving Data Protection Requirements in Data Management Planning. [Technical Report] 2021. Available from: `https://biohackrxiv.org/cuvqw/`

[A.26] Cardoso, J.; Castro, L.J.; Ekaputra, F.J.; Jacquemot, M.C.; Suchánek, M.; Miksa, T.; Borbinha, J. DCSO: Towards an Ontology for Machine-Actionable Data Management Plans. In: *Journal of Biomedical Semantics*. BioMed Central. 2022, ISSN 2041-1480. Available from: `https://jbiomedsem.biomedcentral.com/articles/10.1186/s13326-022-00274-4`

The paper has been cited in 2 publications.[1]

[A.27] Basajja, M.; Suchánek, M.; Taye, G.T.; Amare, S.; Nambobi, M.; Folorunso, S.; Plug, R.; Oladipo, F.O. et al. Proof of Concept and Horizons on Deployment of FAIR Data Points in the COVID-19 Pandemic. In: *Data Intelligence*. MIT Press. 2022, ISSN 2641-435X. Available from: `https://direct.mit.edu/dint/article/4/4/917/112733`

The paper has been cited in 1 publication.[1]

[A.28] Benhamed, O.M.; Burger, K.; Kaliyaperumal, R.; Bonino da Silva Santos, L.O.; Suchánek, M.; Slifka, J.; Wilkinson, M.D. The FAIR Data Point: Interfaces and Tooling. In: *Data Intelligence*. MIT Press. 2022, ISSN 2641-435X. Available from: `https://direct.mit.edu/dint/article/doi/10.1162/dint_a_00161`

The paper has been cited in 4 publications.[1]

Selected Relevant Supervised Theses

[A.29] Rolník, M.-D. *Modular Web-based Information System for Leisure Complex.* Bachelor Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2018. Available from: http://hdl.handle.net/10467/76984

[A.30] Hamza, J. *Modular Web-based Reservation System for Clinic.* Bachelor Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2018. Available from: http://hdl.handle.net/10467/76980

[A.31] Junek, M. *Modular Web-based Information System for Small and Medium-sized Enterprises.* Bachelor Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2018. Available from: http://hdl.handle.net/10467/76822

[A.32] Dunaevskiy, S. *Study Project Management Information System.* Bachelor Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2019. Available from: http://hdl.handle.net/10467/86181

[A.33] Starý, T. *System for Composing and Managing Evolvable Documents.* Bachelor Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2019. Available from: http://hdl.handle.net/10467/83133

[A.34] Jirásko, P. *Academic Collaboration Information System.* Master Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2019. Available from: http://hdl.handle.net/10467/82709

[A.35] Volodin, V. *Personal Expense Management Web Application.* Bachelor Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2020. Available from: http://hdl.handle.net/10467/88352

[A.36] Janáček, M. *Lightweight Enterprise Relationship Management Information System.* Bachelor Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2020. Available from: http://hdl.handle.net/10467/88170

[A.37] Gallas, M. *Information System for Paper Document Management.* Bachelor Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2020. Available from: `http://hdl.handle.net/10467/88164`

[A.38] Vaner, M. *Web Information System for Electronic Classbook.* Bachelor Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2021. Available from: `http://hdl.handle.net/10467/94888`

[A.39] Zotkina, O. *Analysis of Model-Driven Development Methods for Generating Web-Based Information Systems.* Bachelor Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2021. Available from: `http://hdl.handle.net/10467/94919`

[A.40] Chodounská, M. *Design Patterns and Principles Analysis for Home Assistant Application using Reverse Engineering Methods.* Bachelor Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2021. Available from: `http://hdl.handle.net/10467/96911`

[A.41] Svoboda, P. *Workflow: Web Application Implementing Company Processes using State Machines.* Master Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2021. Available from: `http://hdl.handle.net/10467/94595`

[A.42] Kužmová, A. *Design of Task Management System for Chain Stores.* Bachelor Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2021. Available from: `http://hdl.handle.net/10467/95029`

[A.43] Baláž, R. *Architecture Design of a Universal License Server.* Master Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2022. Available from: `http://hdl.handle.net/10467/102070`

[A.44] Machačová, T. *Streamlining the Use of Conceptual Models in OntoUML with RDF Technologies.* Master Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2023. Available from: `http://hdl.handle.net/10467/107233`

Appendix A

Electronic Resources

The electronic attachment contains all the artefacts and their implementation emerged from this dissertation thesis. The structure is as described by Figure A.1. Parts of the appendix are mentioned from the relevant chapters and sections of the thesis. Note that the electronic attachment is available upon request via email or via the persistent URL[*] following permission approval.

Figure A.1: Contents of the electronic attachment

```
├── cm-ontologies ........................ ontologies for conceptual modelling
├── demonstration ........................ demonstration inputs and results
│   ├── input-models ..................... input models in original and RDF formats
│   ├── scenario-1 ....................... resources of the CM-to-NS scenario
│   ├── scenario-2 ....................... resources of the semantic integration scenario
│   ├── scenario-3 ....................... resources of the evolvability scenario
│   └── scenario-4 ....................... resources of the NS-to-CM scenario
├── gateway-ontology ..................... gateway ontology and mappings
│   ├── layers ........................... gateway ontology (layers)
│   └── mappers .......................... mappings for conceptual modelling
│       ├── borm-mapping ................. mapping for BORM
│       ├── bpmn-mapping ................. mapping for BPMN
│       ├── orm-mapping .................. mapping for ORM
│       ├── rdfs-owl-mapping ............. mapping for RDFS/OWL
│       └── uml-mapping .................. mapping for UML
├── tools ................................ directory of tools implementation
│   ├── ns-rdf ........................... NS-RDF/OWL project
│   └── sparql-trans ..................... SPARQL Transformation project
├── CITATION.cff ......................... citation file (referring this thesis)
└── README.md ............................ description of the contents
```

[*] http://purl.org/nsgo4cm/electronic-resources