TU WIEN Informatics

A Model Driven Architecture Transformation from Ontological Enterprise Models to Low-code

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Nicholas Arthur Bzowski, BSc, BA
Registration Number 12141086

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Henderik A. Proper, PhD
Assistance: Marien R. Krouwel, PhD

Vienna, May 12, 2025

Nicholas Arthur Bzowski	Henderik A. Proper

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

TU WIEN Informatics

Eine MDA-Transformation von ontologischen Unternehmensmodellen zu Low-Code

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Nicholas Arthur Bzowski, BSc, BA
Matrikelnummer 12141086

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Henderik A. Proper, PhD
Mitwirkung: Marien R. Krouwel, PhD

Wien, 12. Mai 2025

Nicholas Arthur Bzowski | Henderik A. Proper

Erklärung zur Verfassung der Arbeit

Nicholas Arthur Bzowski, BSc, BA

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

Wien, 12. Mai 2025

Nicholas Arthur Bzowski

Acknowledgements

I would first and foremost like to extend my most heartfelt gratitude to my supervisors, Univ.Prof. Henderik A. Proper and Marien R. Krouwel, PhD for the opportunity they gave me to work with them, as well as for their guidance, feedback, and unwavering willingness to assist me in my work throughout the course of my thesis project.

I would also like to thank the lectures of the course *Seminar for Master Students in Business Informatics*, Ao.Univ.Prof. Mag. Dr. Christian Huemer, Univ.-Prof. Dr.-Ing. habil. Fazel Ansari, and O.Univ.Prof.in Dipl.-Ing.in Mag.a Dr.in techn. Gerti Kappel, for their very constructive feedback on my thesis proposal and progress.

Additionally, I would like to thank my colleagues at the Business Informatics Group at TU Wien, many of whom provided valuable advice and encouragement as I worked to finish writing this thesis. Special thanks to Univ.Ass. DI Miki Zehetner for proofreading the German translation of the abstract.

Last but certainly not least, I would like to thank my parents Arthur and Assunta, my brother Matthew, and my sister Julia for all of their love, support, and encouragement throughout my academic journey.

Abstract

As the complexity of modern enterprises and software simultaneously increases, it becomes increasingly challenging for enterprises to maintain business-IT alignment, especially when tumultuous business environments necessitate frequent changes to information systems requirements. Two promising techniques which can improve business-IT alignment are the DEMO method and low-code software development. The DEMO method is used to model the stable essence of the organization of enterprises, while low-code development facilitates agile software development.

In a recent study by Krouwel et al., a model-driven engineering approach was designed to generate low-code software artifacts for the Mendix platform directly from DEMO enterprise models. Although this approach showed promise, the authors noted that the process could potentially be improved by applying the Model Driven Architecture (MDA) approach, whereby the problem and solution spaces of the software are modelled together using models across three levels of abstraction: a computation-independent model (CIM), a platform-independent model (PIM), and a platform-specific model (PSM). This motivated the research problem of this thesis: how do we transform DEMO models to Mendix low-code artifacts using the MDA approach?

This thesis contributes to solving this research problem by applying the agile design science research methodology, through which multiple experimental design cycles were conducted to create transformation mappings from DEMO to Mendix via a PIM, modelled using different UML profiles as candidate modelling languages. Each transformation was demonstrated using an academic case study, and the semantic correctness of the transformations was evaluated. Using fact statements captured by the CIM as the ground truth of the domain, the resulting MDA transformation exhibited a high degree of semantic correctness, thereby demonstrating that the MDA approach can be used to effectively transform DEMO fact models to Mendix low-code application artifacts.

Over the course of this study, three major deliverables were produced: working definitions of the MDA abstraction levels, an MDA transformation meta-design, and an MDA transformation specification using a novel UML profile, *pimUML*. The deliverables and findings of this study benefit academia by providing insight and guidance for future studies applying the MDA approach and benefit industry by contributing to developing a technique to improve business-IT alignment, as well as to combating challenges hindering the adoption of low-code solutions, such as vendor lock-in.

Kurzfassung

Mit der zunehmenden Komplexität von Unternehmen in der modernen Gesellschaft und der zunehmenden Nutzung Software wird es für diese immer herausfordernder, das Business-IT-Alignment aufrechtzuerhalten, vor allem, wenn häufig sich verändernde Geschäftsumgebungen Änderungen der Anforderungen an Informationssysteme erfordern. Zwei vielversprechende Methode, die das Business-IT-Alignment verbessern können, sind die DEMO-Methode und die Low-Code-Softwareentwicklung. Die DEMO-Methode wird verwendet, um das stabile *Wesen* der Unternehmensorganisation zu modellieren, während die Low-Code-Entwicklung eine agile Softwareentwicklung begünstigt.

In einer jüngst durchgeführten Studie von Krouwel et al. wurde ein modellgetriebener Ansatz entwickelt, um Low-Code-Softwareartefakte für die Mendix-Plattform direkt aus DEMO-Unternehmensmodellen zu erzeugen. Obwohl dieser Ansatz vielversprechend ist, stellten die Autoren fest, dass der Prozess durch die Anwendung des MDA-Ansatzes (Model Driven Architecture) verbessert werden könnte. Bei diesem Ansatz werden die Problem- und Lösungsräume der Software mit Hilfe von Modellen auf drei Abstraktionsebenen gemeinsam modelliert: ein computation-independent-model (CIM), ein platform-independent-model (PIM) und ein platform-specific-model (PSM). Dies war die Motivation für die Forschungsfrage dieser Arbeit: Wie können DEMO-Modelle mit Hilfe des MDA-Ansatzes in Mendix-Low-Code-Artefakte transformiert werden?

Konkret wurden für diese These, mithilfe des Agile Design Science Ansatzes, mehrere experimentelle Designzyklen durchgeführt, um Transformationsmappings von DEMO-Modellen zu Mendix Low-Code Artefakten über ein platform-independent-model (PIM) zu erstellen, welches mit verschiedenen UML-Profilen als Modellierungssprachen modelliert wurden. Jede Transformation wurde anhand einer akademischen Fallstudie nachvollziehbar dargestellt, und die semantische Korrektheit der einzelnen Transformationenwurde bewertet. Unter Verwendung des computation-independent-model (CIM) als Ground-Truth der Domain wies die endgültige MDA-Transformation einen hohen Grad an semantischer Korrektheit auf und zeigte damit, dass der MDA-Ansatz zur effektiven Transformation von DEMO-Faktenmodellen in Mendix-Low-Code-Anwendungsartefakte verwendet werden kann.

Im Laufe dieser Studie wurden drei wichtige Ergebnisse erzielt: Arbeitsdefinitionen der MDA-Abstraktionsebenen, ein MDA-Transformations-Metadesign und eine MDA-Transformationsspezifikation unter Verwendung eines neuen UML-Profils, mit dem Namen

pimUML. Die Ergebnisse und Erkenntnisse dieser Studie geben für die Wissenschaft von Nutzen sein, indem sie Einblicke und Anleitungen für künftige Studien zur Anwendung des MDA-Ansatzes, und sind dadurch für die Industrie von Nutzen, indem sie einen Beitrag zur Entwicklung einer Technik zur Verbesserung des Business-IT-Alignments sowie den Einsatz von Low-Code-Lösungen, durch offene Technologien und Standards, begünstigen.

Contents

CHAPTER 1

Introduction

Over the past several decades, the power and usage of information technology in enterprises has grown tremendously. Business computing began in the 1960s as mainframe computer programs, supporting simple automations and inventory control, and has evolved into present-day enterprise information systems (EIS) and management information systems (MIS), supporting day-to-day operations across corporate functional units, such as finance, human resources, engineering, and project management [1, 2]. This advancement has resulted in the role of IT becoming increasingly intertwined with corporate strategy and operations, all while new technology emerges faster and increases in complexity [2, 3]. At the same time, business environments have become ever more tumultuous, with modern enterprises constantly being faced with increasing opportunities and threats relating to competition, customer expectations, regulations, and indeed, digital disruption [4, 5]. As enterprises must be able to simultaneously adapt to these forces of changing business environments and evolving technology, *business-IT alignment* has become a critical key success factor for the enterprise of the information age [6, 5].

1.1 Motivation & Problem Statement

Maintaining business-IT alignment can be a challenging task, largely due to the complexity of modern enterprise information systems. Enterprises themselves are complex entities with several different components or concerns comprising their business operations. As such, information systems can be comprised of various types of enterprise applications to support the different needs of the operations, supporting tasks as processes automation, workflow management, information management, e-commerce, and business intelligence [7]. In order to discern exactly how an application can and should be used, enterprise management must understand their IT needs. Moreover, the application itself should be flexible and adaptable, so that it can adapt quickly as those identified needs evolve or change in highly dynamic business environments.

The complexity of maintaining business-IT alignment has motivated IS researchers to improve the process of software development by pursuing methods to generate software artifacts from business-oriented models [8]. Technologies, practices and approaches have emerged to guide business experts and software engineers to identify business needs, communicate these needs, and rapidly build IT solutions. These include enterprise engineering, model-driven software engineering, and low-code software development.

A popular approach to model-driven software engineering is *Model Driven Architecture*®[1] (MDA). Following MDA, high-level enterprise models can be transformed to concrete code via a series of model-to-model transformations between models at three levels of abstraction. These models span the both the problem and solution spaces, supporting flexible and agile software development [9, 10]. This process is illustrated in Figure 1.1.

MDA starts with high-level models known as *computation-independent models* (CIM). These models depict the business processes and structures which the eventual running software artifacts are to support. CIMs can be transformed, via mappings, to more concrete *platform-independent models* (PIM). PIMs model the software to be built, excluding platform specific constructs. These PIMs can again be further transformed to *platform-specific models* (PSM). PSMs contain enough concrete details such that code can be generated from such models [9].

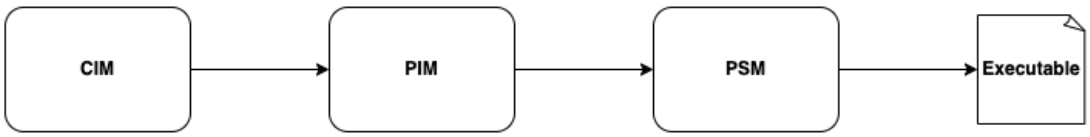


Figure 1.1: The Model Driven Architecture process, described in [11]

A particular challenge in creating mappings between modelling languages is addressing *semantic mismatches* [12, 13, 14]. These arise when there are language constructs that exist in the language of the target model but not in that of the source model and *vice versa* [13]. To accomplish semantically rich transformations, models and transformation must capture semantics from models at higher levels of abstraction and make design decisions based on these semantics to bridge any semantic mismatches. This can be achieved through the use of transformation specifications, consisting of *mapping rules* and *helper functions* to translate semantics between similar constructs and to perform any needed auxiliary computations [10]. These constructs applied to the MDA framework are illustrated in Figure 1.2.

Krouwel et al. [5] recently explored the possibility of mapping DEMO models *directly* to low-code models, using Mendix as the low-code platform of choice. In terms of the MDA approach, this constitutes a direct CIM to PSM transformation. However, they note in the paper that introducing a further level of abstraction – a PIM – could increase the flexibility of such transformations in that one PIM can possibly yield many PSMs of

[1] Model Driven Architecture® is a registered trademark of The Object Management Group (OMG).

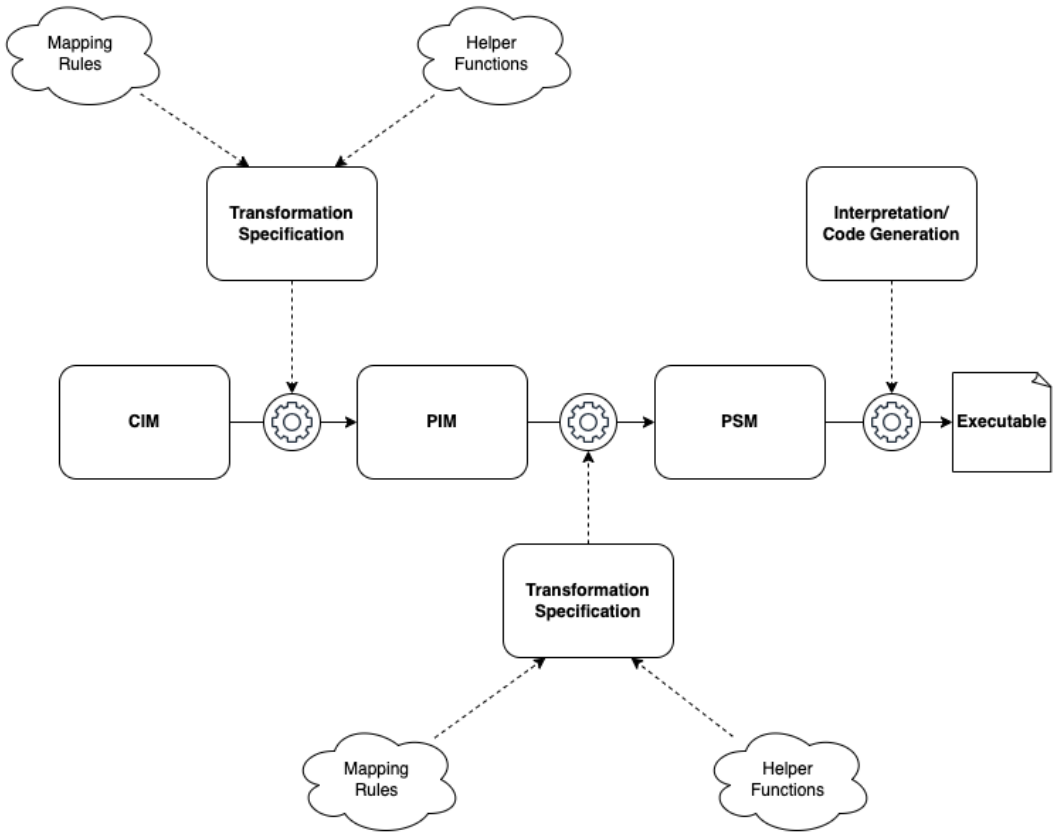


Figure 1.2: The MDA process described in [11], annotated with *mapping rules* and *helper functions* as inputs to the model transformations

different languages; thus the target low-code or high-code platform would not be fixed. Moreover, models at the PIM level could be generated from models of various languages at the CIM level, allowing multiple types of enterprise models to be included and thus amalgamating various types of information in the process [15, 16]. In such an approach, the PIM acts as an interface, possibly mapping constructs between various source models in the CIM to various target models in the PSM, analogous to a network router in the field of computer networking. In order to facilitate this flexible transformation of DEMO to low-code or high-code technologies, where Mendix is an example of a target low-code platform, an intermediary PIM layer is required. However, *there is a gap in the knowledge of how to map enterprise models to low-code via MDA*. This is the research gap that this thesis addresses.

In sum, with the goal of improving business-IT alignment, the research problem of this thesis is the following: *How do we transform DEMO models to Mendix low-code artifacts using the MDA approach?*

1.2 Research Questions

To address the aforementioned research problem, we posed the following research questions:

RQ1 What are the distinct characteristics of the three Model Driven Architecture (MDA) abstraction levels (CIM, PIM, and PSM) and what are their intended uses in the context of Model-Driven Engineering (MDE)?

Answering this research question clarifies the notions and requirements of the models of the three MDA abstraction levels.

RQ2 What concepts or constructs of CIMs, PIMs, PSMs can be identified to guide the usage of MDA for the generation of enterprise applications?

Answering this research question informs the decisions on the scope of the designed artifacts of this thesis and ascertains the suitability of DEMO as a CIM and Mendix as a PSM.

RQ3 What architectural principles can be inferred from the design sprints to the guide the design of a PIM?

Answering this research question allows the key findings that emerged from the foundational meta-design and the experimental design cycles to be generalized so that they can contribute back to the knowledge base.

1.3 Deliverables

The methods to answer the research questions will yield the following deliverables:

RQ1 Deliverable: A table of empirical characteristics and working definitions of each of the MDA abstraction levels (CIM, PIM, and PSM), as found in literature.

RQ2 Deliverable: A meta-design in the form of a conceptual framework.

RQ3 Deliverable: A novel MDA transformation specification from DEMO to Mendix via a PIM and a set of generalized PIM architectural design principles.

1.4 Methodological Approach

One of the most popular research paradigms in Information Systems (IS) research is design science research (DSR). Originally suggested by Hevner, this framework guides researchers to create and evaluate an artifact to contribute to solving a given research problem [17]. A key weakness of DSR is that despite DSR consisting of three key cycles, their prescribed ordering of first the relevance cycle, then the rigour cycle, and then the design cycle [18] tends to result in an overall waterfall-like workflow, such that the

problem space is fixed in the beginning and not revisited during the design phase. For studies in which the research problem may need to evolve as the study progresses, this can be limiting. For such studies in which the problem space requires the ability to evolve, Conboy et al. introduced the Agile Design Science Research Methodology (ADSRM) [19].

At the core of ADSRM is the design science research methodology (DSR) introduced by Hevner [17]. Peffers et al. further explored and elaborated the use of DSR in the research of information systems (IS) by proposing a procedure that is "structured in a nominally sequential order" which can be followed by IS researchers using DSR in their studies [20]. This procedure is illustrated in Figure 1.3.

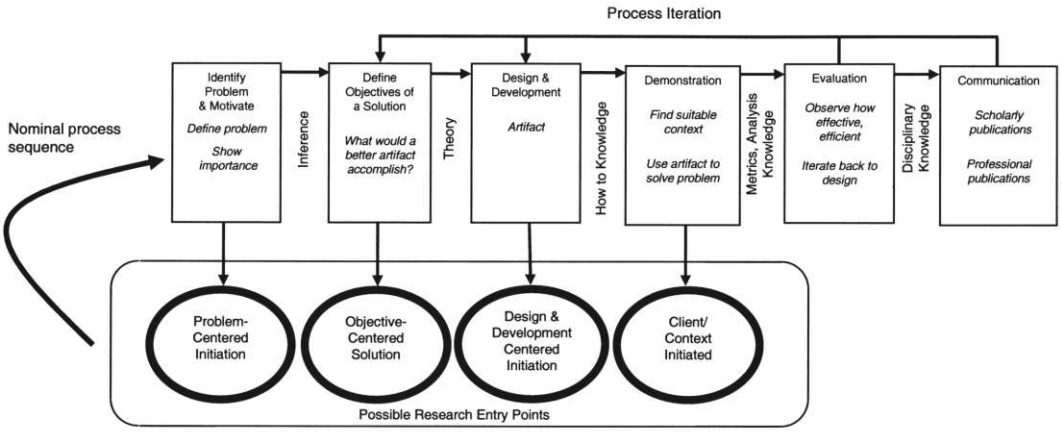


Figure 1.3: The IS design science research procedure, adopted from [20]

As previously mentioned, this view of DSR had a critical downside when used in finding new solutions to new problems: its procedural flow results in projects progressing in a waterfall-like flow, inhibiting experimental design [19, 20].

ADSRM resolves this issue by modifying the traditional workflow of DSR to allow for multiple agile design sprints, borrowing key notions from the practice of agile software development. This allows for a co-evolving problem and solution space, yielding multiple incremental artifacts which can be compared against each other or unified to form aggregate solutions [19]. The ADSRM process is illustrated in Figure 1.4. This figure is an extension of Figure 1.3, wherein the added agile components are drawn in red. These components and the traditional DSR components are elaborated in the following subsections.

1.4.1 Problem Backlog

At the beginning of each iteration, problems are extracted from the study's "problem space" and are added to the list of problems which are to be addressed at some point during the study [19].

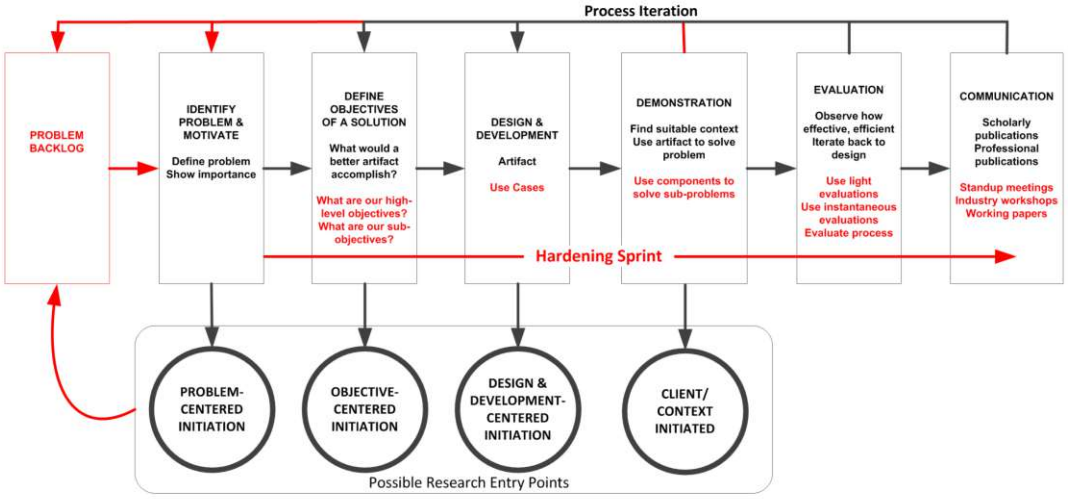


Figure 1.4: The agile design science research procedure, adopted from [19]

The approach to solving each research question evolved over the course of this thesis. Each deliverable stands as a major step towards answering the research questions, solving some sub-questions while raising new ones. Findings collected gave insights that were reflected back into the problem space, possibly shedding light on new sub-questions to be added to the problem backlog. As the project progressed, this cycle of solving sub-questions and finding new ones stabilized. Once there were no further sub-problems in the problem backlog to be solved for a given research question, the research question was considered to be answered.

1.4.2 Problem Identification and Motivation

The first step is to identify the research problem and to establish the motivation of the research, as to understand the relevance of the problem and to set the initial course of the design activities. This is important, as the starting stage of the research process depends on the motivation of the project.

Given that research problem for this thesis arose from suggestion for future research from Krouwel et al. on how to use the MDA approach to transform DEMO models to low-code applications [21], this constitutes *problem-centred initiation* [19]. Therefore, the first step in this thesis project was to identify the problem and motivation (see Figure 1.4).

1.4.3 Solution Objectives Definition

To set the foundation of a potential solution to each iteration's principal problem, the characteristics of a solution to the problem should be defined [19].

In this thesis project, the concrete objectives of the agile design sprints were primarily identified through three stages:

1. A review of theoretical background knowledge (see Chapter 2)
2. A review of the state of the art (see Chapter 3)
3. An investigation into the empirical characteristics of the three MDA abstraction levels found in literature (see Chapter 4)
4. The deductive formulation of a conceptual framework (see Chapter 5)

1.4.4 Design and Development

At the *design and development* step, the artifact of the design sprint is produced. Such artifacts may be any object, such as constructs, models or instantiations, that is designed on the basis of foundational theory and that contributes to solving the research question.

The artifacts produced by each of the design sprints of this thesis are model-to-model transformation mappings. The meta-design derived in Chapter 5, as a compilation of knowledge on enterprise ontology, software architecture, and the MDA framework, guided the design and development of these mappings. The design procedure of each sprint follows the same design model-to-model transformation procedure, based on the procedures proposed in [22, 23].

1.4.5 Demonstration

At the *demonstration* step, the utility of an artifact developed during the previous steps is demonstrated against the current iteration's principal problem. This could be accomplished through the use of experimentation in the context of an evaluation method or a case study. The created artifact from this step will allow for insights to be drawn which can be translated back to the problem backlog to be addressed and used to develop another artifact in a future cycle. In the spirit of agile, ADSRM allows for partial solutions to be demonstrated and evaluated, not just those that are complete [19].

In this thesis, the mappings created during the experimental design sprints – the complete MDA transformation specification and the partial CIM-to-PIM mappings – are demonstrated using an academic case study (see Chapter 6 for the case study description; see Chapters 7, 8, 9, 10, and 11 for the details and results of each sprint).

1.4.6 Evaluation

At the *evaluation* step, the artifact of each design cycle is assessed and measured to determine how well the artifact fulfilled the objectives of a solution to the research problem through its demonstration.

The demonstrated transformation mappings of this thesis were evaluated both quantitatively and qualitatively. To measure the results quantitatively, the semantic completeness and correctness of the transformation is measured [24, 25]. To evaluate the results qualitatively, the strengths and weaknesses of the first three experimental CIM-to-PIM designs are assessed. These are taken as input into the hardening sprint to produce the final CIM-to-PIM design.

1.4.7 Communication

At the *communication* step, the motivation, background, and details of the research study are communicated to stakeholders. Forms of communication can vary over the course of the study and can include presentations, articles, or posters. Feedback gathered from this step may result in new problems added to the problem backlog or modifications made to artifacts.

The thesis proposal, this written report, a final review presentation, and the defence presentation serve as the methods of communicating this thesis project's progression and results.

1.4.8 Hardening Sprint

As agile projects can become quite tumultuous over the course of several iterations, this may come at the cost of lowered rigour. To ensure that the rigour is protected, every few iterations should be structured as a *hardening sprint*. This is possible by employing one of the three following options: freeze the problem, freeze the procedure, or add additional elements of rigour to the procedure [19].

A hardening sprint was undertaken following the first three experimental design sprints of this thesis. After three partial mappings were explored in Chapters 7, 8, and 9, Chapter 10 details a hardening sprint whereby the strongest components of the first three partial mappings were combined to derive a novel UML profile and the CIM-to-PIM mappings of the final MDA transformation artifact of this thesis.

1.5 Structure of the Work

The contents of the remaining chapters of this thesis are briefly summarized below:

Chapter 2 – Theoretical Background

This chapter introduces the fields of model-driven engineering and Model Driven Architecture®, enterprise ontology and DEMO, and low-code software development, discussing relevant advantages and drawbacks of each.

Chapter 3 – State of the Art

A review of the current state of the art is provided to explore the complementary relationships between the three key fields of this thesis: DEMO, model-driven engineering, and low-code development. The chapter concludes by discussing the implications that the explored works have for this thesis.

Chapter 4 – Characteristics of the MDA Abstraction Levels

To answer research question one, this chapter presents a semi-structured literature review, through which existing research relating to the MDA approach was searched to uncover empirical characteristics of CIM, PIM, and PSM models. These characteristics are presented in table-form and working definitions of the three abstraction layers are provided.

Chapter 5 – MDA Transformation Meta-Design

To answer research question two, key findings, theories, and notions from Chapters 2, 3, and 4 are synthesized and presented as a conceptual framework that was used to position enterprise ontology within the context of software architecture by analyzing common semantics. The conceptual framework as a meta-design served to scope and guide the project's experimental design cycles.

Chapter 6 – Experimental Design Preliminaries

To set the stage for the experimental design sprints, the intended artifact is scoped in terms of the conceptual framework presented in Chapter 5, and static aspects of the experiments are introduced. These include the DEMO fact model (FM) modelling concepts, the candidate UML profiles, the Mendix low-code development platform and its metamodel, and the academic case study, "Rent-A-Car", through which the designs were demonstrated and evaluated. The findings of the subsequent agile design sprint (AS) chapters together provide insights that contribute to answering research question three.

Chapter 7 – AS1: DEMO FM to UML

A transformation mapping from the DEMO fact model to Standard UML is presented, demonstrated, and evaluated.

Chapter 8 – AS2: DEMO FM to xUML

A transformation mapping from the DEMO fact model to xUML is presented, demonstrated, and evaluated.

Chapter 9 – AS3: DEMO FM to fUML

A transformation mapping from the DEMO fact model to fUML is presented, demonstrated, and evaluated.

Chapter 10 – HS: DEMO FM to pimUML

A hardening sprint undertaken to synthesize the findings of the previous three agile sprints to formulate a novel UML profile to be used as a PIM modelling language: *pimUML*. A mapping from the DEMO fact model to pimUML is presented, demonstrated, and evaluated.

Chapter 11 – AS4: pimUML to Mendix

A transformation mapping from pimUML to Mendix via pimUML, is presented, demonstrated, and evaluated. The demonstration constitutes the full two-stage MDA transformation, from the DEMO fact model to pimUML and from pimUML to Mendix.

Chapter 12 – Conclusion

To conclude this thesis, the research questions are answered, the deliverables are summarized, contributions of the research to industry and academia are explained, limitations of the study are discussed, and future work is proposed.

CHAPTER 2

Theoretical Background

This chapter elaborates the three fundamental concepts that comprise the theoretical background of this thesis. First, the discipline of model-driven engineering and the Model Driven Architecture (MDA) approach are discussed. Next, enterprise ontology and the related DEMO method are introduced, explaining their core concepts and foundational theories. Finally, the concept of low-code development is introduced, and common features and challenges of low-code development platforms are explained. The chapter concludes with a discussion on how leveraging DEMO, MDA, and low-code together has the potential to realize synergy which can aid enterprises in improving business-IT alignment, which is the motivating goal of this thesis.

2.1 Model-Driven Engineering & Model Driven Architecture®

As software becomes increasingly complex, models play a significant role in ensuring the efficiency and effectiveness of development efforts. Model-driven software engineering – shortened to MDSD or MDE – is a software engineering methodology that leverages the expressive power of models to effectively and efficiently develop enterprise software. Models in MDE may be used for various engineering activities throughout the development process, from serving as a means of communication between stakeholders, to simulating proposed software designs, and even to generating code. MDE therefore addresses the process of software development from two orthogonal dimensions: conceptualization and implementation [10]. The conceptualization of software consists of using conceptual models to represent the problem and solution spaces of the development. The implementation of software is supported by realizing the modelled software solution as executable artifacts. As models are central to both the conceptualization and implementation dimensions, MDE regards models as “first-class citizens” in the software engineering process.

Models in MDE are specified using modelling languages and are expressed using a graphical or textual notation. These modelling languages should be formally defined by models themselves of higher levels of abstraction, known as metamodels. Metamodels define a modelling languages concepts and valid links between concepts.

Different models depict different aspects of a system at different levels abstraction, depending on the model's notation. As some modelling notations are more expressive than others and may be more suitable than others for modelling certain aspects, MDE encourages the use of different views and notations across and within each abstraction level [10, 26]. This principle is known as *multi-view modelling*. Following this principle, MDE approaches can capture both the structural and behavioural dimensions at all relevant levels of abstraction. This has the benefit of comprehensively capturing all aspects of the software's structure and behaviour while separating concerns into different coherent views. Views within the same abstraction levels would be interlinked through commonalities between the views.

Translating models between modelling languages at different levels of abstraction – as well as generating code artifacts from models – is achieved through *model transformations*. By defining mappings between elements of different modelling languages, models can either have their individual views directly transformed into a semantically equivalent view at a different level of abstraction, or models can be weaved together to combine semantics from different views at one abstraction level into a common view at a different abstraction level. These mappings are defined at the metamodel level.

MDE approaches can be implemented and realized by modelling tools. Such tool support increases the efficiency and effectiveness of MDE even more by enabling the automation of tasks such as model development, validation, and simulation, as well as code generation or model execution.

2.1.1 Model Driven Architecture®

Model Driven Architecture® (MDA) is an MDE framework introduced and maintained by the Object Management Group (OMG) [27]. MDA models serve two primary purposes: to foster communication between stakeholders and to increase the efficiency, reliability, and flexibility of enterprise software development, especially when it comes to implementing strategic change. Simultaneously fulfilling these dual purposes requires MDA models to be expressive enough to capture both domain information and precise implementation details while remaining understandable to both technical and non-technical stakeholders alike. To achieve this, MDA follows the principles of *multi-view modelling* and *separation of concerns* to capture different aspects of both the problem and solution spaces from different viewpoints and at different levels of abstraction [10, 27].

The aforementioned principles are applied by modelling the business context and software solution across three levels of abstraction: the *computation-independent model* (CIM), the *platform-independent model* (PIM), and the *platform-specific model* (PSM). Through

a series of model transformations, the high-level, business-oriented CIM models are transformed into software-oriented PIM models, and then further into PSM models which can be either directly executed or used to generate software artifacts. Thus, the development is driven by modelling at all stages, shifting the effort of the development from the solution implementation phase to the problem analysis and solution design phases.

Software development via the MDA approach begins with the development of a CIM model. Often known as the business model or domain model, the CIM model captures information on the context of the enterprise problem space, without making any reference to technologies that may be used to support the enterprise operations [26]. Various types of models may be suitable for the CIM level, such as enterprise architecture models, business process models, or ontological models [27]. These models are typically expressed using enterprise modelling notations and vocabulary that are commonly understood by domain experts and non-technical stakeholders. [10]. The primary responsibility of the CIM is to provide a means of communication and to capture domain entities, processes, rules, and vocabulary [27].

The PIM model is a software-oriented model, modelling key design decisions of software artifacts that are to be implemented to support enterprise operations, while excluding features or constructs of any particular platform. As the PIM is derived from the CIM, the PIM model captures the needs of the enterprise as software requirements and appropriate design specifications. The CIM-to-PIM transformation therefore plays a key role in bridging the semantic gaps between the enterprise engineering domain and the software engineering domain, translating business requirements into software requirements and functional design decisions [10].

The PSM model contains platform-specific constructs, as well as the necessary execution semantics required for implementing and executing the application on a specific platform. Such an implementation can either be realized by generating code from the PSM or by directly executing the PSM itself [10, 27]. As multiple platforms may be required to realize a software system, the PIM-to-PSM transformation may transform a single PIM into multiple PSMs, each for a different platform and with their own respective responsibilities, together forming a cohesive system [27].

There are a few considerations to keep in mind when designing or adopting an MDA approach. First, since platforms can exist at different levels of abstraction, naturally, so do PSMs. As such, the difference in abstraction between the PIM and PSM is relative, varying dependently on the abstraction level of the target platform [27]. Second, each modelled view should be expressed in a notation that is well-defined and commonly understood by the relevant stakeholders at each of the three abstraction levels [10]. This could be achieved by using a well-known enterprise modelling language at the CIM level, such as BPMN, and a well-known software modelling language at the PIM level, such as UML. Third, to benefit most from the MDA approach, transformations should ideally be automated. Although, it is possible for transformations to be conducted manually [27].

The MDA approach can benefit enterprises in maintaining business-IT alignment in three ways. First, MDA encourages the activity of modelling collaboratively. This aids stakeholders, both technical and non-technical, to engage in communication with each other to find a common vision [27]. Moreover, the MDA models themselves are highly valuable artifacts of documentation. Not only do these models comprehensively capture information on the enterprise, software architecture, and software implementation at multiple perspectives and abstraction levels, but since code is generated from them, they remain accurate representations of the actual implemented systems, even as the enterprise and its systems undergo change. Second, modelling the problem and solution spaces at three levels of abstraction assists not only deriving executable artifacts from models, but also in drawing a logical line of reasoning through the related design decisions from the CIM through to the PSM and *vice versa* [10]. For example, implementation decisions at the PSM level can better understood given context provided by the PIM level, and architectural decisions at the PIM level can be understood given context provided by the CIM level. Third, using models to generate executable artifacts saves time and reduces the number of bugs in the code caused by manual coding. This allows software to be modified quicker and more easily, thus increasing the adaptability of enterprise software as the requirements of the enterprise change. Moreover, models can be used for simulation and analytics to evaluate the designs of proposed systems before the implementation phase begins, further reducing project risk [27].

Despite its benefits, there is one prominent drawback of MDA. There is little guidance in the official MDA reference document [27] on what concrete information should be captured and how this information should be structured at each of the MDA abstraction levels. As a result, it is not immediately clear how the MDA approach should be applied in the context of a fundamental part of the development of information systems: software architecture design [28].

2.2 Enterprise Ontology & DEMO

As modern enterprises are growing in complexity, without a clear understanding of how an enterprise works, solving business problems, changing strategy, and reacting to various enterprise phenomena becomes a challenge [29]. Fundamental to understanding observed phenomena is explanatory theory [30, 31]; enterprise ontology fulfills this role in the field of enterprise engineering, with the aim of understanding the nature of enterprises. Enterprise ontology is a scientific discipline grounded in various philosophical and ontological theories, to understand, analyze, and model the construction and operation of enterprises. Enterprise ontology serves to assist managers in gaining an overview of and insight into the complexity of their respective organizations by providing concepts and tools to understand and model the *essence* of enterprises in any domain.

2.2.1 Construction and Operation of Enterprises

To understand enterprise operation, enterprise ontology frames the notion of enterprises as social systems which operate through the cooperation of human actors, constantly engaging in transactions to generate value and complete business processes. These actors, the roles they assume, and the transactions they fulfill constitute the fundamental building blocks of business processes. At the same time, enterprise ontology expresses the construction of enterprises through the notion of facts and how human actors behave and interact in fulfillment of the processes and corresponding acts that result in the creation of facts.

In enterprise ontology, facts are the key concept used for expressing the construction of enterprises. Facts are essentially propositions or statements about an enterprise that are understood to be true [32]. Facts together form the "state of the 'business' world of an enterprise". Concrete objects of unary facts are called entities and abstract objects of unary facts are called values. In enterprise ontology, these concepts are captured as business entities. These include the products of the enterprise, the actors who concern those products, and any other concepts that are relevant to the products and services of the enterprise and any business rules that apply [33].

In the context of enterprises, a *fact* is anything that is considered to be true about the existence of entities and objects within an enterprise. They are therefore used to describe every aspect of the construction of the business, such as business entities, relationships between entities, and business events. They are also the basis upon which business rules are formed and evaluated [34]. Facts are created through the fulfillment of *acts*. The creation of a new fact is called an *event*. As actors work and communicate, they engage in two types of acts, resulting in two corresponding types of events and facts: coordination (C-act, C-event, and C-fact) and production (P-act, P-event, and P-fact).

2.2.2 Transactions

Actors go about doing their work by responding to requests from other actors and completing those tasks in accordance with business rules and laws set by the enterprise. The performance of a C-act by an actor – including creating a request – is known as a C-event. In order to move a transaction along, a C-event requires a response from the other cooperating actor of the transaction. The resulting prompt for a response from a C-event is known as an *agendum*. An actor continuously works by waiting for tasks to complete, selecting an agendum when one arises, choosing an appropriate response in accordance with any applicable business rules, and performing the desired C-act or P-act. This cycle is known as the *actor cycle* (see Figure 2.1).

When actors request tasks or products to be completed by other actors, the sequences of C-acts and P-acts that follow from the request to the completion or rejection of the commitment occur in generic patterns known as *transactions*. The role of the actor making a request is the *initiator* of the transaction, and the role of the actor who is being requested to fulfill a task is the *executor* of the transaction.

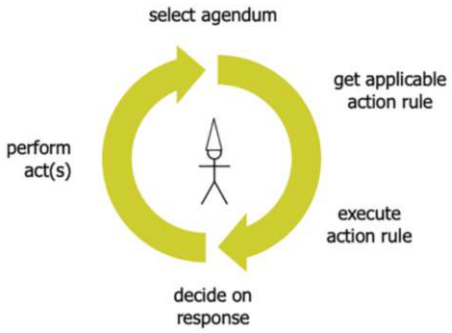


Figure 2.1: The operating cycle of actors, adopted from [33]

There are three phases of a transaction: the order phase, the execution phase, and the result phase. During the order phase, the initiator makes their request to the executor for something to be produced. In other words, the initiator requests a product of the executor – products can either be material or immaterial [33, 34]. If the executor promises to act on the request, the transaction moves into the execution phase, whereby the executor produces the product that was requested. After the product has finished produced, the executor declares that the product is complete and delivers the result to the initiator. If the initiator accepts the result, the transaction has completed successfully (see Figure 2.2).

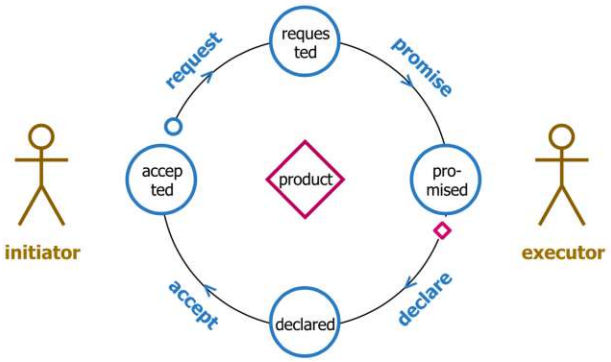


Figure 2.2: The stepwise pattern of transactions, adapted from [33]

New P-facts are only created after the successful execution of a transaction [33]. Every transaction is related to a certain kind of product and a certain actor role who executes that transaction. A product can have multiple different kinds of transactions that concern it; each of which amounts to a change of state of the product. For example, one transaction may result in a certain kind of product being created, while another transaction may result in that same kind of product being destroyed. As such, transactions determine the relevant business events that affect a product, changing its state, over its lifetime.

By applying the principles of automata theory to the cycle in Figure 2.2, the pattern of transactions can be viewed as an automaton, thus yielding the *basic transaction pattern* (see Figure 2.3).

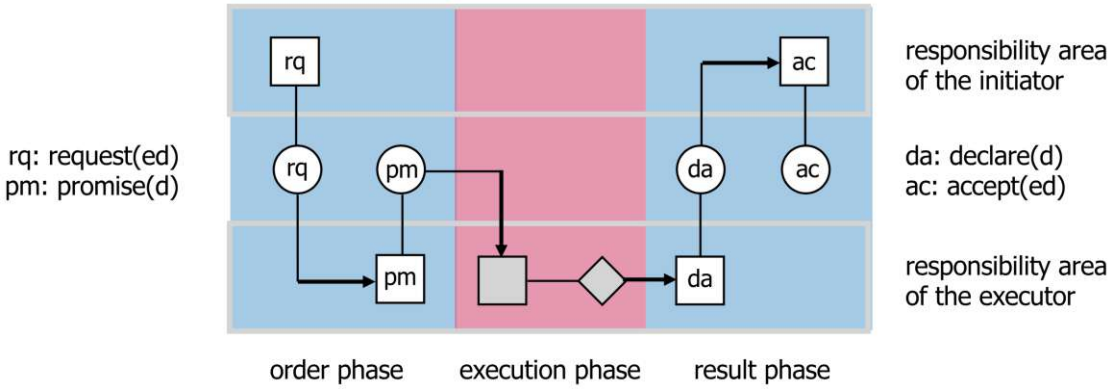


Figure 2.3: The basic transaction pattern, adopted from [33]

The function of an enterprise is known as its *business*. The business is defined by what services are offered to customers of the enterprise. The business of the enterprise is realized by the *organization* of the enterprise. Transaction kinds combined with their associated executor actor roles are known as *transactor roles*. Transactor roles, when organized together into tree structures, form business processes. Transactions are therefore the building blocks of business processes. Together, these tree structures give the total view of the enterprise processes and products. In this way, modelling an enterprise's business transactor roles and applicable business rules which form the business processes to produce the products offered by the enterprise gives a complete view of the organization of the enterprise. Thus, a model that captures this information is known as the essential model of the enterprise [33].

2.2.3 DEMO

The Design and Engineering Methodology for Organizations (DEMO) is an enterprise ontology method for deriving the essential model of any enterprise. As the essential model captures the structure of transactor roles of which business processes are composed, DEMO is effectively a coordination-based business modelling approach [33].

As a method, DEMO is comprised of two core components. The first component – the *Way of Modelling* – is a collection of metamodels and a notation language for expressing ontological enterprise models. The second component – the *Way of Working* – is a method for analyzing an enterprise system or situation to produce ontological enterprise models. Such ontological enterprise models produced with the DEMO method are herein referred to as *DEMO models*.

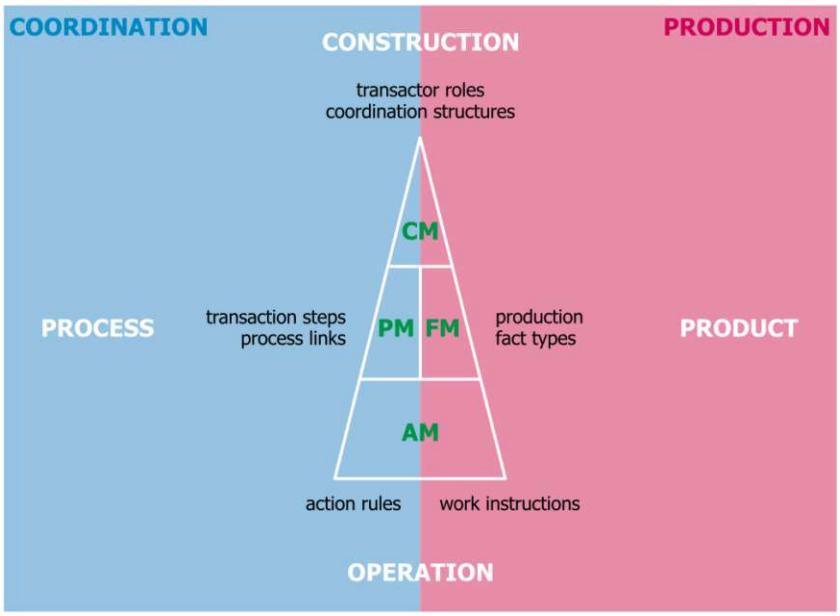


Figure 2.4: The DEMO aspect models, adopted from [33]

DEMO models are composed of four aspect models[1] that each capture a key view of the organization of an enterprise. These models are the *cooperation model*, the *action model*, the *process model*, and the *fact model* (see Figure 2.4). The cooperation model captures the *construction* of the enterprise, illustrating the transactor roles and the relationships between them which form the coordination structure of business processes. The action model captures the *operation* of the enterprise, containing business rules in the form of action rule specifications and work instruction specifications. The process model captures the chains of actor acts and transactions composing the business *processes* of the enterprise, primarily expressed using process structure diagrams. Finally, the fact model captures the *products* of the enterprise, illustrating business entities and the state and transition spaces of these entities. The four DEMO aspect models are expressed graphically and textually using the DEMO Specification Language (DEMO-SL) [35, 33].

While the main benefit of the DEMO method is that it allows for the modelling of the essential model of any enterprise, a key drawback of DEMO models is that they are highly abstract, and as a result, they lack concrete implementation details which would be required to develop software [36]. This would make it a challenge to generate software directly from DEMO models in an MDE approach.

[1] For a detailed explanation of each of these aspect models, the reader is encouraged to refer to [35].

2.3 Low-Code Development

Low-code is a software development approach that allows developers to visually build applications, primarily by using graphical editors with drag-and-drop functionality, with minimal manual coding required. Low-code applications are developed, deployed, and maintained using low-code development platforms (LCDP) [37, 38, 39, 40]. Drawing on principles of MDE, low-code applications are built at a high level of abstraction, while low-level concerns are largely handled by the LCDP, thus requiring minimal development effort. Given the ease-of-use of these platforms, LCDPs are primarily aimed at business users with little to no technical knowledge or coding experience. Such users are known as *citizen developers*. The productivity increases afforded by these platforms make them ideal for many enterprise use cases, such as rapid prototyping or digital transformation [41, 40]. As such, the adoption of low-code development platforms (LCDPs) in industry, as well as research into such platforms in academia, has been growing tremendously in recent years [42, 43].

LCDPs are particularly desirable for enterprises when agility is necessary, business process management is paramount, and the supply of developers is low [38]. As an enterprise of the information age may have an arsenal of enterprise applications to support different functions of day-to-day business operations, LCDPs have become quite popular for building many different applications commonly found as part of modern enterprise information systems. These include e-commerce systems [41], workflow management systems, (WfMS), customer relation management systems (CRM), and business intelligence processes [38, 40].

LCDPs come in a variety of forms, some more tailored for specific use cases and some more applicable for general application development. LCDPs can roughly be distinguished as belonging to one of the following four categories [40]:

- Data management platforms
- Workflow management systems
- Extended graphical user interface (GUI) and data-centric IDEs
- Complex multi-use platforms

Regardless of form, LCDPs tend to share common characteristics and features. LCDPs provide dialogs, graphical editors, and other tools as visual means of developing applications. LCDPs are typically composed of the following components and features [39]:

- Application modeller
  - Domain model, navigation model, etc.
  - Local run environment

  - Configuration of database, deployment, external API and service integration, version control repositories etc.

- Back-end and database server
  - SQL or NoSQL database
  - Analysis and optimizations
  - Hosting (container orchestration)
- Collaborative development support
  - Includes project management tools such as agile, kanban, and scrum

Despite the differences between LCDPs, the process of building applications also tends to follow the same procedure, which of course may be followed iteratively through an agile approach [39]:

1. Data modelling
2. User interface definition
3. Business logic rules and workflows specification
4. External services and API integrations
5. Application deployment

Despite the benefits of efficiency and better business-IT alignment brought by LCDPs, there are still challenges that come with their adoption. The two most significant challenges are the high learning curve of platforms and the risk of vendor lock-in [41, 44, 38]. Despite being targeted towards non-technical citizen developers, learning low-code development can still be a challenge. These users may lack not just software development skills, but also knowledge of important software design principles required to model applications, even following a low-code approach. The risk of vendor lock-in is also a significant challenge with the use of LCDPs. In fact, the risk of vendor lock-in was found to be one of the top reasons why companies choose against using LCDPs [38]. As platforms tend to be closed sourced, some even using proprietary modelling languages, this inhibits the extensibility and interoperability of these platforms. This could become a critical problem in the situation that support for a platform is discontinued, as was the case with Google AppMaker in 2020 [45]. Given these challenges and the risks that they bring to teams considering the adoption of LCDPs, researching ways to combat vendor lock-in and to lower the learning curve of these platforms is imperative [41, 38].

2.4 Conclusions

In light of the challenges of market turbulence and digital disruption that the enterprise of the information age faces, improving enterprise agility while maintaining optimal business-IT alignment are paramount objectives. This chapter introduced model-driven engineering and MDA, enterprise ontology and DEMO, and low-code software development. Each of these methodologies and technologies have their own respective advantages, but as this thesis aims to explore, combining these notions into a novel development approach could create synergy that boosts an enterprise's ability to achieving the aforementioned objectives more effectively and efficiently than the individual tools could on their own.

The notions of enterprise ontology and the DEMO method can assist with gaining a better understanding of the inner workings of the enterprise, thus making it easier to identify how improvements can be made internally in response to external market forces and other enterprise phenomena, improving enterprise agility.

Low-code development can also help to boost enterprise agility, especially as it relates to software adaptability [5, 44]; however, the significant challenges of high-learning curves and risk of vendor lock-in still hinder business users, and low-code technology itself, from fully realizing this potential. While low-code development already draws on principles from MDE, recent research on the challenges of low-code development suggests that expanding the incorporation of MDE principles to develop low-code applications could solve some of these issues.

The MDA framework, under the umbrella of MDE, is a prime candidate to create an approach to integrate ontological DEMO models into the development process of building enterprise applications using low-code technology. The reason for this is twofold. First, MDA can bridge the semantic gaps between enterprise ontology and software engineering, streamlining the application development process while maintaining business-IT alignment. Second, MDA could also alleviate some of the challenges of current low-code platforms. The learning curve of working with LCDPs becomes lowered, as business experts can collaborate by working on the CIM level to capture enterprise concerns using a notation with which they are familiar. Meanwhile, the software experts can work on the software design concerns independent of any target platform, mitigating the risk of vendor lock-in.

To better explore the potential synergies using DEMO, MDE, and low-code together, the next chapter reviews of the state of the art, providing a deeper examination of the existing literature on the interrelationships between the three fields.

CHAPTER 3

State of the Art

This thesis is grounded in the foundational theories of enterprise ontology, practices of model-driven engineering, and approaches of low-code development. Over the past several years, there have been several studies into the interrelationships between these three fields. This chapter discusses the state of the art at the following four intersection points of these three fields:

- Model-Driven Engineering and Low-Code Development
- DEMO and Model-Driven Engineering
- Low-Code Development and DEMO
- DEMO, Model-Driven Engineering, and Low-Code Development

The respective positioning of the papers reviewed at the above listed intersection points is illustrated in Figure 3.1.

To conclude the chapter, the implications that the findings of the reviewed state of the art studies have on this thesis are discussed.

3.1 Model-Driven Engineering & Low-Code Development

The principles behind model-driven engineering (MDE) and low-code development platforms (LCDPs) greatly overlap. However, there do exist differences between MDE approaches and low-code. This has motivated researchers to explore the similarities and differences between MDE and low-code, thus uncovering challenges and opportunities for future research into approaches using MDE to develop applications for LCDPs [42, 46].

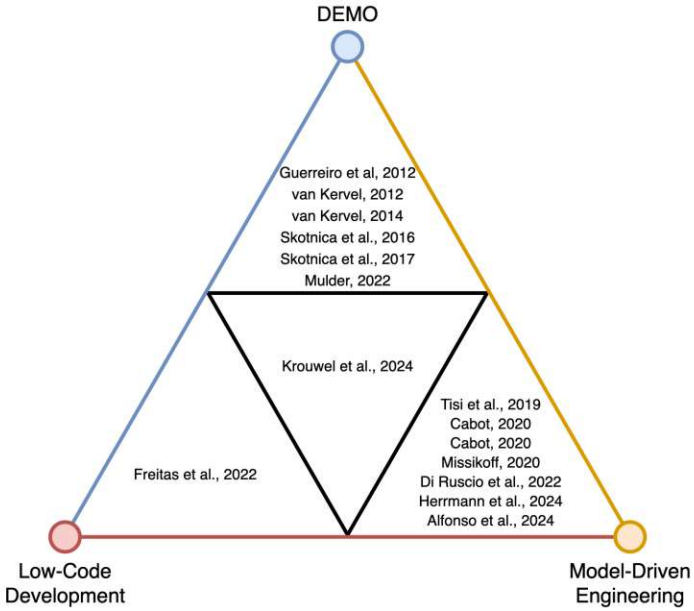


Figure 3.1: Diagram of the intersecting fields of the papers comprising the state of the art of this thesis

Cabot's 2020 assessment [46] of the link between MDE and LCDPs classifies low-code as a restrictive form of MDE; restrictive in the sense that the generated output cannot be accessed nor modified. The relation between MDE and low-code development becomes clear when considering that the generation of executable software artifacts from graphical models is central to both paradigms. However, a key differentiating factor is that MDE approaches often involve multiple intermediate model transformations, whereas low-code uses a single-step model transformation to generate executable artifacts. Moreover, with low-code development, there is no flexibility with the choice of modelling language to be used; they are fixed features of the LCDPs.

A similar assessment [42] by Di Ruscio et al. in 2022 also identifies similarities between the two approaches. The authors found that graphical modelling languages are not only both central to the design and development stages of the two approaches, but both approaches enable stakeholders – even those with minimal programming experience – to contribute to the development of enterprise application by expressing applications graphically. The authors also identify three key aspects which distinguish model-driven approaches from low-code development approaches: platforms, users, and domains.

Having identified similarities and differences between model-driven engineering and low-code development, the authors identify a few challenges and opportunities. A common challenge of using LCDPs is the risk of *vendor lock-in*. This was also found in [44] to be a challenge commonly considered by those using LCDPs in industry. An opportunity to combat this is to leverage existing notions and standards of MDE; for example, by using

standard modelling languages such as BPMN and UML or meta-modelling languages such as MOF or Ecore [42].

In [47], Tisi et al. present Lowcomote - an innovative training network (ITN) with the aim of enhancing the capabilities of LCDPs through the integration of MDE, as well as machine learning and cloud computing technologies, to evolve low-code *development* platforms into low-code *engineering* platforms. The role of MDE in this project is to solve three prominent issues of LCDPs: scalability, fragmentation, and homogeneity. Scalability refers to the difficulty of managing large low-code models of complex and mission-critical applications; especially when it comes to visualizing them graphically. Fragmentation refers to how different LCDP vendors largely use proprietary modelling languages to build low-code models; thus, hindering interoperability and promoting vendor lock-in. Finally, homogeneity refers to the restrictiveness caused by LCDPs largely using modelling languages that originate from the software engineering discipline and not from the enterprise engineering discipline. The authors postulate that principles and approaches of MDE can alleviate the problems of fragmentation and homogeneity. For example, fragmentation can be solved by facilitating the use of open standard modelling languages. Homogeneity of models can be reduced by introducing multi-view modelling to LCDPs.

The approach presented by Hermann et al. in 2024 [48] builds upon the notion of multi-view modelling, proposing an integrated ecosystem consisting of LCDPs, MDE tools and even high-code development environments. This is powered by an amalgamation of multiple metamodels known as a *virtual single underlying model* (V-SUM), also known as a *pragmatic SUM*. The V-SUM is used to "project" the information contained in the V-SUM to generate artifacts – or views – of the various platforms and tools involved in the ecosystem. Low-code artifacts thus become one view in the multi-view system. Consistency is automatically maintained between the low-code artifacts and the other views of the ecosystem. A key benefit of this approach is that stakeholders of varying levels of technical backgrounds can contribute to developing an application using a platform with which they are most comfortable.

Creating a V-SUM requires establishing mapping rules – which the authors call *consistency preservation rules* – between the metamodels of the LCDP, the source code, and any intermediate models. This was demonstrated with a case study in which a V-SUM was created by combining the metamodels of UML, Java, and a mock LCDP. The V-SUM integration strategy used was to include the LCDP metamodel as part of the V-SUM and map this metamodel with the UML metamodel. The UML metamodel was then mapped to the Java metamodel; thus, consistency between the low-code model and the source code was maintained through an intermediate UML model. Although the case study demonstrated promise, a key challenge in realizing the V-SUM approach is that including a LCDP as part of a V-SUM ecosystem requires a LCDP to offer features that allow for the export and import of the underlying application model in order for the platform to commit and fetch application changes with the V-SUM.

In [49], Missikoff presents *EasInnova*, a model-driven engineering approach for enterprise innovation – specifically, process innovation. Process innovation requires new or adapted enterprise application support in order to maintain business-IT alignment as the enterprise transforms. This approach involves modelling the current state of the enterprise, known as an *AsIs* model. Through problem analysis, business experts identify any problems that the enterprise is facing. The transformed enterprise with proposed solutions to these problems is modelled as a *ToBe* model. The gap between the AsIs and ToBe models is bridged using a *Transformation* model to capture the necessary changes. These three models are each expressed at the three abstraction levels of the MDA framework: CIM, PIM, and PSM (see Section 2.1.1).

Expressing the models at different levels of abstraction facilitates the engagement and input of business experts together with software developers. The CIM level captures the enterprise ontology through a pattern known as the OPAAL scheme, which captures information about the objects, processes, and actors, as well as any attributes of and links between these entities. The PIM is expressed using BPMN diagrams, UML class diagrams, and UML use case diagrams. The PSM-AsIs is primarily focused on the data model of the current information system and how to migrate it to the proposed application. To automatically generate software to support the transformed enterprise, the PSM-ToBe is expressed using low-code models. However, no automatic transformations between models are proposed.

In [50, 51] Alfonso, Cabot, et al. introduce *BESSER*, an open-source low-code, low-modelling development platform. The goal of BESSER is to build enterprise applications that integrate smart systems technologies while easing the process of application modelling by addressing challenges currently associated with low-code development. This is achieved by following the principles of a concept called *low-modelling*.

Low-modelling eases the processes of three phases of software modelling: model generation, model enrichment, and model inference. By using heuristics, leveraging domain knowledge from existing sources such as taxonomies or ontologies, and using machine learning to extract model content from documents or images, the effort required to model complex software can be reduced [52]. For example, heuristics can be used to infer the CRUD operations of domain entities represented in structural models. As Cabot explains,

> "The key idea is that any data model will require a number of basic CRUD (create/read/update/delete) operations to visualize and manipulate the data specified in the model... operations can be deduced from an analysis of the static model elements and relationships by systematically applying a number of heuristics."

BESSER applications are built using *BESSER's Universal Modeling Language* (B-UML) [50, 53, 51]. B-UML models are used to generate Python code via model-to-text transformations. The Python code generated targets various platform tools to implement

application components, such as Django or SQLAlchemy. B-UML is heavily based on UML; however, taking a pragmatic approach, the language does not fully conform with the UML metamodel, as it deviates from the UML specification where necessary, omitting, modifying, or adding modelling concepts where necessary to improve modelling efficiency and ease-of-use. The current version of B-UML (version 2.5.1) [53] consists of eight sub-models, which may either be used individually or together to specify constructs as the modeller needs. Each model serves a different purpose in defining various aspects of enterprise applications, such as defining a domain model, specifying both its structure and runtime constraints; defining a GUI model, specifying the application's GUI components and how they integrate with the domain model to enable the creation, modification, and deletion of domain model entity instances; defining a state machine model, specifying the runtime behaviours of the application; and so on.

3.2 DEMO & Model-Driven Engineering

There have been many recent studies on the generation of enterprise information systems (EIS) through the application of model-driven engineering approaches incorporating the theories and constructs of enterprise ontology and DEMO. These include the direct execution of DEMO models and the translation of DEMO models into code or into models of different modelling languages.

In his 2022 PhD thesis [54], Mulder identified a weakness in the DEMO method in that it lacked the necessary constructs and methods to support the automatic verification and exchange of DEMO models. The main reason for this is that there existed no complete metamodel of DEMO – something that is imperative to facilitate MDE tasks, such as model-to-model transformations. Therefore, the solution was to derive a new and complete DEMO metamodel.

The metamodel proposed by Mulder is comprised of five partial metamodels: an ontological metamodel, a verification metamodel, a visualization metamodel, a data exchange metamodel, and a visualization exchange metamodel. The ontological metamodel is a high-level refinement of the underlying metamodel of the four DEMO sub-models. The verification metamodel contains rules to verify a model's adherence to data rules, mathematical rules, and other model restrictions. The visualization metamodel provides the rules for the graphical representation of the DEMO models. Finally, the exchange metamodels, based on XML Schema Definition (XSD), facilitate the electronic storage and exchange of DEMO models and are necessary for the automatic transformation of DEMO models into models of complementary modelling languages, such as BPMN or ArchiMate. The proposed DEMO metamodel was implemented and demonstrated using the modelling tool, Sparx Enterprise Architect (SEA). The solution was evaluated against several case studies and uncovered a number of possible improvements to the DEMO method in the process.

In his 2012 PhD thesis [55], van Kervel presents an ontology-based model-driven engineering approach to generating an enterprise information system that directly executes the four DEMO aspect models as source code input. This approach has the benefit of effectively skipping the software engineering phase of implementation design, thus eliminating the chance of introducing errors into the system through transforming an ontological model to an implementation model. Another key benefit of this approach is that the ontological models can be simulated and validated before application deployment. The authors postulate that conducting such validation earlier in the development process yields a better degree of business-IT alignment.

This approach is realized by directly executing DEMO enterprise models through a computation construct known as the *DEMO processor*. The DEMO processor works by serializing the four DEMO aspect models in an XML-based language called XML DEMO Modelling Language (DMOL). There is a one-to-one mapping between the four DEMO aspect models and DMOL; therefore, this transformation does not suffer from information loss. The DEMO processor then instantiates and executes these models. At the core of the DEMO processor is the logical axioms of the enterprise ontology PSI theory, which is key to the enforcement of model compliance. This is achieved by interpreting the action rules expressed in the action model to determine allowed versus disallowed actor actions at runtime. The applications that the DEMO processor generates therefore run as workflow systems.

In [56] Guerreiro et al. leverage the DEMO processor's ability to enforce business rules to generate an enterprise dynamic control system (EDCS). The DEMO processor is first provided with a complete DEMO model consisting of the four DEMO aspect models. Once validated, these models constitute the business transaction model of the enterprise. An enterprise information system (EIS) is then generated from this business transaction model. As enterprise actors use the EIS, the state of the application at any given time constitutes a real-time instance of the business transaction model. This model instance can then be validated against the original business transaction model, and action can be taken if any deviations from valid business transactions occur, as to ensure enterprise governance.

The DEMO processor was also demonstrated in a real-world use case in [57] in which it was used to develop a case management system for a Dutch utility company. By applying the discipline of enterprise ontology to the development of an IT artifact, the majority of the design decisions were made during the modelling phase, rather than the software development phase, allowing the process of building the application to be much more grounded in sound engineering principles. This also allowed stakeholders to actively participate in the development of the system such that the system was built according to the way client does their work, which also helped to maintain business-IT alignment.

In [58] Skotnica et al. also present a theoretical automaton, known as the *DEMO Machine*, that directly executes DEMO models expressed in the DEMO Specification Language (DEMO-SL). As the primary purpose of DEMO-SL for the interpretation of DEMO models by humans, it lacks formal execution semantics which are required by machines for model simulation and validation purposes. As such, the aim of this paper was to define formal execution semantics for DEMO-SL in the form of formal mathematical axioms. This paper defines three sets of formal axioms which can be interpreted by the theoretical automaton: fact axioms, agenda axioms, and rules and dependencies axioms. These axioms are together known as the FAR ontology. As a formal computation model, the DEMO Machine is based purely on mathematical constructs and is independent of any kind of software implementation.

The DEMO Machine builds on the ideas of van Kervel's DEMO processor [55] such that the DEMO Machine is also based on the axioms and notions of the PSI, FI, and TAO theories. The DEMO Machine is also based on fundamental IS and MDE theories, such as the normalized systems theory and the *Generic Systems Development Process for Model Driven Engineering* (GSDP-MDE). By incorporating the principles of the FAR ontology in accordance with the applicable software engineering theories, better support is added for interpreting fact and rule expressions of which business rules are composed in the DEMO action model.

In a follow-up paper [59], the authors build on the foundational axioms for the fact and action models derived in their previous paper by adding formal axioms and constructs which formalize the foundational concepts of the structure of the DEMO Machine itself, as well as of DEMO models themselves, such as actors and transactions. In addition, an algorithm is presented for calculating the agenda of an actor at any given instance during runtime.

3.3 DEMO & Low-Code Development

In [60], published in 2022, Freitas et al. present *DISME*, a DEMO-based low-code development platform. DISME allows users to build applications using the DEMO method by providing a system modeller, through which DEMO models are created and edited graphically using a diagram editor and refined using tables or forms. The platform also provides a form editor for designing forms which are used to create and modify entity instances. All of an application's data is stored in a database, which is structured based on the DEMO fact-based modelling approach. Users specify program logic through the definition of action rules, also by using a graphical editor. To run the modelled applications, DISME provides an execution engine which interprets the models in real time, allowing the application to dynamically to respond to any runtime changes to the underlying DEMO model.

3.4 DEMO, Low-Code, & Model-Driven Engineering

This thesis was motivated by and primarily builds on the work of Krouwel et al. in their 2024 paper on generating low-code application models from DEMO models via a direct MDE mapping approach [5]. The paper identifies *enterprise agility* as a key success factor for the modern enterprise. As modern enterprise operations are heavily reliant on software, software adaptability goes hand-in-hand with enterprise agility. It is therefore imperative that enterprises ensure that their software is highly adaptable to rapidly changing business requirements so that software can be in continuous alignment with the enterprise as it undergoes change. As a solution to maintaining this alignment, the authors propose the use of MDE as an approach to generate software from enterprise models.

In their approach, DEMO was the modelling language of choice for modelling the enterprise, as DEMO comprehensively and concisely captures the essence of the enterprise. As the ontology of an enterprise does not change often, enterprise ontology models provide a solid foundation from which enterprise information systems can be generated to support enterprise operations. The four DEMO sub-models are transformed directly into low-code application artifacts – specifically, Mendix application artifacts. These artifacts consist of a domain model for storing business entities, microflows for processing business rules, and pages for creating and modifying the business entities via the graphical user interface.

The proposed approach was demonstrated through a case study on a Dutch social housing program. The business entities of this domain included program registrations and persons involved in the registrations. These entities were captured using the DEMO fact model, while the business rules themselves, pertaining to who may register and when they may register into the program, were captured through the DEMO action model. The demonstration showed that DEMO models in combination with organization implementation variables (OIVs) can indeed form a semantically rich basis for generating enterprise applications via MDE. The authors also found that Mendix artifacts fit well as a target models, as they can be rapidly and reliably modified in response to modified DEMO models, thus increasing the adaptability of software artifacts.

The authors provided a few key areas which could be further explored to improve on their proposed approach. One limitation they encountered was difficulty in comparing their DEMO-based MDE approach against other MDE approaches using different source languages, noting that having a framework that could be used for this purpose would be beneficial. Another possible improvement could be realized by employing the MDA approach to MDE, rather than a direct mapping approach. This approach could increase the versatility and flexibility of the mapping by reducing the semantic gap between the target software artifacts and the models from which they were generated. However, the authors suggested it would first have to be determined how low-code models would fit within the MDA framework – either as a PSM or as output generated from a PSM.

3.5 Conclusions

There have been many studies in recent years on model-driven engineering, low-code development, and enterprise ontology. Those that are the most relevant to this thesis are at the intersections of these three disciplines, from which many key insights can be drawn from the strengths and weaknesses of those studies.

The similarities and differences between low-code development and model-driven engineering highlights opportunities to realize benefits by further integrating the two practices. For example, as suggested in [41], a possible way to combat LCDP vendor lock-in is by introducing standard modelling languages into MDE approaches targeting low-code development platforms. Using the MDA framework allows for this possibility, especially given its use of different abstraction levels.

The application of the theories of enterprise ontology to model-driven engineering and low-code development approaches has also been explored, demonstrating existing interest in using DEMO as a source model in MDE approaches. Many studies focus on achieving the direct execution of DEMO models; however there has not been any studies found which employ DEMO models in an MDA approach. A key challenge highlighted in the studies reviewed that employ DEMO in MDE approaches that needs to be taken into consideration is that DEMO on its own lacks formal execution semantics. While previous studies have worked to add such semantics to DEMO itself, another possible way to achieve this is by using the MDA approach, whereby capturing the execution semantics is the responsibility of models at the PIM and PSM levels.

Finally, Krouwel et al. present a DEMO model transformation directly to Mendix low-code artifacts, constituting an approach leveraging all three fields of enterprise ontology, low-code, and model-driven engineering. The findings of this study support the notion of using DEMO models as a source and Mendix low-code models as a target in an MDE approach. However, the authors suggested that using the MDA framework to add an additional abstraction level could increase the flexibility and value of such an approach.

To summarize, the key insights from the papers reviewed on the state of the art reveal opportunities and challenges which support the use of MDA in an approach to generate low-code artifacts from DEMO enterprise models. With regards to the use of low-code development platforms, the benefits of such an approach could contribute to combating vendor lock-in and lowering the learning curve of using such platforms, further incentivizing the adoption of such platforms. With regards to business-IT alignment, the use of DEMO enterprise models could help to ensure that the software generated meets the requirements of the enterprise and its business processes. Finally, the use of the MDA approach could also increase the flexibility of development pipelines, allowing for different platforms to be targeted from a single platform-independent model.

To better assess the potential fit of enterprise ontology and low-code development within the MDA framework, the next chapter aims to answer RQ1 by exploring the empirical characteristics and purposes of models of each of the three MDA abstraction levels.

CHAPTER 4

Characteristics of MDA Abstraction Levels

To better understand the applicability of the MDA approach to generating low-code applications from DEMO models, it was first important to precisely establish definitions of the three abstraction levels – CIM, PIM, and PIM – based on theoretical literature and empirical studies. To achieve this, a semi-systematic literature review was conducted to search for existing literature in which the MDA approach was applied, taking note of how the authors defined CIMs, PIMs, and PSMs and which languages were used. The findings of the literature review were compiled into comprehensive descriptions of each of these three abstraction levels, both in tabular form and in prose. Most consequentially, models at CIM, PIM, and PSM levels were found to differ among the following key dimensions: concerns, views, common languages, and users. The findings of this chapter served as an initial answer to RQ1 and assisted in better understanding the research problem, as well as in setting the objectives of this thesis.

4.1 Semi-Structured Literature Review

To answer research question one, a semi-systematic literature review was followed – the procedure of which is prescribed in [61]. Through this literature review, previous studies that have dealt with the MDA approach were examined. The similarities and differences of CIMs, PIMs, and PSMs were qualitatively analyzed and the choices of modelling languages for each abstraction level were recorded. Establishing a thorough understanding of the MDA approach by deriving clear definitions of what constitutes a CIM *versus* a PIM *versus* a PSM helped to best position DEMO and Mendix within the MDA framework and to further scope this research. This deliverable therefore reports on the distinct common characteristics of CIMs, PIMs, and PSMs, respectively.

The literature review began by first reviewing a batch of papers pre-identified as being relevant to this study which were supplied by the author's advisor. The details of the search protocol of the semi-systematic literature review (review protocol, search strategy, study selection criteria, study selection procedures, quality assessment checklist, and query results) can be found in Appendix A.

4.1.1 Included Papers

The Table 4.1 lists the papers that were included as part of the literature review. The list includes papers which were pre-identified as relevant by the author's advisors in addition to papers found through conducting the semi-structured literature review procedure.

4.2 Results

The semi-systematic literature procedure was executed over several iterations. As the review progressed, the similarities and differences between CIMs, PIMs, and PSMs began to emerge. The most prominent resulting characteristics can be grouped according to the following four dimensions: concerns, views, common languages, and users. The results of the literature review are summarized in Table 4.2 and working definitions for CIM, PIM, and PIM were derived and are presented in the following subsections.

4.2.1 Working Definition of Computation-Independent Model

The results from the literature review show that a computation-independent model (CIM) is a high-abstraction level model used to illustrate various aspects of the *system*[1] of business operations. As such, models at the CIM level capture information required to understand how a business operates, including domain-specific vocabulary, entities, processes, transactions, communication, actors. The structural aspects can be prescriptive, such as requirements or constraints, as well as descriptive, such as business entities and processes, including their inter-dependencies, and hierarchies. The behavioural aspects are typically captured as business processes and are constrained using business rules. These models are used primarily by business analysts or enterprise architects [62, 15, 63]. As such, they tend to model the functional aspects of the enterprise with as little specification of technology in use as possible [15]. Additionally, CIM models are typically illustrated using modelling languages that are commonly understood by business experts [63]. BPMN is one such modelling language that was used at the CIM level in many of the studies reviewed.

[1]The word "system" in this context refers to the social-technical system of an enterprise.

ID	Title	Year	Citation
P1	The Fast Guide to Model Driven Architecture: The Basics of Model Driven Architecture	2006	[9]
P2	Transformation From CIM to PIM: a Systematic Mapping	2022	[16]
P3	Transformation from CIM to PIM: A Feature-Oriented Component-Based Approach	2005	[65]
P4	Conceptual Modeling of Multimedia Search Applications using Rich Process Models	2009	[64]
P5	The Impact of the Computational Independent Model for Enterprise Information System Development	2010	[66]
P6	Automate Model Transformation From CIM to PIM up to PSM in Model-Driven Architecture	2019	[62]
P7	An Approach for Transforming CIM to PIM up To PSM in MDA	2020	[67]
P8	Model Transformation with ATL into MDA from CIM to PIM Structured through MVC	2016	[68]
P9	An MDA approach to business process model transformations	2010	[15]
P10	Transformation From CIM to PIM Using Patterns and Archetypes	2008	[69]
P11	Applying CIM-to-PIM model transformations for the service-oriented development of information systems	2011	[63]
P12	Code generation using model driven architecture: A systematic mapping study	2020	[70]
P13	A model transformation in MDA from CIM to PIM represented by web models through SoaML and IFML	2016	[71]
P14	Disciplined approach for transformation CIM to PIM in MDA	2015	[72]
P15	An Approach for MDA Model Transformation Based on JEE Platform	2008	[73]
P16	Mapping Approach for Model Transformation of MDA Based on XMI/XML Platform	2009	[74]
P17	Mapping approach for model transformation of MDA based on xUML	2009	[75]
P18	A methodology for transforming CIM to PIM through UML: From business view to information system view	2015	[76]
P19	Transformation approach CIM to PIM: from business processes models to state machine and package models	2015	[77]
P20	A set of QVT relations to transform PIM to PSM in the Design of Secure Data Warehouses	2007	[78]

Table 4.1: Papers reviewed through the semi-structured literature review

	CIM	PIM	PSM
Concerns	Business process P2, P4, P6, P10, P14 Business actors P11 Domain specific vocabulary P2, P4 Process requirements P4 Activity types P4, P10 Objects P10 Input/output parameters P4 System requirements P2, P3, P10 Tasks P11 Constraints P3	Architecture & programming paradigm P3, P4 Process advancement logic P4 Functionality, classes & relationships, packages P7, P14 System states P7, P14, P19 Component construction P3	Code P4 How a system uses a platform P2 Platform-specific constructs P6 Database construction P15 User interface (view, content, & navigation) P7, P15
Views	Domain view P2, P4 Process view P2, P4,P10,P11 Requirements view P2, P3, P10 Value view P11	Process metadata & application model P4 Conceptual Model & Analysis Model P6 Functional View, Dynamic View, & Static View P7,P10,P14	Web service interface model P11 Extended service composition model P11 Application business logic model P11 Code model P8
Common Languages	BPMN P2,P4,P6,P7,P8,P11,P14,P19 UML P2,P3,P4,P10,P12,P13,P14,P18,P19 e^3 value P11	UML P2,P3,P6,P7,P8,P9,P10,P12,P14,P16 xUML P17 WebML P4 SysML P12 SoaML P13 IFML P13 Domain-specific UML profile P20	UML P12 IFML P7 SysML P12 DSLs P11, P12 Programming languages P4
Users	Business experts/analysts P6, P9, P11 Developers P9	Information system analysts and designers P6,P11 Software architects P11 Service modellers P11	Software developers P6,P11 Service developers P11 System & DB administrators P11

Table 4.2: Empirical characteristics of the three abstraction levels of MDA as found in literature

4.2.2 Working Definition of Platform-Independent Model

Platform-independent models (PIM) were found to describe the computational concerns of applications, primarily by capturing architectural and functional aspects of software systems. The structure of PIMs may vary depending on the target technology of the MDA model. As such, it is often the case that the views of PIMs are structured according to an appropriate architectural pattern and programming paradigm. Not only do the contents of PIMs consist of the structural and behavioural domain semantics received from the CIM, but also design decisions that are required in order to realize the solution in an execution environment. Static domain constructs are typically captured at the PIM using a conceptual model, also known as a domain model. Behavioural domain semantics are typically captured in process models, but with added constructs to explicitly define how the processes are to be performed when automated, both by humans and by machines. For tasks that are performed by humans or require human input, such as manual exception handling, user interfaces can be modelled which capture the human input required to carry out such tasks [64].

PIMs express information typically using notations which are commonly understood by software architects, such as UML [63]. Users of PIMs are primarily those who work with building enterprise applications, typically software architects and system analysts [62, 63]; however, they can also be used as a means of discourse together with non-technical stakeholders, especially when examined in conjunction with CIM models. A key implication of this is that in order to be useful, a PIM must capture information and present it in such a way that it is understandable to the users of the PIM, while also capturing sufficiently explicit execution semantics such that platform-specific design decisions can be easily inferred based on the PIM, ensuring an accurate generation of the PSM.

4.2.3 Working Definition of Platform-Specific Model

Platform-specific models (PSM) are the most interesting of the three levels of abstraction, as PSMs may take many different forms depending on the desired target technology. Moreover, PSMs may concern various application aspects, from database construction to user interface design. Some studies considered graphical modelling languages to be PSMs while others considered code to be PSMs. This is because according to the MDA specification, the abstraction levels of PIMs and PSMs are *relative* [27]. In other words, the extent to which a PIM is abstracted from concrete execution instructions depends on how abstract the PSM language is. For example, if a PSM is expressed as code itself, such as C++, versus if a PSM is expressed using low-code models. Given that PSMs must capture all necessary execution semantics for implementation on a specific platform, graphical PSM models could either be used to generate code or the PSM models could even be executed themselves. Given their technical nature and detailed implementation information, PSMs are typically used by software developers and system administrators. Given that low-code applications are simultaneously both graphical models and code, this means that they fit the profile of PSMs.

4.3 Conclusions

The semi-structured literature review searching for empirical characteristics of CIMs, PIMs, and PSMs laid the foundation for the remaining chapters of this thesis in two ways. First, understanding the characteristics of CIMs and PSMs allowed for the assessment of the suitability of applying the MDA approach to solving the research question. Candidate modelling languages could also be assessed according to the characteristic dimensions as outlined in Table 4.2 to determine how suitable they are for use at any of the three abstraction levels of MDA. Second, the identified characteristics of PIMs could be used to define the objectives, as well as to guide design decisions, of the PIM to be developed in this thesis.

An important characteristic of PIM models specifically is the *dual purpose* these models fulfill in MDA approaches. On the one hand, PIMs serve as a means to facilitate *stakeholder communication*, among both technical and non-technical stakeholders. On the other hand, PIMs are used for *code generation*, and as such, they must be capable of capturing functional design decisions that are critical to implementing software to support business processes. As such, the CIM-to-PIM transformation has the responsibility of inferring functional design decisions based on the domain information captured by the CIM. However, several studies emphasize or make it evident that creating a CIM-to-PIM mapping can be expected to a challenge [65, 63, 62, 16, 66], as the nature of CIM models focusing on business concerns, versus PIM and PSM models focusing on software concerns, means that there is a larger the semantic gap to be bridged by the CIM-to-PIM transformation than by the PIM-to-PSM transformation.

The next chapter builds on the results of the semi-systematic literature review by defining a high-level conceptual framework as a meta-design that outlines the required information and structure of an MDA transformation. This was used to assess the suitability of DEMO as a PIM and Mendix as a PSM and to define the objectives of the MDA solution to effectively bridge the semantic gaps that exist between DEMO and Mendix.

CHAPTER 5

MDA Transformation Meta-Design

Enterprise applications are complex in nature, being comprised of an orchestra of technologies, such as databases, messaging systems, scripts, services, and user interfaces, working together to perform tasks for users. As such, the scope of the problem of generating software artifacts from enterprise models is massive. The previous chapter echoes this complexity, having identified many common characteristics of MDA models and revealing that each abstraction level pertains to its own range of concerns, views, modelling languages, and users. Therefore, building on the results of the previous chapter, the next step was to reduce the complexity to allow for a better understanding of the research problem and to identify a structure of the MDA design decisions so that the objectives and scope of a solution could be accurately set. This was achieved by applying the principle of *separation of concerns* [79, 80, 81] to formulate a conceptual framework combining and structuring the notions of foundational theory and practice to better understand the interrelationships between these notions in the context of the MDA framework and to serve as a meta-design to guide the design and development sprints of this thesis.

To effectively reduce the complexity of building enterprise applications, each of the three abstraction levels of the MDA approach must capture the right information, structured in a logical way, so that the details of the enterprise and software can be analyzed by humans and so that software can be generated rapidly and platforms can be targeted interchangeably. Software architecture patterns are often applied to MDA approaches to achieve this [27, 28]. This was exhibited in many of the studies reviewed in the previous chapter; in such studies, the PIM and PSM levels were each comprised of multiple views concerning different aspects of the application architecture. A challenge of this is choosing an architectural pattern that is capable of applying a sufficient degree of

separation of concerns without enforcing the use of an architecture style that may not be implementable with all target platforms.

Another key insight from the literature review in the previous chapter is that formulating a transformation to generate a PIM from a CIM is not trivial. As CIM models were found to be focused on business constructs and PIM models were found to be focused on software constructs, this means that the CIM-to-PIM transformation is responsible to bridge semantic gap between the enterprise and software worlds. In order for an MDA model to be used to generate an application, it needs to capture details of the fundamental aspects of the desired software system based on information from the enterprise domain.

Given these challenges, before an MDA transformation could be designed, it had to be clearly established what information is capable of being extracted from the CIM level to automatically make design decisions at the PIM level. Similarly, in order for the transformation meta-design to be generalizable – ensuring that a conforming PIM would be truly platform-independent –, it was important that the meta-design be formulated based on the analysis of appropriate information systems theories, rather than on the analysis of a particular information systems platform. To address this challenge, the fundamental notions of *conceptual schema-centric development* were leveraged to understand how constructs of a domain can relate to constructs of a software system.

5.1 Formulation of a Conceptual Framework

The meta-design was formulated following the procedure presented in [82] for constructing a conceptual framework to guide information systems research. A conceptual framework is typically presented as a diagram that illustrates how the concepts of the applicable theories or frameworks are interconnected in the context of the research problem, thus providing a clear theoretical basis upon which the objectives of the research are built.

This procedure consists of the following steps to understand the research problem, identify the applicable background theories, determine how they relate to each other in the context of the research problem, and to synthesize the conceptual framework unifying those theories and frameworks.

Step 1. Conceptualize a topic

Step 2. Identify problem statement

Step 3. Read and develop literature review

Step 4. Identify objectives/questions

Step 5. Explore IS theory(ies) or frameworks

Step 6. Select suitable theory(ies) or framework/s

Step 7. Modify

Steps one to three of the above procedure are fulfilled through the elaborations provided in Chapter 1. This chapter therefore begins with step four, outlining the objectives of the conceptual framework. These correspond to sub-questions of RQ2.

A generalizable MDA transformation meta-design should achieve the following two objectives:

- Comprehensively bridge the gap between enterprise engineering and enterprise software engineering
- Inform the choice of one or a set of modelling languages for use at each abstraction level

The resulting conceptual framework should assist in answering the questions:

- What design decisions (e.g. programming choices) required by the PIM can be made based on information from the CIM?
- Which design decisions can be extracted from models at higher abstraction levels and which can be inferred?
- How can the design decisions captured at each abstraction level be structured?
- Can DEMO fit as a stand-alone CIM modelling language?
- Which modelling language(s) should be used at the PIM?

To fulfill steps five and six of the procedure, drawing on the background fields discussed in Chapter 2 and key characteristics of MDA approaches found in Chapter 4, the conceptual framework leverages the following theories and practices which were selected as foundational to solving the research problem of this thesis:

- Enterprise Ontology
- Model Driven Architecture
- Conceptual Schema-Centric Development
- Enterprise Application Architecture

Step seven of the procedure is to modify the existing frameworks to fit the objectives and context of the research problem. To achieve the objectives listed above in a scientifically rigorous way, the theories and practices from the knowledge base were applied in a top-down approach starting by broadly addressing the overarching research goal of conceptual schema-centric development and then putting it into the context of the three abstraction levels of the MDA framework. Principles of enterprise application architecture were then added in to break down each abstraction level into coherent architectural layers. Finally, the principles of enterprise ontology were mapped into this framework.

5.2 Conceptual Framework

The conceptual framework was developed following a top-down approach, whereby the application of a broad theory to solve the research problem was bolstered by applicable, specific theories. First, a broad theory providing principles of generating application artifacts from enterprise knowledge was applied, known as *conceptual schema-centric development* (CSCD). An elaboration of CSCD applied to the context of MDA provided high-level requirements and principles to achieve CSCD by following MDA.

5.2.1 Conceptual Schema-Centric Development

The goal of automatically generating information systems (IS) software from enterprise engineering models has existed since the advent of the IS in the 1960s. In [83], the authors assign the name, *conceptual schema-centric development* (CSCD), to this goal. According to CSCD, in order to generate software artifacts to support the business of an enterprise, general information about the enterprise and the tasks which the enterprise business performs must be captured in what is known as a *conceptual schema*. In order to generate software from a conceptual schema, CSCD prescribes that a conceptual schema must be *explicit*, *executable*, and *evolving*.

The core of CSCD is grounded in the following guiding principles:

- **The Principle of Necessity**: "To develop an information system it is necessary to define its conceptual schema."
- **The 100 Percent Principle**: "All relevant general static and dynamic aspects, i.e. all rules, laws, etc. of the universe of discourse should be described in the conceptual schema. The information system cannot be held responsible for not meeting those described elsewhere, including in particular those in application programs."
- **The Conceptualization Principle**: "A conceptual schema should only include conceptually relevant aspects, both static and dynamic, of the universe of discourse, thus excluding all aspects of (external or internal) data representation, physical data organization and access as well as all aspects of particular external user representation such as message formats, data structures, etc."

The MDA approach aligns closely with the guiding principles of CSCD, especially as it relates to the CIM-to-PIM transformation challenge. When MDA is applied as the approach to achieve CSCD, the PIM takes the role of the conceptual schema of the IS [83]. This is especially underscored by the 100 percent principle and the conceptualization principle, as they reflect the need for a conceptual schema to exclude details of a particular platform as well as the need of the conceptual schema to capture domain information of the enterprise. Thus, the principles of CSCD were followed to guide the formulation of MDA meta-design.

To achieve CSCD, the conceptual schema requires information found in the domain of the enterprise. This domain information is translated into functional information which pertains to one of the three categories of information systems functionality: memory, informative, and active. The purposes of these functions are to *store* information, *communicate* information, and to *process* information, respectively.

The pieces of information must be captured by the domain vs. information system, as prescribed by CSCD, are presented in Table 5.1.

Information Found in Domain	**Information Required in IS Conceptual Schema**
• Entity and relationship types • Derivation rules • Integrity constraints • Domain states • Domain events (a state change consisting of a set of elementary changes in instances of entity or relationship types) – Type – Effect	• Information Base (state of the domain) – Temporal or non-temporal • Requests (commands) – Type – Required output • Notifications – Conditions – Required output • Inference capability – Inference mechanism – Derivation rules • Actions – Request types or generating condition – Behaviours

Table 5.1: Requirements of conceptual schema-centric development from [83]

5.2.2 Why Enterprise Ontology?

DEMO aligns closely with the theory of conceptual schema-centric development, as enterprise ontology focuses on the exact domain information that is required by the conceptual schema. In fact, the notion of the conceptual schema is also central to the theory of enterprise ontology and the DEMO method.

- Entity and relationship types are part of the DEMO fact model
- Derivation rules are expressed as part of the DEMO fact model and action model

- Integrity constraints are expressed as imperative and declarative business rules
- The state of the domain (known as the "business world" in enterprise ontology) is captured as original facts of the production and coordination worlds
- Domain events are expressed as event types in the DEMO fact model and action model

A DEMO model also aligns with the working definition of a computation independent model (CIM) of this thesis (see Section 4.2.1). A CIM is primarily concerned with the functional details of the business of an enterprise. Specifically, the components and behaviour of its operations activities, such as resources, actors, processes, rules, etc. These concepts are the fundamental building blocks of enterprise ontology. As explained in [33],

> "Enterprise ontology offers a coherent, comprehensive, consistent and concise understanding of the essence of an enterprise, fully abstracted from realisation and implementation."

The alignment of enterprise ontology with conceptual schema-centric development means that enterprise ontology is an ideal theory to apply to guide the creation of CIM models and that DEMO is suitable for modelling at the CIM level.

5.2.3 Why the Four-Layered Software Architecture Pattern?

In his seminal book, *Patterns of Enterprise Application Architecture* [84], Martin Fowler's perspective on enterprise applications follows a three-layered view of software architecture: the *presentation logic layer*, the *domain logic layer*, and the *data source logic layer*. The responsibilities of these layers are to display, manipulate, and store enterprise data, respectively. According to Fowler,

> "We can identify the three common responsibility layers of presentation, domain, and data source for every enterprise application."

This was found to be an ideal architectural pattern to add structure to guide the development of a PIM, because any enterprise application, regardless of the target platform, can be expected to exhibit elements of these three layers. Additionally, a fourth *persistence logic layer* [85] was factored out from the business logic layer in order to emphasize that information on the persistence of domain entities is not directly available at the CIM level and therefore must be inferred. Given that the primary concern of the PIM is to capture elements of the application architecture, the conceptual framework is expressed in terms of both MDA and four-tier layered software architecture pattern.

MDA Transformation Meta-Design			
Target Architecture: *4-Tier Layered Architecture*			
Target Technology: *Enterprise Application*			
	CIM	**PIM**	**PSM**
Presentation Logic			• Views • Layout • Navigation • UI Elements • Styles
Business Logic	• Business vocabulary • Business rules • Business entities & relationships • Business events & effects	• Data processing algorithms (notifications, requests, & actions) • Inference capabilities • Data model (entities & states)	• Data representation (external or internal) • Message formats • Data structures • Control structures • Naming conventions
Persistance Logic		• CRUD operations • Data integrity constraints	• CRUD commands/statements • Data mapping • State management • Naming conventions
Data Source Logic			• Physical data organization • Physical data access • DBMS design & structure • Schema design (if applicable)

Figure 5.1: The MDA transformation meta-design matrix structured with the four-layered software architectural pattern. Design concepts are captured at the relevant intersections of different architecture layers and abstraction levels in the matrix.

5.2.4 MDA Transformation Meta-Design

Figure 5.1 presents the conceptual framework as a meta-design to solving the research problem, showing the arrangement of the design decisions when enterprise ontology is put in the context of the four-layered software architecture pattern to achieve conceptual schema-centric development. This allowed the MDA conceptual framework to be structured in terms of the software architecture domain. In order to bridge the semantic gap, information captured by enterprise ontology was embedded into the business logic layer of the layered software architecture pattern. The constructs and considerations positioned in each cell of the matrix are based on findings from the aforementioned background literature on enterprise ontology [33], CSCD [83], and enterprise software architecture [84], as well as from the characteristics of the MDA abstraction levels from Chapter 4.

Mapping constructs from the CIM to the PIM in the context of the four-layered software architecture pattern sheds light on which layers and design decisions are applicable or out of scope at both abstraction levels. Moreover, it makes clear critical design aspects that may be missing when designing an MDA transformation. Where aspects are missing, either a new view could be added, or a different modelling language could be used to capture the missing information.

The meta-design of an MDA transformation allows for the MDA transformation to be designed following the systems thinking approach[1]. This applies to the three abstraction layers of the MDA transformation in the following ways:

- The CIM level must capture a holistic view of the construction or *organization* of the enterprise, as opposed to the function or *business* of the enterprise.
- The PSM level must capture a holistic view of the architecture of the software that will be generated to support business of the enterprise.
- The PIM level must capture the business requirements of the enterprise from the CIM and express them as software requirements, components, and functionality to be implemented by the PSM.

In order to provide a holistic view, a CIM or PIM should be composed of multiple views, possibly expressed using different modelling languages, so that each distinct aspect of the enterprise or software is expressed in a model that best captures its semantics [81].

As a PIM takes the role of the conceptual schema when MDA is applied to achieve CSCD, the PIM is the key component of the MDA approach. The complexity of modern enterprise applications necessitates that not only must a PIM capture the domain aspects expressed in the CIM, but it must also express core computational aspects that allow for

[1] "Systems thinking is an approach to problem-solving, which goes hence and forth between a global, holistic view on a system and a detailed, specific view on its constituting parts." [33]

the management of enterprise data and automation of business processes. We therefore postulate that a conceptual schema not only capture structural and behavioural semantics of domain entities, but also execution semantics. This makes explicit how behaviours are to be computationally implemented and ensures that the generated executable artifacts behave at runtime according to the expected behaviour as expressed in the CIM. This is exhibited in the meta-design in Figure 5.1 at the PIM level, which introduces design decisions pertaining to the persistence level of the software architecture.

The constructs of DEMO at the CIM level align with the business logic layer of the PIM. The meta-design also supports the suitability of low-code as a target technology using MDA. As discussed in Chapter 3, low-code development has been greatly influenced by principles of model-driven engineering, and for good reason. Key to achieving the purported benefits of low-code development – such as increased productivity and better business-IT alignment – is the use of conceptual models. As Bock & Frank emphasize in [40],

> "Conceptual modelling components are among the most important components of low-code platforms and one of the principal ways in which they are able to decrease the need for traditional coding."

With the aim of achieving the goal of CSCD, the conceptual framework in Figure 5.1 makes explicit the domain information to be captured at each MDA level and how this information can be structured by the software architecture layers that should be considered to generate enterprise applications implemented by low-code technology at the PSM level. The PIM level therefore has the crucial role of extracting the domain information from the CIM level and inferring design decisions to realize the conceptual schema of the information system to support the enterprise and its processes.

5.3 Conclusions

The development of a meta-design, consisting of a conceptual framework composed of foundational theories and frameworks, ensured that choices made in the design of the MDA transformation specification, such as included elements and views at the PIM level, are grounded in scientific rigour. In this thesis, these theories and frameworks are conceptual schema-centric development, enterprise ontology, enterprise application architecture, and MDA.

Conceptual schema-driven development (CSCD) provided requirements of information that must be captured from the enterprise domain and by the conceptual schema of the information system in order to generate software artifacts for the information system. Given the alignments between the requirements of CSCD with the disciplines of DEMO and enterprise software architecture, these disciplines were found to be suitable for use in a meta-design to achieve CSCD.

The four-layered software architecture pattern and the information captured at each layer were framed within the context of MDA, formulating the conceptual framework to serve as the meta-design. The meta-design thus makes explicit how information design decisions are positioned with regards to the MDA abstraction levels and the layers of the four-layered software architecture pattern. This makes it easily identifiable not only what information must be present for each aspect of the software architecture, but it also makes it distinguishable what information may be translated directly from a higher abstraction level and what information and design decisions must be inferred.

The meta-design also provides requirements to better inform the choice of a PIM modelling language for the DEMO to Mendix MDA transformation. Given the information to be captured at the CIM level and the PSM level, the views and notation of the PIM level must be able to capture all the required constructs from the CIM level and express the structural, behavioural, and execution semantics of these constructs in a computational sense at the business logic layer and persistence logic layer such that they can be easily understood by humans for analysis and be passed on to the PSM level for implementation.

The conceptual framework thus served as a guide for the design sprints in the later chapters of this thesis. The conceptual framework aided the remainder of thesis to be scoped by targeting a subset of specific architectural layers at which DEMO could be straightforwardly mapped to the PIM and PSM levels for transformation to a low-code application. The next chapter discusses the preliminaries of the experimental design sprints to realize an MDA transformation in accordance with this chapter's meta-design.

CHAPTER 6

Experimental Design Preliminaries

The MDA transformation mapping artifacts produced through this thesis study were experimentally designed through agile design sprints. This chapter provides foundational details of the modelling languages and procedure of these sprints. To keep a manageable scope, the sprints focus on using only the DEMO fact model as the source CIM. Three UML profiles were first experimentally explored as PIM modelling languages before deriving a novel UML profile, *pimUML*, to be used as the PIM. The metamodel of the Mendix low-code development platform was used as the target PSM. The mappings of each sprint were demonstrated and evaluated using the *Rent-A-Car* academic case study.

6.1 DEMO Fact Model

The scope of the thesis design sprints focuses on the DEMO fact model (FM). The FM captures facts constituting the state space and transition space of the *production* world of the scope of interest (SoI). As such, it models the various types of P-facts which can exist within the SoI. The FM is expressed graphically with an Object Fact Diagram (OFD). The metamodel of the OFD of the DEMO fact model is presented in Appendix B. Additionally, facts that are difficult to represent graphically can instead be included textually as derived fact specifications (DFS). The FM plays a key role in capturing the static information of the enterprise domain. In the context of the meta-design in Figure 5.1, it specifically captures the business entities and relationships, as well as the domain events which pertain to these entities.

The OFD captures elements of the fact model with notation elements representing various P-fact types, such as declared and derived entities, their attributes and properties, the events which concern these entities, and the types of values of which attributes may be. The concrete modelling concepts that can appear in the OFD are listed in Table 6.1.

DEMO FM Modelling Concepts
Declared entity type
Attribute type
Event type
Derived entity type (specialization of entity type with *event type*)
Derived entity type (specialization of entity type with *derivation rule*)
Derived entity type (nameless specialization of value type with *event type*)
Derived generalized entity type
Derived aggregate entity type
Property type
Cardinality laws
Value type - user-defined, categorical
Value type - user-defined, non-categorical
Value type - pre-defined

Table 6.1: Modelling concepts represented in the Object Fact Diagram of the DEMO FM

Event types may appear inside specialized entity types that are derived upon the state change induced by an occurrence of the event represented by the event type. Some entity types may be specialized upon the fulfillment of a condition specified by a *derivation rule*. For example, in [33], an example is given of a `Person` entity being specialized as a `Student` entity upon being admitted to an educational institution. The `Student` is identified with their `student number` which is a different logical identifier than that of the `Person`, which could be their `name`.

A peculiar type of model construct in the DEMO fact model is the nameless derived entity which is a specialization of a value type. This construct is not documented in the DEMO fact model in [35] nor in the GOSL syntax description in [33]. However, it appears in the Rent-A-Car case study (see Figure 6.4). The assumption is hereby made that the purpose of this model concept is that, since value types cannot be the domain nor range of property types, the nameless entity type is derived from the value type upon the occurrence of a concerning event type so that the value type may be transitively related to another entity type.

The information captured by the FM can be effectively translated to design decisions at the business logic layer of the PIM. Likewise, architectural decisions belonging to the persistence logic layer of the PIM can be inferred from information captured in the FM.

Due to the complexity, textual transformations are outside the scope of this thesis. Therefore, only elements of the DEMO fact model that are expressed with the Object Fact Diagram (OFD) are considered in the transformation mappings from DEMO.

6.2 UML Profiles

The Unified Modeling Language (UML) was chosen as the primary candidate conceptual modelling language for use in the experimental designs of the PIM level for a few key reasons. First and foremost, it is recommended for use as a PIM by the OMG [86]. This is reflected in the results of the semi-structured literature review of this thesis, as UML was found to be the most commonly used PIM language among the studies reviewed. Lastly, UML is the most commonly used conceptual modelling language in the field of software engineering, both in industry and in academia, as found in [87]. This aligns with a key characteristic of MDA approaches found in Section 4.2.2, which is that PIM models are most commonly used by software architects and software analysts. Therefore, it would be beneficial to use UML in this thesis, for the sake of the proposed transformation's potential adoptability among software professionals and academics.

Another key benefit of UML is its ability to be extended and tailored for particular uses through the definition of profiles [27, 86]. This allows for a selection of the most pertinent UML diagrams and elements to be used for a particular use case.

6.2.1 Standard UML

The latest version of UML, version 2.5.1 [88], is herein referred to as "Standard UML", as it contains all elements of the standard version of UML. In this sprint, all UML diagrams and elements were considered for use as target elements being transformed from the DEMO FM.

6.2.2 Executable UML (xUML)

Executable UML, stylized as xUML, is an executable UML profile first proposed by Stephen Mellor in his book, *Executable UML: A Foundation for Model-Driven Architecture* [89]. It prescribes the use of the UML state machine diagram and action language to capture the execution semantics of domain entities modelled in a UML class diagram.

6.2.3 Foundational UML (fUML)

Foundation UML, stylized as fUML, is an executable UML subset published and maintained by the OMG [90]. It is similar to xUML in that it is focused on capturing execution semantics of applications. It also includes an action language, known as Alf, for the purpose of precisely defining execution semantics. It differs from xUML in that it prescribes the use of activity diagrams for graphically expressing entity behaviours.

6.3 Mendix

Mendix is a low-code application development platform owned and maintained by Siemens. A Mendix application consists primarily of four different components which realize the application architecture. These are pages, domain models, microflows or nanoflows, and

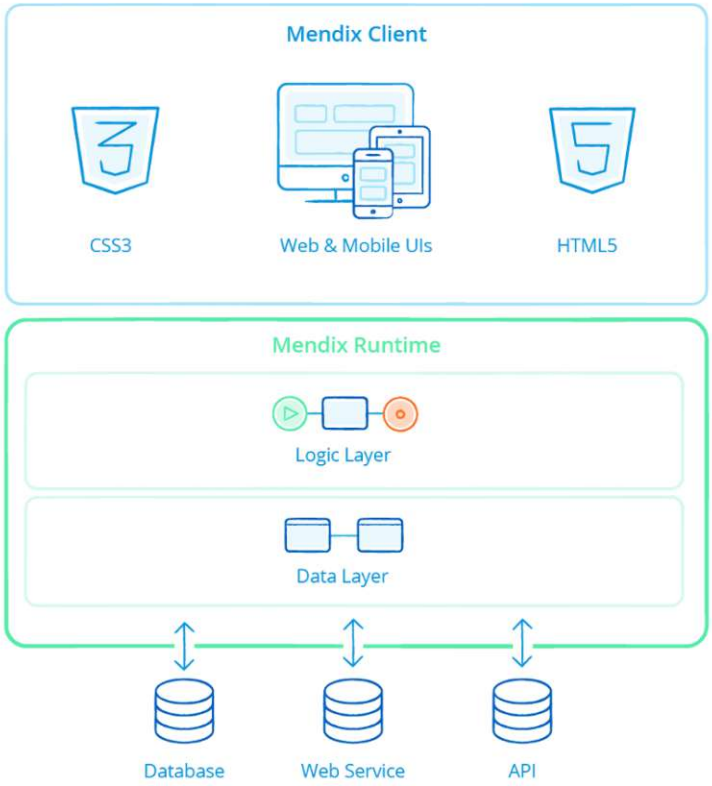


Figure 6.1: The layered runtime architecture of Mendix applications, adopted from [94]

workflows. Application logic is implemented by microflows and workflows which are interpreted at runtime. The data source logic is handled by a built-in HyperSQL (HSQL) database when deployed locally or by a PostgreSQL database when deployed in the Mendix cloud environment [91, 92]. Alternatively, if the Mendix app is deployed in an external environment, an independent database management system must be used. Such database management systems supported by Mendix currently include, among others, PostgreSQL, MySQL, and Oracle Database [93].

The Mendix runtime engine interprets the logic from the domain model and the microflows at runtime and produces database tables and web pages from the domain model and page files, respectively. The runtime architecture of Mendix applications, shown in Figure 6.1, illustrates how the key components of Mendix applications support each other, mapped to layers which are very similar to those presented in Figure 5.1.

6.4 Rent-A-Car Case Study

The designs of the transformation of this thesis are demonstrated with the *Rent-A-Car* academic case study from [95] which is used to demonstrate approaches of conceptual modelling, as well as business process modelling[1].

Rent-A-Car (RAC) is a rental car company that rents cars for both personal and commercial use. RAC operates 50 branches across Europe, offering customers the convenience of returning the rental car at a different location than they picked it up, if they wish. Customers can either rent a car on demand at one of the RAC branches or they can reserve online, by email, or by phone, specifying their desired rental starting day and ending day. However, the starting day must fall within the rental horizon, which is 200 days in advance, and the rental duration must be equal to or less than the maximum rental period of 10 days. The rental horizon and maximum rental period may change from year to year. Additionally, there are extra fees for returning the car at a different location than was agreed on in the contract and for returning the car late – these fees are also updated on a yearly basis.

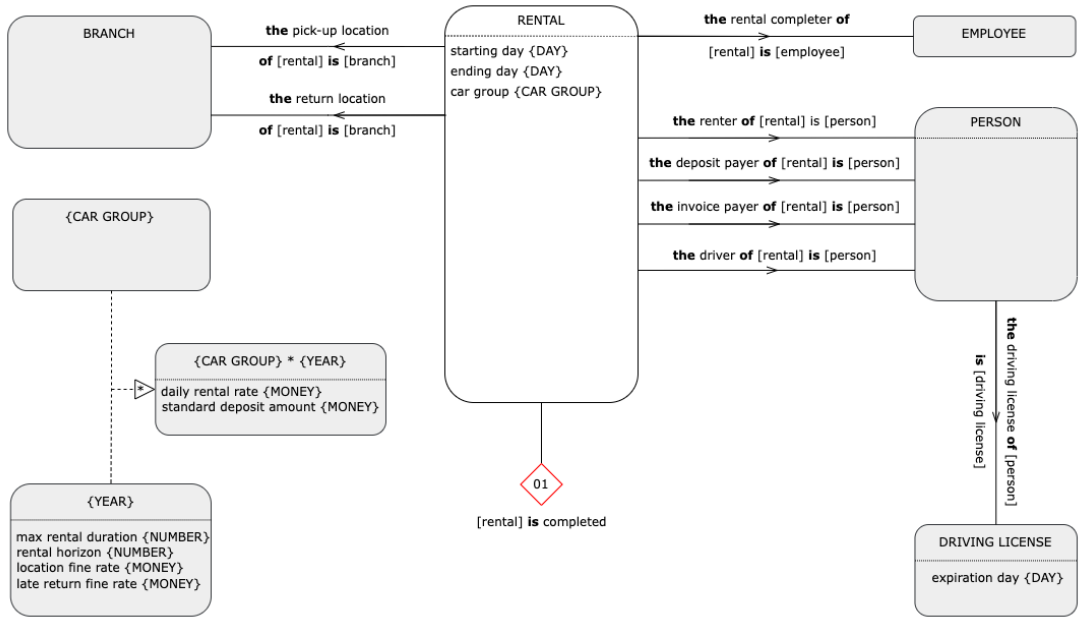


Figure 6.2: The first section of the Object Fact Diagram of the DEMO fact model for Rent-A-Car, adapted from [95].

RAC also offers the flexibility of allowing different persons to be included in a rental as the renter, the rental deposit payer, the final invoice payer, or the driver. As part of the rental form, the person who is to be the driver must provide information from

[1]The *Rent-A-Car* case study is a simplified version of the popular *EU-Rent* case study from [96].

their driver's license, such as its expiration date. Once the rental request form has been completed and submitted with the aforementioned information, an employee is assigned to complete the rental request.

RAC offers a wide assortment of cars for rent; each being categorized as part of a car group with other cars of similar features. All cars within a car group have the same daily rental rate and the same deposit amount, which are updated on a yearly basis.

The above details of the rental parameters are captured as entities, attributes, properties, and events in the fact model as an OFD in Figure 6.2[2].

The rental order is approved as soon as the deposit amount is paid by the rental deposit payer, at which point a car is assigned to the rental. The rental begins as soon as the car is taken from the agreed upon pickup branch.

Once the rental car is returned to an RAC branch, the final invoice amount is calculated as the rental duration times the rental rate plus any applicable fees if the car was returned late or if the actual return branch is different than the agreed upon return branch.

Once the invoice amount is paid, the rental is then complete.

The details on the phases of the rental are captured as entities in a second section of the OFD of the RAC fact model in Figure 6.3.

Finally in order to make sure that there are enough cars available for rent at each branch, cars must be periodically transported between branches by RAC employees. These transports are scheduled to be carried out daily, being managed by a different employee each day.

The details of the transport jobs are captured as in the third section of the OFD of the RAC fact model in Figure 6.4.

[2]The `CarGroup` enumeration contains no values due to a nuance of DEMO: custom categorical value classes only receive their category values upon model instantiation and are therefore not present in the schema illustrated by the Object Fact Diagram.

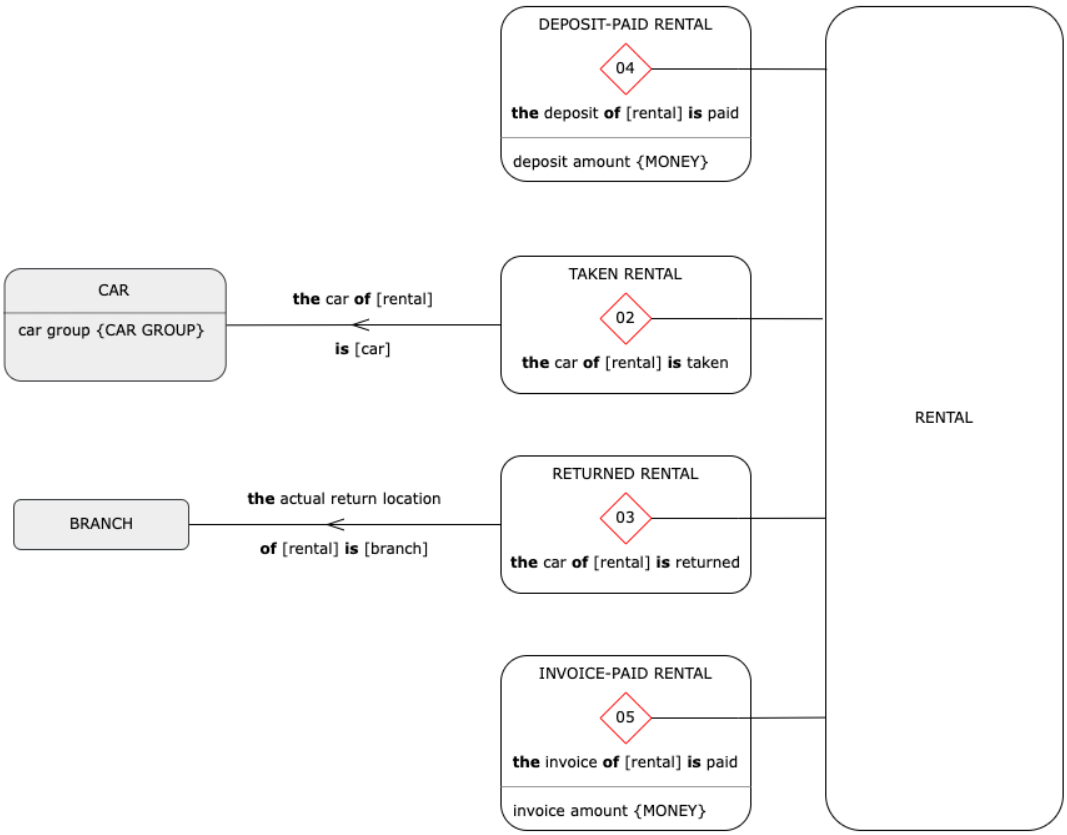


Figure 6.3: The second section of the Object Fact Diagram of the DEMO fact model for Rent-A-Car, adapted from [95].

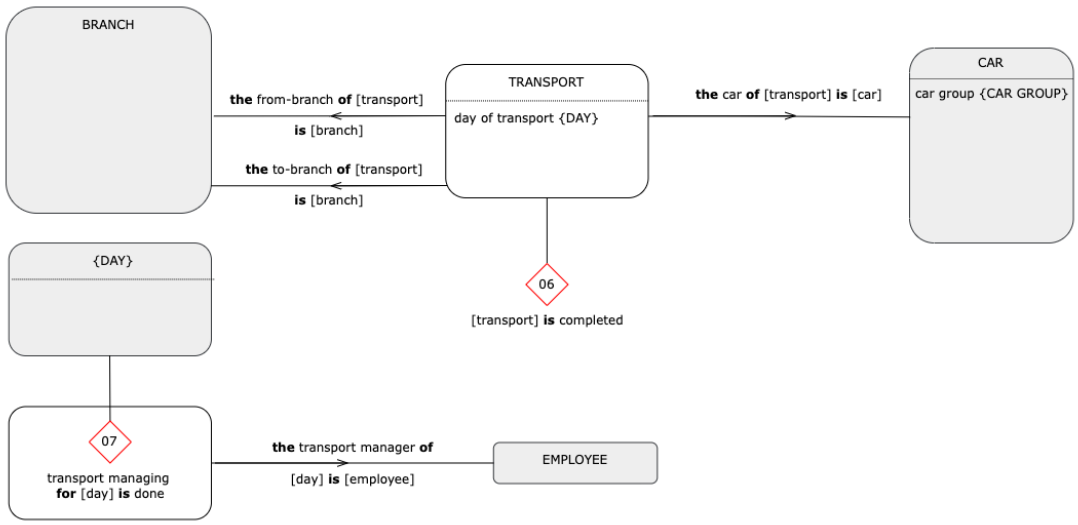


Figure 6.4: The third section of the Object Fact Diagram of the DEMO fact model for Rent-A-Car, adapted from [95].

6.5 Structure of the Agile Design Sprints

Following the agile design science research methodology (ADSRM), five agile design sprints were undertaken to design the mappings of the MDA transformation from DEMO to Mendix. For the first three initial experimental design sprints, the DEMO fact model was fixed as the source modelling language and the candidate UML profiles, discussed in Section 6.2, were varied as the target modelling language. A hardening sprint was then conducted, through which the strongest findings of the first three sprints were synthesized to design mappings to a novel UML profile, *pimUML*. In the final design sprint, pimUML was used as the source model and was mapped to Mendix artifacts.

The three initial agile design sprints (AS1, AS2, and AS3) each produced the experimental CIM-to-PIM transformations, the hardening sprint (HS) produced the CIM-to-PIM transformation to pimUML, and the final agile design sprint (AS4) produced the PIM-to-PSM transformation to Mendix. The next five chapters detail these design sprints:

1. AS1 – DEMO Fact Model to Standard UML
2. AS2 – DEMO Fact Model to xUML
3. AS3 – DEMO Fact Model to fUML
4. HS – DEMO Fact Model to pimUML
5. AS4 – pimUML to Mendix

The design procedure of each sprint follows the following procedure (based on the procedures proposed in [22, 23]):

Step 1: Setting the mapping context

The sprint's target modelling language is assessed to select which of its sub-models are suitable for capturing the structural, behavioural, and execution semantics of the modelling concepts of the DEMO fact model.

Step 2: Understanding the similarities and differences

Given the sub-models selected in **Step 1**, the compatibility of these models with the various aspects of the sprint's source modelling will be briefly discussed.

Step 3: Approaching the metamodel of the target modelling language

An overview of the relevant parts of the metamodel of the sprint's target modelling language is undertaken to identify elements and links to use in the transformation mappings.

Step 4: Determining specific mappings

The non-trivial mappings between the sprint's source modelling language and target modelling language are established and detailed.

Step 5: Determining the complete mapping

The complete mapping from the sprint's source modelling language to the target modelling language is designed in a tabular form, including both trivial and non-trivial mappings.

This procedure yields a series of transformation mappings, presented in a tabular format. These mappings consist of matched pattern rules, matching input patterns of model elements from input models of the source modelling language and producing model elements in the target modelling language according to corresponding output patterns. Some rules may also include references to helper functions that could be implemented to assist with the transformation of values. As these transformations are between modelling languages of different families, the resulting transformations are therefore *exogenous, out-place transformations* [10].

The transformations in this thesis were manually demonstrated and evaluated; however, the transformation mapping rules are assumed to be executed in sequential order, from the top of the table to the bottom. Additionally, it is assumed that each rule can access and modify outputted model elements generated by earlier executed mapping rules.

Together, the resulting CIM-to-PIM mappings from the hardening sprint and the PIM-to-PSM mappings from the final design sprint constitute the main artifact of this thesis: a full MDA transformation specification from the DEMO fact model to Mendix.

CHAPTER 7

AS1: DEMO FM to Standard UML

This chapter details the first design sprint, which aimed to map the DEMO fact model (FM) to the current version of UML, referred to in this thesis as *Standard UML*.

7.1 Design

The Unified Modeling Language (UML) is a general-purpose modelling language which is commonly used for object-oriented modelling [88, 97]. Since its introduction in 1997, UML has evolved into a highly expressive and powerful general-purpose modelling language and is capable of modelling complex object structures and complex processes. As such, UML has grown into the most commonly used modelling language by software engineers, both in industry and in academia [98, 87].

7.1.1 Step 1: Setting the mapping context

Object orientation is primarily concerned with modelling systems from the perspective of classes and objects and how the instantiated objects of a system behave at runtime. The current version of UML consists of 14 different diagrams, each offering a different view of the structural or behavioural aspects of the system. Some of these diagrams are used to model the structure and behaviour of *objects*. Three of the most important and most commonly used of these [97] are the following:

- Class diagram
- State machine diagram
- Activity diagram

Structural Semantics

The most important UML structural diagram is the class diagram. Heavily based on the notions of conceptual modelling, the class diagram models the domain elements, the relationships between these elements, and the resulting data structures of the system. Classes consist of attributes, operations, and associations to other classes or themselves. Using generalization links, classes can also be organized into hierarchical structures, in which higher-order classes are referred to as *generalizations* and lower-order classes are referred to as *specializations*. Objects always have states, which are defined in terms of sets of values of the object's attributes at runtime. To capture the behavioural semantics of objects, the states of an object and the transitions between them are modelled using the UML state diagram.

Behavioural Semantics

The UML state machine diagram captures the behavioural semantics of objects. Using the notions of automata theory, the lifecycle of an object is modelled as the states which an object can assume over its lifecycle [97]. Over an object's lifecycle, the state of the object changes in response to events which *trigger* state transitions. The effects that these events have on the object, in terms of state transitions and operations invoked as an effect of state transitions, is the object's behaviour. Such operations are called *transition effects*. Not only does the state machine show the permissible behaviour of objects, but it also can express conditional constraints on this behaviour through the use of Boolean guard conditions on state transitions.

Execution Semantics

While the state machine diagram depicts the transitions between object states, the UML activity diagram can be used to express how the change of state is realized. As an object's state is defined by the values of the object's attributes, the activity diagram can be used to model the step-by-step algorithmic process of an operation that is executed to change such attribute values to realize the state change. Activities can also express side effect processes, such as CRUD operations, to be executed as a result of a state change. The elements in the activity diagram of which these processes are composed are actions, events, parameters, control flows and object flows.

Actions are the atomic processing steps of activities. They work by accepting input values, processing those values, and outputting output values. These values are passed between the actions of an activity via object edges. The UML specification includes several pre-defined executable actions, such as object creation and variable modification actions. Additionally, an action can be user-defined or can refer to another activity that itself is composed of other actions.

Constraints also provide important execution semantics by imposing rules on how the system should behave at runtime [89]. To specify model constraints, the OMG created the Object Constraint Language (OCL) [99]. OCL is a formal language for writing

expressions, based on predicate logic. These expressions can be used to query the objects in a model and to specify constraints to restrict an object's permissible attribute values and relationships. A model can then be evaluated to validate whether or not the model's objects adhere to these constraints. As OCL is a formal language, the benefit of using OCL, as opposed to natural language, is that OCL expressions are expressed unambiguously.

7.1.2 Step 2: Understanding the similarities and differences

The closest similarity between the DEMO fact model and the chosen set of UML diagram is with the UML class diagram. As both models have their roots in conceptual modelling, they overlap nicely, and many elements from the DEMO fact model can therefore be trivially mapped to corresponding elements of the UML class diagram.

The largest difference between the DEMO fact model and the chosen set of UML diagrams is that the DEMO fact model models structural semantics of business entities while excluding any sort of behavioural semantics. As expressed in Chapter 5, in order for business entities to be implemented in a runtime environment, they require CRUD operations to support their creation, destruction, and modification. Moreover, it must be explicit how such operations are invoked in response to the occurrence of certain events. As such, the UML activity diagram was used to capture the algorithmic processes of the CRUD operations of entities and the UML state machine diagram was used to specify the behaviours and state changes of entities.

7.1.3 Step 3: Approaching the metamodel of the target modelling language

The metamodel of Standard UML was analyzed to determine the elements which could be feasibly used to capture semantics from corresponding elements of the DEMO fact model. The full UML metamodel can be found in the publicly available UML 2.5.1 specification documentation [88].

7.1.4 Step 4: Determine the specific mappings

Explanations behind non-trivial mappings presented in Table 7.1 are discussed below.

i. Declared entity type

A declared entity in DEMO can be trivially translated to a class in UML. Each class has a logical identifier of the data type integer. Although this is not a property directly translated from an attribute type in the DEMO FM, it is added in to ensure data integrity, enabling a logical link between object instances with the event instances which resulted in the objects' creation.

ii. Derived entity type

Derived entities are also translated to classes in UML. Differing from the mapping of declared entities, derived entities do not require a logical identifier, as this is inherited from the original declared entity from which the derived entity is (possibly transitively) derived. Alternatively, if the derived entity is an aggregate entity, it can receive multiple logical identifiers – one from each of its component entities.

iii. Event type

An event type in DEMO represents the acceptance C-act of a transaction and a creation of a new P-fact, resulting in a change of state of an associated entity. To capture the structural semantics of this, the state of an object is captured as a Boolean attribute in the class diagram and can therefore be accessed at runtime. The two values of the Boolean are represented as states in an orthogonal region in a state machine.

A transition between the states with the transaction kind as the trigger captures the behavioural semantics of the event type. Execution semantics are captured as an activity that updates the value of the Boolean when executed, realizing the state change. These regions are orthogonal as to allow for the object to be in multiple states at once, as is implied by the structural semantics of the DEMO fact model. This also does not enforce any such ordering on the state transitions, which is important as there are no such orderings imposed by the Object Fact Diagram of the DEMO fact model.

To capture the execution semantics, an activity diagram is used. This activity diagram consists of three key elements. First, upon operation invocation, a `ReadSelfAction` places the context object of the activity – being the context object of the state machine that invoked the operation, thus being an entity object from the class diagram – on the action's output pin. Next, an `AddStructuralFeatureValueAction` sets the Boolean attribute representing the event type in the object to true. This effectively realizes the state change in the entity object.

iv. Specialization of entity type with *event type*

Instances in DEMO, known as *things*, can belong to a certain type as soon as it conforms to the description of that type. If the thing conforms to the descriptions of multiple types, then the thing belongs to multiple types at the same time. In UML, these semantics can be captured using the generalization set `{incomplete, overlapping}`. Incomplete specifies that an instance of the generalization hierarchy does not have to belong to one of the more specific classifiers; rather, it can belong to the general classifier. Overlapping specifies that an instance can be of multiple specific sibling classifiers at the same time. Therefore, with this generalization set, an instance can be of a supertype or of the supertype and one or many of its subtypes [88]. As an effect of this, the specialized classes do not require

their own logical identifiers, as any instance of these classes are simultaneously an instance of the subtype and the supertype, and thus the identifier attribute and its value are inherited from the supertype.

v. Specialization of value type with *event type* + Property type

Given the semantics behind this construct, a UML association class is used to represent this concept in the PIM. As the derived entity would have been transformed to a class through the rule realizing Definition ii., this class should be replaced with an association class. The property type is then transformed into the association represented by the association class. As this construct is contains a derived entity with an event type, a state machine is also created to capture the entity's behavioural and execution semantics in the PIM.

vi. Specialization of entity type with *derivation rule*

Such specializations in DEMO do not begin to exist when a transaction has completed, but when a fact unrelated to a transaction comes into existence. These types of specializations are accompanied by a derivation rule which specifies the conditions of a thing conforming to the specialized type. A generalization set is still used to capture this, as it still nicely captures the *is-a* semantics. However, the derivation rule is captured as an OCL constraint stating the logical condition that an instance must fulfill in order to become the associated type.

7.1.5 Step 5: Determining the complete mapping

Table 7.1 presents mappings from elements of the DEMO FM (OFD) to UML, with an ID code for each mapping. The descriptions in Step 4 provide the intuition behind the non-trivial mappings.

7.2 Demonstration

In this section, the experimental CIM-to-PIM mapping from the DEMO FM to Standard UML presented in Table 7.1 is demonstrated using the Rent-A-Car (RAC) case study (see Section 6.4).

7.2.1 RAC Standard UML Class Diagram

The UML class diagram of Rent-A-Car is presented in Figure 7.1. Each entity of Rent-A-Car is represented as a class in the pimUML class diagram. The value classes `{YEAR}` and `{DAY}` become custom data types while `{CAR GROUP}` becomes an enumeration. The specializations of `Rental`, namely `TakenRental`, `DepositPaidRental`, `ReturnedRental`, and `InvoicePaidRental`, are related to `Rental` through a generalization set `{incomplete, overlapping}`. The semantics of this is that an instance of Rental can simultaneously also be an instance of any number of the specializations of `Rental`. The Aggregation Entity Type `{CAR GROUP}*{YEAR}` becomes its own

ID	DEMO FM	Standard UML
A1	Value type - user-declared, categorical	Enumeration (class diagram)
A2	Value type - user-declared, non-categorical	User-defined data type (class diagram)
A3	Declared entity type	Class (class diagram) Logical identifier (class diagram) State machine (state machine)
A4	Derived entity type	Class (class diagram) State machine (state machine)
A5	Attribute type	[Derived] Attribute (class diagram)
A6	Event type	Boolean attribute (class diagram) Orthogonal region for corresponding Boolean (state machine diagram) Initial node (state machine diagram) False state and true state (state machine diagram) Transition from initial node to false state [t1] (state machine diagram) Transition from false state to true state [t2] (state machine diagram) Trigger and effect behaviour expression on transition t2 (state machine diagram) Set<Entity><Event> activity (activity diagram)
A7	Specialization of entity type with *event type*	Generalization set {incomplete, overlapping} (class diagram)
A8	Specialization of value type with *event type* Property type	Association class (class diagram)
A9	Specialization of entity type with *derivation rule*	Generalization set {complete, overlapping} (class diagram) OCL expression (class diagram)
A10	Generalization	Generalization with abstract parent (class diagram)
A11	Aggregation	Aggregation (class diagram)
A12	Property type Cardinality laws	Association (class diagram) Multiplicities (class diagram)

Table 7.1: Experimental mapping from the DEMO Fact Model to Standard UML

class `CarGroupByYear` to which its component classes and data types are related via a shared aggregation relationship (except for the enumeration). This class is identified by a compound identifier consisting of its `CarGroup` value and `Year` (as `Year` is a data type, it is additionally represented as a String for instance identification purposes). The states of the entities are made explicit at runtime with the use of the Boolean attributes for the Event Types.

The nameless Derived Entity Type that is a specialization of the Value Type `{DAY}` is represented in the class diagram as an association class named `DayEmployee`. This name reflects the semantics that `{DAY}` is the domain of the association and `Employee` is the range. The Property Type that relates this nameless Derived Entity Type to Employee is directly attached to the `Day` data type and the `Employee` class in the class diagram, and `DayEmployee` describes this relationship.

The property types in the fact model are translated as directed associations, and as the cardinality laws are not specified in the fact model of RAC, the OFD defaults of 0..* on the domain side and 1..1 on the range side are used.

7.2.2 RAC Standard UML State Machine Diagram

Each class in the class diagram – including association classes – has a corresponding state machine that shows the possible state changes of those entities in response to business

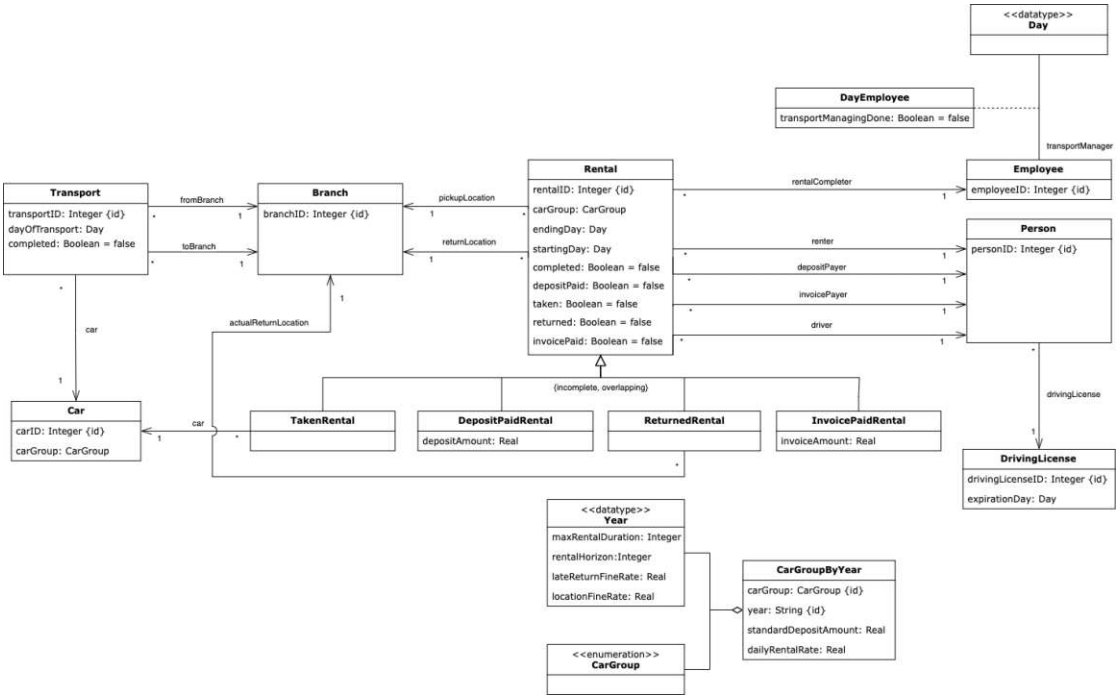


Figure 7.1: UML class diagram of Rent-A-Car

events, which in the case of DEMO are transaction kind accept C-acts. Figure 7.2 shows the UML state machine diagram for the `Rental` class.

Each region has two states: one representing that that Boolean is false and one representing that the Boolean is true. As the default value of each of these Boolean attributes is false, the false state is immediately entered after the initial node. The transition between the false state and the true state contains key behavioural elements: the transition trigger and the transition effect.

The transition trigger corresponds to the transaction accept event which results in the creation of the corresponding P-fact. The event is therefore identified by the transaction number, as well as the "TK" prefix denoting it as a Transaction Kind and the "ac" suffix denoting it as an accept C-event. The transition effect links the behavioural semantics with the activity diagram capturing the execution semantics. That is, each activity depicts the actions which much be executed to realize the state change in the system; specifically, changing the value of the Boolean attribute. For example, in the event that the deposit of an instance of `Rental` has been paid, this is realized in the system as the accept event of a transaction of `transaction kind 04`. Setting `depositPaid` Boolean attribute of the `Rental` object to true and creating the corresponding instance of `DepositPaidRental` are handled by the `RentalDepositPaid` operation referenced by the transition effect behaviour expression.

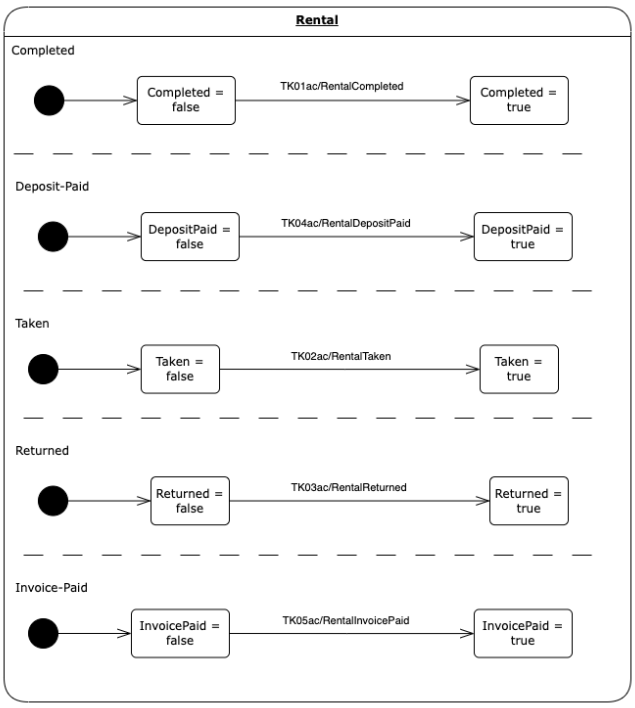


Figure 7.2: UML state machine diagram of the `Rental` class of Rent-A-Car

7.2.3 RAC Standard UML Activity Diagram

For each operation referenced by the effect behaviour expressions of the transitions within the entity state machines, a UML activity diagram is used to illustrate the computational steps invoked to execute the operation. Each activity is composed of predefined UML actions, the semantics of which are defined in the UML specification documentation [88].

Figure 7.3 shows the UML activity diagram of the operation `RentalCompleted`. As the activity is owned by the state machine of the `Rental` object, the activity inherits this as its context object from the state machine, which also has is the `Rental` object as its context object. This means that the first action invoked in the activity — `ReadSelfAction` — can directly access on the context object and place it on its output pin. The object is then passed to the `AddStructuralFeatureValueAction` which also takes in the value `true`, which is an alias for a `ValueSpecificationAction`, and assigns the value true to the `completed` Boolean attribute of the `Rental` object. This effectively completes the state change of the `Rental` object, and the activity is finished.

Figure 7.4 shows the slightly more complex UML activity diagram of the operation `RentalDepositPaid`. This operation has the dual responsibility of changing the state of the `Rental` object and specializing the `Rental` object as a `DepositPaidRental`. The process to change the value of the `depositPaid` attribute of the `Rental` object is the same as the processed described above for Figure 7.3. The additional

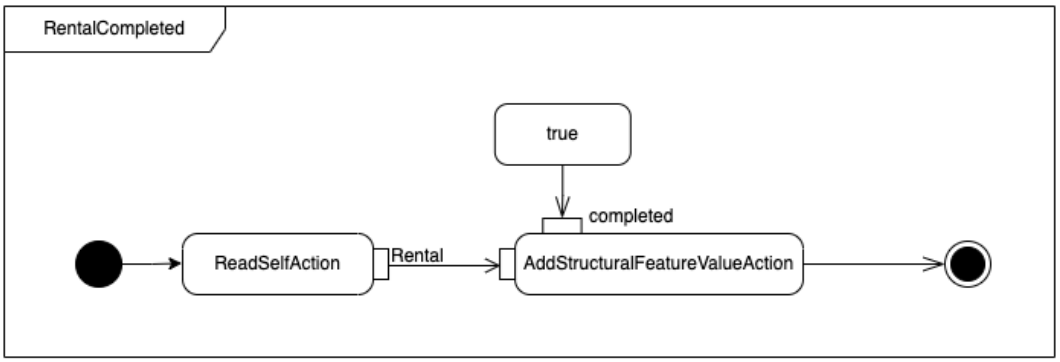


Figure 7.3: UML activity diagram of the `RentalCompleted` operation of Rent-A-Car

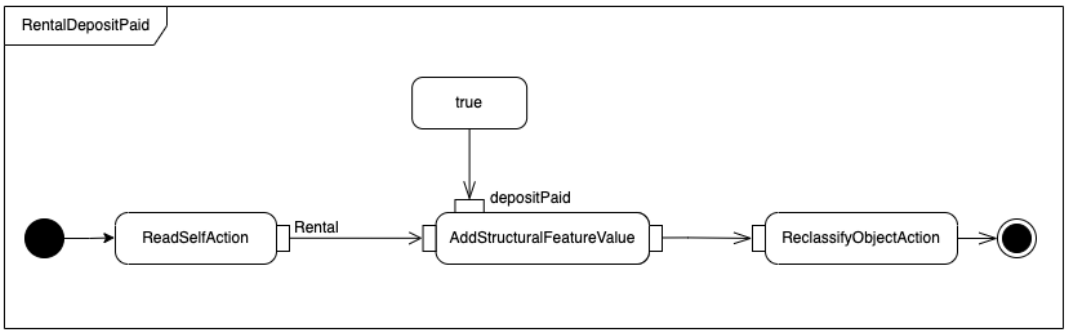


Figure 7.4: UML activity diagram of the `RentalDepositPaid` operation of Rent-A-Car

step is the `ReclassifyObjectAction` which is assigned to add the classifier of `RentalDepositPaid` to the list of classifiers which classify the input object. In effect, the `Rental` object then becomes an instance of both the `Rental` class and the `DepositPaidRental` class. Once this action has completed, the operation is also finished.

7.3 Evaluation

The experimental transformation design from DEMO FM to Standard UML was evaluated in two ways. First, it was quantitatively evaluated by assessing the semantic completeness and correctness of the transformation applied to the Rent-A-Car case study. Second, it was qualitatively evaluated with respect to the key strengths and weaknesses of using Standard UML as a PIM to capture and communicate the semantics of the Rent-A-Car case.

7.3.1 Model Semantic Completeness

The semantic completeness of the DEMO FM to Standard UML transformation is first evaluated against the Rent-A-Car case. The domain semantics captured in the DEMO fact model were extracted from the model elements and listed as domain fact statements through the *verbalization* process described in [33]. These facts were taken as the ground truth of the domain. Second, each fact statement was checked to determine if the semantics were adequately captured and communicated at the PIM level. The number of correctly preserved fact statements of the UML PIM was aggregated for each of the major model concepts (see Appendix G for the full results). The aggregated results are presented in Table 7.2.

DEMO FM Concept	# fact statements in DEMO FM	# preserved statements in UML
Entity Types	12	12
Value Types	3	3
Event Types	14	14
Attribute Types	42	42
Property Types	42	42

Table 7.2: Semantic completeness of the Standard UML PIM against the Rent-A-Car case study

All fact statements were deemed to have been sufficiently preserved in the Standard UML PIM.

7.3.2 Standard UML PIM Strengths

The identified strengths of using Standard UML as a PIM for the Rent-A-Car case study are listed below.

1. **Full suite of UML models and elements available.** This approach freely uses any UML elements from any diagrams which could possibly be used to capture domain and execution semantics from the CIM level. To ensure the understandability of the class diagram, the elements of the UML class diagram that are used in the PIM were selected primarily on the basis of capturing and communicating both domain semantics and the structural semantics of the DEMO FM. The element that best demonstrates this is the *generalization set*. The generalization set construct nicely captures the particular structural semantics of specialized types in the DEMO fact model. The generalization set `{incomplete, overlapping}` makes it clear that a `Rental` can be a `DepositPaidRental` or `TakenRental`, or both at the same time, or neither of them.

2. **Similar structural semantics as the DEMO fact model.** Many of the UML class diagram modelling concepts are very similar to the modelling concepts of the object fact diagram, allowing like model elements between the CIM and PIM levels to be identified easily.

3. **Behavioural semantics are well captured.** State machines create good visualizations of the behavioural semantics of the entities in response to events. The use of orthogonal parallel regions reflects that the entity can be in multiple states at once, and the transitions clearly show how the entities react in response to business events.

4. **Execution semantics are well illustrated.** Activities nicely visualize the algorithmic procedure of the operations that realize the execution semantics of state changes.

7.3.3 Standard UML PIM Weaknesses

The identified weaknesses of using Standard UML as a PIM for the Rent-A-Car case study are listed below.

1. **Missing or vague semantics makes inhibits model execution.** For example, in the state machine diagram, the transition effect label specifies the corresponding activity using a *behaviour-expression*. However, the exact syntax expected for this behaviour-expression is not provided. Rather, according to the Standard UML documentation it is open to being "vendor-specific or some standard language" [88].

2. **The use of ReadSelfAction is not platform-independent enough.** This action assumes that behaviour depicted in the activity diagram belongs to an object of a certain class. This assumption means that the context object could then be retrieved in the target platform by using a corresponding keyword, such as "self" or "this". However, this does not fit well in the case of platforms such as Mendix, in which the behaviours (expressed as microflows) do not belong necessarily to any particular object instance.

3. **Generalization Sets nicely capture domain semantics from the CIM level, but they do not express execution semantics very well.** The model does not clearly show how the implemented constrains and relationships between parent and child objects would vary depending on the modelled generalization set. Execution semantics are difficult to extract from a generalization set. For example, to implement the `{incomplete, overlapping}` generalization set, multiple classification must be possible in the target platform. If a platform does not allow this, it is not immediately clear how the construct could be otherwise implemented.

7.4 Conclusions

In this chapter, an experimental CIM-to-PIM mapping from the DEMO fact model as a CIM to Standard UML as a PIM was designed, demonstrated, and evaluated. As a general-purpose modelling language, the many constructs of UML make it very well suited to capture the domain semantics of the DEMO fact model, particularly the structural

and behavioural semantics of the domain being modelled. Where it has shown to lack is with regards to execution semantics. Although the evaluation shows that a high degree of domain semantics was able to be captured by the UML PIM, it is not clear how some of the constructs used could be easily transformed into PSM constructs.

CHAPTER 8

AS2: DEMO FM to xUML

This chapter details the second design sprint, which aimed to map the DEMO fact model (FM) to Executable UML (xUML).

8.1 Design

Executable UML [89] – also known as *xUML* – is an executable UML profile with the objective of ensuring that models capture precise execution semantics while still remaining platform independent. This is done by prescribing that each entity in the UML class diagram has a corresponding UML state machine diagram to capture the entity's behaviour over its lifecycle.

8.1.1 Step 1: Setting the mapping context

xUML ensures that the behavioural semantics of classes and associations are captured explicitly. For this purpose, state machines are used in xUML exclusively to model object lifecycles [100]. Each state in a state machine has a set of *procedures*, comprised of *actions*, that captures the execution semantics of the state changes. In other words, procedures define the actions that are undertaken to realize an object's change of state. These procedures are expressed in *action language*. Together, the class diagram, state machine diagrams, and procedures capture rich enough structural, behavioural, and execution semantics such that they are together computationally complete. Moreover, since xUML leverages the expressive power of UML, xUML models are not only executable by machines, but they are also comprehensible to humans [89].

Structural Semantics

Just like with Standard UML, in xUML, the class diagram is primarily used to capture the structural semantics of the application, modelling the *data* of the application. As

such, the class diagram in xUML is referred to as the application *domain model*. Domain models are thus expressed using classes and attributes, associations, and constraints.

One aspect in which xUML adds to the existing semantics of UML is with regards to data types. xUML *core data types* include the default UML primitive data types, as well as additional data types which would be pertinent to model execution. These are the date, timestamp, and identifier data types. The whole selection is presented in Table 8.1.

Core Data Type	**Description**
`boolean`	binary value true or false
`string`	a sequence of characters
`integer`	whole number
`real`	decimal number
`date`	calendar date and clock time
`timestamp`	clock time
`arbitrary_id`	an arbitrarily assigned identifier value

Table 8.1: xUML Core Data Type descriptions from [89]

If there is no direct matching data type in the list above for a specialized data type from the CIM, xUML recommends creating a domain-specific data type instead.

A key distinguishing aspect of the class diagram in xUML is its differentiation between implicit and explicit object identifiers. For object-oriented environments, each object must be uniquely identified by the runtime environment. However, this identifier is implicit in the sense that is created, is accessed, and is only meaningful to the runtime platform itself. It is therefore semantically insignificant from the perspective of the domain. In order to better ensure the data integrity of the domain semantics, xUML prescribes that each entity in a domain model have a unique, explicit, and semantically relevant identifier that is clearly indicated in the class diagram, denoted by `{I}`. An identifier in xUML can be a single attribute identifier, a multiple attribute identifier, or a compound identifier that includes referential attributes.

Referential attributes are unique to xUML. These attributes automatically receive their values from the identifier attribute of a related class, much like foreign keys in a database. They are not mandatory in xUML, but including them in the class diagram adds structural and execution semantics by clearly indicating which entity attributes receive their values from an object of an associated entity at runtime. The notation of referential attributes is similar to that of identifiers, taking the form of `{Rn}`, where `n` is the identifying number of the association over which the attribute receives its value.

Lastly, xUML imposes strict restrictions on the use of generalization relationships in class diagrams. Namely, superclasses must be abstract, and the generalization set `{disjoint, complete}` applies to all generalization hierarchies.

Behavioural Semantics

The key technique through which xUML captures the behavioural semantics of domain entities is by modelling their *lifecycles*. The states and transitions of a state machine diagram describe the entity's behaviour as its progresses through different stages in response to domain events. The stages are represented as states and the domain events are represented as transitions between states.

The xUML state machine diagram models the *control* of the application. xUML state machines consist of states, events, transitions, and procedures. While states, events, and transitions capture the state of the domain and the possible state changes, states themselves also contain *procedures* that express precisely what happens to the entity's attributes or relationships to realize a state change, including object creation and deletion.

Execution Semantics

> "By formalizing behavior as actions in procedures, we make the procedures, the state machines, and so the entire domain model, executable."
> ∼ *Executable UML* [89], Chapter 10

To capture the execution semantics of behaviours, each state in an xUML state machine diagram includes an *entry activity* which is expressed as a procedure, written in action language. The entry activity is invoked as soon as the state is entered. The procedure specifies the operation that is to be executed in order to realize the state transition in the system. The procedures of the states in the xUML state machine are composed of *actions*. Actions are primitive units of computation, the semantics of which were originally defined in the *OMG Unified Modeling Language Action Semantics Specification*[1] [89]. xUML relies on these action semantics to provide the execution semantics of xUML models, thus making them computationally complete. Actions can create and delete objects, access object attributes and other objects via links, transform data, and perform other general computations. As such, procedures and their actions model the *algorithms* of the application. In relation to the MDA meta-design (see Figure 5.1), these algorithms are applicable to both the business logic layer and the persistence logic layer of the PIM to perform business computations and CRUD operations, respectively.

Actions in xUML are expressed using action language. Action languages are formal textual modelling languages, expressed using natural language, which specify the effects of actions [101]. They are highly abstract and are therefore platform independent. Originally, the BridgePoint Action Language[2] was used in xUML [89]. Today, there are many different action languages that are compatible with xUML [100].

[1] The specification of the Action Semantics for UML 1.4 can be found at `https://www.omg.org/cgi-bin/doc?ptc/02-01-09`

[2] The syntax specification of the BridgePoint Action Language can be found at `http://www.ooatool.com/docs/BPAL97.pdf`

Constraints are also fundamental concepts that provide execution semantics to xUML models. Constraints in xUML are written primarily in OCL; although, action language may be used as well. They provide execution semantics in the form of Boolean expressions, providing explicit logical rules to runtime environments to ensure data integrity. Constraints are used to add execution semantics to elements of the class diagram, such as identifiers, unique attributes, referential attributes, and derived attributes. [89] provides a list of general OCL constraint patterns, known as *constraint idioms*, which are commonly used in xUML models.

8.1.2 Step 2: Understanding the similarities and differences

The largest similarity between DEMO and xUML is the common prominent influence of both automata theory and conceptual modelling. The largest difference is with regards to how xUML includes constructs to explicitly capture execution semantics, which are not present in DEMO FM. As such, in order to transform DEMO FM models to xUML models, design decisions in which these constructs are used must be inferred from the domain semantics provided by the CIM.

8.1.3 Step 3: Approaching the metamodel of the target modelling language

xUML itself has no published metamodel [100]. However, xUML is based on UML version 1.4, and so its metamodel is loosely taken as the metamodel for xUML.

8.1.4 Step 4: Determine the specific mappings

Explanations behind non-trivial mappings presented in Table 8.2 are discussed below.

i. Declared entity type

As with Standard UML, a declared entity in DEMO is translated to a class in xUML. Each declared entity has a logical identifier, of the data type integer, which not only identifies the entity itself, but is also used to relate instances of the declared entity to its child instances of derived entities. A state machine is also created to capture the behavioural and execution semantics of the entity state changes in response to events.

ii. Derived entity type

Similar to Definition i., except that derived entities have compound identifiers to capture the *is-a* semantics by placing a constraint that the identifier of the object of the class representing the derived entity gets its value from the object of the class representing the parent entity from which the child instance was derived.

iii. Event type

The class of the entity that the event type concerns receives a Boolean attribute (the default value of which is `false`) to represent the event type, capturing the structural semantics. The behavioural semantics are captured by a state and incoming state transition in the entity's state machine diagram. The execution semantics are captured by an entry activity procedure, consisting of an action language command that sets the corresponding Boolean attribute to `true`. The syntax of this command is stated below:

```
<object reference>.<attribute name> = true;
```

iv. Specialization of entity type with *event type*

Special consideration was made to properly capture the behavioural and execution semantics of the specialization of an entity with an event type, both at the business logic layer and at the persistence logic layer. The behavioural semantics of this modelling concept is that in response to an occurrence of the event type, the entity which the event type concerns must undergo a change of state, and an object of the specialized entity type must be instantiated. To realize the execution semantics of this, an entry activity procedure is used as in the state representing the event type. In this procedure, a `create object` action handles the instantiation of the specialized entity type, a `write attribute` action sets the event Boolean in the parent entity to `true`, and a `create link` action links the object of the specialized entity to the object of the entity from which it was derived. The syntax of these commands is stated below:

```
create object instance <object reference> of <class>;
<object reference>.<attribute name> = <expression>;
relate <object reference> to <object reference>
    across <association>.'<verb phrase>';
```

v. Specialization of value type with *event type* + Property type

Given the semantics behind this modelling construct, the association class construct in xUML is used to represent this concept in the PIM. The property type is then transformed into a role on the association represented by the association class.

vi. Specialization of entity type with *derivation rule*

The derivation rule can be captured as an OCL constraint expression in the xUML class diagram. However, the derivation rule itself is expressed as part of the derived fact specifications and does not appear explicitly in the object fact diagram. It therefore falls outside the scope of this thesis and is not explored any further.

vii. Property type

Instead of the names of property types being represented as association labels in xUML, they are instead represented as the role names on the side of the property's range. This is to retain the navigation direction semantics, as associations in xUML do not have navigation adornments.

8.1.5 Step 5: Determining the complete mapping

Table 8.2 presents mappings from elements of the DEMO FM (OFD) to xUML, with an ID code for each mapping.

ID	DEMO FM	xUML
A1	Value type - user-declared, categorical	Domain-specific enumerated type (class diagram)
A2	Value type - user-declared, non-categorical	Domain-specific data type (class diagram)
A3	Declared entity type	Class (class diagram) Logical identifier (class diagram) State machine (state machine)
A4	Derived entity type	Class (class diagram) State machine (state machine) Two logical identifiers, integer attribute and link to parent (class diagram)
A5	Attribute type	Attribute (class diagram)
A6	Event type	Boolean attribute (class diagram) State and transition (state machine diagram) Procedure to change state (state machine diagram)
A7	Specialization of entity type with *event type*	Association with referential attribute constraint {I , Rn} (class diagram) Procedure to change state and create new derived entity instance (state machine diagram)
A8	Specialization of value type with *event type* Property type	Association class (class diagram) Association (class diagram)
A9	Specialization of entity type with *derivation rule*	Association with parent attribute (class diagram) OCL constraint on association (class diagram)
A10	Generalization	Generalization with abstract parent (class diagram)
A11	Aggregation	Association (class diagram)
A12	Property type Cardinality laws	Association (class diagram) Multiplicities (class diagram)

Table 8.2: Experimental mapping from the DEMO Fact Model to xUML

8.2 Demonstration

In this section, the experimental CIM-to-PIM mapping from the DEMO FM to xUML presented in Table 8.2 is demonstrated using the Rent-A-Car (RAC) case study (see Section 6.4).

8.2.1 RAC xUML Class Diagram

The xUML class diagram of Rent-A-Car is presented in Figure 8.1. Each entity of Rent-A-Car is represented as a class in the xUML class diagram. The value classes `{YEAR}` and `{DAY}` become custom data types, while `{CAR GROUP}` becomes an enumeration.

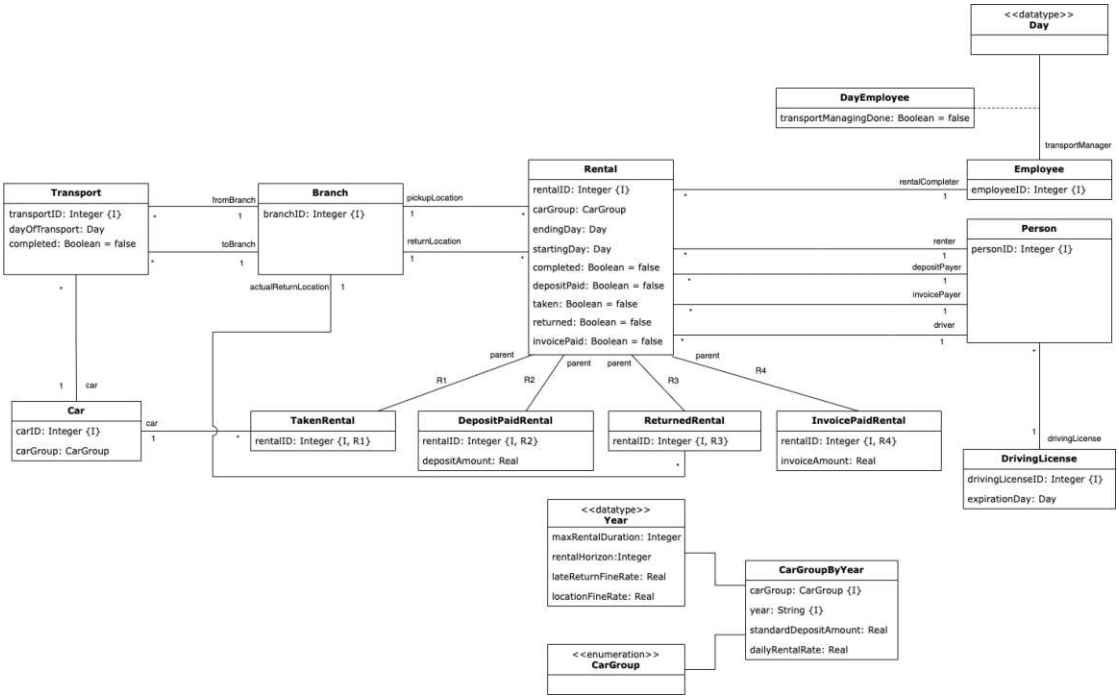


Figure 8.1: xUML class diagram of Rent-A-Car

The associations are not directed in xUML, and aggregation adornments do not exist on xUML associations either. Therefore, all associations in the RAC xUML class diagram are non-directed binary associations. This makes the semantics slightly less precise, constituting a form of information loss.

Due to the restrictiveness of generalization relationships in xUML, *is-a* semantics need to be explicitly added with different constructs. Rather than using generalization links, the *is-a* semantics are communicated in the class diagram by using a regular association with a `parent` role on the side of the parent. Theses associations are arbitrarily labelled in accordance with the naming convention of xUML, which is `Rn`. Each derived entity class requires its own logical identifier that is equal to that of its parent at runtime. In xUML, relations can be used as identifiers. In xUML, identifiers are denoted using `{I}`. Similarly, the specialized classes have the compound identifiers; for example, `TakenRental` has the compound identifier of `{I,R1}`. The semantics of this are that the `rentalID` attribute of `TakenRental` receives its value from the identifying attribute of the class connected over `R1`, which is `rentalID` of `Rental`.

The property types in the fact model are translated as directed associations, and as the cardinality laws are not specified in the fact model of RAC, the OFD defaults of 0..* on the domain side and 1..1 on the range side are used.

8.2.2 RAC xUML State Machine Diagram

Each class in the xUML class diagram – including association classes – has a corresponding state machine that shows the possible state changes of domain entities in response to business events, which in the case of DEMO are transaction accept C-acts. Each state in the state machine therefore corresponds to an event type in the DEMO fact model. The transitions between the states correspond to the transaction accept C-acts that the event types represent. Figure 8.2 shows the xUML state machine diagram for the `Rental` class.

There are no orthogonal states in xUML; therefore, states must be traversed sequentially. This means that the order in which the business events occur must be available in the CIM or it must be inferred at the PIM level. This information is not available in the DEMO fact model, and therefore this constitutes a form of semantic mismatch. However, for demonstration purposes, the order of the states and state transitions in Figure 8.2 are assumed based on the RAC case study description in Section 6.4.

Semantic variation exists between UML and xUML with regards to the initial pseudostate of the state machines. Although business events cannot be assigned to the outgoing transition of the initial state [97] in UML, this is allowed in xUML. Therefore, in the `Rental` state machine, after instantiation, the `Rental` object remains in the initial state until the occurrence of the first business event, `TK01ac`.

8.2.3 RAC xUML Action Language Procedures

There are two different entry activity procedures that can be identified in the `Rental` state machine diagram in Figure 8.2. The first kind has the sole responsibility of changing a Boolean attribute value of the `Rental` object. An example of this is the entry activity procedure in the `completed` state, which simply sets the value of the `completed` Boolean attribute to true. The second kind has the dual responsibility of changing a Boolean attribute value of the `Rental` object and instantiating the class representing the derived entity type that is created as a result of the occurrence of the associated event type in the DEMO fact model. For example, in the `depositPaid` state, the entry activity first creates the new instance of the `DepositPaidRental` class. The next command in the procedure links the newly created `DepositPaidRental` object to the `Rental` object from which it was derived over the association labelled `R1`. Finally, the last command of the procedure sets the `depositPaid` Boolean attribute of `Rental` to true.

8.3 Evaluation

The experimental transformation design from DEMO FM to xUML was evaluated in two ways. First, it was quantitatively evaluated by assessing the semantic completeness and correctness of the transformation applied to the Rent-A-Car case study. Second, it was

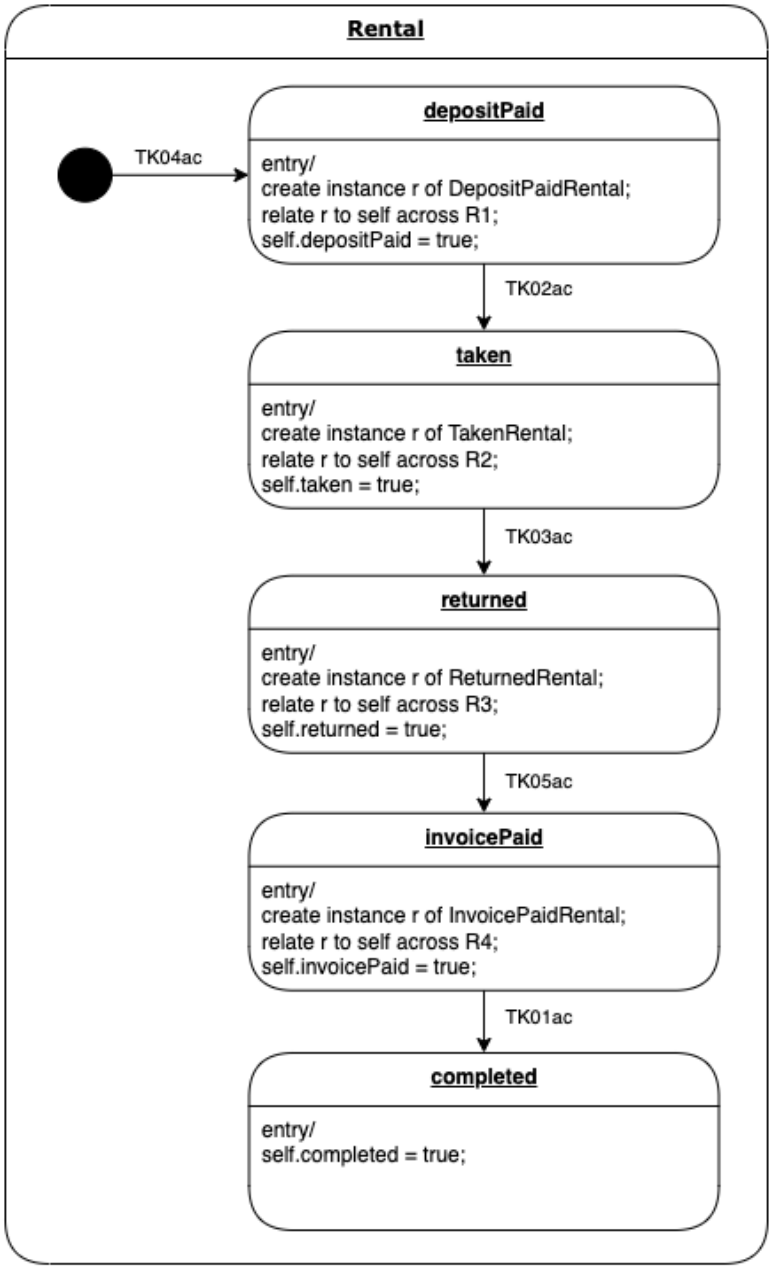


Figure 8.2: xUML state machine diagram of the `Rental` class from Rent-A-Car

qualitatively evaluated with respect to the key strengths and weaknesses of using xUML as a PIM to capture and communicate the semantics of the Rent-A-Car case.

8.3.1 Model Semantic Completeness

The semantic completeness of the DEMO FM to xUML transformation was evaluated against the Rent-A-Car case. First, the domain semantics captured in the DEMO fact model were extracted from the model elements and listed as domain fact statements through the *verbalization* process demonstrated in [33]. These facts were taken as the ground truth of the domain. Second, each fact statement was checked to determine if the semantics were adequately captured and communicated at the PIM level. The number of correctly preserved fact statements of the xUML PIM was aggregated for each of the major model concepts (see Appendix G for the full results). The aggregated results are presented in Table 8.3.

DEMO FM Concept	# fact statements in DEMO FM	# preserved statements in xUML
Entity Types	12	11
Value Types	3	3
Event Types	14	14
Attribute Types	42	42
Property Types	42	42

Table 8.3: Semantic completeness of the xUML PIM against the Rent-A-Car case study

The only fact statement that was not sufficiently captured using xUML was regarding the existence of the aggregate entity type `{CAR GROUP} * {YEAR}`. As adornments do not exist for xUML associations, it is not clear that the relationship between CarGroupByYear and its component classes are specifically aggregation relationships. Therefore, this fact statement was deemed to have not been sufficiently preserved in the xUML PIM.

8.3.2 xUML PIM Strengths

The identified strengths of using xUML as a PIM for the Rent-A-Car case study are listed below.

1. **All necessary aspects of computation captured as execution semantics.** The xUML profile prescribes a set of models which capture all the necessary information such that programs can be executed based off the models alone. This includes data objects, control mechanisms, and algorithms.

2. **Class diagrams in xUML capture entity-relationship semantics better than standard UML.** Using annotations in curly brackets, it is easy to denote a foreign key which receives its value from a related object.

3. **Procedures of entry activities are described directly in the states of the state machine.** The procedure of each entry activity is thoroughly expressed in action language directly in the state to which the procedure belongs, enhancing readability.

8.3.3 xUML PIM Weaknesses

The identified weaknesses of using xUML as a PIM for the Rent-A-Car case study are listed below.

1. **xUML is based on UML v1.4**. UML has changed and greatly evolved since this version. For example, actions are no longer allowed to belong directly to state machines. Rather, they belong to *activities*.

2. **xUML does not fully conform to the UML metamodel**. Some constructs used in xUML do not exist in UML, such as the date data type or referential constraints. Moreover, the semantics of the entry activity do not align with those of the UML specification. In xUML, the procedure which is in invoked to change the state of an object occurs upon entering the new state, effectively *after* the state transition, whereas this procedure should be executed *before* the state change in order to execute actions that result in a state change.

3. **Use of action language for application logic forces a textual-to-graphical transformation.** It would be more intuitive to implement a transformation from a graphical PIM flowchart to a graphical PSM flowchart, rather than from a textual procedure script in a PIM to a graphical flowchart in a PSM.

4. **Objects cannot be in two states at once.** This does not match the semantics of the DEMO fact model, in which entities may be simultaneously in multiple states. This demonstrates a shortcoming of the expressiveness of the xUML state machine diagram.

5. **States must be traversed sequentially.** In order for the state machine in xUML to capture the state and state transitions properly, the order of states must be explicitly drawn. This information is not available in the DEMO fact model; therefore, in order to use the xUML state machine diagram based on the fact model alone, the order of the state transitions must be assumed or inferred, which has a large margin of error.

6. **Graphical notation of xUML is sometimes less expressive than Standard UML.** xUML associations have no association adornments; therefore, there are no aggregation adornments, and associations have no direction in xUML.

8.4 Conclusions

This sprint focused on creating a CIM-to-PIM mapping from the DEMO fact model using the xUML modelling language to model the PIM. This sprint highlighted the important role that behavioural semantics should play at the PIM level. Not only is the PIM responsible for capturing the domain semantics from the CIM, but it also adds an additional perspective: a computational way of thinking. The notion of capturing the lifecycle of objects aligns well with the connection between entity types and event types in DEMO. Specifically, the event of the accept C-act of a transaction results in a new P-fact, and this constitutes an internal state change of an entity. Using a state machine nicely captures the behavioural semantics of this. But xUML takes this further by adding in the necessary execution semantics to define how a system should actually change the variables that comprise the state of the object in question. However, a major drawback of using xUML is its reliance on action language to capture program logic. Although action language is used directly inside of the states of a state machine diagram, having benefits when it comes to model readability, using an activity diagram would perhaps be more useful when using MDA to derive a further downstream model, as it would avoid having to create complex textual-to-graphical transformation patterns to extract notation elements from action language statements.

CHAPTER 9

AS3: DEMO FM to fUML

This chapter details the third design sprint, which aimed to map the DEMO fact model (FM) to Foundational UML (fUML).

9.1 Design

Foundational UML (fUML) [90] is an executable UML profile published and maintained by the OMG. While xUML captures behavioural and execution semantics primarily by using the UML state machine diagram, fUML captures these semantics primarily by using the UML activity diagram.

The aim of fUML is twofold:

- To serve as foundational and minimal subset of UML elements which can replace higher-level UML constructs
- To precisely capture application execution semantics

9.1.1 Step 1: Setting the mapping context

fUML was created as a tool to implement MDA, and one of the key aspects of fUML to assist modellers in designing MDA transformations is the *minimal subset* concept. The complete UML superset contains many constructs that can be useful to communicate semantics graphically to humans for interpretation; however, as a general-purpose modelling language, the semantics of many of its modelling constructs are ambiguous. This ambiguity makes it difficult to derive execution semantics from some UML constructs. Therefore, fUML was designed by taking a computationally complete subset of core UML elements from which higher-level modelling constructs can be derived. An MDA transformation can therefore be more easily created from fUML to any targeted platform

language, as there are fewer elements that need to be mapped than if the entire UML superset were to be used. As fUML is a minimal subset of UML, there are several UML elements and constructs which are excluded from fUML. Some of the most pertinent of which are the following:

- State machines
- Constraints
- Generalization sets
- Association classes

The other key aspect of fUML, which makes it suitable for use in MDA transformations, is its ability to precisely capture application execution semantics. While behavioural semantics capture "changes over time to instances in the semantic domain", execution semantics capture "operational action over time" [90]. Similar to xUML, this is partially achieved through the use of a formal action language. The OMG has its own action language specification designed to provide such additional execution semantics to the fUML profile. This language is Action Language for fUML, or Alf for short [102].

Structural Semantics

As with Standard UML and xUML, the class diagram is primarily responsible for capturing structural semantics in fUML. Classes are therefore used in fUML to model the domain entities and relationships between them. In the fUML metamodel, the `Class` construct inherits from `BehavioredClassifier` and thus inherits the `ownedBehavior` relationship to the `Behavior` construct. Owned behaviours define the behavioural semantics of the entities captured in fUML class diagrams. A `BehavioredClassifier` may have multiple `ownedBehaviors`, but only one of these may be used to specify the runtime behaviour of the owning `BehavioredClassifier` itself [88]. In fUML, the only type of `Behavior` that is included which can concretely capture behavioural semantics of `BehavioredClassifiers` is the `Activity` [90].

Behavioural Semantics

The execution semantics of each behaviour in an fUML model is captured as an *activity* using the UML activity diagram. The activity can be expressed using the graphical notation, using Alf, or using a mixture of both. Activity diagrams are graph models, similar to Petri-nets [88]. As such they are composed of nodes and edges, known as `ActivityNodes` and `ActivityEdges`, respectively.

There are three types of `ActivityNodes` in fUML: *executable nodes*, *object nodes*, and *control nodes*. Object nodes and control nodes are used to model the object and control flows through the activity. Executable nodes express the actual computational steps taken during the run of the modelled behaviour. They are realized as *actions* in fUML.

Execution Semantics

Actions are the sole type of executable node in UML [88]. The following object action subtypes are included in fUML:

- CreateObjectAction
- DestroyObjectAction
- ReadSelfAction
- ValueSpecificationAction
- ReclassifyObjectAction

These actions are used to specify elementary CRUD operations on objects. Objects are passed into and out of object actions through input and output pins, respectively. Links between objects are created and destroyed using link actions. The following link actions exist in fUML:

- ReadLinkAction
- CreateLinkAction
- DestroyLinkAction

The two types of edges used in activities are control flow edges and object flow edges. Whereas control flow edges denote the step-by-step procedural flow through the activity, object flow edges denote how objects are passed between object nodes.

To precisely specify execution semantics in fUML models, the OMG introduced the Action Language for fUML (Alf). Alf was created to serve as an alternative method of expressing execution semantics, whereas the only official way to do so up until its introduction was graphically via the activity diagram [103]. As an action language with a "Java like" syntax, the primary use of Alf is to precisely specify behavioural aspects of a modelled system in a textual way within or alongside a greater graphical diagram of the system [102]. For example, it can be used to specify the execution semantics of function belonging to a class. In the context of behaviours, Alf can be used to specify the execution semantics of the behaviour while the graphical notation is used to specify the behavioural semantics [103]. For example, Alf can be used to precisely define the effect behaviour of a transition between states in a state machine. It is possible to use Alf entirely in place of the graphical notation to model a system, as an extension of Alf's textual syntax maps to the abstract syntax of fUML. This means that Alf can actually serve as a complete textual substitute to the graphical way of specifying behaviours using the activity diagram [102]. This fills a major gap of Standard UML: as no concrete syntax was previously provided for specifying behavioural expressions, the choice of a syntax was left to the users or tool vendors [88].

9.1.2 Step 2: Understanding the similarities and differences

The largest similarity between the DEMO FM and fUML is with regards to the notion of conceptual modelling. The class diagram in fUML contains many constructs which can be used to straightforwardly capture structural semantics from the DEMO fact model.

The largest difference between the DEMO FM and fUML is their respective approaches to the notion of entity state. In particular, fUML excludes the UML state machine diagram, thus excluding detailed specifications of object state and state transitions.

9.1.3 Step 3: Approaching the metamodel of the target modelling language

The metamodels of fUML and Alf were analyzed to determine the elements which could be feasibly used to capture semantics from corresponding elements of the DEMO fact model. The fUML metamodel can be found at [90] and the Alf metamodel can be found at [102].

9.1.4 Step 4: Determine the specific mappings

Explanations behind non-trivial mappings presented in Table 9.1 are discussed below.

i. Value type - TIME scale

While many of the value types from DEMO can be trivially mapped to similar data types from UML, this is difficult to achieve with the TIME value class, as there is no date nor time data type included in the UML specification by default. This is therefore captured in the transformation by declaring a user-defined DateTime data type (maximum one per application).

ii. Declared entity type

A declared entity type in the DEMO fact model is translated to the fUML class diagram by first creating a corresponding class for the entity and then creating an identifier attribute for the entity, marking this attribute as an identifier using the `isID` meta-property. The identifier is assigned upon the instantiation of the class.

iii. Derived entity type

Same as Definition ii., except that the form of the logical identifiers of classes representing derived entities depend on the way by which each of these entities are derived, and thus the logical identifiers are mapped by later translations.

iv. Event type

To capture its structural semantics, a Boolean attribute is used to represent the event type in the class diagram; the default value of this attribute is false. To capture the behavioural and execution semantics, an activity diagram is used. This

activity diagram consists of three key elements. First, an `AcceptEventAction` is assigned to listen for an event indicating the occurrence of the event type, which starts the activity process. Second, an `ActivityParameterNode` receives an object of the class representing the entity that the event type concerns. Third, an `AddStructuralFeatureValueAction` sets the Boolean attribute representing the event type in the object to true.

v. Specialization of entity type with *event type*

As generalization sets are excluded from fUML, to capture the structural semantics of the specialization, particularly, the *is-a* semantics between the derived entity type (this child) and the entity type from it is derived (the parent), a directed association is created from the child object to the parent object. The child object receives an integer attribute to serve as its logical identifier, which must equal that of its parent. In order to capture the execution semantics to ensure the data integrity of this at runtime, an activity diagram is created to specify the create operation of the child object. This activity diagram consists of seven elements. Using an `ActivityParameterNode`, the operation receives the parent object as input. An `AcceptEventAction` is assigned to listen for an event indicating the occurrence of the event type, which starts the activity process. The first action executed is a `CreateObjectAction`, which creates the child object. Next, a `ReadStructuralFeatureValueAction` extracts the logical identifier from the parent object and passes it to an `AddStructuralFeatureValueAction` which assigns this value to the logical identifier of the child object. Next, a `CreateLinkAction` creates the `parent` link from the child object to the parent object. Finally, an `AddStructuralFeatureValueAction` sets the Boolean attribute representing the event type in the parent object to true.

vi. Specialization of value type with *event type* + Property type

Given the semantics behind this modelling construct, the nameless entity type is transformed into a class in fUML to represent this concept in the PIM. The property type is then transformed into a directed association from this class to the class representing the range entity. To capture the inheritance semantics of the entity type, a parent directed association is created from the child class to the parent data type.

To capture the execution semantics of the event type, an activity diagram is created with an `AddStructuralFeatureValueAction` that sets the Boolean attribute representing the event type in the object to true.

vii. Specialization of entity type with *derivation rule*

Similar to Definition v., the class representing the child derived entity receives a logical identifier; however, there is no requirement that the value of the logical identifier of the child equal that of the parent. The activity in this case simply consists of an `ActivityParameterNode` to receive the parent object

as input, a `CreateObjectAction` to create the new specialized object, and a `CreateLinkAction` to link this child object to the parent object.

As constraints are excluded from fUML it is not clear how to explicitly capture the semantics of the derivation rule; this is a form of information loss.

9.1.5 Step 5: Determining the complete mapping

Table 9.1 presents mappings from elements of the DEMO FM (GOSL) to fUML, with an ID code for each mapping.

ID	DEMO FM	fUML
A1	Value type - user-declared, categorical	Enumeration (class diagram)
A2	Value type - user-declared, non-categorical	User-defined data type (class diagram)
A4	Declared entity type	Class (class diagram) Logical identifier (class diagram)
A3	Derived entity type	Class (class diagram)
A5	Attribute type	Property (class diagram)
A6	Event type	Boolean attribute (class diagram) Set<Entity><Event> activity (activity diagram)
A7	Specialization of entity type with *event type*	Identifier property for child class (specialized) (class diagram) Directed association from child to parent with "parent" role (class diagram) CreateNew<ChildEntity> activity (activity diagram)
A8	Specialization of value type with *event type* Property type	Class (class diagram) Associations (domain and range) (class diagram)
A9	Specialization of entity type with *derivation rule*	Identifier property for child class (specialized) (class diagram) Directed association from child to parent with "parent" role (class diagram) CreateNew<ChildEntity> activity (activity diagram)
A10	Generalization	Abstract parent (class diagram) Generalization (class diagram)
A11	Aggregation	Aggregation, shared (class diagram) Logical identifiers (class diagram)
A12	Property type Cardinality laws	Association (class diagram) Multiplicities (class diagram)

Table 9.1: Experimental mapping from the DEMO Fact Model to fUML

9.2 Demonstration

In this section, the experimental CIM-to-PIM mapping presented in Table 9.1 is demonstrated using the Rent-A-Car (RAC) case study (see Section 6.4).

9.2.1 RAC fUML Class Diagram

The fUML class diagram of the Rent-A-Car is presented in Figure 9.1. Each entity of Rent-A-Car is represented as a class in the fUML class diagram. The value classes `{YEAR}` and `{DAY}` become custom data types while `{CAR GROUP}` becomes an enumeration.

Each class in fUML has its own logical identifier. As these relationships do not inherently include inheritance semantics, these needed to be added with explicit structural and execution semantics. In order to ensure the data integrity of the *is-a* semantics of

the data model, is an important that these logical identifiers be managed as to ensure that the specializations of `Rental`, namely `TakenRental`, `DepositPaidRental`, `ReturnedRental`, and `InvoicePaidRental`, each have a logical identifier that is to be equal to that of the object from which it was derived. These classes are related to `Rental` through directed associations with the `parent` role on the side of `Rental`.

The `DayEmployee` class was translated through transformation rule A8. As association classes are excluded from the fUML specification, the nameless Derived Entity Type, which is a specialization of the Value Type `{DAY}`, is represented as a regular class in the fUML class diagram. The Property Type between the nameless Derived Entity Type and the `EMPLOYEE` entity in the DEMO fact model is represented as an association from the `DayEmployee` class to the `Employee` class in fUML. It should be noted however that using a regular class makes it appear as though it carries similar semantics as other classes, when in fact, it represents a special form of a derived entity in the DEMO fact model. Moreover, using two separate directed associations to represent the Property makes the semantics also less clear. Thus, this constitutes a form of information loss.

The property types in the fact model are translated as directed associations, and as the cardinality laws are not specified in the fact model of RAC, the OFD defaults of 0..* on the domain side and 1..1 on the range side are used.

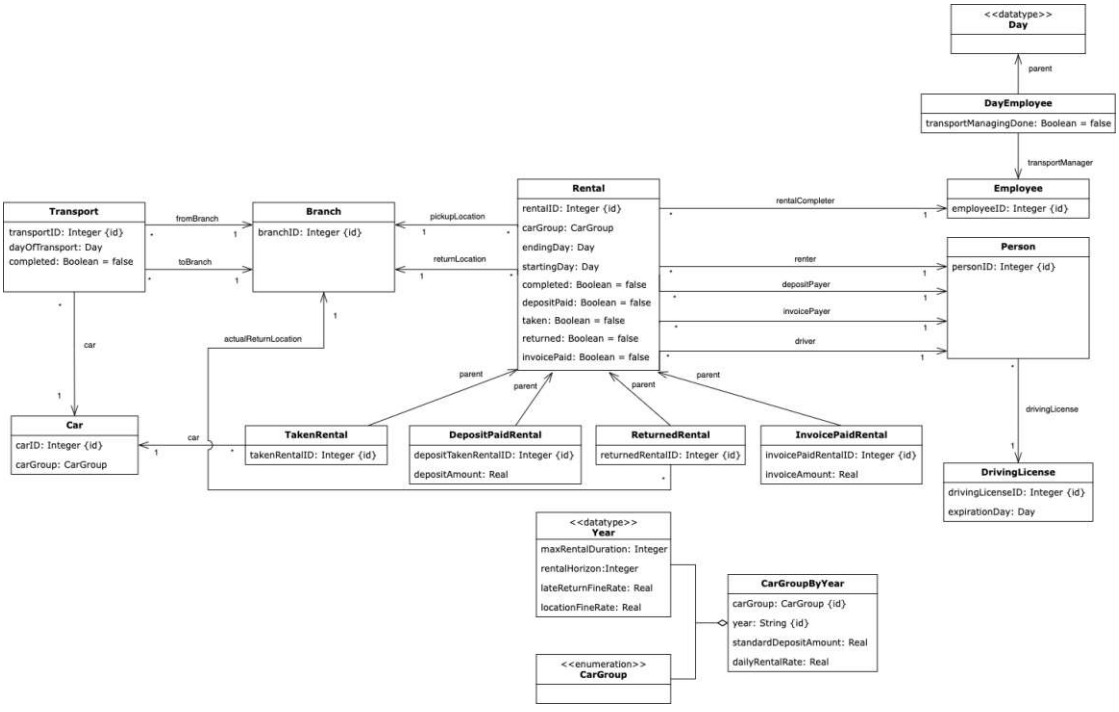


Figure 9.1: fUML class diagram of Rent-A-Car

9.2.2 RAC fUML Activity Diagrams

Figure 9.2 shows the fUML activity diagram of the operation `RentalCompleted`. The activity receives an object of the `Rental` class via the parameter node of the activity. The activity is invoked upon receipt of the event of the accept event action, labelled `TK01ac`, which corresponds to the accept C-event of `transaction kind 01` from the DEMO fact model. The `Rental` object on the parameter node is then passed to the `AddStructuralFeatureValueAction` which takes in the value `true`, which is an alias for a `ValueSpecificationAction`. This assigns the value true to the `completed` Boolean attribute of the `Rental` object. This effectively completes the state change of the `Rental` object, and the activity is finished.

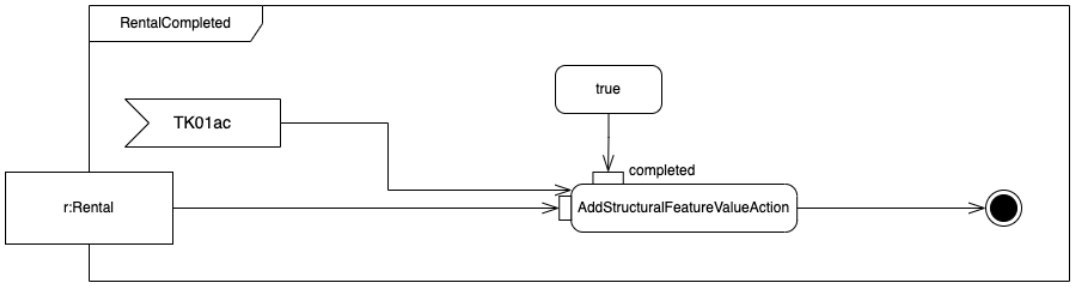


Figure 9.2: fUML activity diagram of the `RentalCompleted` operation of Rent-A-Car

Figure 9.3 shows the slightly more complex fUML activity diagram of the operation `RentalTaken`. This operation has the dual responsibility of changing the state of the `Rental` object and specializing the `Rental` entity instance as a `TakenRental`. The activity receives an object of the `Rental` class via the parameter node of the activity. The activity is invoked upon receipt of the event of the accept event action, labelled `TK04ac`, which corresponds to the accept C-event of `transaction kind 04` from the DEMO fact model. The first action executed is the `CreateObjectAction`, which creates a new instance of `TakenRental`. The next action is a `ReadStructuralFeatureValueAction` which reads the `RentalID` of the parameter `Rental` object and passes it to the next action, the `AddStructuralFeatureValueAction`. This action also receives the newly created `TakenRental` object and assigns the value of `RentalID` to the attribute `takenRentalID` of the `TakenRental` object. Next, the `CreateLinkAction` is executed to create the `parent` link between the `TakenRental` object and the parameter `Rental` object. Finally, the last step is to set the value of the Boolean attribute `taken` of the `Rental` object, representing the occurrence of the event type, to true. The process to change the value of the `taken` attribute of the `Rental` object is the same as the processed described above to change the value of `completed` in Figure 9.2.

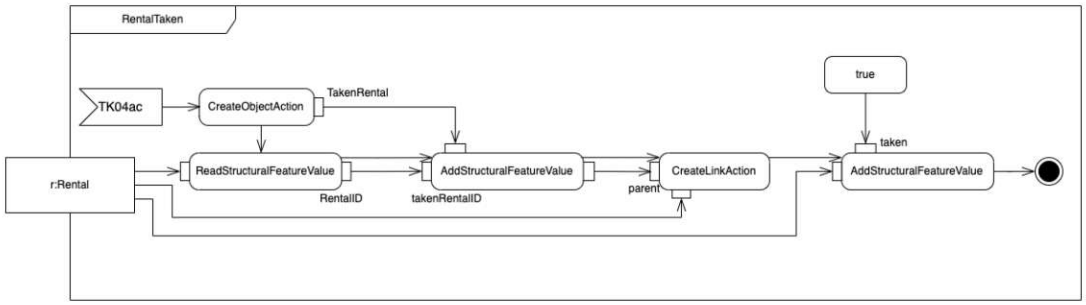


Figure 9.3: fUML activity diagram of the `RentalTaken` operation of Rent-A-Car

9.3 Evaluation

The experimental transformation design from the DEMO FM to fUML was evaluated in two ways. First, it was quantitatively evaluated by assessing the semantic completeness and correctness of the transformation applied to the Rent-A-Car case study. Second, it was qualitatively evaluated with respect to the key strengths and weaknesses of the fUML PIM which emerged from the Rent-A-Car case.

9.3.1 Model Semantic Completeness

The semantic completeness of the DEMO FM to fUML transformation was evaluated against the Rent-A-Car case. First, the domain semantics captured in the DEMO fact model were extracted from the model elements and listed as domain fact statements through the *verbalization* process demonstrated in [33]. These facts were taken as the ground truth of the domain. Second, each fact statement was checked to determine if the semantics were adequately captured and communicated at the PIM level. The number of correctly preserved fact statements of the fUML PIM was aggregated for each of the major model concepts (see Appendix G for the full results). The aggregated results are presented in Table 9.2.

DEMO FM Concept	# fact statements in DEMO FM	# preserved statements in fUML
Entity Types	12	12
Value Types	3	3
Event Types	14	13
Attribute Types	42	42
Property Types	42	41

Table 9.2: Semantic completeness of the fUML PIM against the Rent-A-Car case study

Semantic loss has been identified with two fact statements, both stemming from the omission of association classes from fUML. For the concept `{CAR GROUP} * {YEAR}`, a regular class with two directed associations was used instead.

With regards to the event type fact statement, *transport managing for day concerns day*, as with the other classes, it should be that the Boolean attribute representing this event type belongs to the class representing the value type `{DAY}`. However, instead it belongs to the class `DayEmployee`. Since this is a regular class, the exact semantic relationship between `DayEmployee` and `Day` is not immediately clear, especially since they are linked with a regular association and not a generalization. Therefore, this fact statement was deemed to have not been sufficiently preserved in the fUML PIM.

Similarly, with regards to the fact property type statement, *the domain of transport manager of day is day*, it is unclear as to the actual domain of the `transportManager` association in the fUML class diagram. In the class diagram, it appears that the domain is `DayEmployee`, and as `DayEmployee` is implemented as a regular class, this would imply that the domain of `transportManager` is `DayEmployee` and not `Day`. Therefore, the fact that the domain of `transportManager` is supposed to be `Day` is not sufficiently preserved in the fUML PIM.

9.3.2 PIM Strengths

1. **Minimal subset that includes only elements which can be used to express execution semantics.** Using a smaller amount of modelling concepts would make the task of creating mappings to downstream PSM models for new platforms easier.

2. **Activity diagrams could be nicely mapped to Mendix microflows.** A graphical-to-graphical transformation is much more intuitive than a textual-to-graphical transformation.

3. **Includes Alf action language for the expression of precise supplementary execution semantics.** This platform-independent language can be used to fill in the blanks where execution semantics are missing in the regular UML specification. Such as in the case where a `behavior-expression` requires a formal language to be properly expressed.

9.3.3 PIM Weaknesses

1. **Too restrictive of a subset for capturing all possible domain semantics.** fUML's characteristic of being a minimal subset of UML elements allows models to be formed which are semantically rich enough for model execution while being constructed using a minimal amount of distinct model elements. However, this minimal subset reduces the models to a point at which the interpretation of the semantics by humans could be difficult. Although there is a benefit to fUML being a minimal subset when it comes to the reduction of effort required to create mappings from fUML to other models, as there are less elements to be mapped, a trade-off exists with regards to how behavioural semantics can be effectively communicated to human software analysts with fewer notation constructs to use. For example,

without state machine diagrams, analysis of the states and transitions of objects at runtime by software developers would be more difficult. As state machines clearly illustrate the runtime behaviour of objects, without them as human analysts would have to piece together the behaviours using a collection of activity diagrams and for each class in the class diagram, making the understanding of the behaviour of objects at runtime less intuitive.

2. **Behavioural semantics less intuitive.** While activity diagrams nicely capture and express the execution semantics of operations, the behavioural semantics of objects in the class diagram are not as clear with the exclusion of state machine diagrams. For example, an important concept is that the occurrence of an event type results in the change of state of an object of the entity which the event concerns. Moreover, an entity in DEMO can be in multiple states at once. A state machine clearly shows these semantics; however, they are not as intuitively expressed with just a class diagram and activity diagrams.

9.4 Conclusions

This sprint focused on creating a CIM-to-PIM mapping from the object fact diagram of the DEMO fact model to the fUML modelling language. This sprint highlighted the important role that expressing execution semantics plays at the PIM level. Not only is the PIM responsible for capturing the domain semantics from the CIM, but it needs to clearly show how the behavioural aspects of the domain semantics may be realized in a computational way. However, while the formal syntax of fUML and Alf allow precise execution semantics to be captured and expressed, the behavioural semantics are not as clearly communicated as they were with state machines in the previous two sprints. This is something that should be considered in the design of a PIM.

CHAPTER 10

HS: DEMO FM to pimUML

The previous three chapters detailed experimental design sprints which were undertaken to explore how the DEMO Fact Model (FM) could be mapped to Standard UML, xUML, and fUML. As each of these experimental transformations were found to have their own strengths and weaknesses, the hardening sprint detailed in this chapter synthesizes the strongest alternative choices of various mappings of the previous three agile sprints to produce a UML profile, *pimUML*. The designed transformation mappings from the DEMO FM to pimUML are then presented, demonstrated, and evaluated.

10.1 Design

As Standard UML, xUML, and fUML all draw from the UML family of modelling languages, different mapping alternatives can be easily interchanged to select the mappings that best achieve the objectives of the MDA transformation. Selecting key elements to be used as a PIM model into a new UML profile focuses the scope of the PIM and makes the evaluation metrics more meaningful. While the focus and aim of the first three sprints was to create mappings using elements from each modelling language that could best capture the semantics of the DEMO fact model, while prioritizing the implications of these choices on readability over executability, the goal of pimUML is to prioritize semantics favourable to executable artifact generation over those favourable to diagram readability.

10.1.1 Step 1: Setting the mapping context

The derivation of pimUML is largely influenced by the challenges faced in light of the nuances of the DEMO FM and how to ensure that the semantics of these nuances are retained when translating models. At the same time, the design decisions of pimUML were also influenced by the aim of the UML profile to effectively capture execution

semantics while remaining platform independent. The following challenges and trade-offs were examined with regards to these concerns:

1. Entity instance identification
2. Entity state management
3. Inheritance (*is-a*) semantics
4. Multiple classification
5. Association adornments
6. Data type mapping
7. Operation ownership
8. Action language usage

Structural Semantics

The UML class diagram was selected for use as the primary model view to capture structural semantics in pimUML, constituting a domain model. As the aim of pimUML is to ensure the understandability of the model diagrams while including as precise execution semantics as possible, the elements of the pimUML class diagram were selected primarily on the basis of capturing and communicating the domain semantics in accordance with the structural semantics of the DEMO FM in such a way that they could be easily implemented on any platform. One such way in which this is achieved is by ensuring that each class has a logical identifier to identify which object instantiations represent which individual domain entities. If an instance of a domain entity conforms to entity types represented by multiple types of classes simultaneously, then instantiations of multiple different classes will have the same value of their logical identifiers (but of course, the value of the logical identifier must be unique among objects of the same class).

Similar to xUML, to ensure that date and time values can be captured and stored, pimUML adds a custom `DateTime` data type to the set of default data types of Standard UML. The full list of data type mappings is presented in Appendix E.

Finally, to ensure that the proper semantics of aggregation relationships are captured and communicated, pimUML includes association adornments on association ends.

Behavioural Semantics

As the state machine was shown in agile sprints one and two to nicely capture the lifecycles of objects and the behaviours of those objects in response to events at runtime, the state machine is used in pimUML to capture the behavioural semantics of entity

instances. A state machine is therefore "started" upon the instantiation of the class of which the state machine represents the behaviour.

A key limitation of the state machines in xUML, as demonstrated in Chapter 8, is that the objects cannot be in multiple states at once. Therefore, in pimUML, state machines consist of orthogonal regions, with a pair of states representing the values "true" and "false" of corresponding Boolean attributes, representing DEMO event types, in the class diagram.

The most important elements of the state machines are the transition triggers, the transition effect behaviours, and the state entry events. These three elements are primarily responsible for capturing the behavioural semantics, expressing what transition occurs in response to what event, what operation must be executed to realize that state transition, and what post-operations must be executed in response to the state transition, respectively.

Execution Semantics

Leveraging the precise execution semantics expressed by the actions included in UML, the activity diagram is used in pimUML as the primary means to capture and express the execution semantics of operations. The actions used are exclusively those predefined in the official UML specification [88].

Activities are supplied an object via an activity parameter node. The procedure illustrated in each activity diagram starts at the initial node and ends at a final node. There are two types of flows in the activity diagram: *object flow edges* and *control flow edges*. Control flow edges are the only edges in the pimUML activity diagram that denote the step-by-step ordering of action execution. Object flow edges simply denote the links between sources of objects and the actions that use those objects as input.

Additionally, it was decided to use (albeit sparingly) the action language *Alf*, explored in Chapter 9, to ensure that precise execution semantics are captured by using a formal language, rather than using natural language as Standard UML often suggests. Specifically, the `this` expression is used to precisely communicate that the context object of the state machine is being passed as a parameter to the operation that is called as either the effect behaviour of a transition or the entry activity of a state in the state machine of the context object.

10.1.2 Step 2: Understanding the similarities and differences

As this step in the first three design sprints already discussed the differences between the DEMO FM and each of the UML profiles upon which pimUML is based, this step will instead focus on discussing the rationale behind the choices made where differences existed between AS1, AS2, and AS3.

The decisions of which views and elements to include were largely based on addressing the differences between DEMO and the UML family of languages. Fundamentally, it was

important to ensure that decisions on capturing behavioural and execution semantics could be inferred based on the structural semantics of the OFD. Key to this was the decision on which UML diagrams to include. The choice was made to include all three diagrams explored in AS1, AS2, and AS3: the UML class diagram, the UML state machine diagram, and the UML activity diagram.

Together, these achieve the following,

1. Business entities, as well as their attributes and relationships, are represented in a domain model using the UML class diagram
2. The lifecycles of entities, which include state changes in response to relevant events, are represented using the UML state machine diagram
3. CRUD operations associated with the lifecycles of entities (i.e., effects of state changes) are specified using UML activity diagrams

10.1.3 Step 3: Approaching the metamodel of the target modelling language

The metamodel of pimUML is a subset of the latest version of UML (version 2.5.1). The Action Language for Foundational UML (Alf) is included as well; however, rather than including it as part of the metamodel, pimUML only prescribes its use wherever expressions are required in the diagrams, such as to specify the effect behaviour of a transition in a state machine. Lastly, the Object Constraint Language (OCL) is prescribed for use in places, but the exact details of its usage in the PIM and in translations are outside the scope of this thesis. See Appendix C for the full pimUML metamodel.

10.1.4 Step 4: Determine the specific mappings

Explanations behind non-trivial mappings presented in Table 10.1 are discussed below.

i. Value type - user-declared, categorical

In the DEMO fact model, Value types can appear either with or without associated attribute types. When a value type of a *categorical* scale is declared, it does not include attribute types, as value types of this scale do not have measurement scales nor units [35]. In DEMO, these value types do not receive their categories until model instantiation, and they are therefore left without any associated attribute types nor values beforehand. Therefore, any such value type appearing in a model that does not include any attribute types will be recognized as a value type of a categorical scale. Such value classes are translated to enumeration data types in pimUML.

ii. Declared entity type

Entities are translated to pimUML such that both the structural and behavioural semantics of entities are explicitly captured. The entities themselves are translated as classes in pimUML. These classes are automatically assigned a logical identifier in the form of an integer attribute. The value of such a logical identifier may be accessible and understandable by a user of the system being modelled, as opposed to using an arbitrarily assigned identifier, such as a GUID, which is primarily used by the machine running the system. Drawing on the notion of capturing object lifecycles from xUML, the lifecycles of entities are captured in pimUML using state machines. A state machine is instantiated upon the instantiation of its associate class, the object of which becomes the context object of the state machine. Therefore, the state machine captures the behavioural semantics over the entire lifecycle of the object. In these state machines, each associated event type is captured as a region consisting of two states: a false state to indicate that the event has not yet occurred and a true state to indicate that the event has occurred. State machine diagrams in pimUML are orthogonal because of the semantics of DEMO that entities can be in multiple states at once.

iii. Derived entity type

Same as Definition iii., but without a logical identifier. Since the identifier is dependent on the type of derivation relationship through which the derived entity instance is created, the logical identifier of derived entities is assigned to each corresponding pimUML class at the same time the specialization relationships are translated.

iv. Event type

Event types in the DEMO fact model represent state changes that occur to entity types in response to business events, which in the case of the fact model are transaction accept C-acts. Event types therefore indicate the state into which the entity enters, as well as the event that occurs for the state change to happen. Given this, it is critical that in pimUML, three semantic elements must be captured: what the state is and to what class it belongs (structural semantics), and what event triggers the state change (behavioural semantics), and how the state change is realized in response to the event occurrence (execution semantics).

Whether or not the event concerning a given instance of an entity has occurred is represented as a Boolean: true or false. Upon entity instantiation, the Boolean representing the event is assigned a default value of false.

The behavioural semantics of the occurrence of the event type resulting in a change of state of its concerning entity is captured in the state machine corresponding to the entity's class as a transition between two states representing the two possible values of the event type's corresponding Boolean: true and false. The source of the transition is the false state, and the target of the transition is the true state. The

event is captured as the trigger of the transition and the activity that is executed to realize the change of state is captured as the effect behaviour of the transition. Since Standard UML does not prescribe a formal language to express this, pimUML uses Alf to explicitly express the call of the effect behaviour operation with the context object of the state machine being passed as a parameter to the activity realizing the operation.

The effect behaviour is what captures the execution semantics of the event type. As the execution semantics are not explicitly expressed in the DEMO fact model these semantics are inferred. To realize a state change that is represented as a Boolean attribute, the value of this attribute must change. Therefore, the effect activity is represented using a UML activity model. As previously mentioned, the context object of the state machine is passed into the activity as a parameter. The Boolean attribute of this object corresponding to the event is assigned the value `true` using a UML standard action called an `AddStructuralFeatureValueAction`.

v. Specialization of entity type with *event type*

As in fUML, the inheritance (*is-a)* semantics between the two entity types in DEMO are captured in pimUML as a "parent-child" relationship whereby a directed association is created from the class of the derived entity (the child) to the class of the source entity (the parent). This relationship is made explicit by assigning the parent class the role of `parent` in the association.

The additional behavioural semantics of the instantiation of the derived type are captured using an entry activity in the state machine of the parent entity, whereby an operation is invoked to create an instance of the derived entity upon the state change resulting from the occurrence of the event type. The state entry activity is used to explicitly capture the execution semantics of the instantiation of the derived entity type.

Similar to Definition v., such semantics are not explicitly in the DEMO fact model and are therefore inferred. The activity not only creates the new entity instance, but it also by ensures the data integrity of the underlying "parent-child" relationship. The activity receives the parent object as a parameter. First, the derived entity is instantiated using the UML action `CreateObjectAction`. Next, the logical identifier of the parent object is read and assigned to a variable using the UML action `ReadStructuralFeatureValueAction`. The logical identifier of the newly created child object is then assigned the value of the logical identifier of its parent object. This is achieved using the UML action `AddStructuralFeatureValueAction`. Second, the association with the parent role is instantiated between the child and its parent using the UML action `CreateLinkAction`. After the completion of these actions, the activity is finished.

vi. Specialization of value type with *event type* + Property type

As with Standard UML and xUML, given the semantics behind this construct, a UML association class is used to represent this concept in pimUML. As the derived

entity would have been transformed to a class through the rule realizing Definition ii., this class should be replaced with an association class. The property type is then transformed into the association represented by the association class. The name of the property type is assigned as the role on the end of the association of the class representing the entity on the side of the range of the property type. As this construct is contains a derived entity with an event type, a state machine is also created to capture the entity's behavioural and execution semantics in the PIM.

vii. Specialization of entity type with *derivation rule*

Essentially the same as Definition vi. except that the "parent-child" relationship is looser in the case of specialization through a derivation rule. In this case, the derived entity (the child) may not represent the exact same *thing* as the parent entity, whereas in Definition vi. the semantics are that the derived entity is essentially representing the same *thing* as its parent entity, just in a different state. Therefore, in the case of specialization via a derivation rule, the derived entity may be represented by its own logical identifier, which is dictated by the derivation rule. Therefore, to allow this to happen, the value of the logical identifier is not automatically assigned upon instantiation of the derived entity.

The derivation rule is expressed textually in DEMO and such translations are outside the scope of this thesis and are therefore not further discussed.

10.1.5 Step 5: Determining the complete mapping

A high-level specification of the complete mapping from the DEMO fact model to pimUML is presented in Table 10.1. The descriptions from the previous sub-section provide explanations of non-trivial mappings.

10.2 Demonstration

In this section, the CIM-to-PIM mapping from the DEMO FM to pimUML presented in Table 10.1 is demonstrated using the Rent-A-Car (RAC) case study (see Section 6.4).

10.2.1 RAC pimUML Class Diagram

The pimUML class diagram of the Rent-A-Car is presented in Figure 10.1.

As with all explored UML profiles, each entity of Rent-A-Car is represented as a class in the pimUML class diagram. The value types `{YEAR}` and `{DAY}` become custom data types while `{CAR GROUP}` becomes an enumeration. Each class in pimUML has its own logical identifier. This could correspond to the transaction kind identifier that was responsible for the instantiation of the entity type that the class represents; however, how exactly classes of declared entities receive their identifiers is outside the scope of this thesis.

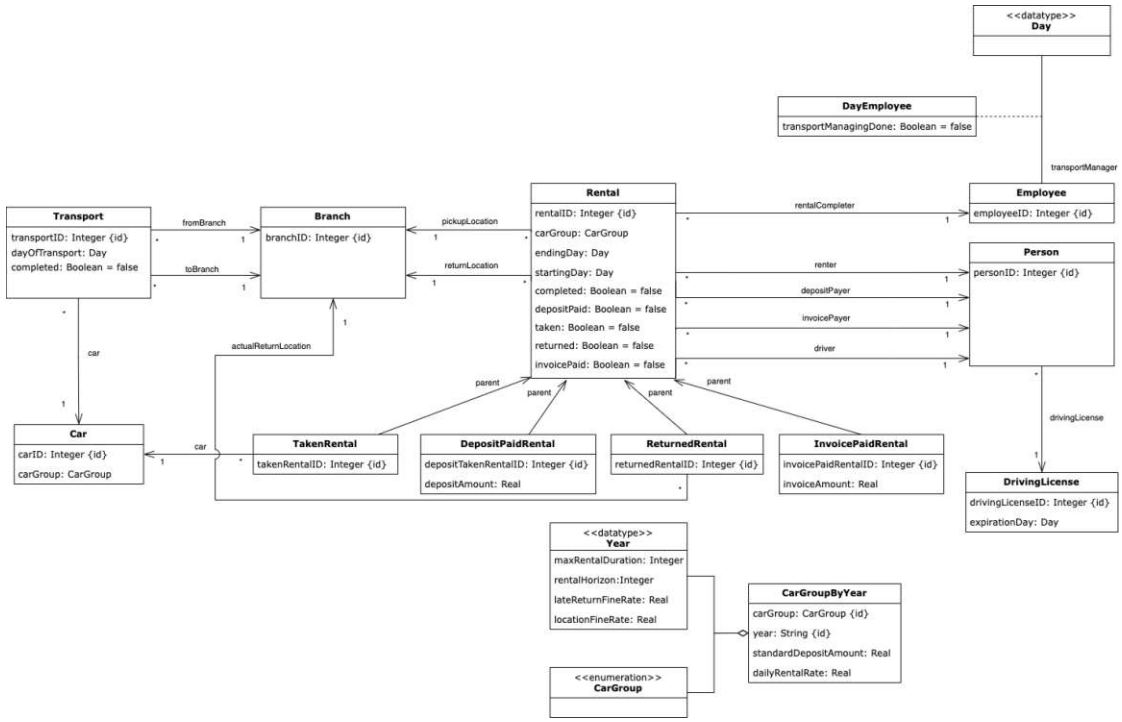


Figure 10.1: pimUML class diagram of Rent-A-Car

ID	DEMO FM	pimUML
A1	Value type - user-declared, categorical	Enumeration (class diagram)
A2	Value type - user-declared, non-categorical	User-defined data type (class diagram)
A3	Declared entity type	Class (class diagram) Logical identifier (integer) (class diagram) State machine (state machine diagram)
A4	Derived entity type	Class (class diagram) State machine (state machine diagram)
A5	Attribute type	Property (class diagram)
A6	Event type	Boolean attribute, default false (class diagram) Orthogonal region for corresponding Boolean (state machine diagram) Initial node (state machine diagram) False state and true state (for corresponding Boolean) (state machine diagram) Transition from initial node to false state [t1] (state machine diagram) Transition from false state to true state [t2] (state machine diagram) Trigger on t2 with Alf expression for transition effect behaviour activity (state machine diagram) Set<entity><event> activity (activity diagram)
A7	Specialization of entity type with *event type*	Logical identifier for child class (class diagram) Directed association from child to parent with "parent" role (state machine diagram) Entry activity in the corresponding true state of the parent (state machine diagram) Create<child> activity (activity diagram)
A8	Specialization of value type with *event type* Property type	Association class (class diagram)
A9	Specialization of entity type with *derivation rule*	Identifier property for child class (class diagram) Directed association from child to parent with "parent" role (class diagram) OCL expression for derivation rule (class diagram) Create<child> activity to create new instance of child class (activity diagram)
A10	Generalization	Abstract parent (class diagram) Generalization (class diagram)
A11	Aggregation	Identifier property in whole class corresponding to each part class (class diagram) Association with shared aggregation adornment (class diagram)
A12	Property type Cardinality laws	Association (class diagram) Multiplicities (class diagram)

Table 10.1: High-level mapping specification from the DEMO Fact Model to pimUML

Generalization sets are not used in pimUML, due to the lack of precise execution semantics and the lack of generalizability of the construct, as in order for the semantics to be implemented, the target platform must include the ability for instances to belong multiple classes of a hierarchy at once, which is not always the case. Instead, as with fUML, transformation A10 uses regular associations for generalization/specialization relationships. Therefore, specializations of `Rental`, namely `TakenRental`, `DepositPaidRental`, `ReturnedRental`, and `InvoicePaidRental`, are related to `Rental` through directed associations with a `parent` role. As these relationships do not inherently include inheritance semantics, these needed to be added with explicit structural and execution semantics. To ensure the data integrity of the *is-a* semantics of the data model, is an important that these logical identifiers be automatically managed as to ensure that objects of derived entities have a logical identifier that is equal to that of the object from which it was derived. Such execution semantics are explicitly added in and are captured by CRUD operations.

The class `CarGroupByYear`, representing an aggregate derived entity type, is related to its parts, `Year` and `CarGroup`, via associations baring a *shared aggregation* association adornment.

The property types in the fact model are translated as directed associations in pimUML, just as with Standard UML and fUML. As the cardinality laws are not specified in the fact model of RAC, the OFD defaults of 0..* on the domain side and 1..1 on the range side are used.

10.2.2 RAC pimUML State Machine Diagram

To capture the behavioural semantics of the entities, particularly their lifecycles, pimUML includes state machines. State machines are created for each class in the class diagram, except for association classes.

Each class in the class diagram – including for association classes – has a corresponding state machine that shows the possible state changes of those entities in response to business events, which in the case of DEMO are transaction kind accept events (C-events). Figure 10.2 shows the UML state machine diagram for the `Rental` class[1]

Each region corresponds to a Boolean value representing an Event Type. As with Standard UML, these regions are orthogonal as to allow for the `Rental` object to be in multiple states at once, as is inferred by the structural semantics of the DEMO fact model. This also does not enforce any such ordering on the state transitions, which is important as there is no such ordering depicted in the Object Fact Diagram of the RAC fact model. Each region has two states: one representing that that Boolean is false and one representing that the Boolean is true. As the default value of each of these Boolean attributes is false, the false state is immediately entered after the initial

[1]All other classes also would have corresponding state machines, but as no other entities have any concerning Event Types in the FM, these state machines are empty and are therefore not shown for simplicity's sake.

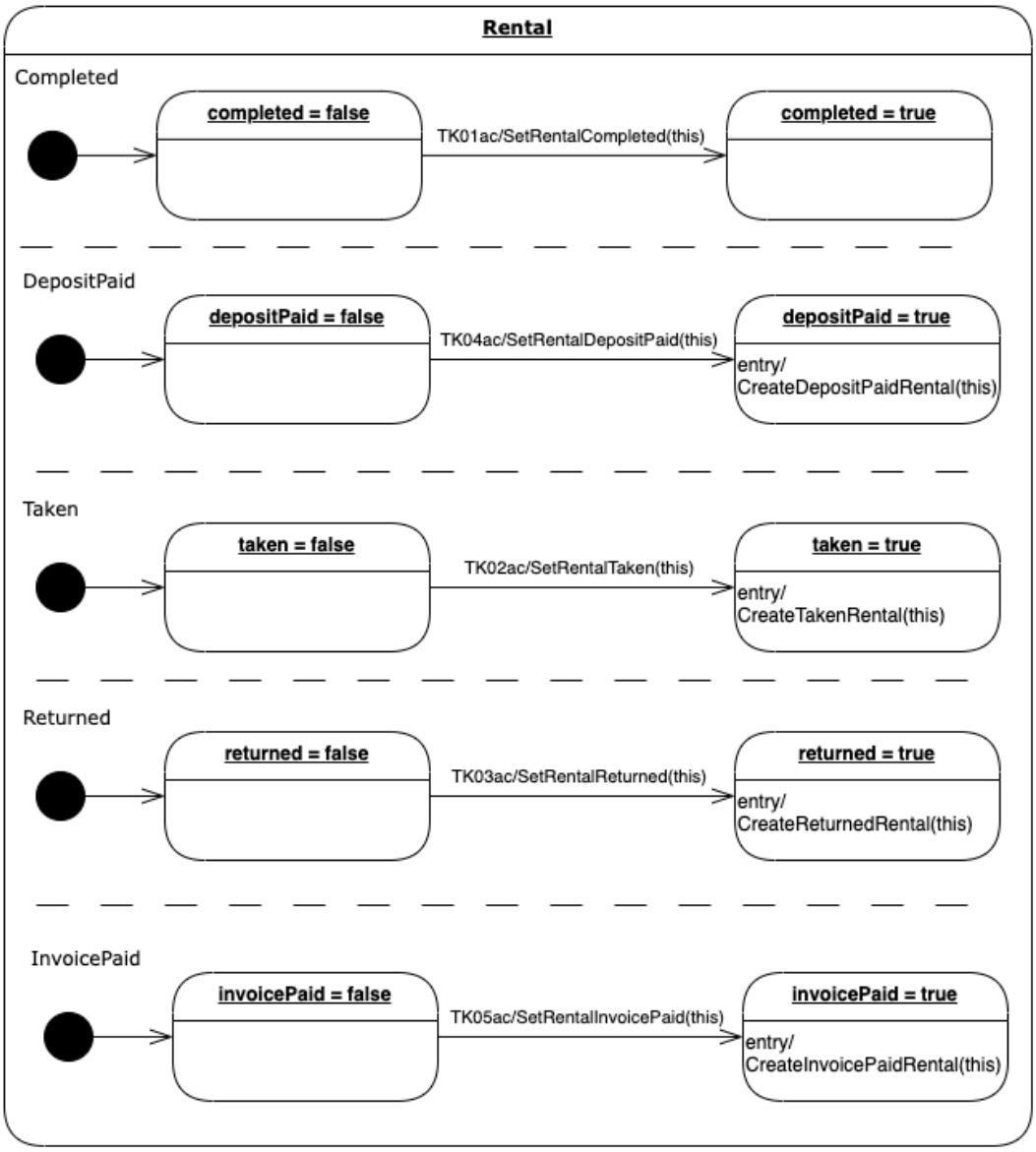


Figure 10.2: pimUML state machine diagram of the `Rental` class from Rent-A-Car

node. The transition between the false state and the true state contains two crucial behavioural components: the transition trigger and the transition effect. The transition trigger corresponds to the transaction accept C-act which results in the creation of the P-fact corresponding to the event occurrence. The event is therefore identified by the transaction number as well as the "TK" prefix denoting it as a Transaction Kind and the "ac" suffix denoting it as an accept C-event.

The transition effect links the behavioural semantics with the execution semantics captured in an activity diagram. Each activity depicts the operations which must be executed to realize the state change in the system; that is, changing the value of the Boolean attribute. For example, in the event that the deposit of an instance of `Rental` has been paid, this is realized in the system as the occurrence of an accepted transaction of `transaction kind 04`. The `depositPaid` Boolean attribute of the `Rental` object is then set to true and the effect of creating the corresponding instance of `DepositPaidRental` is handled by the `RentalDepositPaid` operation referenced by the transition effect behaviour expression.

Each event type that has a corresponding derived entity type in the DEMO fact model requires the invocation of a create operation to instantiate the class representing the derived entity. To realize this, within the region corresponding to such an event type, the `true` state contains an entry event activity expression, expressed in Alf, which denotes an invocation of an operation that handles the creation of the derived object. This operation is depicted in an activity diagram to specify the actions executed to create the new object instance and to associate it with its parent. For example, in the `depositPaid = true` state in Figure 10.2, the expression of the entry activity, `CreateDepositPaidRental(this)`, invokes the `CreateDepositPaidRental` operation and passes the context object of the state machine (referred to by the `this` expression) as the parameter, which is an instance of the `Rental` class.

10.2.3 RAC pimUML Activity Diagram

Figure 10.3 shows the pimUML activity diagram of the operation `SetRentalCompleted`.

The activity receives an object of the `Rental` class via the parameter node of the activity. The activity is invoked upon receipt of a `Rental` object on the parameter node of activity. This object is immediately passed to the `AddStructuralFeatureValueAction` which takes in the value `true`, which is an alias for a `ValueSpecificationAction`, and assigns the value true to the `completed` Boolean attribute of the `Rental` object. This effectively completes the state change of the `Rental` object, and the activity is finished.

Figure 10.4 shows the slightly more complex pimUML activity diagram of the operation `CreateTakenRental`. This operation has the responsibility of creating a new instance of the specialized entity, `TakenRental`, and relating it to the `Rental` entity instance from which it is derived. The activity receives an object of the `Rental` class via the parameter node of the activity. The activity is invoked upon the op-

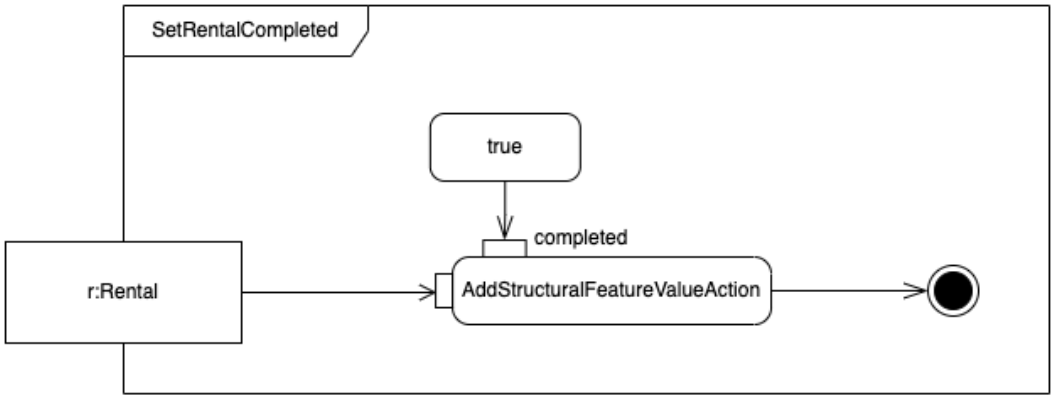


Figure 10.3: pimUML activity diagram of the `SetRentalCompleted` operation of Rent-A-Car

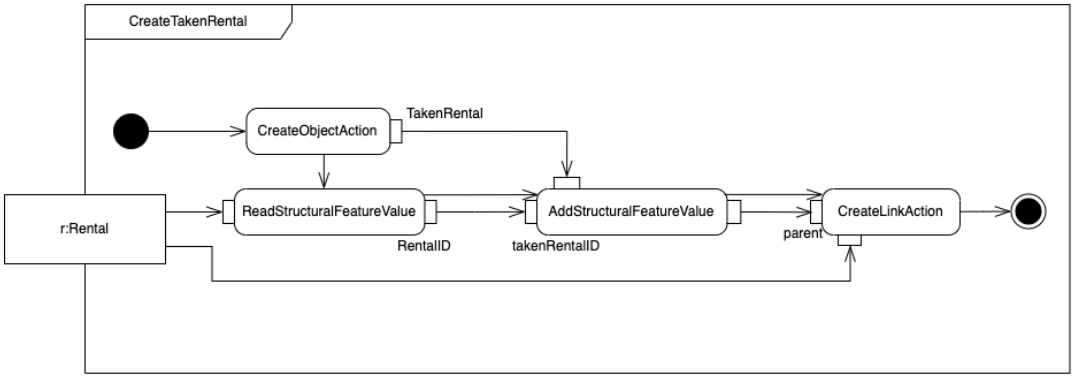


Figure 10.4: pimUML activity diagram of the `CreateTakenRental` operation of Rent-A-Car

eration call by the entry activity of the state machine of the `Rental` object. The first action executed is the `CreateObjectAction`, which creates a new instance of `TakenRental`. The next action is a `ReadStructuralFeatureValueAction` which reads the `RentalID` of the parameter `Rental` object and passes it to the next action, the `AddStructuralFeatureValueAction`. This action also receives the newly created `TakenRental` object and assigns the value of `RentalID` to the attribute `takenRentalID` of `TakenRental`. Finally, the `CreateLinkAction` is executed to create the `parent` link between the `TakenRental` object and the parameter `Rental` object. After this step, the operation is finished.

10.3 Evaluation

The CIM-to-PIM transformation design from DEMO FM to pimUML was quantitatively evaluated by assessing the semantic completeness and correctness of the transformation of the Rent-A-Car case.

10.3.1 Model Semantic Completeness

The semantic completeness of the DEMO FM to pimUML transformation was evaluated against the Rent-A-Car case. First, the domain semantics captured in the DEMO fact model were extracted from the model elements and listed as domain fact statements through the *verbalization* process demonstrated in [33]. These facts were taken as the ground truth of the domain. Second, each fact statement was checked to determine if the semantics were adequately captured and communicated at the PIM level. The number of correctly preserved fact statements of the pimUML PIM was aggregated for each of the major model concepts (see Appendix G for the full results). The aggregated results are presented in Table 10.2.

DEMO FM Concept	# fact statements in DEMO FM	# preserved statements in pimUML
Entity Types	12	12
Value Types	3	3
Event Types	14	14
Attribute Types	42	42
Property Types	42	42

Table 10.2: Semantic completeness of the pimUML PIM against the Rent-A-Car case study

All fact statements were deemed to have been sufficiently preserved in the pimUML PIM.

10.4 Conclusions

The CIM-to-PIM mapping from the object fact diagram of the DEMO fact model to a novel UML profile, *pimUML*, was presented, demonstrated, and evaluated in this chapter. This transformation constitutes the first stage in the MDA approach designed in this thesis. The demonstration and evaluation results showed all fact statements from the domain semantics of the CIM were preserved within the semantics of the produced PIM model elements, confirming that a CIM-to-PIM transformation from the DEMO fact model to pimUML can be used to bridge the enterprise domain semantics to software design semantics with a high degree of semantic correctness. The next chapter details the final agile sprint in this study: the PIM-to-PSM transformation from pimUML to Mendix.

CHAPTER 11

AS4: pimUML to Mendix

The previous four chapters focused on designing a CIM-to-PIM mapping from the DEMO FM, resulting in a novel UML profile, called *pimUML*, and a transformation specification from the DEMO fact model (FM) to pimUML. While multiple agile sprints and a hardening sprint were undertaken to explore different possibilities to achieve the CIM-to-PIM transformation, this chapter focuses on a single sprint to design the PIM-to-PSM mapping from pimUML to Mendix. The designed transformation specification is then demonstrated and evaluated with an academic case study.

11.1 Design

The target platform of the PSM is Mendix, a low-code development platform (LCDP) from Siemens. As a LCDP, Mendix applications are built primarily using graphical notation to implement various components across the layers of the application's architecture.

11.1.1 Step 1: Setting the mapping context

Mendix applications are built using four major components to implement the application presentation logic, business logic, and persistence logic:

- Domain Model
- Microflows
- Workflows
- Pages

Each of these components have their own individual metamodels which are interlinked via relations between common metaclasses. The four components are discussed in detail in the subsections below.

Domain Model

The domain model visually specifies the data of the application; specifically, the domain entities, their attributes, and the relationships between entities. The domain model itself is an abstract visualization of the data model of the application, which is implemented in an underlying (relational) database schema. There are three different types of entities in a domain model: persistable, non-persistable, and external. Objects of non-persistable and external entities are stored in memory at runtime, while objects of persistable entities are stored in the built-in HSQL database, the cloud-based PostgreSQL database, or a supported external database management system. In addition to entities and relationships, annotations can also be included in domain models. Annotations have no functional purpose; they simply serve to provide supplementary information that may be useful to anyone who is trying to understand a given domain model [104].

Microflows

Application logic is primarily realized and implemented in Mendix using microflows. This includes creating, modifying, and deleting objects, as well as application flow decisions, such as user interface (UI) navigation. Microflows are visualized as flowcharts using notation which is based on the Business Process Model and Notation (BPMN). As such, microflow elements consist of parameters, events, flows, decisions, and activities. Microflows start with a start event, end with an end event, and perform one or more activities. The flow through the microflow is guided by sequence flow arrows and control constructs, such as decision nodes (which make decisions through evaluating conditions) and merge nodes [105]. Microflow expressions in Mendix are key to implementing many business logic computations, such as mathematical calculations, as well as to modifying objects of the domain model, such as assigning attribute values and creating links between entity objects [106]. There also exist several types of activities which can be used to perform application operations, such as object activities for manipulating objects and integration activities for calling external services [107].

Workflows

For long-running application processes in Mendix, workflows are used [108]. Processes typically handled by workflows are those which are well-defined, are repeatable, and are frequently executed. Such processes may include both human tasks and automated tasks. As such, workflow processes may take a long time to complete. Workflows typically can run from hours to even months [109]. Therefore, the states of these processes are persisted [108]. Workflows themselves do not perform application operations; rather they

can be used to coordinate the execution of microflows and the assignment of human tasks which perform actions delegated by the workflow [110].

Workflows are assigned an entity to receive as a parameter on which to work, known as the "WorkflowContext". Workflows consist of *activities* which each serve various purposes in the workflow. For example, the *wait for notification* activity instructs the workflow to pause its execution until the notification of an event occurrence arrives. There are also *system action* activities, which instruct the workflow to execute some other operation, such as a microflow or another workflow [111].

Pages

Pages are the interactive views that make up the user interface (UI) of Mendix applications. They are primary means by which the information of an application, such as entity objects and the values of their attributes, are presented and modified [112]. The metamodel of pages consists of elements pertaining to the contents, layout, data sources, and functionality of the UI [113].

Since the DEMO fact model mapped primarily to the business logic layer and persistence layer of the software architecture, and not at all to the presentation layer, Mendix pages fell outside of the scope of this thesis and are therefore not explored.

11.1.2 Step 2: Understanding the similarities and differences

The Mendix domain model syntax and semantics are very close to those of pimUML class diagram. However, some class diagram constructs do not have exact matches in the Mendix domain model, including abstract classes, association classes, and association adornments.

Another difference between pimUML and Mendix is with regards to representation of object state. pimUML provides the state machine diagram for explicitly illustrating the behaviours of objects and the states they assume in response to events. While there exists no such state machine view for entities in Mendix, the pimUML state machine does contain key behavioural semantics which should be captured in the PSM. Specifically, these are the state transition trigger events, the transition effect activities, and the state entry activities. The precise execution semantics of the activities are captured in pimUML activity diagrams to prescribe the algorithmic process to realize object state changes.

Activity diagrams in pimUML are very similar to microflows in Mendix. However, a key difference between the appearance of these two constructs is with regards to edges. pimUML activity diagrams consist of both control flows and object flows. Control flows make explicit the stepwise ordering of actions, while object flows make explicit how objects are passed between actions. In Mendix microflows, flow edges exist only to specify the stepwise ordering of actions. Object flows are not needed, because as soon as an object or variable is made available in a microflow instance, either by a parameter or an action, then it is accessible to all subsequent actions in the microflow.

11.1.3 Step 3: Approaching the metamodel of the target modelling language

The metamodel of Mendix was analyzed to determine the elements which could be feasibly used to capture semantics from elements of pimUML. A subset of the Mendix metamodel containing elements relevant to the transformation mappings can be found in Appendix D.

11.1.4 Step 4: Determine the specific mappings

Explanations behind non-trivial mappings presented in Table 11.1 are discussed below.

i. Abstract class

All elements of Mendix domain models must be concrete. Despite this, abstract classes are mapped to entities in Mendix. It is therefore left up to the domain logic to ensure that these entities are not instantiated.

ii. User-defined data type (custom)

User-defined data types do not exist in Mendix. These are therefore implemented as entities in Mendix.

iii. Property (`isID == true`)

The `isID` meta-property of Properties in pimUML denotes a property that acts as the identifier as a class. In pimUML, these are always integer values; therefore, they are translated to Mendix as Attributes with the IntegerAttributeType as the type. The AutoNumberAttributeType is also a suitable data type for identifiers in Mendix; however, in order to allow for the possibility for the entities to receive an identifier that is domain-relevant, the IntegerAttributeType is used instead.

iv. State Machine

In pimUML, a state machine is instantiated upon the instantiation of its context object. The state machine listens for events and invokes operations in response to both these events and the state changes which result from triggered transitions. The key aspect of the state machine itself in this translation is the mechanism of listening for events that concern the machine's context object.

A state machine is created as a controller of object behaviour in response to events concerning the object. To realize this in Mendix, a microflow is invoked upon object creation. These kinds of microflows in Mendix are known as "After Create" microflows and are identified with the prefix "ACR_" in the name of the microflow.

v. ValueSpecificationAction + AddStructuralFeatureValueAction

The effect of these two actions is to change the attribute value of an object specified by the `AddStructuralFeatureValueAction` to the value specified

by the `ValueSpecificationAction`. This action is achieved in Mendix as a ChangeObjectAction in a microflow, whereby the action is provided the following parameters: the object, the attribute to be modified, and the value that is to be assigned.

vi. AddStructuralFeatureValueAction

The remaining `AddStructuralFeatureValueActions` that were not translated in the mapping described in Definition v. can be deduced (when looking at the pimUML metamodel) to be those actions that receive as input an object and the value of a variable. The value of an attribute is passed from an `OutputPin`, rather than a constant specified by a `ValueSpecificationAction`. In this case, the `AddStructuralFeatureValueAction` is translated to Mendix as a ChangeObjectAction that is instructed, using an expression, to assign the specified attribute of the given object the value of the microflow variable that represents the attribute provided by the output pin.

vii. CreateLinkAction

This action object, creating a link between two objects, is achieved in Mendix as a ChangeObjectAction in a microflow, whereby the action is provided as parameters the association link to be created and other the object that is to be linked.

viii. ReadStructuralFeatureValueAction

In a Mendix microflow, to read the value of an attribute of an object, a variable must be created during an execution instance of the microflow, and the value of the attribute is assigned to this variable. This is achieved with the CreateVariableAction in Mendix. This action is provided an expression string as a parameter that directs the action to receive the value of the identifier attribute of the entity object that was provided as a parameter. For example, for accessing the identifier of a "Membership" entity object, the expression would be "`$Membership/MembershipID`".

ix. Transition + Trigger

A state machine transition and its trigger contain valuable execution semantics regarding what action should be undertaken in response to some domain event. In Mendix, workflows can be used to take on such responsibility. While microflows are meant for short-lived operations, workflows of entities can be active indefinitely as they await the fulfillment of their steps to move them along. To realize the trigger of a transition in a pimUML state machine, Mendix workflows can be instructed wait for a specific event to occur using a *Wait For Notification Activity*. After such an event occurs, a *Call Microflow Activity* is used to invoke a microflow that implements the logic specified by an activity diagram to which the transition's effect behaviour expression refers.

x. State entry activity

States in pimUML use entry activities to specify operations which are to be performed upon entering a new state. In the case of pimUML, the entry activities are created to specify the operation that is to be performed to create a new instance of the child entity that is derived upon the occurrence of a domain event concerning an instance of the parent entity. The exact execution semantics of these activities are expressed in an activity diagram. In the state machine, the entry activity expression is given in the form of an operation call of the activity, expressed in Alf. To capture this behaviour, the action of calling this operation is translated to Mendix as a *Call Microflow Activity* that invokes the microflow which implements the referenced activity.

11.1.5 Step 5: Determining the complete mapping

Table 11.1 presents mappings from elements of pimUML to Mendix, with an ID code for each mapping.

11.2 Demonstration

In this section, the PIM-to-PSM mapping presented in Table 11.1 is demonstrated using the Rent-A-Car (RAC) case study (see Section 6.4). This demonstration constitutes a demonstration of the entire MDA transformation specification of this thesis, from the CIM expressed using a DEMO fact model to the PIM expressed using pimUML (see Section 10.2) and from the PIM to the PSM expressed as Mendix executable model artifacts.

11.2.1 RAC Mendix Domain Model

The Mendix domain model of Rent-A-Car is presented in Figure 11.1. Each entity of Rent-A-Car is represented as an entity in the Mendix domain model, translated from classes at the PIM level. The value classes `{YEAR}` and `{DAY}` become entities as well, translated from custom data types at the PIM level, while `{CAR GROUP}` becomes an enumeration, which is also an enumeration at the PIM level.

As no aggregation adornments exist for associations in the Mendix domain model, the association between `Year` and `CarGroupByYear` is implemented as a regular association.

There also exists no such construct for association classes in Mendix. Therefore, association classes are implemented as regular entities in the Mendix domain model via transformation mapping rule B5. The association that the association class represents is implemented as two separate associations: one from the owning entity to the entity representing the association class and one from the entity representing the association class to the other entity.

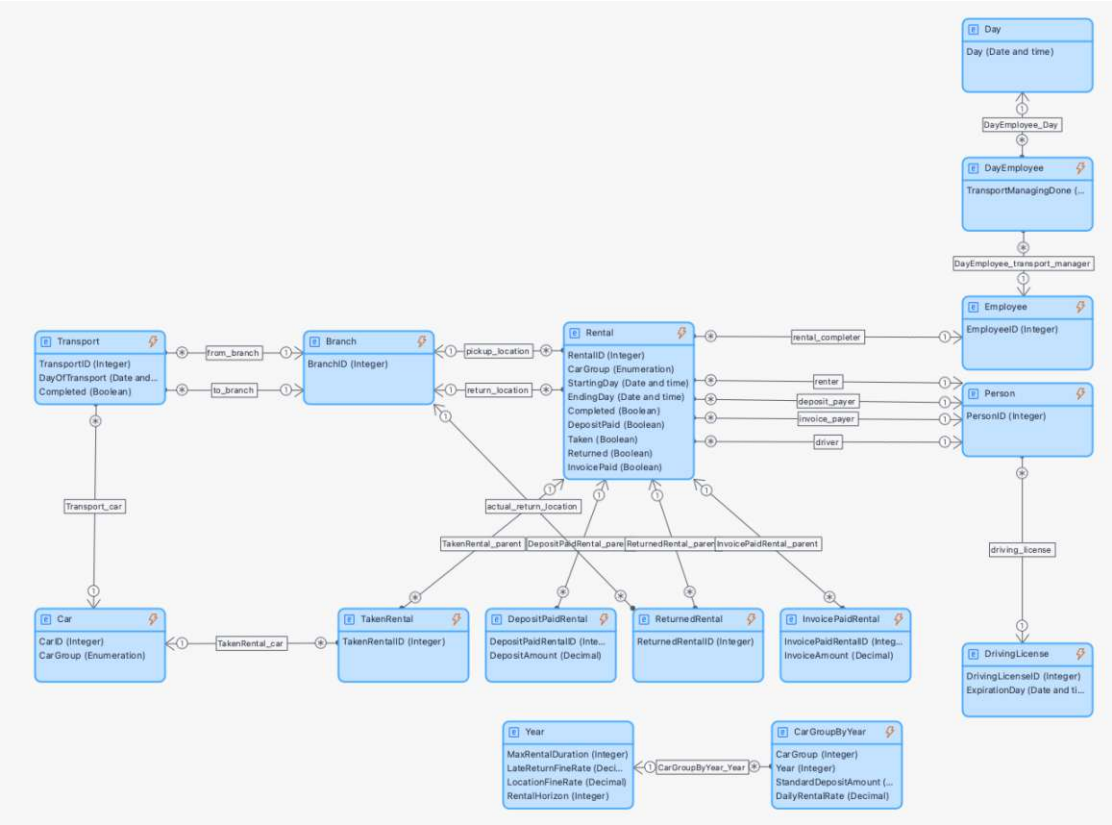


Figure 11.1: Mendix domain model of Rent-A-Car

ID	pimUML	Mendix
B1	Class	Entity (domain model)
B2	Abstract Class	Entity (domain model)
B3	Enumeration EnumerationLiteral	EnumerationAttributeType (domain model) Enumeration (domain model)
B4	User-defined data type	Entity (domain model)
B5	Association Class	Entity (domain model) Association [domain] (domain model) Association [range] (domain model)
B6	Property (`isID == true`)	Attribute Integer (domain model)
B7	Property (of Datatype)	Attribute (domain model)
B8	Property (of Class)	Attribute (domain model)
B9	Association	Association (domain model)
B10	Generalization	Generalization/Specialization (domain model)
B11	State Machine	ACR Microflow (domain model + microflow)
B12	Region Pseudostate (initial)	Call Workflow activity (microflow) Workflow (workflow) Start element (workflow)
B13	Activity	Microflow (microflow)
B14	InitialNode	StartEvent (microflow)
B15	ActivityFinalNode	EndEvent (microflow)
B16	Parameter ActivityParameterNode	Entity reference (microflow) Parameter (microflow)
B17	ValueSpecificationAction AddStructuralFeatureValueAction	Change Object (microflow)
B18	AddStructuralFeatureValueAction	ChangeObjectAction (microflow)
B19	CreateLinkAction	ChangeObjectAction (microflow)
B20	CreateObjectAction	CreateObjectAction (microflow)
B21	ReadStructuralFeatureAction	CreateVariableAction (microflow)
B22	Control flow	Sequence flow (microflow)
B23	Transition Trigger	CallMicroflowTask (workflow) WaitForNotificationActivity (workflow)
B24	State entry activity	CallMicroflowTask (workflow)

Table 11.1: High-level mapping specification from pimUML to Mendix

Each association must have a unique name in a Mendix domain model. Therefore, wherever there are multiple entities with different associations sharing the same name, the entity name is appended to the beginning of the association name, followed by an underscore, in order to distinguish the associations.

11.2.2 RAC Mendix Workflows

To implement the behaviour of entities in response to business events, for each transition in a pimUML state machine at the PIM level, the triggers, transition effect behaviours, and entry activity expressions are translated via transformation rules B23 and B24 to Mendix workflow elements. Figure 11.2 shows the workflow translated from the `completed` region of the state machine in Figure 10.2. This workflow consists of two workflow activities. The first is a *Wait For Notification Activity* which listens for the notification of the accept C-event of `transaction kind 01` for an instance of `Rental` – this constitutes the state transition trigger. The second workflow activity responds to this event by invoking a microflow to set the `completed` attribute of the `Rental` object to true (this microflow is shown in Figure 11.5). This constitutes a transition effect behaviour in a state machine in the PIM.

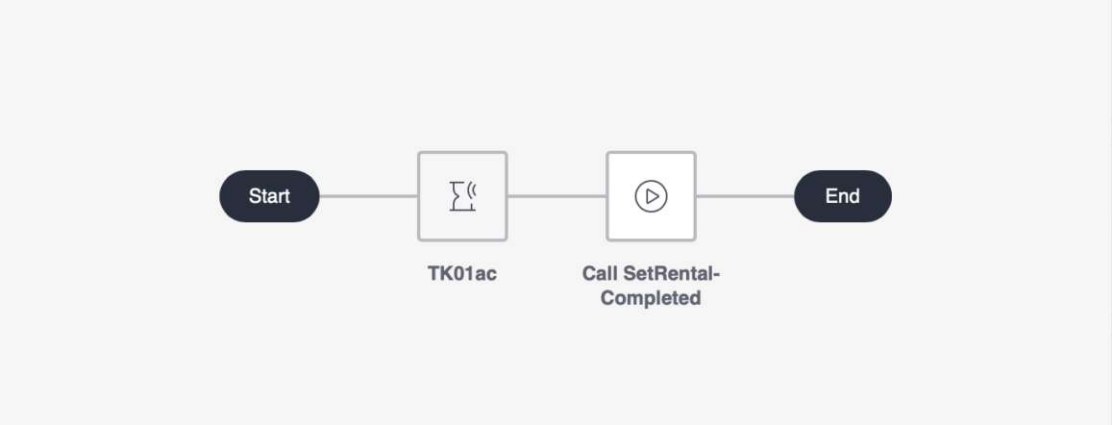


Figure 11.2: Mendix workflow to listen for the Rental "`completed`" event of Rent-A-Car

Figure 11.3 shows the workflow translated from the `taken` region of the state machine in Figure 10.2. This workflow consists of three workflow activities. The first is a *Wait For Notification Activity* which listens for the notification of the accept C-event of `transaction kind 02` for an instance of `Rental` – this constitutes the state transition trigger in the PIM. The second workflow activity responds to this event by invoking a microflow to set the `taken` attribute of the `Rental` object to true (this microflow is shown in Figure 11.5) – this constitutes the transition effect behaviour in the PIM. The third workflow activity invokes the microflow to create a new instance of the `TakenRental` entity, derived from the `Rental` entity – this constitutes the entry activity of the `true` state in the PIM.

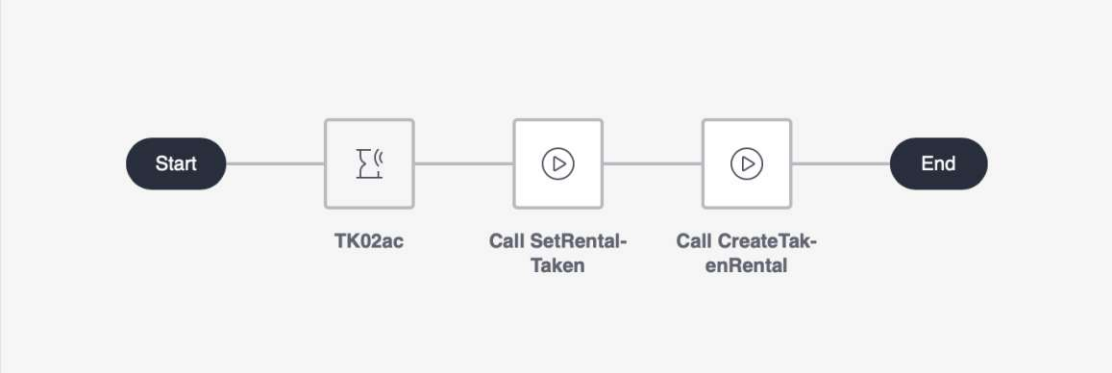


Figure 11.3: Mendix workflow to listen for the Rental "taken" event of Rent-A-Car

11.2.3 RAC Mendix Microflows

To implement the operations specified by the activity diagrams at the PIM level, Mendix microflows are used. These microflows implement the two primary roles of the behaviours specified at the PIM level: to realize state transitions and to implement CRUD operations[1].

According to the semantics of pimUML, a state machine is started immediately upon the instantiation of the class of which the state machine describes the behaviour. In Mendix, this is implemented with an *after create event handler*. This is a microflow that is specifically assigned to be invoked upon the instantiation of an entity. Figure 11.4 shows the `ACR_Rental` microflow. This workflow consists of five `CallWorkflowActivitys`; one for each of the regions of the `Rental` state machine diagram in Figure 10.2. These activities each start the workflows which listen for the trigger events of the transitions between the states in these regions, as described in Section 11.2.2.

The transition effect behaviours are implemented as Mendix microflows that have the responsibility to change the Boolean attribute of the entity, representing the occurrence of the associated event type from the DEMO fact model, which is specified at the PIM level as a pimUML activity diagram. Figure 11.5 shows the microflow implementing the `SetRentalCompleted` activity from the pimUML activity diagram. This microflow sets the `completed` attribute of the parameter `Rental` object to true using a *change object* action.

The state entry activities are implemented as Mendix microflows that have the responsibility to create and initialize new objects of derived entities from the DEMO fact model in response to the occurrence of an associated event type; the execution steps of which are specified at the PIM level as a pimUML activity diagram. Figure 11.6 shows the microflow implementing the `CreateTakenRental` activity from the pimUML activity diagram.

[1]It should be noted that as the PIM level in this thesis only captures the semantics from the object fact diagram of the DEMO fact model, there would certainly be additional purposes for microflows to implement behaviours expressed by other DEMO aspect models. These are outside the scope of this thesis.

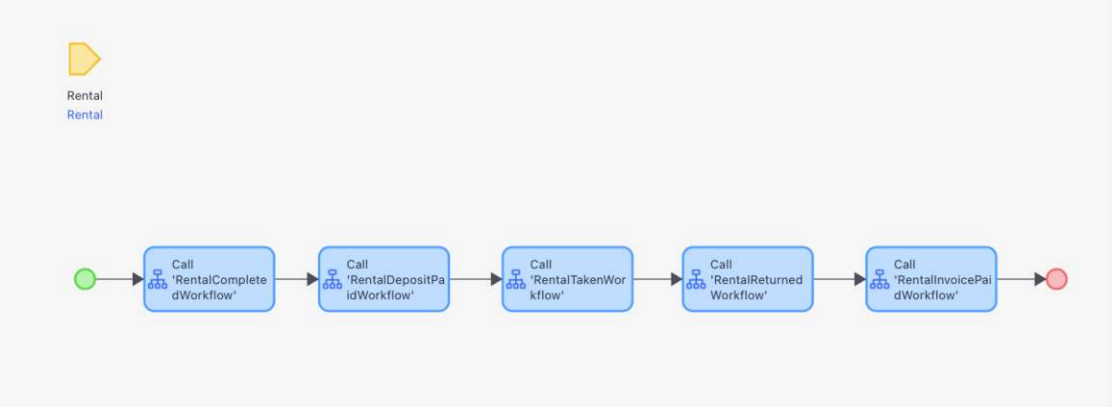


Figure 11.4: Mendix "After Create" microflow to call the event listener workflows of the `Rental` entity of Rent-A-Car

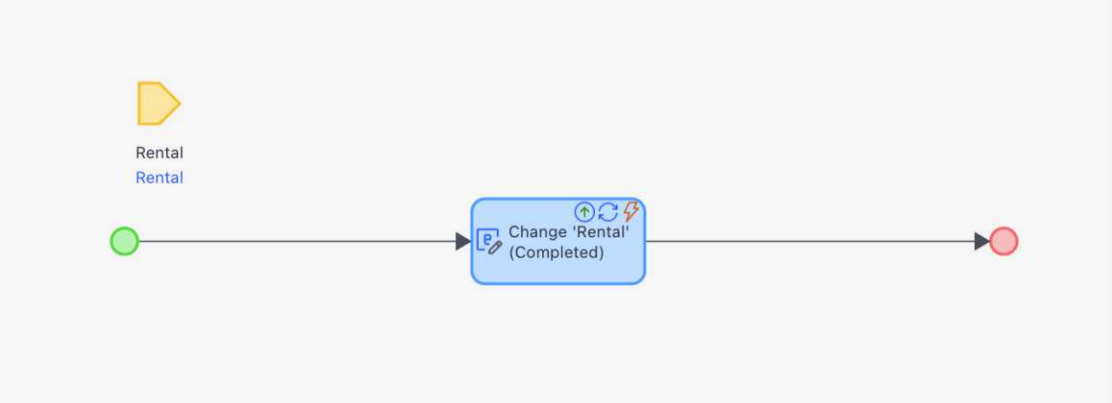


Figure 11.5: Mendix microflow to set a `Rental` object of Rent-A-Car to `completed`

First, the microflow creates a new `TakenRental` object. Next, the microflow creates a new variable to hold the value of the `RentalID` attribute of the parameter `Rental` object. The next step assigns the value of this variable to the `TakenRentalID` attribute of the newly created `TakenRental` object, ensuring that they share the same value thus preserving the *is-a* semantics. Finally, the parent link from the new `TakenRental` object to the parameter `Rental` object is created. After this, the `TakenRental` creation and initialization process is finished.

11.3 Evaluation

The PIM-to-PSM transformation design from pimUML to Mendix was quantitatively evaluated by assessing the semantic completeness and correctness of the transformation of the Rent-A-Car case study.

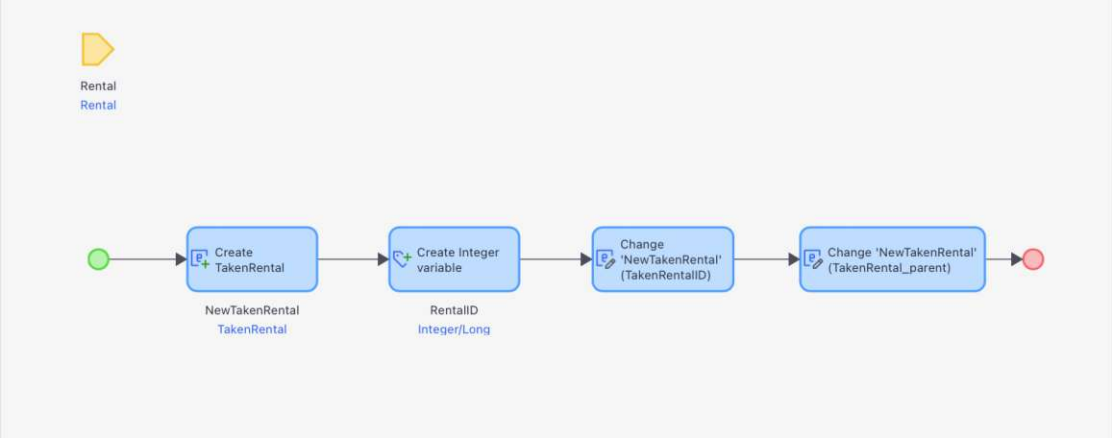


Figure 11.6: Mendix microflow to create a new `TakenRental` object of Rent-A-Car

11.3.1 Model Semantic Completeness

The semantic completeness of the DEMO FM to pimUML and pimUML to Mendix MDA transformation chain was evaluated against the Rent-A-Car case. First, the domain semantics captured in the DEMO fact model were extracted from the model elements and listed as domain fact statements through the *verbalization* process demonstrated in [33]. These facts were taken as the ground truth of the domain. Second, each fact statement was checked to determine if the semantics were adequately captured and communicated at the PSM level. The number of correctly preserved fact statements of the Mendix PSM was aggregated for each of the major model concepts (see Appendix G for the full results). The aggregated results are presented in Table 11.2.

DEMO FM Concept	# fact statements in DEMO FM	# preserved statements in Mendix
Entity Types	12	11
Value Types	3	3
Event Types	14	14
Attribute Types	42	42
Property Types	42	42

Table 11.2: Semantic completeness of the Mendix PSM against the Rent-A-Car case study

The only fact statement that was deemed to have not been sufficiently preserved is *aggregate entity type car group * year exists*. This is due to the fact that Mendix does not have specific construct in its metamodel to denote entities which are composed of other entities, thus constituting an aggregate entity. Instead, the aggregate entity type `{CAR GROUP} * {YEAR}` can be traced from the RAC DEMO fact model to being implemented as a regular entity in Mendix.

11.4 Conclusions

The PIM-to-PSM mapping from pimUML to Mendix was presented, demonstrated, and evaluated in this chapter. This transformation constitutes the second stage in the MDA approach designed in this thesis. The demonstration and evaluation results showed all fact statements from the domain semantics of the CIM were preserved within the semantics of the produced model elements of the target platform, confirming that an MDA transformation from the DEMO fact model to Mendix via pimUML can be used to generate low-code executable artifacts from DEMO ontological enterprise models with a high degree of semantic correctness. The next chapter concludes this thesis report by providing answers to the research questions, summarizing the produced deliverables, detailing the research contributions, discussing study limitations, and suggesting avenues for future research.

CHAPTER 12

Conclusion

The research conducted in this thesis aimed to explore the possibility and benefits of using the MDA approach to improve the process of generating low-code software from enterprise models. Through the chapters of this thesis, the theoretical background was elaborated, the state of the art was explored, foundational definitions were derived through a semi-systematic literature review, a meta-design was formulated, and experimental MDA transformation mappings were designed, demonstrated, and evaluated. To conclude this thesis, answers to the research questions are provided and elaborated, the deliverables are summarized, the contributions to academia and industry are highlighted, the limitations of the research are explained, and work that is left open to future research is discussed.

12.1 Answers to Research Questions

RQ1 *What are the distinct characteristics of the three Model Driven Architecture (MDA) abstraction levels (CIM, PIM, and PSM) and what are their intended uses in the context of Model-Driven Engineering (MDE)?*

The semi-structured literature review in Chapter 4 revealed empirical characteristics of the three abstraction levels of the MDA framework (see Table 4.2). These characteristics pertain to four key aspects on which the MDA abstraction levels can be differentiated:

- Concerns
- Views
- Modelling languages
- Users

While the CIM has the primary use of stakeholder communication and the PSM has the primary use of executable artifact generation, the PIM equally has the dual purpose of being used both for communicating designs among technical stakeholders and for generating executable artifacts for one or multiple target platforms. This was a key insight that was considered while designing the CIM-to-PIM transformation during the design sprints.

RQ2 What concepts or constructs of CIMs, PIMs, PSMs can be identified to guide the usage of MDA for the generation of enterprise applications?

The process of constructing a conceptual framework provided a means of structuring the notions of the three major background field of this thesis – enterprise ontology, model-driven engineering, and low-code development – in like terms to understand how they can be used together in a common framework. No such framework existed in the literature; nevertheless, it was important to have such a framework as to ensure that the PIM artifact produced in this thesis would be truly platform independent and thus generalizable.

Drawing on notions of *enterprise application architecture*, it was chosen to structure the MDA conceptual framework in terms of the four-layered architecture pattern: the *presentation layer*, the *business logic* layer, the *persistence logic* layer, and the *data source logic* layer. By structuring the MDA framework in terms of these four layers, it became clear that each of these layers should be targeted at the PSM level in order to produce an enterprise application. This provided the *structure* of the conceptual framework.

Drawing on the notions of *conceptual schema-centric development* gave important insight into what information should be expected to be captured at the CIM level and what information is feasible to represent at the PIM level. This made it clear that the CIM level can only explicitly provide information of the business logic layer of the enterprise application. This includes business domain semantics such as business vocabulary, business entities, business rules, and business events. The PIM level expands the range of semantics capable of being captured, with information being able to be explicitly extracted from the CIM at the business logic layer while design decisions can be made at the persistence logic layer through inference. Such information includes data processing algorithms to handle business processes and a domain model to organize information on domain entities at the business logic layer, while data and data integrity rules are managed at the persistence layer. Only at the PSM level can information pertaining to the presentation logic and data source logic be captured, such as user interface design and database construction. Added across the matrix of abstraction levels the architecture layers, these are the *concepts* of the conceptual framework, and thus, the concepts that must be captured by CIMs, PIMs, and PSMs to generate enterprise application artifacts with the MDA approach.

The resulting conceptual framework in Chapter 5 allowed it to be deduced that the constructs captured by DEMO and Mendix make these models suitable for

use as a stand-alone CIM and a stand-alone PSM, respectively. Reflecting back on the MDA transformation design sprints, using the DEMO fact model as the CIM constituted a mapping to the persistence logic layer and part of the domain logic layer of the PIM. Semantics and design decisions and how these are made to capture the necessary information at the presentation logic and data source logic layers fell outside of the scope of this thesis.

RQ3 *What architectural principles can be inferred from the design sprints to the guide the design of a PIM to build low-code enterprise applications?*

As this thesis served as an exploratory study into the use of the MDA approach to generate low-code application artifacts from ontological enterprise models, over the course of developing the conceptual framework in Chapter 5 and executing the experimental design sprints through Chapters 7, 8, 9, 10, and 11, several insights were revealed which can serve to guide the development of CIM-to-PIM transformations in the future, such that they capture and propagate structural, behavioural, and execution semantics necessary for the MDA approach to be used for both analysis and executable artifact generation of low-code enterprise applications. These findings are generalized and expressed below as the following architectural principles:

AP01 The PIM should capture structural and behavioural semantics on the domain logic (business logic) from the CIM and the CIM-to-PIM transformation should be able to infer execution semantics to realize the structural and behavioural semantics.

Assuming the CIM contains the necessary domain semantics, not only must the PIM be able to interpret these the explicit domain semantics in a computational sense, but a PIM should also have precise enough execution semantics such that implementation at the PSM level is straightforward. Design decisions regarding the data source layer and the presentation layers will require knowledge about the features of a target platform. According to [83], presentation logic cannot be captured by a conceptual schema, and thus, it is not captured by the PIM. However, the presentation logic is influenced by the conceptual schema and design decisions can be expected to be inferred based on information contained in the PIM. Therefore, presentation logic is added at the PSM layer.

AP02 The PIM should add in or infer design decisions about the persistence logic layer.

The persistence logic layer of the PIM should introduce functionality to ensure data integrity when new business entity instances are created, modified, or deleted. The explicit specification of CRUD operations realizes this functionality. When appropriate (see AP03), data integrity constraints may be implemented and enforced by CRUD operations. These shall exist at the persistence logic layer. This relates to AP01, as execution semantics are added to ensure that the structural semantics captured by the domain model at the

CIM level are preserved throughout the runtime of the application (i.e., over the course of the lifecycles of the object instances).

AP03 Business rules should be enforced at the domain logic layer; data integrity should be enforced at the data source level.

According to the enterprise ontology PSI theory [33], there exist two kinds of business rules: declarative rules and imperative rules. Declarative rules are constraints which apply to the creation and existence of facts in the enterprise ontology. Imperative rules are the realization of declarative rules in the terms of procedures or protocols. Therefore, although they are two different kinds of rules, each imperative business rule has a declarative counterpart. Enforcement of business rules must be appropriately captured and implemented.

The notions of imperative and declarative business rules also align closely with the notions of *behaviour business rules* and *definitional rules* as defined by the *Semantics of Business Vocabulary and Business Rules* (SBVR) specification published by OMG [32]. The key difference between such rules is that behavioural business rules allow for a certain degree of free will on the part of the executor, whereas definitional rules do not. It is important that these types of rules are distinguishable in the CIM level, as this difference has major implications for how they are enforced at the PIM and PSM levels.

Enforcing rules from a declarative standpoint could be realized by persistence level constraints on CRUD operations, whereby any deviation is disallowed. However, the PSI theory also acknowledges that sometimes business rules are broken for good reasons. If they were broken for improper reasons, they should be corrected at the promise state of the transaction [33]. Therefore, the imperative business rules could instead be implemented as inherent effects of the implemented procedures to which the business rules apply[1].

AP04 When choosing notation elements, domain semantics take precedence over modelling language semantics.

This relates to the findings of the characteristics of the PIM; not only should a PIM be capable of producing an executable PSM or code, but it should also express, in a human readable manner, the behavioural and structural semantics of the application for the purpose of analysis. This principle aligns closely with the xUML design principles from [89]. Balancing the dual purpose of the PIM for human communication and executable artifact generation, at times, trade-offs had to be made when choosing modelling elements based on how well they captured the semantics to best fulfill these purposes. For example, *is-a* semantics of generalization *versus* specialization relationships between entities must be propagated to the target platform at the PSM level and must also be clear and understandable to software analysts reading the diagrams at the PIM level.

[1] For this reason, in this thesis, any declarative business rules were not considered for translation from the CIM to the PIM at the persistence logic layer.

12.2 Summary of Deliverables

Following the agile design science research methodology (ADSRM), three major deliverables were produced: working definitions of the MDA abstraction levels based on empirical characteristics found in literature, an MDA transformation meta-design, and an MDA transformation specification, including a novel UML profile, from the DEMO fact model to Mendix.

12.2.1 Characteristics and Definitions of MDA Abstraction Levels

Through the semi-structured literature review, a table of the characteristics of models of the MDA abstraction levels was composed. The characteristics in the table were grouped by four major themes: concerns, views, common languages, and users. From those characteristics, working definitions of each of the three abstraction levels were derived. The table of characteristics and the working definition of CIM, the working definition of PIM, and the working definition of PSM are found in Table 4.2, Section 4.2.1, Section 4.2.2, and Section 4.2.3, respectively.

The sets of characteristics and working definitions served to fill the gaps in the understanding of what defines each of the MDA abstraction levels. This was used to better understand the applicability of the MDA approach to solve the research problem and to better set the scope and objectives of the design sprints of this thesis.

12.2.2 MDA Transformation Meta-Design

By synthesizing fundamental notions from enterprise ontology, enterprise application architecture, and MDA into a conceptual framework, an MDA transformation meta-design was formulated. This meta-design structured the information captured by the MDA abstraction levels in terms of the four-layered software architecture pattern. The meta-design is found in Figure 5.1.

By contextualizing enterprise ontology in terms of the layers of enterprise application architecture, it became clearer as to how information from the PIM level can be directly extracted from the CIM level and what information must be inferred. This also allowed for a critical assessment as to the fit of DEMO as a CIM and Mendix as a PSM; moreover, it informed the decision as to what modelling languages are suitable candidates for modelling at the PIM level. Lastly, it also assisted in further refining the objectives and scope of the design sprints.

12.2.3 pimUML and MDA Transformation Specification

The first three design sprints focused on creating CIM-to-PIM mappings from DEMO to each of the following three UML profiles: Standard UML, xUML, and fUML. The strengths and weaknesses of these modelling languages in use as a PIM were assessed, and from these findings, a novel UML profile was derived in a hardening sprint: pimUML.

As a UML profile, pimUML is restricted to include elements that can effectively capture the structural and behavioural semantics of domain entities, as well as the execution semantics of operations concerning those entities. pimUML was used as the PIM modelling language in the two stage MDA transformation specification: from DEMO to pimUML and from pimUML to Mendix. The pimUML metamodel, the graphical CIM-to-PIM transformation mappings, and the graphical PIM-to-PSM mappings are found in Appendix C, Appendix E, and Appendix F, respectively.

Through an evaluation of the semantic correctness of the information modelled in the Mendix PSM with regards to the domain facts captured by the DEMO CIM, the MDA transformation was shown to effectively transform DEMO models to pimUML models and then to Mendix executable artifacts with a high degree of semantic correctness.

12.3 Research Contributions

The research presented in this thesis in creating a novel MDA transformation specification from DEMO to Mendix benefits academia by furthering the knowledge of how the MDA approach can be applied to bridge the gap between enterprise ontology and low-code application development. Likewise, it benefits industry by offering a new way to use MDA to realize the flexibility of designing low-code enterprise applications at multiple levels of abstraction, making it easier to interchange the target platform for which executable artifacts are generated to support business processes. Not only can this aid in improving business-IT alignment, but it also lessens the impact of common challenges of using low-code development platforms (LCDPs); one of the most prominent of those discussed in this thesis is the challenge of *vendor lock-in*.

12.3.1 Contributions to Academia

The findings throughout this research can be leveraged in future studies involving MDA. The characteristics of the MDA abstraction levels found by answering RQ1 serve to better inform the use of the MDA approach in research, providing guidance on what concerns, views, modelling languages, and users are typical of each level. The MDA transformation meta-design formulated to answer RQ2 provides the structure and an overview of the various concepts which should be captured by CIMs, PIMs, and PSMs, in order to generate executable artifacts for the major architectural aspects of enterprise applications. This also can assist with the choices of modelling languages, views, and design patterns used by each model of the three MDA abstraction levels. Finally, the architectural principles derived from the design sprints to answer RQ3 can be used in future studies looking to explore the use of the MDA approach, particularly as it relates to the design of a PIM.

12.3.2 Contributions to Industry

The deliverables and findings of this thesis can be applied in industry efforts to develop techniques to improve business-IT alignment. Using enterprise ontology and DEMO models as a source in the MDA approach ensures that code artifacts are generated directly from the essential model of the enterprise; as such, this ensures that the enterprise application is built directly from the specifications of the business processes which it is to support. Thus, this approach contributes to improving business-IT alignment.

This approach also contributes to combating two significant challenges hindering the adoption of LCDPs in industry: vendor lock-in and high learning curves. With the design of the software being expressed both at platform-independent and platform-specific levels of abstraction, using the MDA approach to design and build low-code enterprise applications makes it easier to interchange the target platform or to use multiple target platforms for different parts of the application, avoiding the need for the application to be redesigned from scratch using the proprietary modelling languages of individual LCDPs. At the same time, using the MDA approach to develop low-code applications also encourages the collaboration of non-technical stakeholders. As the contribution of non-technical stakeholders can be hindered due to the learning curve associated with low-code platforms [41], the MDA approach can help by allowing non-technical stakeholders to contribute to the project design using the DEMO notation to capture the problem space at the CIM level while technical stakeholders can work with pimUML models on the solution space at the PIM level. This is similar to the benefits of the V-SUM approach by Hermann et al. in [48] (discussed in Section 3.2), and it also aligns with the suggestions found in [41, 39, 42] of using standard modelling notations as part of the design and development of low-code applications to ease the learning curve of using LCDPs. Alleviating these challenges while allowing enterprises to realize the benefits of low-code development, such as faster software development and "democratizing" the development process to involve non-technical stakeholders [40, 37], further contributes to the effort of maintaining business-IT alignment.

12.4 Limitations

There are a few challenges and limitations of the thesis which pose threats to the validity of the results.

First, the use of the DEMO fact model as a CIM model and thus as a source metamodel was difficult due to inconsistencies and omissions in the metamodel itself. Many such inconsistencies and omissions in the DEMO metamodel have been reported in the PhD thesis of Mulder [54]. For example, despite the DEMO-SL specification [35] making clear that various fact types have names, fact type names are not explicitly included as a meta-property in the graphical metamodel of the DEMO fact model. This information therefore had to be assumed to be present in the metamodel for use in the transformations. Also, although the metamodel does not permit the association between an event type and a value type, such a construct appeared in the Rent-A-Car case study; this construct

had to be accounted for in the transformation designs in order to satisfy the semantics of the case study. Similarly, due to the aim of DEMO being to capture the *essential* model of the enterprise, some attributes which are not pertinent to the business process and business rules are omitted from the fact model. If these are desired in an information system, they will need to be added manually at the PIM level.

Second, as the transformations in the case study were conducted manually, this had an increased risk of modelling mistakes being made; however, this risk was mitigated through modelling the case demonstrations over multiple iterations during each sprint.

Lastly, due to the outputted Mendix artifacts not constituting a full application, the runtime functionality of the Mendix artifacts was not validated nor verified. Such a task is left open for future work.

12.5 Future Work

There are three key aspects of this research which could be further explored in future research. The first is the potential use of the remaining DEMO aspect models; the second is with regards to varying the target platform of the PSM, and the third is with regards to the concrete implementation and automation of the MDA transformation specification.

12.5.1 Expansion of CIM and PIM

This thesis had a narrowly focused scope on applying the Object Fact Diagram of the DEMO fact model in the MDA transformation approach. The usage of the DEMO fact model was a sensible first step in exploring the use of the MDA approach to generate low-code applications from DEMO, as the entities of the conceptual model must first be captured in the system before those entities can be acted upon in business processes. However, there is still much to explore with regards to the usage of DEMO as a CIM. As textual transformations were not considered in this thesis, additional fact model information captured as derived fact specifications were excluded from the transformation designs. The use of these as part of the CIM should be explored in future research. Beyond the fact model, there are also the other aspect models of DEMO, as well as the transaction pattern constructs, containing highly valuable domain information on business rules and business processes, which is crucial for realizing a semantically rich PIM. These should also be explored for use at the CIM level in future research.

With addition of other DEMO aspect models at the CIM level, the PIM and PSM could also be enhanced. For example, any automated processes requiring human participation must have a corresponding user interface dialog. Likewise, any information which must be provided as input to human tasks must be displayed to the user. It is therefore important to be able delineate whether a task is to be performed by a human or a machine at the PIM level. Where human input is required, the appropriate user interface methods and constructs should be generated at the PSM level. Capturing these additional design decisions could be achieved possibly by expanding pimUML with additional UML models

and constructs or even adding an additional modelling language into the mix, such as SoaML[2], SysML[3], or IFML[4].

12.5.2 Targeting Different Platforms

One of the most substantial benefits of the MDA approach is the flexibility afforded by the PIM to easily interchange the platform of the PSM, possibly even using different PSMs for different aspects of the application architecture at once. Drawing on the notions of *Ashby's law of requisite variety* from the field of cybernetics [114], to account for the complexity induced by variety on both the side of the enterprise and the side of the technology, a PIM that is truly platform-independent should be able to handle as large a variety of input and to produce as large a variety of desired output as possible, thus reducing risk of vendor lock-in on the side of the PSM and accounting for increased firm agility and changes of requirements on the side of the CIM. To truly test the platform independence factor of the PIM modelled using pimUML, the ability of pimUML to be mapped to different platforms – potentially both low-code and high-code platforms – should be explored. As such, the flexibility of pimUML to be used to target different platforms should be explored and the flexibility afforded should be measured.

12.5.3 Automation of the MDA Transformation

While the transformation mappings designed in this thesis were demonstrated and evaluated manually, automating these designs would not only allow for a more effective evaluation of these transformation designs, but it would also bring them a step closer to realizing value for real-world users. Once the transformations are automated, tested, and refined accordingly, tool support could also be developed, allowing the transformations to be used in real-world projects to assist enterprises in improving business-IT alignment.

[2]Information on SoaML can be found at: `https://www.omg.org/spec/SoaML`
[3]Information on SysML can be found at: `https://sysml.org`
[4]Information on IFML can be found at: `https://www.ifml.org`

APPENDIX A

Semi-Structured Literature Review: Review Protocol

The details of the search protocol design and execution of the semi-systematic literature review conducted as part of answering RQ1 of this thesis are outlined in this appendix.

The major steps followed to conduct the semi-systematic literature review in this thesis, as defined in [61], are outlined below:

Step 1. Designing the review

The study review protocol detailed in Section A.1 provides the research question, search terms, queries, and research databases of the review.

Step 2. Conducting the review

Section A.2 provides the resulting papers found after having conducted the review.

Step 3. Analysis

See Chapter 4.

Step 4. Writing the review

See Chapter 4.

A.1 Review Protocol

A.1.1 Research Question

RQ1: What are the distinct characteristics of the three Model Driven Architecture (MDA) abstraction levels (CIM, PIM, and PSM) and what are their intended uses in the context of Model-Driven Engineering (MDE)?

A.1.2 Search Terms

	Synonyms		
Terms	mda	model driven architecture	model-driven architecture
	cim	computational independent model	
	pim	platform independent model	
	psm	platform-specific model	platform specific model
	approach	method	methodology
	systematic mapping study	systematic literature review	
	transformation	mapping	

Table A.1: Search terms used in queries of the semi-structured literature review

A.1.3 Queries

Q1	**OR**
AND	mda OR model driven architecture OR model-driven architecture
	cim OR pim OR psm OR computational independent model OR platform independent model OR platform specific model
	transform OR transformation OR transformations OR transforming

Q2	**OR**
AND	mda OR model driven architecture OR model-driven architecture
	approach OR method OR methodology
	model
	transform OR transformation OR transformations OR transforming

Q3	**OR**
AND	cim OR pim OR psm OR computational independent model OR platform independent model OR platform specific model
	approach OR method OR methodology
	transform OR transformation OR transformations OR transforming

Q4	OR
AND	cim OR pim OR psm OR computational independent model OR platform independent model OR platform specific model
	transform OR transformation OR transformations OR transforming

Q5	OR
AND	mda OR model driven architecture OR model-driven architecture
	"systematic literature review" OR "systematic mapping study"

A.1.4 Research Databases

The above queries were executed as searches in the following research databases:

- Google Scholar
- IEEE Xplore

A.1.5 Study Selection Criteria

Inclusion Criteria

- Primary study papers *applying* MDA, yielding a new *artifact*
- Primary study papers *augmenting* or suggesting a new approach to MDA transformations, yielding a new *process*
- Secondary study papers *about* MDA (systematic literature reviews, systematic mapping studies, etc.)

Exclusion Criteria

- Papers on MDE but *not* on the MDA process from OMG
- Papers that do not apply MDA to transform models between modelling languages (i.e. code to code transformations)
- Papers which do not provide clear definitions of CIMs, PIMs, or PSMs
- Papers not in English

A.1.6 Study Selection Procedure

1. First review papers supplied by my supervisors
2. Second search for papers in online research databases

3. Read narrowed down list of articles, looking for choices of CIM/PIM/PSMs (and perhaps their characteristics)
4. Review the interesting papers found in the references lists of the previously read papers
5. Summarize findings
6. Draw insights using the *line of argument synthesis* technique

A.1.7 Study Selection Quality Checklist

1. Age of paper
2. Level of detail of characteristics of CIM/PIM/PSM
3. Quantity of interesting citations

A.2 Query Results

This section provides lists of papers found which were selected for use in this thesis. There is a table for each combination of query and research database.

Q1			
DATABASE: Google Scholar			
QUERY: allintitle: (MDA | "model driven architecture" | "model-driven architecture") (CIM | PIM | PSM | "computational independent model" | "platform independent model" | "platform specific model") ("transform" | "transformation" | "transformations" | "transforming")			
Total # of papers found: 36			
ID	**Title**	**Year**	**Citation**
P7	An Approach for Transforming CIM to PIM up To PSM in MDA	2020	[67]
P6	Automate model transformation from CIM to PIM up to PSM in model-driven architecture	2019	[62]
P8	Model Transformation with ATL into MDA from CIM to PIM Structured through MVC	2016	[68]

Q2			
DATABASE: Google Scholar			
QUERY: allintitle: (MDA | "model driven architecture" | "model-driven architecture") ("approach" | "method" | "methodology") ("model") ("transform" | "transformation" | "transformations" | "transforming")			
Total # of papers found: 49			
ID	**Title**	**Year**	**Citation**
P9	An MDA approach to business process model transformations	2010	[15]

Q3			
DATABASE: Google Scholar			
QUERY: allintitle: (CIM | PIM | PSM | "computational independent model" | "platform independent model" | "platform specific model") ("approach" | "method" | "methodology") ("transform" | "transformation" | "transformations" | "transforming")			
Total # of papers found: 40			
ID	**Title**	**Year**	**Citation**
P9	An MDA approach to business process model transformations	2010	[15]
P3	Transformation from CIM to PIM: A feature-oriented component-based approach	2005	[65]

Q4			
DATABASE: Google Scholar			
QUERY: allintitle: (CIM | PIM | PSM | "computational independent model" | "platform independent model" | "platform specific model") ("transform" | "transformation" | "transformations" | "transforming")			
Total # of papers found: 159			
ID	**Title**	**Year**	**Citation**
P2	Transformation from CIM to PIM: A systematic mapping	2022	[16]
P10	Transformation from CIM to PIM using patterns and archetypes	2008	[69]
P11	Applying CIM-to-PIM model transformations for the service-oriented development of information systems	2011	[63]

Q1			
DATABASE: IEEE Xplore			
QUERY: (“Document Title”:“MDA” OR “Document Title”:“Model driven architecture” OR “Document Title”:“Model-driven architecture”) AND (“Document Title”:“CIM” OR “Document Title”:“PIM” OR “Document Title”:“PSM” OR “Document Title”:“computational independent model” OR “Document Title”:“platform independent model” OR “Document Title”:“platform specific model”) AND (“Document Title”:“transform” OR “Document Title”:“transformation” OR “Document Title”:“transformations” OR “Document Title”:“transforming”)			
Total # of papers found: 2			
ID	**Title**	**Year**	**Citation**
P13	A model transformation in MDA from CIM to PIM represented by web models through SoaML and IFML	2016	[71]
P14	Disciplined approach for transformation CIM to PIM in MDA	2015	[72]

Q2			
DATABASE: IEEE Xplore			
QUERY: (“Document Title”:“MDA” OR “Document Title”:“model driven architecture” OR “Document Title”:‘model-driven architecture”) AND (“Document Title”:“approach” OR “Document Title”:“method” OR “Document Title”:“methodology”) AND (“Document Title”:“model”) AND (“Document Title”:“transform” OR “Document Title”:“transformation” OR “Document Title”:“transformations” OR “Document Title”:“transforming”)			
Total # of papers found: 9			
ID	**Title**	**Year**	**Citation**
P15	An Approach for MDA Model Transformation Based on JEE Platform	2008	[73]
P16	Mapping Approach for Model Transformation of MDA Based on XMI/XML Platform	2009	[74]
P17	Mapping approach for model transformation of MDA based on xUML	2009	[75]

Q3			
DATABASE: IEEE Xplore			
QUERY: (“Document Title”:“CIM” OR “Document Title”:“PIM” OR “Document Title”:“PSM” OR “Document Title”:“computational independent model” OR “Document Title”:“platform independent model” OR “Document Title”:“platform specific model”) AND (“Document Title”:“approach” OR “Document Title”:“method” OR “Document Title”:“methodology”) AND (“Document Title”:“transform” OR “Document Title”:“transformation” OR “Document Title”:“transformations” OR “Document Title”:“transforming”)			
Total # of papers found: 5			
ID	**Title**	**Year**	**Citation**
P18	A methodology for transforming CIM to PIM through UML: From business view to information system view	2015	[76]
P19	Transformation approach CIM to PIM: from business processes models to state machine and package models	2015	[77]

Q4			
DATABASE: IEEE Xplore			
QUERY: (“Document Title”:“CIM” OR “Document Title”:“PIM” OR “Document Title”:“PSM” OR “Document Title”:“computational independent model” OR “Document Title”:“platform independent model” OR “Document Title”:“platform specific model”) AND (“Document Title”:“transform” OR “Document Title”:“transformation” OR “Document Title”:“transformations” OR “Document Title”:“transforming”)			
Total # of papers found: 15			
ID	**Title**	**Year**	**Citation**
P20	A set of QVT relations to transform PIM to PSM in the Design of Secure Data Warehouses	2007	[78]
P10	Transformation from CIM to PIM Using Patterns and Archetypes	2008	[69]
P13	A model transformation in MDA from CIM to PIM represented by web models through SoaML and IFML	2016	[71]
P2	Transformation From CIM to PIM: a Systematic Mapping	2022	[16]

APPENDIX B

Metamodel: DEMO Fact Model

This appendix presents the metamodel of the Object Fact Diagram (OFD) of the DEMO fact model (FM). The metamodel is expressed in the General Ontology Specification Language (GOSL)[1].

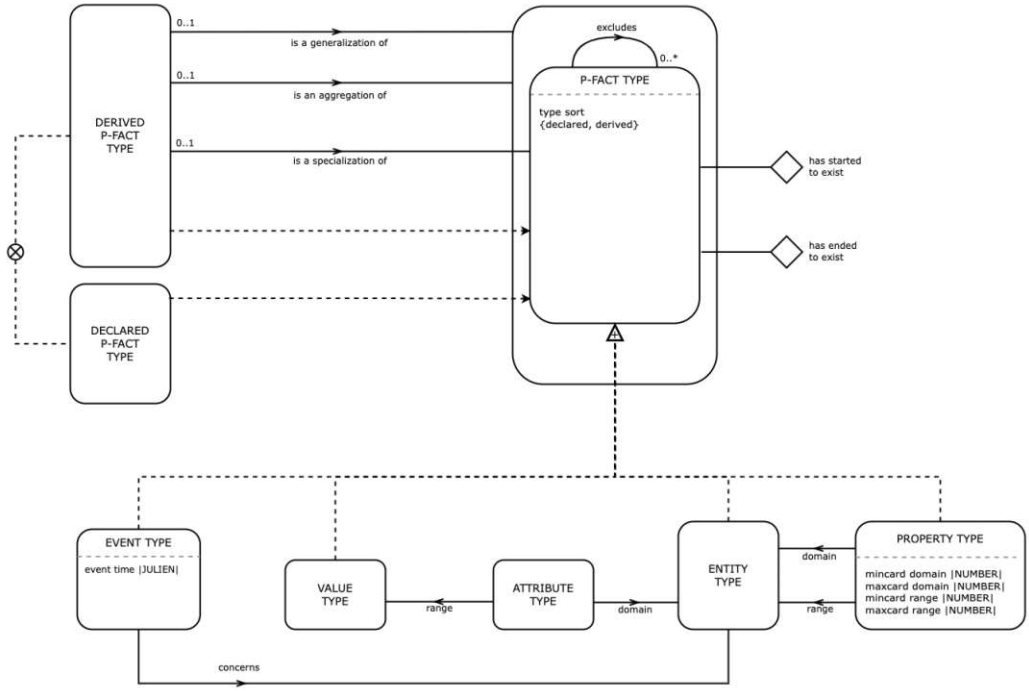


Figure B.1: Metamodel of the DEMO fact model, adopted from [35]

[1]For a reference on the syntax of GOSL and the DEMO metamodel, the reader is encouraged to see [35].

APPENDIX C

Metamodel: pimUML

This appendix presents the metamodel of the novel UML profile of this thesis, *pimUML*. The metamodel is divided into the three major UML diagrams of which pimUML is composed: the class diagram, the state machine diagram, and the activity diagram. The pimUML metamodel is based on the metamodel of UML version 2.5.1, which is available at [88].

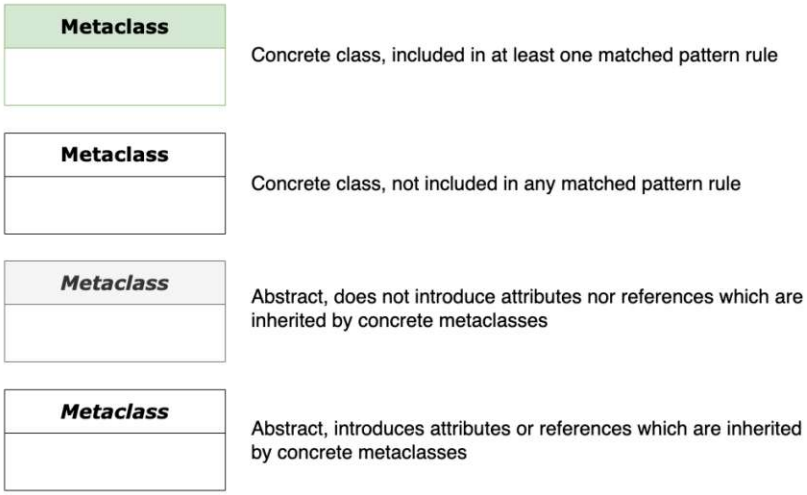


Figure C.1: Legend describing the meaning of different pimUML metamodel element appearances.

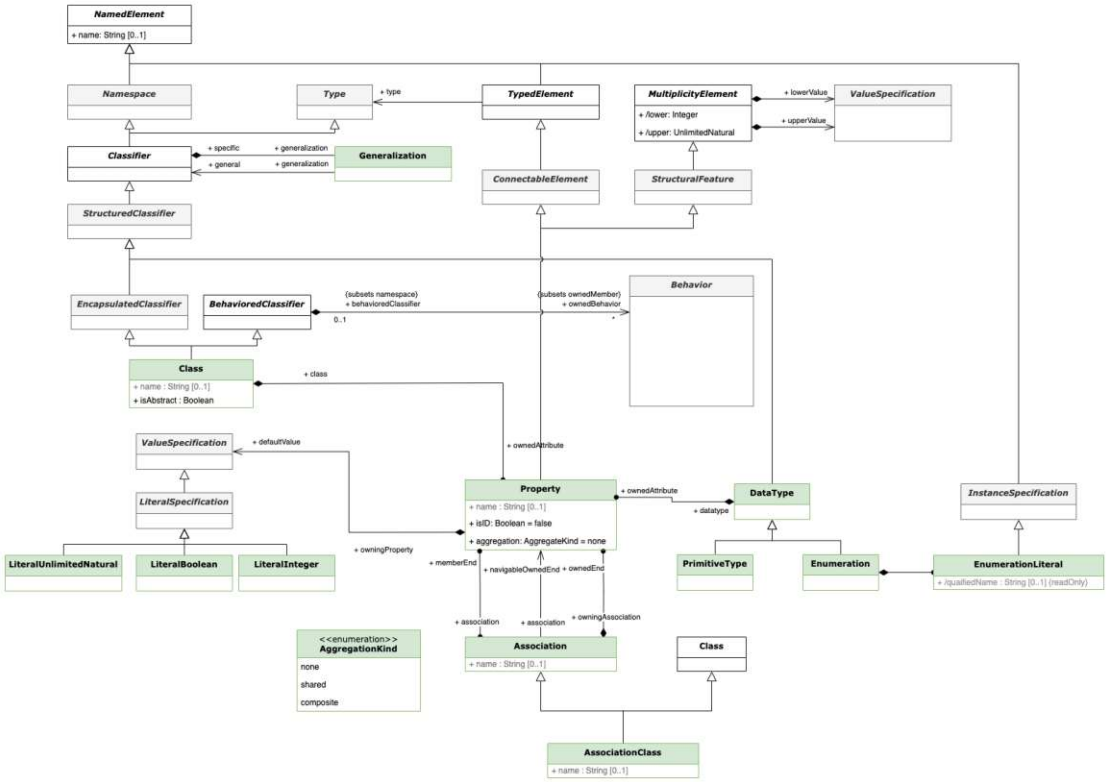


Figure C.2: Metamodel of the pimUML class model

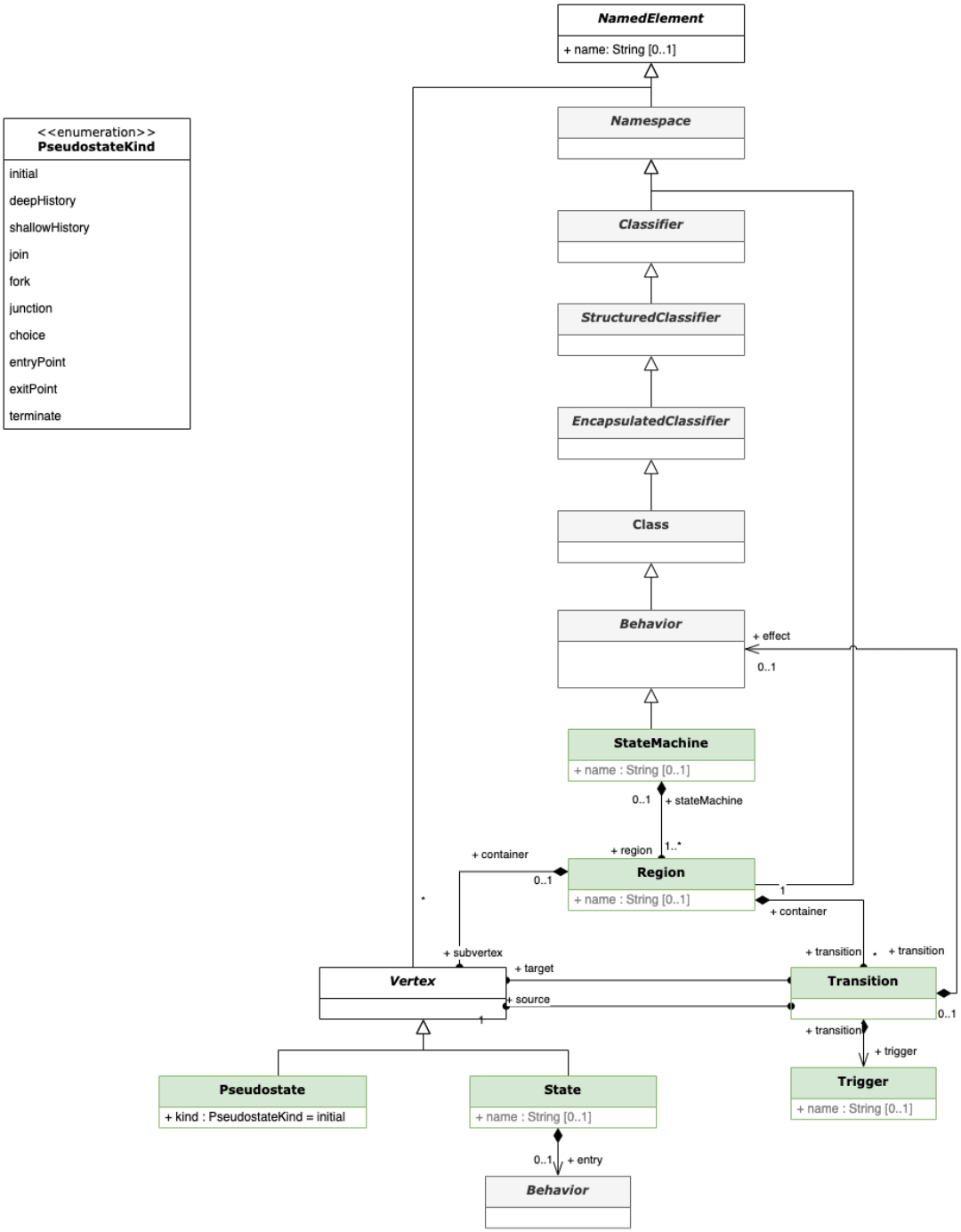


Figure C.3: Metamodel of the pimUML state machine model

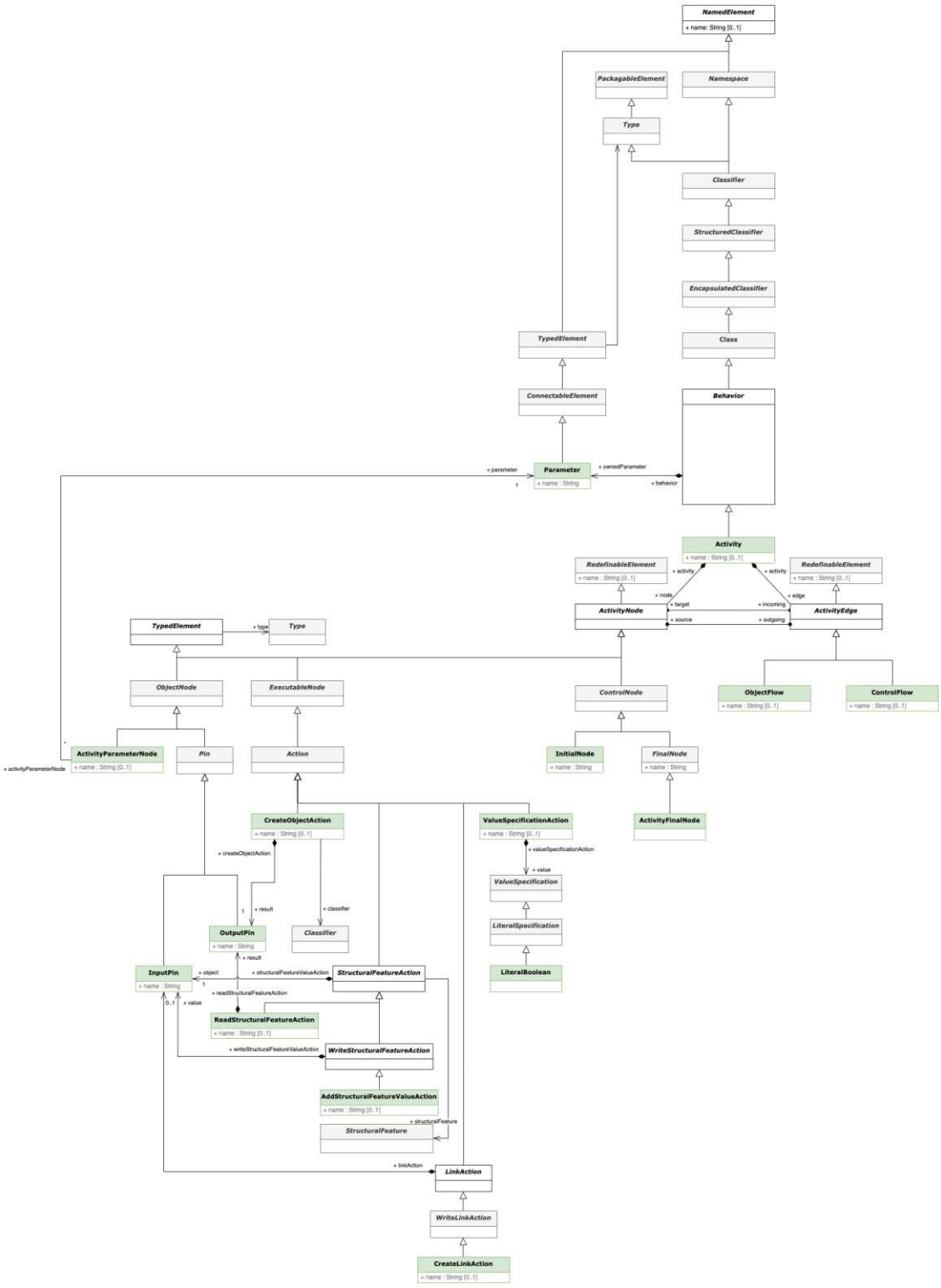


Figure C.4: Metamodel of the pimUML activity model

APPENDIX D

Metamodel: Mendix

This appendix presents partial metamodels of the Mendix domain model, microflow, and workflow. The metamodels of the domain model and microflow are adapted from graphical metamodels provided on the online Mendix documentation, whereas the metamodel of the workflow was graphically adapted based on the Mendix SDK documentation.

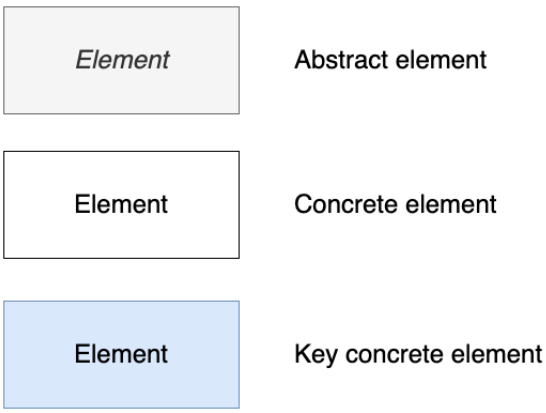


Figure D.1: Legend describing the meaning of different Mendix metamodel element appearances.

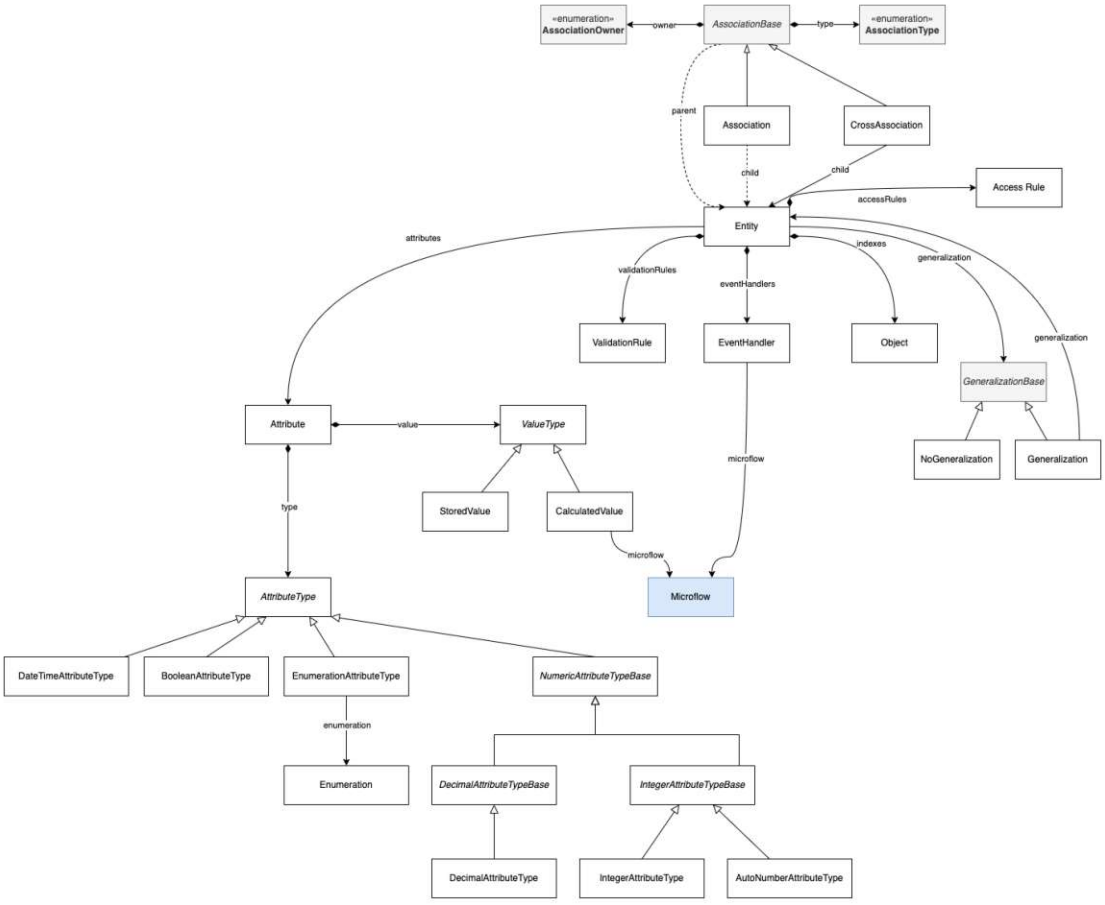


Figure D.2: Partial metamodel of the Mendix domain model, adapted from [115]

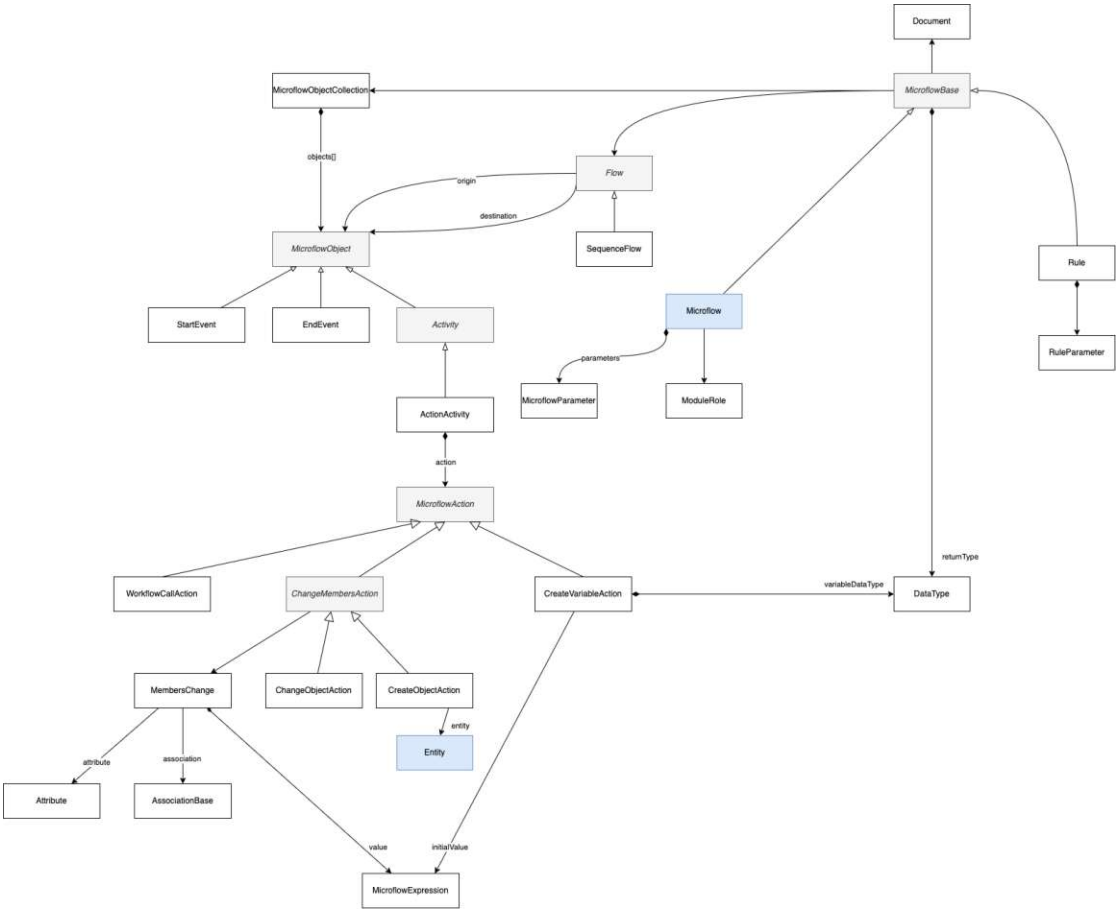


Figure D.3: Partial metamodel of the Mendix microflow, adapted from [116]

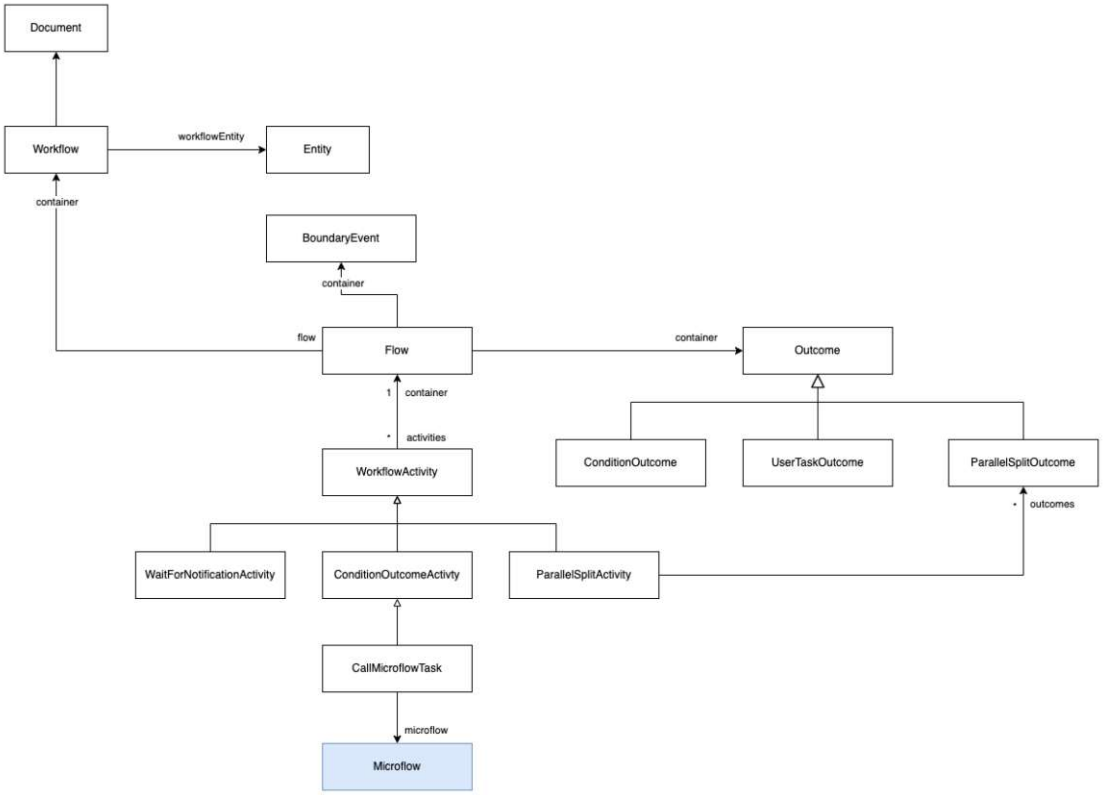


Figure D.4: Partial metamodel of the Mendix workflow, adapted from [117]

APPENDIX E

Transformation Mappings: DEMO FM to pimUML

This appendix provides the details of the CIM-to-PIM transformation specification from the DEMO fact model (FM) to pimUML. There are three key components of the transformation specification: value mapping functions, helper functions, and graphical matched pattern transformation rules. The graphical matched pattern transformation rules demonstrate how various input patterns of DEMO FM modelling concepts, expressed in GOSL graphical notation, are mapped to output patterns of pimUML modelling concepts, expressed in the standard graphical notation of UML. The value mapping functions may appear in elements of the output pattern for various transformation mappings; for example, to map value types from the DEMO FM to data types in pimUML in generated elements containing data types. Helper functions operate on values specified by variables in input patterns to produce new values for use in elements generated in the corresponding output patterns.

E.1 Value Mapping Functions

- $v \mapsto \phi$ in Table E.1 maps DEMO value types (v) to pimUML data types (ϕ)

E.2 Helper Functions

- $+$ denotes a string concatenation operation
- `toCamelCase(x)` reformats an inputted string `x` into camel case format
- `first(x)` returns the first character of an inputted string `x` in lowercase

$v \mapsto \phi$	
v	ϕ
Time	DateTime
Duration	Integer
Amount	Real
Mass	Real
Length	Real
Area	Real
Volume	Real
Velocity	Real
Temperature	Real
Number	Integer
True Value	Boolean
Sort	Enumeration
Custom value type	User-defined data type

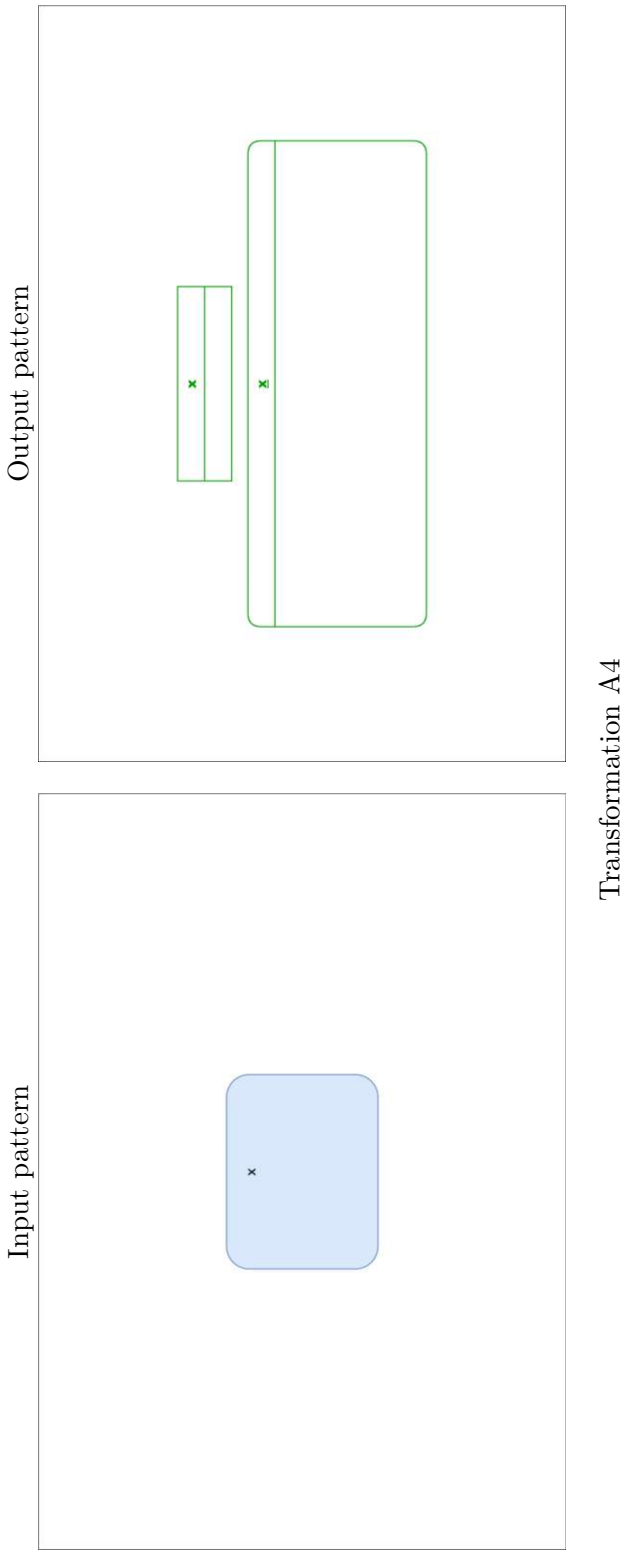
Table E.1: Mapping function from DEMO value types to pimUML data types

E.3 Graphical Matched Pattern Transformation Rules

The following pages provide illustrations of the matched pattern transformation rules corresponding to the transformation mappings listed in Table 10.1. The key elements being mapped in the input patterns are tinted blue, unless the default tint colour of the element has semantic meaning (i.e., Value Types are tinted grey to distinguish them from Entity Types), in which case the element is outlined in blue. The key elements being generated in the output patterns are tinted green.

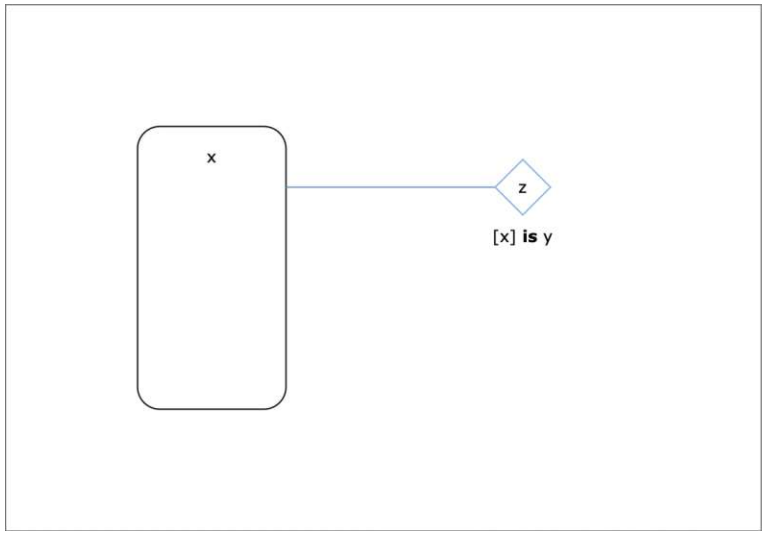
Transformation A1

Transformation A3

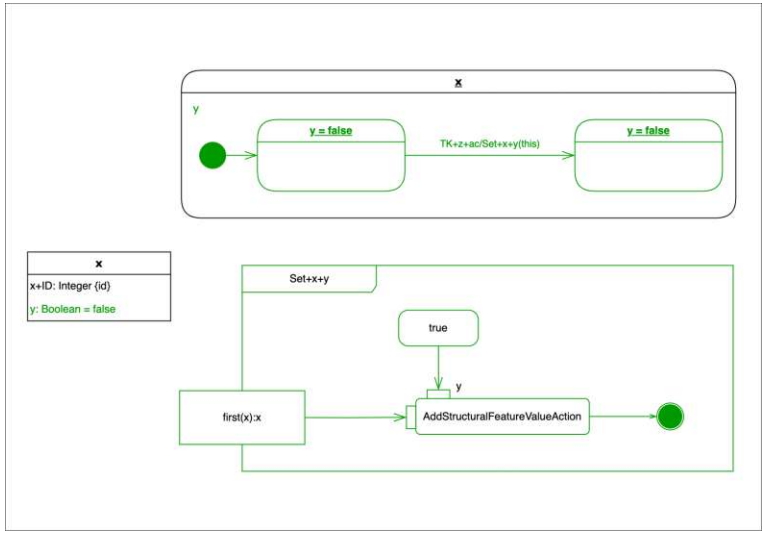


Transformation A5

Input pattern

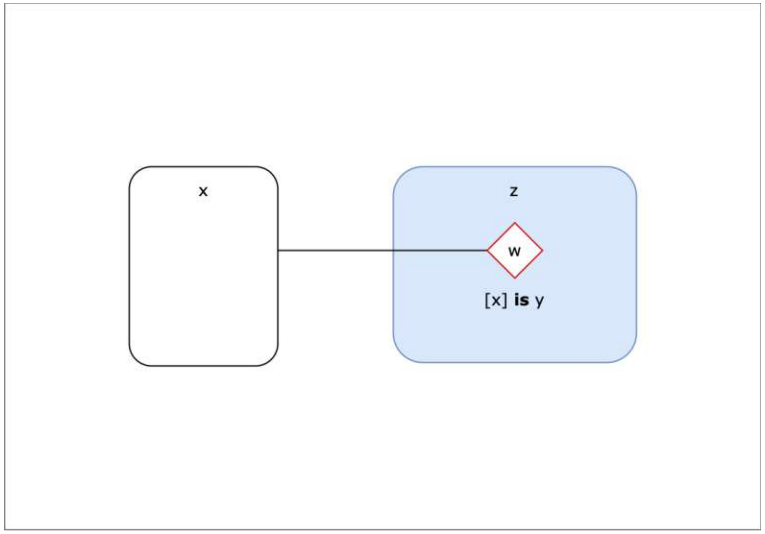


Output pattern

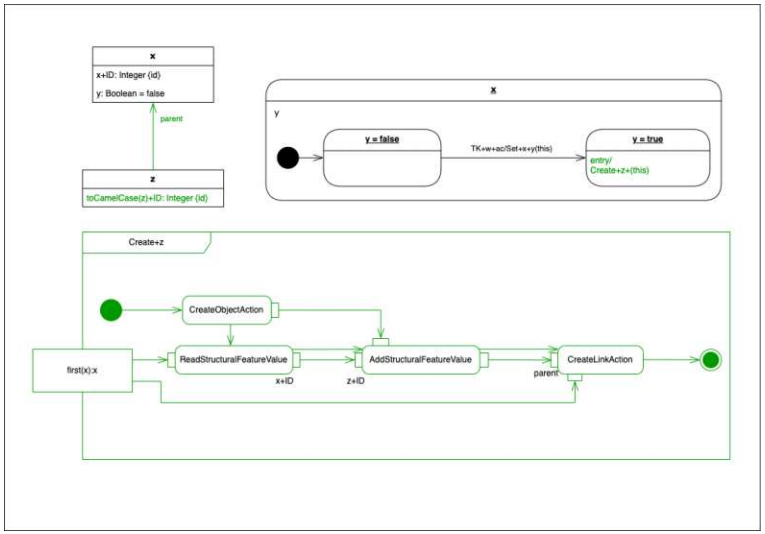


Transformation A6

Input pattern

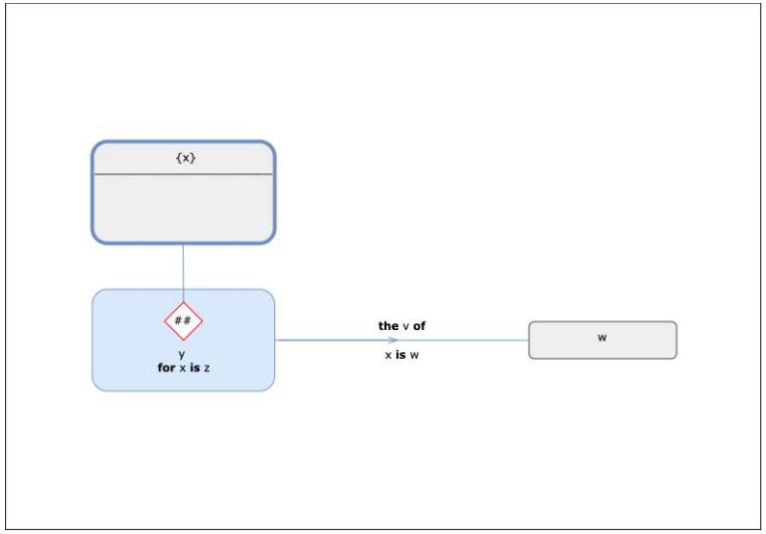


Output pattern

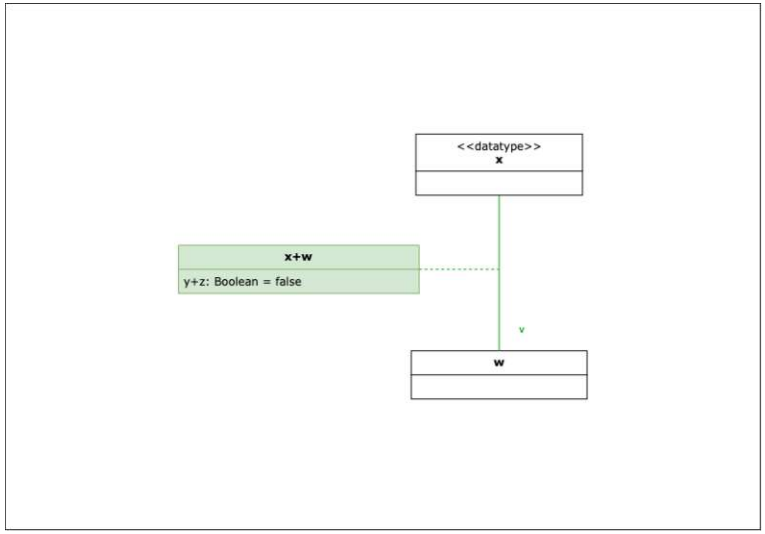


Transformation A7

Input pattern

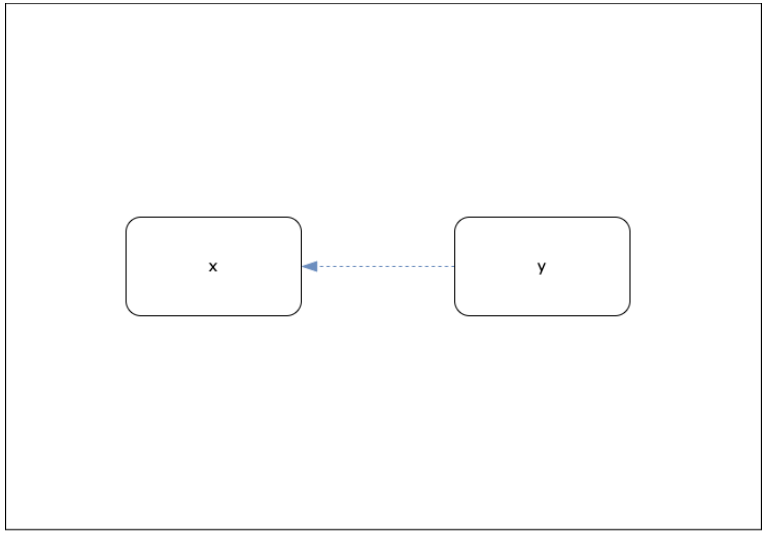


Output pattern

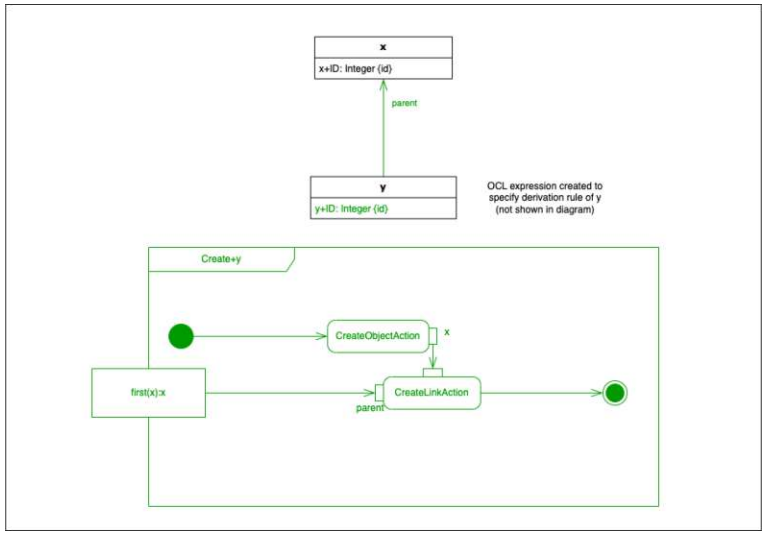


Transformation A8

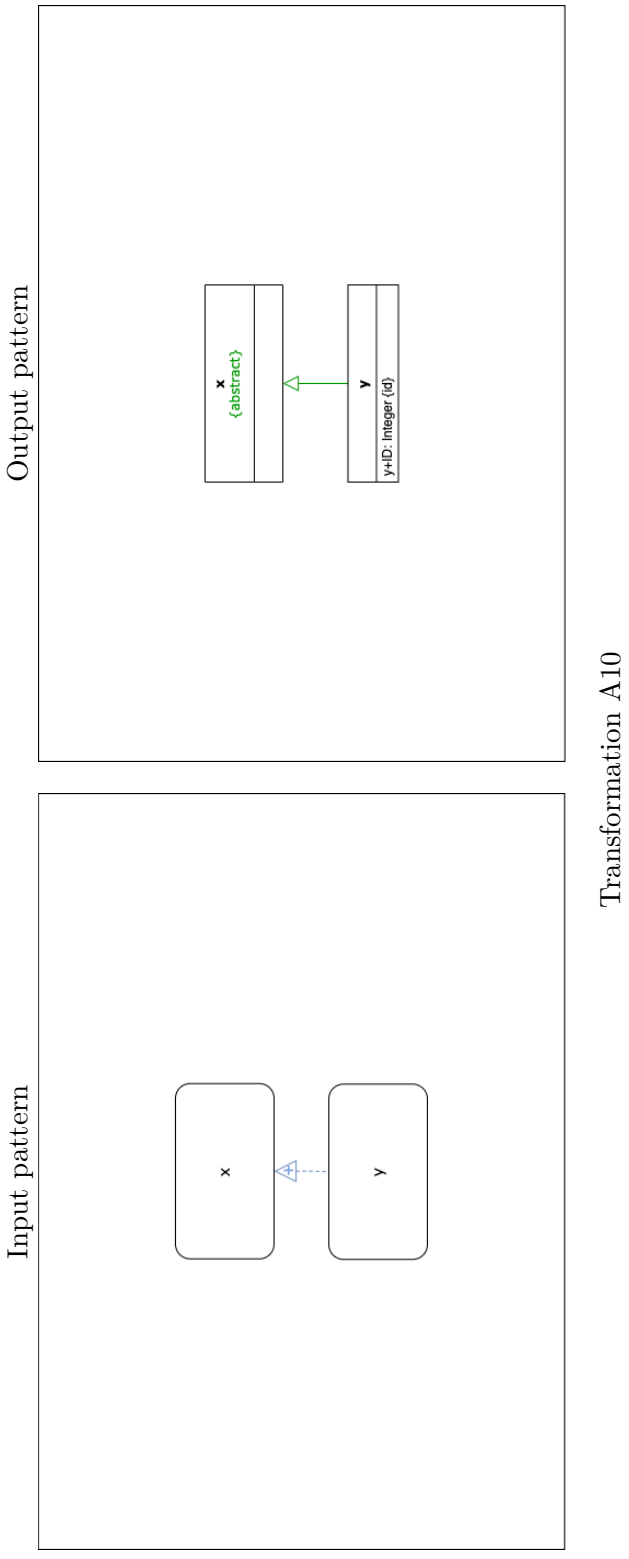
Input pattern



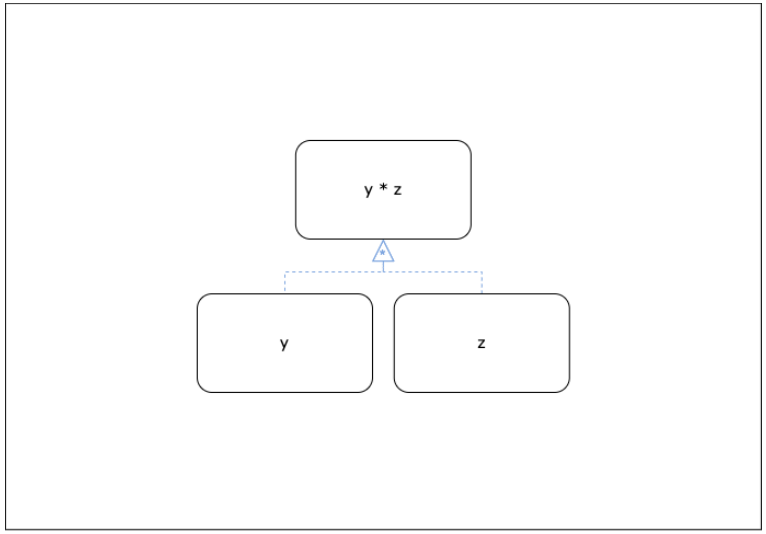
Output pattern



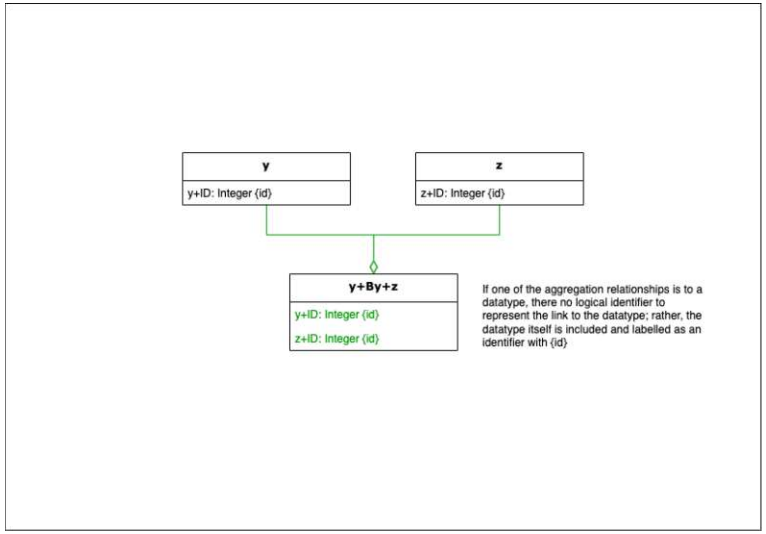
Transformation A9



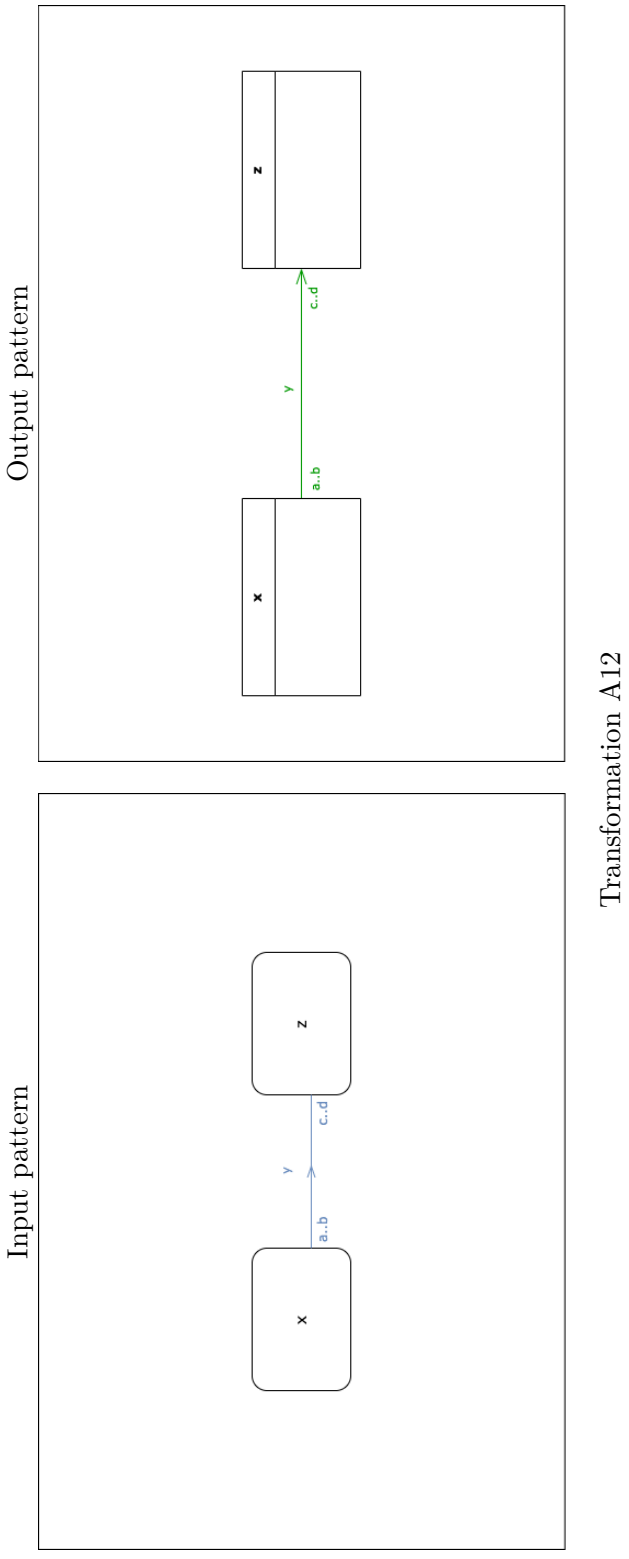
Input pattern



Output pattern



Transformation A11



Transformation A12

APPENDIX F

Transformation Mappings: pimUML to Mendix

This appendix provides the details of the PIM-to-PSM transformation specification from pimUML to Mendix. There are three key components of the transformation specification: value mapping functions, helper functions, and graphical matched pattern transformation rules. The graphical matched pattern transformation rules demonstrate how various input patterns of pimUML modelling concepts, expressed in the standard graphical notation of UML, are mapped to output patterns of Mendix modelling concepts, expressed as UML object diagrams representing instantiated Mendix elements[1]. The value mapping functions may appear in elements of the output pattern for various transformation mappings. Helper functions operate on values specified by variables in input patterns to produce new values for use in elements generated in the corresponding output patterns.

F.1 Value Mapping Functions

- $v \mapsto \sigma$ in Table F.1 maps pimUML data types (v) to Mendix data types (ϕ)
- $(a, b) \mapsto (\alpha, \beta)$ in Table F.2 maps multiplicities on associations in the pimUML class diagram to Mendix association owner and type values in the domain model

[1]It should be noted that the numbers identifying the objects in the output patterns are arbitrarily assigned and serve no further purpose than to uniquely identify different instances of like elements. Likewise, these numbers are specific to each transformation rule and therefore do not intend to suggest that output pattern objects of the same type and with equivalent identifiers represent the same instantiation across different transformation rules.

$v \mapsto \sigma$	
v	σ
DateTime	DateTime
Integer	Integer
Real	Decimal
Boolean	Boolean
Enumeration	Enumeration

Table F.1: Mapping function from pimUML data types to Mendix data types

$(a, b) \mapsto (\alpha, \beta)$			
a	b	α	β
1	1	Both	Reference
1	*	Default	Reference
*	1	Default	Reference
*	*	Default	Reference set

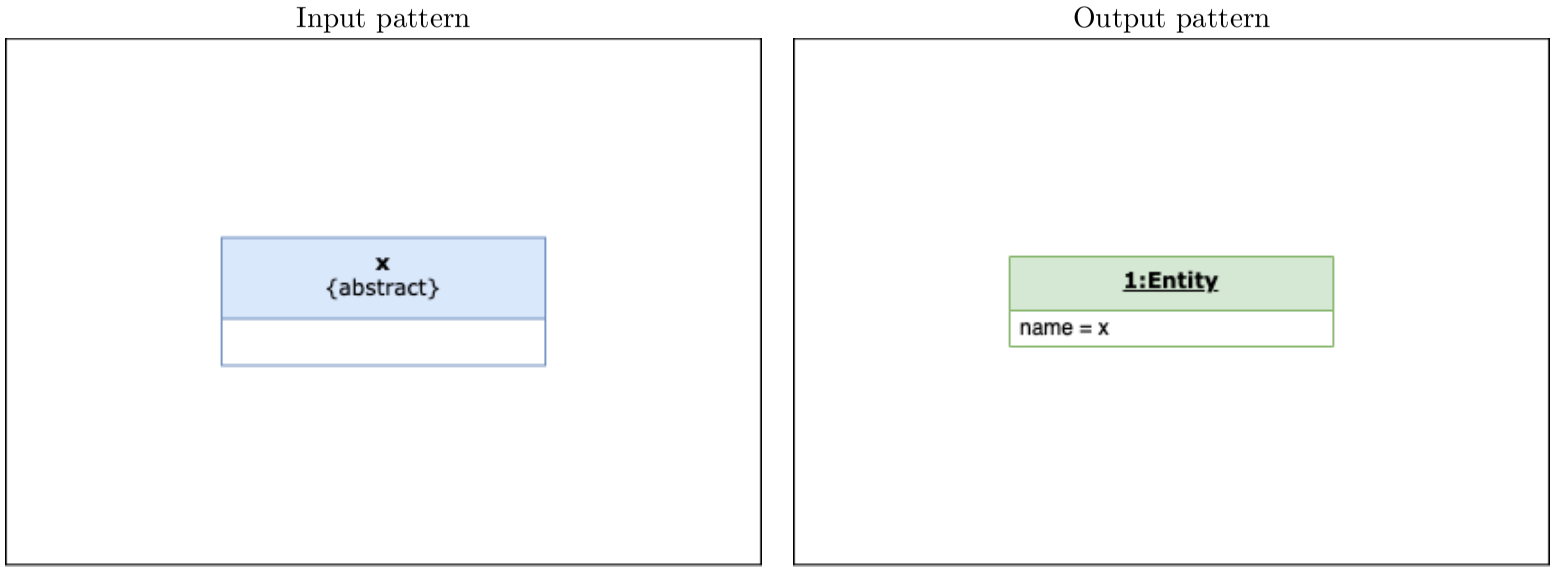
Table F.2: Mapping function from pimUML association multiplicity upper bounds to Mendix association type and ownership values

F.2 Helper Functions

- + denotes a string concatenation operation
- `toUpperCamelCase(x)` reformats an inputted string `x` into upper camel case format

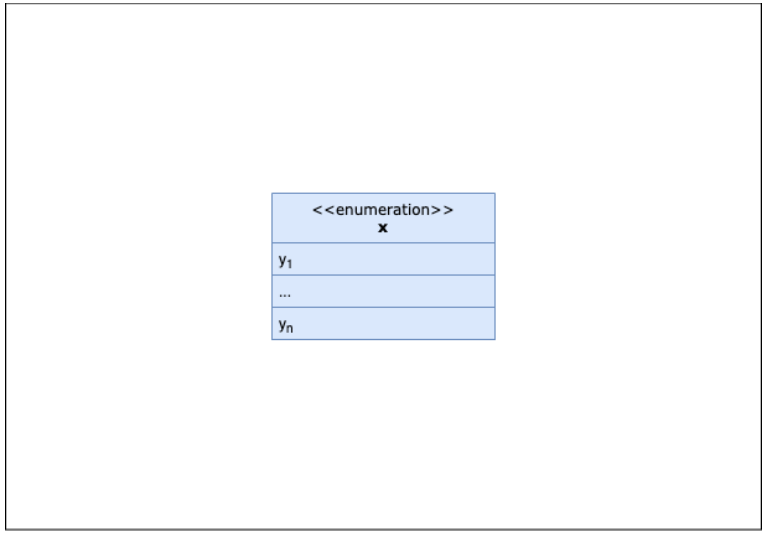
F.3 Graphical Matched Pattern Transformation Rules

The following pages provide illustrations of the matched pattern transformation rules corresponding to the transformation mappings listed in Table 11.1. The key elements being mapped in the input patterns are tinted blue, and the key elements being generated in the output patterns are tinted green.

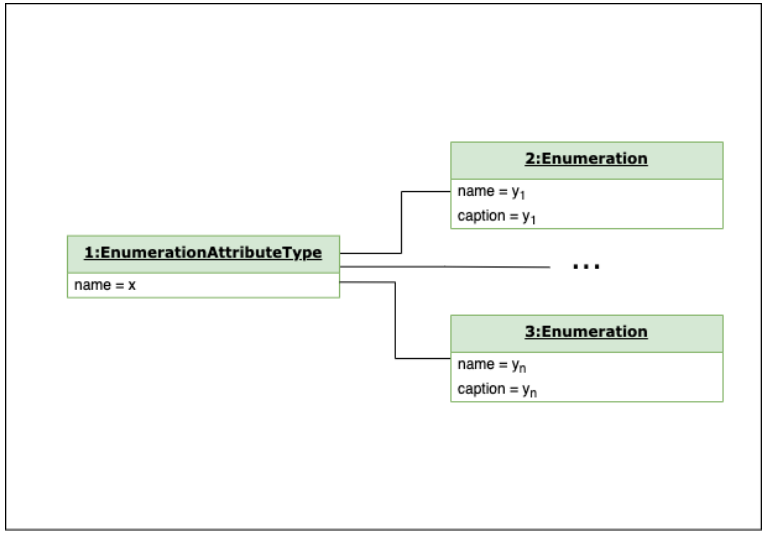


Transformation B2

Input pattern



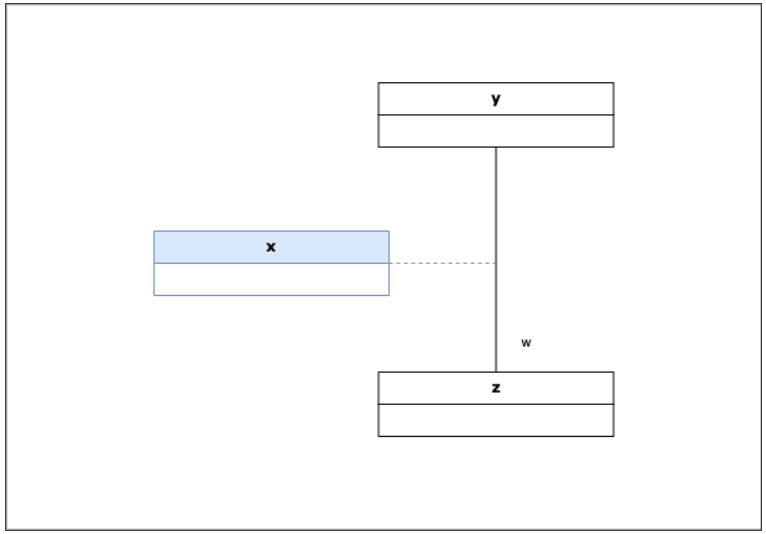
Output pattern



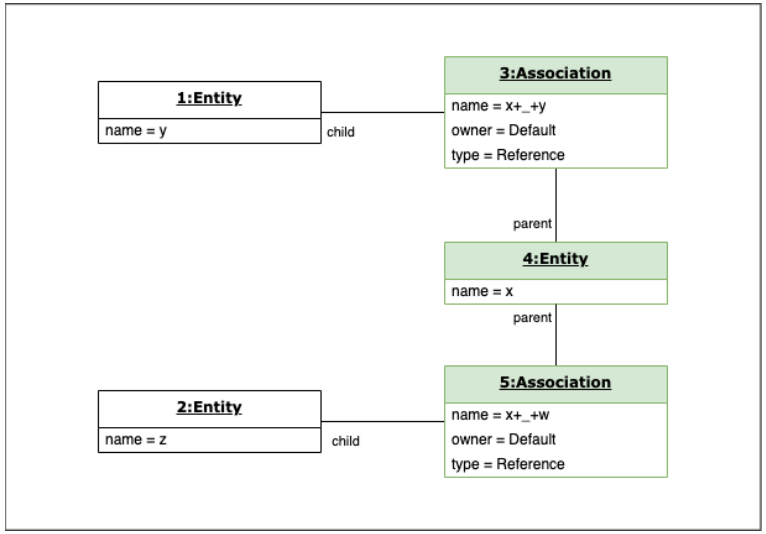
Transformation B3

Transformation B4

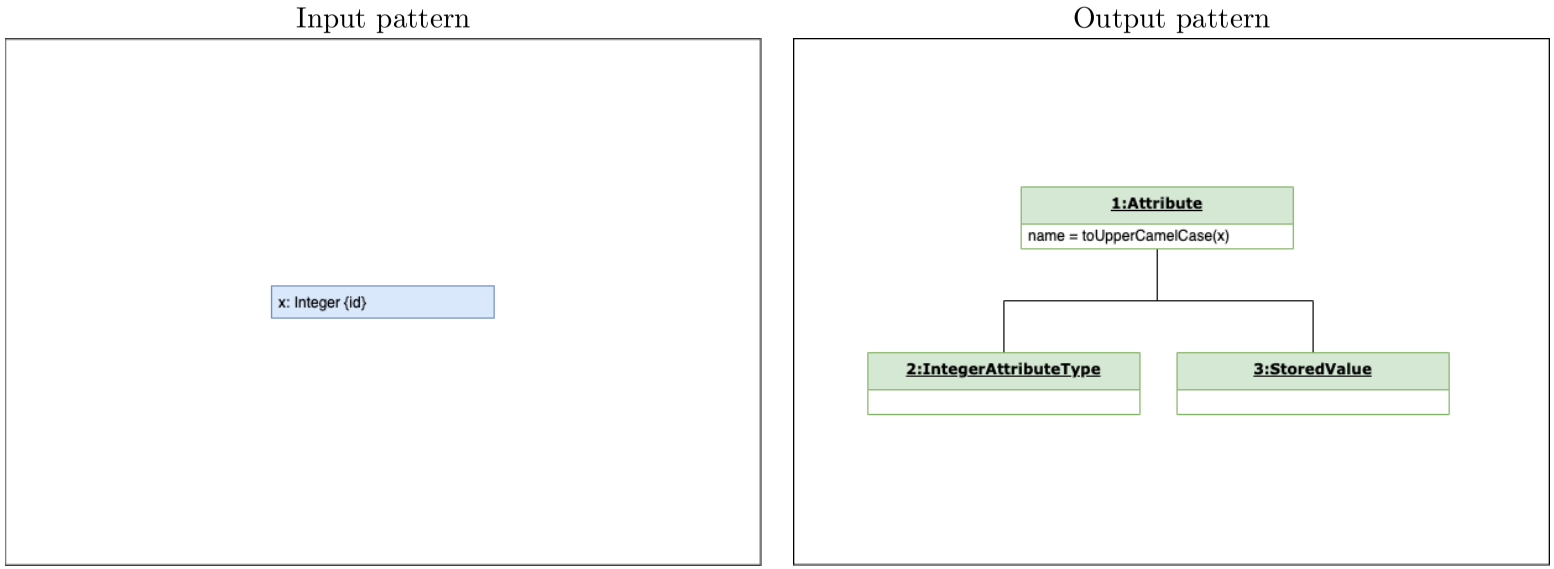
Input pattern



Output pattern

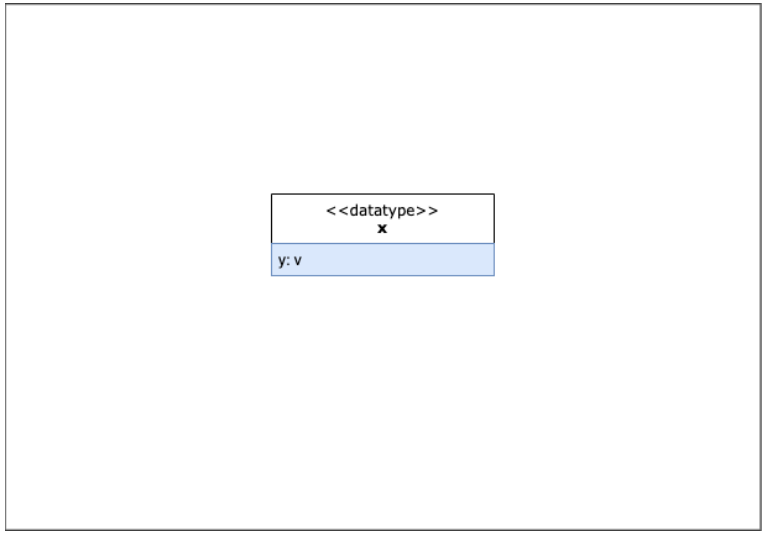


Transformation B5

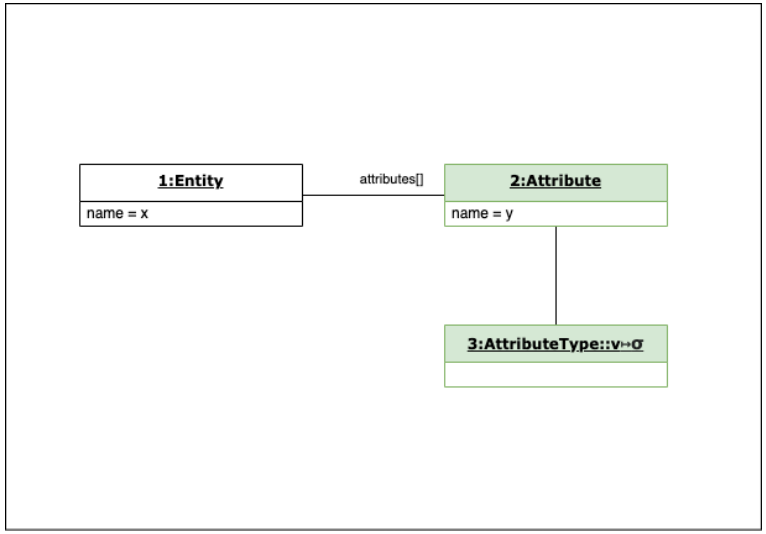


Transformation B6

Input pattern

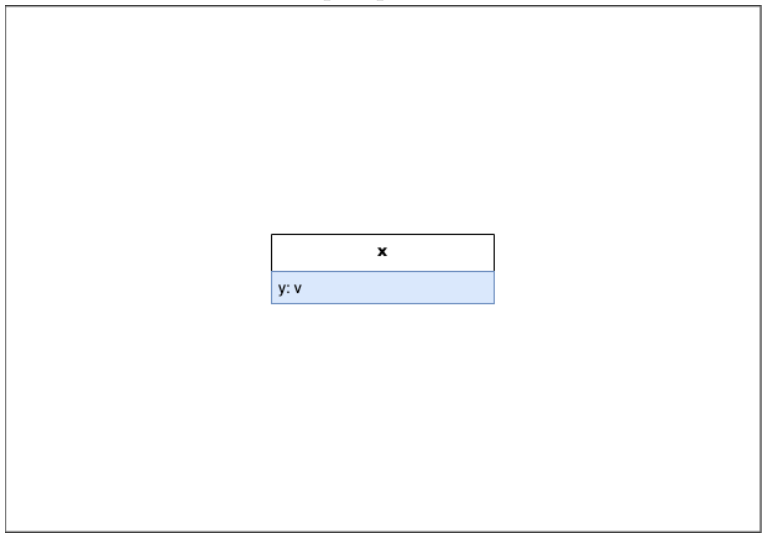


Output pattern

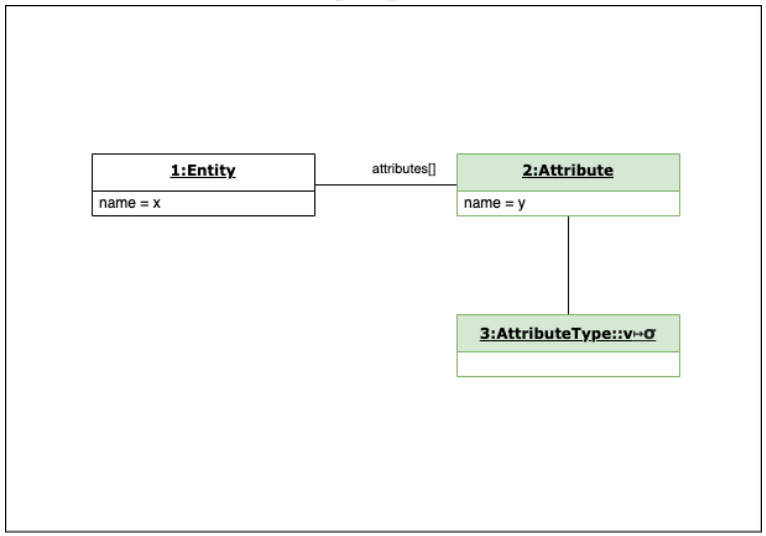


Transformation B7

Input pattern

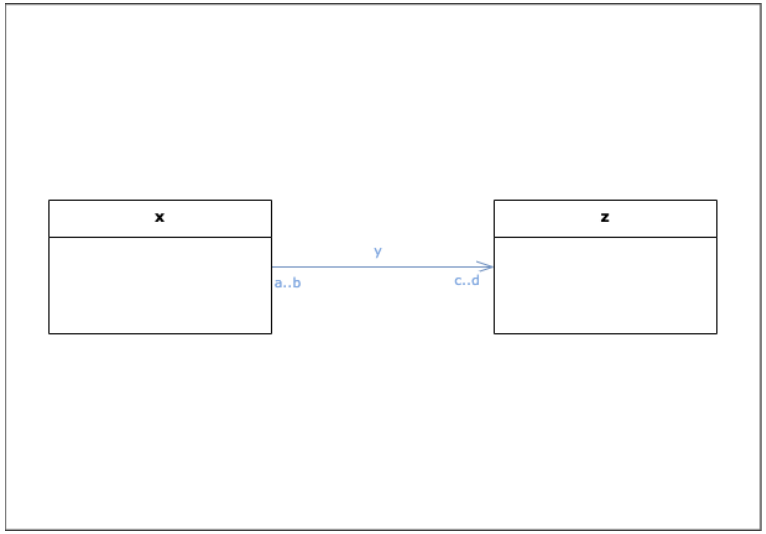


Output pattern

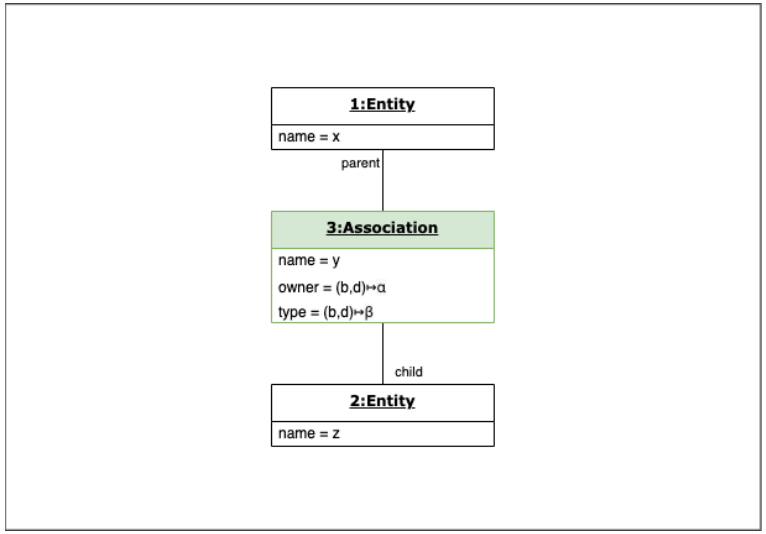


Transformation B8

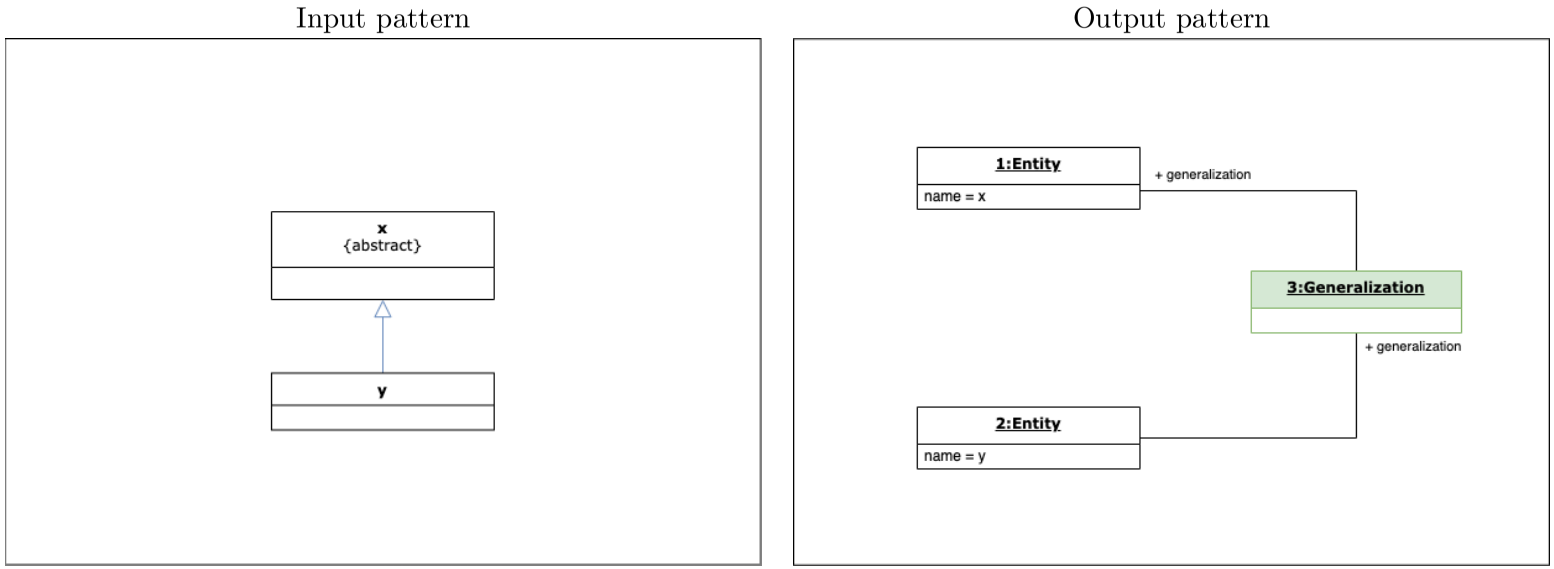
Input pattern



Output pattern

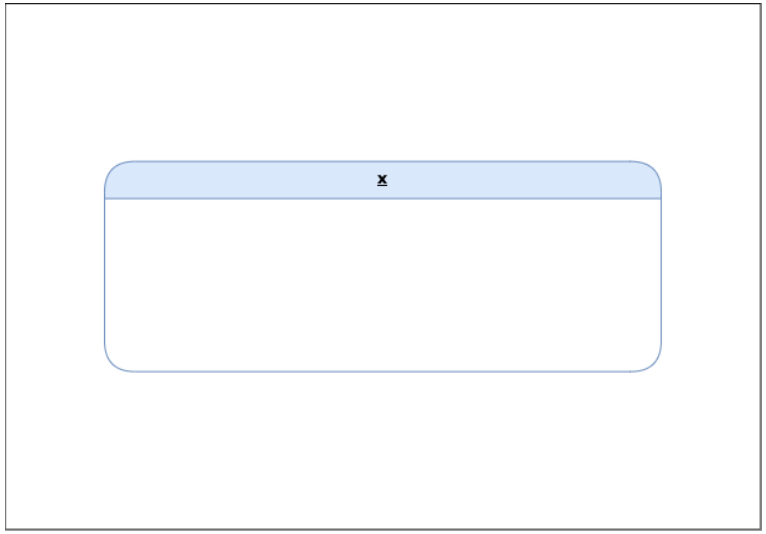


Transformation B9

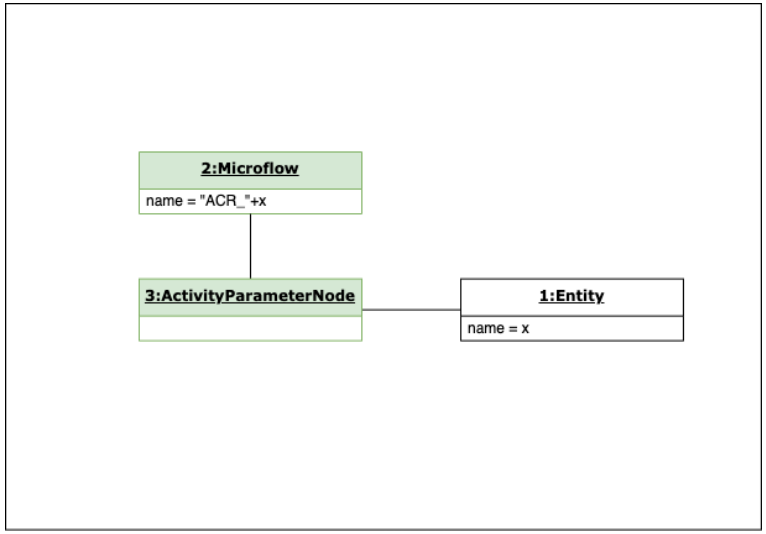


Transformation B10

Input pattern

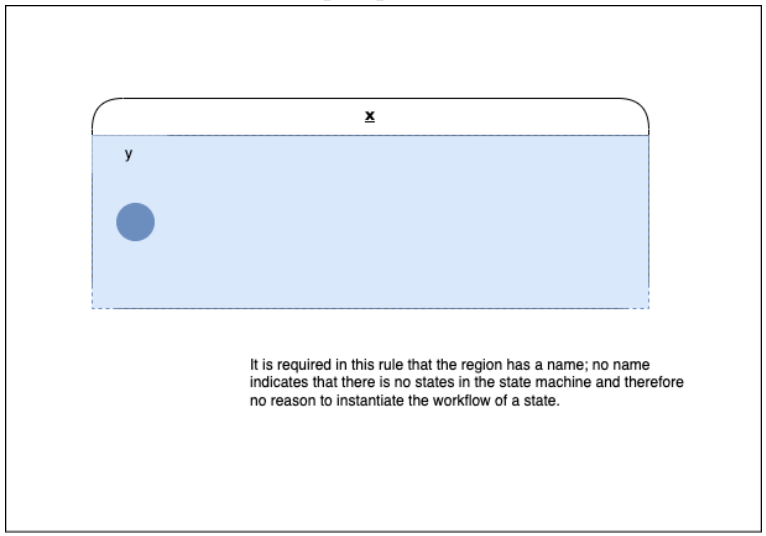


Output pattern

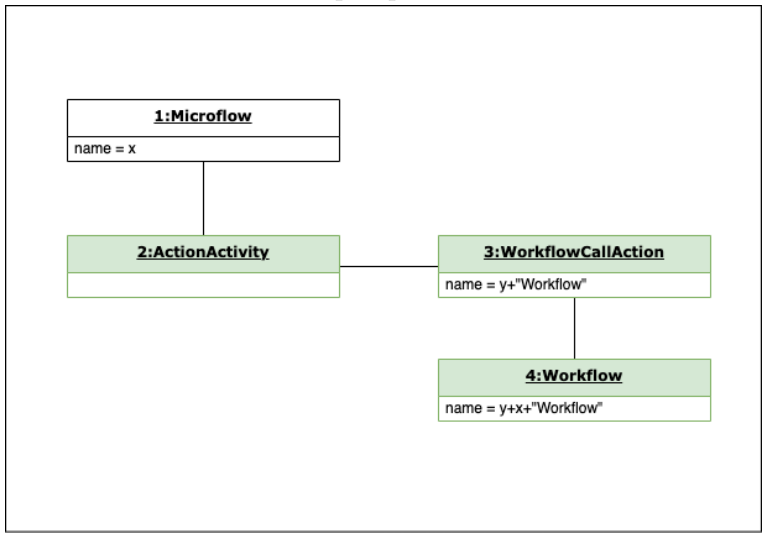


Transformation B11

Input pattern

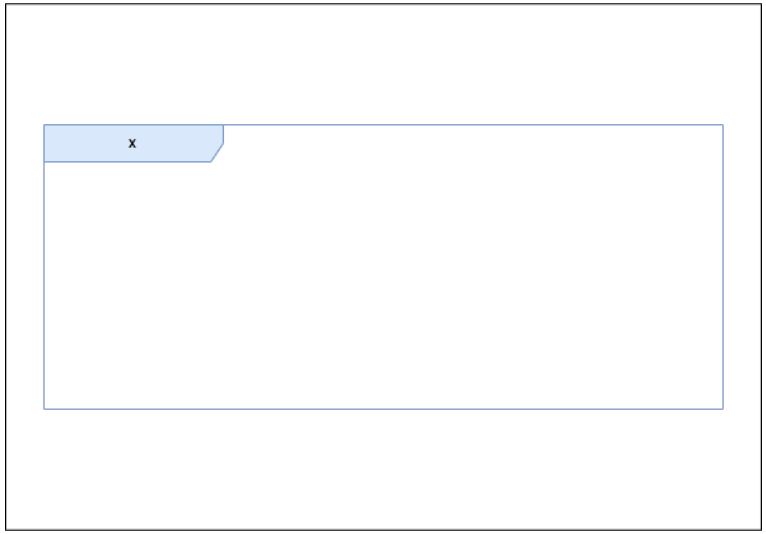


Output pattern

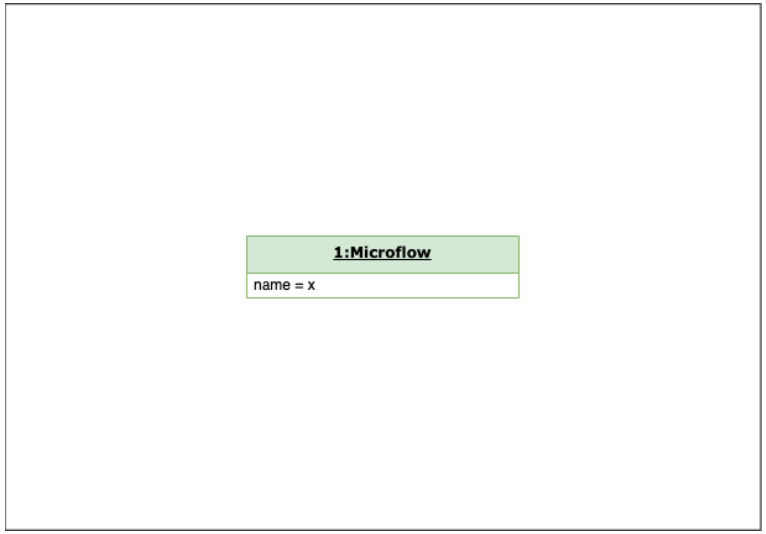


Transformation B12

Input pattern

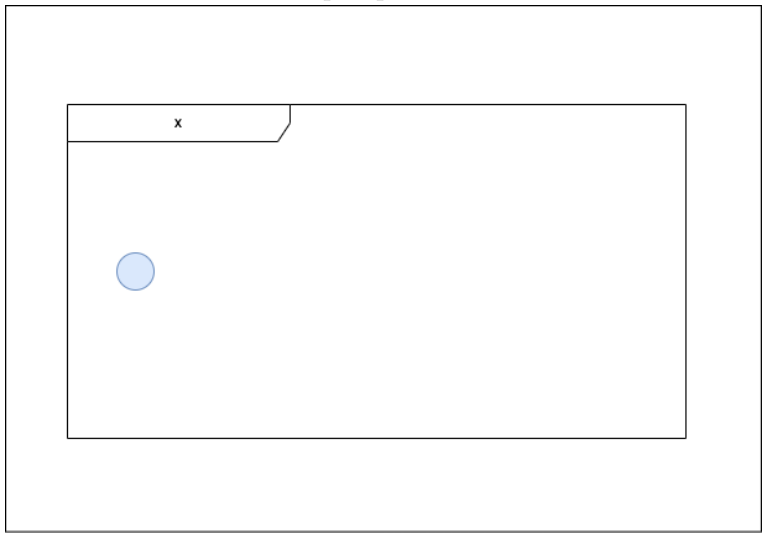


Output pattern

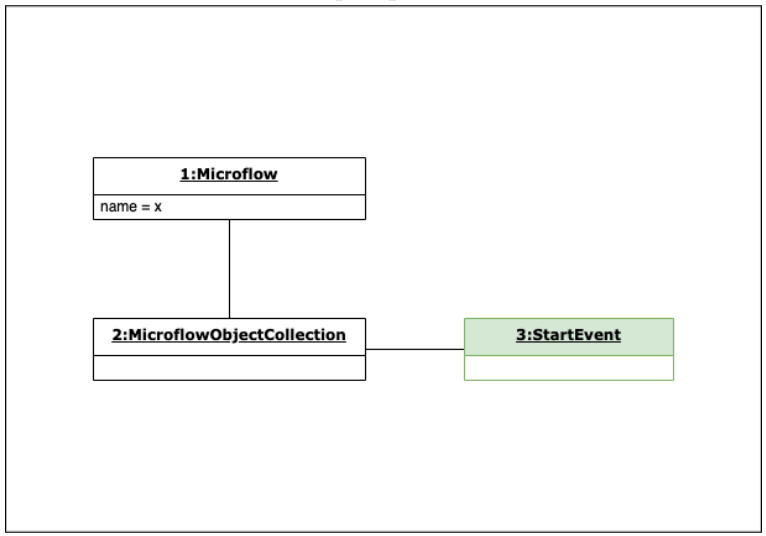


Transformation B13

Input pattern

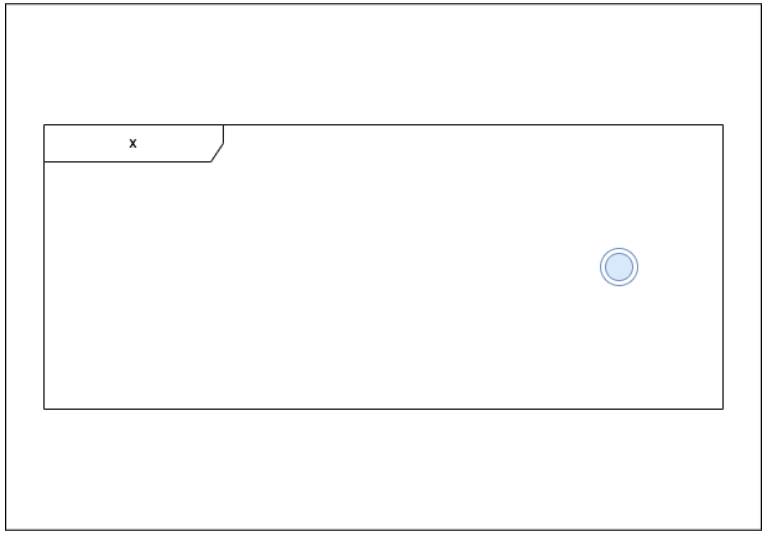


Output pattern

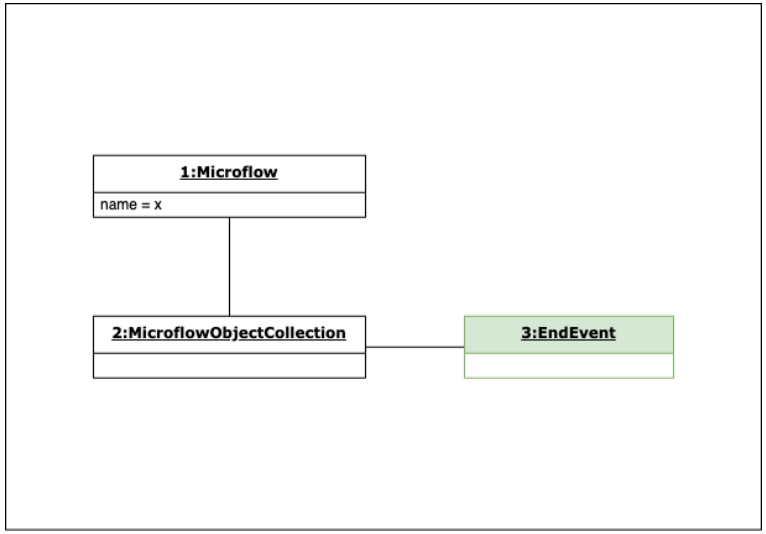


Transformation B14

Input pattern

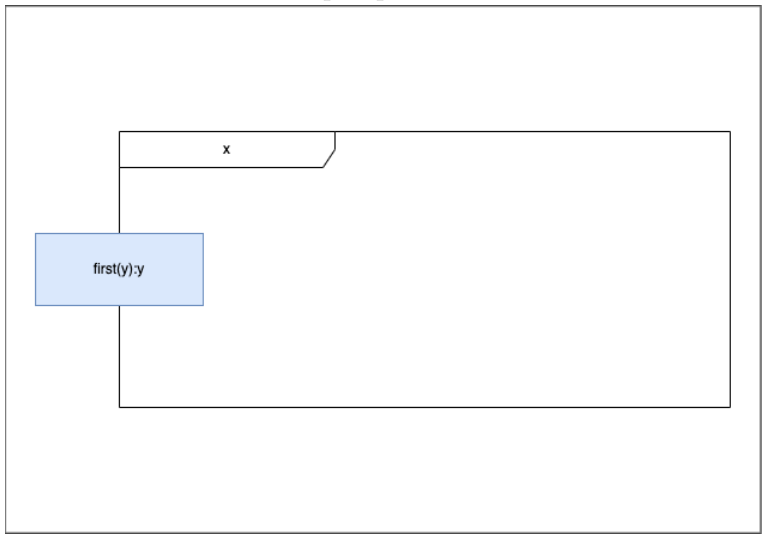


Output pattern

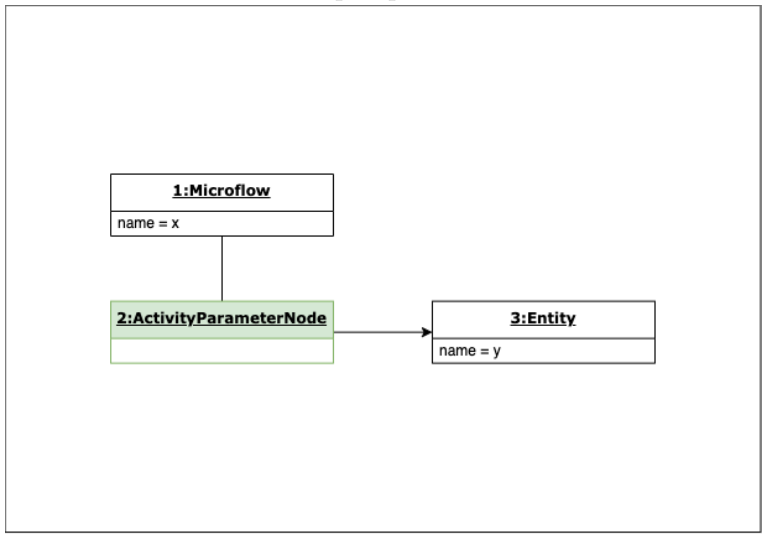


Transformation B15

Input pattern

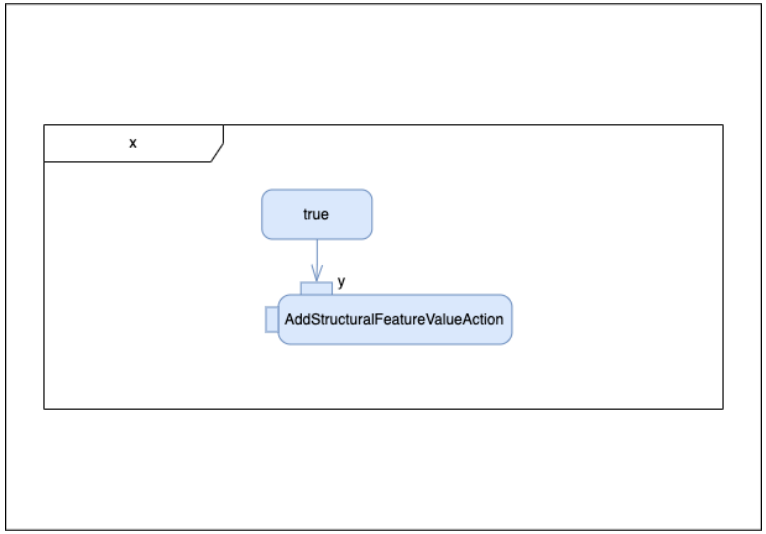


Output pattern

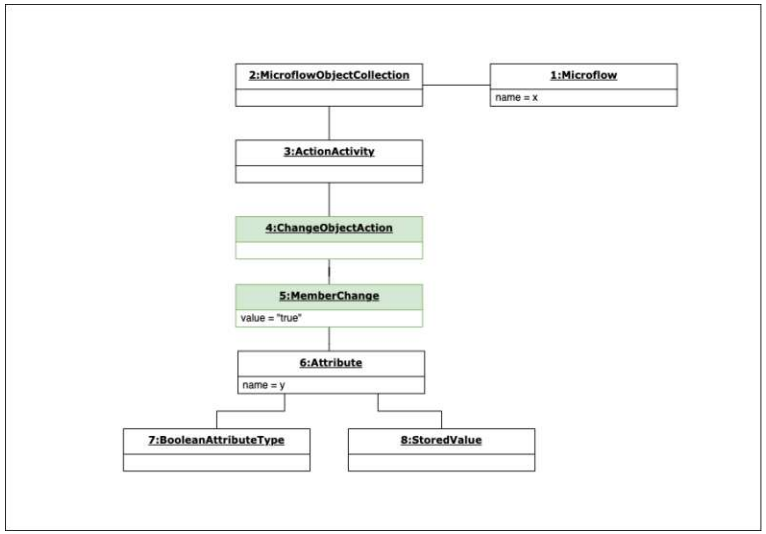


Transformation B16

Input pattern

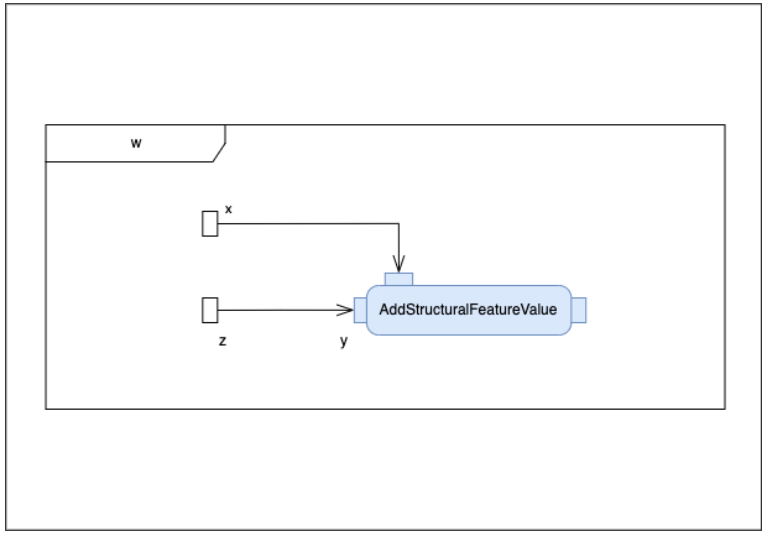


Output pattern

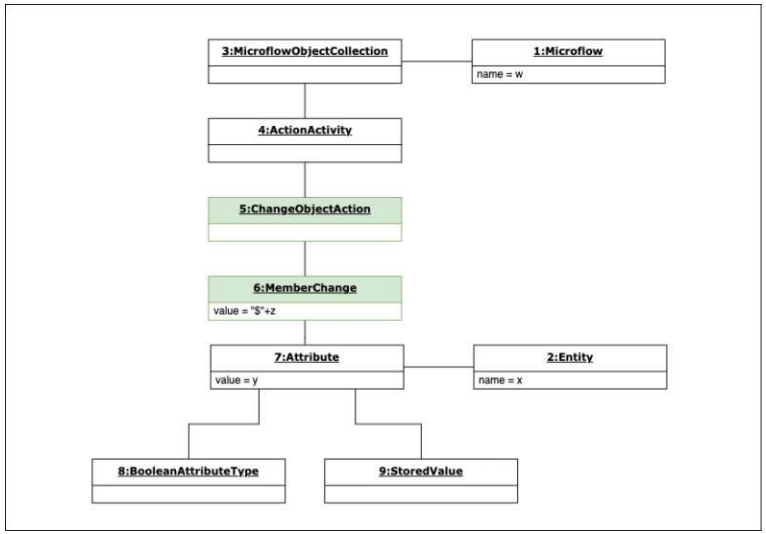


Transformation B17

Input pattern

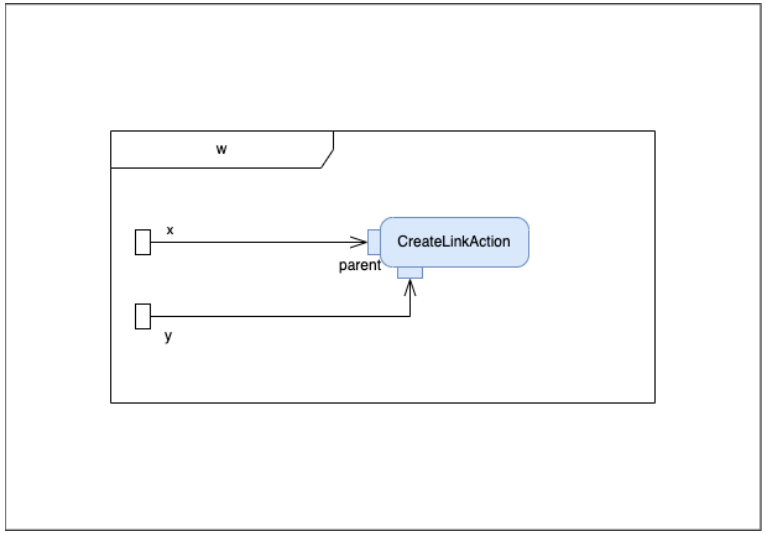


Output pattern

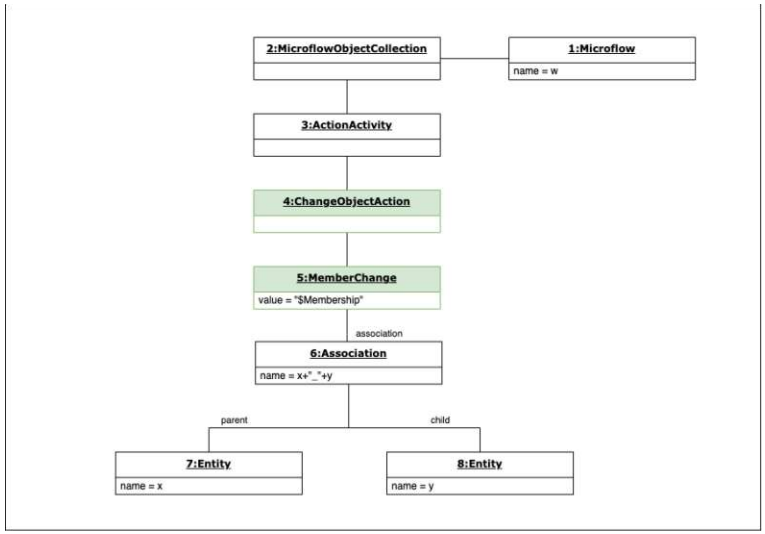


Transformation B18

Input pattern

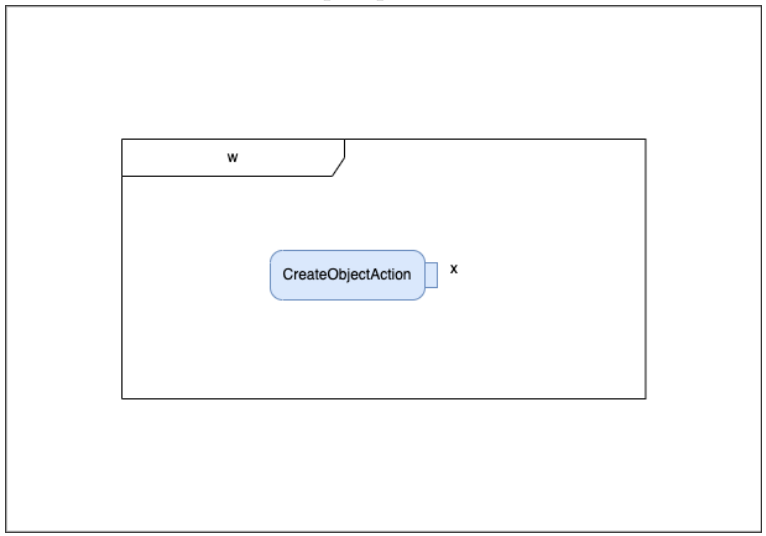


Output pattern

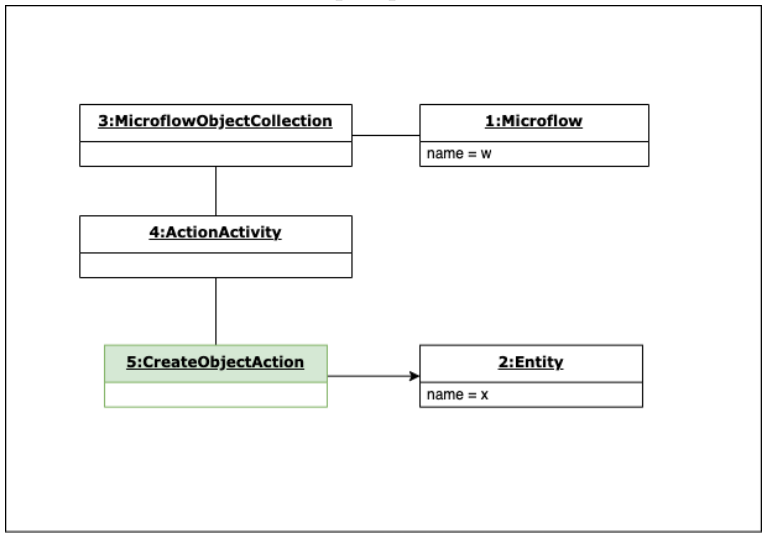


Transformation B19

Input pattern

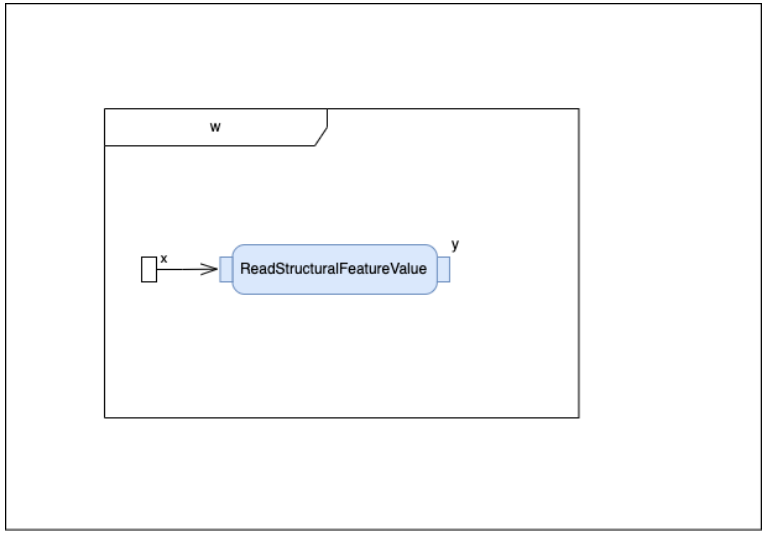


Output pattern

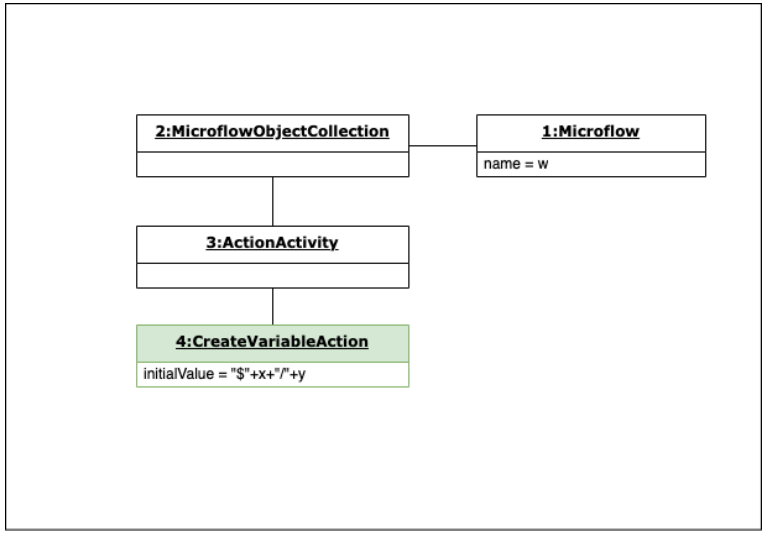


Transformation B20

Input pattern

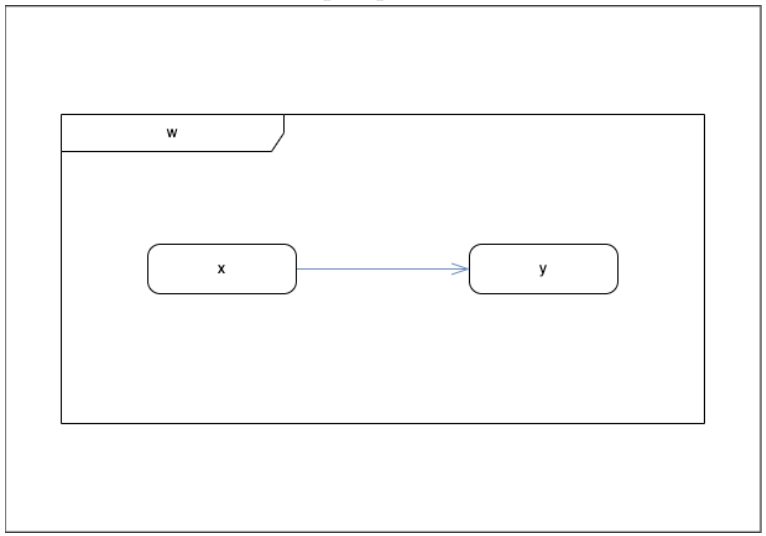


Output pattern

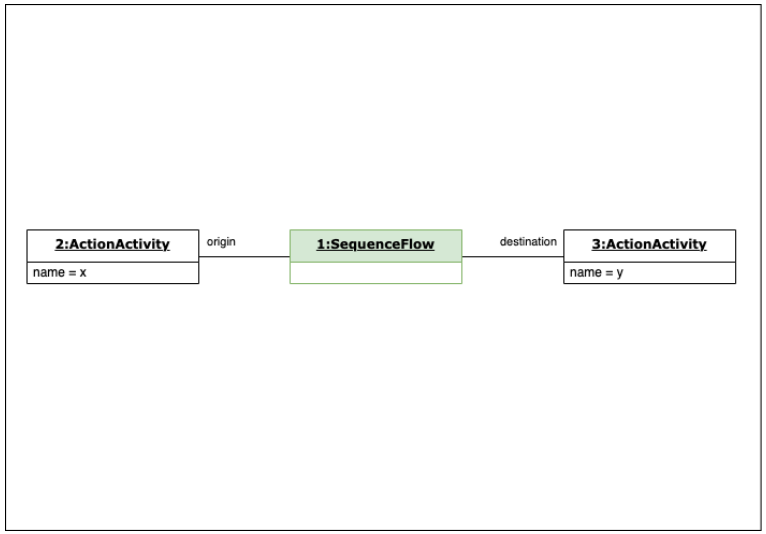


Transformation B21

Input pattern

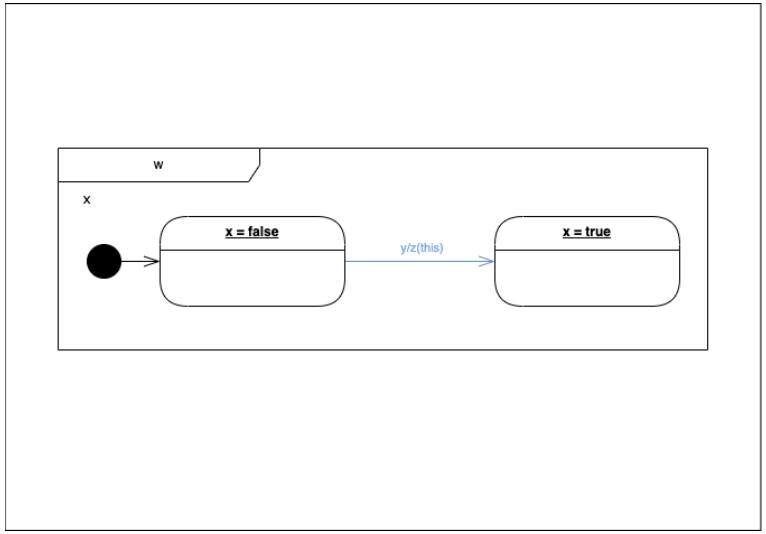


Output pattern

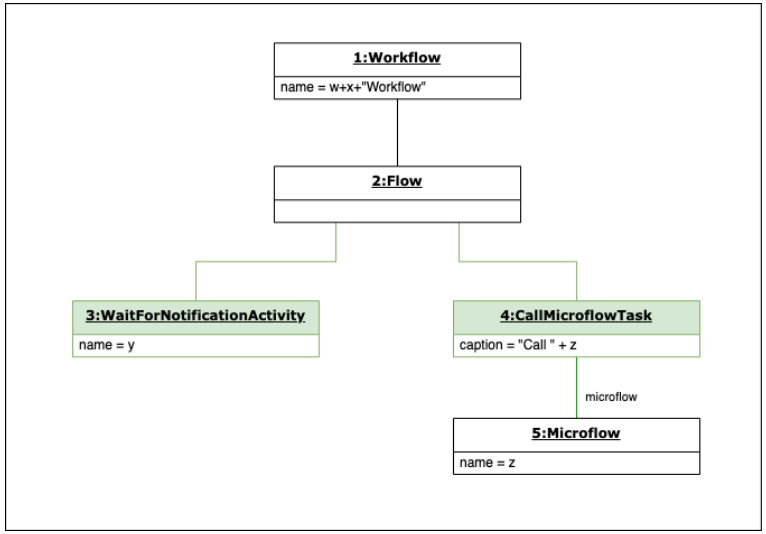


Transformation B22

Input pattern

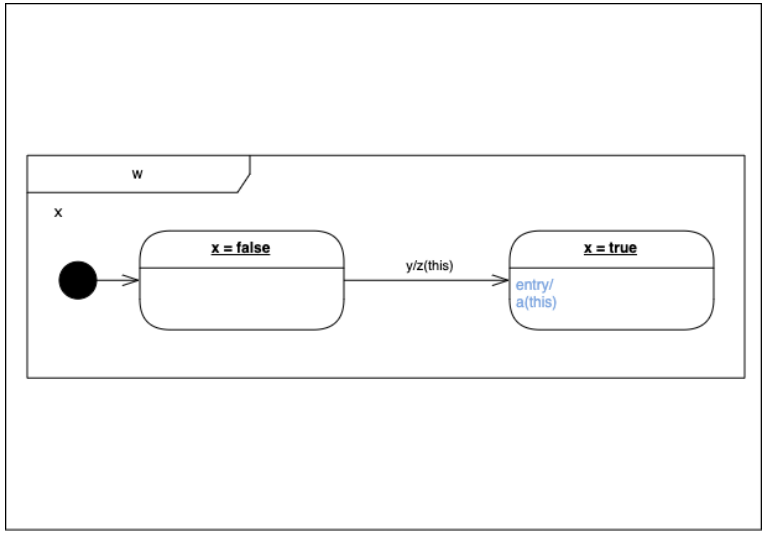


Output pattern

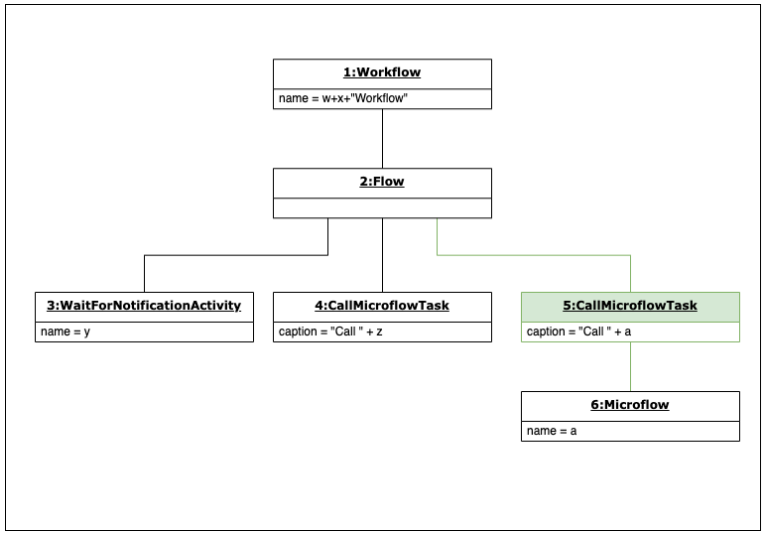


Transformation B23

Input pattern



Output pattern



Transformation B24

APPENDIX G

RAC Case Study Fact Statements

This appendix contains the domain fact statements derived from the Rent-A-Car (RAC) case study, derived through the *verbalization* process demonstrated in [33]. The fact statements are organized into separate tables, grouped by the fact types by which they are represented in the Object Fact Diagram of the RAC DEMO fact model. For each of the CIM-to-PIM mappings demonstrated from Chapters 7 through 10, as well as the PIM-to-PSM mappings in Chapter 11, the semantic correctness of each fact statement was assessed. Each of the transformation target modelling languages of each of these chapters has a column in the below tables. A ✓ is used to denote if the semantics of each fact statement was deemed to be correctly captured by the chosen elements of each target modelling language.

Domain Concept	Expressed in OFD as	Fact Statements	UML?	xUML?	fUML?	pimUML?	Mendix?
Rental	Declared entity type	entity type rental exists	✓	✓	✓	✓	✓
Deposit-Paid Rental	Derived entity type (Specialization w/ event type)	derived entity type deposit-paid rental exists	✓	✓	✓	✓	✓
Taken Rental	Derived entity type (Specialization w/ event type)	derived entity type taken rental exists	✓	✓	✓	✓	✓
Returned Rental	Derived entity type (Specialization w/ event type)	derived entity type returned rental exists	✓	✓	✓	✓	✓
Invoice-Paid Rental	Derived entity type (Specialization w/ event type)	derived entity type invoice-paid rental exists	✓	✓	✓	✓	✓
Transport	Declared entity type	entity type transport exists	✓	✓	✓	✓	✓
Car	Declared entity type	entity type car exists	✓	✓	✓	✓	✓
Branch	Declared entity type	entity type branch exists	✓	✓	✓	✓	✓
Person	Declared entity type	entity type person exists	✓	✓	✓	✓	✓
Employee	Declared entity type	entity type employee exists	✓	✓	✓	✓	✓
Driving License	Declared entity type	entity type driving license exists	✓	✓	✓	✓	✓
Yearly Car Group	Aggregate entity type	aggregate entity type car group * year exists	✓		✓	✓	

Table G.1: RAC preservation of fact statements of Entity Types for each design sprint

Domain Concept	Expressed in OFD as	Fact Statements	UML?	xUML?	fUML?	pimUML?	Mendix?
Car Group	Value type - user-defined, categorical	value type car group exists	✓	✓	✓	✓	✓
Year	Value type - user-defined, non-categorical	value type year exists	✓	✓	✓	✓	✓
Day	Value type - user-defined, non-categorical	value type day exists	✓	✓	✓	✓	✓

Table G.2: RAC preservation of fact statements of Value Types for each design sprint

Domain Concept	Expressed in OFD as	Fact Statements	UML?	xUML?	fUML?	pimUML?	Mendix?
TK01 rental is completed	Event type,	event type rental completed exists	✓	✓	✓	✓	✓
	Derived entity type (Specialization w/ event type)	rental completed concerns rental	✓	✓	✓	✓	✓
TK02 the car of rental is taken	Event type,	event type rental taken exists	✓	✓	✓	✓	✓
	Derived entity type (Specialization w/ event type)	rental taken concerns rental	✓	✓	✓	✓	✓
TK03 the car of rental is returned	Event type,	event type rental returned exists	✓	✓	✓	✓	✓
	Derived entity type (Specialization w/ event type)	returned rental concerns rental	✓	✓	✓	✓	✓
TK04 the deposit of rental is paid	Event type,	event type rental deposit paid exists	✓	✓	✓	✓	✓
	Derived entity type (Specialization w/ event type)	rental deposit paid concerns rental	✓	✓	✓	✓	✓
TK05 the invoice of rental is paid	Event type,	event type rental invoice paid exists	✓	✓	✓	✓	✓
	Derived entity type (Specialization w/ event type)	rental invoice paid concerns rental	✓	✓	✓	✓	✓
TK06 transport is completed	Event type,	event type transport completed exists	✓	✓	✓	✓	✓
	Derived entity type (Specialization w/ event type)	transport completed concerns transport	✓	✓	✓	✓	✓
TK07 transport managing for day is complete	Event type,	event type transport managing for day complete exists	✓	✓	✓	✓	✓
	Derived entity type (Specialization w/ event type)	transport managing for day concerns day	✓	✓		✓	✓

Table G.3: RAC preservation of fact statements of Event Types for each design sprint

Domain Concept	Expressed in OFD as	Fact Statements	UML?	xUML?	fUML?	pimUML?	Mendix?
Starting day of rental	Attribute type,	attribute type starting day of rental exists	✓	✓	✓	✓	✓
	Value type - user-defined, non-categorical	the domain of starting day of rental is rental	✓	✓	✓	✓	✓
		the range of starting day is day	✓	✓	✓	✓	✓
Ending day of rental	Attribute type,	attribute type ending day of rental exists	✓	✓	✓	✓	✓
	Value type - user-defined, non-categorical	the domain of ending day of rental is rental	✓	✓	✓	✓	✓
		the range of ending day is range	✓	✓	✓	✓	✓
Car group of rental	Attribute type,	attribute type car group of rental exists	✓	✓	✓	✓	✓
	Value type - user-defined, categorical	the domain of car group of rental is rental	✓	✓	✓	✓	✓
		the range of car group of rental is car group	✓	✓	✓	✓	✓
Expiration day of driving license	Attribute type,	attribute type expiration day of driving license exists	✓	✓	✓	✓	✓
	Value type - user-defined, non-categorical	the domain of expiration day of driving license is driving license	✓	✓	✓	✓	✓
		the range of expiration day of driving license is day	✓	✓	✓	✓	✓
Car group of car	Attribute type,	attribute type car group of car exists	✓	✓	✓	✓	✓
	Value type - user-defined, categorical	the domain of car group of car is car	✓	✓	✓	✓	✓
		the range of car group of car is car group	✓	✓	✓	✓	✓
Deposit amount of deposit-paid rental	Attribute type,	attribute type deposit amount of deposit-paid rental exists	✓	✓	✓	✓	✓
	Value type - pre-defined	the domain of deposit amount of deposit-paid rental is deposit-paid rental	✓	✓	✓	✓	✓
		the range of deposit amount of deposit-paid rental is money	✓	✓	✓	✓	✓
Invoice amount of deposit-paid rental	Attribute type,	attribute type invoice amount of invoice-paid rental exists	✓	✓	✓	✓	✓
	Value type - pre-defined	the domain of invoice amount of invoice-paid rental is invoice-paid rental	✓	✓	✓	✓	✓
		the range of invoice amount of invoice-paid rental is money	✓	✓	✓	✓	✓
Yearly daily rental rate of car group	Attribute type,	attribute type daily rental rate of car group by year exists	✓	✓	✓	✓	✓
	Value type - pre-defined	the domain of daily rental rate of car group by year is car group by year	✓	✓	✓	✓	✓
		the range of daily rental rate of car group by year is money	✓	✓	✓	✓	✓
Yearly standard deposit amount of car group	Attribute type,	attribute type standard deposit amount of car group by year exists	✓	✓	✓	✓	✓
	Value type - pre-defined	the domain of standard deposit amount of car group by year is car group by year	✓	✓	✓	✓	✓
		the range of standard deposit amount of car group by year is money	✓	✓	✓	✓	✓
Max rental duration of year	Attribute type,	attribute type max rental duration of year exists	✓	✓	✓	✓	✓
	Value type - pre-defined	the domain of max rental duration of year is year	✓	✓	✓	✓	✓
		the range of max rental duration of year is number	✓	✓	✓	✓	✓
Rental horizon of year	Attribute type,	attribute type rental horizon of year exists	✓	✓	✓	✓	✓
	Value type - pre-defined	the domain of rental horizon of year is year	✓	✓	✓	✓	✓
		the range of rental horizon of year is number	✓	✓	✓	✓	✓
Location fine rate of year	Attribute type,	attribute type location fine rate of year exists	✓	✓	✓	✓	✓
	Value type - pre-defined	the domain of location fine rate of year is year	✓	✓	✓	✓	✓
		the range of location fine rate of year is money	✓	✓	✓	✓	✓
Late return fine rate of year	Attribute type,	attribute type late return fine rate of year exists	✓	✓	✓	✓	✓
	Value type - pre-defined	the domain of late return fine rate of year is year	✓	✓	✓	✓	✓
		the range of late return fine rate of year is money	✓	✓	✓	✓	✓
Day of transport of transport	Attribute type,	attribute type day of transport of transport exists	✓	✓	✓	✓	✓
	Value type - user-defined, non-categorical	the domain of day of transport of transport is transport	✓	✓	✓	✓	✓
		the range of day of transport of transport is day	✓	✓	✓	✓	✓

Table G.4: RAC preservation of fact statements of Attribute Types for each design sprint

Domain Concept	Expressed in OFD as	Fact Statements	UML?	xUML?	fUML?	pimUML?	Mendix?
The from-branch of transport is branch	Property type (entity-to-entity), Cardinality laws	property type from-branch of transport exists	✓	✓	✓	✓	✓
		the domain of from-branch of transport is transport	✓	✓	✓	✓	✓
		the range of from-branch of transport is branch	✓	✓	✓	✓	✓
The to-branch of transport is branch	Property type (entity-to-entity), Cardinality laws	property type to-branch of transport exists	✓	✓	✓	✓	✓
		the domain of to-branch of transport is transport	✓	✓	✓	✓	✓
		the range of to-branch of transport is branch	✓	✓	✓	✓	✓
The transport manager of day is employee	Property type (entity-to-entity), Cardinality laws	property type transport manager of day exists	✓	✓	✓	✓	✓
		the domain of transport manager of day is day	✓	✓		✓	✓
		the range of transport manager of day is employee	✓	✓	✓	✓	✓
The car of transport is car	Property type (entity-to-entity), Cardinality laws	property type car of transport exists	✓	✓	✓	✓	✓
		the domain of car of transport is transport	✓	✓	✓	✓	✓
		the range of car of transport is car	✓	✓	✓	✓	✓
The pick-up location of rental is branch	Property type (entity-to-entity), Cardinality laws	property type pick-up location of rental exists	✓	✓	✓	✓	✓
		the domain of pick-up location of rental is rental	✓	✓	✓	✓	✓
		the range of pick-up location of rental is branch	✓	✓	✓	✓	✓
The return location of rental is branch	Property type (entity-to-entity), Cardinality laws	property type return location of rental exists	✓	✓	✓	✓	✓
		the domain of return location of rental is rental	✓	✓	✓	✓	✓
		the range of return location of rental is branch	✓	✓	✓	✓	✓
The rental completer of rental is employee	Property type (entity-to-entity), Cardinality laws	property type rental completer of rental exists	✓	✓	✓	✓	✓
		the domain of rental completer of rental is rental	✓	✓	✓	✓	✓
		the range of rental completer of rental is employee	✓	✓	✓	✓	✓
The renter of rental is person	Property type (entity-to-entity), Cardinality laws	property type renter of rental exists	✓	✓	✓	✓	✓
		the domain of renter of rental is rental	✓	✓	✓	✓	✓
		the range of renter of rental is person	✓	✓	✓	✓	✓
The deposit payer of rental is person	Property type (entity-to-entity), Cardinality laws	property type deposit payer of rental exists	✓	✓	✓	✓	✓
		the domain of deposit payer of rental is rental	✓	✓	✓	✓	✓
		the range of deposit payer of rental is person	✓	✓	✓	✓	✓
The invoice payer of rental is person	Property type (entity-to-entity), Cardinality laws	property type invoice payer of rental exists	✓	✓	✓	✓	✓
		the domain of invoice payer of rental is rental	✓	✓	✓	✓	✓
		the range of invoice payer of rental is person	✓	✓	✓	✓	✓
The driver of rental is person	Property type (entity-to-entity), Cardinality laws	property type driver of rental exists	✓	✓	✓	✓	✓
		the domain of driver of rental is rental	✓	✓	✓	✓	✓
		the range of driver of rental is person	✓	✓	✓	✓	✓
The driving license of person is driving license	Property type (entity-to-entity), Cardinality laws	property type driving license of person exists	✓	✓	✓	✓	✓
		the domain of driving license of person is person	✓	✓	✓	✓	✓
		the range of driving license of person is driving license	✓	✓	✓	✓	✓
The car of rental is car	Property type (entity-to-entity), Cardinality laws	property type car of rental exists	✓	✓	✓	✓	✓
		the domain of car of rental is taken rental	✓	✓	✓	✓	✓
		the range of car of rental is car	✓	✓	✓	✓	✓
The actual return location of rental is branch	Property type (entity-to-entity), Cardinality laws	property type actual return location of rental exists	✓	✓	✓	✓	✓
		the domain of actual return location of rental is returned rental	✓	✓	✓	✓	✓
		the range of actual return location of rental is branch	✓	✓	✓	✓	✓

Table G.5: RAC preservation of fact statements of Property Types for each design sprint

Overview of Generative AI Tools Used

No generative AI tools were used in the creation of this work.

Übersicht verwendeter Hilfsmittel

Bei der Erstellung dieser Arbeit wurden keine generativen KI-Tools verwendet.

List of Figures

List of Tables

Glossary

ADSRM Agile Design Science Research Methodology. 5

Alf Action Language for Foundational UML. 84, 85

AS agile sprint. 9

CIM computation-independent model *or* computation independent model. 2, 12, 34

CSCD conceptual schema-centric development. 42

DEMO Design and Engineering Methodology for Organizations. 17

DEMO-SL DEMO Specification Language. 18

FM DEMO fact model. 49, 141

fUML Foundational UML. 51, 83

GOSL General Ontology Specification Language. 141

LCDP low-code development platform. 19

MDA Model Driven Architecture®. 12

MDE model-driven engineering *or* model-driven software engineering. 11

OFD Object Fact Diagram. 49, 141

PIM platform-independent model *or* platform independent model. 2, 12, 37

PSM platform-specific model *or* platform specific model. 2, 12, 37

RAC Rent-A-Car. 53

UML Unified Modeling Language. 59

xUML Executable UML. 51

Bibliography

[1] D. L. Olson and S. Kesharwani, "Enterprise information system trends," in *Enterprise Information Systems*, J. Filipe and J. Cordeiro, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 3–14.

[2] S. Mithas and F. W. McFarlan, "What is digital intelligence?" *IT Professional*, vol. 19, no. 4, pp. 3–6, 2017.

[3] R. Dove, "Agile enterprise cornerstones: Knowledge, values, and response ability," in *Business Agility and Information Technology Diffusion*, R. L. Baskerville, L. Mathiassen, J. Pries-Heje, and J. I. DeGross, Eds. Boston, MA: Springer US, 2005, pp. 313–330.

[4] "Taming the digital dragon: The 2014 CIO agenda," Gartner Executive Programs, Technical Report, 2014. [Online]. Available: https://www.gartner.com/imagesrv/cio/pdf/cio_agenda_insights2014.pdf

[5] M. R. Krouwel, M. Op 't Land, and H. A. Proper, "From enterprise models to low-code applications: mapping DEMO to Mendix; illustrated in the social housing domain," *Softw. Syst. Model.*, vol. 23, no. 4, p. 837–864, Apr. 2024. [Online]. Available: https://doi.org/10.1007/s10270-024-01156-2

[6] K. Hinkelmann, A. Gerber, D. Karagiannis, A. V. der Merve, and R. Woitsch, "A new paradigm for the continuous alignment of business and it: Combining enterprise architecture modelling and enterprise ontology," *A new paradigm for the continuous alignment of business and IT: Combining enterprise architecture modelling and enterprise ontology*, vol. 79, pp. 77–86, June 2016. [Online]. Available: http://eprints.cs.univie.ac.at/4907/

[7] M. Finio and A. Downie, "What are enterprise applications?" May 2024. [Online]. Available: https://www.ibm.com/think/topics/enterprise-applications

[8] Z. Hemel, L. Kats, D. Groenewegen, and E. Visser, "Code generation by model transformation: A case study in transformation modularity," *Software and Systems Modeling*, vol. 9, pp. 375–402, Jun. 2009.

[9] F. Truyen, "The fast guide to model driven architecture, the basics of model driven architecture (MDA)," Jan. 2006.

[10] M. Brambilla, J. Cabot, and M. Wimmer, *Model-driven software engineering in practice*, 2nd ed., ser. Synthesis lectures on software engineering. [San Rafael, Calif.]: Morgan & Claypool Publishers, 2017.

[11] M. Kardos and M. Drozdova, "Analytical method of CIM to PIM transformation in model driven architecture (MDA)," *Journal of Information and Organizational Sciences*, vol. 34, no. 1, Jun. 2010.

[12] G. Sedrakyan and M. Snoeck, "A PIM-to-code requirements engineering framework," in *MODELSWARD 2013 - Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development*, Feb. 2013.

[13] R. Goncalves, C. Agostinho, E. Silva, Y. Ducq, G. Zacharewicz, H. Bazoun, and H. Boyé, "MDA/MDI model transformation: Application to MDSEA," Sep. 2012.

[14] N. Elleuch Ben Ayed and H. Ben-Abdallah, *From an Annotated BPMN Model to a Use Case Diagram: DESTINY Methodology*, Jan. 2020, pp. 379–390.

[15] M. Argañaraz, A. Funes, and A. Dasso, *An MDA Approach to Business Process Model Transformations*, Jan. 2010, pp. 24–48.

[16] N. Silega Martínez, M. Noguera, Y. Rogozov, V. Lapshin, and T. Galban, "Transformation from CIM to PIM: A systematic mapping," *IEEE Access*, vol. PP, pp. 1–1, Jan. 2022.

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

[18] A. Hevner, "A three cycle view of design science research," *Scandinavian Journal of Information Systems*, vol. 19, no. 2, Jan. 2007. [Online]. Available: https://aisel.aisnet.org/sjis/vol19/iss2/4

[19] K. Conboy, R. Gleasure, and E. Cullina, "Agile design science research," May 2015.

[20] K. Peffers, T. Tuunanen, M. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *Journal of Management Information Systems*, vol. 24, pp. 45–77, Jan. 2007.

[21] M. Krouwel, M. Op 't Land, and H. Proper, "Presentation: Generating low-code applications from enterprise ontology," Nov. 2022.

[22] E. Domínguez and M. A. Zapata, "Mappings and interoperability: A meta-modelling approach," in *Proceedings of the First International Conference on Advances in Information Systems*, ser. ADVIS '00. Berlin, Heidelberg: Springer-Verlag, 2000, p. 352–362.

[23] E. Domínguez, Á. L. Rubio, and M. A. Zapata, "Mapping models between difierent modeling languages," Aug. 2002.

[24] K. Lano, S. Kolahdouz Rahimi, and I. Poernomo, "Comparative evaluation of model transformation specification approaches," *Int J Software Informatics*, vol. 6, pp. 233–269, Jan. 2012.

[25] J. Krogstie and A. Sølvberg, *Information Systems Engineering: Conceptual Modeling in a quality perspective.* The Norwegian University of Science and Technology, Dec. 2001.

[26] A. G. Kleppe, W. Bast, and J. Warmer, *MDA explained: The Model Driven Architecture; practice and promise.* Addison-Wesley, 2007.

[27] "MDA guide rev. 2.0," Jun. 2014. [Online]. Available: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01

[28] E. Marcos, C. J. Acuña, and C. E. Cuesta, "Integrating software architecture into a mda framework," in *Software Architecture*, V. Gruhn and F. Oquendo, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 127–143.

[29] J. Dietz, J. Hoogervorst, A. Albani, D. Aveiro, E. Babkin, J. Barjis, A. Caetano, P. Huysmans, J. Iijima, S. van Kervel, H. Mulder, M. Op 't Land, H. Proper, J. Sanz, L. Terlouw, J. Tribolet, J. Verelst, and R. Winter, "The discipline of enterprise engineering," *International Journal of Organisational Design and Engineering*, vol. 3, pp. 86–114, May 2013.

[30] R. Rohatyński, "A new approach to modeling of the design processes," in *DS 31: Proceedings of ICED 03, the 14th International Conference on Engineering Design, Stockholm*, ser. ICED, A. Folkeson, K. Gralen, M. Norell, and U. Sellgren, Eds. Stockholm: The Design Society, Aug. 2003, pp. 673–674.

[31] F. Davidoff, "Understanding contexts: How explanatory theories can help," *Implementation Science*, vol. 14, no. 1, Mar. 2019.

[32] "Semantics of business vocabulary and business rules: Version 1.5," Oct. 2019. [Online]. Available: https://www.omg.org/spec/SBVR/1.5/About-SBVR

[33] J. Dietz and H. Mulder, *Enterprise Ontology: A Human-Centric Approach to Understanding the Essence of Organisation.* Springer Nature Switzerland AG, Jan. 2020.

[34] J. L. G. Dietz, "On the nature of business rules," in *Advances in Enterprise Engineering I*, J. L. G. Dietz, A. Albani, and J. Barjis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–15.

[35] ——, "DEMO Specification Language 4.9.1," Sep. 2024. [Online]. Available: https://ee-institute.org/download/demo-specification-language-4-9-1/

[36] M. Krouwel, "On the design of enterprise ontology-driven software development," Ph.D. dissertation, Maastricht University, Dec. 2023.

[37] IBM, "What is low-code?" Aug. 2024. [Online]. Available: https://www.ibm.com/topics/low-code

[38] Y. Luo, P. Liang, C. Wang, M. Shahin, and J. Zhan, "Characteristics and challenges of low-code development: The practitioners' perspective," in *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ser. ESEM '21. New York, NY, USA: Association for Computing Machinery, Oct. 2021. [Online]. Available: https://doi.org/10.1145/3475716.3475782

[39] A. Sahay, A. Indamutsa, D. Di Ruscio, and A. Pierantonio, "Supporting the understanding and comparison of low-code development platforms," in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug. 2020, pp. 171–178.

[40] A. C. Bock and U. Frank, "Low-code platform," *Business & Information Systems Engineering*, vol. 63, pp. 733–740, Dec. 2021. [Online]. Available: https://doi.org/10.1007/s12599-021-00726-8

[41] K. Rokis and M. Kirikova, "Challenges of low-code/no-code software development: A literature review," in *Perspectives in Business Informatics Research*, Ē. Nazaruka, K. Sandkuhl, and U. Seigerroth, Eds. Cham: Springer International Publishing, 2022, pp. 3–17.

[42] D. Di Ruscio, D. Kolovos, J. de Lara, A. Pierantonio, M. Tisi, and M. Wimmer, "Low-code development and model-driven engineering: Two sides of the same coin?" *Softw. Syst. Model.*, vol. 21, no. 2, p. 437–446, Apr. 2022. [Online]. Available: https://doi.org/10.1007/s10270-021-00970-2

[43] O. Matvitskyy, K. Davis, and A. Jain, "Magic quadrant for enterprise low-code application platforms," Gartner Research, Technical Report, Oct. 2024. [Online]. Available: https://www.gartner.com/en/documents/5844247

[44] J. Sijtstra, "Quantifying the effectiveness of low-code development platforms in the Dutch public sector," Master's Thesis, LIACS, Leiden University, 2022. [Online]. Available: https://theses.liacs.nl/2221

[45] A. Li, "Google shutting down App Maker for enterprise in 2021," Jan. 2020. [Online]. Available: https://9to5google.com/2020/01/27/google-app-maker-shutdown

[46] J. Cabot, "Positioning of the low-code movement within the field of model-driven engineering," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, ser. MODELS '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3417990.3420210

[47] M. Tisi, J.-M. Mottu, D. S. Kolovos, J. de Lara, E. Guerra, D. Di Ruscio, A. Pierantonio, and M. Wimmer, "Lowcomote: Training the next generation of experts in scalable low-code engineering platforms," in *Proceedings of the STAF 2019 Co-Located Events: 1st STAF Junior Researcher Community Event (JRCE-STAF 2019), 2nd International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems (MDE@DeRun 2019) and 1st STAF Research Project Showcase Workshop (RPS-STAF 2019)*, 2019, pp. 67–72. [Online]. Available: http://ceur-ws.org/Vol-2405/

[48] A.-K. Hermann, L. König, E. Burger, and R. Reussner, "Towards integrating low-code in view-based development," in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS Companion '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 866–875. [Online]. Available: https://doi.org/10.1145/3652620.3688333

[49] M. Missikoff, "A simple methodology for model-driven business innovation and low code implementation," 2020. [Online]. Available: https://arxiv.org/abs/2010.11611

[50] I. Alfonso, A. Conrardy, A. Sulejmani, A. Nirumand, F. Ul Haq, M. Gomez-Vazquez, J.-S. Sottet, and J. Cabot, "Building BESSER: An open-source low-code platform," in *Enterprise, Business-Process and Information Systems Modeling*, H. van der Aa, D. Bork, R. Schmidt, and A. Sturm, Eds. Cham: Springer Nature Switzerland, 2024, pp. 203–212.

[51] J. Cabot, "BESSER – a brand new open-source low-code platform," Jan. 2024. [Online]. Available: https://modeling-languages.com/lowcode-opensource-besser/

[52] ——, "Low-modeling of software systems," in *Software Technologies*, H.-G. Fill, F. J. Domínguez Mayo, M. van Sinderen, and L. A. Maciaszek, Eds. Cham: Springer Nature Switzerland, 2024, pp. 19–28.

[53] "Welcome to BESSER's documentation," 2025. [Online]. Available: https://besser.readthedocs.io/en/latest/

[54] M. A. T. Mulder, "Enabling the automatic verication and exchange of DEMO models," Ph.D. dissertation, Radboud University, 2022.

[55] S. J. H. van Kervel, "Ontology driven enterprise information systems engineering," Ph.D. dissertation, TU Delft, 2012. [Online]. Available: https://resolver.tudelft.nl/uuid:8c42378a-8769-4a48-a7fb-f5457ede0759

[56] S. Guerreiro, S. J. H. van Kervel, A. Vasconcelos, and J. Tribolet, "Executing enterprise dynamic systems control with the Demo processor: The business transactions transition space validation," in *Knowledge and Technologies in Innovative Information Systems*, H. Rahman, A. Mesquita, I. Ramos, and B. Pernici, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 97–112.

[57] J. Hintzen, S. J. H. van Kervel, T. v. Meeuwen, J. Vermolen, and B. Zijlstra, "A professional case management system in production, modeled and implemented using DEMO," *The 8th Workshop on Transformation & Engineering of Enterprises (TEE 2014)*, 2014. [Online]. Available: https://ceur-ws.org/Vol-1182/

[58] M. Skotnica, S. J. H. van Kervel, and R. Pergl, "Towards the ontological foundations for the software executable DEMO action and fact models," in *Advances in Enterprise Engineering X*, D. Aveiro, R. Pergl, and D. Gouveia, Eds. Cham: Springer International Publishing, 2016, pp. 151–165.

[59] M. Skotnica, S. J. van Kervel, and R. Pergl, "A DEMO machine - a formal foundation for execution of DEMO models," in *Advances in Enterprise Engineering XI: 7th Enterprise Engineering Working Conference, EEWC 2017, Antwerp, Belgium, May 8-12, 2017, Proceedings 7*. Springer, 2017, pp. 18–32.

[60] V. Freitas, D. Pinto, V. Caires, L. Tadeu, and D. Aveiro, "The DISME low-code platform - from simple diagram creation to system execution," *Proceedings of the 22nd CIAO! Doctoral Consortium, and Enterprise Engineering Working Conference Forum 2022 co-located with 12th Enterprise Engineering Working Conference (EEWC 2022)*, 2022. [Online]. Available: https://ceur-ws.org/Vol-3388/

[61] H. Snyder, "Literature review as a research methodology: An overview and guidelines," *Journal of Business Research*, vol. 104, pp. 333–339, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0148296319304564

[62] Y. Rhazali, A. Hachimi, I. Chana, M. Lahmer, and A. Rhattoy, *Automate Model Transformation From CIM to PIM up to PSM in Model-Driven Architecture*, Oct. 2019, p. 22.

[63] V. De Castro, E. Marcos, and J. M. Vara, "Applying CIM-to-PIM model transformations for the service-oriented development of information systems," *Inf. Softw. Technol.*, vol. 53, no. 1, p. 87–105, Jan. 2011. [Online]. Available: https://doi.org/10.1016/j.infsof.2010.09.002

[64] A. Bozzon, M. Brambilla, and P. Fraternali, "Conceptual modeling of multimedia search applications using rich process models," in *Proceedings of the 9th International Conference on Web Engineering*, ser. ICWE '9. Berlin, Heidelberg: Springer-Verlag, 2009, p. 315–329. [Online]. Available: https://doi.org/10.1007/978-3-642-02818-2_26

[65] W. Zhang, H. Mei, H. Zhao, and J. Yang, "Transformation from CIM to PIM: A feature-oriented component-based approach," *Model Driven Engineering Languages and Systems*, p. 248–263, 2005.

[66] Y. Singh and M. Sood, "The impact of the computational independent model for enterprise information system development," *International Journal of Computer Applications*, vol. 11, Dec. 2010.

[67] M. Melouk, Y. Rhazali, and H. Youssef, "An approach for transforming CIM to PIM up to PSM in MDA," *Procedia Computer Science*, vol. 170, pp. 869–874, 2020, the 11th International Conference on Ambient Systems, Networks and Technologies (ANT) / The 3rd International Conference on Emerging Data and Industry 4.0 (EDI40) / Affiliated Workshops. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050920305603

[68] Y. Rhazali, Y. Hadi, and A. Mouloudi, "Model transformation with ATL into MDA from CIM to PIM structured through MVC," *Procedia Computer Science*, vol. 83, pp. 1096–1101, Dec. 2016.

[69] S. Kherraf, E. Lefebvre, and W. Suryn, "Transformation from CIM to PIM using patterns and archetypes," Mar. 2008.

[70] G. Sebastián, J. A. Gallud, and R. Tesoriero, "Code generation using model driven architecture: A systematic mapping study," *Journal of Computer Languages*, vol. 56, p. 100935, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2590118419300607

[71] Y. Rhazali, Y. Hadi, and A. Mouloudi, "A model transformation in MDA from CIM to PIM represented by web models through SoaML and IFML," in *2016 4th IEEE International Colloquium on Information Science and Technology (CiSt)*, 2016, pp. 116–121.

[72] ——, "Disciplined approach for transformation CIM to PIM in MDA," in *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, 2015, pp. 312–320.

[73] Y. Liu and Y. Ma, "An approach for MDA model transformation based on JEE platform," in *2008 4th International Conference on Wireless Communications, Networking and Mobile Computing*, 2008, pp. 1–4.

[74] X.-m. Yang, P. Gu, and H. Dai, "Mapping approach for model transformation of MDA based on XMI/XML platform," in *2009 First International Workshop on Education Technology and Computer Science*, vol. 2, 2009, pp. 1016–1019.

[75] Y. Xiao-mei, D. Heng, and G. Ping, "Mapping approach for model transformation of MDA based on xUML," in *2009 4th International Conference on Computer Science & Education*, 2009, pp. 862–865.

[76] Y. Rhazali, Y. Hadi, and A. Mouloudi, "A methodology for transforming CIM to PIM through UML: From business view to information system view," in *2015 Third World Conference on Complex Systems (WCCS)*, 2015, pp. 1–6.

[77] ——, "Transformation approach CIM to PIM: from business processes models to state machine and package models," in *2015 International Conference on Open Source Software Computing (OSSCOM)*, 2015, pp. 1–6.

[78] E. Soler, J. Trujillo, E. Fernandez-Medina, and M. Piattini, "A set of QVT relations to transform PIM to PSM in the design of secure data warehouses," in *The Second International Conference on Availability, Reliability and Security (ARES'07)*, 2007, pp. 644–654.

[79] E. W. Dijkstra, *On the Role of Scientific Thought*. New York, NY: Springer New York, 1982, pp. 60–66. [Online]. Available: https://doi.org/10.1007/978-1-4612-5695-3_12

[80] C. Reade, *Elements of functional programming*. Addison-Wesley, 1989.

[81] J. Ingeno, *Software Architect's Handbook: Become a successful software architect by implementing effective architecture concepts*. Packt Publishing, 2018.

[82] J. E. Chukwuere, "Theoretical and conceptual framework: A critical part of information systems research process and writing," *Review of International Geographical Education (RIGEO)*, vol. 11, no. 9, p. 2678–2683, Oct. 2021. [Online]. Available: https://rigeo.org/menu-script/index.php/rigeo/article/view/2205

[83] A. Olivé, "Conceptual schema-centric development: A grand challenge for information systems research," vol. 3520, Jun. 2005, pp. 1–15.

[84] M. Fowler, *Patterns of Enterprise Application Architecture*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[85] M. Richards, *Software architecture patterns*. O'Reilly Media, 2015.

[86] "MDA specifications." [Online]. Available: https://www.omg.org/mda/specs.htm

[87] H. Störrle, "How are conceptual models used in industrial software development? a descriptive survey," in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 160–169. [Online]. Available: https://doi.org/10.1145/3084226.3084256

[88] "Unified Modeling Language: Version 2.5.1," Dec. 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1/

[89] S. J. Mellor, M. Balcer, and I. Jacoboson, *Executable UML: A Foundation for Model-Driven Architectures*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[90] "Semantics of a foundational subset for executable UML models," Jun. 2021. [Online]. Available: https://www.omg.org/spec/FUML/1.5

[91] "Building a responsive web app," Sep. 2024. [Online]. Available: https://docs.mendix.com/quickstarts/responsive-web-app/

[92] "Data storage," Aug. 2024. [Online]. Available: https://docs.mendix.com/refguide/data-storage/#2-supported-databases

[93] "System requirements," Apr. 2025. [Online]. Available: https://docs.mendix.com/refguide/system-requirements/#databases

[94] "Enterprise runtime architecture: Mendix evaluation guide," Aug. 2024. [Online]. Available: https://www.mendix.com/evaluation-guide/enterprise-capabilities/architecture/runtime-architecture/

[95] J. L. G. Dietz and H. B. F. Mulder, *Exercise: Case Rent-A-Car*. Cham: Springer International Publishing, 2020, pp. 323–348. [Online]. Available: https://doi.org/10.1007/978-3-030-38854-6_15

[96] D. C. Mansourov and Nikolai, "EU-rent example." [Online]. Available: https://kdmanalytics.com/sbvr/EU-Rent.html

[97] M. Seidl, M. Scholz, C. Huemer, and G. Kappel, *UML @ Classroom - An Introduction to Object-Oriented Modeling*, ser. Undergraduate Topics in Computer Science. Springer, 2015. [Online]. Available: https://doi.org/10.1007/978-3-319-12742-2

[98] K. Żyła, A. Ulidowski, J. Wrzos, B. Włodarczyk, K. Krocz, and P. Drozd, "UML – a survey on technical university students in Lublin," *Journal of Computer Sciences Institute*, vol. 13, p. 279–282, Dec. 2019.

[99] "Object Constraint Language (OCL), version 2.4," Feb. 2014. [Online]. Available: https://www.omg.org/spec/OCL/2.4/

[100] L. Starr, A. Mangogna, and S. Mellor, *Models to Code: With No Mysterious Gaps*, 1st ed. USA: Apress, 2017.

[101] M. Gelfond and V. Lifschitz, "Action languages," *ETAI*, vol. 3, Apr. 1999.

[102] "Action language for Foundational UML (Alf): Version 1.1," Jul. 2017. [Online]. Available: https://www.omg.org/spec/ALF/1.1

[103] J. Cabot, "The new executable UML standards: fUML and Alf," Jan. 2011. [Online]. Available: https://modeling-languages.com/new-executable-uml-standards-fuml-and-alf/

[104] "Data in the domain model," Mar. 2025. [Online]. Available: https://docs.mendix.com/refguide/domain-model/

[105] "Microflows," Feb. 2025. [Online]. Available: https://docs.mendix.com/refguide/microflows/

[106] "Expressions," Mar. 2025. [Online]. Available: https://docs.mendix.com/refguide/expressions/

[107] "Activities," Mar. 2025. [Online]. Available: https://docs.mendix.com/refguide/activities/

[108] "Application logic," Aug. 2024. [Online]. Available: https://docs.mendix.com/refguide/application-logic/

[109] "Workflows," Sep. 2024. [Online]. Available: https://docs.mendix.com/refguide/workflows/

[110] "Workflow elements," Sep 2024. [Online]. Available: https://docs.mendix.com/refguide/workflow-elements/

[111] "Workflow elements," Sep. 2024. [Online]. Available: https://docs.mendix.com/refguide/workflow-elements/

[112] "Pages," Aug. 2024. [Online]. Available: https://docs.mendix.com/refguide/pages/

[113] "Pages in the Mendix metamodel," Aug. 2024. [Online]. Available: https://docs.mendix.com/apidocs-mxsdk/mxsdk/pages-metamodel/

[114] F. Heylighen, "Principles of systems and cybernetics: an evolutionary perspective," *Cybernetics and Systems '92*, p. 3–10, 1992. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=7a069d802354f2fbd01cf4ef5155a6492640c575

[115] "Domain model in the mendix metamodel," Aug. 2024. [Online]. Available: https://docs.mendix.com/apidocs-mxsdk/mxsdk/domain-model-metamodel/

[116] "Microflows in the mendix metamodel," Aug. 2024. [Online]. Available: https://docs.mendix.com/apidocs-mxsdk/mxsdk/microflows-metamodel/

[117] "Namespace workflows." [Online]. Available: https://apidocs.rnd.mendix.com/modelsdk/latest/modules/workflows.html