UNIVERSIDAD
NACIONAL
DE COLOMBIA

A METHOD FOR OBTAINING FORMAL SOFTWARE SPECIFICATIONS FROM KNOWLEDGE REPRESENTATION LANGUAGES

Roberto Antonio Manjarrés Betancur

Universidad Nacional de Colombia

Facultad de Minas, Departamento de Ciencias de la Computación y la Decisión

Medellín, Colombia

2023

A METHOD FOR OBTAINING FORMAL SOFTWARE SPECIFICATIONS FROM KNOWLEDGE REPRESENTATION LANGUAGES

Roberto Antonio Manjarrés Betancur

Tesis presentada como requisito parcial para optar al título de:

Doctor en Ingeniería – Sistemas e Informática

Director:
Ph.D., Carlos Mario Zapata Jaramillo
Codirectora:
Ph.D., Bell Manrique Losada

Línea de Investigación:

Ingeniería de Software

Grupo de Investigación:

Grupo de Investigación en Lenguajes Computacionales

Universidad Nacional de Colombia

Facultad de Minas, Departamento de Ciencias de la Computación y la Decisión

Medellín, Colombia

2023

To my parents, family, and friends.

They constantly make me want to do better.

Declaración de obra original

Yo declaro lo siguiente:

He leído el Acuerdo 035 de 2003 del Consejo Académico de la Universidad Nacional. «Reglamento sobre propiedad intelectual» y la Normatividad Nacional relacionada al respeto de los derechos de autor. Esta disertación representa mi trabajo original, excepto donde he reconocido las ideas, las palabras, o materiales de otros autores.

Cuando se han presentado ideas o palabras de otros autores en esta disertación, he realizado su respectivo reconocimiento aplicando correctamente los esquemas de citas y referencias bibliográficas en el estilo requerido.

He obtenido el permiso del autor o editor para incluir cualquier material con derechos de autor (por ejemplo, tablas, figuras, instrumentos de encuesta o grandes porciones de texto).

Por último, he sometido esta disertación a la herramienta de integridad académica, definida por la universidad.

Fecha 15/05/2024

Resumen

Un método para la obtención de especificaciones formales de software a partir de leguajes de representación del conocimiento

Los analistas de software usan lenguajes de representación del conocimiento para caracterizar el conocimiento proveniente de los interesados en la fase de ingeniería de requisitos. Estos lenguajes comprenden modelos de software basados en el lenguaje unificado de modelado y métodos estructurados, incluyendo diagramas entidad-relación, diagrama de clases, diagramas causa-efecto y grafos conceptuales. Si bien estos modelos se usan para representar características básicas de un sistema de software de manera efectiva, estos modelos todavía tienen limitaciones al representar el comportamiento y componentes de sistemas de software complejos. Las especificaciones formales incluyen textos, matemáticas y notaciones basadas en lógica para representar el conocimiento relacionado a un dominio específico. Los analistas de software usan especificaciones formales para mitigar el impacto de la ambigüedad inherente en los lenguajes de representación del conocimiento tradicionales, facilitando procesos computacionales, inferencia y la validación no ambigua de conocimiento. Sin embargo, la generación y validación de especificaciones formales requiere experiencia en matemáticas y lógica por parte de los analistas de software e interesados, impidiendo una comunicación y validación efectiva del dominio de software. En esta Tesis de Doctorado, proponemos un novedoso enfoque para cerrar la brecha entre los lenguajes formales y los lenguajes de representación de conocimiento. Este enfoque contribuye a los campos de representación del conocimiento y lenguajes formales, permitiéndole a los analistas de software generar especificaciones formales precisas y completas desde cualquier lenguaje de representación del conocimiento sin importar su experiencia en matemáticas y lógica. Este enfoque facilita la validación de especificaciones a los interesados. El método propuesto es validado desarrollando tres casos de estudio. Este enfoque es un nuevo producto de trabajo para la representación de conocimiento no ambigua. Además, este método está diseñado para apoyar la fase de ingeniería de requisitos como una nueva herramienta para el analista de software.

Palabras clave: Representación del conocimiento, Ingeniería de requisitos, Modelo de software, Especificación formal.

Abstract

Software analysts use knowledge representation languages for characterizing the knowledge from stakeholders in the requirements engineering phase. Such languages encompass software models based on the unified modeling language and structured methods, including entity-relationship diagrams, class diagrams, cause-and-effect diagrams, and conceptual graphs. While such models are used for effectively representing the basic features of a software system, they still fail on including the behavior and components coming from complex software systems. Formal specifications comprise texts, math, and logic-based notations for representing the knowledge of a given domain. Software analysts use formal specifications for mitigating the impact of the ambiguity coming from traditional knowledge representation languages, easing computational processes, inference, and unambiguous validation of knowledge. However, the generation and validation of formal specifications require expertise in mathematics and logic from both software analysts and stakeholders, thereby impeding effective communication and validation of the software domain. In this Ph.D. Thesis, we propose a novel approach for bridging the gap between formal languages and knowledge representation languages. Our approach contributes to the fields of knowledge representation and formal languages by allowing software analysts for generating precise and comprehensive formal specifications from any knowledge representation language, regardless of their mathematical and logical proficiency. This approach eases the validation of resulting specifications by stakeholders. The proposed method is validated by developing three case studies. Our approach is a new work product for unambiguous knowledge representation. Also, our method is designed for supporting the requirements engineering phase as a new tool for software analysts.

Keywords: Knowledge representation, Requirements Engineering, Software model, Formal specification.

Content

Page

1. Introduction 12
2. Background 15
2.1 Conceptual Framework 15
2.1.1 Knowledge Representation Languages 15
2.1.2 Formal specification languages 16
2.1.3 Meta-Models for Software Knowledge Representation 18
2.1.3 Pre-conceptual Schemas....18
2.1.4 Requirements Engineering 19
2.2 Ph.D. Thesis Focus....20
2.3 Research Methodology 21
2.3.1 Exploration 21
2.3.2 Problem Formulation 21
2.3.3 Solution 21
2.3.4 Validation 22
3. Research Problem....23
3.1 Motivation 23
3.2 State of the Art....24
3.2.1 Planning Literature Review....24
3.2.2 Executing Systematic Literature Review....24
3.3 Problem Statement 32
3.4 Objectives....33
3.4.1 General Objective....33
3.4.2 Specific Objectives 33
3.5 Justification....34

4. A method for transforming KRL-FSL pairs 35
4.1 Characterizing KRLs in the context of RE 35
4.2 Characterizing FSLs in the context of RE 36
4.3 Defining a meta-model for KRLs in the context of RE 37
4.4 Defining a meta-model for FSLs in the context of RE 38
4.5 Defining a meta-model for transformation rules for KRL-FSL pairs 41
4.6 Defining a method for transforming KRL-FSL pairs 43
5. Validation 46
5.1 Case study planning 46
5.1.1 Objective 46
5.1.2 Case studies 46
5.1.3 Theory 47
5.1.4 Validation research questions 47
5.1.5 Methods 48
5.2 Case study validation 48
5.2.1 Performing a KRL-to-FSL transformation 48
5.2.2 Performing a FSL-to-KRL transformation 66
5.3 Work products 71
5.3.1 Meta-models 71
5.3.2 A prototype for transforming KRL-FSL pairs 71
6. Conclusions and challenges 72
6.1 Conclusions 72
6.2 Challenges 74
References 75

List of Figures

Figure 2-1. O-A-V example. The Authors adapted from Devedžic et al. (2009). 15
Figure 2-2. Semantic network example. The Authors adapted from Devedžic et al. (2009). 16
Figure 2-3. Pre-conceptual schemas structures. The Authors adapted from (Noreña C., 2020) 18
Figure 4-1. A meta-model for KRLs. The Authors 38
Figure 4-2. A meta-model for FSLs. The Authors. 41
Figure 4- 3. A meta-model for model-to-model transformation rules for KRL-FSL pairs. The Authors. 43
Figure 4-4. A method for transforming KRL-FSL pairs. The Authors 44
Figure 5-1. CS1 source model representation. The Authors. 48
Figure 5- 2. Process diagram representation (Zapata & Arango, 2009) 56
Figure 5-3. Class diagram representation (Zapata & Arango, 2009) 57
Figure 5-4. Case diagram representation (Zapata & Arango, 2009) 57
Figure 5-5. CS2 target model representation. The Authors 69

List of Tables

Table 3-1: Study criteria. The Authors 25
Table 3-2: Primary studies summary. The Authors. 29
Table 4-1: KRL-FSL pair equivalencies. The Authors. 42
Table 5-1: CS1 represented elements. The Authors. 54
Table 5-2: CS1 CM values summary. The Authors. 56
Table 5-3: CS2 represented elements. The Authors. 63
Table 5-4: CS2 CM values summary. The Authors. 65
Table 5-5: CS3 transformation process summary. The Authors. 70
Table 5-6: CS2 CM values summary. The Authors. 71

1.Introduction

Software analysts employ Knowledge Representation Languages (KRLs) to effectively characterize and capture the knowledge provided by stakeholders during the requirements engineering phase (Ang & Hartley, 2007; Dubois et al., 1986). Such languages include various software models, such as entity-relationship diagrams, class diagrams, cause-effect diagrams, and conceptual graphs, based on the Unified Modeling Language (UML) and structured methods (Karolita et al., 2023). KRLs ease the systematic organization and representation of stakeholder knowledge, enabling a comprehensive understanding of the requirements of a software system. Software analysts ensure the captured knowledge is aligned with the needs and expectations coming from stakeholders while using such languages, serving as a foundation for the subsequent phases of the software development process (Sabri, 2015; Sonbol et al., 2020).

Despite the effectiveness of such models in representing the basic features of software systems, some limitations still remain for representing complex software systems (Maio, 2021). Such models use to fail on representing the interdependencies, interactions, and relationships among the subsystems, intricated components, and processes composing a complex software system, hardening for a comprehensive representation of the dynamic behavior of complex software systems (Popescu & Dumitrache, 2023).

Formal Specifications (FSs) provide an unambiguous and rigorous representation of the knowledge coming from a software system. Such specifications comprise well-defined textual, mathematical, and logical notations, facilitating computational processes, automated reasoning, verification, and validation of captured knowledge (Pang et al., 2016). KRLs comprise natural language and graphical elements for representing specific-domain knowledge, easing the understanding and validation of software domains by the stakeholders (Ang & Hartley, 2007; Dubois et al., 1986). Software analysts use FSs for identifying, capturing, and communicating complex software systems, including their static

and dynamic features. Also, such specifications allow software analysts for mitigating the impact of misunderstanding and ambiguity, enhancing their communication with the stakeholders (Pang *et al.*, 2016).

FSs are strongly based on logical and mathematical notation. Also, such specifications include specialized terminology, domain-specific languages, and symbols, so software analysts and stakeholders are challenged to grasp a comprehensive understanding of the captured knowledge (Alkhammash, 2020; Rabinia & Ghanavati, 2017). Thus, a simpler language becomes necessary for both software analysts and stakeholders for enhancing their understanding of formal specifications and the gathered knowledge, easing the communication and validation of the software domain.

Consequently, in this Ph.D. Thesis we propose a novel method for obtaining an FS from a knowledge representation model and vice versa regardless of the nature of the KRL and the FS. Also, we analyze and characterize state-of-the-art proposals coming from the KRL and FS fields to identify their features and equivalences, so we facilitate the generation of an adequate FS, minimizing the need for a deeper understanding of mathematics and logic, and allowing software analysts and stakeholders to validate the produced FS and KRL.

Some authors propose KRL-FL pairs for representing software domains, abstracting some of the complex aspects of the FSs. However, such pairs are limited to a specific type of domain and KRL-FL pair, hardening generalization among other types of domains and KRLs (Alkhammash, 2020; Ang & Hartley, 2007). Also, some KRL-FL pairs only include one-way transformation rules, *i.e.,* from KRL to FL, minimizing the flexibility of the proposals as software analysts may not transform FLs into KRLs. Our approach overcomes such limitations by using a meta-model including concepts such as nodes, names, features, types, processes, sequences, classes, relationships, and constraints. Such concepts are designed for representing recurrent elements coming from well-known KRLs and FSs, eliminating the dependency on specific domains, KRLs, and FSs. Also, we provide a set of heuristic rules for generating such FSs. Our approach allows software analysts for generating precise and comprehensive FSs from any KRL, regardless of their mathematical and logical background. Also, we provide an easier graphical language, helping them for communicating and validating the produced FSs and KRLs with stakeholders.

We follow the guidelines of the empirically based technology transfer methodology (Wohlin et al., 2012) and the design science methodology for information systems and software engineering (Wieringa & Wieringa, 2014) for performing our research, including four phases: *exploration* where we perform a systematic literature review for identifying and characterizing relevant studies to our proposal; *problem formulation* where we state the problem; *solution* where we propose a new method for obtaining FS from any KRL; *validation* where we validate our proposal.

We validate our proposal by using three case studies including state-of-the-art object-oriented KRL-FL pairs. The validation process is performed by using the experimental process of software engineering: planning, executing, and analyzing a mechanism experiment (Wieringa, 2014; Wohlin *et al.*, 2012). Our method is a new work product for unambiguous knowledge representation. Such a method supports software analysts in the generation of FSs during the requirements engineering phase.

This Ph.D. Thesis is structured as follows: in Chapter 2, background, we present the conceptual framework, Thesis focus, and methodology; in Chapter 3, research problem, we describe our motivation, state of the art, problem statement, research question, hypothesis, objectives, and justification; in Chapter 4, a method for obtaining formal software specifications and knowledge representation languages, we propose a novel method for supporting the generation of formal software specifications and knowledge representation languages, comprising a meta-model for representing any KRL, a meta-model for representing any FS, and a set of heuristic rules for generating FSs and KRLs; in Chapter 5, validation, we experimentally evaluate our method and present the derived work products; in Chapter 6, conclusions and challenges, we discuss contributions and challenges.

2. Background

2.1 Conceptual Framework

2.1.1 Knowledge Representation Languages

KRLs comprise a wide set of graphical and textual components allowing software analysts for representing key components of a software domain. Such languages play a key role in the RE phase as software analysts may easily organize and validate the gathered software domain knowledge with stakeholders. Also, the produced KRs provide the foundation for subsequent phases in the software development process.

Software systems comprise several intertwined components which may be represented by using some KRLs, including entity-relationship diagrams, class diagrams, sequence diagrams, conceptual graphs, and other structured methods. Such representations allow software analysts for representing the dependencies, components, and constraints related to a software system. Also, the formal nature of KRLs allows software analysts for performing reasoning, inferencing, and automated analysis and validation of the gathered knowledge.

Some examples of KRL approaches for representing concepts, relationships, and constraints coming from a specific software domain are described as follows:

Object-Attribute-Value (O-A-V) triplets are used for representing facts about objects and their attributes. An O-A-V triplet includes an attribute value of an object, *e.g.*, the English phrase “the color of the ball is yellow” may be written in O-A-V form as “Ball-color-yellow” and graphically represented as shown in Figure 2-1.

Figure 2-1. O-A-V example. The Authors adapted from Devedžic et al. (2009).

Rules include one or more premises, conditions, and antecedents (*i.e.*, situation), to one or more consequents (*i.e.*, conclusions). Both antecedents and consequents may be used for representing complex rules, *e.g.*, IF the time is after midnight AND I am hungry (antecedents), THEN I should not eat now (consequents).

Semantic networks (also known as conceptual maps) are used for capturing and representing cognition. KRLs contain object graphs, concepts, and situations related to a specific domain, representing the psychological model of the human associative memory. The nodes in the graph are connected by using links and arcs representing relationships. Labels are used for improving the understandability of the represented relationships, setting them a type such as "kind-of," "part-of," and "is-a." Also, some abstracts of the semantic network may be defined as O-A-V triplets and Object-Oriented-Programming (OOP) elements, including classes (concepts), instances (objects), attributes, values, and relationships.

The statement "Billy is Labrador, which is a kind of a dog. Other types of dogs are Setters and Bulldogs. All of them have 4 legs" may be represented as a semantic network of interconnected facts, as shown in Figure 2-2.

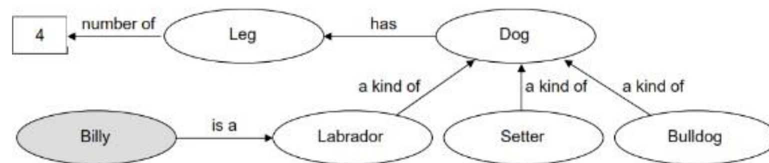

Figure 2-2. Semantic network example. The Authors adapted from Devedžic et al. (2009).

Frames are used for modeling stereotypical knowledge related to some concept/object. Frames may be understood as classes coming from the OOP context. Class frames are used for representing a template for a set of similar objects. Instance frames are used for describing instances of objects and their slots (*i.e.*, attributes) in the set.

2.1.2 Formal specification languages

Formal methods are used for describing the function and architecture of software systems. Such methods include strict notations and deductive principles, facilitating the application of completeness and correctness proofs. FSLs are the notations of formal methods. Such languages include three main components (Pang *et al.*, 2016): *alphabets* comprise a finite set of tokens, including symbols (*e.g.*, punctuation marks and special symbols) and strings (*e.g.*, single characters, words, and meaningful units); *syntax* defines a rule-based logic for composing well-formed expressions within a specific FSL; *semantics* is concerned with the meaning and the interpretation of the expressions within a FSL; *formal grammar* comprises a set of rules defining the syntax of a FSL.

Software analysts use FSLs for writing specifications related to a particular domain, including assertions, formulas, and sentences which may be computed, validated, and used for inference processes (Hahn et al., 2022; Pang *et al.*, 2016). FSLs allow software analysts for enhancing the precision and rigor of the RE process, bridging the gap between high-

level informal requirements specifications and formal and unambiguous specifications which may be analyzed and validated (Pang *et al.*, 2016; Sammi et al., 2010). However, FSLs are based on mathematic and logic-based notations, hardening the learning and understanding for inexperienced software analysts and stakeholders (Hahn *et al.*, 2022; Parkes, 2002).

Some types of FSL are explained as follows:

Logic-based representation languages are used for representing sentences as assertions, limiting reasoning in formal logic to the derivation of truth values and proofs from such assertions. The formal nature is challenging for representing other forms of human reasoning beyond logical deduction, such as interrogations, beliefs, doubts, and desires. Also, designing a comprehensive and valid inference procedure for multiple logics remains a complex task (Bruijn, 2007).

First-Order Logic (FOL), also known as first-order predicate calculus, comprise three fundamental components: *syntax*, which is used for defining the rules governing the constructions of well-formed formulae related to the logical language; *semantics*, which is used for relating meaning to a well-formed formula. Semantics provides a formal framework for interpreting and assigning truth values to the expressions related to the logical language, allowing for understanding the relationships and implications included in the formulae; *proof procedures* are used for deriving deductive consequences by using syntactic operations and semantic methods. Such procedures allow for a systematic exploration and manipulation of well-formed formulae, facilitating the inference of logical consequences (Rabinia & Ghanavati, 2017).

Propositional-logic-based languages are used for representing knowledge as propositions, *i.e.*, true/false expressions (Parkes, 2002). Symbolic variables are assigned to propositions expressed in propositional logic, allowing for symbolic reasoning (Varzi, 2022). More complex expressions may be represented by using logical connectors such as *AND* ($\wedge$), *OR* ($\vee$), *NOT* ($\neg$), *IMPLIES* ($\rightarrow$ *or* $\Rightarrow$), *and EQUIVALENCE* ($\Leftrightarrow$). For example, the proposition "The princess is in the palace" may be assigned to the symbolic variable A. Such a proposition may be further used for representing more complex statements and rules, *e.g.*, IF The princess is in the palace (A) AND The king is in the garden (B) THEN The king cannot see the princess (C). Such expression may be written by using symbols as follows: $A \wedge B \Rightarrow C$. *Propositions* comprise arguments and predicates, asserting certain knowledge about the world, *e.g.*, "The princess is in the palace" may be represented as *in(Princess, palace)*, where the predicate *in* is used for capturing and generalizing the knowledge about the relationship between two variables X and Y, *i.e.*, *in(X, Y)*. Such a predicate may be used for some examples such as *in(King, garden)* and *in(Queen, throne)*. Also, *functions* allow for representing the relationship between elements from different sets, *e.g.*, *father(princess) = king*. *Rules* are used for representing more complex expressions, including such symbols and notations, *e.g.*, the expression "if the princess is in the palace

and the king is in the garden, then the king does not see the princess" may be represented as *in(princess, palace)* ∧ *in(king, garden)* → ¬*see(king, princess)*.

2.1.3 Meta-Models for Software Knowledge Representation

Software systems comprise a wide range of complex components so several KRLs and FSLs are needed for a comprehensive representation of their knowledge. Since most KRLs and FSLs are focused on specific domains, some authors have proposed KRL-FSL pairs for representing specific software domains, including a set of transformation rules, allowing software analysts for obtaining FSs from specific KRLs.

Meta-models are aimed at representing several software domains at a time, mitigating the domain-dependency nature of classical KRLs. Such models comprise KRL-FSL pairs, allowing software analysts for representing software system artifacts, such as code, models, and specifications in a more generalized manner. Some proposals are focused on representing reusable components (Knowledge-Discovery meta-model; Pérez-Castillo et al., 2011), data entities, user interface entities (MoDisco meta-model; Bruneliere et al., 2014), code analysis activities (Abstract Syntax Tree meta-model; Son & Kim, 2017), programming languages (GASTM meta-model Son & Kim, 2017), and object-oriented features, including classes, attributes, and relationships (FAMIX; Tichelaar et al., 2000).

2.1.3 Pre-conceptual Schemas

Pre-conceptual Schemas (PCSs) are intended to capture and represent a software domain. Such representations pose an unambiguous syntax, exhibiting a high degree of resemblance to natural language, thus facilitating the stakeholder comprehension of the software domain. PCSs comprise a collection of graphical and textual structures, enabling the depiction of both dynamic and static features coming from a software domain within a unified representation (Zapata, 2012). Such structures are shown in Figure 2-3 and described as follows:

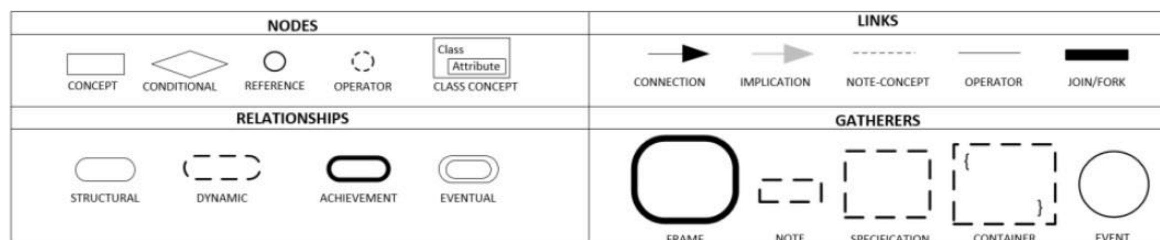

Figure 2-3. Pre-conceptual schemas structures. The Authors adapted from (Noreña C., 2020).

Nodes. Concepts are used for representing roles, entities, and categories related to a software domain. Concepts are represented by nouns and noun phrases, *e.g.*, formal language, software analysts; *conditionals* are used for describing an expression conditioning a dynamic relationship, *e.g.*, if state = validated; *references* are used for linking concepts and relationships; *operators* are used for either comparing two concept or assigning values to a concept. Operators comprise *logical* (AND, OR), *basic* (+, -, *, /), and *relational* (<, <=, >, >=, =); class concepts are used for defining an attribute of a specific concept, *e.g.*, language syntax.

Relationships. Structural relationships are used for expressing a dependency relationship between two concepts including the verbs *has* and *is*, *e.g.*, user is software analyst; *Dynamic relationships* are used for representing operations, actions, and functions in a software domain by using action verbs, *e.g.*, software analyst models software domain; *Eventual relationships* are used for representing events, *e.g.*, file arrives.

Links. *Connections* are used for linking concepts with either dynamic/structural relationships; *Implications* are used for representing cause-and-effect relationships between dynamic relationships and events; *Operators* are used for linking operators with either operators or concepts.

Gatherers. Frames are used for grouping concepts and dynamic relationships; *Notes* are used for assigning values to a concept; *Specifications* are used for characterizing a dynamic relationship; *Constraints* are used for describing constraints related to a concept.

2.1.4 Requirements Engineering

Requirements are formally defined as statements capturing and expressing needs, constraints, and conditions (IEEE, 2018). Some authors have defined two primary requirement types: *functional requirements* pertain to the desired functionality of a specific software system, including constraints, conditions, and related entities, *e.g.*, “the user may upload her profile picture;” *non-functional requirements* are address features related to performance, reliability, interfaces, and design constraints, *e.g.*, “the system time response for queries must take less than two seconds” (Wang & Zeng, 2009).

Requirements Engineering (RE) is defined as the process of identifying the requirements and constraints related to a software system. Such a process is key for subsequent phases of the software development process (Ross & Schoman, 1977). RE encompasses the identification of stakeholders, which are individuals who are directly involved or indirectly affected by the development and implementation process related to a specific software system (Lapouchnian, 2005), and their requirements to define the domain scope of the software system. Dick et al. (2017) describe several closely related activities related to RE as follows:

Domain analysis focuses on characterizing stakeholders, identifying opportunities for improvement, and establishing target objectives within the software domain.

Elicitation is concerned with identifying and characterizing the requirements of the identified stakeholders. Such activity involves several techniques, including interviews, surveys, and observations, facilitating the gathering of relevant information.

Negotiation and agreement aim to select solution alternatives by evaluating the identified requirements and engaging in negotiations with stakeholders to resolve conflicts and reach a consensus.

Specification involves formulating and detailing the requirements based on the selected solution, allowing software analysts for creating models, diagrams, and textual representations for documenting the identified software system behavior.

Documentation involves the creation of requirements documents capturing the decisions made during the entire requirements engineering process. Such documents serve as a reference for stakeholders, software analysts, and other actors during the software development process.

Such activities produce a high amount of knowledge to be understood, represented, and validated (Karolita *et al.*, 2023). Software analysts use KRLs for representing such knowledge, allowing them for understanding, organizing, and sharing it with stakeholders and other actors in the software development process (Ang & Hartley, 2007; Dubois *et al.*, 1986). KRLs are key for RE, allowing software analysts for describing the software system concepts, relationships, constraints, and rules in a formal and unambiguous manner by using several approaches such as OWL (Web Ontology Language; Alkhammash, 2020), UML (Unified Modeling Language; Abdelnabi et al., 2021; Siddique et al., 2014), and FOL (First Order Logic; Rabinia & Ghanavati, 2017).

2.2 Ph.D. Thesis Focus

This Ph.D. Thesis is focused on knowledge representation in the context of RE by integrating three fields: requirements engineering, knowledge representation languages, and formal specification languages. We specifically work on the model-to-model transformation area coming from the KR and FS fields, proposing a new method for transforming KRLs into FSLs and vice versa, allowing software analysts to easily understand and communicate complex specifications regardless of their nature.

Our research is aimed at the improvement of the knowledge representation field, allowing for addressing any KRL and FSL, mitigating the need for high expertise in mathematics and logic to understand and communicate complex specifications. Therefore, a new method for model-to-model transformation is proposed by using two meta-models, representing key

elements from well-known KRLs and FSLs, and a transformation model, describing the rules of transformation from model to model.

2.3 Research Methodology

We define four research phases by following the guidelines coming from the empirically based technology transfer methodology (Wohlin *et al.*, 2012) and the design science methodology for information systems and software engineering (Wieringa, 2014), including exploration, problem formulation, solution, and validation.

2.3.1 Exploration

During the exploration phase, a systematic literature review is performed for identifying, selecting, and characterizing the existing evidence related to the research area (Wohlin *et al.*, 2012). This review adheres to the software engineering guidelines put forth by Kitchenham *et al.* (2009) and aligns with the principles of experimentation in software engineering as outlined by Wohlin *et al.* (2012). Moreover, several tasks are carried out and refined for establishing the review protocol, gathering background, compiling a list of primary studies, and conducting study analysis. Such activities encompass planning the literature review and executing a systematic examination of the literature.

2.3.2 Problem Formulation

Specifying the problem statement, *formulating research questions*, and *developing hypotheses* are activities performed during the problem formulation phase. Such activities are key for defining the research objectives, which are based on the three problems identified during the exploration phase.

2.3.3 Solution

A solution is proposed for improving the model-to-model transformation approach for the KR and FS fields by creating a method for obtaining FSs and KRLs. Characterizing KRLs and FSLs, defining a meta-model for KRLs, defining a meta-model for FSLs, defining a transformation model, and proposing a method for model-to-model transformation are activities performed for producing a KRL-related key elements report, FSL-related key elements report, a rule-based approach for model-to-model transformation, and a method for transforming KRLs into FSs and vice versa.

2.3.4 Validation

Three experiments are performed for evaluating the flexibility and the capabilities of the proposed method. *Planning experiment, executing experiment, and analyzing experiment* are activities developed for producing an *experiment design, experiment data,* and an *experiment report.*

3. Research Problem

3.1 Motivation

RE is a stakeholder-centered approach so natural language is a common means for documenting software requirements. Since natural language is inherently ambiguous, the RE process is prone to misunderstandings and misinterpretations, hardening the communication and validation of the software domain between the stakeholders and software analysts (Pang *et al.*, 2016). Software analysts improve the RE process by using several models comprising graphical and textual elements for representing the knowledge related to a software domain, including KRLs and FSLs (Ahmad *et al.*, 2020; Ang & Hartley, 2007; Dubois *et al.*, 1986).

KRLs are used for easing the knowledge representation task by including graphical and textual elements so inexperienced practitioners should understand and validate the software domain of interest (Karolita *et al.*, 2023). Such languages are aimed at representing the semantics and relationships of key concepts coming from the software domain, easing the representation of the dynamics related to a software system (Sonbol *et al.*, 2020). FSLs are used for representing and analyzing the structure and behavior of a software system by using mathematics and logic-based formalisms. Such languages allow software analysts for formalizing the knowledge representation process, enabling them for representing complex software systems, and providing an unambiguous representation of the gathered knowledge (Pang *et al.*, 2016).

KRLs and FSLs are often used together with transformation rules, allowing software analysts for representing and communicating domain knowledge, and validating the correctness and completion of the software system being developed (Ang & Hartley, 2007). KRL-FSL pairs are often focused on a specific domain, so software analysts should learn several KRL-FSL pairs and their transformation rules for representing various software domains (Alkhammash, 2020). Some authors have proposed meta-models for knowledge representation, minimizing the number of KRL-FSL pairs to be learned for representing some software domains (Finne, 2011).

Even though software analysts address a software domain at a time, they are interested in the same kind of abstractions while they are characterizing a software domain, including classes, relationships, and constraints. Such knowledge may be framed into a meta-modeling language, allowing for a generalized representation of the knowledge regardless of the nature of the software domain. In the context of meta-models, software analysts are

aimed at representing recurrent components coming from key KRLs and FSLs (Son & Kim, 2017). Thus, such components may be framed into a KRL meta-model and a FSL meta-model, characterizing any KRL and FSL. Also, since both KRLs and FSLs comprise some overlapping and complementing elements, a set of heuristic rules enabling model-to-model transformation may be built based on the meta-model representations, allowing software analysts for transforming any KRL to a FSL and *vice versa*.

This Ph.D. Thesis is motivated by the growing trends of applicability of the model-to-model transformation approaches and the potential of meta-models for representing any KRL and FSL, allowing software analysts for representing any software domain regardless of its nature in an unambiguous manner, improving the RE process.

3.2 State of the Art

3.2.1 Planning Literature Review

The formulation of *research questions* (RQs) is guided by primary studies, enabling for defining a comprehensive *review protocol* and characterizing the *study criteria* (Wohlin *et al.*, 2012). Software analysts use KRL-FSL pairs for addressing more complex software systems focused on specific software domains. Such KRL-FSL pairs comprise different set of heuristic rules, allowing practitioners for transforming one model into another model, *e.g.*, transforming a KRL into a FSL and *vice versa*. Consequently, we suggest research questions RQ1, RQ2, and RQ3. KRL-FSL pairs are composed of single KRLs and FSLs, so they may have different capabilities and limitations in terms of the domain elements they can represent (Maio, 2021; Torres *et al.*, 2019). Therefore, we suggest research questions RQ4, RQ5, and RQ6.

Conforming to the study criteria outlined in Table 3-1, we systematically identify and examine KRL to FSL and FSL to KRL proposals which are aimed at supporting the RE process.

3.2.2 Executing Systematic Literature Review

The *study selection process* encompasses four steps (Kitchenham *et al.*, 2009): *initial search, remove duplicates, exclude studies, and include external studies*. After performing the study selection process, we obtained 21 primary studies as follows: initial search (391 studies); remove duplicates (376 studies); exclude studies based on title, abstract, keywords, introduction, and conclusions (17 studies); include external studies (21 studies).

Table 3-1: Study criteria. The Authors.

<table>
<tr><td rowspan="10">Inclusion criteria</td><td rowspan="2">Search criteria</td><td colspan="2">(i) KRL to FSL model transformation proposals supporting the RE process</td></tr>
<tr><td colspan="2">(i) FSL to FSL model transformation proposals supporting the RE process</td></tr>
<tr><td>Search sources</td><td colspan="2">IEEE Explore, Science Direct, Springer Links, Scopus, Google Scholar</td></tr>
<tr><td rowspan="5">Search terms</td><td>Main keyword</td><td>Derived keywords</td></tr>
<tr><td>Model transformation</td><td>“model to model transformation” OR “model transformation framework” OR “model transformation language”</td></tr>
<tr><td>Knowledge representation language</td><td>"knowledge modeling language" OR "knowledge representation notation"</td></tr>
<tr><td>Formal specification language</td><td>“formal language” OR “formal specification” OR “formal modeling language” OR “logical formalism”</td></tr>
<tr><td>Requirements engineering</td><td>"requirements elicitation" OR "requirements analysis" OR "software requirements"</td></tr>
<tr><td>Literature</td><td colspan="2">Paper, chapter, book, thesis, and technical document</td></tr>
<tr><td colspan="2" rowspan="3">Exclusion criteria</td><td colspan="2">(i) The proposal does not include a KRL-FSL pair</td></tr>
<tr><td colspan="2">(ii) The proposal does not have a well-defined and formalized syntax and semantics for representing knowledge</td></tr>
<tr><td colspan="2">(ii) The proposal is not focused on RE</td></tr>
<tr><td colspan="2">Research questions</td><td colspan="2">-RQ1. What model-to-model transformation proposals are used for RE?
-RQ2. What domain scopes are analyzed by using model-to-model transformation proposals for RE?
-RQ3. What transformation rules are applied by using the model to model transformation proposals for RE?
-RQ4. What KRLs are represented in the model-to-model transformation proposals for RE?
-RQ5. What FSLs are represented in the model-to-model transformation proposals for RE?
-RQ6. What domain elements are represented by using KRLs and FSLs proposals focused on RE?</td></tr>
<tr><td colspan="2">Hypothesis</td><td colspan="2">A model-to-model transformation method for RE including a meta-model for representing KRLs, a meta-model for representing FSLs, and a meta-model for representing transformation rules can be used for improving the domain knowledge representation in the context of RE.</td></tr>
</table>

Model-to-Model transformation proposals for RE

We analyze 15 proposals in the field of model-to-model transformation in the context of RE. Such proposals comprise 37 different models including, KRLs, FSLs, and domain-specific languages. We characterize such approaches and answer RQ1, RQ2, and RQ3 as follows:

Awan et al. (2022) introduce a comprehensive framework for characterizing the concepts, relationships, functions, and instances related to specific software systems by using natural language processing and parts-of-speech techniques. Such an approach allows practitioners for representing natural-language-based specifications by using XText, an intermediate representation based on domain-specific languages (Awan et al., 2022), which may be transformed into formal specifications based on the Z-notation (Spivey, 1989).

Djaoui et al. (2018) propose IOD2Maude, a model-to-model transformation focused on the Interaction Overview Diagram (IOD) and the Maude logic specification language. IOD diagrams are a specialization of the activity diagrams described in the UML 2.0 specification (Jena et al., 2015). Also, Maude is used for representing dynamic transitions and alterations from complex software systems in the context of the logical paradigm (Clavel et al., 2002). The IOD2Maude framework is aimed at establishing a comprehensive meta-model of IODs, allowing modelers for encapsulating key IOD elements and mapping the relationships among them. Such a framework comprises a set of well-defined rules, including graph-based grammar and pivotal elements for transforming the IOD meta-model elements into Maude specifications.

Amjad et al. (2018) propose UMLPACE, a tool for transforming Event-driven Process Chain (EPC) models into timed automata formal specifications. UMLPACE extends EPC models by incorporating activity diagram components, extending the expressiveness of such models, and allowing modelers for addressing more complex software systems (Kleppe & Warmer, 2000). Subsequently, the authors provide a set of procedural sequences, expressing how the augmented model may be transformed into timed automata formal specifications, integrating heuristic rules for representing the equivalences between EPC, activity diagrams, and timed automata formal specifications.

Some authors propose model-to-model transformations based on KRL-FSL pairs by using Event-B as a target model. Event-B is a formal method for describing complex software systems, including their events and intricate processes (Abrial, 2010). Sun et al. (2016) introduce KM3, a model-to-model transformation method focused on KRL-FSL pairs. They use sequence diagrams and use case diagrams as source models so they can derive the represented knowledge into the target model, Event-B specifications (Abrial, 2010). Such a transformation is framed by the Rodin platform which serves as a validation platform for the resultant specification (Abrial, 2010).

BPM2.0 provides a high number of model constructs, allowing modelers for representing complex workflows while characterizing the dynamics of software systems, including events, data types, tasks, and collaborations (Correia & e Abreu, 2012). Even though such a model is more expressive than classical approaches such as the activity diagram (Ramadan et al., 2020), such a model lack well-defined formal semantics, thereby impeding the validation of the generated model instances. Ben Younes et al. (2019) propose BPMN2EVENTB for addressing such a limitation, allowing modelers for transforming BPMN2 models into Event-B formal specifications by using the Rodin platform, including model constructs for formally representing events, concepts, datatypes, preconditions, relationships, parameters, and constants (Abrial, 2010). Hlaoui et al. (2017) propose SD2EventB, a KRL to FSL transformation approach focused on transforming sequence diagrams (OMG, 2011) into EventB formal specifications in the context of cloud services development. Furthermore, Boussetoua et al. (2015) propose a model-to-model transformation method for translating BPMN2 representations into Pi-Calculus, allowing modelers for representing agents and interactions related to complex software systems (Sangiorgi & Walker, 2001).

Tariq et al. (2017) introduce a novel methodology for automating the analysis and validation of activity diagrams. Such an approach is aimed at formally characterizing UML behavioral models lacking well-defined semantics. The authors propose a model-to-model transformation including Colored Petri Netas (CPNs) as the target model. Such a model is key as it allows modelers for analyzing and validating activity diagrams (Zimmermann, 2008). Also, CNPs are used for validating class diagrams by using a KRL-FSL transformation process, including domain concepts such as concepts, attributes, actions, and relationships (Sharaff & Rath, 2020).

Couto et al. (2014) introduce a model-to-model transformation approach for identifying requirements patterns in use cases. Such an approach comprises two well-defined steps: (i) *use case formalization* where use cases are formalized by using RUS, an XML-based representation for use cases (Couto et al., 2014) and (ii) *transformation* where the RUS specification is transformed into an ontology focused on RE, including domain concepts such as concepts, instances, and relationships.

Borgida et al. (2014) address model-to-model transformations from the context of goal-oriented models. They use i*CORE as a source model for characterizing complex software systems from a goal view (Giorgini et al., 2002). Such a characterization is further analyzed and verified by using FOL, allowing practitioners for mapping domain concepts such as goals, processes, and relationships.

Domain-Specific Languages (DSLs) are tools for capturing the features and relationships of software systems related to specific domains (Rodríguez-Gil et al., 2019). Jiang et al. (2016) propose a method for automating the analysis and evaluation of DSLs by using a model-to-model transformation approach. Such an approach includes XMML, a meta-

model for representing DSLs, allowing practitioners for representing key components of DSLs, and a set of heuristic rules for transforming XMML into FOL.

Class diagrams are effective means for representing the concepts, features, and relationships related to a software domain (E. A. Abdelnabi et al., 2020). However, such diagrams lack representation of constraints and conditions. Some authors include OCL constraints for augmenting the expressiveness of class diagrams, allowing modelers for representing more complex software systems. Gogolla et al. (2017) propose a model validation, verification, and exploration approach for class diagrams by using a model-to-model transformation process, including a class diagram extension based on OCL conditions and transformation heuristics for the Kodkod formal specification (Torlak and Jackson, 2007). Pérez and Porres (2019) propose a similar approach extending class diagrams with OCL conditions and providing heuristic rules for transforming such diagrams into Formula specifications, allowing modelers for evaluating the completeness and correctness of software domain models.

Chu and Dang (2020) propose a model-to-model transformation method focused on KRL-KRL pairs, especially, class diagrams and use cases. Such a method provide equivalences between the main concepts of the class diagram, including classes, attributes, association relationships, and methods, and the main concepts of the use cases, including actors and use cases. Such an approach is aimed at extending the expressiveness of complex software domain representations.

Some authors propose model-to-model transformation approaches for evaluating the completeness and correctness of UML diagrams such as use case diagrams (Saratha et al., 2017; Sengupta & Bhattacharya, 2006) and activity diagrams (Jamal & Zafar, 2016) by using the formal Z-notation, allowing modelers for characterizing domain concepts such as actors, actions, processes, and relationships.

Xu (2011) proposes a KRL-FSL transformation method for formalizing the activity diagram by using Process Algebra (Bernardo et al., 2002). Such an approach is aimed at mapping the key concepts from activity diagrams such as nodes and actions by using agents and communication relationships, allowing modelers for characterizing the dynamic behavior of complex software systems.

Meziani et al. (2018) introduce a FSL-KRL transformation method for translating colored Petri nets into state machine diagrams, including processes, actions, initial values, constraints, and processes. Such an approach allows modelers for mitigating the inherent complexity and formalism coming from colored Petri nets by using a simpler representation based on state machine diagrams.

We summarize the primary studies (21) in Table 3-2 for answering the remaining research questions of the literature review.

Table 3-2: Primary studies summary. The Authors.

Proposal	Model pairs	Transformation direction	Source model	Represented source model elements	Target model	Represented target model elements	Represented domain elements
NLP2FM (Awan et al., 2022)	KRL-FSL	One-way	NL (Xtext domain-specific language)	System state space schema, system state space name, system state space variable, system state space predicate, system state initial state, initial state name, initial state variable, schema name, schema input variable, schema output variable, schema predicate, predicate input, predicate output	Z-Notation (Spivey, 1991)	Sets, predicate, schema, expressions, operator, state, invariant relationship, operation, input, output, relationship, change of state, procedure, variable, inference rule	Concepts, instances, relationships, functions
BPMN2EVENTB (Bessifi et al., 2019)	KRL-FSL	One-way	BPMN2 (Correia and Abreu, 2012)	Event, gateway, datatype, activity, activity marker, task, flow, basin, corridor, collaboration	EVENTB (Abrial, 2007)	Event, btype, action, invariant, collaboration, constraint, parameter, multiplicity, predicat, constants, sets	Events, concepts, datatypes, preconditions, relationships, parameters, multiplicity, constants
IOD2Maude (Djaoui et al., 2018)	KRL-FSL	One-way	Interaction Overview Diagram (Santosh et al., 2015)	Initial Node, final node, fork node, join node, decision node, merge node, transition, interaction, interaction use	Maude (Clave et al., 2002)	Functional modules, object-oriented modules, parametrized modules, theories, views, module renaming, tuples, conditions, reflection, datatypes	Class, attributes, relationships, constraints, conditions, data types
UMLPACE (Amjad et al., 2018)	KRL-FSL	One-way	Activity diagram (Kleppe and Warmer, 2000)	Initial node, control flow, action, activity final node	Timed automata (Bengtsson and Yi, 2003)	Initial location, committed location, location, Edge	Event, relationship, action
KM3 (Weixuan et al., 2016)	KRL-FSL	One-way	Use Case diagram (Siau and Cao, 2001)	System, actor, use case, association, include, extend	EVENTB (Abrial, 2007)	Context, constant, set, machine, variable, invariant, event, guard, action	Concept, actor, action, relationship
SD2EventB (Daly Hlaoui et al., 2017)	KRL-FSL	One-way	Sequence diagram (OMG, 2013)	Sequence model, interaction fragment, resource, lifeline, interaction, message, types, interaction operator	EVENTB (Abrial, 2007)		Process, action, parameter, data type, operator
(Tariq et al., 2017)	KRL-FSL	One-way	Activity diagram (Lleppe and Warmer, 2000)	Initial node, control flow, action, activity final node	Colored petri net (Zimmermann, 2008)	Types, coloured set, place, transition, arch, color function, data value, guard function, initial marking	Process, action, relationship, initial value, constraint
(Couto et al., 2014)	FSL-KRL	One-way	RUS (Cout et al., 2014)	Individual, Entity, Property	OWL (Antoniou and Grigoris, 2004)	Individual, Entity, Property, Fact	Concept, instance, relationship, attribute
(Borgida et al., 2014)	KRL-FSL	One-way	i*CORE (Borgida et al., 2014; Giorgini et al., 2002)	Goal, entity, task, relationship, evidence values	First-order-logic (Moore, 1988)	Indivdual, predicate, relationship, variable, constant	Concept, goal, relationship, process
XMML (Jiang et al., 2016)	DSL-FSL	One-way	XMML (Jiang et al., 2016)	Model, entity, relationship	First-order-logic (Moore, 1988)	Containment, attachment, source role, assignment association, target role assignment assocaition, refinement, symbol, constraint formula	Concept, goal, relationship, process

Table 3-2: Primary studies summary. The Authors. (Continuation)

Proposal	Model pairs	Transformation direction	Source model	Represented source model elements	Target model	Represented target model elements	Represented domain elements
USE (Gogolla et al., 2018)	KRL-FSL	One-way	Class diagram and OCL (Gogolla et al., 2018)	Class, attribute, method, constraint, Object, relationship, class invariant	Kodkod (Torlak and Jackson, 2007)	Atom, relation declaration, relational formula	Concept, attribute, initial value, constraint, relationship, instance
Formula (Pérez and Porres, 2019)	KRL-FSL	One-way	Class diagram and OCL (Pérez and Porres, 2019)	Class, attribute, method, constraint, Object, relationship, class invariant	Formula (Pérez and Porres, 2019)	Domain, model, partial model, class, constraint, instance, class, data type, property, association relationship, generalization relationship	Concept, attribute, initial value, constraint, relationship, instance
(Boussetoua et al., 2015)	KRL-FSL	One-way	BPMN2 (Correia and Abreu, 2012)	Start event, intermediate event, end event, task, and gateway, xor gateway, subprocess, sequence flow	Pi-Calculus (Sangiorgi and Walker, 2001)	Agent, interaction, process, name	Concept, process, relationship
(Ries et al., 2021)	KRL-FSL	One-way	DRCModel	Dataset, equivalence class, data, property, invariant property, typed variable	Alloy (Jackson, 2012)	Signatures, abstract signatures, fields, invariant properties, signature facts, predcates	Concepts, attribute, relationship
(Chu and Dang, 2020)	KRL-KRL	One-way	Class diagram (OMG, 2013)	Class, attribute, method, association relationship	Use case (OMG, 2013)	Actor, use case	Concept, attribute, relationship
(Jamal et al., 2016)	KRL-FSL	One-way	Activity diagram (Lleppe and Warmer, 2000)	Initial node, control flow, action, activity final node	Z-Notation (Spivey, 1991)	Sets, predicate, schema, expressios, operator, state, invariant relationship, operation, input, output, relationship, change of state, procedure, variable, inference rule	Process, action, relationship, initial value, constraint
(Sengupta and Bhattacharya, 2006)	KRL-FSL	One-way	Use Case diagram (Siau and Cao, 2001)	System, actor, use case, association, include, extend	Z-Notation (Spivey, 1991)	Sets, predicate, schema, expressios, operator, state, invariant relationship, operation, input, output, relationship, change of state, procedure, variable, inference rule	Concept, actor, action, relationship
(Saratha et al., 2017)	KRL-FSL	One-way	Use Case diagram (Siau and Cao, 2001)	System, actor, use case, association, include, extend	Z-Notation (Spivey, 1991)	schema, expressios, operator, operation, input, output, relationship, procedure,	Concept, actor, action, relationship
(Xu, 2011)	KRL-FSL	One-way	Activity diagram (Lleppe and Warmer, 2000)	Initial node, control flow, action, activity final node	Process Algebra (Bernardo et al., 2002)	Agent, system, communication	Activity, concept, relationship
(Meziani, 2018)	FSL-KRL	One-way	Colored petri net (Zimmermann, 2008)	Types, coloured set, place, transition, arch, color function, data value, guard function, initial marking	State Machine (OMG, 2013)	State, transition, event, initial state, final state	Process, action, relationship, initial value, constraint
(Sharaff and Kumar, 2020)	KRL-FSL	One-way	Class diagram (OMG, 2013)	Class, attribute, method, association relationship	Colored petri net (Zimmermann, 2008)	Types, coloured set, place, transition, arch, color function, data value, guard function, initial marking	Process, action, relationship, concept

The answer to RQ4 is the following:

Some proposals (19 out of 21) comprise KRLs as their source models. Such proposals include different types of KRL elements which are specific to the transformation process: Awan et al. (2022) include KRL elements such as system state space schema, system state space name, system state space variable, system state space predicate, system state initial state, initial state name, initial state variable, schema name, schema input variable, schema output variable, schema predicate, predicate input, predicate output; Ben Younes et al. (2019) and Boussetoua et al. (2015) analyze KRL elements such as start event, intermediate event, end event, task, and gateway, xor gateway, subprocess, sequence flow; Amjad et al. (2018), Djaoui et al. (2018), and Tariq et al. (2017) explore KRL elements related to the activity diagram and the interaction overview diagram such as initial Node, final node, fork node, join node, decision node, merge node, transition, interaction, interaction use; Sun et al. (2016) and Jiang *et al.*, (2016) represent KRL elements such as system, actor, use case, association, include, and extends relationships in the context of use case diagrams; Hlaoui et al. (2017) represent KRL elements in the context of sequence diagrams, including sequence model, interaction fragment, ressource, lifeline, interaction, message, types, interaction operator; Cuoto *et al.* (2014) use KRL elements such as entities, properties, and facts for representing OWL characterizations; Borgida *et al.* (2014) address goal-oriented concepts by using KRL elements such as goals, entities, tasks, relationships, and evidence values; Gogolla *et al.* (2018), Pérez and Porres (2019), and Chu and Dang (2020) characterize class diagramas by using KRL elements such as classes, attributes, methods, and association relationships.

The answer to RQ5 is the following:

Some proposals (20 out of 21) comprise FSLs as their target and source model. Such proposals include different types of FSL elements which are specific to the transformation process: Mehboob *et al.* (2022), Jamal and Zafar (2016), Saratha et al. (2017), and Sengupta and Bhattacharya (2006) include FLS elements such as sets, predicates, schemas, expressions, operators, states, invariant relationships, operations, relationships, changes of state, procedures, variables, and inference rules related to the context of the formal Z-notation; Ben Younes et al. (2019), Hlaoui et al. (2017), and Sun et al. (2016) use Event-B as target model in the KRL-FSL transformation process, including FSL elements such as context, constant, set, machine, variable, invariant, event, guard, and action; Djaoui *et al.* (2018) include some FSL elements in the context of the Maude formal specifications, including functional modules, object-oriented modules, parametrized modules, theories, views, module renaming, tuples, conditions, reflection, and datatypes; Amjad *et al.* (2018) explore timed automata as FSL in the UMLPACE method, including FSL elements such as initial location, committed location, location, and edge; Tariq *et al.* (2017) include FSL elements related to CPNs, including types, coloured set, place, transition, arch, color function, data value, guard function, and initial marking; Couto *et al.* (2014) represent FSL elements based on the RUS model, including individuals, entities, and properties; Borgida

et al. (2014) address goal-based scenarios including FSL elements such as individual, predicate, relationship, variable, and constants coming from the FOL language; Jiang *et al.* (2016) include more FOL elements, including source role, assignment association, target role, and constraint formula; Gogolla *et al.* (2018) include the Kodkod formal specification as FSL in the ISE method, including FSL elements such as atom, relation declaration, and relational formula; Pérez and Porres (2019) address the class diagram formalization into Fomrula, including FSL elements such as domain, model, partial model, class, constraint, instance, class, data type, property, association relationship, and generalization relationship; Boussetoua *et al.* (2015) use agents, interactions, processes, and names as FSL elements in the context of Pi-Calculus; Rise *et al.* (2021) use Alloy elements, including signatures, abstract signatures, fields, invariant properties, signature facts, and predicates as FSL elements in the KRL-FSL transformation process.

The answer to RQ6 is the following:

KRL and FSL elements are used for representing domain elements at different levels of abstraction, including concepts (Awan *et al.*, 2022; Bessifi *et al.*, 2019; Sun *et al.*, 2016; Couto *et al.*, 2014; Borgida *et al.*, 2014; Jiang *et al.*, 2016; Gogolla *et al.*, 2018; Pérez and Porras, 2019; Boussetou *et al.*, 2015; Ries *et al.*, 2021; Chu and Dang, 2020), instances (Awan *et al.*, 2022; Couto *et al.*, 2014), relationships (Awan *et al.*, 2022; Bessifi *et al.*, 2019; Sun *et al.*, 2016; Couto *et al.*, 2014; Borgida *et al.*, 2014; Jiang *et al.*, 2016; Gogolla *et al.*, 2018; Pérez and Porras, 2019; Boussetou *et al.*, 2015; Ries *et al.*, 2021; Chu and Dang, 2020; Tariq *et al.*, 2017), functions (Awan et al., 2022), events (Amjad et al., 2018; Ben Younes et al., 2019), data types (Ben Younes et al., 2019; Djaoui et al., 2018; Hlaoui et al., 2017), preconditions (Ben Younes et al., 2019), parameters (Ben Younes et al., 2019; Hlaoui et al., 2017), multiplicity (Ben Younes et al., 2019), constants (Ben Younes et al., 2019), attributes (Chu & Dang, 2020; Djaoui et al., 2018; Gogolla et al., 2017; Pérez & Porres, 2019; Ries et al., 2021), actors (Sun et al., 2016), processes (Hlaoui et al., 2017; Jiang et al., 2016; Tariq et al., 2017), and goals (Borgida et al., 2014; Jiang et al., 2016).

3.3 Problem Statement

The field of KRLs and FSLs is an active area of research aiming at improving the RE process. KRLs and FSLs enable software analysts for rigorously and unambiguously capturing, modeling, and reasoning about the domain knowledge and the software system requirements. However, some challenges remain in adopting and using KRLs and FSLs in the context of RE. Such challenges are focused on two problems: limitations on the domain scope and model transformation. The first problem refers to the ability of software analysts for using KRL-FSL pairs for representing different types of software domains. The second problem refers to the ability of software analysts for using KRL-FSL pairs which provide consistent and automated mechanisms for transforming models between different levels of representation, such as from KRL to FSL and from FSL to KRL.

Software analysts use KRL-FSL pairs for describing concepts and relationships coming from a software domain, and the features and constraints related to software systems. While such languages are effective tools for representing different aspects of the software domain, by combining them, software analysts may use such KRL-FSL pairs for comprehensively capturing both the semantic and syntactic aspects related to a software domain. Also, KRL-FSL pairs allow software analysts for improving the communication of the software domain with the stakeholders and the verification and validation of the software system. However, KRL-FSL pairs are not suitable for addressing software domains in a generalized way. Therefore, since each software domain comprises different features, choosing the adequate KRL-FSL pair is key for the RE process. The literature review shows some of the existing KRL-FSL pair proposals (4 out of 21) are tailored to specific software domains, including domain-specific languages. While such proposals are used for improving the RE process in such specific software domains, the applicability and reusability of such KRL-FSL pairs on other software domains are still limited.

Model transformation allows software analysts for obtaining one model (target model) from another model (source model) in different levels of representation, such as KRL to FSL and FSL to KRL. Such a process comprises a set of heuristic rules, characterizing the equivalences between the related models, so software analysts may transform one model into another. However, most of the analyzed proposals focused on model transformation provide one-way transformation rules, including from KRL to FSL (17 out of 21), from FSL to KRL (2 out of 21), and from KRL to KRL (1 out of 21). Such proposals are strongly limited by the lack of bi-directionality of the transformation rules between the source and target model and the dependency on specific features related to the KRL and the FSL. Hence, software analysts should be proficient in different KRLs and FSLs so they can address diverse software domains.

3.4 Objectives

3.4.1 General Objective

Proposing a method for obtaining FL-based specifications based on KRL-based descriptions, independent of the KRL-FL pair to be considered.

3.4.2 Specific Objectives

- Identifying the general features of the FL, independent of the orientation the FL have.
- Selecting a KRL as a starting point of the mapping process to FL-based specifications. The KRL should exhibit the general features identified in the previous objective.

- Defining the method for obtaining FL-based specifications from the selected KRL. The method should include the rules for transforming each element of the KRL into all of the features identified on the FL.
- Validating the proposed method by representing at least three case studies from the state-of-the-art review, related to several KRL-FL pairs.

3.5 Justification

Wieringa (2014) defines three key components related to the *design science methodology for information systems and software engineering*: the *work product* which is the result of an improvement to a problem; the *social context* which describes the stakeholders who are affected by the work product; the *knowledge context* which includes well stated scientific and engineering theories, available practical knowledge and products, and produced knowledge as a result of the experimentation process from researches. In this Ph.D. Thesis we introduce a novel method for obtaining FSLs from KRLs and *vice versa* as a work product. Such a work product is aimed at supporting software analysts (social context) as a new model-to-model transformation approach for RE (knowledge context).

In this Ph.D. Thesis we integrate RE, KRLs, and FSLs. Our method allows software analysts for overcoming the limitations on the domain scope by using a meta-model for KRLs and a meta-model for FSLs for representing any KRLs and FSLs in the context of RE. In addition, our method allow software analysts for performing model-to-model transformations bidirectionally, mitigating the impact of the one-way transformation rules coming from most of the state-of-the-art proposals by using a meta-model for representing transformation rules regardless of the KRL-FSL pair. Therefore, our method allows software analysts for representing domain knowledge in a more general way, improving the RE process.

4. A method for transforming KRL-FSL pairs

KRL-FSL pairs are effective means for representing features and constraints inherent in software domains (Ang & Hartley, 2007). Software analysts use different KRL-FSL pairs for effectively capturing and modeling the intricacies of different software systems (Alkhammash, 2020). One of the key advantages of employing KRL-FSL pairs is the provision of one-way transformation rules, allowing software analysts for transforming representations from one language to another. Such capability allows software analysts for exploring and analyzing several software systems by leveraging the expressiveness of different languages (Alkhammash, 2020).

We propose in this Ph.D. Thesis a method for transforming KRL-FSL pairs in the context of RE. Our method includes three main components: a meta-model for representing KRLs, a meta-model for representing FSLs, and a meta-model for representing model-to-model transformations. Such components are used for defining what software domain concepts a software analyst should represent in any KRL/FSL and how they should do it while using our method, allowing them for representing any software domain by using any KRL and FSL and transforming them into any other KRL/FSL.

We propose a new method for transforming KRL-FSL pairs according to the research methodology in six steps: (i) we characterize state-of-the-art KRLs; (ii) we characterize state-of-the-art FSLs; (iii) we define a meta-model for representing KRLs; (iv) we define a meta-model for representing FSLs; (v) we define a meta-model for representing model-to-model transformations; (vi) we present a method for transforming KRL-FSL pairs.

4.1 Characterizing KRLs in the context of RE

While KRLs are used for representing different software domains, such languages are often focused on the same domain elements (Caetano et al., 2017). Some authors define KRL elements for representing well-known software domain elements:

Domain concepts are explored by using *system spaces* and *schemas* (Awan et al., 2022), *events* (Ben Younes et al., 2019; Boussetoua et al., 2015), *nodes* (Amjad et al., 2018; Djaoui et al., 2018; Tariq et al., 2017), *entities* (Borgida et al., 2014; Couto et al., 2014), and *actors* (Sun et al., 2016).

Attributes are studied by using *variables* (Awan et al., 2022), *interactions* (Djaoui et al., 2018), *properties* (Couto et al., 2014; Ries et al., 2021), and *attribute values* (Chu & Dang, 2020; Gogolla et al., 2017).

Processes are analyzed by using *activities* and *tasks* (Amjad et al., 2018; Ben Younes et al., 2019; Borgida et al., 2014; Boussetoua et al., 2015; Tariq et al., 2017), *methods* (Chu & Dang, 2020; Gogolla et al., 2017), and subprocesses (Boussetoua et al., 2015).

Parameters are represented by using *variables* and *input variables* (Awan et al., 2022; Ries et al., 2021).

Relationships are described by using *collaborations* (Ben Younes et al., 2019), *transitions* and *interactions* (Djaoui et al., 2018; Hlaoui et al., 2017), *associations* (Chu & Dang, 2020; Sun et al., 2016), and class and event relationships (Borgida et al., 2014; Chu & Dang, 2020; Gogolla et al., 2017; Jiang et al., 2016).

4.2 Characterizing FSLs in the context of RE

We analyze eleven different FSL proposals in the literature review on model-to-model transformation approaches focused on KRL-FSL pairs (see Section 3.2). Most of such proposals (10 out of 19) are focused on object-oriented programming. Therefore, we characterize such proposals based on object-oriented programming concepts:

Classes are explored by using *schemas* and *initial states* (Awan et al., 2022), *object-oriented modules* (Djaoui et al., 2018), *initial locations* (Amjad et al., 2018), *sets* (Sun et al., 2016; Tariq et al., 2017), *agents* (Boussetoua et al., 2015), *signatures* (Ries et al., 2021), *individuals* (Borgida *et al*., 2014), and *atoms* (Gogolla *et al*., 2018).

Attributes are described by using *properties* (Couto et al., 2014; Pérez & Porres, 2019), and *fields* (Ries *et al*., 2021).

Data types are represented by using *data types* (Pérez and Porres, 2019).

Functions are studied by using *operations* (Awan et al., 2022), *functional modules* and *parametrized modules* (Djaoui *et al*., 2018), *actions* (Sun et al., 2016), *color functions* (Tariq *et al*., 2017), and processes (Boussetoua et al., 2015; Tariq et al., 2017).

Invariants are analyzed by using *invariant relationships* and properties (Awan *et al*., 2022; Ries *et al*., 2021) and *invariants* (Awan et al., 2022; Ben Younes et al., 2019; Ries et al., 2021; Sun et al., 2016).

Relationships are addressed by using *generic relationships* (Awan et al., 2022; Borgida et al., 2014; Pérez & Porres, 2019), *collaborations* (Ben Younes et al., 2019), *arches* (Tariq *et al*., 2017), and association and *generalization relationships* (Pérez and Porres, 2019).

Constraints are characterized by using *conditions* (Djaoui et al., 2018), *constraints* (Ben Younes et al., 2019; Gogolla et al., 2017; Pérez & Porres, 2019), and *constraint formulas* (Jiang *et al*., 2016).

Processes are analyzed by using *procedures* (Awan *et al*., 2022) and *interaction processes* (Boussetou *et al*., 2015).

4.3 Defining a meta-model for KRLs in the context of RE

We define a meta-model for representing KRLs in the context of RE by using common components coming from the KRL characterization (see Section 4.1). Such components and their relationships represent the common knowledge related to the KRL domain. Thus, we structure such knowledge by using PCSs, representing the KRL domain in a general way as shown in Figure 4-1 and described as follows:

Nodes are used for representing key entities related to a software domain, including concepts, objects, properties, features, and abstract concepts. Nodes are characterized by three elements: *name* (*e.g.*, "modeler"), *feature, and process*.

Features are used for characterizing specific nodes. Features are described by three components: *name* is used for textually representing the feature (*e.g.*, "years of experience"); *visibility* is used for representing the scope of the feature in the context of object-oriented programming (*i.e.*, public, private, and protected); *type* is used for representing the data type related to the feature (*e.g.*, integer).

Processes are used for characterizing the dynamic behavior related to a specific node. Processes are described by five components: *visibility* is used for representing the scope of the process in the context of object-oriented programming (*i.e.*, public, private, and protected); *type* is used for representing the data type related to the resulting element of the process (*e.g.*, string); *name* is used for representing the process (*e.g.*, "Transforms"); *parameter* is used for representing the components and inputs influencing the behavior and outcome of the process. Parameters are represented by type (*i.e.*, data type) and name; *sequence* is used for describing the order in which the process is to be performed (*e.g.*, 1 for representing first, 2 for representing second, and so on). An example of a process is "modeler **transforms** model." Such a process may be described as follows: the name is "transforms," the visibility is public so the process may be reused; the type is Model (*i.e.*, model data type, considering model as an already identified node); the parameters are source model and target model, both of which have type Model and name "class diagram" and "first-order-logic," respectively; the sequence is 1, so such a process is the first process to be performed.

Relationships are used for linking two nodes (*i.e.* the *source node* and the *target node*). Relationships are described with five components: *type* is used for representing the type of relationship in the context of object-oriented programming, including *association* (*e.g.*, the

node "Teacher" is associated with the node "Student" as a teacher knows about some features of the students), *composition* (*e.g.*, the node "house" has some features such as "room" and "kitchen" which are also represented by using nodes), *aggregation* (*e.g.*, *generalization* (*e.g.*, the node "car" is a type of the node "vehicle"), and dependency (*e.g.*, the node "delivery driver" depends on the node "car" for delivering products); *cardinality* is used for representing the number of occurrences of one node being related to another node in a relationship, including *one-to-one cardinality* which is used for representing the relationship between one instance of the source node with one instance of the target node, *one-to-many cardinality*, which is used for representing the relationship between one instance of the source node and many instances of the target node, and *many-to-many cardinality*, which is used for representing the relationship between many instances of the source node and many instances of the target node. We define a cardinality for each node (*i.e.*, *source cardinality* and *target cardinality*); *roles* are used for representing the participation role which each node is playing in the relationship, *e.g.*, in a relationship between the nodes "software analyst" and the node "software domain," the role of the node "software analyst" (*i.e.*, *source role*) could be "modeler" and the role of the node "software domain" (*i.e.*, *target role*) could be "characterized requirement."

Figure 4-1. A meta-model for KRLs. The Authors.

4.4 Defining a meta-model for FSLs in the context of RE

We define a meta-model for representing FSLs in the context of RE by defining common components related to the FSL domain as we described in Section 4.2. According to the

systematic literature review, some FSL proposals (11 out of 11) are focused on the object-oriented paradigm, allowing practitioners for modeling object-oriented concepts such as classes and functions. Thus, we define seven components describing the key features of any object-oriented FSL as shown in Figure 4-2 and described as follows:

Classes are used for representing key entities and concepts related to a software domain, including their structure and behavior. Such representation comprises a set of elements allowing for a broader characterization of the class: *names* are used for representing a textual identification of the class (*e.g.*, "Software analyst"); *protocols* are used for specifying the behavior and capabilities related to a class so practitioners may establish a clear contract for interacting with the related class while using other classes. Protocols may include information about key features, processes, and constraints related to a class; *aliases* are used for representing alternative names for an existing class (*e.g.*, "Modeler" could be an alias for the class "Software Analyst").

Features are used for representing the characteristics of a class which is specified within the FSL. Features provide a structured way for representing classes, including a *name* for textually identifying the feature (*e.g.*, "years of experience"), a *type* for representing the feature data type (*e.g.*, integer), an *initial value* for representing a default value for the feature (*e.g.*, 0), a *cardinality* for representing the number of occurrences of such feature in the classes (*e.g.*, 1, representing the class "Software analyst" has one value related to its feature "years of experience"), a *visibility* value for representing the scope of the feature in the context of oriented-object programming (*i.e.*, public, private, protected), and a *derivation* expression for representing the derivation process so practitioners may determine how they can derive a specific feature based on other feature structure and behavior.

Processes are used for representing the behavior of a specified class. Such components provide a dynamic view of the class, allowing practitioners for representing the interaction capabilities of the class. Processes are based on several components: *names* are used for identifying a specific process (*e.g.*, "Models"); *visibility* is used for representing the scope of the process in the context of object-oriented programming (*i.e.*, private, public protected) so the usage of the process may be clearly limited; *type* is used for representing the data type of the returning element related to the process (*e.g.,* Model as data type, representing an object of the class Model); *parameters* are used for representing the input elements used while performing the process, including their name and type (*e.g.*, "Software domain" representing an object of the class Software domain); *valuations* are used for defining formulae unambiguously representing a specific feature including values, truth values, and previously specified objects; *preconditions* and *postconditions* are used for representing the conditions which should remain true before and after the execution of a specific process, so practitioners may express constraints and assumptions related to the behavior of a process. Such components are represented by using formulae; *participation mode* is used for representing the type of interaction which should be performed by using the process, including *called member* and *shared-with member*. Called member participation refers to a

process which is invoked by another component within the process. Shared-with member refers to a process which is being shared and accessed from several components within the process; *Constraints* are used for defining the limitations of the specified class. Such limitations are expressed by using formulae including related features.

Functions are used for representing specific transformations, computations, and mappings of input data (function parameters) to output data. Functions comprise four components: *name* is used for textually identifying the function; *type* is used for representing the data type related to the expected output of the function; *formula* is used for representing the algorithm and logic related to the computation process being performed by using the function; *parameters* are used for representing the input data of the function. Parameters are characterized by a specific name and type (*i.e.*, data type). While the structure is similar to class processes, *functions* provide a broader scope as they may be performed in the context of the FSL rather than the scope of a specific class.

Data type is used for representing data types defined in the context of the FSL. Such a component comprises well-known data types such as integer, string, and boolean, and user-defined data types such as Modeler, representing the data type of an object of the specified class Modeler.

Invariants are used for representing the constraints and assertions which remain unchanged while performing specific processes and functions. Invariants are identified by specific *names* and provide a set of *invariant attributes*, which are represented by name and type, a set of *invariant preconditions*, which are represented by formulae defining the conditions to be true before evaluating the specified invariant, and a *formula* formally representing the components of the invariant.

Relationships are used for linking two classes (*i.e.* the *source class* and the *target class*). Relationships are described with five components: *type* is used for representing the type of relationship in the context of object-oriented programming, including association, composition, aggregation, generalization, and dependency; *cardinality* is used for representing the number of occurrences of one class being related to another class in a relationship, including *one-to-one cardinality* which is used for representing the relationship between one instance of the source class with one instance of the target class, *one-to-many cardinality*, which is used for representing the relationship between one instance of the source class and many instances of the target class, and *many-to-many cardinality*, which is used for representing the relationship between many instances of the source class and many instances of the target class. We define a cardinality for each class (*i.e.*, *source cardinality* and *target cardinality*); *roles* are used for representing the participation role which each class is playing in the relationship, *e.g.*, in a relationship between the classes “software analyst” and the class “software domain,” the role of the class “software analyst” (*i.e.*, *source role*) could be “modeler” and the role of the class “software domain” (*i.e.*, *target role*) could be “characterized requirement.”

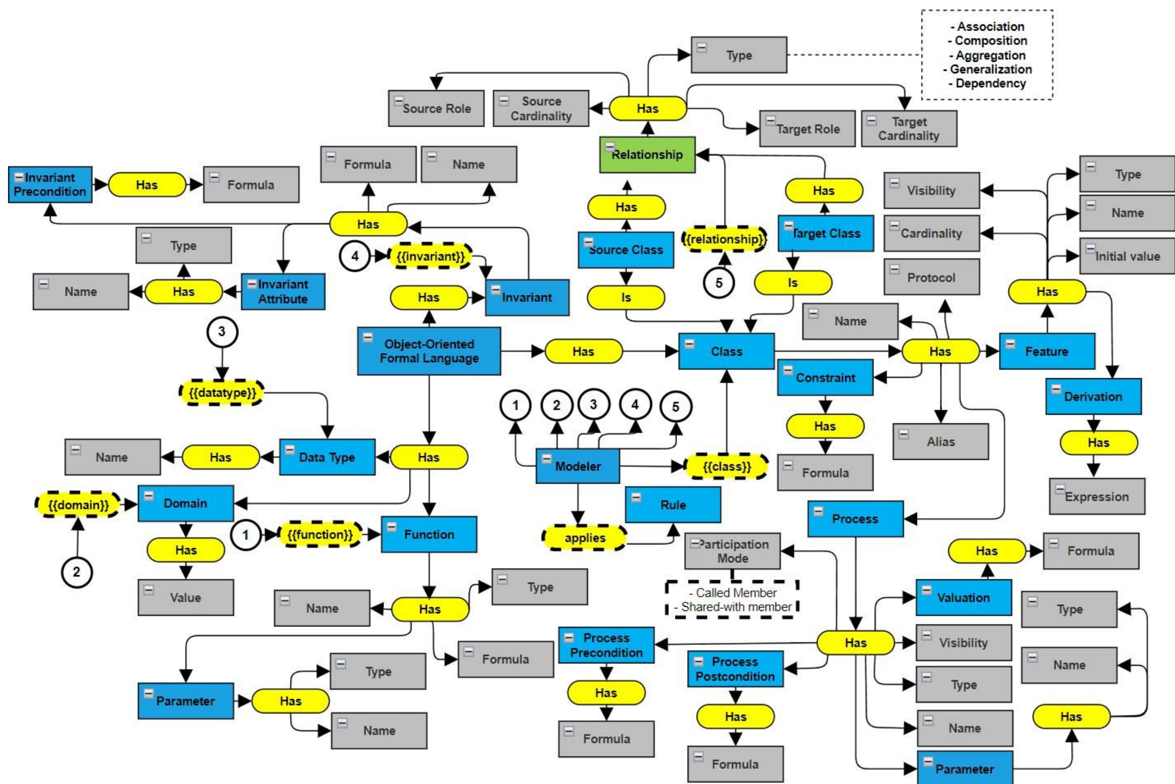

Figure 4-2. A meta-model for FSLs. The Authors.

4.5 Defining a meta-model for transformation rules for KRL-FSL pairs

We define a meta-model for representing model-to-model transformation rules in the context of KRL-FSL pairs. The meta-model is built upon the previously proposed meta-models for KRLs and FSLs. Eventhough KRLs and FSLs are distinct languages used for representing different perspectives of software domains, both KRLs and FSLs share common components, such as concepts, relationships, and features, which are fundamental to representing any software domain. We identify 22 equivalencies between the meta-model for KRLs and the meta-model for FSLs by using an in-depth analysis of the proposed meta-models, as outlined in Table 2. Such equivalencies serve as the basis for defining heuristic rules encompassing both KRLs and FSLs, enabling practitioners (*i.e.*, Modelers) to perform bidirectional transformations between the two representations. The heuristic rules encompass two principal components: *direction* is used for specifying whether the equivalence should be applied from KRL to FSL or *vice-versa*, while *equivalence* is employed to denote the corresponding KRL and FSL elements being used on the transformation when the rule is applied.

Table 4-1: KRL-FSL pair equivalencies. The Authors.

Number	KRL Element	KRL Name	FSL Element	FSL Name
1	Node		Class	Class
2	Node	Node.Name	Class	Class.Name
3	Node	Node.Feature	Class	Class.Feature
4	Node	Node.Process	Class	Class.Process
5	Node	Source Node.Name	Class	Source Class
6	Node	Target Node.Name	Class	Target Class
7	Feature	Feature.Visibility	Feature	Feature.Visibility
8	Feature	Feature.Type	Feature	Feature.Type
9	Feature	Feature.Name	Feature	Feature.Name
10	Relationship	Relationship	Relationship	Relationship
11	Relationship	Relationship.Source Cardinality	Relationship	Relationship.Source Cardinality
12	Relationship	Relationship.Target Cardinality	Relationship	Relationship.Target Cardinality
13	Relationship	Relationship.Source Role	Relationship	Relationship.Source Role
14	Relationship	Relationship.Target Role	Relationship	Relationship.Target Role
15	Relationship	Relationship.Type	Relationship	Relationship.Type
16	Process	Process	Process	Process
17	Process	Process.Visibility	Process	Process.Visibility
18	Process	Process.Type	Process	Process.Type
19	Process	Process.Name	Process	Process.Name
20	Process	Process.Parameter	Process	Process.Parameter
21	Parameter	Parameter.Type	Parameter	Parameter.Type
22	Parameter	Parameter.Name	Parameter	Parameter.Name

Such equivalencies comprise several key attributes: *number* which is used for uniquely identifying each equivalence; *KRL element* and *FSL element* which are textual identifications of the KRL and FSL elements (*e.g.*, node and class); *KRL name* and *FSL name* which are the specific nomenclatures used in both the KRL and FSL meta-models (*e.g.*, node.feature and class.feature); *KRL value* and *FSL value* which are used for representing the associated values for each component in the representation. The proposed meta-model is summarized in Figure 4-3.

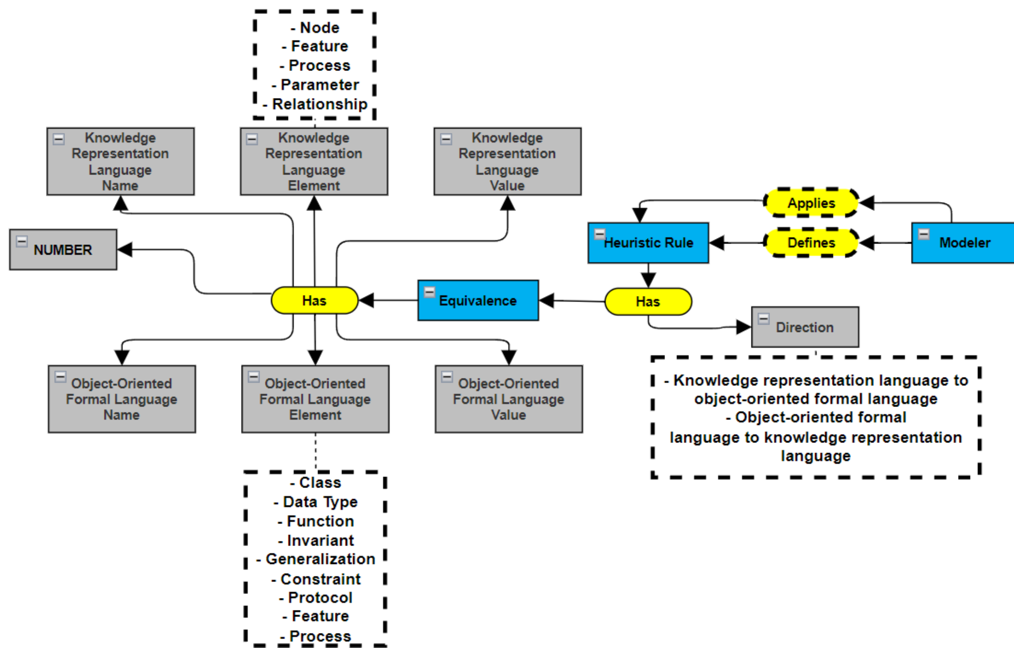

Figure 4- 3. A meta-model for model-to-model transformation rules for KRL-FSL pairs. The Authors.

4.6 Defining a method for transforming KRL-FSL pairs

We present a novel method for the bidirectional transformation of KRL-FSL pairs, regardless of the specific type of KRL, FSL, and software domain. To achieve this, we introduce a comprehensive integration of three meta-models, which offer a unified approach for representing and transforming KRLs and FSLs in the context of both RE and object-oriented programming.

The proposed meta-models serve as high-level abstractions, representing key components and relationships inherent in the fields of KRLs and FSLs. Such meta-models provide a structured method, easing a systematic transformation process across different types of KRLs and FSLs, enabling modelers for exchanging knowledge between KRLs and FSLs, and grasping a broader understanding of the software domain.

Our method provides a flexible framework for software analysts as they can represent any KRL and FSL and transform them into other FSL and KRL, respectively, by using bidirectional equivalences related to the proposed meta-models for KRLs and FSLs. Also, we represent such meta-models by using PCSs as they are models close to natural language, easing the understanding and interpretation of the proposed meta-models, and allowing non-technical and novice modelers for using our method. Such an approach is intended to minimize the inherent complexity related to representing knowledge coming from complex software systems by using KRLs and FSLs.

The proposed method for transforming KLR-FSL pairs comprises five different steps. We summarize such steps in Figure 4-4 and describe them as follows:

Software domain selection: the modeler selects a specific software domain for knowledge representation, defining the context of the software system to be represented. Such a step is key as the modelers should use the software domain elements for guiding the representation process.

Source model selection: the modeler selects a *source model* which is the starting point for the transformation process. The source model may be either a KRL or a FSL.

Figure 4-4. A method for transforming KRL-FSL pairs. The Authors.

Source model representation: the modeler uses the proposed meta-model representation for the selected source model (either the meta-model for KRLs or the meta-model for FSLs) for representing the source model. In this step, the modeler characterizes the software domain elements such as classes, attributes, and relationships by using the selected source model, allowing her for building the source model representation.

Heuristic rule application: the modeler uses the proposed meta-model for transformation rules by applying the identified equivalencies so she can build the target model. In this step, the modeler maps the values coming from the source model elements to the target model elements rule by rule as they may be applied. Such a step is key as it allows modelers for maintaining the consistency between models.

Target model generation: the modeler generates the target model representation by using the applied equivalences and the target model meta-model. Such a representation comprises the same software domain elements represented in the context of the target model, including all its elements and relationships, when applicable.

5. Validation

5.1 Case study planning

We validate this Ph.D. Thesis by using an experimental process based on case study research. We develop a prototype based on the proposed method and we evaluate it on three case studies by using a consistency metric for measuring the number of elements coming from KRLs and FSLs which can be represented and transformed by using our method. We validate our proposal based on the *design science methodology for information systems and software engineering* (Wieringa, 2014) and the *case study research process of the experimentation in software engineering* (Wohlin *et al*., 2012), including five components: *objective, case studies, theory, research questions, and method*.

5.1.1 Objective

Analyze the representation and transformation capabilities of the proposed method.

5.1.2 Case studies

The case studies comprise well-known representation languages in the context of RE.

- Case study #3 (CS3): OASIS is an object-oriented FSL (Pastor *et al*., 1992). In this case study, we validate the KRL to FSL transformation capability by using a class diagram as a source model and the OASIS model as the target model.

- Case study #2 (CS2): the UNC-Method is a software development method focused on problems and goals (Zapata & Arango, 2009). Such a method comprises several KRLs such as a process diagram, class diagram, and use case diagram. In this case study, we validate the KRL to FSL transformation capabilities of our method by using well-known KRLs and show how they can be transformed into different object-oriented FSLs.

- Case study #3 (CS3): UN-LEND is an object-oriented formal language (Mosquera, 2021). In this case study, we validate the FSL to KRL transformation capability by using UN-LEND as a target source and the KRL meta-model representation as the target model.

5.1.3 Theory

We propose the *Consistency Metric* (CM) for evaluating the representation capabilities of our method. Such a metric is aimed at measuring the proportion of elements of the source model which can be represented by the target model. The CM depends on two variables, the *Number of Represented Elements in the Target model* (NRET) and the *Number of Represented Elements in the Source model* (NRES). The CM is formally described in Equation (1):

$$CM = \frac{NRET}{NRES} \qquad (1)$$

We compute four different CM values for evaluating the transformation process:

- The *Source-Model-to-Source-Model-Meta-Model CM* (SM2SMM-CM) is aimed at computing the number of elements coming from the source model which may be represented by the source model meta-model.

- The *Source-Model-Meta-Model-to-Target-Model-Meta-Model CM* (SMM2TMM-CM) is used for computing the number of elements coming from the source meta-model which may be represented by the target model meta-model.

- The *Target-Model-Meta-Model-to-Target-Model CM* (TMM2TM-CM) is used for computing the number of elements coming from the target model meta-model which may be represented by the target model.

- The *Source-Model-to-Target-Model CM* (SM2TM-CM) is used for computing the number of elements coming from the source model which may be represented by the target model.

The CM may be represented by three types of value: (i) a CM lower than 1 indicates the evaluated source model (*i.e.*, the source model for computing the CM value) lacks some domain elements which are included in the evaluated target model; (ii) a CM greater than 1 indicates the evaluated target model lacks some domain elements for representing all the domain elements coming from the evaluated source model; (iii) a CM value equals to 1 indicates both the source model and the target model fully represent the domain of interest, so they are equivalent for such a domain.

5.1.4 Validation research questions

We define two validation research questions:

- VR1: what are the CM values of the proposed method on KRL-to-FSL transformations?

- VR2: what are the CM values of the proposed method on FSL-to-KRL transformations?

5.1.5 Methods

We follow the proposed method for transforming the models related to the case studies. We manually count the number of elements we can represent by using the proposed meta-models before and after the transformation process. Then, we use such counts for evaluating the resulting representation by using the CM. We support such a process by using a prototype application which helps us to automate the transformation process.

5.2 Case study validation

5.2.1 Performing a KRL-to-FSL transformation

We analyze CS1 based on the method for transforming a KRL into a FSL by following its five steps:

Software domain selection: we select the software domain, which in CS1 is related to the medical field.

Source model selection: we represent the selected software domain by using a class diagram as shown in Figure 5-1.

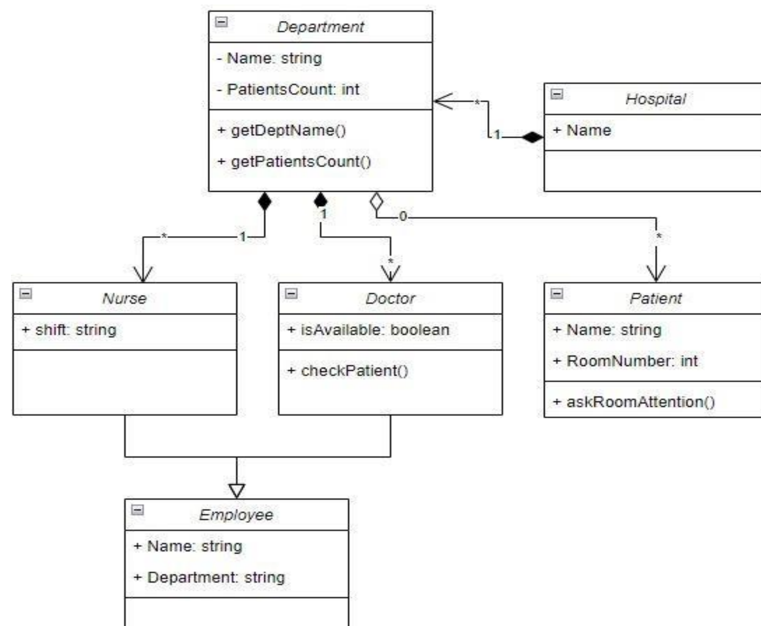

Figure 5- 1. CS1 source model representation. The Authors.

Source model representation: we use the KRL meta-model for representing our source model. The underlying structure of the class diagram can be represented with the KRL

meta-model we defined in the previous Chapter as follows. We will use ***bold italics*** for representing the meta-model elements and *italics* for instances.

- We have six ***nodes*** with the ***names*** *Hospital*, *Department*, *Nurse*, *Doctor*, *Patient*, and *Employee*.

- The ***node named*** *Department* has two ***features***: one with the ***name*** *Name*, the ***visibility*** *private*, and the ***type*** *string* and one with the ***name*** PatientCount, the ***visibility*** *private,* and the ***type*** *int*.

- The ***node named*** *Nurse* has one ***feature*** with ***name*** *shift*, ***visibility*** *public*, and ***type*** *string*.

- The ***node named*** *Doctor* has one ***feature*** with the ***name*** *isAvailable*, ***visibility*** *public*, and ***type*** *boolean*.

- The ***node named*** *Patient* has two ***features***: one with the ***name*** *Name*, ***visibility*** *public*, and ***type*** *string*, and one with the ***name*** *RoomNumber*, ***visibility*** *public*, and ***type*** *int*.

- The ***node named*** *Employee* has two ***features***: one with the ***name*** *Name*, ***visibility*** *public*, and ***type*** *string*, and one with the ***name*** *Department*, ***visibility*** *public*, and ***type*** *string*.

- The ***node named*** *Department* has one ***process*** with the ***name*** *getDeptName*, the ***visibility*** *public*, and the ***sequence*** *1*.

- The ***node named*** *Department* has one ***process*** with the ***name*** *getPatientsCount*, the ***visibility*** *public*, and the ***sequence*** *2*.

- The ***node named*** *Doctor* has one ***process*** with the ***name*** *checkPatient*, the ***visibility*** *public*, and the ***sequence*** *1*.

- The ***node named*** *Patient* has one ***process*** with the ***name*** *askRoomAttention*, the ***visibility*** *public*, and the ***sequence*** *1*.

- The ***node named*** *Hospital* has one ***relationship*** with ***type*** *composition* in which it is the ***source node*** with the ***target node named*** *Department.*

- The ***node named*** *Department* has one ***relationship*** with ***type*** *composition* in which it is the ***source node*** with the ***target node named*** *Doctor.*

- The ***node named*** *Department* has one ***relationship*** with ***type*** *composition* in which it is the ***source node*** with the ***target node named*** *Nurse.*

- The ***node named*** *Department* has one ***relationship*** with ***type*** *aggregation* in which it is the ***source node*** with the ***target node named*** *Patient.*

- The ***node named*** *Department* has one ***relationship*** with ***type*** *composition* in which it is the ***source node*** with the ***target node named*** *Doctor.*

- The ***node named*** *Employee* has two ***relationships*** with ***type*** *generalization* in which it is the ***source node***: one with the ***target node named*** *Doctor* and one with the ***target node named*** *Nurse*.

The previous information completely covers the class diagram as follows:

- The basic information is extracted from the class diagram.

- ***Nodes*** are considered ***Classes*** in the class diagram, *e.g.*, *Doctor*.

- ***Features*** are considered ***Class attributes*** in the class diagram, *e.g.*, *Name*.

- ***Processes*** are considered ***Class methods*** in the class diagram, *e.g., checkPatient*.

- *Aggregation*, *composition*, and *generalization* ***relationships*** are represented equivalently in the class diagram.

Heuristic rule application: we analyze the equivalences from the heuristic rules, so we can represent the target model into one instance of the FSL meta-model.

After we apply the transformation rules, we obtain the following results:

- We have six ***classes*** with the ***names*** *Hospital*, *Department*, *Nurse*, *Doctor*, *Patient*, and *Employee*.

- The ***class named*** *Department* has two ***features***: one with the ***name*** *Name*, the ***visibility*** *private*, and the ***type*** *string,* and one with the ***name*** PatientCount, the ***visibility*** *private,* and the ***type*** *int*.

- The ***class named*** *Nurse* has one ***feature*** with ***name*** *shift*, ***visibility*** *public*, and ***type*** *string*.

- The ***class named*** *Doctor* has one ***feature*** with the ***name*** *isAvailable*, ***visibility*** *public*, and ***type*** *boolean*.

- The ***class named*** *Patient* has two ***features***: one with the ***name*** *Name*, ***visibility*** *public*, and ***type*** *string*, and one with the ***name*** *RoomNumber*, ***visibility*** *public*, and ***type*** *int*.

- The ***class named*** *Employee* has two ***features***: one with the ***name*** *Name*, ***visibility*** *public*, and ***type*** *string*, and one with the ***name*** *Department*, ***visibility*** *public*, and ***type*** *string*.

- The ***class named*** *Department* has one ***process*** with the ***name*** *getDeptName*, the ***visibility*** *public*, and the ***sequence*** *1*.

- The ***class named*** *Department* has one ***process*** with the ***name*** *getPatientsCount*, the ***visibility*** *public*, and the ***sequence*** *2*.

- The ***class named*** *Doctor* has one ***process*** with the ***name*** *checkPatient*, the ***visibility*** *public*, and the ***sequence*** *1*.

- The ***class named*** *Patient* has one ***process*** with the ***name*** *askRoomAttention*, the ***visibility*** *public*, and the ***sequence*** *1*.

- The ***class named*** *Hospital* has one ***relationship*** with ***type*** *composition* in which it is the ***source class*** with the ***target class named*** *Department.*

- The ***class named*** *Department* has one ***relationship*** with ***type*** *composition* in which it is the ***source class*** with the ***target class named*** *Doctor.*

- The ***class named*** *Department* has one ***relationship*** with ***type*** *composition* in which it is the ***source class*** with the ***target class named*** *Nurse.*

- The ***class named*** *Department* has one ***relationship*** with ***type*** *aggregation* in which it is the ***source class*** with the ***target class named*** *Patient.*

- The ***class named*** *Department* has one ***relationship*** with ***type*** *composition* in which it is the ***source class*** with the ***target class named*** *Doctor.*

- The ***class named*** *Employee* has two ***relationships*** with ***type*** *generalization* in which it is the ***source class***: one with the ***target class named*** *Doctor* and one with the ***target class named*** *Nurse*.

Target model generation: the previous information can be represented in different object-oriented FSLs as follows. In this case, we represent the transformed information by using the OASIS model (Pastor *et al.*, 1992).

class Hospital:

identification

```
        constant attributes
        derived attributes
        derivations
        events
        preconditions
end class
class Department:
        identification
        constant attributes
        derived attributes
                Name:string;
                PatientsCount:int;
        derivations
        events
                getDeptName
                getPatientsCount
        preconditions
end class
class Nurse:
        identification
        constant attributes
        derived attributes
                shift:string;
        derivations
        events
        preconditions
end class
class Doctor:
        identification
        constant attributes
        derived attributes
                isAvailable:boolean;
```

```
    derivations
    events
        checkPatient
    preconditions
end class
class Employee:
    identification
    constant attributes
    derived attributes
        Name:string;
        Department:string;
    derivations
    events
    preconditions
end class
class Patient:
    identification
    constant attributes
    derived attributes
        Name:string;
        RoomNumber:int;
    derivations
    events
        askRoomAttention
    preconditions
end class
Hospital dynamic composition of
Department;
Department dynamic aggregation of Nurse;
Department dynamic aggregation of Doctor;
Department dynamic aggregation of Patient;
Doctor static specialization of Employee;
```

Nurse static specialization of Employee;

Be advised that the OASIS language has no equivalences for ***relationships*** with ***type*** *association*, and for ***sequences*** of the ***processes***. Also, we have no information coming from the knowledge representation language related to ***derivations*** and ***preconditions*** of OASIS, as well as many other details linked to the OASIS specification.

We summarize the transformation process in Table 5-1.

Table 5-1: CS1 represented elements. The Authors.

Software domain element name	KRL element	KRL meta-model element	KRL meta-model complementary elements	FSL meta-model element	FSL meta-model complementary elements	FSL element (OASIS)
Hospital	Class	Node	-	Class	-	Hospital
Department	Class	Node	-	Class	-	Department
Nurse	Class	Node	-	Class	-	Nurse
Doctor	Class	Node	-	Class	-	Doctor
Patient	Class	Node	-	Class	-	Patient
Employee	Class	Node	-	Class		Employee
Hospital.Name	Class atribute	Feature	Visibility: public Type: string	Feature	Visibility: public Type: string	Hospital.Name: string
Department.Name	Class attribute	Feature	Visibility: private Type: string	Feature	Visibility: private Type: string	Department.Name: string
Department.PatientsCount	Class attribute	Feature	Visibility: private Type: int	Feature	Visibility: private Type: int	Department.PatientsCount: int
Nurse.shift	Class attribute	Feature	Visibility: public Type: string	Feature	Visibility: public Type: string	Nurse.shift: string
Doctor.isAvailable	Class attribute	Feature	Visibility: public Type: boolean	Feature	Visibility: public Type: boolean	Doctor.isAvailable: boolean
Patient.Name	Class attribute	Feature	Visibility: public Type: string	Feature	Visibility: public Type: string	Patient.Name: string
Patient.RoomNumber	Class attribute	Feature	Visibility: public Type: int	Feature	Visibility: public Type: int	Patient.RoomNumber: int
Employee.Name	Class attribute	Feature	Visibility: public Type: string	Feature	Visibility: public Type: string	Employee.Name: string
Employee.Department	Class attribute	Feature	Visibility: public Type: string	Feature	Visibility: public Type: string	Employee.Department: string
getDeptName	Class Method	Process	Visibility: public Sequence: 1	Process	Visibility: public Sequence: 1	getDeptName
getPatientsCount	Class Method	Process	Visibility: public Sequence: 1	Process	Visibility: public Sequence: 1	getPatientsCount
checkPatient	Class Method	Process	Visibility: public Sequence: 1	Process	Visibility: public Sequence: 1	checkPatient
askRoomattention	Class Method	Process	Visibility: public Sequence: 1	Process	Visibility: public Sequence: 1	askRoomattention
Composition relationship between Hospital and Department	Composition relationship	Composition relationship	Source node: Hospital Target node: Department Type: Composition	Composition relationship	Source node: Hospital Target node: Department Type: Composition	Aggregation relationship between Hospital and Department

Table 5-1: CS1 represented elements. The Authors. (Continuation)

Software domain element name	KRL element	KRL meta-model element	KRL meta-model complementary elements	FSL meta-model element	FSL meta-model complementary elements	FSL element (OASIS)
Composition relationship between Department and Doctor	Composition relationship	Composition relationship	Source node: Department	Composition relationship	Source node: Department	Aggregation relationship between Department and Doctor
Composition relationship between Department and Nurse	Composition relationship	Composition relationship	Target node: Doctor Type: Composition Type: Composition	Composition relationship	Target node: Doctor Type: Composition Type: Composition	Aggregation relationship between Department and Nurse
Aggregation relationship between Department and Patient	Aggregation relationship	Aggregation relationship	Source node: Department Target node: Patient Type: Aggregation	Aggregation relationship	Source node: Department Target node: Patient Type: Aggregation	Aggregation relationship between Department and Patient
Generalization relationship between Employee and Doctor	Generalization relationship	Generalization relationship	Source node: Employee Target node: Doctor Type: Generalization	Generalization relationship	Source node: Employee Target node: Doctor Type: Generalization	Generalization relationship between Employee and Doctor
Generalization relationship between Employee and Nurse	Generalization relationship	Generalization relationship	Source node: Employee Target node: Nurse Type: Generalization	Generalization relationship	Source node: Employee Target node: Nurse Type: Generalization	Generalization relationship between Employee and Nurse

CS1 comprises 19 different domain elements, including classes, features, and processes. We manage to represent and extend such domain elements by using 63 KRL meta-model elements and FSL meta-model elements.

We compute 4 different CM values for the CS1:

- We achieve a SM2SMM-CM value of 1 which shows that our KRL meta-model fully represents the source model, including all its classes, class attributes, class methods, and relationships.

- We achieve a SMM2TMM-CM value of 1 which shows that the proposed KRL meta-model and the FSL meta-model are consistent with the representation of the CS1. Such a CM value indicates that every represented element coming from the KRL meta-model is fully represented in the FSL meta-model.

- We achieve a TMM2TM-CM value of 1 which shows that the proposed FSL meta-model represents all the necessary elements of the target model in the context of the CS1. Therefore, the context model may be represented by the FSL meta-model.

- We achieve a SM2TM-CM value of 1. Such a value represents the method capability for transforming a class diagram into the OASIS model in the context of the CS1, showing that the represented software domain may be fully represented and bidirectionally transformed.

We summarize the CM values related to CS1 in Table 5-2.

Table 5-2: CS1 CM values summary. The Authors.

	NES	NET	CM
SM2SMM (KRL to KRLMM)	63	63	**1**
SMM2TMM (KRLMM to FSLMM)	63	63	**1**
TMM2TM (FSLMM to Target FSL)	63	63	**1**
SM2TM (Source KRL to Target FSL)	63	63	**1**

We analyze CS2 based on our method for transforming a KRL into a FSL by following its five steps:

Software domain selection: we select the software domain, which in CS1 is related to the sales management field.

Source model selection: since the UNC-Method includes some knowledge representation languages such as the process diagram, the class diagram, and the use case diagram, we select the three of them as source models. While our method is intended to work with one source model at a time, we may use it for representing any KRL. Such diagrams are depicted in Figure 5-2, Figure 5-3, and Figure 5-4.

Source model representation: we use the KRL meta-model for representing our source models. The underlying structure of the three diagrams can be represented with the KRL meta-model we defined in the previous Chapter as follows. We will use ***bold italics*** for representing the meta-model elements and *italics* for instances.

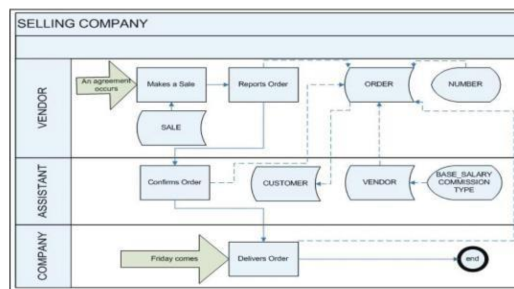

Figure 5- 2. Process diagram representation (Zapata and Arango, 2009).

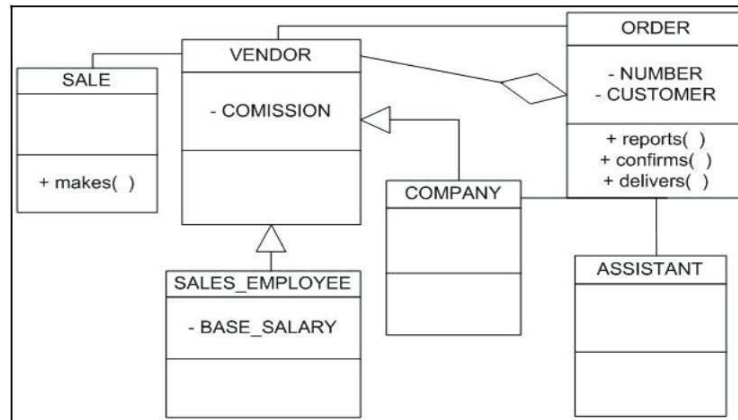

Figure 5-3. Class diagram representation *(Zapata & Arango, 2009)*.

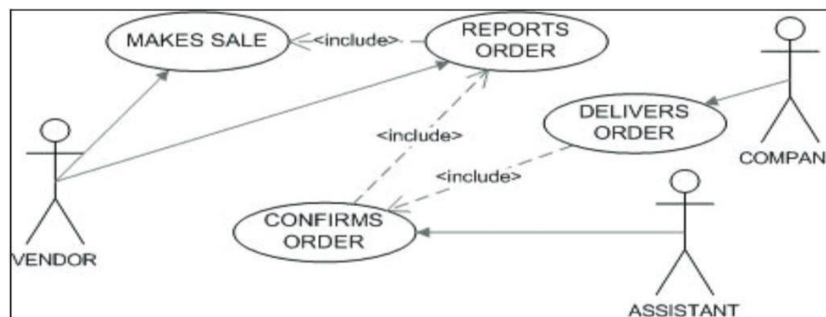

Figure 5- 4. Case diagram representation (Zapata and Arango, 2009).

- We have six ***nodes*** with the ***names*** *sale, vendor, sales employee, company, order,* and *assistant.*
- The ***node named*** *vendor* has one ***feature*** with the ***name*** *commission* and the ***visibility*** *private*. Be advised that we need at least the ***type*** of the ***feature*** with a value *int*, even though it is not declared in the information we have in the diagrams.
- The ***node named*** *order* has two ***features:*** one with the ***name*** *number* and the ***visibility*** *private* and one with the ***name*** *customer* and the ***visibility*** *private*. Again, we need at least the ***type*** of the ***feature*** *number* with a value *int* and the ***type*** of the ***feature*** *customer* with a value *string* even though it is not declared in the information we have in the diagrams.
- The ***node named*** *order* has three ***processes:*** one with the ***name*** *reports*, the ***visibility*** *public,* and the ***sequence*** 2; one with the ***name*** *confirms*, the ***visibility*** *public,* and the ***sequence*** 3; and one with the ***name*** *delivers,* the ***visibility*** *public,* and the ***sequence*** 4.

- The ***node named*** *sales employee* has one ***feature*** with the ***name*** *base salary* and the ***visibility*** *private*. Additionally, the ***type*** of the ***feature*** is *int*, even though it is not declared in the information we have in the diagrams.
- The ***node named*** *sale* has one ***process*** with the ***name*** *makes*, the ***visibility*** *public,* and the ***sequence*** 1.
- The ***node named*** *vendor* has two ***relationships*** with ***type*** *generalization* in which it is the ***source node***: one with the ***target node named*** *sales employee* and one with the ***target node named*** *company*.
- The ***node named*** *order* has one ***relationship*** with ***type*** *aggregation* in which it is the ***source node*** with the ***target node named*** *vendor*.
- The ***node named*** *order* has one ***relationship*** with ***type*** *association* in which it is the ***source node*** with the ***target node named*** *vendor*.
- The ***node named*** *vendor* has one ***relationship*** with ***type*** *association* in which it is the ***source node*** with the ***target node named*** *sale*.
- The ***node named*** *order* has one ***relationship*** with ***type*** *association* in which it is the ***source node*** with the ***target node named*** *company*.
- The ***node named*** *order* has one ***relationship*** with ***type*** *association* in which it is the ***source node*** with the ***target node named*** *assistant*.

The previous information completely covers the three diagrams as follows:

- The basic information is extracted from the class diagram.
- ***Nodes*** with ***features*** are considered storages with features of the process diagram, *e.g., order* with *number*.
- ***Nodes*** with ***processes*** are considered processes of the process diagram or use cases of the use case diagram, *e.g.*, *confirms order.*
- ***Nodes*** with ***relationships*** with ***type*** *association* with nodes with ***processes*** are considered lanes or actors of the process diagram and actors of the use case diagram, *e.g., vendor.*

Heuristic rule application: we analyze the equivalences from the heuristic rules, so we can represent the target model into one instance of the FSL meta-model.

After we apply the transformation rules, we obtain the following results:

- We have six ***classes*** with the ***names*** *sale, vendor, sales employee, company, order,* and *assistant.*
- The ***class named*** *vendor* has one ***feature*** with the ***name*** *commission*, the ***visibility*** *private*, and the ***type*** *int*.
- The ***class named*** *order* has two ***features:*** one with the ***name*** *number*, the ***visibility*** *private,* and the ***type*** *int* and one with the ***name*** *customer*, the ***visibility*** *private* and the ***type*** *string*.
- The ***class named*** *sales employee* has one ***feature*** with the ***name*** *base salary*, the ***visibility*** *private*, and the ***type*** *int*.
- The ***class named*** *order* has three ***processes:*** one with the ***name*** *reports* and the ***visibility*** *public,* one with the ***name*** *confirms* and the ***visibility*** *public,* and one with the ***name*** *delivers* and the ***visibility*** *public.*
- The ***class named*** *vendor* has two ***relationships*** with ***type*** *generalization* in which it is the ***source class***: one with the ***target class named*** *sales employee* and one with the ***target class named*** *company*.
- The ***class named*** *order* has one ***relationship*** with ***type*** *aggregation* in which it is the ***source class*** with the ***target class named*** *vendor*.
- The ***class named*** *order* has one ***relationship*** with ***type*** *association* in which it is the ***source class*** with the ***target class named*** *vendor*.
- The ***class named*** *vendor* has one ***relationship*** with ***type*** *association* in which it is the ***source class*** with the ***target class named*** *sale*.
- The ***class named*** *order* has one ***relationship*** with ***type*** *association* in which it is the ***source class*** with the ***target class named*** *assistant*.
- The ***class named*** *order* has one ***relationship*** with ***type*** *association* in which it is the ***source class*** with the ***target class named*** *company*.
- The ***class named*** *sale* has one ***process*** with the ***name*** *makes* and the ***visibility*** *public.*

Target model generation: the previous information can be represented in different object-oriented FSLs as follows. In this case, we only generate two of them.

- UN-LEND (Mosquera 2021):

```
class Vendor:
        attributes:
                int: commision
        constraints:
        operations:
class SalesEmployee extends Vendor:
        attributes:
                int: baseSalary
        constraints:
        operations:
class Company extends Vendor:
        attributes:
        constraints:
        operations:
class Order:
        attributes:
                int: number
                string: customer
        constraints:
        operations:
                void: reports () {
                        }
                void: confirms () {
                        }
                void: delivers () {
                        }
class Sale:
        attributes:
        constraints:
        operations:
                void: makes () {
                        }
```

```
class Assistant:
        attributes:
        constraints:
        operations:
```

Be advised that the UN-LEND language has no equivalences for ***relationships*** with ***type*** *aggregation* and *association,* and for ***sequences*** of the ***processes***. Also, we have no information coming from the knowledge representation language related to ***constraints*** of UN-LEND.

- OASIS (Pastor *et al.,* 1992):

```
class vendor:
        identification
        constant attributes
        derived attributes
                commission:int;
        derivations
        events
        preconditions
end class
class SalesEmployee:
        identification
        constant attributes
        derived attributes
                baseSalary:int;
        derivations
        events
        preconditions
end class
class Order:
        identification
                number:int;
```

```
    constant attributes
        customer:string;
    derived attributes
    derivations
    events
        reports
        confirms
        delivers
    preconditions
end class
class Sale:
    identification
    constant attributes
    derived attributes
    derivations
    events
        makes
    preconditions
end class
class Assistant:
    identification
    constant attributes
    derived attributes
    derivations
    events
    preconditions
end class
Order dynamic aggregation of
    Vendor;
SalesEmployee static specialization of
    Vendor;
Company static specialization of
```

Vendor;

Be advised that the OASIS language has no equivalences for ***relationships*** with ***type*** *association*, and for ***sequences*** of the ***processes***. Also, we have no information coming from the knowledge representation language related to ***derivations*** and ***preconditions*** of OASIS, as well as many other details linked to the OASIS specification.

We summarize the transformation process in Table 5-3.

Table 5-3: CS2 represented elements. The Authors.

Software domain element name	KRL element	KRL meta-model element	KRL meta-model complementary elements	FSL meta-model element	FSL meta-model complementary elements	FSL element (UN-Lend)	FSL element (OASIS)
Assistant	Storage (Process diagram); Class (Class diagram); Lane or Actor (Process diagram); Actor (Use case diagram)	Node	-	Class		Assistant	Assistant
Base_Salary	Feature related to a storage (Process diagram); Attributes (Class diagram)	Feautre (related to Sales_employee)	Visibility: private	Feautre (related to Sales_employee)	Visibility: private	BaseSalary:int	BaseSalary:int
			Type: int		Type: int		
Commission	Feature related to a storage (Process diagram); Attributes (Class diagram)	Feautre (related to Vendor)	Visibility: private	Feautre (related to Vendor)	Visibility: private	Comission:int	Comission:int
			Type: int		Type: int		
Company	Storage (Process diagram); Class (Class diagram)	Node	-	Class		Company	Company
Confirms	Process (Process diagram); Use case (Use case diagram)	Process (related to Order)	Visibility: public	Process (related to Order)	Visibility: public	Confirms	Confirms
			Sequence: 3		Sequence: 3		
Customer	Feature related to a storage (Process diagram); Attributes (Class diagram)	Feature (related to Order)	Visibility: private	Feature (related to Order)	Visibility: private	Customer:string	Customer:string
			Type: string		Type: string		
Delivers	Process (Process diagram); Use case (Use case diagram)	Process (related to Order)	Visibility: public	Process (related to Order)	Visibility: public	Delivers	Delivers
			Sequence: 3		Sequence: 3		
Makes	Process (Process diagram); Use case (Use case diagram)	Process (related to Sale)	Visibility: public	Process (related to Sale)	Visibility: public	Makes	Makes
			Sequence: 1		Sequence: 1		
Number	Feature related to a storage (Process diagram); Attributes (Class diagram)	Feature (related to Order)	-	Feature (related to Order)	-	Number	Number
Order	Storage (Process diagram); Class (Class diagram)	Node	-	Class	-	Order	Order
Reports	Process (Process diagram); Use case (Use case diagram)	Process (related to Order)	Visibility: public	Process (related to Order)	Visibility: public	Reports	Reports
			Sequence: 2		Sequence: 2		
Sale	Storage (Process diagram); Class (Class diagram)	Node	-	Class	-	Sale	Sale

Table 5-3: CS2 represented elements. The Authors. (Continuation)

Software domain element name	KRL element	KRL meta-model element	KRL meta-model complementary elements	FSL meta-model element	FSL meta-model complementary elements	FSL element (UN-Lend)	FSL element (OASIS)
Sales_Employee	Lane or Actors (Process diagram); Actors (Use case diagram)	Node	-	Class	-	SalesEmployee	SalesEmployee
Vendor	Lane or Actors (Process diagram); Actors (Use case diagram)	Node	-	Class	-	Vendor	Vendor
Aggregation relationship between Order and Vendor	Relationship (Class diagram)	Relationship	Source node: Order Target node: Vendor Type: aggregation	Relationship	Source node: Order Target node: Vendor Type: aggregation	NA	Aggregation relationship between Order and Vendor
Association relationship between Order and Vendor	Relationship (Class diagram)	Relationship	Source node: Order Target node: Vendor Type: association	Relationship	Source node: Order Target node: Vendor Type: association	NA	NA
Association relationship between Vendor and Sale	Relationship (Class diagram)	Relationship	Source node: Vendor Target node: Sale Type: association	Relationship	Source node: Vendor Target node: Sale Type: association	NA	NA
Association relationship between Order and Company	Relationship (Class diagram)	Relationship	Source node: Order Target node: Company Type: association	Relationship	Source node: Order Target node: Company Type: association	NA	NA
Association relationship between Order and Assistant	Relationship (Class diagram)	Relationship	Source node: Order Target node: Assistant Type: association	Relationship	Source node: Order Target node: Assistant Type: association	NA	NA
Generalization relationship between Sales Employee and Vendor	Relationship (Class diagram)	Relationship	Source node: Sales_Employee Target node: Vendor Type: generalization	Relationship	Source node: Sales_Employee Target node: Vendor Type: generalization	Generalization relationship between Sales_Employee and Vendor	Generalization relationship between Sales_Employee and Vendor
Generalization relationship between Sales Employee and Company	Relationship (Class diagram)	Relationship	Source node: Sales_Employee Target node: Company Type: generalization	Relationship	Source node: Sales_Employee Target node: Company Type: generalization	Generalization relationship between Sales_Employee and Company	Generalization relationship between Sales_Employee and Company

CS2 comprises 35 different domain elements, including storages, classes, use cases, features, and relationships. We manage to represent and extend such domain elements by using 49 KRL meta-model elements and FSL meta-model elements.

We compute 6 different CM values for the CS1:

- We achieve a SM2SMM-CM value of 1.4 showing our KRL meta-model fully represents the source model (CS1 comprises several source models, including a class diagram, process diagram, and use case diagram). Also, since such a value is higher than 1, it indicates the source meta-model representation includes more domain elements than the elements represented by the source model. Specifically, in CS1 we complement the source model representation by adding visibility and type features for the represented nodes.

- We achieve a SMM2TMM-CM value of 1 showing the proposed KRL meta-model and the FSL meta-model are consistent with the representation of the CS1. Such a CM value indicates every represented element coming from the KRL meta-model is fully represented in the FSL meta-model.

- We compute two different TMM2TM-CM values, one for representing the transformation into the UN-LEND model, and one for representing the transformation process into the OASIS model. Such values are 0.59 and 0.65, respectively. Since the TMM2TM-CM values are lower than 1, they show the target models lack sufficient elements for representing all the domain elements coming from the FSL meta-model. Also, such values show the OASIS model provides a higher expressiveness than the UN-LEND model as it can include more elements, according to the TMM2TM-CM value.

- We compute two different SM2TM-CM values for representing the main KRL-to-FSL transformation processes. Such transformations are targeted at two different FSLs: the UN-LEND model with a SM2TM-CM value of 0.83, and the OASIS model with a SM2TM-CM value of 0.91. Such values indicate the target models are less expressive than the source models.

We summarize the CM values related to the CS2 and answer VR1 in Table 5-4.

Table 5-4: CS2 CM values summary. The Authors.

	NES	NET	CM
SM2SMM (KRL to KRLMM)	35	49	**1.4**
SMM2TMM (KRLMM to FSLMM)	49	49	**1**
TMM2TM (FSLMM to Target FSL, UN-LEND)	49	29	**0.59**
TMM2TM (FSLMM to Target FSL, OASIS)	49	32	**0.65**
SM2TM (Source KRL to Target FSL, UN-LEND**)**	35	29	**0.83**
SM2TM (Source KRL to Target FSL, OASIS)	35	32	**0.91**

5.2.2 Performing a FSL-to-KRL transformation

We follow the method for transforming a FSL into a KRL by following its five steps:

Software domain selection: we select the software domain, which is related in CS3 to the householding field.

Source model selection: we select UN-LEND as source model. Mosquera (2021) extends the UN-LEND language proposed by Zapata and Arango (2004) by adding the specification of the operations. UN-LEND is pretty similar to other object-oriented formal languages like OASIS and ObjectZ. The following specification was extracted from Mosquera (2021):

```
class Person:
    attributes:
        string: name [0,1]
        string: lastName [0,1]
        int: age [0,1]
    constraints:
        alive: age>0 and age<100
    operations:
class Man extends Person:
    attributes:
    constraints:
    operations:
class Woman extends Person:
    attributes:
    constraints:
    operations:
class Household:
    attributes:
    constraints:
    operations:
        void: contHousehold() {
            int: countOfHouseholdMembers = 0
```

```
                    for (HouseholdMembership: householdMembershipList):
                              countofHouseholdMembers = countofHouseholdMembers +
1
          }
          void: establishHousehold() {
          }
class HouseholdMembership:
     attributes:
     constraints:
     operations:
          void: addHouseholdMembership() {
          }
```

Source model representation: we use the FSL meta-model for representing our source model. The underlying structure of this specification can be represented with the OOFL metamodel we defined in the previous Chapter as follows. We will use ***bold italics*** for representing the metamodel elements and *italics* for instances.

- We have five ***classes*** with the ***names*** *person, man, woman, household,* and *householdmembership.*
- The ***class named*** *person* has three ***features***: one ***feature*** with the ***name*** *name*, the ***cardinality*** *[0,1]*, and the ***type*** *string*; one ***feature*** with the ***name*** *lastName*, the ***cardinality*** *[0,1]*, and the ***type*** *string;* and one ***feature*** with the ***name*** *age*, the ***cardinality*** *[0,1]*, and the ***type*** *int.*
- The ***class named*** *person* has one ***constraint*** with the ***formula*** *alive: age>0 and age<100*.
- The ***class named*** *household* has two ***processes:*** one with the ***name*** *counthousehold* and the ***type*** *void;* and one with the ***name*** *establishhousehold* and the ***type*** *void.*
- The ***process named*** *counthousehold* has one ***valuation*** with ***formula*** *{int: countOfHouseholdMembers = 0 for (HouseholdMembership: householdMembershipList): countofHouseholdMembers = countofHouseholdMembers + 1}*.

- The ***class named*** *householdmembership* has one ***process*** with the ***name*** *addhouseholdmembership* and the ***type*** *void*.
- The ***class named*** *person* has two ***relationships*** with ***type*** *generalization* in which it is the ***source class***: one with the ***target class named*** *man* and one with the ***target class named*** *woman*.

Heuristic rule application: we analyze the equivalences from the heuristic rules, so we can represent the target model into one instance of the KSL meta-model.

After applying the equivalences from the heuristic rules, we have the following results for instantiating the knowledge representation language metamodel:

- We have five ***nodes*** with the ***names*** *person, man, woman, household,* and *householdmembership.*
- The ***node named*** *person* has two ***relationships*** with ***type*** *generalization* in which it is the ***source node***: one with the ***target node named*** *man* and one with the ***target node named*** *woman*.
- The ***node named*** *person* has three ***features***: one ***feature*** with the ***name*** *name* and the ***type*** *string*; one ***feature*** with the ***name*** *lastName* and the ***type*** *string;* and one ***feature*** with the ***name*** *age* and the ***type*** *int.*
- The ***node named*** *household* has two ***processes:*** one with the ***name*** *counthousehold* and the ***type*** *void;* and one with the ***name*** *establishhousehold* and the ***type*** *void.*
- The ***node named*** *householdmembership* has one ***process*** with the ***name*** *addhouseholdmembership* and the ***type*** *void*.

As you can see, too much information is missing in the transformation since the usual syntax of the knowledge representation languages lacks several elements. For example, the ***valuations*** of the ***processes***, the ***constraints*** of the ***classes***, and the ***cardinality*** of the ***features*** have no equivalences in the knowledge representation language meta-model. On the other side, elements like the ***relationships*** with ***type*** *association* and *aggregation* and the ***sequence*** of the ***processes*** are absent from the specification based on UN-LEND.

Target model generation: the previous information can be represented in different KRLs. In this case, we represent the gathered knowledge by using a class diagram as shown in Figure 5-5.

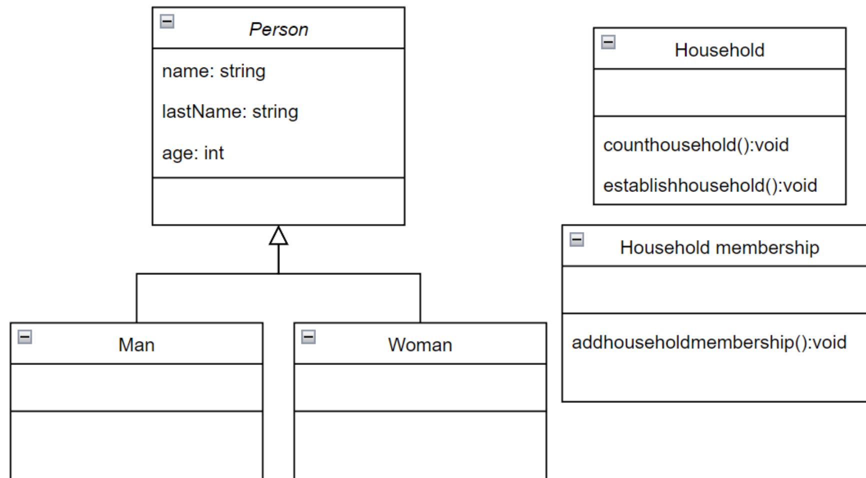

Figure 5- 5. CS3 target model representation. The Authors.

We summarize the CS3 transformation process in Table 5-5.

CS2 comprises 27 different domain elements, including classes, attributes, operations, constraints, and relationships. We manage to represent and extend such domain elements by using 27 KRL meta-model elements and FSL meta-model elements.

We compute 4 different CM values for the CS3:

- We achieve a SM2SMM-CM value of 1, showing all the domain elements coming from the source model (UN-LEND) may be represented by the FSL meta-model. In this case, we maintain the integrity of the source model by representing all its knowledge.

- We achieve a SMM2TMM-CM value of 0.89 which indicates we lost some knowledge in the transformation process as the value is lower than 1. Such a value shows the target model (the KRL meta-model) lacks some of the knowledge from the source model. Specifically, in CS2, we lack constraints in the KRL meta-model.

-We achieve a TMM2TM-CM value of 1, showing all the elements represented by the KRL meta-model may be represented by the target model (class diagram). While such a value indicates we can achieve a full transformation from the KRL meta-model to the target KRL model, we probably fail in achieve the full transformation from the initial source model (UN-LEND).

Table 5-5: CS3 transformation process summary. The Authors.

Software domain element name	FSL element (UN-LEND)	FSL meta-model element	FSL meta-model complementary elements	KRL meta-model element	KRL meta-model complementary elements	KRL element (Class diagram)
Person	Class	Class	-	Node	-	Person
name	Person attribute, type:int, cardinality: [0,1]	Feature	Type: string Cardinality: [0,1]	Feature	Type: string	name
lastname	Person attribute, type:int, cardinality: [0,1]	Feature	Type: string Cardinality: [0,1]	Feature	Type: string	lastname
age	Person attribute, type:int, cardinality: [0,1]	Feature	Type: int Cardinality: [0,1]	Feature	Type: int	age
alive	Person constraint	Constraint	formula - alive: age>0 and age<100	NA	NA	alive
Man	Class	Class	-	Class	-	Man
Woman	Class	Class	-	Class	-	Woman
Household	Class	Class	-	Class	-	Household
contHousehold	Household operation	Process	type: void valuation with formula {int: countOfHouseholdMembers = 0 for (HouseholdMembership: householdMembershipList): countofHouseholdMembers = countofHouseholdMembers + 1}.	Process	type: void NA	contHousehold
establishHousehold	Household operation	Process	type: void	Process	type: void	establishHousehold
HouseholdMembership	Class	Class	-	Class	-	HouseholdMembership
addHouseholdMembership	HouseholdMembership operation	Process	type: void	Process	type: void	addHouseholdMembership
Man extends Person	Generalization relationship	Relationship	type: generalization source class: Person target class: Man	Relationship	type: generalization source node: Person target node: Man	Man extends Person relationship
Woman extends Person	Generalization relationship	Relationship	type: generalization source class: Person target class: Woman	Relationship	type: generalization source node: Person target node: Woman	Woman extends Person relationship

- We achieve a SM2TM-CM value of 0.89 which indicates we lack a full transformation of the source model (UN-LEND) into the target model (class diagram). In the context of the FSL-to-KRL transformation, UN-LEND is more expressive than the class diagram as such a model can be used for representing more complex domain elements such as constraints.

We summarize CS3 CM values and answer to VRQ2 in Table 5-6.

Table 5-6: CS2 CM values summary. The Authors.

	NES	NET	CM
SM2SMM (FSL to FSLMM)	27	27	**1**
SMM2TMM (FSLMM to KRLMM)	27	24	**0.89**
TMM2TM (KRLMM to Target KRL, class diagram)	24	24	**1**
SM2TM (Source FSL to Target KRL, UN-LEND-to-class diagram)	27	24	**0.89**

5.3 Work products

5.3.1 Meta-models

The proposed method comprises three meta-models: the *KRL meta-model*, the *FSL meta-model*, and the *Transformation rules meta-model*. Since such meta-models comprise well-known elements coming from the KRL and FSL fields, such meta-models may be independently reused for supporting other KRL and FSL proposals in the context of RE.

5.3.2 A prototype for transforming KRL-FSL pairs

The prototype of the method for transforming KRL-FSK pairs is available at https://github.com/ramanjar/prototipo. Such a prototype integrates the meta-models and the heuristic rules, so modelers can automatically transform KRLs into FSLs and vice-versa.

6. Conclusions and challenges

6.1 Conclusions

In this Ph.D. Thesis we proposed a comprehensive method for KRL-FSL pair transformation for improving the knowledge representation and transformation process in the context of RE. We obtained the following contributions:

Regarding representation language characterization

KRLs and FSLs were characterized by performing a systematic literature review of model-to-model transformation proposals, focused on KRL-FSL pairs.

Recurrent elements related to KRLs were identified from the related KRLs within the KRL-FSL pair transformation proposals, including nodes, features, processes, and relationships.

Recurrent elements related to FSLs were identified from the related FSLs within the KRL-FSL pair transformation proposals, including classes, features, functions, processes, constraints, invariants, relationships, data types, and derivations.

Regarding representation language meta-model definition

A new KRL meta-model is defined for representing KRLs in the context of RE.

New KRL meta-model elements are based on the KRL components identified, including nodes, features, processes, and relationships.

A new FSL meta-model is defined for representing FSLs in the context of RE and object-oriented programming.

New FSL meta-model elements are based on the FSL components identified, including classes, features, functions, processes, constraints, invariants, relationships, data types, and derivations.

PCSs were used for representing meta-models as they are models close to natural languages, mitigating the need for technical skills for understanding and modeling software domains while using KRLs and FSLs.

Modelers should use the meta-models for grasping a more intuitive understanding of KRLs and FSLs.

Regarding transformation heuristic rule definition

New heuristic rules are defined for representing the equivalences between the KRL meta-model and the FSL meta-model.

Modelers should use such heuristic rules for performing model-to-model transformations in the context of KRL-FSL pairs.

Regarding the method for transforming KRL-FSL pairs

A new method for KRL-FSL pair transformation was proposed for improving the knowledge representation and transformation process in the context of RE, including a *KRL meta-model* for representing any KRL in the context of RE, a *FSL meta-model* for representing any FSL in the context of RE and object-oriented programming, and a set of *heuristic rules* for describing the equivalences between the KRL and FSL meta-models.

The method integrates the fields of KRL, FSL, and RE, allowing for describing concepts and relationships coming from a software domain, and the features and constraints related to more complex software systems.

Modelers should use the method for enhancing the comprehension and representation of complex software systems as they can combine several KRLs and FSLs, allowing them for representing different views of the software system.

The method does not impose restrictions on the type of KRL and FSL utilized or the software domain targeted, allowing modelers for adapting the method to various RE-related contexts.

Modelers should use the method for transforming KRLs into FSLs and *vice-versa*. The bidirectional nature of the proposed method eases the knowledge exchange between KRL and FSL representations, fostering the navigation between the expressive capabilities of KRLs and the unambiguous representations related to FSLs.

Our method is aimed at enhancing the comprehensibility of software domain modeling by integrating three powerful meta-models and enabling bidirectional transformations, thereby empowering software analysts and modelers for tackling increasingly complex software systems.

Regarding the method validation

The *Consistency Metric* was defined for indicating the representation and transformation capabilities related to the proposed method, including the representation of KRLs, FSLs, and the bidirectional transformation of KRL-FSL pairs.

Three *case studies* were defined for evaluating the transformation capabilities of our method, including some KRLs, FSLs, and software domains.

Validation results were obtained by using the consistency metric by indicating the method prototype can be used for representing KRLs, FSLs, and performing bidirectional transformation between KRL-FSL pairs in the context of RE regardless of the KRL-FSL pair and the software domain.

Modelers should use the consistency metric for evaluating model-to-model transformation processes in the context of KRL-FSL pairs, easing the understanding of the expressiveness and shortcomings of KRLs and FSLs in the context of RE.

6.2 Challenges

The following challenges are identified as future work from this Ph.D. Thesis:

Our method is focused on KRL-FSL pairs. However, other unattended types of representation languages can be addressed, such as domain-specific languages, ontology languages, control specification languages, and rule-based languages.

Our method allows modelers for performing bidirectional transformations. However, ensuring the bidirectional transformation method produces equivalent and complete representations remains challenging as KRLs are not as expressive as FSLs, hardening the FSL-to-KRL transformation because of the loss of knowledge in the process.

The CM can be used and extended in other model transformation scenarios such as model-to-text and text-to-model.

The method for transforming KRL-FSL pairs can be used as an educational environment for novice software analysts and modelers, allowing them for mitigating the impact of the complexity related to the understanding and adoption of KRLs and FSLs while learning and performing software modeling.

References

Abdelnabi, E. A., Maatuk, A. M., Abdelaziz, T. M., & Elakeili, S. M. (2020). Generating UML Class Diagram using NLP Techniques and Heuristic Rules. *2020 20th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA)*, 277–282. https://doi.org/10.1109/STA50679.2020.9329301

Abdelnabi, E., Maatuk, A., & Hagal, M. (2021). *Generating UML Class Diagram from Natural Language Requirements: A Survey of Approaches and Techniques*. 288–293. https://doi.org/10.1109/MI-STA52233.2021.9464433

Abrial, J.-R. (2010). Modeling in Event-B - System and Software Engineering. In *Modeling in Event-B: System and Software Engineering*. https://doi.org/10.1017/CBO9781139195881

Alkhammash, E. (2020). Formal modelling of OWL ontologies-based requirements for the development of safe and secure smart city systems. *Soft Computing*, *24*. https://doi.org/10.1007/s00500-020-04688-z

Amjad, A., Azam, F., Anwar, M., Haider, W., Rashid, M., & Naeem, A. (2018). UMLPACE for Modeling and Verification of Complex Business Requirements in Event-driven Process Chain (EPC). *IEEE Access*, *6*, 1. https://doi.org/10.1109/ACCESS.2018.2883610

Ang, A., & Hartley, M. (2007). *Object oriented knowledge representation framework for requirements engineering*. 477–482.

Awan, M. M., Butt, W. H., Anwar, M. W., & Azam, F. (2022). Seamless Runtime Transformations from Natural Language to Formal Methods – A usecase of Z-Notation. *2022 17th Annual System of Systems Engineering Conference (SOSE)*, 375–380. https://doi.org/10.1109/SOSE55472.2022.9812644

Ben Younes, A., Ben Daly Hlaoui, Y., Ben Ayed, L., & Bessifi, M. (2019). From BPMN2 to Event B: A Specification and Verification Approach of Workflow Applications. *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, *2*, 561–566. https://doi.org/10.1109/COMPSAC.2019.10266

Bernardo, M., Ciancarini, P., & Donatiello, L. (2002). Architecting Families of Software Systems with Process Algebras. *ACM Trans. Softw. Eng. Methodol.*, *11*(4), 386–426. https://doi.org/10.1145/606612.606614

Borgida, A., Horkoff, J., & Mylopoulos, J. (2014). Applying knowledge representation and reasoning to (simple) goal models. *2014 IEEE 1st International Workshop on Artificial Intelligence for Requirements Engineering (AIRE)*, 53–59. https://doi.org/10.1109/AIRE.2014.6894857

Boussetoua, R., Bennoui, H., Chaoui, A., Khalfaoui, K., & Kerkouche, E. (2015). An automatic approach to transform BPMN models to Pi-Calculus. *2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA)*, 1–8. https://doi.org/10.1109/AICCSA.2015.7507176

Bruijn, J. (2007). Logics for the Semantic Web. *Semantic Web Services: Theory, Tools and Applications*, 24–43. https://doi.org/10.4018/978-1-59904-045-5.ch002

Bruneliere, H., Cabot, J., Dupé, G., & Madiot, F. (2014). MoDisco: a Model Driven Reverse Engineering Framework. *Information and Software Technology*, *56*. https://doi.org/10.1016/j.infsof.2014.04.007

Caetano, A., Antunes, G., Pombinho, J., Bakhshandeh, M., Granjo, J., Borbinha, J., & da Silva, M. M. (2017). Representation and analysis of enterprise models with semantic techniques: an application to ArchiMate, e3value and business model canvas. *Knowledge and Information Systems*, *50*(1), 315–346. https://doi.org/10.1007/s10115-016-0933-0

Chu, M.-H., & Dang, D.-H. (2020). Automatic Extraction of Analysis Class Diagrams from Use Cases. *2020 12th International Conference on Knowledge and Systems Engineering (KSE)*, 109–114. https://doi.org/10.1109/KSE50997.2020.9287702

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., & Quesada, J. (2002). Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, *285*, 187–243.

Correia, A., & e Abreu, F. (2012). Adding Preciseness to BPMN Models. *Procedia Technology*, *5*, 407–417. https://doi.org/10.1016/j.protcy.2012.09.045

Couto, R., Ribeiro, A., & Campos, J. (2014). Application of Ontologies in Identifying Requirements Patterns in Use Cases. *Electronic Proceedings in Theoretical Computer Science*, *147*. https://doi.org/10.4204/EPTCS.147.5

Dick, J., Hull, E., & Jackson, K. (2017). *Requirements Engineering* (4th ed.). Springer, Chan.

Djaoui, C., Kerkouche, E., Chaoui, A., & Khalfaoui, K. (2018). A Graph Transformation Approach to Generate Analysable Maude Specifications from UML Interaction Overview Diagrams. *2018 IEEE International Conference on Information Reuse and Integration (IRI)*, 511–517. https://doi.org/10.1109/IRI.2018.00081

Dubois, E., Hagelstein, J., Lahou, E., Ponsaert, F., & André, R. (1986). A knowledge representation language for requirements engineering. *Proceedings of the IEEE*, *74*, 1431–1444. https://doi.org/10.1109/PROC.1986.13644

Finne, A. (2011). Towards a quality meta-model for information systems. *Software Quality Journal*, *19*(4), 663–688. https://doi.org/10.1007/s11219-011-9131-1

Gasevic, D., Djuric, D., & Devedzic, V. (2006). Model Driven Architecture and Ontology Development. In *Model Driven Architecture and Ontology Development*. https://doi.org/10.1007/3-540-32182-9

Giorgini, P., Mylopoulos, J., Nicchiarelli, E., & Sebastiani, R. (2002). Reasoning with Goal Models. *LNCS*, *2503*, 167–181. https://doi.org/10.1007/3-540-45816-6_22

Gogolla, M., Hilken, F., & Doan, K.-H. (2017). Achieving Model Quality through Model Validation, Verification and Exploration. *Computer Languages, Systems & Structures*, *54*. https://doi.org/10.1016/j.cl.2017.10.001

Hahn, C., Schmitt, F., Tillman, J., Metzger, N., Siber, J., & Finkbeiner, B. (2022). *Formal Specifications from Natural Language*. https://doi.org/10.48550/arXiv.2206.01962

Hlaoui, Y. B., Younes, A. Ben, Ben Ayed, L. J., & Fathalli, M. (2017). From Sequence Diagrams to Event B: A Specification and Verification Approach of Flexible Workflow Applications of Cloud Services Based on Meta-model Transformation. *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, *2*, 187–192. https://doi.org/10.1109/COMPSAC.2017.135

Jamal, M., & Zafar, N. A. (2016). Formalizing structural semantics of UML 2.5 activity diagram in Z Notation. *2016 International Conference on Open Source Systems & Technologies (ICOSST)*, 66–71. https://doi.org/10.1109/ICOSST.2016.7838579

Jena, A., Swain, S., & Mohapatra, D. (2015). Model Based Test Case Generation from UML Sequence and Interaction Overview Diagrams. *Smart Innovation, Systems and Technologies*, *32*, 247–257. https://doi.org/10.1007/978-81-322-2208-8_23

Jiang, T., She, Y., & Wang, X. (2016). An Approach for Automatically Verifying Metamodels Consistency. *International Journal of Simulation Systems, Science and Technology*, *17*, 20.1-20.7. https://doi.org/10.5013/IJSSST.a.17.27.20

Karolita, D., Kanij, T., Grundy, J., McIntosh, J., & Obie, H. (2023). *Use of Personas in Requirements Engineering: A Systematic Literature Review*.

Kleppe, A., & Warmer, J. (2000). Making UML activity diagrams object-oriented. *Proceedings 33rd International Conference on Technology of Object-Oriented Languages and Systems TOOLS 33*, 288–299. https://doi.org/10.1109/TOOLS.2000.848769

Lapouchnian, A. (2005). Goal-Oriented Requirements Engineering: An Overview of the Current Research. *Requirements Engineering*, *8*(3), 32. https://doi.org/10.1007/s00766-003-0178-9

Maio, P. Di. (2021). System Level Knowledge Representation for Complexity. *2021 IEEE International Systems Conference (SysCon)*, 1–6. https://doi.org/10.1109/SysCon48628.2021.9447091

Meziani, L., Bouabana-Tebibel, T., & Bouzar-Benlabiod, L. (2018). From Petri Nets to UML Model: A New Transformation Approach. *2018 IEEE International Conference on Information Reuse and Integration (IRI)*, 503–510. https://doi.org/10.1109/IRI.2018.00080

OMG. (2011). *OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1*. http://www.omg.org/spec/UML/2.4.1

Pang, C., Pakonen, A., Buzhinsky, I., & Vyatkin, V. (2016, June). *A Study on User-Friendly Formal Specification Languages for Requirements Formalization*. https://doi.org/10.1109/INDIN.2016.7819246

Parkes, A. (2002). *Introduction to Languages, Machines and Logic*. https://doi.org/10.1007/978-1-4471-0143-7

Pérez, B., & Porres, I. (2019). Reasoning about UML/OCL class diagrams using constraint logic programming and formula. *Inf. Syst.*, *81*, 152–177. https://api.semanticscholar.org/CorpusID:69512866

Pérez-Castillo, R., Guzmán, I., & Piattini, M. (2011). Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems. *Computer Standards & Interfaces*, *33*, 519–532. https://doi.org/10.1016/j.csi.2011.02.007

Popescu, D., & Dumitrache, I. (2023). Knowledge representation and reasoning using interconnected uncertain rules for describing workflows in complex systems. *Information Fusion*, *93*. https://doi.org/10.1016/j.inffus.2023.01.007

Rabinia, A., & Ghanavati, S. (2017). *FOL-Based Approach for Improving Legal-GRL Modeling Framework: A Case for Requirements Engineering of Legal Regulations of Social Media*. 213–218. https://doi.org/10.1109/REW.2017.78

Ramadan, Q., Strüber, D., Salnitri, M., Jürjens, J., Riediger, V., & Staab, S. (2020). A semi-automated BPMN-based framework for detecting conflicts between security, data-minimization, and fairness requirements. *Software and Systems Modeling*. https://doi.org/10.1007/s10270-020-00781-x

Ries, B., Guelfi, N., & Jahic, B. (2021). *An MDE Method for Improving Deep Learning Dataset Requirements Engineering using Alloy and UML*. 41–52. https://doi.org/10.5220/0010216600410052

Rodríguez-Gil, L., García-Zubia, J., Orduña, P., Villar-Martinez, A., & López-De-Ipiña, Di. (2019). New Approach for Conversational Agent Definition by Non-Programmers: A Visual Domain-Specific Language. *IEEE Access*, *7*, 5262–5276. https://doi.org/10.1109/ACCESS.2018.2883500

Ross, D. T., & Schoman, K. E. (1977). Structured Analysis for Requirements Definition. *IEEE Transactions on Software Engineering*, *SE-3*(1), 6–15. https://doi.org/10.1109/TSE.1977.229899

Sabri, M. (2015). REQUIREMENTS ENGINEERING DOMAIN KNOWLEDGE IN INFORMATION TECHNOLOGY. *SSRN Electronic Journal*, *3*, 55–62.

Sammi, R., Rubab, I., & Qureshi, M. A. (2010). *Formal specification languages for real-time systems*. *3*, 1642–1647. https://doi.org/10.1109/ITSIM.2010.5561643

Sangiorgi, D., & Walker, D. (2001). *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press.

Saratha, P., Uma, G. V, & Santhosh, B. (2017). Formal Specification for Online Food Ordering System Using Z Language. *2017 Second International Conference on Recent Trends and Challenges in Computational Models (ICRTCCM)*, 343–348. https://doi.org/10.1109/ICRTCCM.2017.59

Sengupta, S., & Bhattacharya, S. (2006). Formalization of UML use case diagram-a Z notation based approach. *2006 International Conference on Computing & Informatics*, 1–6. https://doi.org/10.1109/ICOCI.2006.5276507

Sharaff, A., & Rath, S. K. (2020). Formalization of UML Class Diagram Using Colored Petri Nets. *2020 First International Conference on Power, Control and Computing Technologies (ICPC2T)*, 311–315. https://doi.org/10.1109/ICPC2T48082.2020.9071490

Siddique, A. B., Qadri, S., Hussain, S., Ahmad, S., Maqbool, I., Karim, A., & Khan, A. K. (2014, June). *INTEGRATION OF REQUIREMENT ENGINEERING WITH UML IN SOFTWARE ENGINEERING PRACTICES*.

Son, H. S., & Kim, R. Y. C. (2017). XCodeParser based on Abstract Syntax Tree Metamodel (ASTM) for SW visualization. *Information (Japan)*, *20*, 963–968.

Sonbol, R., Rebdawi, G., & Ghneim, N. (2020, June). *Towards a Semantic Representation for Functional Software Requirements*. https://doi.org/10.1109/AIRE51212.2020.00007

Spivey, J. M. (1989). *The Z Notation: A Reference Manual*. Prentice-Hall, Inc.

Sun, W., Zhang, H., Feng, C., & Fu, Y. (2016). A Method Based on Meta-model for the Translation from UML into Event-B. *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 271–277. https://doi.org/10.1109/QRS-C.2016.41

Tariq, O., Sang, J., Gulzar, K., & Xiang, H. (2017). Automated analysis of UML activity diagram using CPNs. *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, 134–138. https://doi.org/10.1109/ICSESS.2017.8342881

Tichelaar, S., Ducasse, S., & Demeyer, S. (2000). *FAMIX: Exchange Experiences with CDIF and XMI*.

Torlak Emina and Jackson, D. (2007). Kodkod: A Relational Model Finder. In M. Grumberg Orna and Huth (Ed.), *Tools and Algorithms for the Construction and Analysis of Systems* (pp. 632–647). Springer Berlin Heidelberg.

Varzi, A. (2022). Complementary Logics for Classical Propositional Languages. *Kriterion (Austria)*, *1*. https://doi.org/10.1515/krt-1992-010406

Wang, M., & Zeng, Y. (2009). Asking the right questions to elicit product requirements. *International Journal of Computer Integrated Manufacturing*, *22*(4), 283–298. https://doi.org/10.1080/09511920802232902

Wieringa, R. J., & Wieringa, R. J. (2014). Single-Case Mechanism Experiments. In *Design Science Methodology for Information Systems and Software Engineering* (pp. 247–267). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-43839-8_18

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). Experimentation in software engineering. In *Experimentation in Software Engineering* (Vol. 9783642290). https://doi.org/10.1007/978-3-642-29044-2

Xu, Y. (2011). The formal semantics of UML activity diagram based on Process Algebra. *2011 International Conference on Computer Science and Service System (CSSS)*, 2729–2732. https://doi.org/10.1109/CSSS.2011.5974744

Zapata, C. M. (2012). The UNC-Method Revisited: Elements of the New Approach Eliciting Software Requirements in a Complete, Consistent, and Correct Way. *LAP LAMBERT Academic Publishing GmbH & Co*, *5*, 2013. https://www.lap-publishing.com/catalog/details/store/de/book/978-3-8484-0759-0/the-unc-method-revisited:-elements-of-the-new-approach?search=organic products

Zapata, C. M., & Arango, F. (2009). The UNC-method: a problem-based software development method UNC-Method : un método de desarrollo de software basado en problemas. *Revista Ingeniería e Investigación*, *29*(1), 69–75.

Zimmermann, A. (2008). *Colored Petri Nets*. https://doi.org/10.1007/978-3-540-74173-2_6