A Domain Specific Language for the Models and Data (MODA) Framework in Model-Driven Engineering

Mercy Asamoah

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

McGill University
Montreal, Quebec, Canada

August 2023

Abstract

Over the years, software engineering technology has improved and advanced massively, and as such, it has greatly impacted the development cycle of systems. Some systems can now even be developed entirely with little human involvement with the help of model-driven engineering techniques. Recently, systems have become very data-centric; hence there is the need to understand if these systems can be built in a model-driven way and how data-centric approaches fit into this picture. The Models and Data (MODA) framework is a conceptual reference framework that clarifies the various roles that models and data play in software development and operation of socio-technical systems. Using this framework, we are able to view the various parts of the system that serve as models and different kinds of data, analyze the role they each play, and finally understand how they work together to make the system function. The authors of the MODA framework outline the architecture and vast applicability of the framework but there is currently no tool support to help practitioners build MODA models. Also, the broad applicability of the framework is claimed but only preliminary evidence is given. This thesis introduces a domain-specific language and tool support for the MODA framework. As there is currently no existing metamodel for the framework, this thesis contributes to the existing framework with a well-defined metamodel that accurately depicts the framework's elements. To further validate this metamodel, a tool was built using the Xtext and Sirius language engineering environments. This tool enables modelers to create their own MODA models to represent the socio-technical system they would want to analyze. Furthermore, an evaluation of the proof-of-concept tool and the MODA framework with the help of an education-based exploratory study gives further evidence of the broad applicability of the MODA framework by representing key courses in the Applied Artificial Intelligence minor and Software Engineering

minor to further demonstrate the framework and tool's applicability to a broad range of concepts, technologies, and processes.

Abrégé

Au fil des ans, la technologie du génie logiciel s'est améliorée et a progressé considérablement, et en tant que telle, elle a eu un grand impact sur le cycle de développement des systèmes. Certains systèmes peuvent même maintenant être développés entièrement avec peu d'intervention humaine à l'aide de techniques d'ingénierie basées sur des modèles. Récemment, les systèmes sont devenus très centrés sur les données; il est donc nécessaire de savoir si ces systèmes peuvent être construits selon une manière basée sur les modèles et comment les approches centrées sur les données s'intègrent dans ce projet. "The Models and Data" (MODA), le cadre Modèles et donnés est un cadre conceptuel de référence qui clarifie les différents rôles que jouent les modèles et les données dans le développement logiciel et l'exploitation de systèmes sociotechniques. À l'aide de ce cadre, nous sommes en mesure de visualiser les différentes parties du système qui servent comme modèles et différents types de données, d'analyser le rôle que jouent chacune et, enfin, de comprendre comment elles fonctionnent ensemble pour faire fonctionner le système. Les auteurs du cadre MODA décrivent l'architecture et la vaste applicabilité du cadre, mais il n'existe actuellement aucun outil de soutien pour aider les praticiens à construire des modèles MODA. De plus, l'applicabilité large du cadre est revendiquée, mais seules des preuves préliminaires sont fournies. Cette thèse présente un langage spécifique au domaine et un outil de soutien pour le cadre MODA. Comme il n'existe actuellement aucun métamodèle pour le cadre, cette thèse en contribue un métamodèle bien défini qui décrit en détail les éléments du cadre. Pour valider davantage ce métamodèle, un outil a été construit à l'aide des environnements d'ingénierie linguistique Xtext et Sirius. Cet outil permet aux modélisateurs de créer leurs propres modèles MODA pour représenter le système sociotechnique qu'ils souhaitent analyser. En outre, une évaluation de l'outil de preuve de concept et du cadre MODA à l'aide d'une

étude exploratoire basée sur l'éducation fournit une preuve supplémentaire de l'applicabilité large du cadre MODA en représentant des cours clés de la mineure en génie logiciel et de la mineure en intelligence artificielle appliquée, afin de démontrer davantage l'applicabilité du cadre et de l'outil à un large éventail de concepts, technologies et processus.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Gunter Mussbacher for his invaluable guidance, support, and mentorship throughout my research journey. I learned so much from him over the past few years. His expertise and encouragement have been instrumental in shaping this thesis.

Secondly, I would like to thank Professor Benoit Combemale of the University of Rennes for his valuable feedback and for taking the time to review my models. His insights and suggestions have significantly improved the quality of my research.

To Professor Jörg Kienzle and the entire research group, I am grateful for their collaborative efforts and for engaging in insightful discussions about various courses with me.

I would also like to extend a thank you to my friend Younes for his assistance. His contributions have greatly enriched my work, and I am grateful for his unwavering support.

I would like to express my heartfelt gratitude to my friends, for their encouragement and assistance. To my family, I am eternally grateful for their unconditional love, encouragement, and belief in my abilities. Their unwavering support has been a constant source of inspiration throughout this endeavor.

Finally, I am indebted to God for His grace and provision throughout my MS program and for guiding me to this point in my journey.

Table of Contents

Appendices

List of Tables

List of Figures

List of Programs

List of Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
AQL	Acceleo Query Language
ATDD	Acceptance Test Driven Development
ATL	ATLAS Transformation Language
BDD	Behavior-driven Development
CASE	Computer-Aided Software Engineering
CI	Continuous Integration
CNN	Convolutional Neural Network
DL	Deep Learning
DSL	Domain-specific Language
DT	Digital Twin
EMF	Eclipse Modeling Framework
GAN	Generative Adversarial Network
GRL	Goal-oriented Requirement Language
IDE	Integrated Development Environment
JVM	Java Virtual Machine
L-MODA	Languages, Models, and Data
LSTM	Long Short-Term Memory Network
MDE	Model-driven Engineering
ML	Machine Learning
MLP	Multilayer Perceptron
MODA	Models and Data
MOF	Meta-Object Facility

MP-MODA	Multi-Plane Models and Data
NLP	Natural Language Processing
NN	Neural Network
OCL	Object Constraint Language
OMG	Object Management Group
ORM	Object Relational Mapping
QA	Quality Assurance
RNN	Recurrent Neural Network
SAL	Self-Adaptable Language
SCADA	Supervisory Control and Data Acquisition
SDK	Software Development Kit
SDLC	Software Development Life Cycle
SE	Software Engineering
SLE	Software Language Engineering
SPG	Smart Power Grid
STS	Socio-technical System
SVM	Support Vector Machines
TDD	Test-driven Development
UCM	Use Case Map
UML	Unified Modeling Language
UPTN	Urban Public Transportation Network
URN	User Requirements Notation
USE	UML-based Specification Environment
VAE	Variational Autoencoder
VM	Virtual Machine
VSM	Viewpoint Specification Model
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XP	Extreme Programming

1

Introduction

In this introductory chapter, we provide the motivation for our research, the problem definition, the description of our approach and contributions, as well as the thesis outline for the remaining chapters.

1.1 Problem Definition and Motivation

Over the years, software engineering technology has improved and advanced massively, and as such, it has greatly impacted the development cycle of systems. Some systems can now even be developed entirely with little human involvement, i.e., more and more aspects of software engineering are now automated. Model-driven Engineering (MDE) [27] [118] is a discipline of software engineering that focuses on using models and model transformations to increase software development automation and level of abstraction. The practices in MDE have been proven to improve the effectiveness and

efficiency of the development of software [39]. On the other hand, systems now require large volumes of data to run efficiently. The recent success of Artificial Intelligence (AI) and, in particular, Machine Learning (ML) [12] highlights the importance of data in the development, maintenance, evolution, and execution of systems built with MDE techniques. In sectors such as transportation, energy, and healthcare, numerous systems are considered socio-technical, given the human, organizational, and social factors considered during the system life cycle [31]. Socio-technical systems can have wide-reaching consequences for the lives of their users.

During software development, questions may arise such as "what problem is the system going to solve?", "what type of data can be adopted to achieve this?", "what kind of models can be designed or implemented to build this system?", and "what processes can be assumed to achieve this?", among others. Answers to these questions are crucial in getting a system to function. The Models and Data (MODA) framework is a conceptual reference framework that clarifies the various roles models and data play in software development and operation of socio-technical systems. In a previous publication [31], the authors introduce the MODA framework and discuss and anticipate that the framework may be used:

- as a teaching tool to shed light on the roles of data sources, models, and associated actions across a wide range of life cycles of systems;
- to sort and analyze complicated engineering technologies and processes that will inform important engineering decisions involving significant systems (e.g., smart grid systems); and
- to better integrate diverse types of data sources and models by situating existing research methodologies or initiating new research programs.

The authors outline the architecture and vast applicability of the MODA framework [31] but there is currently no tool support to help practitioners build MODA models. Furthermore, the broad applicability of the framework is discussed but only preliminary evidence is given. The framework is claimed to apply to every software engineering tool, technology, and process. While some examples are given to support this point, a more comprehensive and tangible way of checking its applicability is missing.

1.2 Thesis Methodology and Contributions

This thesis introduces (i) a Domain-specific Language (DSL) as well as tool support (i.e., editor) for the MODA framework and (ii) an evaluation of the proof-of-concept tool and the MODA framework with the help of an education-based analysis. One can use this DSL to illustrate the various models in a chosen domain and how these models interact with the data presented for the selected socio-technical system. The exploratory studies in this thesis focus on understanding how easily we can break down systems into their respective components and represent them with the individual aspects of the MODA framework. Using this framework, we will be able to view the various parts of the system that serve as models and data (i.e., input, output, measured or external sources of data) to the system, analyze the role they each play, and finally understand how they work together to make the system function effectively. This study further validates the framework and tool's applicability to various concepts, technologies, and processes. The contributions of this thesis to this area of study are as follows:

- As there is currently no existing metamodel for the framework, we implement a metamodel to accurately define the elements of the MODA framework.

- Expanding on prior knowledge of the MODA framework, we build a proof-of-concept editor that supports this framework and graphically visualizes how models and data work together in a selected socio-technical system. This editor is initially tested by specifying all existing models from the original MODA publication [31].

- To further validate the applicability of the MODA framework, there has to be a criterion to help us define the scope of tools, technologies, or processes to be considered during the exploratory study. We choose an education-based analysis, as many tools, technologies, and processes are taught in university courses. To further narrow the scope to a manageable size, we look at key courses offered in two minor programs at McGill University. We model the technologies, tools, and techniques used in select courses offered in the Software Engineering Minor degree and Applied AI Minor degree at the Department of Electrical and Computer Engineering. Finally, we validate if MODA will allow us to think and reason about these courses with the help of our prototype editor.

- Observations and analysis are carried out to try and identify areas that the MODA framework does not capture and how effective the framework is in modeling many situations. While the MODA framework generally allows the courses to be captured well, the analysis reveals aspects of the framework that may need additional investigation or expansion beyond the existing definitions provided by the original MODA framework [31].

1.3 Thesis Overview

This thesis is organized into seven chapters as follows:

- **Chapter 1: Introduction**

  This chapter covers the problem context, problem definition, thesis methodology and contribution, and the thesis overview. The author contributed to the full chapter.

- **Chapter 2: Background**

  This chapter provides the background information needed to understand this thesis, i.e., the Eclipse Modeling Framework (EMF) [45], Ecore [44], Sirius [48], and an overview of the Models and Data (MODA) framework, including its types of models and data, and their various roles. The author contributed to the full chapter.

- **Chapter 3: MODA Metamodel**

  This chapter presents the various variations of the metamodels implemented for the MODA framework. We discuss why each variation was considered and how that led to the final metamodel. We also discuss how the final metamodel is validated to ensure accuracy. The author contributed to the full chapter.

- **Chapter 4: Sirius Tool Implementation and Verification**

  This chapter presents the proof-of-concept tool built to support the MODA framework. We delve into how we implement and specify the tool with Sirius, its setup, application, and validation. The author contributed to the full chapter.

- **Chapter 5: MODA Education Application**

  This chapter presents the courses selected to analyze the framework in an exploratory study

and discusses the various views and perspectives obtained from the analysis. The author contributed to the full chapter.

- **Chapter 6: Related Work**

  This chapter presents a review of relevant literature. The author contributed to the full chapter.

- **Chapter 7: Conclusion**

  This chapter summarizes the contributions of the thesis and discusses future work. The author contributed to the full chapter.

- **Appendix A: Xtext Textual Language Definition for the MODA Metamodel**

  In this appendix, the Xtext textual language used to implement the various variations of the MODA metamodel is presented. The author contributed to the full appendix.

- **Appendix B: Java Code for Metamodel Validation**

  The Java code used for validating the framework is presented in this appendix. The author contributed to the full appendix.

2

Background

This chapter provides the background knowledge required to understand the remaining parts of the thesis. First, we discuss models and data and their applicability. We then discuss the MODA framework and its components. Finally, we touch on Xtext, Ecore, and Sirius, i.e., the technologies used to implement the tool support for the MODA framework.

2.1 Models and Data

Modeling is used in many societies and disciplines because it provides a better understanding and rationale of a system. A model is a representation of reality that is abstracted for a specific purpose. A model plays a [31]:

- a descriptive role if it documents a present or previous aspect of the system under study (which could be a software-intensive system or a natural system), making it easier to comprehend

and analyze;

- a prescriptive role if it is a description of the system to be built, driving the constructive process, including runtime evolution in the case of self-adaptive systems (i.e., models@runtime);
- a predictive role if it is being used to forecast facts that cannot or will not be measured (which creates new knowledge and allows decision-making and trade-off analyses to be performed).

As cited by Combemale et al. [31], Engineering, Scientific, and Machine Learning models are some examples of types of models. An engineering model is meant to drive the creation of the future system, maybe with some automation [80]. Along with clearly specified notations for their models, they frequently use systematic procedures and techniques. They can also be used to create software-based systems, physical systems for a system to be developed for a certain objective that complies with physical laws, or both e.g., cyber-physical systems. A scientific model is a depiction of a portion of a real-world phenomenon [53]. Based on accepted scientific information establishing a theory, it is used to define, measure, illustrate, or replicate the phenomenon in order to explain and study it. A machine learning model is derived from sample data, also known as training data, by automated learning algorithms to generate predictions or decisions without being specifically trained to do so [149].

Each model type can take on multiple roles. A scientific model is descriptive at first, but its primary goal is to become predictive, allowing for what-if scenarios [28]. It becomes prescriptive when embedded in a socio-technical system. An engineering model is often descriptive at first (e.g., a domain model detailing essential ideas and relationships), then transformed into a prescriptive model at the design stage. However, once the system is created according to the specifications, the model returns to being descriptive as a kind of documentation [61]. Engineering models can also be predictive models; an architecture model, for example, might be used to forecast the performance of a particular setup. A machine learning model is typically employed in a predictive role, with the goal of inferring new information from hypothetical input data. It could be descriptive about a present or previous connection, or prescriptive if the findings are utilized to influence decisions.

Knowledge and data are required as input for any of the following types of models. Each model type has a different amount of necessary knowledge or the necessity of having the required data available to develop the models. For example, to choose the appropriate ML technique, determine

the ML meta-parameters, choose the input and output variables, and then derive a customized model from the data, there is a need for problem-specific knowledge in ML models. In engineering models, domain knowledge and, in certain cases, data are primarily used to improve or tune the models.

Along with the relative relevance of knowledge and data in the model-building process, the sequence in which models and data are considered varies by model type. External data (e.g., expert or domain knowledge expressed in requirements or constraints) or measured data are used to build descriptive engineering models (e.g., exploitation data from previous systems). After then, engineering models are utilized to specify how the future system would be constructed. ML typically starts with input or output system data or measurable data for training and iteratively (e.g., through feedback loops) revises the model to meet the problem at hand, with the generated models being the major output of the process. External data (e.g., real-world observations) plays a key part in scientific models, whereas off-the-shelf models aim to describe actual events and are thus updated and improved on a regular basis.

2.2 The MODA Framework

To ensure comprehensive support throughout the life cycle of present and future complex socio-technical systems, particularly those heavily reliant on software, it is imperative to adopt a synergistic approach that combines various models and well-established methodologies. By leveraging the collective advantages of these models, diverse goals can be effectively met, thereby ensuring the overall success of the systems. Combemale et al. [31] present a conceptual Models and Data (MODA) framework to assist this integration through engineering processes, which explicitly ties the varied roles of the model types to four forms of data: input data, output data, measured data, and external data. The MODA framework offers valuable insights into the integration of diverse model types with distinct functions, including data sources and associated actions. This framework provides a comprehensive and generalized perspective on typical software development processes, technologies, and systems. In addition to the data being delivered to and generated by the running system, it also encompasses the collection of data related to the software and its environment, such as performance data. To effectively handle the evolving data, descriptive, predictive, and pre-

scriptive models are employed to process this vast amount of information and facilitate necessary modifications to the system.

The MODA framework is depicted in Figure 2.1. The running software is indicated in purple, the socio-technical system in blue, the different model roles in green, and several types of data are displayed in yellow. The arrows reflect actions connected to the models and data. [1]

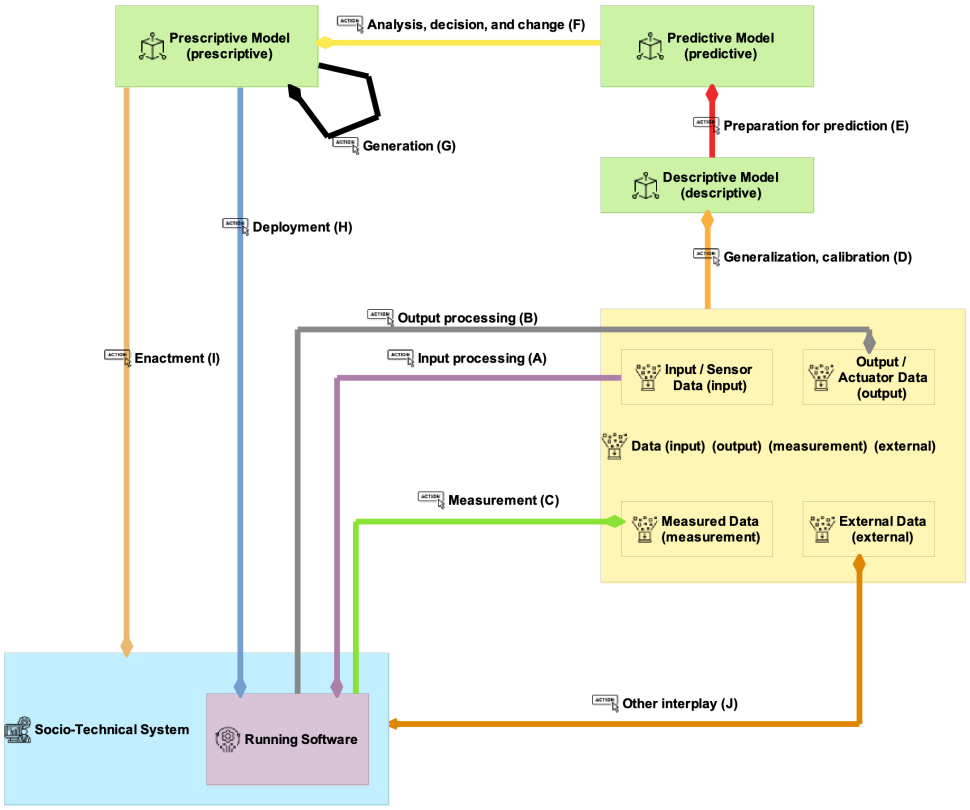


Figure 2.1: The MODA Framework

- The running software processes input data and generates output, as shown by the arrows A and B, labeled Input processing and Output processing, respectively. Input processing refers to the handling and interpretation of incoming data received by the system. This data serves as the input for the software, providing the necessary information for further processing. On the other hand, output processing involves the generation of results or output by the software. This output represents the outcome or response of the system's processing based on the input

[1] The icons used to represent the various elements of the MODA framework are referenced as follows: action icon made by Freepik from "www.flaticon.com", socio-technical system icon made by Eucalyp from "www.flaticon.com", running software icon made by Parzival' 1997 from "www.flaticon.com", model icon made by Pixel perfect from "www.flaticon.com", and data icon made by Parzival' 1997 from "www.flaticon.com"

data it received. It could be in the form of computations, calculations, transformed data, or any other relevant information produced by the software.

- The C arrow labeled Measurement represents information and data gathered from monitoring the system. The information gathered might be filtered or aggregated in real-time, as well as saved for later use.

- External data, the final type of data, is any information that falls outside the scope of the software in the present version of the system. This external data can originate from various sources or systems that interact with the software, such as external databases, external sensors, third-party APIs, or other external interfaces. In the future, the external data can be used to enhance the functionality of the system, provide additional insights, or support specific operations. However, it is important to note that the system does not directly control or manage this external data (J arrow) currently.

- The D arrow, labeled Generalization and Calibration, represents two aspects in the context of the described system. Generalization refers to the strategies employed to derive a descriptive model from diverse types of data. These techniques include conceptual generalization methods such as abstraction and synthesis, as well as statistical approaches such as regression, differential equation analysis, data mining, and advanced machine learning techniques. Generalization can occur in real-time or offline. Calibration, on the other hand, involves the techniques utilized to configure and adjust the data for the purpose of generating a descriptive model.

- The E arrow, Preparation for prediction, focuses on preprocessing techniques that combine data with descriptive models to create predictive models. These models are employed to generate predictions, commonly known as predictive modeling. This process often requires additional steps such as interpolation and extrapolation techniques, statistical methods (e.g., regression), and preparatory measures for simulating and training ML models like neural networks.

- The F arrow, labeled as Analysis, decision, and change, represents activities related to decision support and the implementation of modifications to the prescriptive model based on

those decisions. These activities often involve conducting what-if analysis and making adjustments accordingly. What-if scenarios can be carried out manually or automatically, and the implementation of decisions can take various forms. In self-adaptive systems, a decision may require reconfiguration, which can be achieved by modifying the prescriptive architecture model (e.g., through model transformations) or updating configuration files. In the case of a software product line, anticipated adjustments can be made by selecting characteristics that define previously designed alternatives and then modifying the prescriptive model, for instance, through techniques such as model merging.

- The G arrow, represented as Generation, illustrates the typical software development operations involved in constructing lower-level prescriptive models based on higher-level prescriptive models. This process involves translating more abstract design models or requirements models into executable code. Various approaches such as model transformations, model instantiation, and compilers are employed to facilitate this generation process.

- The H arrow, Deployment, pertains to the activities associated with deploying, executing, or interpreting low-level, executable models such as code. This phase involves putting into operation the implemented changes or improvements.

- The I arrow, labeled Enactment, represents activities that are executed or enforced within a socio-technical system utilizing prescriptive models that incorporate human or social components. These components can include laws, policies, standards, or other measures that guide behavior and actions.

Consider the development of a Smart Power Grid (SPG) application, which is an automated system specifically designed to monitor and regulate the distribution grid. This system is capable of establishing automatic communication between the power provider and the load consumers in order to swiftly restore electricity in the event of a power outage caused by faults or disruptions in the grid. Figure 2.2 shows an example of the MODA framework in the context of an SPG. The running software in this situation is the smart grid system. This consists of digital and other cutting-edge technologies, sensors, and applications that manage and monitor the transportation of power from all generation sources to satisfy the various electricity needs of end customers. The

socio-technical system in context encompasses a broader scope that extends beyond the running software. It incorporates various stakeholders involved in the system, such as the users (consumers of energy), energy companies (providers of energy), regulators (government bodies), advertising companies, and more.

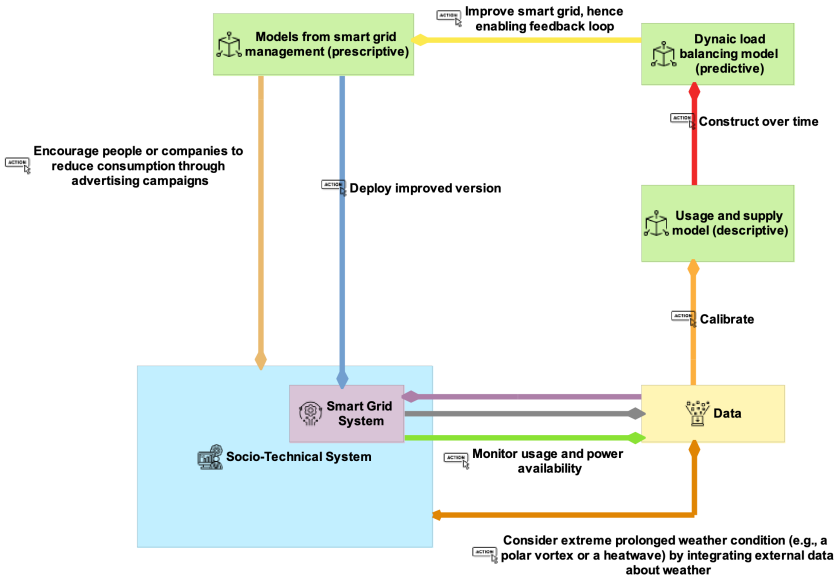


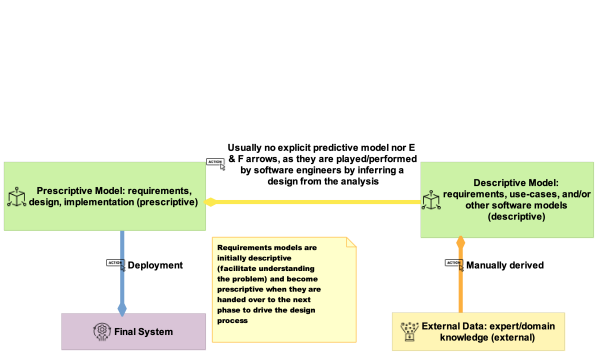
Figure 2.2: Smart Power Grid Models Diagram

These entities play significant roles within the system, contributing to its functioning and influencing its dynamics. Information gathered from sensors and household meters are examples of input data for SPG, and the output data includes information, metrics, and data on power availability and power to be transmitted to an end user. Measured data will be the amount of power an end user consumes. In the case of an external source of data, extreme prolonged weather conditions could affect the distribution and transmission of power to an end user. Hence, weather information when necessitated could be utilized in the future to improve power transmission and distribution (i.e., action type J). For generalization and calibration (i.e., action type D), by correlating received data on electricity consumption with transmitted power, the power company can determine if the end user is consuming more power than is being transmitted to them. Scientific models, such as those modeling the flow of distribution and transmission lines, can be developed to estimate the average power generation required based on received consumption data. Machine learning techniques can also be employed to analyze electricity consumption patterns. In preparation for prediction

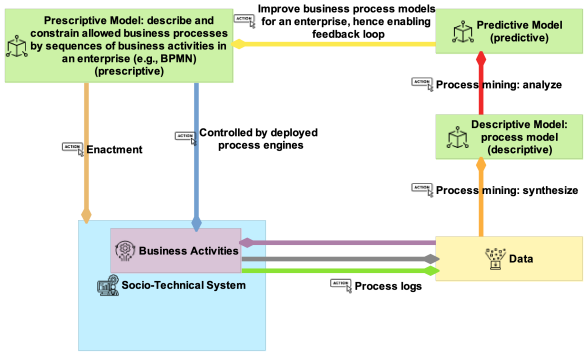
(i.e., action type E), in the context of electricity consumption predictions, the SPG system can utilize historical consumption data, weather conditions, and other relevant factors to forecast the future power consumption of end users. These predictions help the power company anticipate and prevent power outages caused by imbalances in supply and demand. Additionally, by leveraging neural networks, hidden behavioral patterns can be uncovered, offering insights that can be used to prevent potential issues or incidents in the future. Prescriptive models would include safety regulations, simulations, SCADA (Supervisory Control and Data Acquisition) systems, communication algorithms, and other relevant components. When considering analysis, decision, and change (i.e., action type F), a forecast may indicate an upcoming storm in a particular month that could potentially damage transmission lines. In response, a decision may be made to proactively disconnect power through those lines and notify end users of the anticipated power outage, advising them to switch to their renewable energy sources, such as solar power systems.

Action type G (i.e., Generation) is not evident in the smart grid example as depicted in the MODA paper [31], but the generation process could involve transforming higher-level prescriptive models, such as architectural designs or requirements specifications, into lower-level models that can be executed as code. This could include generating code for components responsible for monitoring energy consumption, optimizing power distribution, or managing renewable energy sources. Once the application has been developed, it is deployed (i.e., action type H) to replace the previous version, making it operational and ready for use. Regarding action type I, initiatives may be undertaken to encourage individuals or companies to reduce their energy consumption. This can be achieved through various means, such as advertising campaigns, which promote awareness and provide incentives for adopting energy-efficient practices. These efforts aim to influence human behavior and drive positive changes in energy consumption patterns.

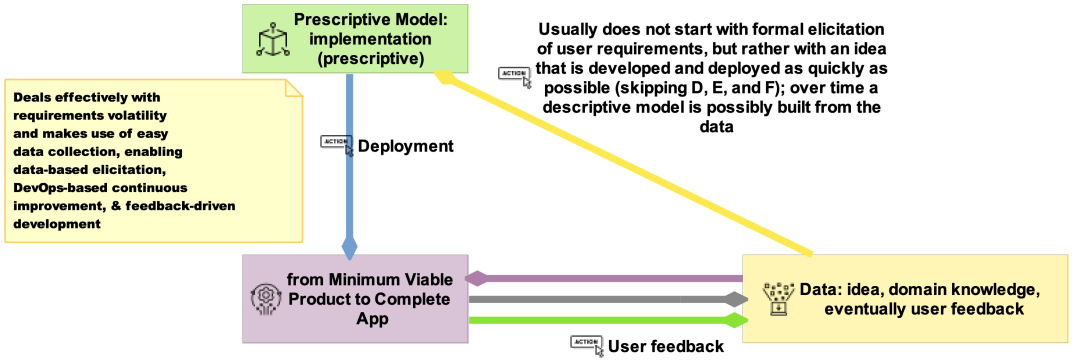
In addition to socio-technical systems like SPG, MODA generalizes to other systems, technologies, and processes that are currently in practice. Figure 2.3 illustrates the applicability of MODA that is claimed in the original paper [31]. Figure 2.3(a) illustrates a software development methodology (i.e., Waterfall Process Model for Software Development). Figure 2.3(b) (i.e., Business Process Modeling and Mining) illustrates ways for modeling and mining business processes. MODA can also be used in system development as shown in Figure 2.3(c), which illustrates the development of Simple Mobile Apps.



(a) Waterfall Process Model



(b) Business Process Modeling and Mining



(c) Simple Mobile Application

Figure 2.3: Examples of the MODA Framework

Based on the examples provided, it is evident that the MODA framework plays a crucial role in facilitating a comprehensive understanding of how different model roles and data sources are integrated. The framework allows for a systematic approach to analyzing and incorporating various models, such as descriptive, predictive, and prescriptive models, within the context of a given system. Furthermore, it helps to identify the relationships and dependencies between these models and how they interact with different data sources. Using the MODA framework, one can gain insights into the overall software development process, from initial data analysis and model generalization to prediction, analysis, decision-making, deployment, and enactment. In addition, this framework provides a structured perspective on the integration and utilization of models to address complex challenges in software engineering, ultimately improving our understanding of the interplay between different model roles and data sources.

2.3 Xtext

Xtext [50] is a framework for creating programming languages and domain-specific textual languages. Utilizing a powerful grammar language, Xtext allows a language engineer to define a language. As a result, the language engineer receives a complete infrastructure that includes a parser, linker, type-checker, compiler, and editing support for (i) Eclipse, (ii) any editor that adheres to the Language Server Protocol, and (iii) the preferred web browser [41]. An Xtext grammar language example is shown in Program 2.1 [51].

Program 2.1: Xtext Grammar Language

```
1  grammar org.xtext.example.mydsl.MyDsl with org.eclipse.xtext.common.Terminals
2  generate myDsl "http://www.xtext.org/example/mydsl/MyDsl"
3
4  DomainModel:
5      (elements+=Type)*;
6
7  Type:
8      DataType | Entity;
9
10 DataType:
11     'dataType' name=ID;
12
13 Entity:
14     'entity' name=ID ('extends' superType=[Entity])? '{'
15         (features+=Feature)*
16     '}';
17
18 Feature:
19     (many?='many')? name=ID ':' type=[Type];
```

The start rule is always the first rule in the grammar.

DomainModel: (elements+=Type)*;

It states that a *DomainModel* has any number (*) of *Types* which are added (+=) to a feature called *elements*. The *Type* rule delegates to the rule *DataType* or (|) the rule *Entity*.

Type: DataType | Entity;

The rule *DataType* begins with the keyword *'datatype'*, followed by an identifier which is parsed by a rule called *ID*. The super grammar *org.eclipse.xtext.common.Terminals* defines the rule ID and parses a single word (i.e., an identifier).

DataType: 'datatype' name=ID;

The definition of the keyword *entity* is presented first in the rule *Entity*, then a *name*.

Entity: 'entity' name=ID ('extends'
superType=[Entity])? '{'(features+=Feature)* '}';

The clause with the '*extends*' keyword follows, which is optional and parenthesized (?). The parser rule *Entity* is not invoked in this case because the feature named *superType* is a cross-reference, and only one identifier (the ID rule of the referenced Entity rule) is parsed instead. During the linking process, the actual *Entity* to assign to the *superType* reference is determined based on the parsed ID rule. The last part of the *Entity* rule specifies features between curly braces, which applies the next rule.

Feature: (many?='many')? name=ID ':' type=[Type];

The *boolean* type of the feature *many* is implied by the assignment operator (?=). The presence of the keyword *'many'* in a model indicates that the value of the boolean type is true. Then, a *name* is defined and a cross-reference to a *Type*.

Once the grammar has been defined, the code generator is run to produce the various language parts. This creates an Ecore and genmodel file, which are then used to initialize an Ecore diagram, i.e., the metamodel, for the domain described with the Xtext language. The Eclipse IDE [46] integration can now be tested by running the generated Eclipse plugin in a new runtime window. The generated Eclipse plugin is an editor for models that conform to the Xtext grammar.

Using a datatype *String*, Program 2.2 shows a textual model that conforms to the language definition in Program 2.1.

Program 2.2: Xtext Textual Model

```
1   dataType String
2
3   entity Author {
4       name: String
5   }
6
7   entity Post {
8       title: String
9       content: String
10  }
11
12  entity Blog extends Post {
13      author: Author
14      many comments: Comment
15  }
16
17  entity Comment {
18      content: String
19  }
```

As seen in the grammar in Program 2.1, all the keywords are found in single quotation marks (i.e., ‘ ’) and are highlighted in purple. These keywords are used when defining the respective classes. For example, when defining a dataType it must begin with the keyword *dataType*, and an entity must begin with the keyword *entity*. Using a dataType String as seen in Program 2.2, four entities are defined in the textual model example. The entity *author* has a name. A *post* has a title and some content. A blog extends a post (i.e., a blog is a post). The entity *blog* has the same attributes as the entity *post*, has an *author*, and also has *many* comments. Author and comments reference entities author and comment, respectively (i.e., an *entity* is composed of *features* which can be *many*, as seen in the rule on lines 13 - 19 in Program 2.1). A *comment* also has some content.

2.4 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is a collection of Eclipse plug-ins that can be used to create code or other output based on the modeling of a data model [79]. EMF distinguishes the actual model and the meta-model. Since a model is a specific instance of this metamodel, the metamodel specifies the structure of the model. The developer can specify the metamodel using various tools, such as XML Metadata Interchange (XMI) [158], Java annotations, Unified Modeling Language (UML) [158], or an Extensible Markup Language (XML) [159] schema [79]. EMF generates Java code from a metamodel specified with Ecore which is part of the core EMF, as is runtime support for the models, which includes change notification, persistence support with XMI serialization by default, and a very effective reflective Application Programming Interface (API) [143] for handling EMF objects in general. Accessors for the meta-objects are included to represent each class, each feature of each class, each interface, each operation of each class or interface, each enum, each literal of each enum, and each data type [44].

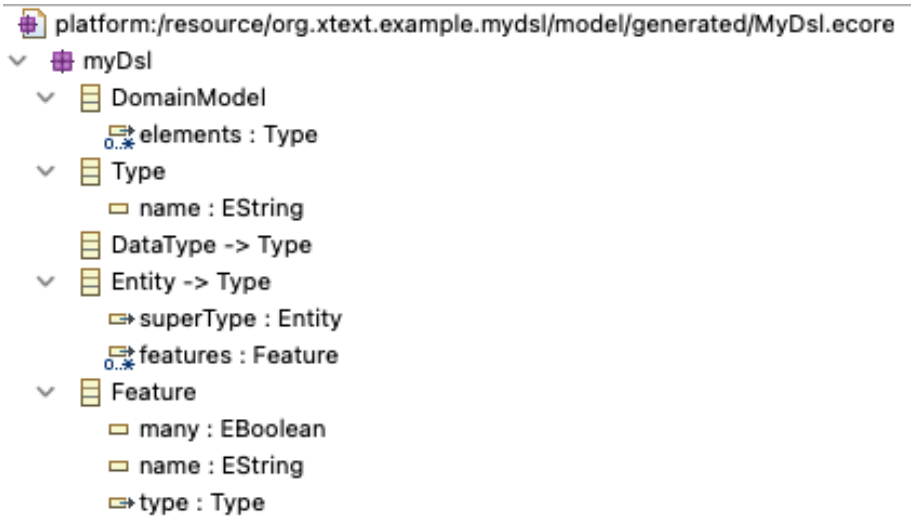


Figure 2.4: Ecore Representation of Xtext Language

The EMF metamodel consists of two parts: the Ecore and the genmodel description files. The details of the declared classes are contained in the Ecore file. Using the same example specified with the Xtext language, its corresponding Ecore file is shown in Figure 2.4. Initializing the Ecore diagram generates a graphical representation of the metamodel as shown in Figure 2.5. The genmodel file includes extra data for code generation, such as file and path information.

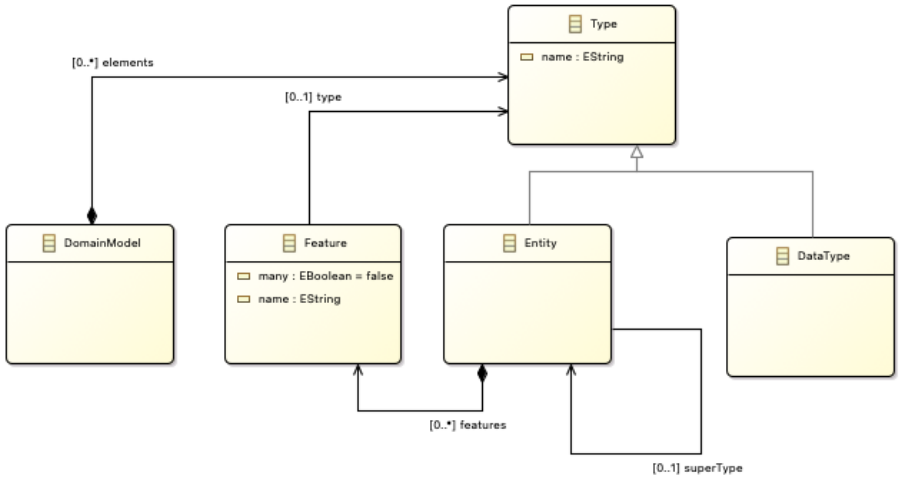


Figure 2.5: Metamodel Representation of Xtext Language

Property	Value
All	
Bundle Manifest	true
Code Style	
Compliance Level	8.0
Copyright Fields	false
Copyright Text	generated by Xtext 2.29.0
Eclipse Platform Version	2022-12 - 4.26
Language	
Model Documentation	
Model Name	MyDsl
Non-NLS Markers	false
OSGi Compatible	false
Runtime Compatibility	false
Runtime Jar	false
Runtime Platform	IDE
Runtime Version	2.20
Edit	
Editor	
Model	
Model Class Defaults	
Model Feature Defaults	
Templates & Merge	
Tests	

Figure 2.6: Genmodel Representation of Xtext Language

The control parameters for the code generation process are also contained in the genmodel file. This is shown in Figure 2.6. Generating all code from the genmodel creates the EMF model implementation in Java.

An Xtext grammar is mapped to class diagram elements in the metamodel as follows:

- rule → class
- = → multiplicity, zero or one
- += → multiplicity, zero to many
- [*Type*] → association with Type
- Type → composition with Type
- | → generalization
- ?= → boolean attribute
- =PrimitiveType → attribute of PrimitiveType (non-boolean) (e.g., name=ID)

Relating the Xtext grammar in Program 2.1 to the metamodel in Figure 2.5, a *DomainModel* has a composition (i.e., no square brackets) of zero to many elements (i.e., the operator +=) with the class *Type*. The superclass *Type* consists of two subclasses, *DataType* and *Entity* (i.e., operator |). A *DataType* has a non-boolean attribute (i.e., name=ID), where the primitive type ID is mapped to EString in the metamodel. An *Entity* also has a non-boolean attribute (i.e., name=ID), but in addition has a zero to one (i.e., the operator =) association (i.e., square brackets) with itself, as well as a composition of zero to many elements with the class *Feature*. Finally, the *Feature* class has an attribute *many*, which is an example of the boolean operator (i.e., ?=). It refers to the fact a feature can optionally be designated as *many* (i.e., the ?= operator, a boolean attribute). Furthermore, a feature has a name and can have a zero or one (i.e., the operator =) association with the class *Type*. The operators ***** and **?** found in lines 5 and 15, and line 14 respectively in Program 2.1 are general multiplicity operations in Xtext; exactly one (i.e., the default, no operator), zero to one (i.e., the operator **?**), one to many (i.e., the operator +), and zero to many (i.e., the operator *****). These operators further refine effectively the multiplicity defined by the = and += operators but are not reflected in the metamodel. However, they are enforced by the generated editor.

2.5 Sirius

Sirius [48] is an Eclipse project that leverages Eclipse Modeling technologies, such as EMF, to facilitate the rapid development of customized graphical modeling workbenches. It provides a flexible and adaptable framework for model-based architecture engineering, allowing language engineers to tailor the workbench to their specific requirements. Sirius employs a viewpoint-based methodology, which enables teams working on complex architectures to effectively collaborate and address specific topics within their domain. Models are created, visualized, and edited on the Sirius system using interactive editors known as modelers. Depending on the nature of visual representations, these modelers might be of many types. By default, Sirius supports three types of representations: tables, trees (hierarchical representations), and diagrams (graphical modelers). Programming can be used to add new representations.

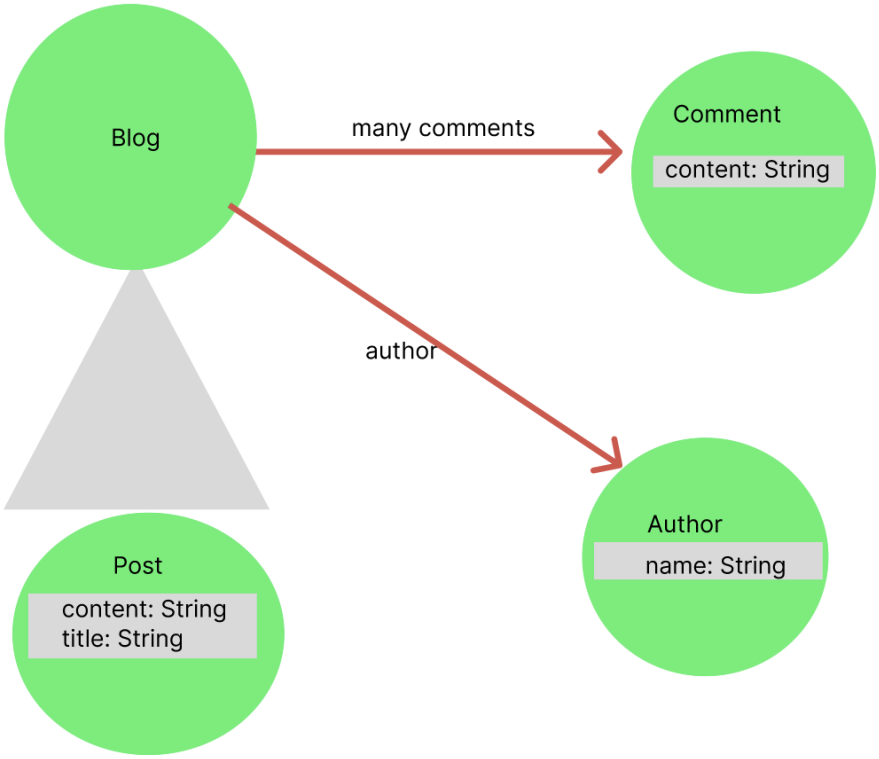


Figure 2.7: Sirius Representation of Xtext Language

By using the appropriate configuration file, known as the Viewpoint Specification Model (VSM) [49], it becomes possible to define how specific elements of the metamodel should be represented visually.

For instance, the VSM can be used to specify that a particular class should be represented by a green circle and its attributes should be represented by a grey rectangle. A red arrow points to an associated class, and a grey triangle represents an inheritance relationship between two classes. Figure 2.7 shows a mockup sample visualization of the running example illustrated in Program 2.2.

This provides a visual representation of the grammar example, allowing for a better understanding. By presenting the information in a graphical format, the visualization aims to enhance clarity and facilitate comprehension of the example's meaning.

2.6 Summary

In this chapter, we present essential background information related to models, data, the MODA framework, and the key technologies utilized for their representation. These technologies, namely Xtext, EMF, Ecore, and Sirius, hold significant relevance to this thesis. Chapter 3 offers an in-depth overview of the MODA metamodel. It incorporates an exploration of different variations that were explored and considered during the development process before arriving at the final metamodel design.

3

MODA Metamodel

In this chapter, we introduce the variations of the metamodel for the Models and Data (MODA) framework by outlining their main ideas and the design choices made to produce the respective version. The initial paper on MODA [31] does not define a metamodel for the framework, hence we opted to develop one as this is a prerequisite for developing a models editor for the MODA framework. We use the Xtext language engineering framework to develop the domain-specific textual language for the models of the MODA framework. Xtext is used to define the grammar of the textual language from which a metamodel is then generated in the Ecore format, which is a component of EMF.

3.1 MODA Metamodel

Numerous analyses and investigations were considered before the final metamodel was developed. This section details all the different variations of the metamodel and why each was taken into consideration. Each variation is thoroughly discussed, highlighting the reasons for its consideration and providing details of its advantages and disadvantages. By delving into the nuances of each variation, this section offers an understanding of the decision-making process behind the development of the final metamodel.

3.1.1 Metamodel Variation 1

The first variation that was considered when designing the metamodel for the MODA framework is depicted in Figure 3.1.

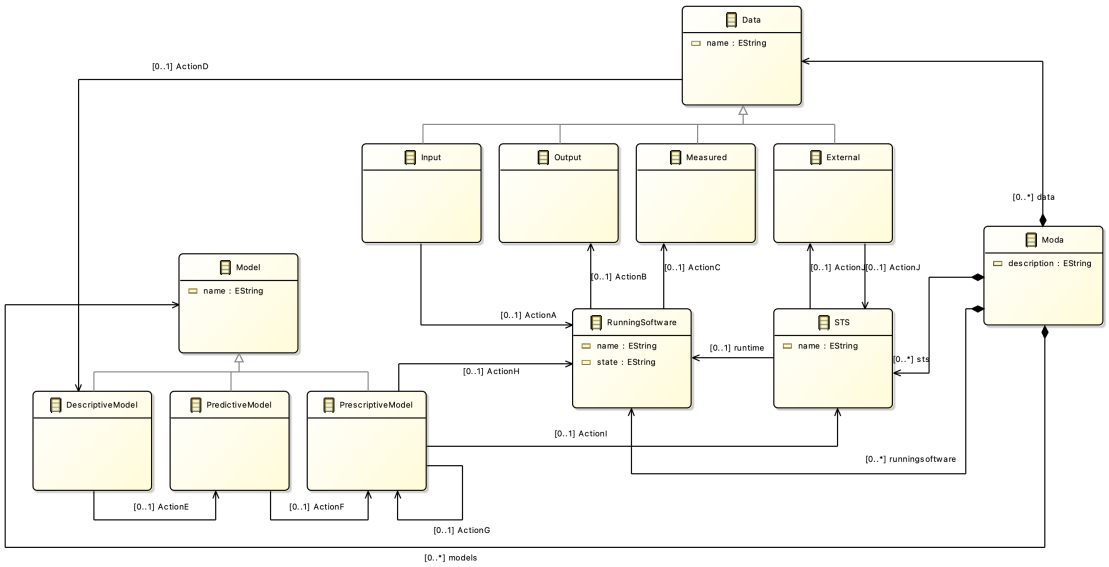


Figure 3.1: MODA Metamodel - First Variation

There is a root class called *Moda* with a description and it is composed of *Model*, *Data*, *RunningSoftware*, and *STS*. MODA can have $0 \ldots *$ *models, data, runningSoftware*, and *sts* depending on the socio-technical system. Both the *Model* and *Data* classes are superclasses and have a name. *DescriptiveModel*, *PredictiveModel*, and *PrescriptiveModel* all inherit from the superclass *Model* and *Input*, *Output*, *Measured*, and *External* all inherit from the superclass *Data*. *RunningSoftware* has a *name* and *state*. Aside from the current data (i.e., input, output, external, and measured

data), the *RunningSoftware* also has another form of data: state (i.e., the information that the system currently has). If one provides a system with some input data, it will do something with this input and make the necessary changes if need be. For example, a library system already has books. If one adds a new book to this collection, the number of books increases by one, and hence the current *state* of the library system changes. A system can only behave differently if the provided information is different. Socio-technical System (*STS*) has a *name* and may or may not have a *RunningSoftware*. Actions *A* through *J* are modeled as individual relationships. A class may optionally have actions that show the relationship from or to that class. Each allowed action is modeled explicitly in this variation. Table 3.1 outlines the type of Action, its name, and the class they move from and to in the metamodel for this variation. Action *J* has a bi-directional relationship between *STS* and *externalData*.

Table 3.1: Action Types Depicted in Variation 1

Action Type	**Action Name**	**From**	**To**
Action A	Input processing	Data (input)	RunningSoftware
Action B	Output or actuator data	RunningSoftware	Data (output)
Action C	Measurement	RunningSoftware	Data (measured)
Action D	Generalization, calibration	Data	Model (descriptive)
Action E	Preparation for prediction	Model (descriptive)	Model (predictive)
Action F	Analysis, decision, and change	Model (predictive)	Model (prescriptive)
Action G	Generation	Model (prescriptive)	Model (prescriptive)
Action H	Deployment	Model (prescriptive)	RunningSoftware
Action I	Enactment	Model (prescriptive)	STS
Action J	Other interplay (bidirectional)	STS	Data (external)

In Appendix A, Program A.1 depicts the Xtext grammar that was specified for the first variation of the MODA metamodel. The main advantage of this model is that it clearly shows every relationship between each class and action, as depicted in the MODA framework in Figure 2.1. The allowed source and target of an action in the MODA framework are modeled explicitly by a directed association in this variation. The downside is that the metamodel is rather complex, and a simpler metamodel that illustrates the MODA framework would make it easier to (i) understand the framework and (ii) manipulate MODA models. Furthermore, the metamodel is inflexible, i.e., it does not allow to specify MODA models where some actions or models do not exist. For example, the metamodel always expects data to be connected to a descriptive model and it is not possible to connect data to a prescriptive model. If the metamodel were to support all of these possibilities,

then it would be even more complex. Finally, the metamodel does not allow the description of an action in more detail, because the actions are modeled as associations.

3.1.2 Metamodel Variation 2

The second variation that was considered when designing the metamodel for the MODA framework is depicted in Figure 3.2. The *Moda* root class is still composed of *Model*, *Data*, *RunningSoftware*, and *STS*. However, actions are now reified into the *Action* class and the *Action* class is now also composed in the *Moda* root class. An *Action* has a name and a type. The *name* allows the action to now be described in more detail. The *type* of action is modeled as an enumeration class that contains all the types available (A to J). The remainder of the metamodel, i.e., the structure of *Data* and *Model*, remains unchanged from the first variation.

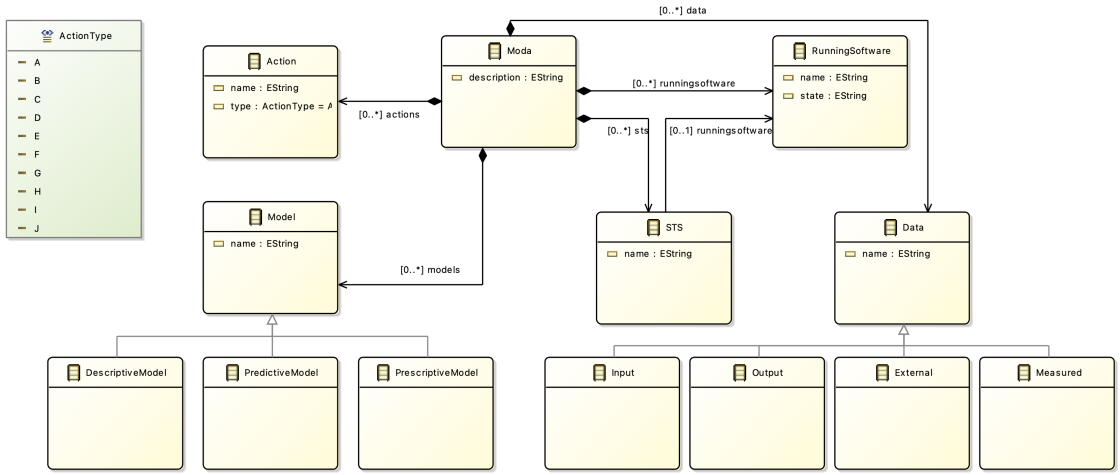


Figure 3.2: MODA Metamodel - Second Variation

The main advantage of this metamodel is that it no longer explicitly shows how each action is connected to the other classes in the metamodel, making the metamodel less complex. All the action types are now modeled in an enumeration that depicts the action. The downside is that the metamodel does not clearly show how the *Action* class is connected to the other classes in the MODA framework, e.g., the fact that Action A only goes from input data to the running software is not captured in the metamodel anymore. Actions are only composed in the root class, and no other relationship is established between the other classes and the *Action* class. Instead, these relationships are implicit and assumed to exist by convention.

3.1.3 Metamodel Variation 3

The third variation that was considered when designing the metamodel for the MODA framework is depicted in Figure 3.3. The *Moda* root class is now composed of an *Action* and an *Element*. The *Action* class represents the arrows or links defined in the MODA framework, and the *Element* class represents every other building block shown in the framework. The *Data*, *Model*, *RunningSoftware*, and Socio-Technical System (*STS*) classes all inherit from this super class and hence also inherit its attributes (i.e., *name*), its associations (target of associations from Action), and its composition with the root class. The remainder of the metamodel, i.e., the structure of *Data* and *Model*, remains unchanged from the second variation.

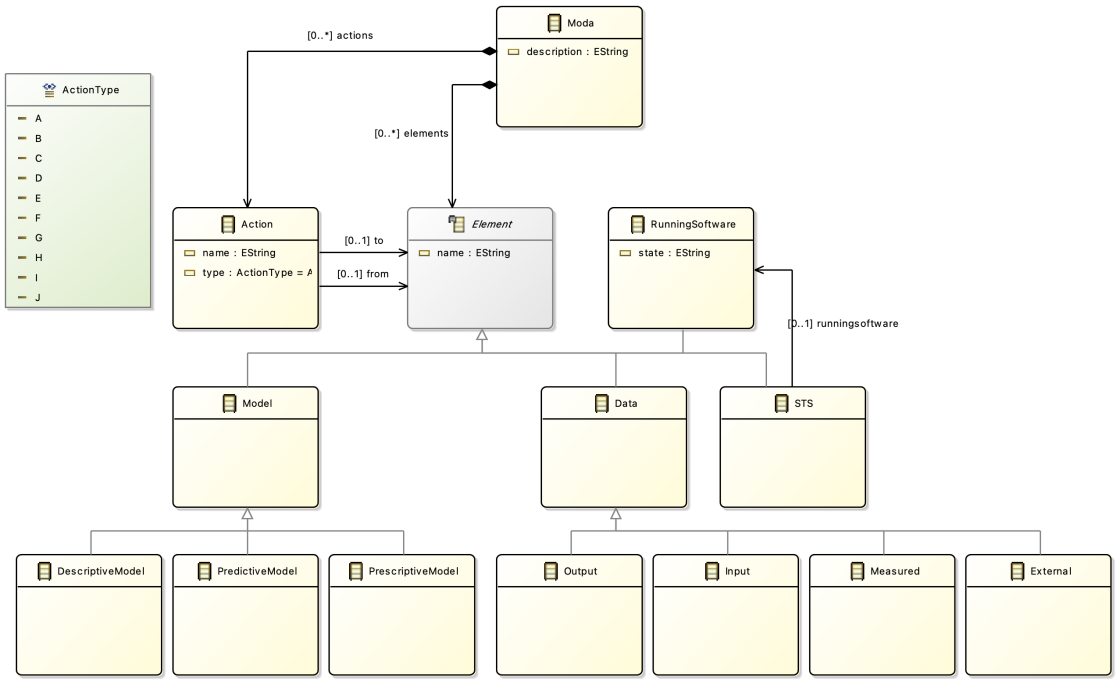


Figure 3.3: MODA Metamodel - Third Variation

The main advantage of this model is that it solves the problem encountered in the second variation of the metamodel shown in Figure 3.2. Also, all other classes are modeled as subclasses to a superclass called *Element*. A relationship from and to is also established between an element and an action, clearly showing how an action is connected to an element in the framework. However, while there is now a relationship between action and element, the specific constraints for a type of action (e.g., Action A must go from input data to the running system) are not covered by this

metamodel. Furthermore, the subclasses solution prevents a model to play more than one role at a time which is certainly possible (e.g., a model may play a prescriptive as well as descriptive role over time). Finally, this variation of the metamodel is still more complex than it needs to be.

3.1.4 Metamodel Variation 4

The fourth variation that was considered when designing the metamodel for the MODA framework is depicted in Figure 3.4. This variation is very similar to the variation shown in Figure 3.3. We modeled the respective subclasses of the *Model* class as attributes to the class, and this simplified the metamodel even further. The main advantage of this metamodel is that it is now possible for a model to have more than one model role at a time. This could also be achieved by applying the player-role pattern to the *Model* class. The boolean attributes are chosen because they result in a simpler metamodel.

The disadvantage is that a similar issue exists for the *Data* class as it did exist for the *Model* class in the earlier variations. With the current metamodel, it is not possible for a piece of data to be, e.g., input and output at the same time. This may not be an issue if only one system is modeled at a time, but more complex MODA models may cover a set of systems where the output of one system may be the input of another system.

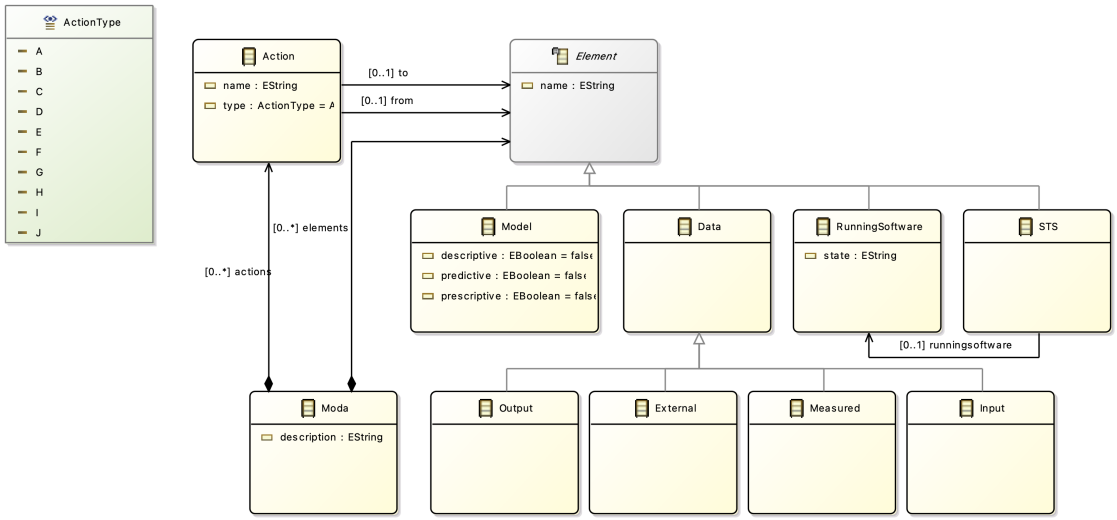


Figure 3.4: MODA Metamodel - Fourth Variation

3.1.5 Final Metamodel

The fifth and final variation that was considered when designing the metamodel for the MODA framework is depicted in Figure 3.5.

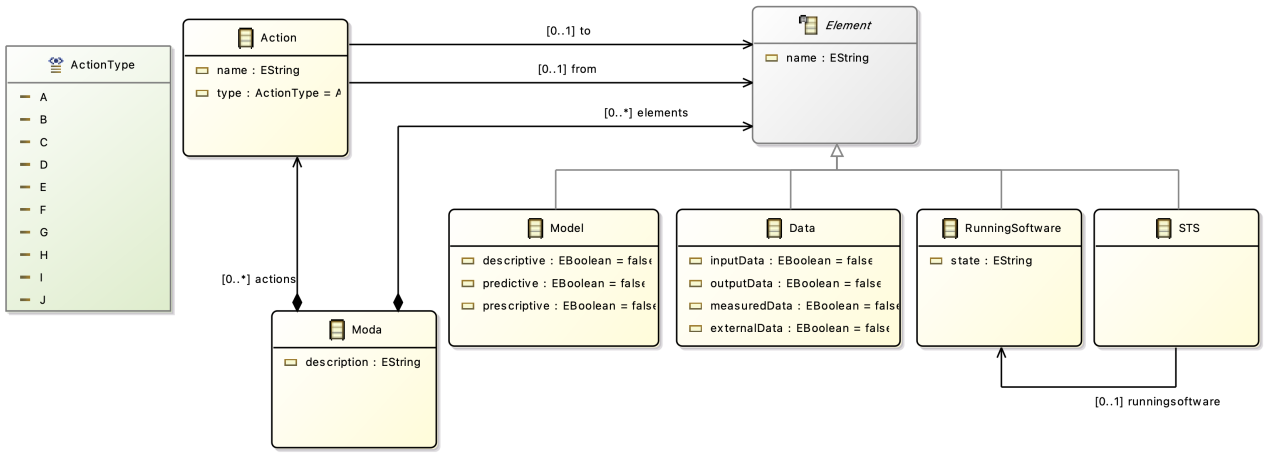


Figure 3.5: MODA Metamodel

The final metamodel is very similar to the variation shown in Figure 3.4. We further modeled the *Data* class just as the *Model* class in Figure 3.4 and this was the last consideration for the MODA metamodel. In this metamodel, we model the types of data used in the framework as attributes that can either be an input, output, measured, or external source of data. While this is the least complex metamodel for the MODA framework, it also allows one piece of data to be of several types. The final metamodel still does not explicitly enforce constraints related to each type of action, because validation rules are specified as explained in the next section.

3.2 Metamodel Validation Rules

A few validation rules were established to act as a way of checking a model and ensuring a model's well-formedness. The validation rules restrict which actions can be created so that invalid connections cannot be made for a MODA model. These rules are written with the Java programming language and can be found in Appendix B. Each rule has a condition, and if a model is modeled improperly, it produces an error message. The following is a list of the rules specified in the Object Constraint Language (OCL) [47].

1. Action name should be unique

   context Moda:

   inv: self.actions → isUnique(name)

2. Element name should be unique

   context Moda:

   inv: self.elements → isUnique(name)

3. At least one model should be present

   context Moda:

   inv: self.elements → select(oclIsKindOf(Model)) → notEmpty

4. At least one of the model attributes should be true

   context Model:

   inv: self.descriptive or self.predictive or self.prescriptive

5. At least one of the data attributes should be true

   context Data:

   inv: self.input or self.output or self.measured or self.external

6. Action A is from Input Data to Running Software

   context Action:

   inv: (self.type = ActionType :: A) implies (from.oclIsKindOf(Data) and
   from.oclAsType(Data).inputData and to.oclIsKindOf(RunningSoftware))

7. Action B is from Running Software to Output Data

   context Action:

   inv: (self.type = ActionType :: B) implies (from.oclIsKindOf(RunningSoftware) and
   to.oclIsKindOf(Data).outputData)

8. Action C is from Running Software to Measurement

   context Action:

   inv: (self.type = ActionType :: C) implies (from.oclIsKindOf(RunningSoftware) and
   to.oclIsKindOf(Data) and to.oclAsType(Data).measuredData)

9. Action D is from Data to Descriptive Model or Prescriptive Model

   context Action:

   inv: (self.type = ActionType :: D) implies (from.oclIsKindOf(Data) and to.oclIsKindOf(Model) and (to.oclAsType(Model).descriptive or to.oclAsType(Model).prescriptive))

10. Action E is from Descriptive Model or Data to Predictive Model

    context Action:

    inv: (self.type = ActionType :: E) implies ((from.oclIsKindOf(Model) and from.oclAsType(Model).descriptive) or from.oclIsKindOf(Data)) and to.oclIsKindOf(Model) and to.oclAsType(Model).predictive)

11. Action F is from Predictive Model, Descriptive Model, or Data to Prescriptive Model

    context Action:

    inv: (self.type = ActionType :: F) implies (((from.oclIsKindOf(Model) and (from.oclAsType(Model). predictive or from.oclAsType(Model).descriptive)) or from.oclIsKindOf(Data)) and to.oclIsKindOf (Model) and to.oclAsType(Model).prescriptive)

12. Action G is from Prescriptive Model to Prescriptive Model

    context Action:

    inv: (self.type = ActionType :: G) implies (from.oclIsKindOf(Model) and from.oclAsType(Model). prescriptive and to.oclIsKindOf(Model) and to.oclAsType(Model).prescriptive)

13. Action H is from Prescriptive Model to Running Software

    context Action:

    inv: (self.type = ActionType :: H) implies (from.oclIsKindOf(Model) and from.oclAsType(Model). prescriptive and to.oclIsKindOf(RunningSoftware))

14. Action I is from Prescriptive Model to Socio-technical System

    context Action:

    inv: (self.type = ActionType :: I) implies (from.oclIsKindOf(Model) and from.oclAsType(Model). prescriptive and to.oclIsKindOf(STS))

15. Action J is from Socio-technical System to External Data or External Data to Socio-technical System

context Action:

inv: (self.type = ActionType :: J) implies ((from.oclIsKindOf(STS) and to.oclIsKindOf(Data) and to.oclAsType(Data).externalData) or (from.oclIsKindOf(Data) and from.oclAsType(Data).externalData and to.oclIsKindOf(STS)))

3.3 Summary

This chapter focuses on the introduction and explanation of the various metamodel variations evaluated for the MODA framework. It highlights their advantages and disadvantages and discusses why the final metamodel was chosen. In Chapter 4, we focus on presenting the proof-of-concept tool built to support the MODA framework. This chapter provides details regarding the implementation and specification of the tool using Sirius. In addition, it covers aspects such as the tool's setup, application, and validation, offering a thorough understanding of its functionalities and capabilities.

4

Sirius Tool Implementation and Verification

This chapter introduces the tool built to support the MODA framework. We delve into how we implement and specify the tool with Sirius, its setup, application, and validation. This tool will enable one to visualize the MODA framework and its various relationships graphically and also provide editing capabilities to create new elements and relationships.

4.1 Metamodel

A metamodel is a visual depiction of tangible things for a language. The metamodel for the MODA framework is defined in Figure 3.5. This metamodel depicts the MODA framework with all its elements and relationships. We define this metamodel with EMF (Ecore Model), as seen in Figure 4.1. When the Xtext artifacts are initialized, the remaining projects and dependencies needed to fully implement the project are automatically generated and added to the main project.

They include the source code in the main project, the separate edit and editor projects, and their dependencies (i.e., generated in MANIFEST.mf).

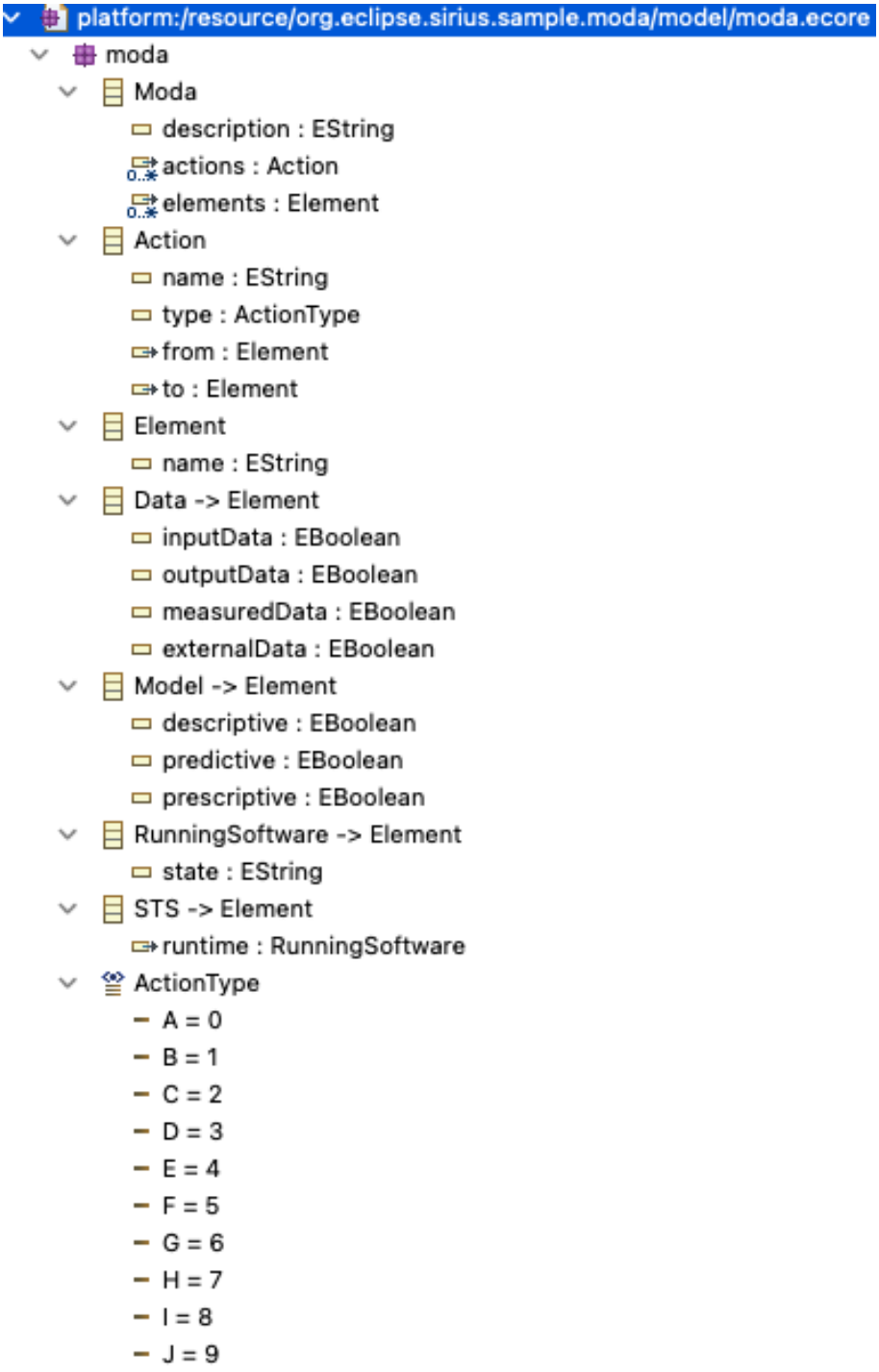


Figure 4.1: Ecore Representation of the MODA Framework

These projects are needed to test the metamodel in a new runtime window in the Eclipse IDE.

The edit and editor projects can be used to change Sirius's default icons. A designer can use separate icons to represent each element, which can be done in these project folders.

4.2 Viewpoint Specification Project

A new runtime eclipse application can be launched based on the Ecore file. Once running, we select the Sirius perspective, which provides specific Sirius menus and a modeling project. The Sirius modeling project consists of models and their related graphical representations used to build Sirius projects. A definition of the modeling workbench is included in the viewpoint specification project. The viewpoint specification project creation wizard creates a new project with a .odesign file. This file outlines the modeling workbench to be implemented, and the Sirius runtime will be able to interpret it.

4.2.1 Defining a Diagram

A diagram is added to the viewpoint and is configured to represent instances of models, as seen in Figure 4.2.

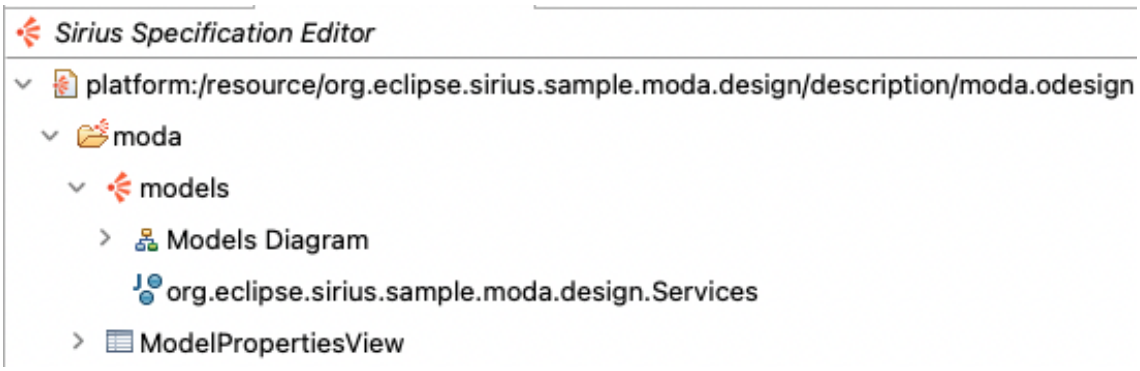


Figure 4.2: Diagram Definition in Sirius Perspective

As we have already defined the Models Diagram as a metamodel in Figure 3.5, this metamodel is linked to the Sirius perspective from the Eclipse registry as seen in Figure 4.3.

We then specify the properties of the Models Diagram by specifying its label and domain class and then add a node to the layer.

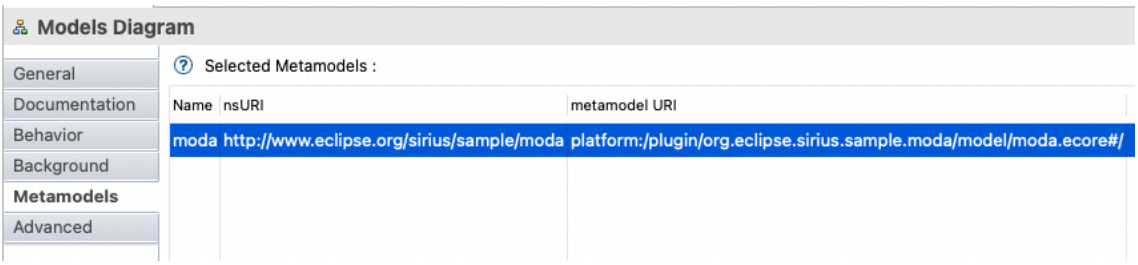


Figure 4.3: Snapshot of Metamodel Linkage in the Sirius Perspective

4.2.2 Nodes

The nodes represent the elements of MODA, i.e., *Data, Model, RunningSoftware*, and *STS*. All nodes created have an ID, i.e., the name of the node, and a metaclass, i.e., the name of their associated class from the metamodel. For example for the Data element, the id will be *DataNode* and the domain class will be *moda::Data*. The style of the node specifies the graphical attributes of the node, i.e., the shape, color, label size, label format, label alignment, and all other properties. The label expression is a set of rules written in the Acceleo Query Language (AQL). It limits the range of factors to consider before designing the graphical elements. They also act as the rules that establish the nature of the element and its associated attributes. Programs 4.1 and 4.2 show the AQL code written to define the name to be shown for a *DataNode* and *ModelNode*, respectively.

Program 4.1: AQL Code for DataNode

```
1  aql: self.name + if self.inputData then ' (input) ' else '' endif + if self.outputData then ' (
       output) ' else '' endif + if self.measuredData then ' (measurement) ' else '' endif + if
       self.externalData then ' (external) ' else '' endif
```

Program 4.2: AQL Code for ModelNode

```
1  aql: self.name + if self.descriptive then ' (descriptive) ' else '' endif + if self.predictive
       then ' (predictive) ' else '' endif + if self.prescriptive then ' (prescriptive) ' else ''
       endif
```

For example, Program 4.1 specifies that the name of the Data instance is shown first, and then *input*, *output*, *measurement*, *external* are shown if the instance's inputData attribute, outputData attribute, measuredData attribute, externalData attribute, respectively, are true. The same applies to all the other elements i.e., *Model*, *STS*, and *RunningSoftware*.

4.2.3 Edges

The relation-based edges are created to display the relationship between nodes. In reference to the MODA framework, each relation-based edge represents the various *ActionTypes*. The source and target nodes are indicated to provide the mapping from where the edge should start and where it should end, respectively, as shown in Figure 4.4. For *ActionTypeA*, it is linked to the Action class in the MODA metamodel which is referenced in the section called Domain class. This action moves from a *DataNode* (i.e., source mapping) to the *RunningSoftwareNode* (i.e., target mapping). The source and target mappings make use of the id defined earlier for nodes.

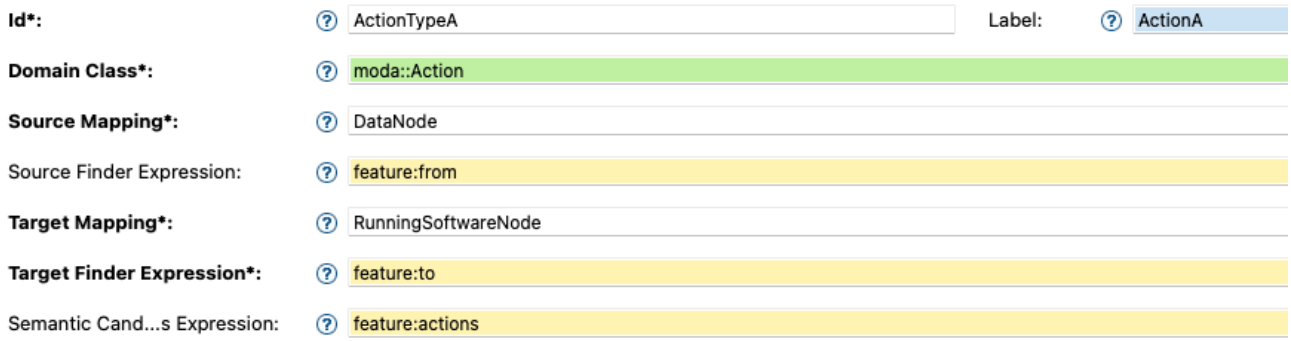


Figure 4.4: Snapshot of Relation-based Edges Properties for Action A

The style of the edges can also be specified, i.e., the shape, color, label size, label format, label alignment, and all other properties. The label expression for the edges is an expression with a boolean return type that will be evaluated on the relational edge. The edge will not be created if the boolean result is false. Program 4.3 shows the AQL code written for the precondition expression of *ActionType* A, i.e., only actions of type A are visualized according to the properties shown in Figure 4.4.

Program 4.3: AQL Code for Precondition Expression of Action A

```
1   aql: self.type = moda::ActionType::A
```

4.2.4 Palette

Aside from designing how the tool should depict MODA models, we also included a palette of tools that help users build new element components, new action components and also give them the

option to reconnect edges from one element to another. Figure 4.5 shows the section for the palette and an example of how the node creation model tool is set up.

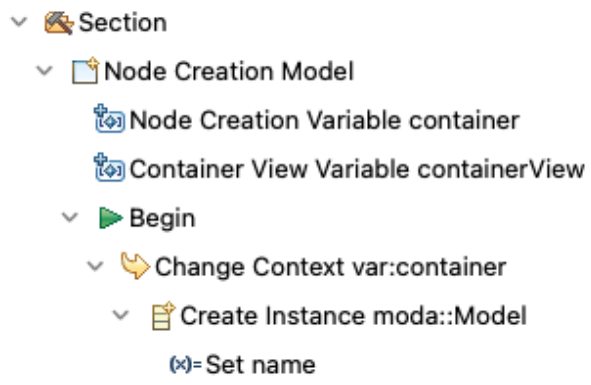


Figure 4.5: Snapshot of Palette Setup for Model Node Creation Tool

The ID and label are set and mapped to the *ModelNode* as shown in Figure 4.6, to allow access to all the properties set for a model in Program 4.2. This will enable the created element to have access to all the properties and rules set for the *Model* class.

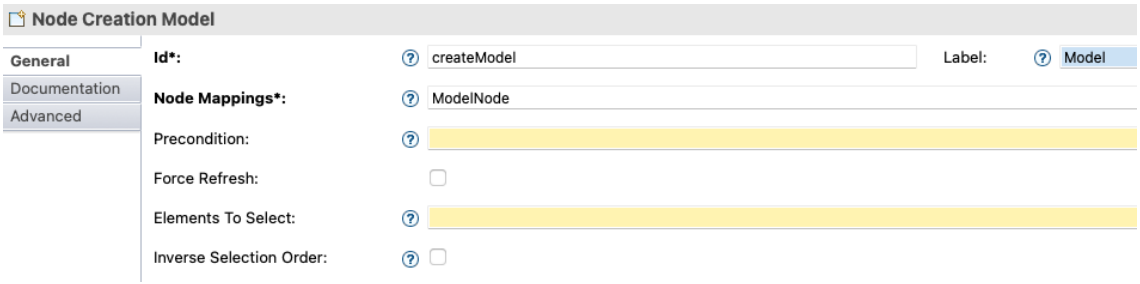


Figure 4.6: Snapshot of the Properties for the Node Creation Model

Containers are graphical elements within a diagram, displayed in a list or with shapes, squares, circles, images, etc, and are arranged based on the user's requirements or design preferences. A container is created and an instance of the model class composed in MODA will be created within this container. When a user creates a new instance of the model class whatever properties and rules are programmed and contained in this container will apply to this new model. The name of the model is also set as shown in Figure 4.7. An AQL code is written in the value expression section which is a value to set on the current feature (name in this case) and returns the type supported by the feature.

The code in Figure 4.7 is used to get the properties set for naming a node as shown in Pro-

gram 4.2 and allows a user to provide a name and set this name for a model. The value expression is used to identify and track the number of elements created (i.e., modelN). For example, the first model created is named by default model1, the second model2, and so on, but the modeler can change the name anytime in the perspective view.

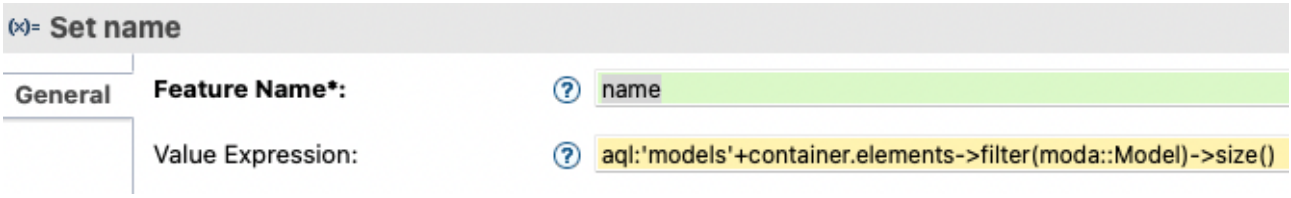


Figure 4.7: Snapshot of the Properties for Setting a Name for the ModelNode

Figure 4.8 shows the section for the palette and an example of how the edge creation for *ActionType* C tool is set up.

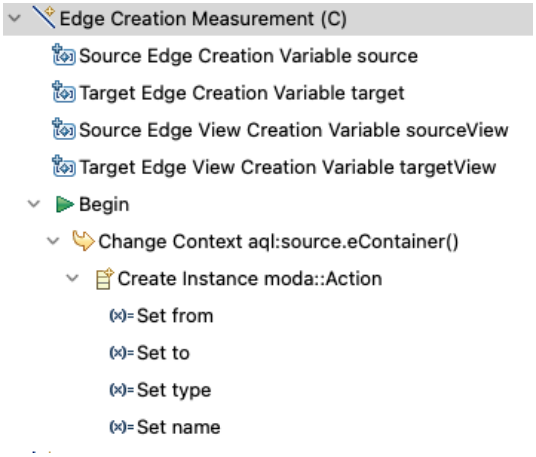


Figure 4.8: Snapshot of Palette Setup for the Edge Creation Tool

The difference between the source, target, *sourceView*, and *targetView* is that the source and target edge creation variable is the source and target of the relation going to be created and the source and target edge view creation variable is the graphical objects representing the source and target respectively. The connection start and complete precondition expressions are specified as shown in the properties tab in Figure 4.9.

These expressions are set to specify from where the edge is going to start and to where it can go. For example, in this case, an edge for Action C must start from the running software as seen in Figure 4.9. Furthermore, the expressions also prevent the user from creating a reflexive relationship to a node other than measured data as shown in Program 4.4 and in Figure 4.9.

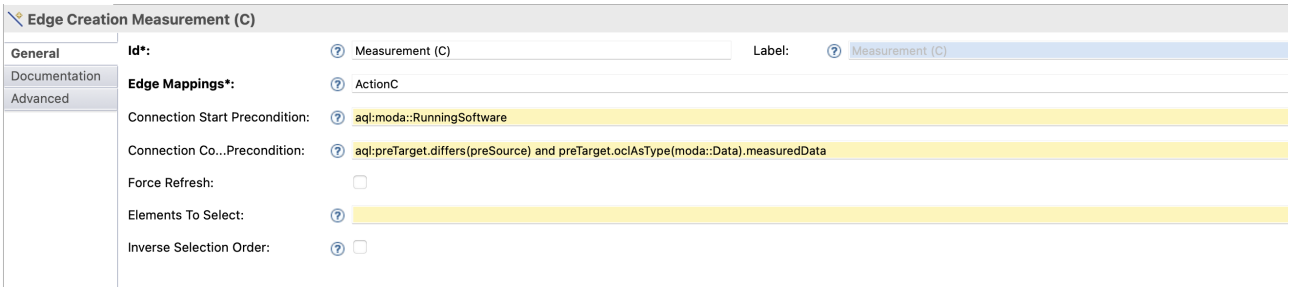


Figure 4.9: Snapshot of the Properties for an Edge Creation

Program 4.4: AQL Code for Connection Complete Precondition of Action C

```
1   aql:preTarget.differs(preSource) and preTarget.oclAsType(moda::Data).measuredData
```

preSource refers to the object on which the user initially clicked or interacted with first. *preTarget* denotes the object that is being pointed at or hovered over by the user's cursor. By using this precondition, the tool is designed to prevent the creation of forbidden links between objects. When a user attempts to create a forbidden link, the tool will display a specific icon or visual indicator to indicate that the action is not allowed. Edges can also be reconnected, which is set up similarly to the edge creation. The only difference is that the source and target edge reconnection properties are set up separately, each with an individual container as shown in Figure 4.10.

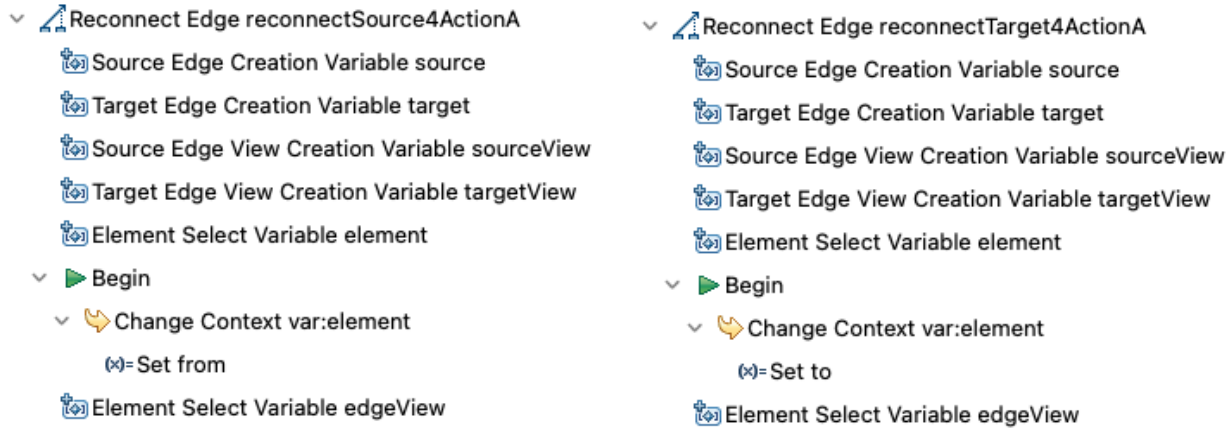


Figure 4.10: Snapshot of Palette Setup for the Edge Reconnection Tool

The source container specifies from which element an action can be moved and the target container specifies to which element it can be moved. This ensures that a user cannot make a connection that does not conform to the MODA framework. For example, *ActionA* is from *InputData* and to *RunningSoftware*. Therefore, a user cannot reconnect *ActionA* to any other

source element except *InputData* and to any target element except *RunninngSoftware*.

4.2.5 Validation

Five rules are written to validate the diagrams created by the tool. Figure 4.11 shows a snapshot of the validation rules generated in the Sirius perspective tool. All the validation rules defined in Chapter 3 are also specified in the Sirius perspective model. Rules 1 to 5 are defined in Figure 4.11 (i.e., at least one model attribute should be true, at least one data attribute should be true, action name should be unique, element name should be unique, and at least one model should be present, respectively).

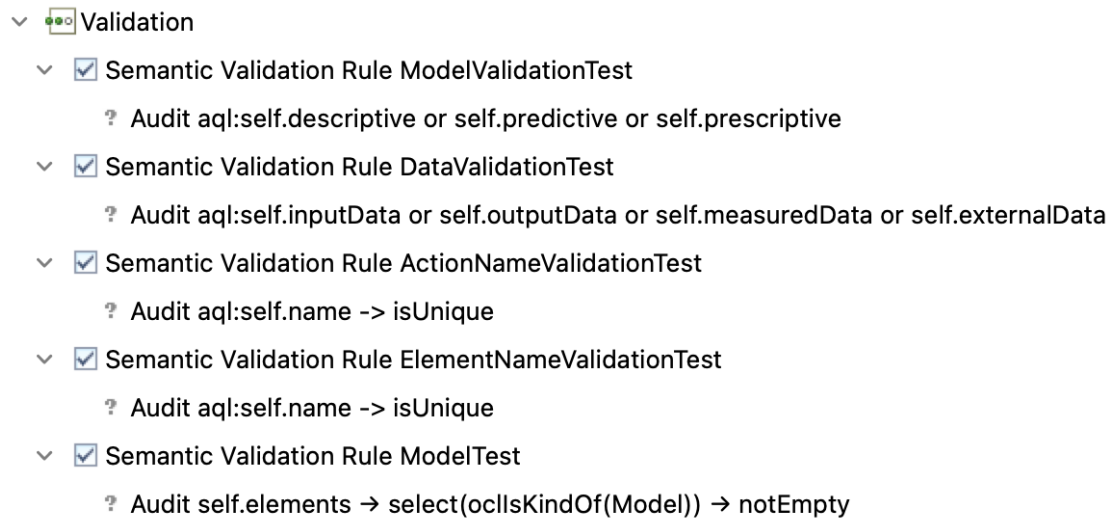


Figure 4.11: Snapshot of the Validation Rules in the Sirius Perspective

If this rule is violated, a warning message is displayed to inform the designer that there is a mistake in their diagram. Figure 4.12 shows a snapshot of the warning and the message to be displayed for the rule *"At least one model role must be set to true for a model"*.

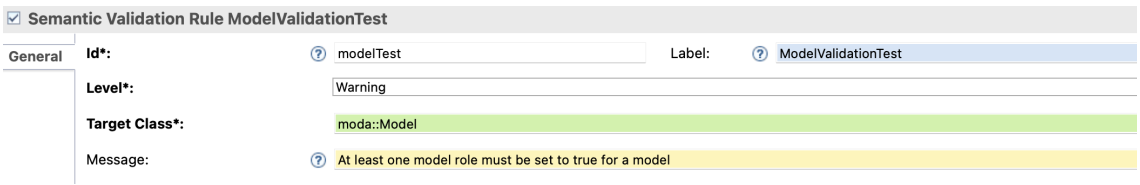


Figure 4.12: Snapshot of the Warning for the Semantic Validation Rule ModelValidationTest

The remaining rules (i.e., rules 6 to 15) are implemented in the action and element setup. For

example, rule 6 states that Action A is from input data to running software, hence in programming the Sirius perspective view a set of rules are written to determine the source and target of action A such that it only moves from an input data to a running software. No other connection will be allowed.

4.3 Diagram Representation

To test the diagram and view how each element is represented graphically as specified and defined in the elements and node setup, the Sirius perspective view enables an editor which displays the models contained in the modeling project and their respective representations. A MODA example from the reference paper [31] is shown in Figure 4.13.

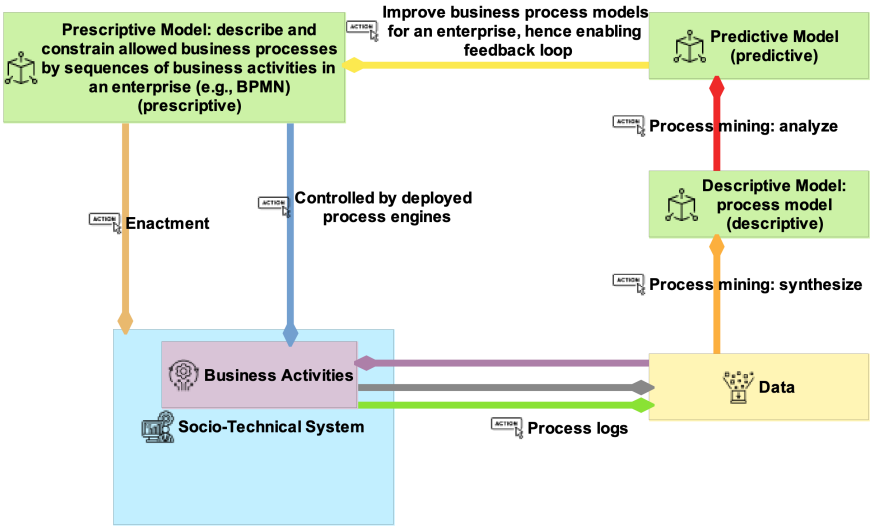


Figure 4.13: Business Process Modeling and Mining

Each element is represented with a rectangle and each action is represented with an arrow. Each element differs based on the color. The *Data* class is yellow, *Model* class is green, *RunningSoftware* purple, and *STS* blue. Each action type arrow has its unique color as well as seen in Figure 4.13. The arrow depicts where an element is moving from and to where (i.e., source and target) and the target is represented with a diamond arrowhead. The names of the elements and actions are the respective labels, as defined in the definition of the diagram. Unique icons are also used to represent the various elements to graphically define them.

4.4 Summary

In this chapter, we present the proof-of-concept tool built to support the MODA framework, focusing on how we implement and specify the tool with Sirius, its setup, application, and validation. In Chapter 5, we delve into the courses selected to analyze the framework in an exploratory study and discuss the various views and perspectives obtained from the analysis.

5

MODA Education Application

With an emphasis on minors in Software Engineering (SE) and Applied Artificial Intelligence (AI), we introduce nine courses offered by the Department of Electrical and Computer Engineering at McGill University in this chapter. The MODA framework [31] claims that it is applicable to many concepts, tools, technologies, and processes. An initial indication that this is the case is provided but a more comprehensive examination of this claim has yet to be done. We are going to do that with the help of the minors in SE and Applied AI as they help us scope the set of concepts that need to be modeled; essentially, we are asking the question of to what degree MODA can cover the concepts in these courses to further validate the claim of wide applicability made by the authors of the MODA framework.

The SE minor [101] offers a foundation in computer science, computer programming, and software engineering practice. The selected key courses taught in this minor that are analyzed in this thesis are ECSE 326, 223, 321, 428, 429, 439, and 250. The Applied AI minor [100] is designed

to provide a good foundation for applications of AI techniques in various fields. The selected key courses taught in this minor that are analyzed in this thesis are ECSE 250, 551, and 552.

As part of the research process, the instructors of seven courses actively participated in the study by providing valuable insights into the course contents and identified the key concepts and models needed to be represented within the MODA framework. This structure included a breakdown of the topics, sub-topics, and relationships between different concepts. The courses were reviewed by the instructors and a detailed structure of the specific requirements and objectives of each course were provided to be implemented with the MODA tool. For the remaining two courses (i.e., ECSE 429 and 552), the course structure available to students was carefully analyzed and reviewed. This analysis involved a thorough examination of the course materials, syllabus, and learning objectives to identify the key concepts and topics covered in each course. For five courses, the instructors also verified the MODA models of their courses and gave feedback that improved the final version of the MODA models.

5.1 ECSE 326 - Software Requirements Engineering

Software requirements engineering [72] [74] is a multidisciplinary effort that functions as a bridge between the acquirer and supplier domains in order to define and manage the requirements that must be met by the system, software, or service of interest. As a result, it serves as a link between the capabilities and opportunities provided by software-intensive technologies and the real-world demands of stakeholders who will be impacted by the system-to-be. It covers areas such as techniques for eliciting requirements, languages and models for specification of requirements, analysis and validation techniques, including feature-based, goal-based, and scenario-based analysis, quality requirements, requirements traceability and management, handling the evolution of requirements, requirements documentation standards, requirements in the context of system engineering and the integration of requirements engineering into the software engineering processes. Figure 5.1 shows the MODA representation of the course.[2]

Before developing a system or performing any form of analysis on the steps to take to build that system, there must be some form of planning. Before designing software, the requirements need to

[2]A MODA model is not a process model but rather a model that shows the roles of models and data and their actions.

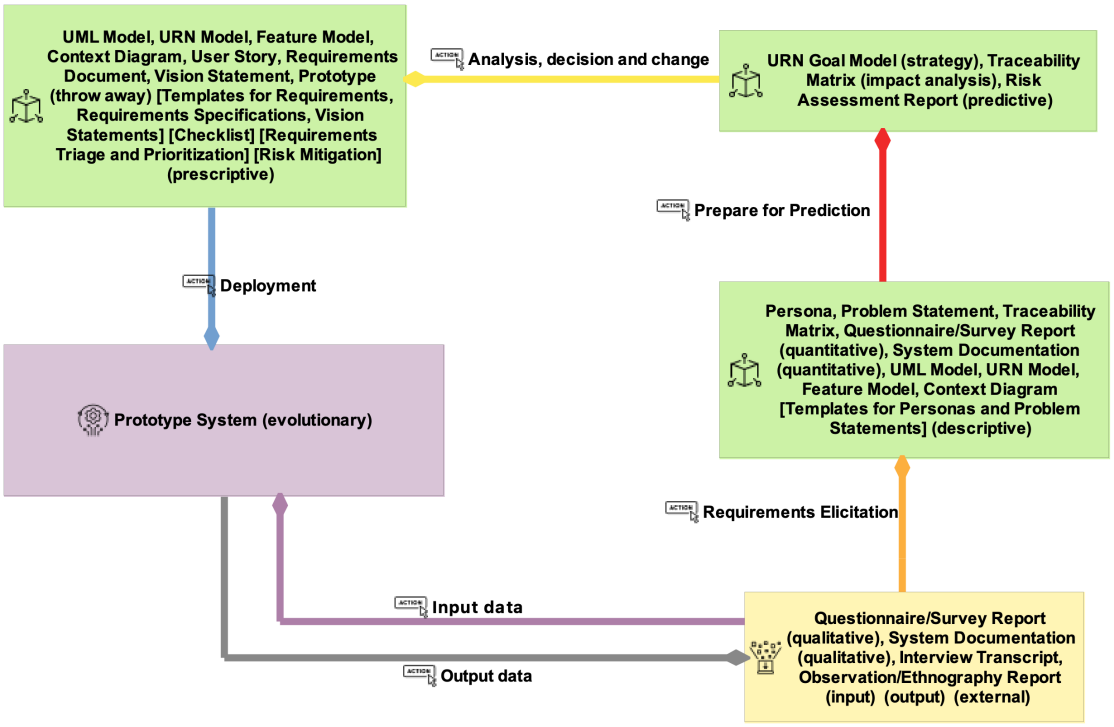


Figure 5.1: ECSE 326 MODA Diagram

be clearly outlined. We modeled all the processes, tools, and technologies used in ECSE 326 with MODA and categorized each as either a model or data based on the role they play in helping one develop requirements for a system. Some tools and technologies clearly operate on data or models but they cannot be modeled as data or models directly because they do not create data or models. We adapt the MODA framework by mentioning these tools and technologies in square brackets in data or models to be able to still capture their use in the course. We use this approach for all courses.

Data is information from the stakeholders on what software they want to build, its functionalities, and any other information that will help the developer create the software. This is *external data* by definition because it is not yet connected to a running system. The monitored data of a running system could also be useful in this context but the course does not explicitly focus on that (i.e., the course focuses mainly on the input and output data in the context of evolutionary prototypes). Hence, we do not include monitored data in Figure 5.1. Before beginning to build the software, the requirements engineer steps in and analyses the data, generalizing it in some cases.

The type of generalization that happens is called *requirement elicitation*. It is the process of

learning about a system's requirements through dialogue with clients, users, and other parties involved in its development. In elicitation, one may seek to learn about the domain, problem, and constraints; identify information sources and applicable methodologies; and create a first document that mainly contains user requirements and elicitation notes [17]. Some elicitation techniques result in more data, while others help generalize data into models. Hence, the *requirements elicitation* arrow in Figure 5.1 refers mostly to model-based requirements elicitation in its widest sense. Qualitative questionnaires/survey reports, interview transcripts, observation or ethnography reports, and any other qualitative system documentation are some of the elicitation techniques that result in more data as they try to discover the requirements for the system.

Surveys and questionnaires gather information from many people who respond to a specific question. When there are many or when stakeholders are geographically dispersed and there is the need to gather the same information from them, the survey/questionnaire elicitation technique is preferred. *System documentation* represents the outcome of the process of documenting the description of the software system and its components in detail [11].

One of the main elicitation techniques used is *interviews*. For example, the requirements engineer may use interviews to elicit information from a person or group by posing questions and documenting the responses in a formal or informal setting [99].

Observation or *Ethnography report* is a document that attempts to discover social, human, and political factors which may impact requirements. Ethnography seeks to collect what is ordinary or what people do (i.e., aim at making the implicit explicit), study the work context, and watch work being done [94].

The requirements engineer may then develop model-based artifacts, such as a persona, problem statement, traceability matrix, questionnaire/ survey report, system documentation, UML model, URN model, feature model, and context diagram. These are considered the descriptive model in MODA as they are models that describe the existing system, with templates used for personas and problem statements considered tools for these descriptive models. Note that we include persona and problem statements as well as quantitative elicitation outcomes as descriptive models because we take a very broad definition of model: anything with a clearly defined structure beyond natural language documents.

A *persona* is an archetypical user of a system, a template of the type of person who would

interact with it. It is often used as a stand-in for a real person when they are not available. The idea is that if you want to create effective software, it must be tailored to a specific user [14].

A *problem statement* asks the question, "What can be done" for a project to succeed, to meet the needs of its stakeholders who are not involved in the development. The template for a problem statement provides a structured outline that can be applied to a specific problem or challenge that the software system intends to solve. When the problem statement is not well-thought-out, there is the risk of creating a product that solves the wrong problem [9].

A *traceability matrix* is an artifact that links test cases, standards, regulations, certification documents, and design artifacts, among others, with requirements at various levels. Often, a customer's requirements and requirement traceability are documented or captured in a database or online tool. As this is an ongoing process, all this happens during the Software Development Life Cycle (SDLC) [59]. For example, one goal of a traceability matrix is to confirm that all requirements are verified through test cases, ensuring that no functionality is overlooked while performing software testing.

While qualitative questionnaire/survey reports and system documentation are categorized as data, they are descriptive models when they are quantitative in nature, because they abstract from the data as they capture or report numerical responses (i.e., multiple choice options, metrics, scales, etc) from participants that are analyzed statistically.

UML [2] is a modeling standard for software system design. This is also useful in requirements engineering, which employs *UML models* such as use case diagrams, class diagrams for domain models, activity diagrams, state machines, and to a certain degree, sequence diagrams [157].

A *URN model* [71] [15] is a standardized requirements engineering language that clearly communicates business goals and high-level functional needs to all stakeholders.

An illustration of the products classified as features in a family of systems is called a *feature model*. A feature of a system is a distinguishing trait, attribute, or characteristic. The link between the parent and child features, as well as which features are required, optional, and dependent upon, are all clearly mapped out in this model, which describes features and their dependencies [98]. A feature model specifies which combinations of features are valid and which are not.

A *context diagram* shows the input or output data flow between external entities (either stakeholders or external systems) and the system to be developed. It clearly shows the scope or bound-

aries of the system to be developed [24].

ECSE 326, to some extent, also covers a level of prediction in the form of a goal model, traceability matrix, and risk assessment. A *URN goal model* captures stakeholders' business goals, alternatives, decisions, and rationales. This model plays the role of the predictive model as it helps make some forecasts about the system. In particular, the aspect of the goal model that is predictive is the outcome of a goal model strategy, i.e., the specification of which solutions to consider and to what extent when analyzing stakeholder satisfaction. This helps depict a plan of action that aids the requirements engineer in planning, directing, and making forecasts that can help offer useful insight into the system. The traceability matrix, when used as a predictive model, focuses more on *impact analysis* and dependency analysis, which covers any analysis of changes that can occur within the system when deployed and their potential consequences. *Risk assessment reports* are the reports created after a risk assessment of the software system has been performed. Risk assessments are done to determine all aspects of the system or artifacts that may be harmful to the system during and after development. The report is predictive as it provides information on future effects on the system and their likelihood.

The requirement engineer then makes some analyses, decisions, and changes based on the data and information collected so far. Then, the engineer creates some form of specification for the system to be built. The techniques used in making these specifications are modeled as the prescriptive model of the system. Some techniques used for descriptive models can also be used for prescriptive models but changing the focus from describing what exists to prescribing what should exist. The techniques that serve as both descriptive and prescriptive models include the UML model, URN model, feature model, and context diagram. User stories, requirement documents, vision statements, and prototype systems (throw-away) are the remaining models that play only a prescriptive role.

A *user story* is a sentence, in a simple business language, that describes functionality to support the responses to the Who, What, and Why questions when posed and is often from the viewpoint of the user or client [52].

A *requirements document* is a document that contains all of the requirements of a product. It is written in such a way that people can understand *what* a product should do. However, in general, the document should avoid anticipating or defining *how* the product will function so that

interface designers and engineers can later use their expertise to provide the best solution to the requirements [153]. The requirements document [21] clearly and accurately describes each of the essential requirements of the system and its external interfaces. Each requirement must be designed in such a way that it is feasible and objectively verifiable by a prescribed method (e.g., by inspection, demonstration, analysis, or test).

A *vision statement* is a document of the system's present and long-term goals, typically provided by a stakeholder [82]. The key structure of a vision statement captures the target customer (i.e., for), the statement of the need (i.e., who), the product name and category, the primary competitive alternative, and a statement of the primary differentiation. The software product vision statement describes the core essence and overall objective of the software product and its outcome [126].

A *software prototype* is a quick implementation of a few features of the final system. It tries to solicit early opinions from stakeholders and prompts further elicitation [19]. When there is unavoidably no intention of putting the prototype in the final system, it becomes a *throw-away* prototype. When there is a plan to develop a highly robust prototype in a structured manner that will be continuously refined and possibly be turned into the actual system, it becomes an *evolutionary* prototype [152]. Hence, such prototypes are deployed to all relevant stakeholders and the output of the prototype is used to further improve the system.

Some tools used to implement these techniques include templates for requirements, requirements specifications, and vision statements. These templates specify the required structure of the artifact. Furthermore, checklists, requirements triage and prioritization, and risk mitigation are techniques used during the specification of requirements.

Checklist is a document that contains a list of procedures and tests that aid in determining whether a product is ready for deployment [96]. Checklists are widely used in a variety of disciplines to reduce human error and improve safety and performance. In addition, checklists can also be used as mnemonic devices, acting as a "reminder system" to assist experts in consistently applying procedures and processes [135].

Requirements triage and prioritization is the process of knowing which requirements are to be satisfied when there is a limited number of resources and time [136].

Risk mitigation is the process of dealing with identified and evaluated risks before they have a negative impact on a project (i.e., dealing with a concern before it develops into a crisis). If an

undesired event has already occurred on the project, it becomes an issue rather than a risk. Since no one can accurately anticipate the future with certainty, risk management is used to minimize the likelihood or impact of potential problems. Doing this increases the project's likelihood of success and lessens any financial or other effects of risks that cannot be avoided [38].

5.2 ECSE 223 - Model-based Programming

Model-based programming [78] is the process of integrating models with programming. The objective of this course is to introduce model-driven engineering for the modern development of software systems by identifying the concepts of a domain and their relationships with the help of a key structural modeling notation (i.e., UML class diagrams), expressing executable behavior with behavioral modeling notations (i.e., UML sequence diagrams and UML state machines), using code auto-generated from UML models in an application, expressing natural language constraints to make UML class diagrams more precise, and applying appropriate design patterns. This project-based course focuses on abstraction in software engineering, structural modeling, state-based modeling, modeling of object-oriented systems, code generation, natural language constraints in modeling notations, architectural and design patterns, integrated development environments, programming tools (i.e., debugging, continuous build or integration, version control and code repositories, defect and issue tracking, refactoring), and code review processes. Figure 5.2 shows the MODA representation of the course.

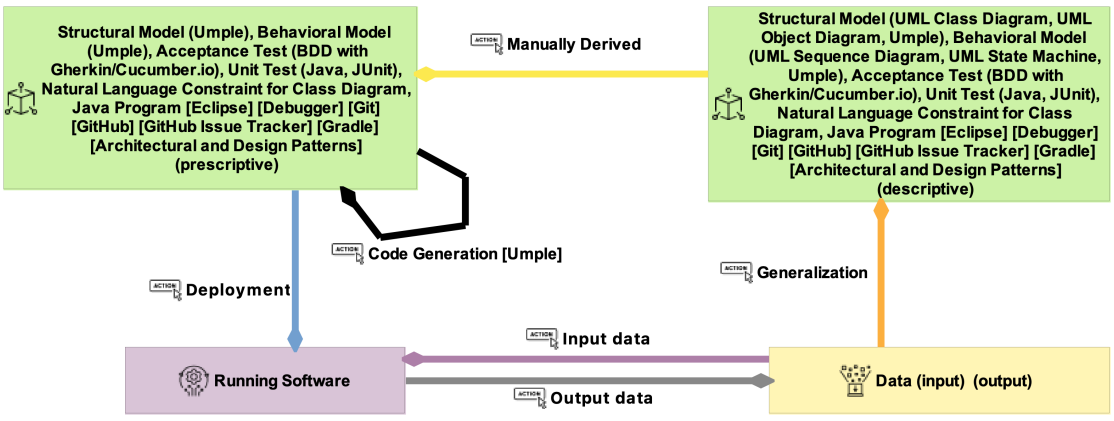


Figure 5.2: ECSE 223 MODA Diagram

The data represents the problem domain and what is to be modeled and this data is then generalized and formulated. The type of data that exists in this course is input and output data (i.e., observations). These observations are obtained from the currently running software. These observations are used to further adjust the problem domain and get a clearer picture of what needs to be modeled. Modeling is the process of producing a representation of a domain or software. The modeling engineer designs or develops structural diagrams (i.e., UML Class Diagram, UML Object Diagram, Umple), behavioral models (i.e., UML Sequence Diagram, UML State Machine, Umple), acceptance tests (i.e., BDD), unit tests (i.e., JUnit), and programs (i.e., Java), and writes natural language constraints for the class diagram. These are all modeled as the descriptive model according to MODA. Although these models could also be all used as prescriptive models, only some are used in this course to implement a system. They include Umple for the structural model and behavioral model, acceptance tests (i.e., BDD), unit tests (i.e., JUnit), natural language constraints for class diagrams, and the Java program. A large part of the Java program is generated automatically from the Umple models.

Structural Modeling is a type of modeling that is used to represent classes and objects found in the domain or in the software. This is done by using tools such as *UML* to create structural diagrams (i.e., class diagrams and object diagrams). Umple also supports class diagrams.

Behavorial Modeling involves depicting the system's states, actions, and how its components interact. UML sequence diagrams and state machines are two types of behavioral models used in this course. *UML sequence diagrams* are visual representations that illustrate the order and flow of operations in a system. They provide a clear depiction of how objects interact and communicate with each other during a particular scenario or sequence of events [103]. *UML state machines* are modeling techniques used to describe the behavior of a system or process by defining a set of states and the transitions between them. Each entity or sub-entity within the system is always in one specific state at any given time, and the state can change based on certain conditions or triggers. This approach helps to organize and understand how the system progresses from one state to another, ensuring that there are clear and defined pathways or rules for transitioning between states [156]. Umple also supports state machines.

Behavior-driven development (BDD) [122] is an agile approach that promotes teamwork and collaboration among business stakeholders, developers, and testers and emphasizes describing the

system's behavior using explicit scenarios and examples. In ECSE 233, the employed BDD approach uses Gherkin features and scenarios. Gherkin is a business-readable, domain-specific language designed specifically for describing behavior, and allows the removal of logic details from behavioral tests. Gherkin serves two functions, project documentation, and test automation [23].

Natural language constraints for class diagrams are the constraints written for class diagrams in a natural language, such as English, to help refine the models by specifying a condition to which the model must conform. In addition, after the constraints are implemented, they can be validated to check that the model adheres to the constraints [63].

Architectural and design patterns can be viewed as frameworks that assist in designing and structuring software systems and aim to help solve specific design problems in a software system. Architectural patterns [132] are the larger blueprint of the system as design patterns [121] are more focused on how the components of the software system are built.

Once the modeling is complete, the engineer reviews and analyses the models to determine the accuracy and completeness of the models. If there is the need for some changes to be effected, the models are revisited and readjusted to these revisions and the code undergoes multiple iterations (i.e., action G). This process goes on until the engineer is satisfied with a final model. The tools and technologies used to define both descriptive and prescriptive models in a programming language include Eclipse, Debugger, Git [55], GitHub and GitHub Issue Tracker [56], Gradle [68], and architectural patterns and design patterns. Once a final model is obtained, the system is implemented and deployed.

5.3 ECSE 321 - Introduction to Software Engineering

Software engineering [160] [62] is a systematic approach to developing computer software using engineering principles. A software engineer is someone who applies the engineering design process to create, maintain, test, and evaluate software. The software development process involves various activities such as defining requirements, writing code, assessing quality, managing changes, and continuously improving the software throughout its life cycle. Software engineering ensures that software is developed using established engineering techniques to achieve reliable and efficient results [155]. This course offers students a thorough introduction to the fundamental concepts

and methodologies of software engineering, focusing on developing large-scale software systems. Students will learn the basic concepts, methods, and best practices of software engineering. In addition, they will study different stages of software development, requirements engineering, software design principles, testing, and software maintenance. The course covers software systems' design, development, and testing, including different stages of the software life cycle, such as requirements analysis, software architecture and design, implementation, integration, test planning, and maintenance. As part of the course, students get to work on a group project, which builds them up to collaborate on projects. Figure 5.3 shows the MODA representation of the course.

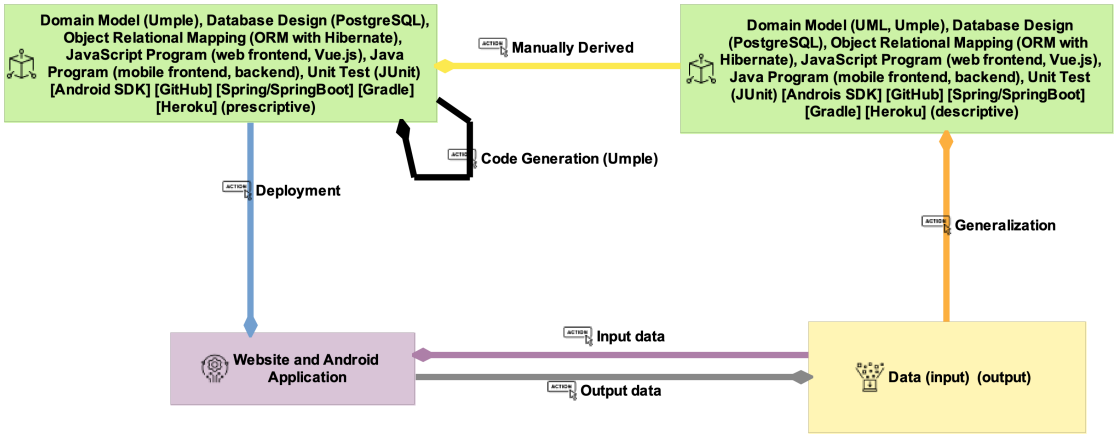


Figure 5.3: ECSE 321 MODA Diagram

Data is vital to make informed decisions, creating effective solutions, and improving software. In this course, input and output data are defined. The output data represents observations. These observations are obtained from the currently running software and are used to further adjust the problem domain and get a clearer picture of what needs to be modeled. The data is generalized into models for further development. Generalizing the data helps simplify and organize complex information for more accessible analysis and use in software development.

In building the application required in the course, the students are to gather requirements, design a multi-tier software solution to satisfy those requirements, implement the system, validate that the system is meeting the requirements, and develop a release pipeline to automate the software delivery process. Just as in the industry, the software engineer will have to do these and more to get an application, whether web or mobile, up and running. Domain model, database design, Javascript program (i.e., web frontend, Vue.js [139]), and Java program (i.e., mobile frontend, backend) are

used when developing a software system. All these techniques are modeled as both descriptive and prescriptive models.

Domain modeling is a software engineering and analysis technique to describe and represent the entities and relationships within a specific problem domain [114].

Database design refers to the process of organizing and structuring data in a way that is suitable for storage and efficient retrieval. The designer analyzes the data requirements and determines how the different data elements relate to each other. This information is then used to create a database model that defines the structure and organization of the data. A database management system is used to effectively manage and handle the data based on the design [146]. *PostgreSQL* is a powerful open-source object-relational database management system that is used for managing structured data. It is known for its reliability, scalability, and extensive feature set [107].

Object Relational Mapping (ORM) is a programming technique that enables developers to use object-oriented paradigms to interact with relational databases. ORM tools allow developers to manipulate and retrieve data using object-oriented syntax and concepts by mapping data stored in databases to objects in programming languages [151]. *Hibernate* is an ORM for Java programming that allows developers to map object-oriented models to relational databases. Hibernate handles object-relational impedance mismatch issues by substituting high-level object-handling functions for direct, persistent database accesses. Developers can use hibernate to define mappings between Java classes and database tables, specifying how objects and their properties are stored and retrieved from the database [148].

Javascript and *Java* are the programming languages used to develop the frontend and backend for web and mobile application in this course. Front-end development and back-end development are two fundamental aspects of software development. *Front-end development* focuses on the parts of a website or web application that users directly interact with. This involves creating and improving the user interface, enhancing the visual elements of web pages, and working on any issues during the debugging process. *Back-end development* primarily focuses on the server-side functionalities of a website or web application. Back-end developers focus on writing code that facilitates communication between web browsers and databases. They concentrate on ensuring the website's proper functioning by working with APIs, coding interactions with databases, utilizing libraries, designing data architecture, and handling other related tasks. The back-end and front-end development

collaborate to provide users with a functional and interactive experience [117].

JUnit is a testing framework designed to make it easy for Java programmers to write and execute tests. It provides a foundation for running testing frameworks on the Java Virtual Machine (JVM) and includes the TestEngine API, which allows developers to create their own testing frameworks that can run on the JVM. With JUnit, developers can write tests to validate the behavior of their Java code and ensure that it functions as intended [6].

Once the software engineer has analyzed and made the necessary decisions and changes, they develop the software system by writing the actual code and implementing the designed solution. The code undergoes multiple iterations of implementation, testing, and debugging until a final working version is achieved. Code generation from Umple models into Java code is also used. First, the development phase entails translating the requirements and specifications into a functioning software product. Next, the engineer utilizes their programming skills and expertise to create the necessary algorithms, data structures, and functionalities outlined in the system design. These help ensure the development process follows established practices and standards for efficient and effective software development. The tools and techniques used to model both prescriptive and descriptive models include Android Software Development Kit (Android SDK) [32], GitHub, Spring/Spring Boot [128], Gradle [68], and Heroku [115].

Android SDK is a software development kit, that encompasses different elements such as APIs that enable interaction with Android devices, tools for application development and debugging, pre-configured system images for testing on emulators or real devices, and documentation to aid developers in understanding and utilizing the Android platform [32].

GitHub is a platform and cloud-based service that facilitates software development and version control using Git. It provides developers with a centralized location to store and manage their code, enabling collaboration and efficient workflow management [56].

Spring is a framework that offers a comprehensive programming and configuration model for building enterprise applications using Java. *Spring boot* simplifies the process of creating standalone, production-ready applications based on the Spring Framework. It provides a streamlined approach where you can simply run your Spring applications without the need for complex setup or configuration [128].

Gradle is a free and open-source build automation tool that prioritizes flexibility and perfor-

mance. It allows developers to define their build processes using either the Groovy or Kotlin programming languages. With Gradle, developers have the flexibility to customize and optimize their build scripts according to their specific project requirements [68].

Heroku is a cloud platform that allows developers to deploy, manage, and scale applications. It supports a wide range of programming languages and frameworks, including Java, Ruby, Python, Node.js, Go, PHP, and more [115].

Once the application is completely programmed and functioning as required, it is deployed. Then, all application system functionality is accessible via the web or mobile front-end for the respective stakeholders.

5.4 ECSE 428 - Software Engineering Practise

Software process elements [137] [108] are the building blocks that make up a software development process. They are the various activities, tasks, and artifacts involved in the entire software development life cycle, from the initial conception of the software to its final deployment. A software engineer utilizes engineering principles in developing software and typically assumes responsibility for designing the overall system of a software application. Upon completion of the coding phase, software engineers conduct thorough testing to ensure that the software meets the specified engineering requirements. While the exact origin of the term remains uncertain, the inaugural software engineering conference, supported by NATO, took place in 1968 [112]. The conference aimed to tackle the issues of inconsistency and unreliability prevalent in software development and emphasized the importance of improving quality assurance (QA) and reliability. A consensus reached at the conference was that the systematic approach employed in traditional engineering disciplines should be extended to software development, given that those disciplines were already designed with similar objectives in mind. A software engineer typically oversees multiple coding projects, but software engineering encompasses much more than simply writing code. In reality, it covers all stages of the SDLC, starting from budget planning and extending to analysis, design, development, software testing, integration, quality assurance, and retirement [161].

In this course, students are taught the software practices to adhere to in the industry related to designing and commissioning large software systems. They are also introduced to the ethical, social,

economic, safety, and legal issues in software development, as well as the metrics, project management, costing, marketing, control, standards, Computer-Aided Software Engineering (CASE) tools, and bug/feature management when developing software. Figure 5.4 shows the MODA representation of the course.

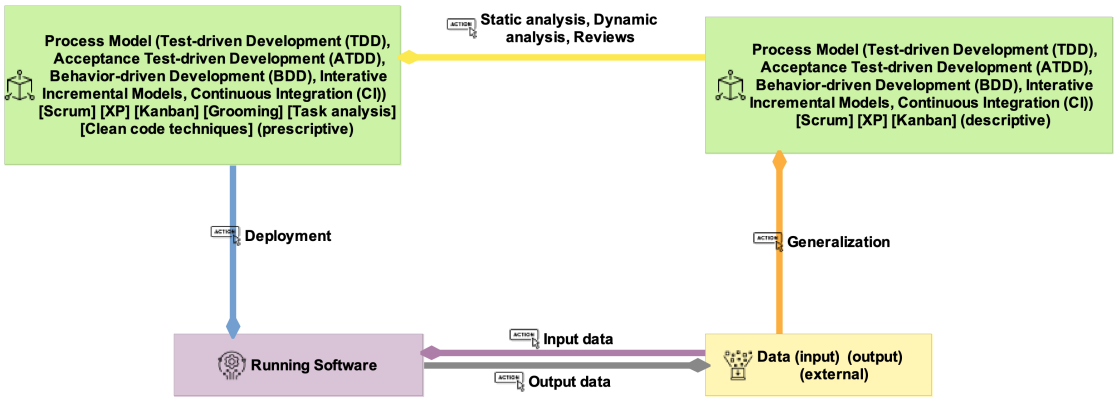


Figure 5.4: ECSE 428 MODA Diagram

There is some form of data that is generalized and by leveraging such data, organizations can gain valuable insights into various aspects of their software development practices. The combination and analysis of relevant internal and external data can help improve processes within organizations. Test-driven Development (TDD), Acceptance Test Driven Development (ATDD), Behavior-driven Development (BDD), iterative incremental models, and Continuous Integration (CI) are modeled as both descriptive and prescriptive models because they define specific methodologies useful in software development by providing guidance on how best to manage and structure the process.

TDD [22] is a development approach where tests are written before the actual code is written. One primary goal of TDD is that it helps the developer think through the requirements, then write the code afterward, aiming to help developers write clean codes [67].

ATDD is a collaborative method that involves business stakeholders, developers, and testers, to ensure that the software meets the required standards by creating upfront acceptance tests. It aligns development with user expectations and business goals. ATDD encourages the whole team to work together and gain a clear understanding before starting development [60].

BDD [122] is an agile approach that promotes teamwork and collaboration among business stakeholders, developers, and testers and emphasizes describing the system's behavior using explicit

scenarios and examples. It encourages teamwork to clearly define how the application should work through discussions and real-life examples. BDD combines techniques from TDD, object-oriented analysis, and domain-driven design to help teams work together efficiently in developing software [144].

Iterative incremental models describe a series of incremental steps in software development. The iterative process usually begins with a partial implementation of the software requirements and enhances the versions that are evolving until the system is completely implemented and deployable [131].

CI [34] is the development practice that automates and integrates code changes from several contributors into a central repository. This practice helps detect and resolve errors early in the development process, improving collaboration and reducing the risk of integration issues [113].

Static analysis, *dynamic analysis*, and *reviews* are performed as a way of analyzing, evaluating, and improving the quality of code and then making some level of decisions and changes to the software system. The tools and technologies used in the course include Scrum, Extreme Programming (XP), Kanban, grooming, task analysis, and clean code techniques.

Scrum is an agile project management framework that provides teams with a structured approach to managing and organizing their work. The framework is based on values, principles, and practices that guide the team's activities throughout the project lifecycle [33].

XP is a collection of engineering practices applied by numerous IT companies. It emphasizes aspects that are technical in software development. It emphasizes practices such as test-driven development, pair programming, continuous integration, and coding standards [10].

Kanban is a framework that implements DevOps and agile software development. It focuses on real-time communication, capacity management, and visual representation of work, allowing teams to view the state of every work at any time [111].

In agile software development, *grooming* is the process of reviewing and refining the product backlog. It involves getting rid of user stories that are no longer necessary, adding new user stories to meet new requirements, changing the priority of existing stories based on their importance, estimating effort, and clarifying requirements. Grooming sessions help everyone understand the work, find connections, and prepare for development [109].

Task analysis is a technique that breaks down complex activities into smaller, easier-to-handle

tasks. It helps understand the steps and actions needed to complete a specific task. This information helps design user interfaces and guide software development and testing. It is a valuable tool for project managers as it helps them overcome several project obstacles and create competent teams. By breaking down complex activities into smaller, manageable tasks, task analysis enables better planning, organization, and resource allocation, leading to improved project outcomes [92].

Clean code techniques offer guidelines and strategies for developers to write code that is easy to read and maintain. These techniques also emphasize good naming practices for variables and functions, organizing code effectively, and avoiding repetitive code, resulting in manageable and quality code.

All the tools and technologies in this course when implemented in real-world software development processes do not just end there. There is a form of feedback loop that occurs because software engineering practices happen every day. They promote adaptability, collaboration, agility, and continuous improvement throughout the software development cycle up until and even after the system is deployed.

5.5 ECSE 429 - Software Validation

Software Validation [1] [105] [130] ensures that a software system meets the specified requirements to fulfill its intended purpose. In software validation, the engineer asks, "Are we building the right product?". Building the right software product implies developing a requirements specification that accurately captures the needs and objectives of the stakeholders. If this document is incomplete or incorrect, the developers may be unable to build the desired product per the stakeholders' expectations [142]. The course focuses on the methods and techniques used to validate and verify software systems. It emphasizes the importance of testing and verification techniques in building reliable and high-quality software systems. ECSE 429 provides students with the necessary knowledge and skills needed to plan, execute, and evaluate software testing activities. The course covers the correct and complete implementation of software requirements, including requirements analysis, model-based analysis, design analysis, and extensive software testing at the unit and system levels, considering aspects like performance, risk management, and software reuse. The emerging field of ubiquitous computing is also explored in relation to software development. Figure 5.5 shows the

MODA representation of the course.

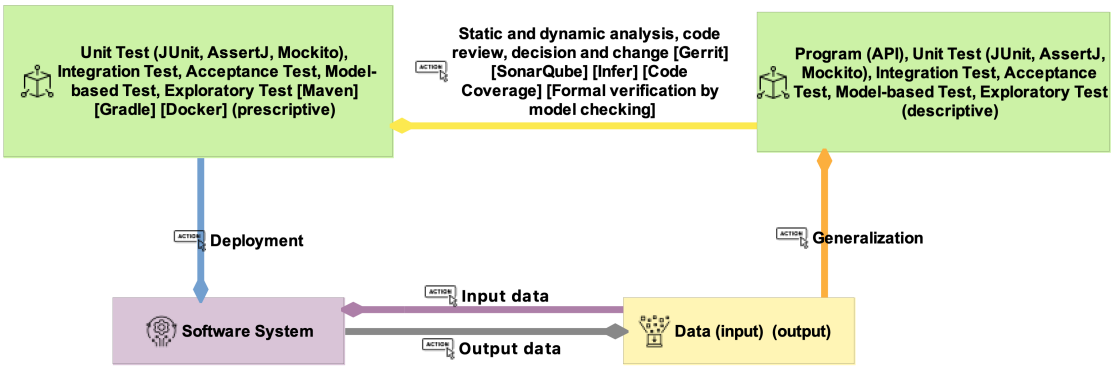


Figure 5.5: ECSE 429 MODA Diagram

Data in software validation can be either input, output, measured, or external, but only input and output data is emphasized in this course. During the validation process, the input data is the data provided to the software, output data are responses generated by the software, measured data refers to the data that is generated when the software is being executed, and finally, external data can be any data from external sources that are used to help validate the software. This data is generalized as generalization helps the data become more useful to eventually derive insights and support decision-making in software development. Software developers and QA engineers are key stakeholders in software validation and the sections of this course are mainly carried out by these engineers. The developer creates Application Programming Interfaces (APIs) (i.e., the descriptive model according to MODA), as part of the software development process and performs several tests to ensure the code is of good quality and functionality. These tests include integration tests, acceptance tests, model-based tests, and exploratory tests. Just like software programs and models that are first prescriptive to define what a system must do and then become descriptive to describe an existing system, these tests are also all modeled as descriptive and prescriptive models. Tests also first prescribe what the system must do, and then become descriptive, describing what the system currently does.

APIs [140] are mechanisms that facilitate communication between software components by utilizing a set of definitions and protocols. For example, consider a weather app on a phone. The app needs access to daily weather data, which is stored in the weather bureau's software system. In order for the app to retrieve and display weather updates on a phone, the app communicates with

the weather bureau's system through APIs. These APIs define how the app can request weather data and receive the appropriate responses. By leveraging APIs, the weather app can seamlessly access and present the desired weather information on a mobile device [13]. APIs are often the focus of tests.

Unit testing is a software testing approach that focuses on evaluating the individual units of source code. These units include computer program modules, along with their associated control data, usage procedures, and operating procedures. The purpose of unit testing is to ensure that each unit of code performs as expected and functions correctly on its own [76]. Tools such as JUnit [6], AssertJ [3], and Mockito [88] are used to perform unit tests.

Integration Testing is a part of software testing where testing is conducted on several complete, integrated systems to assess their ability to successfully communicate with each other and to meet the specified requirements [69].

Acceptance testing is a type of testing carried out to determine if a system meets its specified acceptance criteria and to allow the customer or end user to decide whether or not to accept the system [69].

Model-based testing is a way of testing software or systems using models that represent how the software should behave. These models help testers generate and run tests to check if the software is working correctly [150].

Exploratory testing is a software testing approach that involves a simultaneous process of learning, test design, and execution. It emphasizes the discovery of defects that may not be adequately addressed by predefined test cases. This type of testing relies on the expertise and intuition of individual testers to explore the software system and uncover potential issues that may have been overlooked. It allows for a more flexible and adaptable testing process, enabling testers to dynamically adjust their approach based on real-time observations and insights [104].

Code reviews [29] are a systematic code assessment intended to identify bugs and improve code quality. A code review is an important step in the software development process to get a second opinion on the solution and implementation before it is merged into an upstream branch such as the main branch [57]. GitHub [56] and Gerrit [54] are tools used to perform code reviews. *Static analysis* is performed as part of code review by using static analysis tools such as SonarQube [7] and Infer [5], to analyze and measure source code quality, identify bugs, vulnerabilities, and other

code issues, and provide ways to improve and maintain the code. *Code coverage*, which represents a percentage of the program's source code that is executed when a specific test suite is run [145], and *model checking*, which is a formal verification technique done to make sure that a system behaves correctly based on a specific model, are performed to ensure that the software system is functioning and meets every requirement.

Once the code has been inspected and reviewed and is up to standard, an analysis is performed to evaluate the overall functionality of the software system, and based on the analysis, decisions are made, and changes are made where need be. If the tests pass, the software system is deployed and made available to the respective stakeholders. If not, a report is generated and sent back to the respective engineer for review and change. The build or deployment tools and technologies used include Maven [87], Gradle [68], and Docker [4], as they are the technologies used in automating and validating the software.

5.6 ECSE 439 - Software Language Engineering

A software language is an artificial language used in the development of software systems [154]. Software Language Engineering (SLE) is concerned with the principled techniques and concepts for the construction of software languages [18]. Software languages come in many shapes and sizes, including programming languages, modeling languages, data format languages, specification languages, etc. This course focuses on practical and theoretical knowledge for developing software languages and models. It covers areas such as principles of model-driven engineering, concern-driven development, foundations for model-based software development, structural, intentional, and behavioral models as well as configuration models, constraints, metamodeling, model transformations, language engineering, domain-specific languages, models of computation, model analyses, and modeling tools. The objective of this course is to give students a strong foundation to succeed in a model-driven engineering environment by exposing them to various types of model analysis, introducing them to metamodeling to specify the abstract syntax, the concrete syntax, and execution semantics of languages, understanding how execution semantics may be specified, teaching them how to transform, merge, reuse, and generate models. Figure 5.6 shows the MODA representation of the course.

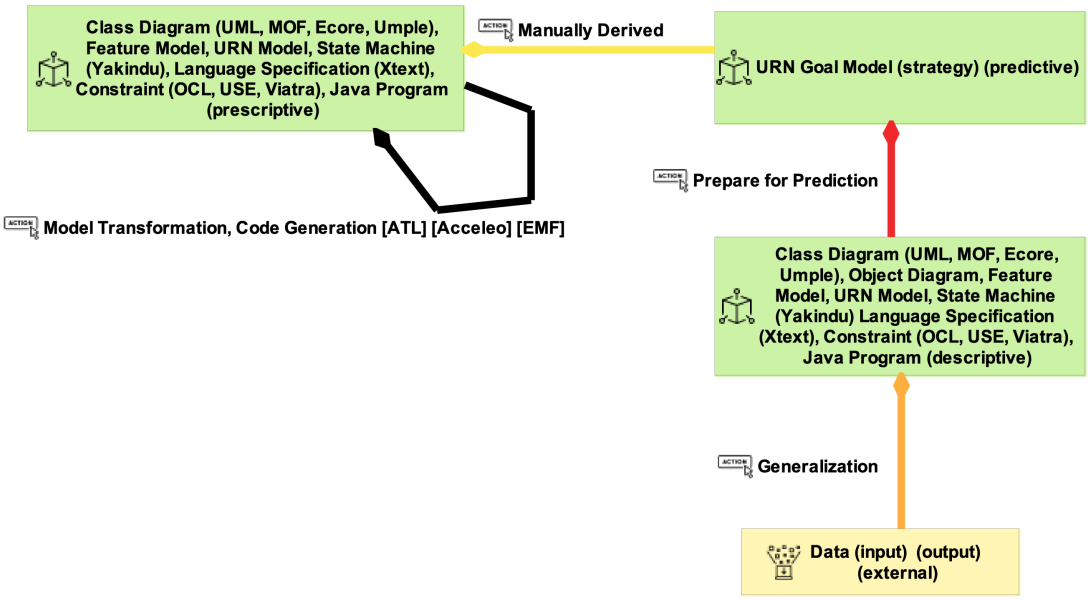


Figure 5.6: ECSE 439 MODA Diagram

The data represents the problem domain and what is to be modeled. This data is generalized and formulated to produce an output which in terms of MODA is a model. As is the case with many modeling notations, the same kind of model can be used as a descriptive or a prescriptive model. This is also the case for a language engineer. The language engineer designs or develops the respective structural diagrams, i.e., often class diagrams using the following tools and technologies to describe an existing language or build a new language. These tools and technologies include Meta-Object Facility (MOF) [127], Ecore [44], Xtext [50], UML-based Specification Environment (USE) [123], OCL [47], and Viatra [119]. Often, these techniques are generative, i.e., Java programs are generated from their models. Other tools and technologies are used as examples of metamodels (Feature Model [147], URN [71], State Machine [156]) or to specify sample systems (UML [2], Umple [133], object diagrams, Yakindu [70]). Note that object diagrams are not used to define a new system.

A *class diagram* is a type of UML diagram that uses classes to capture details about the entities that make up the system and the static relationships between them. A *class* is a description of a set of similar objects that have the same structure and behavior. An *object* is an instance of a class and an object diagram shows how the actual instances of a class are related at a specific instance of time [106]. Many technologies used in this course build on class diagrams. *Meta Object*

Facility (MOF) is an Object Management Group (OMG) standard for model-driven engineering. It is a language that is used for defining languages, including modeling languages such as UML. MOF defines the abstract syntax of modeling languages, allowing them to be precisely defined and understood by different tools and platforms [102]. *Ecore* is a diagram editor that allows one to design a domain model. It provides a design environment for modeling classes, datatypes, references, attributes, and all the classical Ecore constructs [35]. Umple is a modeling tool and programming language family that facilitates model-oriented programming. It extends object-oriented programming languages like Java, C++, PHP, and Ruby by adding abstractions such as attributes, associations, and state machines that are derived from UML. With Umple, one can create UML diagrams textually [133].

Feature modeling is a domain analysis technique that is used to define the software product line and system families. Features are used to identify and organize the commonalities and variabilities within a domain, and to model the functional and non-functional properties [36].

The *User Requirements Notation (URN)* is a graphical language that combines the Goal-oriented Requirement Language (GRL) [25] and Use Case Maps (UCMs) [16] for modeling and analyzing requirements. It allows engineers to define and analyze goals, scenarios, and their relationships in a lightweight and semi-formal manner. URN helps in discovering and specifying requirements for new or evolving systems, and it facilitates the analysis of these requirements for accuracy and comprehensiveness [95].

State machines, commonly used in computer science and represented in UML, are a way to organize and control the behavior of a device, computer program, or other (often technical) process works. They ensure that an entity or its parts are always in one specific state from a set of possible states. Transitions between states are clearly defined and depend on specific conditions. This enables precise control and coordination of how the system operates based on its current state [156].

Xtext is a framework for defining programming languages and domain-specific textual languages, by utilizing a powerful grammar language [50].

UML-based Specification Environment (USE) is a system that helps in specifying and validating information systems using a simplified version of *UML* and *OCL* [47]. It allows one to describe a model using UML class diagrams (i.e., classes, associations, etc). Additionally, *OCL* expressions

can be used to define additional rules and constraints for the model. *OCL* is a language used to express constraints and rules in model-driven development. It is specifically designed to work with EMF and is often used alongside UML and other modeling languages. With *OCL*, one can define precise constraints and expressions that are applied to models and their elements. These constraints serve as checks to ensure that models are correct and consistent, meeting the specified rules and requirements [47]. One of the key features of *USE* is its ability to animate the model (i.e., simulate and test the model against various requirements). During the animation, snapshots of the system's states can be created and manipulated, which gives a better understanding of how the system behaves in different scenarios [134]. *VIATRA* offers a framework for querying and transforming models, facilitating the seamless exchange of information across different documents and models, as it focuses on optimizing the efficiency of these operations. The framework is designed to support the reactive programming paradigm, enabling event-driven transformations that dynamically adapt to changes in the models. This means that transformations can be performed in real time as the models evolve, ensuring the accuracy and timeliness of the results [119].

The *URN goal model* is modeled as the predictive model according to MODA. A goal model captures stakeholders' business goals, alternatives, decisions, and rationales. These are modeled as the predictive model as they help make some forecasts about the system. The aspect of the goal model, modeled as a predictive model in the MODA framework, focuses more on the strategies, which include the model goals, stakeholders' indicators, rationale, and decisions. This helps depict a plan of action that aids in planning, directing, and making forecasts that can help offer useful insight for the system. *URN* is used as an example software language in this course.

According to the MODA paper [31], the Generation action represents the typical software development activities that use high-level prescriptive models (e.g., requirements models) to produce lower-level prescriptive models (e.g., design models or executable code). The techniques used here include model transformations, which are covered by ECSE 439. Model transformation is an automated way of modifying and creating models. It is the automatic manipulation of input models to produce output models that conform to a specification and have a specific intent. This is done to convert models to other software artifacts (e.g., models or code). ATL [43] and Acceleo [42] are tools used to transform models, and EMF [45] is used to generated Java code.

5.7 ECSE 250 - Fundamentals of Software Development

Software development [89] encompasses the entire process of creating software programs. It involves designing the software to meet business needs, developing it according to the design, and deploying it for actual use. There may also be ongoing maintenance of the software, depending on the project [141]. The fundamentals of software development course is designed to teach students a logical and systematic approach to problem-solving. It provides them with the necessary skills to tackle technical challenges and develop reliable software solutions. The course emphasizes industry-standard best practices in software engineering, promoting the creation of high-quality and scalable code. Through a mix of theoretical learning and practical application, students acquire the abilities to solve complex problems, build robust software systems, and meet the professional standards of the software engineering field. The course focuses on software development using object-oriented programming. It covers basic data structures such as lists, stacks, and trees as well as algorithms for searching and sorting. In addition, students learn about asymptotic notation (Big O) and are introduced to tools and practices used in professional software development. Figure 5.7 shows the MODA representation of the course.

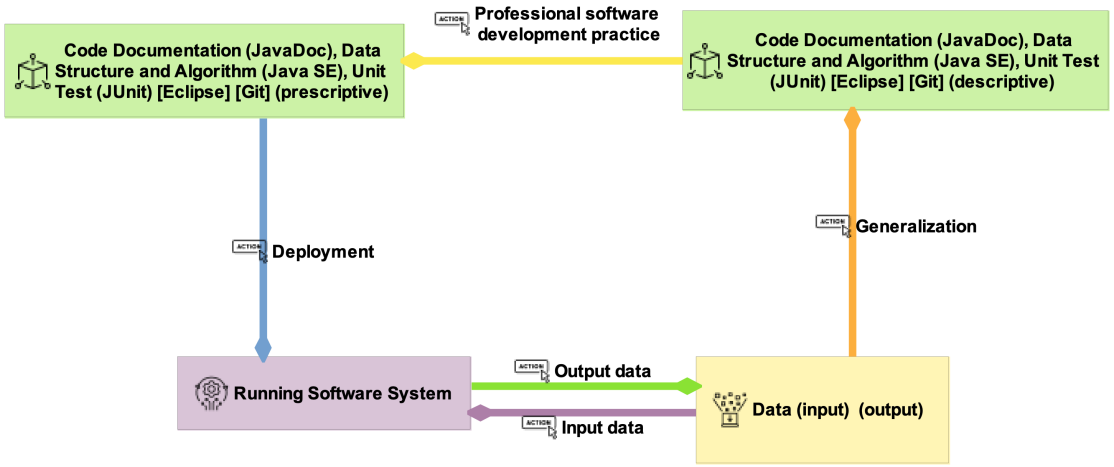


Figure 5.7: ECSE 250 MODA Diagram

Data aids in decision-making and improves the overall software development process. It provides valuable insights for developers to make decisions, create designs that meet user needs, conduct testing, optimize performance, and drive continuous improvement. Data in software development

plays a significant role in enabling developers to make informed decisions, create effective solutions, and continuously improve their software products. According to the MODA framework data in software development takes various forms. Input data serves as the information fed into the system, while output data represents the results or outcomes generated by the software. Measured data refers to the data collected during the execution of the software, which can be used for analysis and evaluation. Finally, external data can also be utilized in software development, depending on the socio-technical system being considered, to provide additional context, information, or resources for the development process. Data in software development plays a significant role in enabling developers to make informed decisions, create effective solutions, and continuously improve their software products. All four types of data are used in software development but only input and output data are emphasized in this course. The data is then generalized into models for developing software. Generalizing the data helps simplify and organize complex information, making it easier to analyze, model, and use in the software development process. Data structures and algorithms (in the Java programming language) and unit tests (in JUnit) are modeled as both the descriptive and prescriptive models of this course according to MODA. Code documentation is modeled in addition as a descriptive model.

Code documentation is the practice in which software developers provide clear explanations and visuals to describe the functionality and usage of their code. It makes the code easier to understand, use, and reproduce [20].

Data structures [8] [93] are the fundamental components in computer science that determine how data can be organized, stored, and retrieved in a computer's memory. Data structures help us work with data efficiently, allowing us to design better algorithms and software systems. The problem domain and its requirements determine which data structure is best for the problem. The proper selection of the appropriate data structure can significantly influence the efficiency and performance of software systems

An *algorithm* [8] is a finite sequence of clear and precise instructions used to solve specific problems or perform calculations. It provides a systematic approach to problem-solving by outlining the steps required to achieve the desired result. They are essential because they provide a structured and efficient way to solve problems and achieve desired outcomes in computer programs.

Unit testing is performed with JUnit and focuses on evaluating the individual units of source

code.

Once the developer writes the code and makes documentation, the relevant development practices are implemented to ensure the creation or development of excellent software that meets user requirements and industry standards. According to MODA, these professional practices are labeled as *ActionF* because they help in analysis, effecting changes, and making decisions to help optimize and improve the software under development. These practices are taught and explained in detail in ECSE 428.

The tools and technologies used to develop the software include Eclipse and Git [55]. Once the software development process is completed, and all the necessary practices are applied to refine and improve the software, the result is a functional and operational software system (albeit small as needed for this course).

5.8 ECSE 551 - Machine Learning for Engineers

Machine learning (ML) [26] [90], a sub-field of computer science and artificial intelligence (AI), aims to accurately mimic the way humans learn using data and algorithms while steadily increasing its accuracy [120]. ML has become an essential aspect of technology because of its capability to take data of any form, learn the data, and then make predictions based on an algorithm. As a result, ML massively aids in decision-making within applications and businesses, which optimizes growth metrics. The structure of the course taught at McGill covers areas such as fundamental ideas and challenges of machine learning, regression and classification under supervised and unsupervised learning, the curse of dimensionality, dimension selection and reduction, estimation of errors, and empirical verification; placing a focus on ethical procedures and practices for the deployment of real systems. Figure 5.8 shows the MODA representation of the course.

ML always begins with data. Data can be of any form, i.e., text, images, or numbers, but pictures may either be very noisy or too sharp, and text may contain unwanted characters, such as punctuations which may not be necessary for the model to learn. Data preprocessing is the process of cleaning up unclean data. Often, there is an extensive dataset available online that can be used to train machine learning algorithms, but before it can be fed into a machine learning model, the data must be cleaned and processed in some way, regardless of whether the ML engineer wishes to

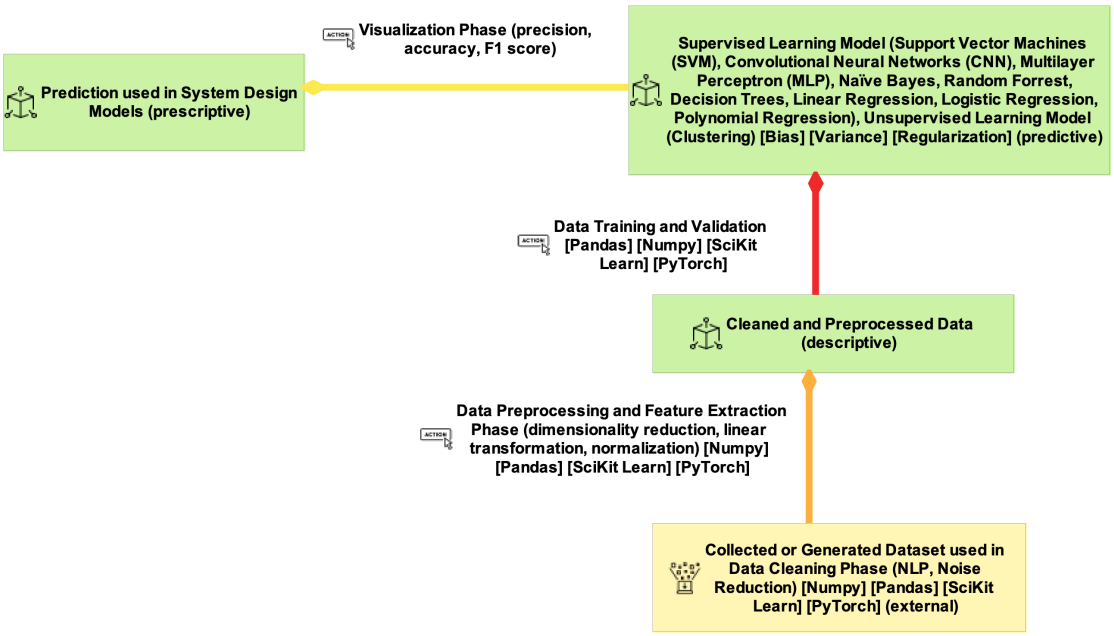


Figure 5.8: ECSE 551 MODA Diagram

obtain this pre-existing data or create a new dataset for the task. Furthermore, the data must be formatted correctly to produce better results. For instance, Random Forest does not handle null values; as a result, all null values from the initial raw data set must be managed if this method is to be used. In addition, the format of the data must meet specific constraints for ML models [40]. Hence data needs to be cleaned and processed before being passed as input to an ML model. Natural Language Processing (NLP) is used in data cleaning to preprocess and clean textual data before it can be used to train machine learning models. Techniques such as stop word removal, tokenization, spelling check, and text normalization among others are used. Noise reduction on the other hand is used to reduce "noise" in a dataset by removing or reducing irrelevant data points. Numpy, Pandas, SciKit Learn, and PyTorch are the tools and ML libraries used in the data-cleaning phase to prepare the data for the model.

After the data is cleaned, it is pre-processed and manipulated to fit the ML model to enhance the data and make it more meaningful. Techniques such as feature extraction, dimensionality reduction, linear transformation, and normalization are incorporated.

If there are a lot of input variables, the performance of the ML model can diminish. *Feature Extraction* is a technique for reducing the number of features in the dataset by extracting new

features from existing ones and subsequently throwing out the original features. It preserves the most information necessary to replicate the original data while reducing feature space [86]. *Dimensionality reduction* is a method used to cut down on the number of variables used as input in the training dataset [138].

Raw data comes in different distributions which, makes it difficult to analyze and create models without some preprocessing. *Normalization* is the process of changing the shape of the distribution. With normalization, the values of the numeric data are adjusted to a common scale without changing the range [81]. *Transformation* converts an input from one domain to an output of the same or another set. A machine-learning model transforms its input data into meaningful outputs, a process that is "learned" through exposure to known input and output examples. As a result, the central problem in machine learning and deep learning is to meaningfully transform data (i.e., to learn useful representations of the input data; representations that are closer to the expected output) [110].

An ML system functions as descriptive if it can use data to throw more light on what happened, predictive if it forecasts what is going to happen, and prescriptive if it can use data to suggest what can be done [65]. According to the MODA model, this cleaned and processed dataset is modeled as the descriptive model as it readily represents the input to the ML model. The ML algorithm learns from known input and output data to forecast outputs when fed unseen data as input. The data is split into three: the training, validation, and testing datasets. The training data is initially fed to the ML model to learn and adapt and the validation data is used as a check to verify how accurately the model is able to predict the known output data. Once the model is run several times and its performance is good enough, the ML engineer will then pass the test data as input to the ML model, i.e., the data set is unknown to the model, to make predictions that can be used as a reference for decision-making and problem-solving. The training data is the input data to the various ML models or algorithms. These models can be categorized into different types of learning, but the two major ones focused on in this course are supervised and unsupervised learning.

Supervised learning is a type of learning in ML that classifies data and makes accurate predictions by training the ML algorithm with a labeled dataset [91]. For example, one practical application is the classification of spam emails into a distinct folder in an email account's inbox. Supervised learning can further be divided into two: *classification* if the target outcomes are cate-

gories or labels and *regression* if the target outcomes are weight or height, scores or income [124]. Linear, Logistic, and Polynomial Regression are typical regression algorithms, while Support Vector Machines (SVM), Convolutional Neural Networks (CNN), Multilayer Perceptron (MLP), Naïve Bayes, Random Forrest, Decision Trees, and Linear Classifiers are standard algorithms for classification in ML.

Unsupervised learning is a type of learning in ML that uses algorithms to analyze and cluster unlabelled datasets [91]. These algorithms can accurately discover similarities, differences, and patterns hidden in the data without any human involvement, which is relevant in exploratory data analysis and image recognition [66]. K-Means Clustering is one example of an algorithm used for unsupervised learning. According to MODA, the predictive models covered by this course are the machine learning models resulting from supervised or unsupervised learning. The tools used for the predictive models are used to analyze and understand the performance and behavior of the ML models. They are bias, variance, and regularization.

Bias and variance are ML prediction errors and the goal of any ML model is to find a model that reduces the prediction errors on unseen data. *Bias* is the difference between the model's average prediction and the true value that is to be predicted. The bias indicates the trained model's ability to predict the true target. The lower the bias, the better a trained model is. *Variance* is a measure of the variability of the predicted values for a given input using the trained model. The lower the variance, the more precisely a trained model can make predictions. *Regularization* on the other hand is a concept that is implemented for the trade-off of the bias and variance and helps to reduce the prediction error. Some common regularization techniques include modifying the cost function, K-Fold Cross-Validation, and further modifying the ML algorithm [162].

Once training is complete, the model produces an output. The output data is visualized and analyzed to determine if the model is the best model to be used to solve a problem. For example, for a generic classification problem to determine if a person is suffering from Y disease, we want to build a model to indicate either Yes or No. Initial output is fed to the model to train the model. Then when the model is trained, the predicted output is compared with the actual output to determine if the ML model could accurately predict the output. Various statistical analyzes are carried out to determine how far or close the model is to the expected output. Based on the results, the model parameters can be adjusted. The model can be trained several times while adjusting

the key parameters that affect the algorithm's behavior. A feedback loop can be observed in the predictive model, and this loop will stop once the accuracy of the model is sufficiently good, and then the model can be finalized. If the final model is integrated into system design models to make predictions for the system that are realized by the system, then it serves as the prescriptive model.

5.9 ECSE 552 - Deep Learning

Deep Learning (DL) is a subset of machine learning, which comprises a neural network with three or more layers. These neural networks make an effort to mimic how the human brain functions, however, they fall far short of being able to match it, enabling it to "learn" from vast volumes of data. Additional hidden layers can help to tune and refine for accuracy even if a neural network with only one layer can still make approximation predictions [64]. The structure of the course taught at McGill covers areas such as an overview of mathematical background and basics of machine learning, deep feedforward networks, regularization for deep learning, optimization for training deep learning models, convolutional neural networks, recurrent and recursive neural networks, practical considerations, applications of deep learning, recent models and architectures in deep learning. Figure 5.9 shows the MODA representation of the course.

Deep learning differs from machine learning by the type of data that it works with and the methods in which it learns. Some of the data pre-processing required for machine learning, in general, is eliminated by deep learning. These algorithms are able to ingest and interpret unstructured data, such as text and images, and automate feature extraction, reducing the need for human subject-matter experts. For example, a set of photos of different pets can be categorized as "cat", "dog", "hamster", etc. Deep learning algorithms can decide which characteristics (e.g., ears), are most important for differentiating one species from another [64]. Similar to ML, the data in deep learning can either be selected from a wide range of available open-source datasets or generated from scratch, depending on the domain that one will want the algorithm to learn.

Before anything is done on the dataset to perform DL analysis on it, the engineer needs to understand the basic mathematical concepts needed to understand DL. These include concepts such as linear algebra, probability, and information theory. The engineer must also have some knowledge and understanding of the basics of ML and generally how ML algorithms are used to

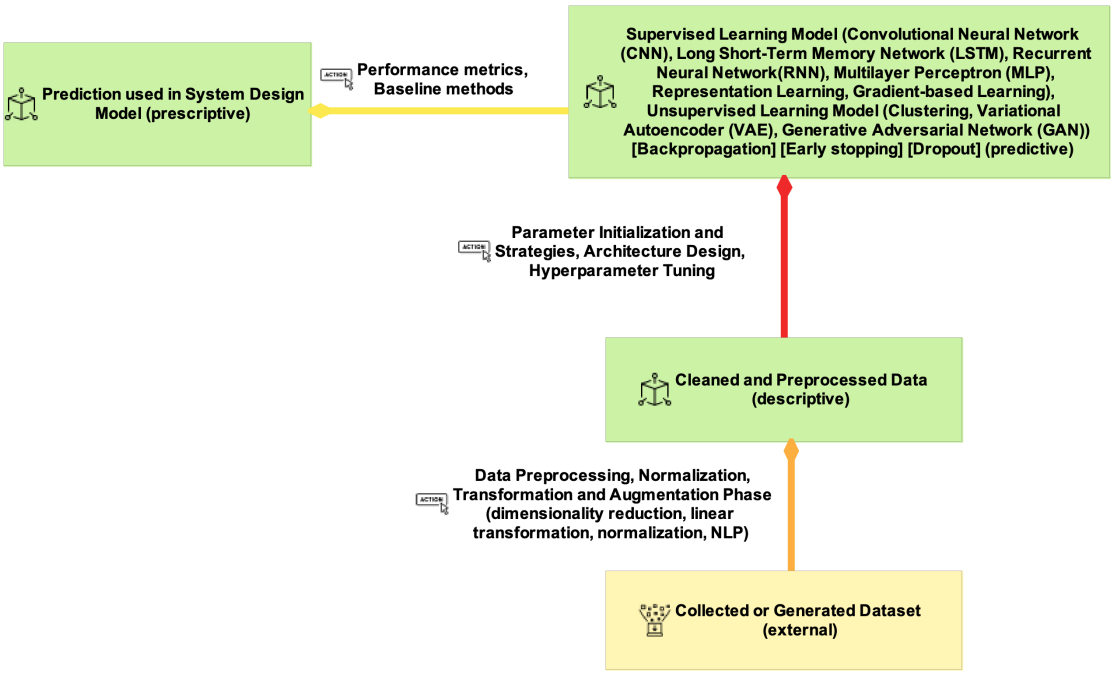


Figure 5.9: ECSE 552 MODA Diagram

make predictions and inform decisions. The data is first cleaned and preprocessed in preparation for the DL algorithms. After the data is cleaned, it is pre-processed and manipulated to fit the model. The data preprocessing techniques used in ML can be used for DL models. Techniques such as dimensionality reduction, normalization, transformation, and NLP, among others, are incorporated to prepare, process, and clean the dataset to make it viable for the learning algorithms.

Once the data is prepared, parameter initialization and strategies for DL can now be selected for the type of dataset at hand, the design of the architecture, and hyperparameter tuning techniques are put in place to prepare the model for prediction (i.e., Action E).

Parameter initialization and strategies are very critical to the model's performance and the right method needs to be selected. It defines the way to set the initial random weights for the DL algorithm. For example, the gradients and weights are initialized with a zero initialization approach, which causes the neurons to learn the same features during training. Any constant initialization strategy will actually perform horribly. If there are two hidden units in the neural network and the biases and weights are both initialized at 0, then the output of both hidden units in this network will be relu(x1, x2), if forward propagated with an input (x1, x2). As a result, the cost will be affected by both hidden units in exactly the same ways, leading to identical gradients. Thus,

throughout training, both neurons will develop in a symmetrical manner, successfully preventing distinct neurons from picking up different information [77].

The *architecture design* of a Neural Network (NN) relates to its overall structure in terms of how many units it has and how these components are linked to one another. The majority of neural networks are structured into layers, which are groups of units. The layers in most neural network topologies are organized in a chain pattern, with each layer being a function of the layer before it. The key architectural issues in these chain-based systems are the network depth and the width of each layer [58].

Hyperparameter tuning involves identifying the best values for a learning algorithm's hyperparameters to achieve optimal performance when applied to any given dataset. By finding the ideal combination of hyperparameters, the model's performance is maximized, resulting in improved results and reduced errors as measured by a predefined loss function. The process aims to fine-tune the algorithm to ensure the best possible outcome for a specific task or problem [97].

Depending on the problem domain, a DL algorithm is selected to be used to run predictions. If the engineer does not know which algorithm will be the best fit to make predictions, the data can be trained with a number of DL algorithms and the one that produces the best accuracy and precision can be selected as the best-fit algorithm to make predictions. DL can be classified into *supervised* and *unsupervised* learning. Some of the algorithms for supervised learning include Convolutional Neural Network (CNN), Long Short-Term Memory Network (LSTM), Recurrent Neural Network (RNN), Multilayer Perceptron (MLP), Representation Learning, and Gradient-based Learning, among others. Clustering, Variational Autoencoders (VAEs), and Generative Adversarial Networks (GANs), among others, are some examples of unsupervised learning. Some of the techniques used to better tune these algorithms include backpropagation, early stopping, and dropout.

Backpropagation [30] is widely employed in the training of feedforward neural networks. It is a key component of many popular deep learning algorithms, such as MLP and CNN. The goal of backpropagation is to adjust the weights of the neural network's connections in order to minimize the difference between the predicted output and the desired output. They are highly effective in calculating the gradient of the loss function concerning the weights of the network. This efficient computation of gradients enables the network to learn and adjust its parameters, leading to improved performance and better accuracy [163].

Early stopping and *dropout* are optimization techniques employed to reduce or prevent overfitting while maintaining the model's accuracy. The concept behind early stopping is to stop training before a model starts to overfit. On the other hand, the concept behind dropout involves temporarily dropping out (i.e., set to zero) a random subset of neurons during the training process. Both techniques help prevent a decline in the performance of the unseen data [58].

Just as in ML, the data set is also split into three in DL: training, testing, and validation dataset. The training and validation data are what is used in the initial training and hyperparameters are selected for the algorithm. Then, through the processes of gradient descent and backpropagation, the deep learning algorithm adjusts and fits itself for accuracy, allowing it to make predictions with increased precision. The output data is visualized and analyzed to determine if the model is the best model to be used to solve the problem. Performance metrics such as accuracy, precision, and heat maps among others can be used. Similar to ECSE 551, a feedback loop can be included in the predictive model as training and retraining will continue to occur until an optimal accuracy is reached.

After training a DL algorithm, it becomes capable of making predictions on unseen data (i.e., testing dataset). During the initial training, the validation dataset is fed to the trained deep learning algorithm, which processes the data through its layers to produce the final prediction. The model has learned from the training data and has adjusted its internal settings (weights and biases) to make accurate predictions based on the patterns it discovered during training. The model's performance is assessed by comparing its predictions to the expected outcomes in the validation dataset. If the outcomes closely match, it indicates that the model has successfully learned the underlying patterns and can make reliable predictions. At this stage, the model can be considered complete and ready for use, and as such the unseen data is passed to the model to make predictions. If the final model is integrated into system design models to make predictions for the system that are realized by the system, then it serves as the prescriptive model.

5.10 Discussion and Observations

In this section, the discussion provides insight into the MODA tool and how it is used to model each course. A detailed analysis of the results obtained from the study is also provided in the

observations, reporting the findings that emerged from the study and shedding light on potential areas that may needs further investigation.

5.10.1 Discussion

Using the course structure, models and key concepts of the courses provided by each course lecturer, we analyze and review each model to determine which aspects could be categorized as model, data, running software, socio-technical system, and action types according to the MODA framework. We then implement each course with the MODA tool, going through several iterations and gathering feedback from course instructors, professors in model-driven engineering, and the research group to gain different perspectives and suggestions for improvement. With the assistance of the MODA tool, changes and updates are made efficiently to the models during each iteration. The tool makes it much easier to model each course using the MODA framework and visually understand their contents. This tool is expected to greatly benefit the modeling community, providing a user-friendly way to build MODA models effectively.

5.10.2 Observations

Based on the study, it is observed that MODA has broad applicability, as each aspect of the course is analyzed and explained using the framework. The courses included in the exploratory study of this thesis cover the main parts of the MODA framework. The core data and aspects of each course used to teach computer science and engineering students the key concepts of SE and AI that they will need in the industry are represented by the MODA framework. The prototype tool introduced in this thesis is able to effectively model all data and model aspects covered in these courses using the MODA framework. However, there are several tools and technologies used in each course that are not directly supported by the MODA framework, but they are incorporated into the respective MODA diagrams by being listed within square brackets of each model and data element to which they apply. These tools and technologies clearly operate on data or models but they do not create data or models and hence cannot be mapped directly to data or models. The framework is nevertheless able to depict each aspect of a course which are model roles, data, action types, running software, and the socio-technical system at play. The MODA diagram provides a graphical representation of these courses such that a student can get an overview of what the course

entails, the tools, techniques, and programming languages that are required by just viewing the MODA model for the course.

Figure 5.10 depicts the heatmaps developed to aid in the analysis of the data obtained from the exploratory studies. Figure 5.10(a) shows a comparison of the selected courses against the elements (i.e., model, data, running software, and STS) discussed in the MODA framework and Figure 5.10(b) shows the respective actions against the selected courses in the exploratory studies.

(a) Courses x Elements

(b) Courses x Actions

Figure 5.10: Heat Map Showing the Occurrence of Elements and Actions in each Course

Using a scale of 0 to 1 in generating the heatmap for the elements, each cell represents the number of topics (i.e., concepts, tools, technologies, processes) covered by a course and is color-coded based on its value. The score is calculated by counting the number of topics for a specific model role in the MODA model of a course and then dividing it by the maximum number of topics from the descriptive, predictive, or prescriptive model for all courses. Topics in square brackets are not counted. In each course, there were aspects that are categorized as both descriptive and prescriptive. Among the courses, ECSE 439 had the highest number of topics represented as descriptive models, followed by ECSE 223, 321 / 326, 429, and others in descending order. Similarly, ECSE 439 had the highest number of topics represented as prescriptive models, followed by ECSE 326 / 321, 223 / 429, and others. As for predictive models, they were found in only three courses, with ECSE 551 having the highest number of topics, ECSE 552 being a close second, and ECSE 326 having the third-highest number of predictive models. As expected, the courses from the SE

minor focus more on descriptive and prescriptive models, with very little coverage of predictive models, whereas the courses from the Applied AI minor (ECSE 551 and 552) place more emphasis on predictive models.

The heat map examines the representation of the remaining elements in each course using the same [0,1] scale used for the models. However, the scoring is adjusted. If a particular element, such as running software, exists in a course, it is assigned a score of 1, as there was always at the most one present in each course. This scoring system also applies to the remaining elements (i.e., input data, output data, measured data, external data, and STS) where a score of 1 indicates that they are present in a course. Each course has running software except ECSE 439, 551, and 552. Instead of software applications, ECSE 439 implements software languages and ECSE 551 and 552 focus on ML models without implementing an application. All the courses indirectly represent a socio-technical system but the STS is not modeled in any of the courses because it is not emphatically implied. Similarly, measured data is not modeled in any of the courses. However, this is because the measurement of running software is not the focus at all for any of the examined courses. This points to a potential gap in the courses in the coverage of concepts important for models and data. Each course does include at least one type of data, i.e., input, output, or external data. Some courses may have two out of the three types, or even all three, but the modeling never shows the presence of measured data.

Similar to the models, each action is also represented by the number of occurrences in the course on a scale of 0 to 1, with 1 being mapped to the maximum number of topics for an action in any of the courses. The action used in every course that is analyzed in this study is generalization and calibration (i.e., Action D). ECSE 552 has the highest number of topics represented by Action type D, followed by ECSE 551. When it comes to Action type E (preparation for prediction), ECSE 552 has the maximum, while for Action type F (analysis, decision, and change), both 429 and 551 have the highest number of representations. As for Action type G (generation), 439 has the maximum number of representations. The remaining courses each have one representation. It is worth noting that measured data and STS are not given significant emphasis in the courses, which is why Action types C (measurement), I (enactment), and J (other interplay) are not utilized. Based on this, it can be concluded that the framework indeed was successful in modeling courses from McGill University and provides further evidence that the vast applicability claim is correct.

Additionally, the analysis uncovers elements of the MODA framework that may require further exploration or expansion beyond what is already defined by the authors of the framework. This is observed specifically for ECSE 551 and ECSE 552. It can be observed that there may be a need for refinement in the predictive model (i.e., similar to Action type G for prescriptive models) as several iterations of training need to be done to arrive at an optimal ML and DL model. A feedback loop hence also exists within the prescriptive model and can be in the form of user feedback on the performance of the model. The feedback received allows for iterative improvements of the ML/DL model and system design. This includes actions such as retraining the model, adjusting parameters, and incorporating user feedback. The objective is to consistently improve the model's performance and ensure it meets the system's goals and requirements. However, the MODA model only covers the feedback loop with a deployed system in a socio-technical context. It does not cover the feedback loop in the development of an ML/DL model (or software in general) if there is no deployed system in the loop. This has also been observed by L-MODA [73] which adds additional feedback loops to MODA. Extending the MODA tool to include L-MODA concepts would allow a more nuanced modeling of the courses in the exploratory studies.

5.11 Summary

This chapter presents the courses selected to analyze the framework in an exploratory study, discusses the various views and perspectives obtained from the analysis, and highlights the effectiveness and potential of the MODA framework in understanding and evaluating educational courses. Chapter 6 presents a review of the relevant literature to this thesis. It covers and summarizes existing studies, research papers, and scholarly articles that apply the MODA framework in several fields of interest.

6

Related Work

This chapter focuses on the review of related literature. Using Google Scholar as the main point of focus, a systematic literature review was concluded for this thesis. First, a search with the keyword *"moda" since 2019* was run and this generated 58,600 results in 0.10 seconds. The year 2019 was chosen since it was the first year a paper on the MODA framework had ever been published. This search stream was too broad and resulted in too many articles with many false positives. On January 11, 2023, an advanced search with the keyword *"moda framework" as the exact phrase anywhere in the article ranging from 2019 - 2023* was made on Google Scholar, and this generated 23 results. 8 papers were directly linked to the MODA framework, 9 papers had the abbreviation MODA but are not related to models and data, 2 papers were not related to the framework in any way, and the remaining 4 papers are not written in English. Table 6.1 outlines the resulting list indicating which papers are applicable to this thesis or not and why.

Table 6.1: List of MODA Papers Found from the Google Scholar Search

No	**Title of Paper**	**MODA Definition**	**Scope**
1	A Hitchhiker's Guide to Model-Driven Engineering for Data-Centric Systems	Models and Data	In Scope
2	Global Decision-Making Over Deep Variability in Feedback-Driven Software Development	Models and Data	In Scope
3	Conceptualizing Digital Twins	Models and Data	In Scope
4	Towards self-adaptable languages	Models and Data	In Scope
5	On reliability and flexibility of scientific software in environmental science: towards a systematic approach to support decision-making	Models and Data	In Scope
6	DataTime:A Framework to smoothly Integrate Past, Present and Future into Models	Models and Data	In Scope
7	AI-driven streamlined modeling: experiences and lessons learned from multiple domains	Models and Data	In Scope
8	Reasoning over Time into Models with DataTimes	Models and Data	In Scope
9	Decision-Making Framework for Evaluating Physicians' Preference Items Using Multi-Objective Decision Analysis Principles	Multi-Objective Decision Analysis Principles	Not Related
10	Homeland Security and Emergency Management Grant Allocation Multi-objective Benefit-Cost Methodology	Multi-Objective Decision Analysis Principles	Not Related

11	Evaluating the Impact of Culture on Customer Satisfaction for FMS Projects	Multiple Objective Decision Analysis	Not Related
12	Assessing Engineering Resilience for Systems with Multiple Performance Measures	Multiple Objective Decision Analysis	Not Related
13	The Dynamics of Multidimensional Poverty in a Cohort of Irish Children	Multidimensional Overlapping Deprivation Analysis	Not Related
14	Measuring Child Poverty in Jakarta Metropolitan Area Using a Multidimensional Perspective	Multidimensional Overlapping Deprivation Analysis	Not Related
15	I Don't Care Who You Are: Adult Respondent Selection Does Not Alter Child Deprivation Estimates	Multiple Overlapping Deprivation Analysis	Not Related
16	Prevalence and correlates of multidimensional child poverty in India during 2015–2021: A multilevel analysis	Multiple Overlapping Deprivation Analysis	Not Related
17	Unfolding the prospects of computational (bio)materials modeling	Modeling Data	Not Related
18	Urban mobility:Leveraging machine learning and data masses for the building of simulators		Not Related
19	"I like relaxing on the trees when the leaves are falling": Children's experiences of relaxation in Early Childhood Education and Care (ECEC)		Not Related

20	Mobilité urbaine : apprentissage automatique pour la construction de simulateurs à l'aide de masses de données		Not in English
21	Framework para formulação de problemas para inovação pelo design de equipamentos médico-hospitalares		Not in English
22	Framework para implementação de estratégias de inovação pelo design no processo de desenvolvimento de produtos de moda em empresas de confecção do		Not in English
23	Der digitale Fußabdruck, Schatten oder Zwilling von Maschinen und Menschen		Not in English

6.1 Review of Related Work

Combemale et al. [31] were the pioneers of the MODA framework. They present a framework for conceptual referencing that aims to provide a strategy that is model-driven and also focuses on data, for incorporating various models and their affiliated data over the entire life-cycle of socio-technical systems. This framework directly correlates the diverse actions that can be performed on three roles of models in line with the four types of data: input/output data, measured data, and external data. The authors claim that the framework supports the characterization of conventional software development processes, systems, and technologies by (a) explaining how the different roles of models (i.e., descriptive, predictive, and prescriptive) integrate with the various sources of data and (b) outlining the necessary actions that connect data and models. In sectors such as transportation, energy, and healthcare, numerous systems are considered socio-technical, given the human, organizational, and social factors considered during the system life cycle. MODA aims to support the entire life cycle of these systems that affect an extensive range of organizations in various

communities and stakeholders. The authors claim that MODA is expressive enough to characterize and generalize cutting-edge engineering approaches and technologies utilized throughout the life cycle of a system, even if it was developed to describe a variety of novel socio-technical systems. The authors pose pertinent research questions and discuss how the framework can also assist in identifying unresolved challenges in the model-driven engineering community. This thesis applies the concepts introduced by the authors and further validates and expands on this prior knowledge by developing a metamodel and a prototype tool that can be used to build MODA models.

The software industry is undergoing radical changes in the way that software is developed and the features it provides consumers. A feedback-driven plan can be used to reduce the uncertainty over how an application must evolve. The fundamental concept is to gather data from the software already in use, analyze that data, and apply the findings as a blueprint for future software development. Kienzle et al. [75] present and outline the Multi-Plane Models and Data (MP-MODA) framework, which expands on the MODA framework. In a feedback-driven software development process where feedback loops are intended to reduce uncertainty, this new framework provides automation and tool support for a multicriteria decision-making process involving numerous stakeholders. For example, a company that is implementing a new phone application decides to define three planes addressing business, software development, and usability concerns, respectively. Each plane is described by a MODA model. The experts of each plane choose a course of action from the options available to them within that plane, and because decisions are documented to be dependent on one another (both intra- and interplane), the experts of each plane can also evaluate how their choices will affect those of other planes. For example, the software expert may realize that a specific application feature must be redesigned to suit a target market. MP-MODA then alerts them that changing the design can impact the choices made by the business experts; hence they need to meet and look at the available options. The business experts were already aware of the issue but were looking at a much more general target market. The main point now is that a specific demographic may use the application more than others if the design is tailored to suit them. They can make the best decision using MP-MODA to help them identify their intra-plane dependencies. MP-MODA may point to the fact that updating the design has an impact both positive and negative, and using information from prior iterations provided through the feedback loop, it estimates that adapting this new design may take time to develop, which can then impact the choice of release date decided

by the business experts. Due to their understanding of these inter-plane (i.e., across different domains) and intra-plane (i.e., within their own domain) dependencies, the experts were able to reach a consensus that the best course of action is not to update the feature just yet but to prioritize the work of the development team so that with each new release, they make some changes until gradually changing the entire application's design. To sum up, if decisions need to be made while working on the next software release, MP-MODA can help assess the proposed change's overall effects and involve the appropriate stakeholders in the decision-making process. MP-MODA and the concepts in this thesis differ as MP-MODA extends the MODA framework by incorporating multiple planes or layers of models and data, and this thesis expands and further validates the existing MODA framework with a prototype tool and metamodel.

Software languages, similar to natural languages, are continually evolving in response to new concepts and relationships that arise, often to address specific needs in particular application domains. A Self-Adaptable Language (SAL) is a software language that simplifies the design and execution of feedback loops and trade-off analyses. It allows users to focus on delivering specific services for their software system without having to worry about implementing complex feedback mechanisms from scratch. SAL also provides flexibility to customize the feedback loop based on feedback from the modeling environment. Self-adaptation requires a feedback loop to respond to changes and analyze trade-offs, considering factors like energy, time, cost, and quality. There is a growing need for systems that can consider multiple factors, such as energy, time, cost, and quality, and cater to a wide range of users. However, modeling and programming such systems involve complex activities that require different software languages and involve various stakeholders in the software engineering process. Jouneaux et al. [73] introduce the concept of SAL and explore the integration of a feedback loop into language semantics, leading to the development of self-adaptable virtual machines, and outline a roadmap highlighting the main expected features throughout the lifespan of SALs. To better understand the key concepts of SALs, the authors introduce a conceptual framework called L-MODA (Languages, Models, and Data), which helps view how different parts of a system are connected (i.e., the running software system and its data, the modeling environment with its data, models, and the self-adaptable language, and the language definition environment). By using L-MODA, the authors explore how these elements relate to each other and contribute to self-adaptation in a software system. After examining three examples of

Self-Adaptable Virtual Machines (VMs), the authors observed that the languages' semantics were successfully adapted based on the context. The outcomes of the VM experiments indicated that the effectiveness of the approach relies on factors such as the expressiveness of the target language. Generally, the authors explore the benefits and challenges pertaining to SAL and the potential impact on software development practices. L-MODA is another extension of the MODA framework that integrates NLP with models and data, but this thesis expands and further validates the existing MODA framework with a prototype tool and metamodel.

A Digital Twin (DT) replicates an actual system that is constantly updated with real-time data throughout its life cycle and can interact with and influence the actual system simultaneously. DTs can be developed for various reasons, including developing, designing, simulating, analyzing, and operating non-digital systems to understand, monitor, and optimize the actual system. Many disciplines use DTs to better understand, regulate, and optimize the behavior of complex systems, either at design time or during runtime. As a result, DTs are becoming a significant software engineering technique for managing the complexity of software engineering in a wide range of application areas. The complexity of systems is increasing rapidly, and models have become critical for understanding them. As a result, today's advanced systems are built using models from several engineering areas. As a result, their DTs must also incorporate diverse heterogeneous models to address the various features of the system. These models could be engineering models, software models, or DSL models. These models must be linked to data acquired from the actual system and its surroundings to make sense of them during the design and runtime of a system. The models and data make it possible to develop services related to the actual system. Eramo et al. [37] used the MODA framework to implement a conceptual reference framework for DT. The authors do not describe specific tools or technologies for implementing DTs by proposing a conceptual framework but rather categorize artifacts' many roles and relationships on a conceptual level. The actual system generates data connected to many parts of the system. The DT takes this data and employs models to perform various operations on the actual system and its environment. According to the MODA framework [31], models can play three roles in a DT: descriptive model, predictive model, and prescriptive model. The authors investigate the technical issues of building a DT, such as data management, integration, and visualization. They also discuss some of the unanswered research questions surrounding digital twins, such as how to increase the precision of

digital twin models, how to merge digital twins with other technologies (i.e., such as augmented and virtual reality), and how to cope with the ethical and legal difficulties surrounding the use of digital twins. Research and practice have resulted in various DT implementations. The authors identified a collection of DT applications involving several stages of a system life cycle, specifically design, maintenance, manufacturing, and utilization [129]. The list of DT applications offered is not exhaustive, but it demonstrates the usefulness of the presented framework, and the described DT applications demonstrate the appropriateness of its foundation, the MODA framework. The approach used in this thesis and used in implementing a DT are similar as they both focus on and utilize the core foundations and building blocks of the MODA framework. However, this thesis introduces a metamodel and prototype tool for MODA.

Scientific software, such as simulation models, plays a crucial role in decision-making in environmental science. However, to be able to use them effectively in decision-making, certain conditions are required. Although scientific software is essential for addressing environmental issues like climate change, its complexity and resource-intensive nature make them less suitable for immediate decision-making as the primary focus is research and development. In the decision-making process, stakeholders aim to bring about change within the organization or system under consideration. For example, in climate change, the government can make decisions by implementing laws and policies. The goal is to adapt simulation models, initially developed for research purposes, to be more suitable for decision-making. This involves improving their execution speed, accuracy, complexity, and flexibility to explore different scenarios. By making these enhancements, the models can provide faster, more reliable, and more versatile insights for decision-makers. Scientific models aim to describe the world and help make decisions based on projections. However, to use them effectively in decision-making, there is a need to increase the execution speed, thus making the decision-making process faster. Sallou [116] in their thesis aims to modify scientific models for decision-making in environmental science by using approximate computing techniques while maintaining their reliability. To ensure the proper use and validity of scientific models, the general features that apply to all models are first defined, as they help determine if the models can generate reliable projections and identify the specific conditions in which they produce reliable results. Once the simulation models are customized, new scenarios are explored using the MODA framework [31], which helps understand and visualize how models and data are integrated into a cyber-physical system and

how they interact with each other according to their respective roles. The author also explored validating scientific models and proposed a V-Model that emphasizes the importance of using appropriate tools to develop reliable scientific software. Sallou's thesis focuses on the challenges of using scientific software to support decision-making in environmental issues and demonstrates how the loop aggregation technique can address these challenges. By understanding the role of scientific models in decision-making and ensuring their validation, better support for decision-making in environmental issues can be achieved. First, the author looks at the different kinds of models used in software: scientific models, engineering models, and machine learning models (also known as empirical models). In the context of cyber-physical systems, the author compares these models and observes that scientific and engineering models have many similarities, despite their differences. In contrast, engineering models with a more prescriptive nature incorporate specific instructions and extensive knowledge about the real world. This led the author to view models not just as different types, but rather in terms of the roles they fulfill, and also suggested applying software- and engineering-oriented techniques to scientific models as well. The MODA framework highlights the significance of a systematic approach to integrating different models based on their roles and the associated data. In Sallou's thesis, the MODA framework improves the decision-making process by modifying scientific models to make them more predictive. It also emphasizes the importance of collaboration between different modeling communities, such as software engineering and environmental science, to ensure the effective integration of models in a socio-technical system. The goal is to enhance the efficiency and relevance of model integration for better decision-making. In Sallou's thesis, the MODA framework is used to help understand and visualize how models and data are integrated into a cyber-physical system and how they interact with each. This thesis also incorporates the MODA framework to help understand and visualize how models and data are applied in a socio-technical system (i.e., education in the case of this thesis).

Models at runtime have initially been explored for adaptive systems, but they are now recognized as important for developing complete digital twins. However, using models at runtime for this purpose brings new challenges, such as seamlessly interacting with different time states of the system. One possible approach to address this is to integrate temporal, spatial, and predictive models within the concept of digital twins (i.e., digital twins are based on the MODA framework). This would allow digital twins to expand their capabilities to include past, present, and future

aspects of the system. To address the need for capturing system states based on both time and space, Lyan et al. [84] [85] propose a new framework called DATATIME. This framework represents the system state as a directed graph, where nodes and edges have independent local states. The framework provides a unified interface to query past, present, and predicted future states of the system. It optimizes the storage of past states, retrieves real-time sensor data, and continuously learns predictive models for future states. The framework has two main parts. The first part is for designers who implement changes for users, and it includes the spatial model and predictors configuration. The second part is the digital twin, which allows users to work with the framework and analyze the spatial model over time. There is also a third component that represents the effort required to use the framework in a specific project. The authors also tested the framework in a real-world urban transportation system in Rennes, France, and evaluated its performance. DATATIME allows experts in specific fields to combine past, present, and future data within existing information systems. However, using DATATIME currently requires knowledge of the Scala programming language, which may not be common among IT departments in organizations such as Keolis (i.e., a multinational transportation company that operates public transport systems). To make it easier to use, the authors plan to provide simplified languages that capture the main concepts of DATATIME, which will enable easier integration of new data sources, editing of network configurations, and smoother data analysis processes. The aspect of DATATIME that is implemented with a DT makes this approach also similar to what is presented in this thesis, as they both focus on and utilize the core foundations and building blocks of the MODA framework. However, this thesis goes a step further by introducing a metamodel and a prototype tool specifically designed for working with MODA.

With time, urban transportation networks will become more closely associated with data and computing technologies. Not only is the use of sensors becoming easier and less expensive as communication technologies and hardware production methods improve but cities and operators are also eager to optimize their transportation networks, keeping them efficient and attractive in upcoming smart cities. Lyan et al. [83] aim to provide domain experts with methods for integrating heterogeneous data, data analysis capabilities, and predictive models in monolithic frameworks. In Lyan's thesis, they propose four contributions that, when combined, support all the eight features of their bus network approaches in the state of the art. These features include the need

for domain knowledge, heterogeneous data sources, use of real data, spatiotemporal data analysis, multi-targets predictions, evolution, scalability, and portability. The authors propose a software-engineered solution for data-centric decision models for urban public transportation networks, with a real-world application on Rennes, France's bus network. The first contribution focuses on Urban Public Transportation Network (UPTN) data quality issues. The second uses large-scale real-world data to assess the impact of exogenous factors on bus speed. The third one proposes a fine-grained prediction approach for predicting bus speed using real-world data. The fourth and final one proposes a framework for bus network operators that provides spatiotemporal data analysis and prediction tools. They do not emphatically use the MODA framework but just reference it as one of the existing works that can be leveraged to help tackle UPTN design problems.

Model-driven practices have been researched and practiced based on the two qualities they offer, automation and abstraction. Automation allows for the desired implementation by generating code automatically, and abstraction makes it easier to create similar systems. However, there is an argument that model-driven practices have been scarcely adopted because their benefits do not outweigh the costs. AI techniques can improve the benefits while reducing the costs of adopting these practices. Still, AI techniques are also not quite used and are generally used in a few modeling lifecycle activities. This prompts a call for broader applications of AI techniques in modeling. These developments highlight the need to leverage upcoming AI techniques in modeling and embrace different models that work together with different types of data. Through their modeling journey, Sunkle et al. [125] address both concerns to some extent by first using models to generate code for business-critical enterprise applications and using models to analyze and help in enterprise problem-solving, which involved numerous industrial case studies where the use of AI techniques was gradually introduced and increased. The authors in this paper present several case studies from various domains, including transportation, energy, and health care, in which AI-driven modeling techniques were used to simplify the modeling process and improve model performance. They present a description that details how they applied and continue to apply various AI techniques to help enhance individual solutions. Their perspective on the case studies presented in the paper shows examples of how AI techniques can be used across multiple modeling activities and how various artifacts and data can be utilized in modeling. They called this approach AI-driven streamlined modeling, and the case studies highlight the key benefits of a diverse set of artifacts

and AI techniques and a discussion of their industrial application contexts. While the MODA framework discusses the nature of models and their interactions with data, it does not explicitly address the specific modeling activities where these interactions occur. The authors highlight and discuss the MODA representations of all the case studies to clarify the modeling activities where descriptive, prescriptive, and predictive models interact with data. The approach presented in this paper is similar to this thesis as both involve conducting a study to provide additional validation for the MODA framework. The authors conclude that for modeling to transition to the MODA framework in terms of interpretation and implementation, the community needs to adopt AI techniques in activities relevant to available data and artifacts. The case studies and their representation as MODA instantiations can aid in corroborating the traditional modeling activities that are enhanced with AI techniques and perform the roles of prescriptive, descriptive, and predictive models.

6.2 Summary

This chapter provides an overview of the work completed in this domain. It discusses previous work by other authors and explains how they applied the MODA framework in several domains and research. Most of the works done by other authors expand on the MODA framework (e.g., L-MODA), analyze and explore the applicability of the MODA framework in other domains, or apply the concept of the framework in a problem domain. Chapter 7 concludes this thesis and discusses future MODA improvements.

7

Conclusion

The thesis is concluded in this chapter with a summary of our contributions in Section 7.1 and a discussion of future work opportunities in Section 7.2.

7.1 Contributions and Findings

The MODA framework, though a conceptual reference framework, provides good insight into the various roles models and data play in software development and the deployment and operation of socio-technical systems. This framework helps in developing better models, improving data collection (i.e., by indicating which type of data will be relevant) and analysis methods, and enhancing decision-making and understanding in various socio-technical systems such as science, engineering, business, and social sciences, and how best models and data can be employed in a holistic manner to represent these systems. Initial work on the MODA framework outlined its architecture and vast

applicability, with no tool support to help practitioners build MODA models. Also, though the vast applicability of the framework is claimed by the authors of the framework [31], only preliminary evidence is given to support it.

In this thesis, we present a DSL and tool support for the MODA framework and an evaluation of the proof-of-concept tool with the help of an education-based analysis. As there is currently no existing metamodel for the framework, we implement a metamodel to accurately define the elements of the MODA framework. Then we build a proof-of-concept editor that supports the framework and graphically visualizes how models and data work together in a selected socio-technical system. Furthermore, in order to validate the vast applicability of the MODA framework, we choose an education-based analysis (i.e., courses offered in two minor programs at McGill University: the Software Engineering Minor degree and the Applied AI Minor degree), and model the technologies, tools, and techniques used in key select courses with the prototype editor tool we built. Hence we further validate if MODA will allow us to think and reason about these courses in terms of the types of models and data represented in the course. Finally, observations and analysis are carried out to try and identify areas that the MODA framework does not capture and how effective the framework will be in modeling many other situations.

Overall, the MODA framework indeed allows for the modeling of all concepts, tools, and technologies covered by the selected courses. The MODA framework could be improved by directly supporting the modeling of tools and technologies that operate on data or models but do not directly create data or models. Currently, they are implicitly identified with square brackets in the MODA models. Furthermore, the MODA framework focuses on the feedback loop with the socio-technical system but does not capture well feedback loops that exist in the development of machine learning models (or software more generally) as is the case in the L-MODA [73] extension of MODA. Extending the MODA tool to include concepts from L-MODA would allow for a more comprehensive and nuanced modeling of the courses in future studies.

7.2 Future Work

The MODA framework can further be explored, improved, and validated as it has great potential in the modeling and software engineering domain. In addition to the further work and research

carried out in this thesis, the following can be done to better improve the MODA framework and also make it readily available in the industry to be utilized in the SDLC.

Firstly further research can be done in other socio-technical systems such as health, energy, law, and business among others, to further validate and investigate the vast applicability of the MODA framework.

To improve the tool implemented in this thesis and make it readily available for external use, a web-based system can be implemented specifically for the MODA framework. This web-based system can utilize Sirius web as a platform, allowing designers and developers to build their own MODA models in the respective domains they need for their software development process. By providing this capability, the system enables the creation of data-centric and model-driven software systems that are tailored to the specific requirements of each domain.

This thesis can be expanded with further research in education and other socio-technical systems with the newly proposed conceptual MP-MODA [75] and L-MODA frameworks [73]. This can be done to explore whether the current tool has the capability to model these variations or if there will be a need to extend the tool introduced in this thesis.

Furthermore, no user study was conducted in this thesis hence as to whether the implemented design of the prototype tool is appropriate or not is not known. A user study can be conducted in the future to get opinions about the tools and ways to improve as well.

Finally, given some input, the automatic creation of MODA models can be incorporated into the prototype such that when given a socio-technical system the prototype will be able to seamlessly predict a MODA model that will help analyze the type of data and model roles that can be incorporated in a software system before development. The specific input data needed may depend on the implementation and requirements of the prototype, and this can be further researched to know exactly what to present as input and its expected output.

Bibliography

[1] *IEEE Standard for System and Software Verification and Validation*. 2012. ISBN 978-0-7381-7268-2. doi: 10.1109/IEEESTD.2012.6204026.

[2] UML, 2023. https://www.uml.org/.

[3] AssertJ - Fluent Assertions Java Library, 2023. https://assertj.github.io/doc/.

[4] Docker, 2023. https://www.docker.com/.

[5] Infer, 2023. https://fbinfer.com/.

[6] JUnit 5, 2023. https://junit.org/junit5/.

[7] SonarQube, 2023. https://docs.sonarqube.org/latest/user-guide/concepts/.

[8] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley series in computer science and information processing. Addison-Wesley, 1983. ISBN 9780201000238. URL https://books.google.ca/books?id=k8pQAAAAMAAJ.

[9] Airship. Build A Better Problem Statement For Software Development, 2021. https://rb.gy/p3bba.

[10] Altexsoft. Extreme Programming: Values, Principles, and Practices, 2021. https://www.altexsoft.com/blog/business/extreme-programming-values-principles-and-practices/.

[11] Altexsoft. System Documentation, 2023. https://rb.gy/n6gey.

[12] Jafar Alzubi, Anand Nayyar, and Akshi Kumar. Machine Learning from Theory to Algorithms: An Overview. *Journal of Physics: Conference Series*, 1142:012012, 11 2018. ISSN 1742-6588. doi: 10.1088/1742-6596/1142/1/012012.

[13] AWS Amazon. What Is An API (Application Programming Interface)?, 2023. https://rb.gy/chr5j.

[14] Scott Ambler. Personas: An Agile Introduction, 2022. https://rb.gy/33kh1.

[15] Daniel Amyot and Gunter Mussbacher. User Requirements Notation: The First Ten Years, The Next Ten Years. *Journal of Software (JSW)*, 6(5):747–768, 2011. doi: 10.4304/jsw.6.5.747-768.

[16] Mussbacher Gunter Amyot, Daniel. Introduction to Use Case Maps, 2001. https://www.itu.int/itudoc/itu-t/com17/urn/urnp5_pp7.ppt.

[17] Hathaway Angela and Hathaway Thomas. *Requirements Elicitation Techniques - Simply Put! Helping Stakeholders Discover and Define Requirements for IT Projects*. CreateSpace Independent Publishing Platform, 2016.

[18] Kleppe Anneke. *Software Language Engineering*. Pearson Education, 2008. ISBN 9780321606464, 0321606469.

[19] Van Lamsweerde Axel. *Requirements Engineering From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[20] Sarafadeen Ibrahim Ayomide. How to Write Code Documentation, September 2022. `https://rb.gy/ws72v`.

[21] Ravi Bandakkanavar. Software Requirements Specification Document with Example, 2023. `https://rb.gy/ypkc0`.

[22] Kent Beck. *Test Driven Development By Example (Addison-Wesley Signature)*. Addison-Wesley Longman, Amsterdam, 2002. ISBN 0321146530.

[23] Behat. Writing Features - Gherkin Language, 2023. `https://rb.gy/lpfna`.

[24] Miro Blog. A Simple Guide to Using and Creating a Context Diagram, 2023. `https://rb.gy/75n5z`.

[25] Gregor v. Bochmann. Goal Modeling and GRL, 2009. `https://rb.gy/ya9xj`.

[26] G. Bonaccorso. *Machine Learning Algorithms*. Packt Publishing, 2017. ISBN 9781785884511. URL `https://books.google.ca/books?id=_-ZDDwAAQBAJ`.

[27] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-Driven Software Engineering in Practice: Second Edition. *Synthesis Lectures on Software Engineering*, 3:1–207, 3 2017. ISSN 2328-3319. doi: 10.2200/S00751ED2V01Y201701SWE004.

[28] Jean-Michel Bruel, Benoit Combemale, Ileana Ober, and Hélène Raynal. MDE in Practice for Computational Science. *Procedia Computer Science*, 51:660–669, 2015. ISSN 18770509. doi: 10.1016/j.procs.2015.05.182.

[29] G. Carullo. *Implementing Effective Code Reviews: How to Build and Maintain Clean Code*. Apress, 2020. ISBN 9781484261613. URL `https://books.google.ca/books?id=-H2VzQEACAAJ`.

[30] Y. Chauvin and D.E. Rumelhart. *Backpropagation: Theory, Architectures, and Applications*. Developments in Connectionist Theory Series. Taylor & Francis, 2013. ISBN 9781134775811. URL `https://books.google.ca/books?id=B71nu3LDpREC`.

[31] Benoit Combemale, Jörg Kienzle, Gunter Mussbacher, Hyacinth Ali, Daniel Amyot, Mojtaba Bagherzadeh, Edouard Batot, Nelly Bencomo, Benjamin Benni , Jean-Michel Bruel, Jordi Cabot, Betty H C Cheng, Philippe Collet , Gregor Engels, Robert Heinrich, Jean-Marc Jézéquel, Anne Koziolek, Ralf Reussner, Houari Sahraoui, Rijul Saini, June Sallou, Serge Stinckwich, Eugene Syriani, and Manuel Wimmer. A Hitchhiker's Guide to Model-Driven Engineering for Data-Centric Systems. *IEEE Software*, 38(4):71–84, 2021. doi: 10.1109/MS.2020.2995125. `https://doi.org/10.1109/MS.2020.2995125`.

[32] Developers. Android Studio, 2023. https://developer.android.com/studio.

[33] Claire Drumond. What is Scrum and How To Get Started: A Guide To Scrum: What It Is, How It Works, and How To Start, 2023. https://www.atlassian.com/agile/scrum.

[34] Paul Duvall, Steven Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Signature Series. Pearson Education, 2007. ISBN 9780321630148. URL https://books.google.ca/books?id=PV9qfEdv9L0C.

[35] Eclipse. Ecore Tools, 2023. https://www.eclipse.org/ecoretools/overview.html.

[36] Susan Entwisle, Sita Ramakrishnan, and Elizabeth Kendall. *Model-Driven Exception Management Case Study*, pages 153–173. IGI Global, 2010. doi: 10.4018/978-1-60566-731-7.ch012.

[37] Romina Eramo, Francis Bordeleau, Benoit Combemale, Mark van den Brand, Manuel Wimmer, and Andreas Wortmann. Conceptualizing Digital Twins. *IEEE Software*, 39:39–46, 3 2022. ISSN 0740-7459. doi: 10.1109/MS.2021.3130755.

[38] Wiegers Karl Eugene and Beatty Joy. *Software Requirements*. Microsoft Press, 3rd edition, 2013.

[39] Liliana Maria Favre. *Non-Mobile Software Modernization in Accordance With the Principles of Model-Driven Engineering*, pages 29–60. 2021. doi: 10.4018/978-1-7998-6463-9.ch002.

[40] Geek for Geeks. ML — Data Preprocessing in Python, 2023. https://www.geeksforgeeks.org/data-preprocessing-machine-learning-python/.

[41] Eclipse Foundation. Xtext-Language Engineering Made Easy, 2023. https://www.eclipse.org/Xtext/index.html.

[42] Eclipse Foundation. Acceleo, 2023. https://www.eclipse.org/acceleo/.

[43] Eclipse Foundation. ATL - A Model Transformation Technology, 2023. https://www.eclipse.org/atl/.

[44] Eclipse Foundation. Ecore - Eclipsepedia, 2023. https://wiki.eclipse.org/Ecore.

[45] Eclipse Foundation. Eclipse Modelling Framework, 2023. https://www.eclipse.org/modeling/emf/.

[46] Eclipse Foundation. Eclipse IDE, 2023. https://www.eclipse.org/downloads/.

[47] Eclipse Foundation. Object Constraint Language, 2023. https://projects.eclipse.org/projects/modeling.mdt.ocl.

[48] Eclipse Foundation. Sirius, 2023. https://www.eclipse.org/sirius/.

[49] Eclipse Foundation. Sirius Modeling Project, 2023. https://www.eclipse.org/sirius/doc/user/general/Modeling%20Project.html.

[50] Eclipse Foundation. Xtext, 2023. https://www.eclipse.org/Xtext/.

[51] Eclipse Foundation. Xtext - 15 Minutes Tutorial, 2023. https://www.eclipse.org/Xtext/documentation/102_domainmodelwalkthrough.html.

[52] Yvette Francino. User Story, 2023. https://www.techtarget.com/searchsoftwarequality/definition/user-story.

[53] Philip Gerlee and Torbjörn Lundh. *Scientific Models*. Springer International Publishing, 2016. ISBN 978-3-319-27079-1. doi: 10.1007/978-3-319-27081-4.

[54] Gerrit. Gerrit Code Review, 2023. https://www.gerritcodereview.com/.

[55] Git. Git – Fast Version Control, 2023. https://git-scm.com/.

[56] GitHub. Github, 2023. https://github.com/.

[57] GitLab. What is A Code Review?, 2023. https://about.gitlab.com/topics/version-control/what-is-code-review/.

[58] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[59] Guru99. What is Requirements Traceability Matrix (RTM) in Testing?, 2023. https://www.guru99.com/traceability-matrix.html.

[60] Markus Gärtner. *ATDD by Example: A Practical Guide to Acceptance Test-Driven Development*. Addison-Wesley Signature Series (Beck). Addison-Wesley Professional, 1st edition, 2012. ISBN 978-0-321-78415-5.

[61] Robert Heinrich, Reiner Jung, Christian Zirkelbach, Wilhelm Hasselbring, and Ralf Reussner. *An Architectural Model-Based Approach to Quality-Aware DevOps in Cloud Applications*, pages 69–89. Elsevier, 2017. doi: 10.1016/B978-0-12-805467-3.00005-3.

[62] J.L. Hodges. *Software Engineering from Scratch: A Comprehensive Introduction Using Scala*. Apress, 2019. ISBN 9781484252062. URL https://books.google.ca/books?id=Re62DwAAQBAJ.

[63] IBM. Uml Constraints, 2023. https://www.ibm.com/docs/en/dma?topic=elements-uml-constraints.

[64] IBM. What is Deep Learning?, 2023. https://www.ibm.com/topics/deep-learning.

[65] IBM. What is Machine Learning?, 2023. https://www.ibm.com/topics/machine-learning.

[66] IBM. Unsupervised Learning, 2023. https://www.ibm.com/topics/unsupervised-learning.

[67] Ambysodt Inc. Introduction To Test Driven Development (TDD), 2022. https://agiledata.org/essays/tdd.html.

[68] Gradle Inc. Gradle Build Tool, 2023. https://gradle.org/.

[69] ISO/IEC/IEEE. ISO/IEC/IEEE International Standard - Systems and Software Engineering, 2010. ISO/IEC/IEEE 24765:2010(E). pp. vol., no., pp. 1–418, 15 Dec. 2010.

[70] Itemis. YAKINDU Statechart Tools, 2023. https://rb.gy/9z44j.

[71] ITU-T. Z.151 : User Requirements Notation (URN) - Language Definition, 2023. https://www.itu.int/rec/T-REC-Z.151/en.

[72] Dick Jeremy, Hull Elizabeth, and Jackson Ken. *Requirements Engineering*. Springer-Verlag, 4th edition, 2017.

[73] Gwendal Jouneaux, Olivier Barais, Benoit Combemale, and Gunter Mussbacher. Towards Self-Adaptable Languages. pages 97–113. ACM, 10 2021. ISBN 9781450391108. doi: 10.1145/3486607.3486753.

[74] Bray Ian K. *An Introduction to Requirements Engineering*. Addison-Wesley, 2002.

[75] Joerg Kienzle, Benoit Combemale, Gunter Mussbacher, Omar Alam, Francis Bordeleau, Lola Burgueno, Gregor Engels, Jessie Galasso, Jean-Marc Jézéquel, Bettina Kemme, Sébastien Mosser, Houari Sahraoui, Maximilian Schiedermeier, and Eugene Syriani. Global Decision Making Over Deep Variability in Feedback-Driven Software Development. pages 1–6. ACM, 10 2022. ISBN 9781450394758. doi: 10.1145/3551349.3559551.

[76] Adam Kolawa and Dorota Huizinga. *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press, 2007. ISBN 978-0-470-04212-0.

[77] Katanforoosh Kunin. Initializing Neural Networks, 2019. `https://www.deeplearning.ai/ai-notes/initialization/index.html`.

[78] H.S. Lahman. *Model-Based Development: Applications*. Pearson Education, 2011. ISBN 9780132757188. URL `https://books.google.ca/books?id=KeGgvtOaeCgC`.

[79] Vogel Lars. Eclipse Modeling Framework (EMF) - Tutorial, 2007 - 2023. `https://www.vogella.com/tutorials/EclipseEMF/article.html`.

[80] Edward A. Lee. Modeling in Engineering and Science. *Communications of the ACM*, 62: 35–36, 12 2018. ISSN 0001-0782. doi: 10.1145/3231590.

[81] Isabella Lindgren. Transformations, Scaling and Normalization, 2019. `https://rb.gy/o609d`.

[82] Liu Liping. *Requirements Modeling and Coding: An Object-Oriented Approach*. World Scientific Publishing Company, 2020.

[83] Gauthier Lyan. *Urban Mobility : Leveraging Machine Learning and Data Masses for the Building of Simulators*. Theses, Université Rennes 1, September 2021. URL `https://theses.hal.science/tel-03520672`.

[84] Gauthier Lyan, Jean-Marc Jézéquel, David Gross-Amblard, and Benoît Combemale. DataTime: A Framework to Smoothly Integrate Past, Present and Future into Models. pages 134–144. IEEE, 10 2021. ISBN 978-1-6654-3495-9. doi: 10.1109/MODELS50736.2021.00022.

[85] Gauthier Lyan, Jean-Marc Jézéquel, David Gross-Amblard, Romain Lefeuvre, and Benoit Combemale. Reasoning Over Time into Models with DataTime. *Software and Systems Modeling*, pages 1–25, December 2022. URL `https://inria.hal.science/hal-03921928`.

[86] Favorskaya Margarita, Pandey R. K., Shaw Rabindra Nath, and Mekhilef Saad. Innovations in Electrical and Electronic Engineering. page 1002. Springer Nature Singapore, 2021.

[87] Maven. Apache Maven, 2023. `https://maven.apache.org/`.

[88] Mockito. Tasty Mocking Framework for Unit Tests in Java - Mockito, 2023. `https://site.mockito.org/`.

[89] H. Mohapatra and A.K. Rath. *Fundamentals of Software Engineering: Designed to Provide an Insight Into the Software Engineering Concepts*. BPB PUBN, 2020. ISBN 9789388511773. URL https://books.google.ca/books?id=puPJDwAAQBAJ.

[90] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of Machine Learning, Second Edition*. Adaptive Computation and Machine Learning series. MIT Press, 2018. ISBN 9780262039406. URL https://books.google.ca/books?id=V2B9DwAAQBAJ.

[91] Mehryar Mohri, Ameet Talwalkar, and Afshin Rostamizadeh. *Foundations of Machine Learning*. The MIT Press, 2nd edition, 2018.

[92] Mondayblog. Task Analysis and How it Can Help Build a Project Team, August 2022. https://monday.com/blog/project-management/task-analysis/.

[93] P. Morin. *Open Data Structures: An Introduction*. Number v. 9 in Online access: Center for Open Education Open Textbook Library. Athabasca University Press, 2013. ISBN 9781927356388. URL https://books.google.ca/books?id=ZZCJvrDe5bIC.

[94] Gunter Mussbacher. Lecture Notes on Requirements Elicitation. ECSE 326 Lecture Notes, Fall 2022. Software Engineering, McGill University.

[95] Gunter Mussbacher. Evaluating Requirements Models with URN: Features, Goals, and Scenarios, Winter 2023. Lecture notes for ECSE439/539 (Advanced) Software Language Engineering, McGill University.

[96] Lyubov N. Improve Your Efficiency: Checklists for Software Testing, 2017. https://rb.gy/sqfpr.

[97] Juan Navas. What is Hyperparameter Tuning?, February 2022. https://www.anyscale.com/blog/what-is-hyperparameter-tuning.

[98] Damir Nešić, Jacob Krüger, Ștefan Stănciulescu, and Thorsten Berger. Principles of Feature Modeling. pages 62–73. ACM, 8 2019. ISBN 9781450355728. doi: 10.1145/3338906.3338974.

[99] Jerry Nicholas. 9 Elicitation Techniques Used by Business Analysts – Tips and Guidance, 2023. https://rb.gy/fs66h.

[100] McGill Faculty of Engineering. Minor Applied Artificial Intelligence, 2023. https://rb.gy/u86uc.

[101] McGill Faculty of Engineering. Minor Software Engineering, 2023. https://rb.gy/zp4at.

[102] Richard Paige. The Meta-Object Facility (MOF), July 2006. https://wiki.eclipse.org/images/0/06/OMCW_chapter04_MOFLecture.York.pdf.

[103] Visual Paradigm. What is Sequence Diagram?, 2023. https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/.

[104] Deepak Parmar. Exploratory Testing, 2023. https://shorturl.at/giRX2.

[105] Hoang Pham. *Software Reliability*. John Wiley & Sons, Inc., 1999. ISBN 9813083840.

[106] Dan Pilone and Neil Pitman. *UML 2.0 in a Nutshell*. O'Reilly Media, 2005.

[107] PostgreSQL. PostgreSQL: The World's Most Advanced Open Source Relational Database, 2023. https://www.postgresql.org/.

[108] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, January 2014.

[109] Productboard. Backlog Grooming or Refinement, 2023. https://rb.gy/h4h4k.

[110] Pranoy Radhakrishnan. A Short Machine Learning Explanation — In Terms of Linear Algebra, Probability and Calculus, 2018. https://rb.gy/4hcwa.

[111] Dan Radigan. Kanban: How The Kanban Methodology Applies To Software Development, 2023. http://surl.li/jwblg.

[112] Naur Peter Randell, Brian. NATO Software Engineering Reports. PDF, October 1968. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

[113] Max Rehkopf. What Is Continuous Integration?, 2023. http://surl.li/jwbkt.

[114] SAFe. Domain Modeling, March 2023. https://scaledagileframework.com/domain-modeling/.

[115] Salesforce. Heroku, 2023. https://www.heroku.com/.

[116] June Sallou. *On Reliability and Flexibility of Scientific Software in Environmental Science : Towards a Systematic Approach to Support Decision-Making*. Theses, Université Rennes 1, February 2022. URL https://theses.hal.science/tel-03854849.

[117] Erin Schaffer. Front-end vs Back-end Development: What's the Difference?, October 2021. https://rb.gy/s123a.

[118] D.C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39:25–31, 2 2006. ISSN 0018-9162. doi: 10.1109/MC.2006.58.

[119] Oszkár Semeráth, András Szabolcs Nagy, and Dániel Varró. A Graph Solver for the Automated Generation of Consistent Domain-Specific Models. In *40th International Conference on Software Engineering (ICSE 2018)*, pages 969–980. ACM, 5 2018.

[120] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning From Theory to Algorithms*. Cambridge University Press, 2014.

[121] Divesh Singh. Differences Between Architecture and Design Pattern, September 2019. https://singhdivesh.medium.com/according-to-wikipedia-b1afa6de08c.

[122] John Smart. *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*. Manning Publications, September 2014. ISBN 9781638353218.

[123] Sourceforge. USE: UML-Based Specification Environment, 2023. https://sourceforge.net/projects/useocl/.

[124] Springboard. Regression vs Classification, 2023. https://www.springboard.com/blog/data-science/regression-vs-classification/.

[125] Sagar Sunkle, Krati Saxena, Ashwini Patil, and Vinay Kulkarni. AI-Driven Streamlined Modeling: Experiences and Lessons Learned from Multiple Domains. *Software and Systems Modeling*, 21:1–23, 6 2022. ISSN 1619-1366. doi: 10.1007/s10270-022-00982-6.

[126] Simon Swords. Creating a Software Product Vision Statement, 2017. `https://rb.gy/syay7`.

[127] Sparx Systems. Meta Object Facility, 2023. `https://bit.ly/40h3FeW`.

[128] VMware Tanzu. Spring Boot 3.1.0, 2023. `https://spring.io/projects/spring-boot`.

[129] Fei Tao, Jiangfeng Cheng, Qinglin Qi, Meng Zhang, He Zhang, and Fangyuan Sui. Digital Twin-Driven Product Design, Manufacturing and Service With Big Data. *The International Journal of Advanced Manufacturing Technology*, 94:3563–3576, 2 2018. ISSN 0268-3768. doi: 10.1007/s00170-017-0233-1.

[130] E. Tran. Verification/Validation/Certification. In P. Koopman, editor, *Topics in Dependable Embedded Systems*. Carnegie Mellon University, 1999. `https://users.ece.cmu.edu/~koopman/des_s99/verification/index.html`.

[131] Tutorialspoint. SDLC - Iterative Model, 2023. `https://www.tutorialspoint.com/sdlc/sdlc_iterative_model.htm`.

[132] Tutorialspoint. UML Modeling Types, 2023. `https://www.tutorialspoint.com/uml/uml_modeling_types.htm`.

[133] Umple. Umple, 2023. `https://cruise.umple.org/umple/`.

[134] USE. The UML-Based Specification Environment, 2015. `https://useocl.sourceforge.net/w/index.php/Main_Page`.

[135] Muhammad Usman, Kai Petersen, Jürgen Börstler, and Pedro Santos Neto. Developing and using Checklists to Improve Software Effort Estimation: A Multi-Case Study. *Journal of Systems and Software*, 146:286–309, 2018. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2018.09.054. URL `https://www.sciencedirect.com/science/article/pii/S0164121218302073`.

[136] Sairam Vakkalanka. Requirements Triage - Challenges and Solutions. *International Journal of Software Engineering Applications*, 3:41–58, 3 2012. ISSN 09762221. doi: 10.5121/ijsea.2012.3204.

[137] Hans Van Vliet. *Software Engineering: Principles and Practice*. Wiley, 3rd edition, May 2008. ISBN 978-0-470-03146-9.

[138] Shrivastava Vineeta, Shrivastava Prashant Kumar (Dr), Kamble Megha (Dr), Shrivastava Gourav (Dr), and Udgir Vaibhav. *An Introduction to Machine Learning*. Blue Rose Publishers, 2023.

[139] Vue.js. The Progressive Javascript Framework, 2023. `https://vuejs.org/`.

[140] L. Weir and Z. Nemec. *Enterprise API Management: Design and Deliver Valuable Business APIs*. Packt Publishing, 2019. ISBN 9781787285613. URL `https://books.google.ca/books?id=0OikDwAAQBAJ`.

[141] Ivy Wigmore. Software Development, 2023. https://www.techtarget.com/whatis/definition/software-development.

[142] Wikepedia. Software Verification and Validation, 2023. https://en.wikipedia.org/wiki/Software_verification_and_validation.

[143] Wikipedia. API, 2023. https://en.wikipedia.org/wiki/API.

[144] Wikipedia. Behavior-Driven Development, 2023. https://en.wikipedia.org/wiki/Behavior-driven_development.

[145] Wikipedia. Code Coverage, 2023. https://en.wikipedia.org/wiki/Code_coverage.

[146] Wikipedia. Database Design, 2023. https://en.wikipedia.org/wiki/Database_design.

[147] Wikipedia. Feature Model, 2023. https://en.wikipedia.org/wiki/Feature_model.

[148] Wikipedia. Hibernate (Framework), 2023. https://en.wikipedia.org/wiki/Hibernate_(framework).

[149] Wikipedia. Machine Learning, 2023. https://en.wikipedia.org/wiki/Machine_learning.

[150] Wikipedia. Model-Based Testing, 2023. https://en.wikipedia.org/wiki/Model-based_testing.

[151] Wikipedia. Object–Relational Mapping, 2023. https://en.wikipedia.org/wiki/Objectrelational_mapping.

[152] Wikipedia. Software Prototyping, 2023. https://en.wikipedia.org/wiki/Software_prototyping.

[153] Wikipedia. Product Requirements Document, 2023. https://en.wikipedia.org/wiki/Product_requirements_document.

[154] Wikipedia. Software Language, 2023. https://en.wikipedia.org/wiki/Software_language.

[155] Wikipedia. Software Engineering, 2023. https://en.wikipedia.org/wiki/Software_engineering.

[156] Wikipedia. UML State Machine, 2023. https://en.wikipedia.org/wiki/UML_state_machine.

[157] Wikipedia. Unified Modeling Language, 2023. https://en.wikipedia.org/wiki/Unified_Modeling_Language.

[158] Wikipedia. XML Metadata Interchange, 2023. https://en.wikipedia.org/wiki/XML_Metadata_Interchange.

[159] Wikipedia. XML, 2023. https://en.wikipedia.org/wiki/XML.

[160] T. Winters, T. Manshreck, and H. Wright. *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly Media, 2020. ISBN 9781492082767. URL https://books.google.ca/books?id=V3TTDwAAQBAJ.

[161] Kinza Yasar. Software Engineering, March 2023. https://www.techtarget.com/whatis/definition/software-engineering.

[162] Aaron Zhu. What is Regularization: Bias-Variance Tradeoff, 2022. https://rb.gy/4t1tt.

[163] Vaughan Jack Zola, Andrew. Backpropagation Algorithm, 2022. https://www.techtarget.com/searchenterpriseai/definition/backpropagation-algorithm.

A

Xtext Textual Language Definition for the MODA Metamodel

The Xtext grammars used to implement the various variations of the MODA metamodel as seen in Chapter 3 are listed in this appendix. The Xtext language engineering environment allowed us to quickly prototype and analyze each metamodel variation. It allowed us to think through the metamodel variations critically as to why the final metamodel is concluded and the advantages and disadvantages of each variation. It was much easier to perform this analysis textually than graphically, i.e., Ecore modeling.

Programs A.1, A.2, A.3, A.4, and A.5 refer to the grammar for the textual language used to generate the metamodel for variations 1, 2, 3, 4, and the final metamodel, respectively.

Program A.1: Xtext Grammar for Variation 1

```
1  grammar org.xtext.example.moda.Moda with org.eclipse.xtext.common.Terminals
2  generate moda "http://www.xtext.org/example/moda/Moda"
3
4  Moda:
5    description = STRING
6    (models += Model)*
7    (data += Data)*
8    (runningsoftware += RunningSoftware)*
9    (sts += STS)*
10 ;
11 Model: DescriptiveModel | PredictiveModel | PrescriptiveModel
12 ;
13 DescriptiveModel:
14   'descriptive' name = ID '=>' ActionE=[PredictiveModel]
15 ;
16 PredictiveModel:
17   'predictive' name = ID '=>' ActionF=[PrescriptiveModel]
18 ;
19 PrescriptiveModel:
20   'prescriptive' name = ID '=>' ActionG=[PrescriptiveModel] '=>' ActionH=[RunningSoftware] '=>'
         ActionI=[STS]
21 ;
```

```
Data: Input | Output | Measured | External
   '=>' ActionD=[DescriptiveModel]
;
Input:
  'inputData' name = ID '=>' ActionA=[RunningSoftware]
;
Output:
  'outputData' name = ID
;
Measured:
  'measuredData' name = ID
;
External:
  'externalData' name = ID '=>' ActionJ=[STS]
;
RunningSoftware:
  'runtime' name = ID '=>' ActionB=[Output] '=>' ActionC=[Measured]
   state = STRING
;
STS:
  'socioTech' name = ID '=>' runningsoftware=[RunningSoftware] '=>' ActionJ=[External]
;
```

Program A.2: Xtext Grammar for Variation 2

```
grammar org.xtext.example.moda.Moda with org.eclipse.xtext.common.Terminals
generate moda "http://www.xtext.org/example/moda/Moda"

Moda:
  description = STRING
  (models += Model)*
  (data += Data)*
  (runningsoftware += RunningSoftware)*
  (sts += STS)*
  (actions += Action)*
;
Action:
  'action' name = ID
   'type' type = ActionType
;
Model: DescriptiveModel | PredictiveModel | PrescriptiveModel
;
DescriptiveModel:
  'descriptive' name = ID
;
PredictiveModel:
  'predictive' name = ID
;
PrescriptiveModel:
  'prescriptive' name = ID
;
Data: Input | Output | Measured | External
;
Input:
  'inputData' name = ID
;
Output:
  'outputData' name = ID
;
```

```
Measured:
  'measuredData' name = ID
;
External:
  'externalData' name = ID
;
RunningSoftware:
  'runtime' name = ID
   state = STRING
;
STS:
  'socioTech' name = ID '=>' runningsoftware=[RunningSoftware]
;

enum ActionType:
  A = 'A'|
  B = 'B'|
  C = 'C'|
  D = 'D'|
  E = 'E'|
  F = 'F'|
  G = 'G'|
  H = 'H'|
  I = 'I'|
  J = 'J'
;
```

Program A.3: Xtext Grammar for Variation 3

```
grammar org.xtext.example.moda.Moda with org.eclipse.xtext.common.Terminals
generate moda "http://www.xtext.org/example/moda/Moda"

Moda:
  description = STRING
  (elements += Element)*
  (actions += Action)*
;
Action:
  'action' name = ID
   from = [Element] '->' to = [Element]
   'type' type = ActionType
;
Element: Data | Model | RunningSoftware | STS
;
Model: DescriptiveModel | PredictiveModel | PrescriptiveModel
;
DescriptiveModel:
  'descriptive' name = ID
;
PredictiveModel:
  'predictive' name = ID
;
PrescriptiveModel:
  'prescriptive' name = ID
;
Data: Input | Output | Measured | External
;
Input:
  'inputData' name = ID
```

```
;
Output:
  'outputData' name = ID
;
Measured:
  'measuredData' name = ID
;
External:
  'externalData' name = ID
;
RunningSoftware:
  'runtime' name = ID
   state = STRING
;
STS:
  'socioTech' name = ID '=>' runningsoftware=[RunningSoftware]
;

enum ActionType:
  A = 'A'|
  B = 'B'|
  C = 'C'|
  D = 'D'|
  E = 'E'|
  F = 'F'|
  G = 'G'|
  H = 'H'|
  I = 'I'|
  J = 'J'
;
```

Program A.4: Xtext Grammar for Variation 4

```
grammarorg.xtext.example.moda.Moda withorg.eclipse.xtext.common.Terminals
generate moda "http://www.xtext.org/example/moda/Moda"

Moda:
    description = STRING
    (elements += Element)*
    (actions += Action)*
;
Action:
    'action' name= ID
    from = [Element] '->' to = [Element]
    'type' type = ActionType
;
Element: Data | Model | RunningSoftware | STS
;
Model:
    (descriptive ?= 'descriptive')?
    (predictive ?= 'predictive')?
    (prescriptive ?= 'prescriptive')?
    'model' name= ID
;
Data: Input | Output | Measured | External
;
Input:
    'inputData' name= ID
;
```

```
Output:
    'outputData' name= ID
;
Measured:
    'measuredData' name= ID
;
External:
    'externalData' name= ID
;
RunningSoftware:
    'runtime' name= ID
    state = STRING
;
STS:
    'socioTech' name= ID '=>' runningsoftware=[RunningSoftware]
;

enum ActionType:
A = 'A'|
B = 'B'|
C = 'C'|
D = 'D'|
E = 'E'|
F = 'F'|
G = 'G'|
H = 'H'|
I = 'I'|
J = 'J'
;
```

Program A.5: Xtext Grammar for Final Metamodel

```
grammar org.xtext.example.moda.Moda with org.eclipse.xtext.common.Terminals
generate moda "http://www.xtext.org/example/moda/Moda"

Moda:
  description = STRING
  (elements += Element)*
  (actions += Action)*
;
Action:
  'action' name = ID
   from = [Element] '->' to = [Element]
   'type' type = ActionType
;
Element: Data | Model | RunningSoftware | STS
;
Data:
  (inputData ?= 'input')?
  (outputData ?= 'output')?
  (measuredData ?= 'measured')?
  (externalData ?= 'external')?
  'data' name = ID
;
Model:
  (descriptive ?= 'descriptive')?
  (predictive ?= 'predictive')?
  (prescriptive ?= 'prescriptive')?
  'model' name = ID
```

```
28 ;
29 RunningSoftware:
30   'runtime' name = ID
31   state = STRING
32 ;
33 STS:
34   'socioTech' name = ID '=>' runningsoftware=[RunningSoftware]
35 ;
36 enum ActionType:
37   A = 'A'|
38   B = 'B'|
39   C = 'C'|
40   D = 'D'|
41   E = 'E'|
42   F = 'F'|
43   G = 'G'|
44   H = 'H'|
45   I = 'I'|
46   J = 'J'
47 ;
```

B

Java Code for Metamodel Validation

The Java code defines the rules written to validate the MODA metamodel.

Program B.1: Java Code for Metamodel Validation

```
1  /*
2   * generated by Xtext 2.29.0
3   */
4
5  package org.xtext.example.moda.validation;
6
7  import java.util.stream.Collectors;
8
9  import org.eclipse.xtext.validation.Check;
10 import org.xtext.example.moda.moda.Action;
11 import org.xtext.example.moda.moda.Data;
12 import org.xtext.example.moda.moda.Moda;
13 import org.xtext.example.moda.moda.ModaPackage;
14 import org.xtext.example.moda.moda.Model;
15 import org.xtext.example.moda.moda.RunningSoftware;
16 import org.xtext.example.moda.moda.STS;
17
18
19 /*
20  * This class contains custom validation rules.
21  *
22  * See https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#validation
23  */
24 public class ModaValidator extends AbstractModaValidator {
25
26 // action names cannot be repeated
27 @Check
28 public boolean checkActionNameIsUniqueWithinModa(Action action) {
29     var actions = ((Moda) action.eContainer()).getActions();
30     var actionName = action.getName();
31
32     // Check if the action name is already used by another action
33       long count = actions.stream()
```

```
34              .filter(a -> a.getName().equals(actionName))
35              .count();
36
37      if (count > 1) {
38          warning("Action name must be unique", ModaPackage.Literals.ACTION__NAME);
39          return false; // Name is not unique
40      }
41
42      // Return true if the name is unique
43      return true;
44  }
45
46  // elements names cannot be repeated
47  @Check
48  public boolean checkElementNameIsUniqueWithinModa(Element element) {
49      var elements = ((Moda) element.eContainer()).getElements();
50      var elementName = element.getName();
51
52      // Check if the action name is already used by another action
53        long count = elements.stream()
54              .filter(a -> a.getName().equals(elementName))
55              .count();
56
57      if (count > 1) {
58          warning("Element name must be unique", ModaPackage.Literals.ELEMENT__NAME);
59          return false; // Name is not unique
60      }
61
62      // Return true if the name is unique
63      return true;
64  }
65
66  // at least one model attribute should be true
67  @Check
68  public boolean checkAtLeastOneModelAttributeIsTrue (Model model) {
69      if (model.isDescriptive() || model.isPredictive() || model.isPrescriptive()) {
70          return true; // At least one model attribute is true
71      }
72
73      warning("At least one model attribute should be true", ModaPackage.Literals.ELEMENT__NAME);
74      return false;
75  }
76
77  // at least one data attribute should be true
78  @Check
79  public boolean checkAtLeastOneDataAttributeIsTrue (Data data) {
80      if (data.isInputData() || data.isOutputData() || data.isMeasuredData() ||
81          data.isExternalData()) {
82          return true; // At least one data attribute is true
83      }
84
85      warning("At least one data attribute should be true", ModaPackage.Literals.ELEMENT__NAME);
86      return false;
87  }
88
89  //At least one model should be present
90  @Check
91  public boolean checkAtLeastOneModelPresent(Moda moda) {
92        for (Element element : moda.getElements()) {
```

```
93              if (element instanceof Model) {
94                  return true; // At least one model is present
95              }
96          }
97          warning("At least one model should be present", ModaPackage.Literals.ELEMENT__NAME);
98          return false;
99  }
100
101 // which elements do arrows A - J move from and to what
102 @Check
103 public boolean checkActionTypeDirections(Action action) {
104     boolean isValid = false;
105     switch (action.getType()) {
106
107     case A:
108       isValid = action.getFrom() instanceof Data && ((Data) (action.getFrom())).isInputData() &&
              action.getTo() instanceof RunningSoftware;
109       break;
110
111     case B:
112       isValid = action.getFrom() instanceof RunningSoftware && action.getTo() instanceof Data && ((
              Data) (action.getTo())).isOutputData();
113       break;
114
115     case C:
116       isValid = action.getFrom() instanceof RunningSoftware && action.getTo() instanceof Data && ((
              Data) (action.getTo())).isMeasuredData();
117       break;
118
119     case D:
120       isValid = action.getFrom() instanceof Data && action.getTo() instanceof Model && ((Model) (
              action.getTo())).isDescriptive()
121           || action.getFrom() instanceof Data && action.getTo() instanceof Model && ((Model) (
                  action.getTo())).isPrescriptive();
122       break;
123
124     case E:
125        isValid = action.getFrom() instanceof Model && ((Model) (action.getFrom())).isDescriptive()
               && action.getTo() instanceof Model && ((Model) (action.getTo())).isPredictive()
126         || action.getFrom() instanceof Data && action.getTo() instanceof Model && ((Model) (action
               .getTo())).isPredictive();
127       break;
128
129     case F:
130       isValid = action.getFrom() instanceof Data && action.getTo() instanceof Model && ((Model) (
              action.getTo())).isPrescriptive()
131         || action.getFrom() instanceof Model && ((Model) (action.getFrom())).isDescriptive() &&
              action.getTo() instanceof Model && ((Model) (action.getTo())).isPrescriptive()
132         || action.getFrom() instanceof Model && ((Model) (action.getFrom())).isPredictive() &&
              action.getTo() instanceof Model && ((Model) (action.getTo())).isPrescriptive();
133       break;
134
135     case G:
136       isValid = action.getFrom() instanceof Model && ((Model) (action.getFrom())).isPrescriptive()
              && action.getTo() instanceof Model && ((Model) (action.getTo())).isPrescriptive();
137       break;
138
139     case H:
```

```
140       isValid = action.getFrom() instanceof Model && ((Model) (action.getFrom())).isPrescriptive()
              && action.getTo() instanceof RunningSoftware;
141       break;
142
143     case I:
144       isValid = action.getFrom() instanceof Model && ((Model) (action.getFrom())).isPrescriptive()
              && action.getTo() instanceof STS;
145       break;
146
147     case J:
148       isValid = action.getFrom() instanceof STS && action.getTo() instanceof Data && ((Data) (action
              .getTo())).isExternalData()
149           || action.getFrom() instanceof Data && ((Data) (action.getTo())).isExternalData() &&
                  action.getTo() instanceof STS ;
150       break;
151
152     default:
153       break;
154     }
155
156     if (!isValid) {
157       warning("Invalid action direction", ModaPackage.Literals.ACTION__TYPE);
158     }
159
160     return isValid;
161   }
162 }
```