# A Business User Model-Driven Engineering Method for Developing Information Systems


**Ahmad F Subahi**


22 February 2015

**This Thesis is submitted for the degree of** *Doctor of Philosophy*


**Verification and Testing Group**

**Department of Computer Science**

**University of Sheffield**


**Supervisor: Dr. Anthony J.H. Simons**

# Abstract

With the rapid development of general-purpose programming languages and platform technologies, software engineers have faced more various challenges in software development to those that occurred in the past decades. Requirements Change might cause several project management and technical conflicts associated with requirement elicitation, intercommunication and later changes of specifications.

In the real-world, there is a demand to adopt an accurate information system that satisfies the requirements and is used effectively for the business. However, having vague or misinterpreted requirements causes errors and extra costs. Therefore, domain experts, who clearly understand the business logic and goals, and are aware of what exactly they need inside an organisation without professional software developing skills, should play key roles in the development lifecycle using high level tools. Model-Driven Engineering, is a software engineering discipline that aims at raising the level of abstraction by capturing system specifications through the employment of Models. MDE supports integration and interoperability, improves software quality, and reduces development cost by supporting the automatic creation of software systems using appropriate toolsets.

This thesis is all about raising the level of abstraction at which information systems are built, using business end-users knowledge and MDE to achieve the result. The work introduces, first, Micro-Modelling Language ($\mu$ML), a lightweight modelling language that is used to express basic structural and behavioural aspects of information systems using effectively business-users knowledge of their desired system. Throughout the work, graphical notation and semantics for the language concepts are identified, providing a simpler and semantically cleaned modelling language than standard UML and other UML-based languages.

The work also proposes BUILD (Business-User Information-Led Development), an End-User MDE approach to support the construction of information systems using high-level specifications and accelerate the development process using layered model transformation and code generation. Throughout the thesis, a number of development phases and model transformation steps are identified to allow the low-level technical detail be introduced and developed automatically by rules, with less end-users engagement. Domain-Specific code generators, for generating executable Java Swing Applications code and MySQL script, are used to demonstrate the validity of the research.

# Acknowledgements

I would like to use this opportunity to express my gratitude and love to all the people in my life, who were there for me and were a part of the process during which this thesis would not have happened without the generous support and encouragement of many people.

The first person I would like to thank is my supervisor Dr. Anthony J.H. Simons, as this PhD research is the synthesis of five years of work whereby I have been accompanied and guided by him. During these years I have known Anthony (Tony) as both a clever researcher, and a genuine and friendly person. Together with him, I would like to thank Prof. Georg Struth for his valuable advice, ideas and suggestions during the design of the formal logic, which surely improved the quality of the thesis. Moreover, many thanks to Dr. Simon Foster for his worthy assistance in the theoritical side of Graph Transformation at the early stage of the research.

I would like to mention all members of the Verification and Testing (vt) group. To those I have spent most of these years together with in the Department of Computer Science, University of Sheffield. Many thanks to all of them. Moreover, huge thanks go to my dear wife and all friends for putting up with me and for their support and patience that were fundamental in concluding my PhD.

I feel a deep sense of gratitude and I owe sincere and earnest thankfulness to my valuable Irish friends for their special caring, Dr. Hayder Ahmed, his lovely family, Dr. A Almanea, and Mrs L Ahmed, You are a part of my life who were here for me when I needed any piece of advice and encouragement. My special thanks and love to the special one who was never-ending support and inspiration in every conceivable way. Without the patience, love and endless support this thesis wouldn't have been written.

## List of Acronyms

| | |
|---|---|
| **ADISSA** | Architectural Design of Information Systems Structure Analysis Methodology |
| **AOSD** | Aspect Oreinted Software Development |
| **ASL** | Action Specification Language |
| **AST** | Abstract Syntax Tree |
| **ATL** | AtlanMod Transformation Language |
| **BPMN** | Business Process Modeling Notation |
| **BUILD** | Business-User Information-Led Development |
| **CASE** | Computer Aided Software Engineering |
| **CBD** | Component-Based Development |
| **CBEADS** | Component-Based EBusiness Application Development and Deployment Shell |
| **CIM** | Computation Independent Model |
| **CRUD** | Create, Read, Update and Delete effects |
| **DBQ** | DataBase and Query |
| **DFD** | DataFlow Diagram |
| **DSL** | Domain Specific Language |
| **DSM** | Domain-Specific Modelling |
| **DTD** | Document Type Definition |
| **ECA** | Event-Condition-Action |
| **EIS** | Enterprise Information System |
| **EMF** | Eclipse Modeling Framework |
| **EUD** | End-User Development |
| **FOOM** | Functional and Object-Oriented Methodology |
| **FOPL** | First-Order Predicate Logic |
| **GEF** | Graphical Editing Framework |
| **GMF** | Graphical Modeling Framework |
| **GPL** | General-Purpose Programming Language |
| **GReAT** | Graph Rewriting and Transformation |
| **GSD** | Generative Software Development |
| **GUI** | Graphical User Interface |
| **HCI** | Human-Computer Interaction |
| **HTML** | HyperText Markup Language |
| **IS** | Information System |
| **J2EE** | Java 2 Platform Enterprise Edition |
| **JaMDA** | Java Model-Driven Architecture |
| **JDBC** | Java Database Connectivity |
| **JMermaid** | Java MERode Modelling AID |
| **JSP** | Java Server Page |
| **KerMeta** | Kernel Metamodeling |

| | |
|---|---|
| **KM3** | Kernel Meta Meta Model |
| **LDE** | Language-Driven Engineering |
| **MDA** | Model-Driven Architecture |
| **MDD** | Model-Driven Development |
| **MDD4EU** | Model-Driven Development for End-Users |
| **MDE** | Model-Driven Engineering |
| **MERODE** | Model driven Existence dependency Relation Object oriented DEvelopment |
| **MIC** | Model-Integrated Computing |
| **MIDAS** | Model Driven Architecture framework for development of Web ISs |
| $\mu$**ML** | Micro-Modelling Language |
| **MOF** | Meta Object Facility |
| **MOSYS** | Methodology for Automatic Object Identification from System Specification |
| **OCL** | Object Constraint Language |
| **OMG** | Object Management Group |
| **OOAD** | Object-Oriented Analysis and Design |
| **OPCAT** | Object Process CASE Tool |
| **OPD** | Object Process Diagram |
| **OPL** | Object Process Language |
| **OPM** | Object Process Methodology |
| **PervML** | Pervasive Modelling Language |
| **PIM** | Platform-Independent Model |
| **PSM** | Platform-Specific Model |
| **QVT** | Query/View/Transformation |
| **ReMoDeL** | Reusable Modelling Design Language |
| **SiTra** | Simple Transformer |
| **SQL** | Structured Query Language |
| **UML** | Unified Modelling language |
| **UML-RSDS** | UML Reactive System Development Support |
| **UWE** | UML-Based Web Engineering |
| **WebML** | Web Modelling Language |
| **WIS** | Web Information System |
| **XML** | Extensible Markup Language |
| **xUML** | Executable Unified Modeling Language |
| **ZOOM** | Z-Based Object Oriented Modelling |

## List of Publications

- Ahmad F. Subahi, Anthony .J.H. Simons. A Multi-level Transformation from Conceptual Data Models to Database Scripts Using Java Agents, *Proceedings in the Workshop on Composition and Evaluation of Model Transformations*, King's College London, 2011.

- Ahmad F. Subahi and Anthony J.H. Simons, A model transformation approach for translating conceptual database schemas into executable database systems, Technical Report, Department of Computer Science, University of Sheffield, 2011, 1-10.

- Ahmad F. Subahi and Anthony J.H. Simons, Domain-Specific Language for Enabling End-Users Model-Driven Information System Engineering, World Academy of Science, Engineering and Technology, *International Science Index 79, International Journal of Computer, Information, Systems and Control Engineering*, 7(7), 2013, 318 - 321.

- Anthony J.H. Simons, Ahmad F. Subahi and Stephen M.T. Eyre, Practical model-to-code transformation in four object-oriented programming languages, Technical Report, Department of Computer Science, University of Sheffield, 2011, 1-25.

# Declaration

I hereby declare that this thesis is my own work and effort and that it has not been submitted anywhere for any award. Where other sources of information have been used, they have been acknowledged.

**Ahmad F Subahi**

# Contents

# List of Figures

# List of Tables

# 1
# Introduction

"Walking on water and developing software from a specification are easy if both are frozen"

Edward V. Berard

## 1.1 Context

The thesis is about raising the level of abstraction at which information systems are built, and this requires a user-friendly modelling language and an appropriate Model-Driven Engineering (MDE) method to achieve the result. In this chapter an overview of the general area addressed by this thesis is presented, highlighting certain aspects such as the principles of MDE and some comparisons of different MDE concepts and techniques. The outline of the thesis is presented next, in the rest of the chapter, including the motivation, aims and a summary of contributions and objectives.

## 1.2 Grand Challenges in Software Development

With the rapid development of general-purpose programming languages and platform technologies, software engineers have in the current decade faced more varied challenges in software development than those encountered in earlier decades. According to [23], these challenges are a result of three major factors, which lead to the accidental increase in lines of code: *Implementation Complexity*, *Platform Diversity*, and *Requirements Change*. For instance, building a robust software system that is feasible, distributed, secure, and portable requires additional sophisticated requirements and more stakeholders to be involved in the development process. This might cause several project management and technical conflicts associated with requirement elicitation, inter-communication and later changes of specifications. Consequently, there will be a demand to expand the project size, which leads to an increase in the development and maintenance cost, extra testing and bug-fixing tasks, and a delay in time of market delivery [62].

Besides this, dealing with the variety of existing platforms brings other issues to developers and their knowledge. It is unattainable for a developer to be a specialist of every programming language, and domain technology. For example, there is no standard way, adopted in all programming languages and platform technologies, to deal with a particular requirement. Each domain has an appropriate approach to handle some types of requirements, which cannot be applied to other domains. Thus, developers must be capable of coping with a variety of domains, implementation technologies and languages to produce good-quality systems. Furthermore, as an on-going process, the clients might change the requirements during the development lifecycle. The major changes might require code rewriting, and disturbing other processes, which increases cost [23].

## 1.3 Recent Possible Solutions for the Problems

A number of software development approaches, such as Model-Driven Engineering (MDE), and Language-Driven Engineering (LDE) [23], have introduced the idea of raising the level of abstraction beyond the General-Purpose Programming Languages (GPLs) through the usage of Models and high-level programming languages tailored to a specific domain, known as Domain-Specific Languages (DSLs). These various strategies are commonly adopted in software development, as Domain-Specific Modelling Frameworks (DSMs) [62], to tackle the challenges stated above. The DSM approaches are based on the usage of DSL with an associated compiler to model and capture system specifications of a particular domain at a higher level of abstraction [62]. They bring a remarkable contribution for reducing software complexity, achieving the separation of concerns between the design and implementation phases, and providing a better view of the system. This enables developers to specify their system and capture its requirements and design problems independently from any platform-specific implementation, associated with a particular technology. Then the mappings from design to code are defined in order to derive the executable software automatically from high-level specifications. As a result, the overall software quality and productivity will be increased [23].

## 1.4 Model-Driven Engineering (MDE)

In MDE, a model is considered a primary entity in the development lifecycle that provides an abstract representation of a system. It supports integration and interoperability, improves software quality, and reduces a development cost [6]. The general strategy of MDE aims to capture system specifications through the employment of models that are expressed at a very high level of abstraction. The models can be defined using a metamodel or an appropriate DSL for a particular domain [62]. In Model-Driven approaches, the metamodel, a model of a model, defines the abstract syntax and the relationship between concepts of the model. Figure 1.1 demonstrates the relationship between model and metamodels.

The Meta-Object Facility (MOF) [87] is a metamodelling strategy that is provided by the Object Management Group (OMG) [86] and supported in Model Driven Architecture (MDA). It has a hierarchical structure that consists of four layers from M0 to M3 (Figure 1.2), each one defines the level below to ensure the consistency and the correctness of the instance model syntax and semantics at each level of abstraction [22].

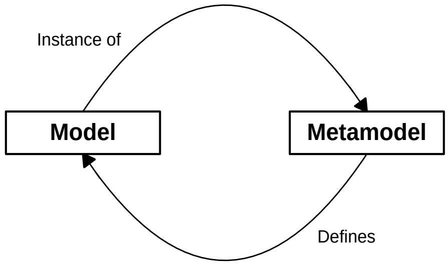



Figure 1.1: The Relationship between Models and Metamodels

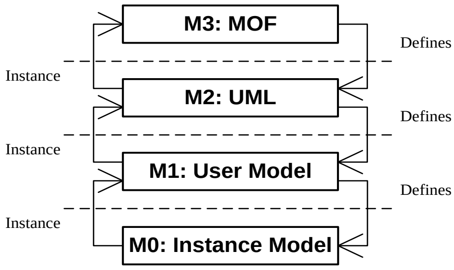



Figure 1.2: MOF Levels

- M3 is the meta-meta-model layer and defines models (metamodels) in the layer below it, the M2 layer. For example UML [70] node and arc diagrams that provide meta language to define UML modelling language [70].

- M2 is the metamodel layer and is a language that describes the models in the M1 layer. For example, UML [70] elements, namely, Class, Association, and Attribute are defined at M2, as illustrated in a snapshot of the UML [70] metamodel (Figure 1.3).

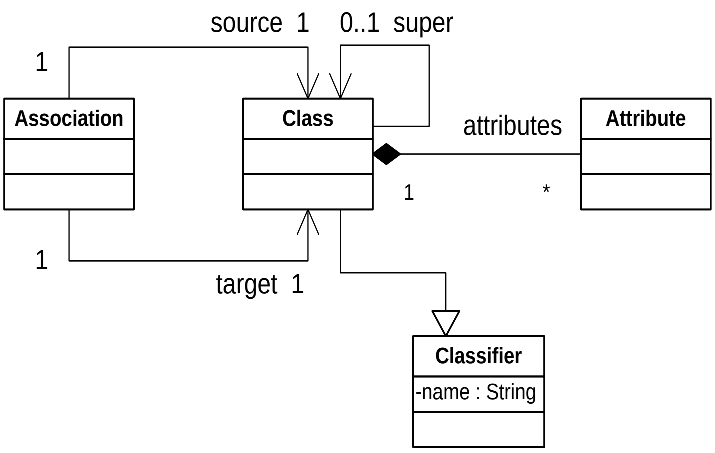



Figure 1.3: An example of M2 level models (metamodels)

- M1 is the model layer. This is where instances of UML [70] diagrams are categorised and is a level where user designed elements are defined. It contains the application model, such as the classes of an object-oriented system, or the table definitions of a relational database. Figure 1.4 illustrates an example of models at M1 level.

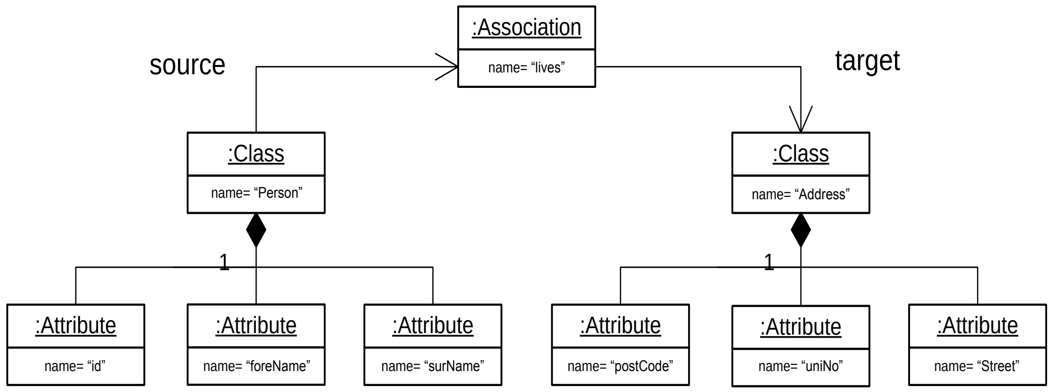



Figure 1.4: An example of M1 model

- M0 is the runtime layer that contains data about the objects that have been created from the definitions in the model to represent an instance of the system at run time. Figure 1.5 shows an example of M0 models of the relationship, called lives, between Person and Address classes.

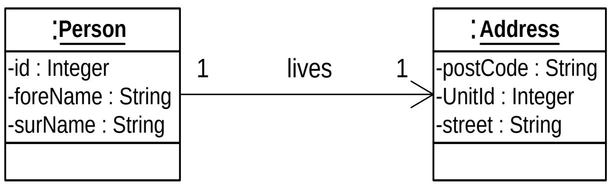



Figure 1.5: An Example of models at M0 level

By using models and metamodels, the automation can be fulfilled via the utilisation of model transformation approaches or tools. In MDE for example, a series of model transformations, that are defined by rules, is applied to an abstract model conforming to a metamodel to form a more concrete target model conforming to the same or different metamodel. The model transformations can be categorised into two types: *horizontal* and *vertical* transformations. A *horizontal* one resides at the same abstraction level in which it might present different views of the system or performs a refactoring process [62]. It might be a model *in-place optimisation* or Refactoring as in Figure 1.6 (A, E). On the other hand, a *vertical* transformation, Figure 1.6 (B, C, D) not only presents a distinct view of the system, but also it moves models between different abstraction levels [62]. It might be a *refinement* or *abstraction* (Figure 1.6 (C)), *code generation* or *visualisation* (Figure 1.6 (B), and (D)) respectively.

The model transformations, shown in Figure 1.6, can also be classified, based on the direction, into two types, namely, Unidirectional, and Bidirectional transformation. Forward, and Reverse Engineering, Figure 1.6 (B) and (D) respectively, are examples of the Unidirectional transformations. Whereas Figure 1.6 (C) illustrates a Bidirectional transformation.

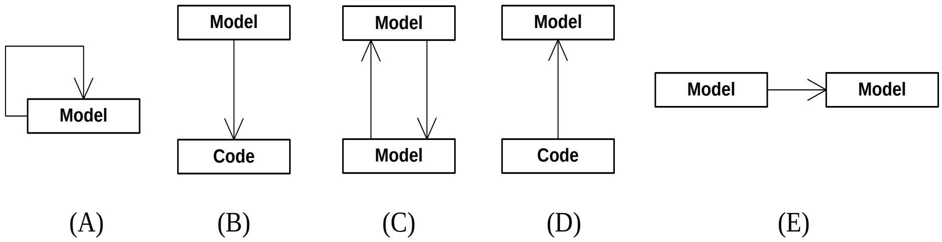



Figure 1.6: Types of model transformations

## 1.5 Motivation

When developing an information system to satisfy a business need, the expectation is that such a system will accurately satisfy its well-specified requirements. However, having vague or misinterpreted requirements causes errors and extra costs. Therefore, domain experts, who clearly understand the business logic and goals, and are aware of what exactly they need inside an organisation without professional software developing skills, should play key roles in the development lifecycle using high level tools. There are many approaches that aim to tackle these issues and reduce the gap between initial requirements and implementation and accelerating the development process, such as Model-Driven Engineering (MDE), Domain-Specific Languages (DSL) and End-user Development (EUD).

Although the Unified Modelling Language, UML [70], is commonly used to express the structure and behaviour of a system within MDE approaches, it suffers from semantic ambiguity and complexity issues. While it is possible to create designs in UML 2.x, from which complete code can be generated, this can only be done at the expense of supplying complete low-level detail in the models; that is, UML cannot at the same time be both abstract and sufficient for code generation. These fully-detailed UML models must necessarily contain far more detailed technical information than a typical end-user would be expected to understand; indeed if they have to create such models, there is a risk that these will be inconsistent, incomplete or simply difficult to understand.

To sum up, we find that current MDE tools require a degree of technical skills when dealing with complex models that are syntactically and semantically unclear. This prevents end-users from expressing directly their functional requirements. Researchers focus on developing appropriate modelling languages and DSLs to accelerate and enhance the development process led by skilful developers. As we will see in the literature review (Chapter 2), there is less attention on the role of business end-users during the MDE development lifecycle, than we would hope to see.. Even though some MDE approaches allow the end-user, at some point of development, to work side by side with developers just for customising system artefacts, it is worth exploring and improving the role of business users by allowing them to lead the development process using their conceptual knowledge of their business.

## 1.6 Research Problems

In MDE, a model has to be designed in such away that is able to express all critical aspects of the system. This requires skilful developers to construct these artefacts. This need for technical skills prevents the MDE approaches from being suitable for end-users [91]. As in the reviewed approaches (Chapter 2), end-users require a degree of technical knowledge in order to be qualified to model their system at a higher level of specification. For example, in MDA [89], end-users might be overwhelmed by the need to learn various OMG [86] standards, such as UML [70], OCL [85], and QVT [88], to be able to construct a rich Platform-Independent-Model (PIM) model that consists of adequate details for enabling a full code generation.

Adopting a DSM approach might solve this issue partially. Although the DSM approach succeeds in raising the level of abstraction, focusing on the problem domain instead of the implementation details, end-users still need to learn a new limited language for a particular domain, even if they were familiar with its concepts. For instance, end-users in the WebML [124] approach should specify all composition and navigation features of their web application using a number of design languages. The modelling process starts with constructing the data model and ends up with designing the hypertext and presentation view. According to that, end-users must act as a web designer to design each part of the system. From that, it can be argued that these approaches are appropriate for designers or engineers rather than business end-users. The highest level of abstraction requires, to some extent, technical skills to express constraints and behaviours.

The premise for this thesis is that UML [70] is too unwieldy to serve as the basis for model-driven engineering. The models in UML [70] are too complex and eclectic to be given a single, clear interpretation, while paradoxically not covering all of the views that are needed to completely specify a software system. We propose some ideas for a simpler notation, with a cleaner semantics, in which the iconography is more consistent. Individual models are smaller and more restricted, but there are more kinds of model to cover the different interlinking views of a system. As a result, it is possible to specify partial and total transformations between different kinds of model. The result is known as $\mu$ML, or the Micro-Modelling Language.

$\mu$ML aims at raising the level of abstraction to suit business end-users, enabling them to construct their system easily. Furthermore, it reduces the ambiguity of requirements and troubles that occur during client-designer communication.

## 1.7 Contribution of the Research

The research contribution here aims at raising the level of abstraction to suit business users, enabling them to construct their system easily by themselves, using less technical knowledge in a more efficient way than existing approaches. It can be said, this tackles some software engineering issues in requirement elicitations to accelerate the development process and meet end-user requirements. For instance, allowing end-users to act as system designers to express, model, and customise their functional requirements might reduce the ambiguity of requirements and troubles that occur during client-designer communication. The investigation aims at exploring to what extent the captured functional requirements, supplied by end-users, can generate a complete information system that meets their need. This raises a number of initial research questions as follows:

- What is the required interaction mechanism between the end-user and the transformation approach to capture their desired system specifications?
- What kind of abstract system views do we need to capture in order to have comprehensive knowledge and behaviour of a system?
- What transformation rules are required to fold and optimise high-level views and introduce extra detailed design information of a system?
- What transformation rules are required to refine high-level models and introduce richer design information to a system?
- At which level of development is end-user engagement required to supply new knowledge in order to be considered by model translators at the next translation step?
- To what extent are end-users able to generate a complete system for their demand?
- With respect to code generation, to what extent are we able to construct and link the information system layers from a generic and less technical specifications?

## 1.8 Objective of the Research

In technical terms, we are aiming to introduce a Model-Driven Engineering framework that supports various model transformation mechanisms to generate information systems (ISs) with a backend relational databases. Although, there are several MDE/MDA and DSL approaches, our work, to some extent, is distinct in its modelling and transformation approach, in which:

- Our transformations implement folding strategies between different views of ISs rather than translating each high-level view into a target code to represent a part of the system (user interfaces, or business classes). This means, new concepts might appear in models during the transformation to be used at the code generation phase.
- End-users are able to customise, modify and then generate their updated systems from intermediate level, rather than executing the whole transformation from the beginning (highest level), as it occurs in forward engineering.

Besides this, we are following such a liberal modelling strategy of the construction of high-level models using simple Java approach for constructing models and rules, which is more accessible than having to learn transformation languages such as ATL [33], Kermeta [52] or other OMG standards (OCL [85] and QVT [88]). Models in the proposed approach are expressed using simpler and more constrained models associated with underlying *XML* [122] representation in place of rich and unconstrained models in traditional UML [70]. The intermediate models and a final system are derived by model transformations supplied with end-user customisations or some decisions.

## 1.9 Organisation of the Thesis

The rest of this thesis is organised as follows:

**Chapter 2 - Literature Review** discusses relevant literature in the field of Model-Driven Engineering (MDE) and End-User Development (EUD). Different paradigms of model-transformation are presented and exemplified using commonly used model-transformation languages. Furthermore, several development approaches of information systems are explored with respect to a metamodel that are used, system's veiws that are considered, development stages, and the suitability to end-users.

**Chapter 3 - Framework Overview and Analysis** introduces a general overview of the proposed MDE method (BUILD) for developing information systems, including its four development phases. In addition to this, a lightweight modelling language ($\mu$ML) and the foundation of the concepts of its metamodel are presented formally using First-Order Predicate Logic (FOPL). This chapter is considered a key chapter of the thesis.

**Chapter 4 - $\mu$ML Concepts and Notations in the Requirement Sketching Phase** describes, in detail, the specification of Micro Modelling Language ($\mu$ML) models appearing in the *Requirement Sketching Phase*. For each model, the definition of its syntax and semantics is provided graphically, and formally using FOPL.

**Chapter 5 - $\mu$ML Concepts and Notations in the Analysis Phase** presents, in depth, the specification of ($\mu$ML) models appearing in the *Analysis Phase*. For each model, concept specification and notation is discussed and formalised using FOPL.

**Chapter 6 - $\mu$ML Concepts and Notations in the Design Phase** defines, in detail, the syntax and semantics of Micro Modelling Language ($\mu$ML) artefacts appearing in the *Design Phase*. For each model, concept and notation is discussed and defined formally using FOPL.

**Chapter 7 - The Architecture of Model Transformation Approach** discusses the overall structure and mechanism of model transformation approach. The design of the top level framework and the architecture of the concrete model transformation frameworks, via the various development stages of BUILD, are discussed with in detail. Moreover, the architecture of the *Code Generation Framework* is discussed, including the internal structure of the two domain-specific generators, namely, *Java Swing* and *MySQL* code generators.

**Chapter 8 - The Rules of Model Transformation** explains and discusses, in depth, mapping rules that are used to derived each element in the intermediate and design models. For each translation step, First-Order Predicate Logic statements are utilised to express formally every rule in that step.

**Chapter 9 - Case Studies** illustrates the adoption of the designed framework (BUILD) via its associated $\mu$ML models to develop a real-world information system using the *University Administration System* case study. The chapter includes three main running experiments with a demonstration of their generated intermediate models and final executable code.

**Chapter 10 - Evaluation and Testing** presents criteria to evaluate the method. Completeness and correctness of results and transformations, as well as the simplicity and the expressiveness of $\mu$ML are checked, traced and inspected using the final output produced from both *JDBC Java Swing Applications* and *MySQL* databases generators. Limitations of the proposed framework are also highlighted.

**Chapter 11 - Conclusion and Future Work** concludes the overall work presented in the thesis, including a summary of findings, contributions with respect to four main dimensions, namely, modelling language, MDE development approach, model transformation strategy, business users participation in the development process. Moreover, possible avenues for future work are highlighted briefly.

# 2

# Literature Review

"The difficulty of literature is not to write, but to write what you mean"

Robert Louis Stevenson

## 2.1 Context

This chapter reviews the literature in the field of Model-Driven Information Systems Engineering and highlights some issues that need to be considered in order to enhance end-user model driven information systems engineering.

The chapter starts first by presenting a brief overview of some of the related model-based methodologies that are used to raise the level of abstraction in software development compared with writing code. Then a more detailed overview of model-transformation paradigms is introduced, including simple examples. This part of the chapter focuses on how each language is able to express, manage the order of execution, design and compose, and reuse, rules of transformation.

The third part of the literature survey discusses some of the software engineering approaches, such as MDE, OOD, and Functional approaches for web applications, enterprise and software systems development. It highlights the features, system views (aspects), semantics and notation of their adopted modelling language (*UML Profile* or DSL). Furthermore, the last part of the literature highlights end-user rules in some of current MDE approaches that allow business user participations.

## 2.2 Overview of Related Methodologies

This section provide a general review of the related Model-based development approach, including the type of modelling language, system artefacts (views), trasformation strategies, adopting technologies and related tools.

### 2.2.1 Model Driven Architecture (MDA)

The MDA standard specification explicitly aims to integrate many Object Management Group (OMG) standards, UML [70], Object Constraint Language (OCL) [85], Meta-Object Facility (MOF) [87], and Query/View/ Transformation (QVT) [88] to produce a coherent MDE approach for managing models, and provide executable systems that are automatically-generated from specifications [93]. According to [62], MDA has three types of models: Computational-Independent Model (CIM), Platform-Independent Model (PIM), and Platform-Specific Model (PSM) [89].

The CIM model acts as use cases and feature-oriented diagrams to represent the business requirements and features [40]. It also describes the system domain and requirements, from a high perspective, without including any computational implementation. Moreover, a *Platform-Independent* Model (PIM) can be defined as a model of a system, that is completely independent of the specific technological implementation. The third model in MDA [89], the *Platform-Specific* model (PSM), is a model of a system that holds technical implementation detail relating to a specific platform or environment [89].

The PIM model is always considered a source model of the transformation program within within the MDA approach [89] approach to derive PSMs [6]. On the other hand, the PSM model, the output of the transformation phase, is considered the lowest-level model in MDA [89], which contains ideal and adequate information about a targeted platform used [15].

### 2.2.2 Model-integrated Computing (MIC)

Model Integrated Computing (MIC) [54] is an example of the Model Integrated Development (MID) approach that has been developed over the past two decades as a software development methodology for building embedded software systems. It uses models as primary entities in the development lifecycle to synthesize, analyze, integrate, test, and operate real-time systems [61]. MIC [54] realises MDE using a different strategy than MDA [89], with fairly similar concepts.

MIC [54] emphasises the employment of Domain-Specific modelling techniques (DSM) using well-tailored Domain-Specific modelling Languages (DSMLs) and multiple views of the system to model comprehensively embedded system domains [61] rather than concentrating on the usage of UML [70] only for modelling tasks as in MDA [89]. Models in MIC [54] consist of logical and functional requirements of a system as well as the physical characteristic of its hardware such as power, time, fault, and size. These physical features are included as physical properties within the platform models and are mapped to the requirements of the application models [111].

The MIC framework [54] consists of a built-in metamodelling mechanism to support the creation of DSMLs, and a meta-generation facility to create DSM tools. Instance models, constructed using these tools, can be translated into other presentations using meta-generation features [61]. The graph-based model transformation tool, GReAT [53], is considered an example of existing tools that support the MIC [54] software development.

### 2.2.3 Aspect-oriented Software Development (AOSD)

AOSD [39] is another model-based software development methodology that provides more powerful localization and encapsulation mechanisms than traditional component technologies [39, 18, 24, 11]. It aims to describe a system using multiple views, such as a data view, a security view, and a business process view, in which the design concerns are separated as different aspects. It emphasises providing a clear separation of crosscutting concerns at the models level. These concerns, together with a primary (base) model, are composed to produce an integrated view of the logical architecture of a particular system in technology-independent terms.

The model transformation process then takes the responsibility for folding these associated aspects together to form the target system. Using AOSD [39] with MDE can contribute in maintaining this separation during the vertical transformation from a high level specification model into low level implementation ones via the designing of an appropriate DSL to demonstrate the localisation ability of AOSD [7].

The *Theme* [48] approach, for example, is an AOSD [39] method that covers various software development phases, including requirements, analysis, design and implementation. This approach uses a *UML* extension, called *Theme/UML* [11] for designing of mobile, context-aware applications, which supports, via an implemented toolset, the automatic generation of PSMs and executable code from a number of generic PIMs (instead of one large PIM) [11, 18].

### 2.2.4 Executable UML (xUML)

Constructing a complete PIM model, which is totally separated from any implementation details, and then deriving one or more PSMs from it via a series of transformations until generating the final code, are considered common steps in various MDA approaches [89, 24, 54]. Stephen Mellor and Marc Balcer introduced an approach called Executable UML (xUML) [2], implemented as an object-oriented analysis (OOA) tool, that jumps from PIM level to a direct code without a model-to-model transformation step (PIM-to-PSM). They consider the executable code a PSM model that conforms to a particular language definition (grammar). The xUML [2] approach promises to express a system in a platform-independent way using rich UML [70] diagrams to represent its structural and behavioural views. As a starting point of model transformation, *UML Class* and *State Machine Diagrams* are adopted to model the structure and behaviour parts of the system respectively.

The detailed procedure, including the execution algorithm and constraints, for each state in the behaviour model are expressed formally using an Action Specification Language (ASL) [1] to express the detailed control flows of a system. This enables and supports the direct transformation from this abstracted level into the target code using a model compiler, unlike other MDA approaches that tend to derive PSMs first, as an entry point for code generation.

The ASL [1] is expressed in a higher level of abstraction than General-Purpose Languages (GPLs), which support the mapping between ASL [1] and various programming languages, such as C++, and Java. However, it is a designer responsibility to write ASL code that is syntactically and semantically correct in order to generate the correct code at the end. Therefore, writing ASL [1] code at the modelling level requires a degree of knowledge about its correct syntax and semantics. From this, it can be argued that raising the level of abstraction at ASL [1] to express the detailed behaviour of a system is not adequate for end-users who do not have this technical knowledge.

### 2.2.5 Software Factories

*Software Factories* [47] is a software development methodology introduced by Microsoft, and it is motivated by two existing software development paradigms, namely, Model-Driven Development (MDD) and Component-Based Development (CBD). According to [47], *Software Factories* is a product line that configures extensible development tools such as the Microsoft Visual Studio Team System (VSTS) with packaged content and guidance, designed for building particular types of applications.

Assembling components is the general strategy of this methodology, in which system functionalities are distributed across these components or assets to form the product family. The Integrated Development Environment (IDE) is used then to collect product family members into a *Software Factory* template and deliver it as a plug-in to the IDE.

Abstraction, as a core dimension in *Software Factory* innovation, aims to reduce complexity, hide implementations, and specify product features. Its purpose is overlapping with the purpose of raising the level of abstraction in MDE. The MDE approaches have a number of platform-specific model compilers/generators to generate implementation for target platforms from a single specification model. Therefore MDE and DSL play a role in *Software Factories* such as maintaining the synchronisation between components and constructing the system from high level specifications [47].

### 2.2.6 Generative Software Development (GSD)

Generative Software Development (GSD) is a software development methodology that aims to achieve an automatic creation of *system-family members*. The system can be generated from its high level generic specifications. The essence of this approach is based on the usage of reusable assets such as models, and components for building systems that are developed using appropriate Domain-Specific Languages. According to several resources, GSD shares some principles of the Component-based, the Software Factory [47], and the MIC [54] approaches, in which all of them seek an ideal integration of associated members to form the final system [61, 24, 47].

Besides this, GSD is considered one of the related approaches to MDE and MDA [89] due to their trend to capture system specifications and to represent them using DSLs, or UML profiles [70]. The mapping from a PIM to the PSMs in MDA matches the mapping from a Problem Space to a Solution Space in GSD. Therefore, the technologies and techniques used to construct the problem space model might contribute to the process of creating the PIM in MDA [24].

### 2.2.7 Summary

- It is clear that MDA [89] and Software Factories [47] are distinct in their delivery. Software Factories delivers plug-ins that are embedded into the used IDE as an extension forming a new product, whereas MDE and MDA [89] aim to generate a complete software system, that can run in the target environment.

- MIC and *Software Factories* are somewhat related to the Component-Based Development (CBD), for example, a product of *Software Factories* is based on assembling the product family members or assets as plug-in components to form a target system [47]. MIC [54] develops the run-time components and maps them to hardware resources to form the embedded system [61].

- Aspect-oriented Software Development (AOSD) and Model-Integrated Computing (MIC) [54] have a strong similarity in describing a system using multiple views in which they separate the design concerns in different aspects; the process of folding them together is considered a part of the model transformation process.

## 2.3 Overview of Model Transformations

Model transformation plays a significant role for supporting and achieving the automation of various Model-Driven Engineering (MDE) tasks such as refining, normalising and refactoring models, synchronizing and merging and weaving [15, 25, 81]. Model transformation is defined as a process of converting one model into another form using transformation rules acting on models at different levels of abstraction [79]. It consists of one or more defined rules for expressing the mapping between the source and target model. Tratt in [115] also defined model transformation as a program or compiler that converts one model into another. The following figure (2.1) demonstrates a general overview of the main concepts of model transformation, by showing the simplest scenario of mapping between an input (source) model and an output (target) model.

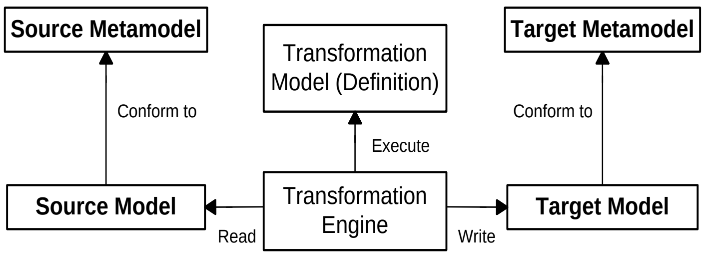



Figure 2.1: Concepts of Model Transformation

Transformation in general can be divided into three types, *endogenous, exogenous*, and *in-place*, based on the languages of the source and target models. On the one hand, the *endogenous* transformation is a transformation between models expressed in the same modelling language. Model *refactoring* and *normalising* are regarded as obvious examples of this type of transformation. On the other hand, the *Exogenous* transformation occurs between different languages as in code generation (from a model to a target code), and model translation (translate a model into an equivalent model) [79]. When having only one model involved in the transformation, it means that all modifications occur on this model, called *in-place* model transformation or modification [81].

Based on the definitions above, model transformations can be divided into two main categories: *model-to-code* and *model-to-model* transformation. The *model-to-code* mapping (code generation) is considered a special case of *model-to-model* transformation in which a detailed model that conforms to a metamodel is translated into a target programming language that defined using language grammar [81]. It is considered a final transformation stage in all MDE approaches for obtaining a complete executable code. However, *model-to-model* transformation is known as layered transformation steps for constructing models in the lowest level of abstraction. In Model-Driven Architecture (MDA) [89] for instance, the process of obtaining the

Platform-Specific Model (PSM) model from one or more Platform-Independent Model (PIM) models reflects the *model-to-model* transformation stage.

### 2.3.1 Model transformation Paradigms

Model transformations are classified into various paradigms based on the chosen presentation style of the concrete syntax of the transformation rules, which are the atomic units of the transformations considered in different approaches. The following is a brief survey of the existing model transformation classifications. The taxonomies proposed in [25, 81] and a survey conducted by [15] discusses the transformation mechanisms offered in a number of transformation paradigms and tools.

#### 2.3.1.1 Direct Manipulation

In this approach, the rules of transformation are implemented directly using one of the well-known General-Purpose Programming Language (GPLs), e.g. Java. This approach is developed as an Object-Oriented framework for facilitating model management tasks such as organising, scheduling, and controlling model transformations. The user is responsible for implementing the rules of transformation from scratch with little support from the tool [81]. Appropriate reading and writing, and possibly traversing, mechanisms are used to visit and read elements in the source model, apply transformation rules, and then attach translated elements into the output models [15]. Here we review two model transformation frameworks: SiTra [9] and JaMDA [105].

The Simple Model Transformations, SiTra [9], can be considered one example of the direct-manipulation approach. It is regarded as a minimal Java-based model transformation framework that consists of a simple Java library for supporting the implementation of algorithms that executes a transformation based on rules. SiTra [9] is not introduced as an alternative to other model transformation paradigms; it only aims to introduce the concepts of transformation rules for experienced programmers without the need to learn any transformation language. It is not designed as a new transformation specification or full transformation framework [5].

SiTra [9] presents transformation rules by considering two interfaces (*Rule* and *Transformer*) and a class for implementing a transformation algorithm. In this regard, they provide a standard way of representing rules and transformations using pure Java code (classes) to mimic the behaviour of rule-based transformation system. The Transformer interface is implemented by transformation algorithm class [5], which is based on pattern-matching rules that implements the Rule interface.

The *Rule* interface is implemented using Java in which a transformation is broken up into a number of rules. This interface includes different methods for checking whether or not the rule is applicable to an input element from the source model, and constructing an output element in the target model. Moreover, the implementation consists of four main methods (two methods for invoking rules, and two methods for mapping between the objects in the source and target models). It takes a transformation rule and an object expected to be transformed as parameters and executes the algorithm explicitly [5]. Listing 2.1 demonstrates the definition of a rule that maps a class to a relational table [9].

Listing 2.1: An example of a transformation class written in Java

```
class Class2Table {
    Table build(Class cls) {
       Table tbl = new Table( cls.getName() );
       for(Attribute att: cls.getAttribute()) {
          tbl.addColumn(
          new Column( att.getName(), att.getType()
             .getName())
          );
       }
       return tbl;
    }
 }
```

As SiTra [9] is introduced as a framework to support the implementation of simple model transformations, its limitations appear when dealing with complex transformations. For instance, in the case of having multiple rules map to the same target object, SiTra [9] has no way to determine which rule is responsible for target object construction and which rule retrieves the object from the target model. Thus, manual contributions, by the designer, are required to decide the creation rule [5].

The second example reviewed here is the Java Model-Driven Architecture, JaMDA [105]. It is another object-oriented framework for generating Java code from UML diagrams [70] of core business classes using their underlying representations (XMI standards). According to [105], the developer usually writes a business logic code in the target programming language, which can be merged into the generated system. It is expected that the developers manually write about 20% of the total code.

JaMDA [105] is noted as a model compiler that follows the direct-manipulation technique of model transformations that consists of a library of independent transformers. Each transformer is designed to handle a special type of transformation [81, 105]. Thus a user can implement horizontal transformations that can optimise a model or vertical transformations that can refine a model towards code. Jamda implements transformers and generators using a visitor-based approach where the input model is traversed and each element has the appropriate transformer to apply the mapping rules [105].

Jamda can be said to be platform-independent, as it is written in Java and produces the source code in Java. However, it does not output any other executable programming languages. For instance, it is used for generating N-tier web applications based on Enterprise Java Beans technology (EJB). Complete UML models [70] that represent the whole application are used to generate each tier. Each tier has a collection of generated java classes that have been derived from the initial UML [70] input models and classes of the previous tier [105].

#### 2.3.1.2 Model transformation approaches for Declarative (Relational) rules.

In this approach, the rules of transformation are expressed declaratively using a set of constraints to specify what the relations between source and target models are [110]. A number of languages and tools that support this relational approach are used, such as QVT Relations [95], Tefkat [73], and XMF-Mosaic. Here we review in detail the QVT Relations language [95].

The QVT Relations language [95] is considered a domain-specific model transformation language that expresses the mapping logic between two models as a set of relations. The relationships between source and target elements are encoded as pattern-matching expressions including predicates (preconditions) that must be satisfied by the input elements [95, 44]. The precondition normally consists of two expressions: an enabling (when clause) that returns a Boolean value, and an enforced (where clause) expression that causes side effects on the target model [95]. Unlike other declarative graph-based model transformation frameworks, namely, GReAT [53], VIATRA2 [32], and AGG [112], that provides unidirectional transformations only, the QVT Relations supports both unidirectional and bidirectional transformations [95]. Listing 2.2 illustrates snapshots of a transformation program writing in Tefkat [73], a model transformation language based on F-logic, to express the relation between persistent UML [70] classes and RDBMS tables.

Listing 2.2: An example of the declarative representation of rules using QVT-Relations

```
transformation umlRdbms (uml : SimpleUML, rdbms:
                         SimpleRDBMS){
    relation ClassToTable {
        enforce domain uml c:Class {
            namespace = p:Package {};
            kind = 'Persistent';
            name = cn;
            };
        enforce domain rdbms t:Table {
            schema = s:Schema {};
            name = cn;
            };
    when { PackageToSchema(p, s); }
    where { AttributeToColumn(c, t); }
    } }
```

Listing 2.3 demonstrates a transformation rule for translating OO *Class* to a relational *Table* using QVT-Relations [95].

Listing 2.3: An example of the declarative representation of rules using Tefkat

```
RULE ClassAndTable(C, T)
    FORALL Class C {
    is_persistent: true;
    name: N;
    }
    MAKE Table T {
    Name: N;
    }
    LINKING ClsToTbl WITH
         class = C, table = T;
```

In the case of having more than one transformation rule to derive a particular element in the target model, QVT-Relations takes into account the ability to check whether or not a target object is already created by one rule in the target model. As a result the second rule only updates the created element rather than constructing it twice. Unlike the direct-manipulation framework (SiTra [9]), QVT Relations [95] may offer a strategy to avoid any duplicate creation, or deletion task of target elements and replaces it by an appropriate update task. This is achieved by introducing the notion of identity to target elements [95, 44]. According to the UML-to-RDBMS case study presented in [95], QVT-Relations adopts the concept of a *key* from relational databases [95] to handle this issue. This new concept is described as a set of properties, or fields, that uniquely identify an object instance of the class in a model.

### Model transformation for Graph-based rules

As a formal approach of the declarative model transformation strategy, Graph Transformation has been widely utilised for expressing model transformations. It provides a theoretical foundation and a formal flavour to transformation frameworks, so, in general usage, source and target models can be expressed visually and textually using the notion of a graph (graph-like structure). The graph consists of a set of vertices $V = v1, v2, , v_n$ and edges $E = e1, e2, , e_n$ in which each edge $e$ in $E$ has a $source(s)$ and $target(t)$, vertex $s(e)$ and $t(e)$ in $V$, respectively. Several types of graphs, such as typed graphs and instance graphs, are specially designed to represent the abstract syntax of diagrammatic class, and the instance models respectively, e.g. a metamodel of the UML Class diagram and an Object diagram [25, 12, 114]. Figure 2.2 illustrates and example of the representation of a system using the visual notation of a graph [110].

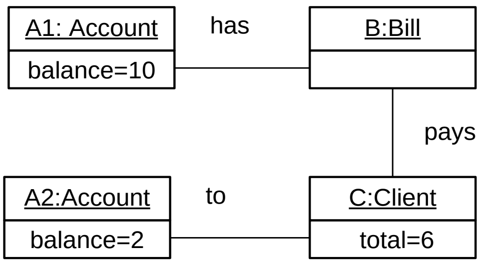
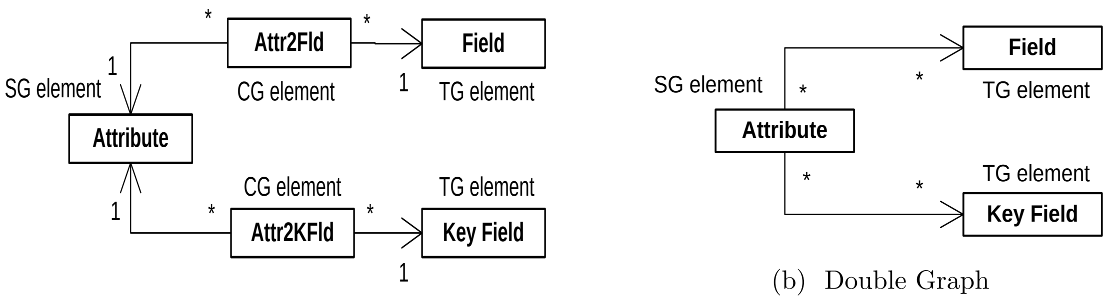



Figure 2.2: A graph representation of an Object Diagram

A graph transformation rule ($GT$) between models is formally considered a function $f : L \longrightarrow R$ that maps elements in a source graph $SG$ to elements in the target one $TG$. The transformation typically consists of a pair of graph patterns [25], left-hand side and right-hand side graph $GT = (L, R)$ in which the union $L \cup R$ is defined. For instance, edges that appear in both $L$ and $R$ are connected to the same vertices in both graphs. The left-hand side pattern is considered a sub-graph of the source model that describes the precondition of a particular rule, whereas the right-hand side pattern is regarded as a sub-graph of the target model that describes the post-condition of the rule [15, 12].

The above technique is successful in *one-to-one* and *many-to-one* mappings between source and target graphs. However, in the case of *many-to-many*, the transformation (function) has to be reconsidered as a *relation* ($R$) in order to allow this kind of mappings. The Triple Graph Grammars ($TGG$) [63] technique introduces a solution of this issue to represent the concrete syntax of transformation rules as a graph-like structure. Each mapping rule consists of two graphs (left and right), as well as a Correspondence Graph $TGG = (L, C, R)$ to enable the double push out strategy between the three graphs. This strategy has been adopted in many graph-based model transformation tools such as AGG [112]. For example, in the basic mapping between *Attributes* and *Fields* in the *Class-to-RDBMS* transformation, the many-to-many mapping can be found in translating *Attributes* to either columns or key columns. Figure 2.3(A) shows how *many-to-many* mapping is splitting up into two *many-to-ones using correspondence* graphs (*Attr2Fld,* and *Attr2KFld*), whereas this is not obvious in the (B) side of Figure 2.3.




(a) Triple Graph Grammar

(b) Double Graph

Figure 2.3: Example to M-to-M mapping using TGG

There are a number of frameworks and tools for graph-based model transformation, such as Fujaba [114], VIATRA2 [32], AGG [112] and GReAT [53]. Fujaba [114] is considered one of the most popular general-purpose graph transformation tools [98] for generating Java code from the UML specification [70] and for round-trip engineering. It has the ability to transform many source models to many target models. It provides an endogenous transformation only [61, 80, 121] in which the source and the target models conform to the same metamodel. Fujaba [114] uses the graph technology to represent the underlying representation of UML [70] diagrams with formal foundations. The transformation rules that are declared using the TGG [63] technique are implemented as method bodies with a control structure [114].

Fujaba [114] is able to generate code from the *UML Class Diagram*, a collection of *Story Diagrams*, and rules of transformation expressed using TGG [63]. The *Story Diagrams* are used to model the system behaviour. A story is used to specify the body of methods related to a class. Whereas, the *UML Class Diagrams* are used to model the structure part of the system. Consequently, a code generator can generate Java code for these story and class diagrams [13].

#### 2.3.1.3 Model transformation approaches for Imperative (Operational) rules

**KerMeta**

KerMeta [52] is considered an Object-Oriented meta language that is used to specify the structure and concrete syntax of models with the ability to define their behaviour using operational semantics. It is fully integrated into Eclipse, in which it offers an EMF-based metamodel, and is used as a MDA [89] model transformation language or tool to specify the mappings between source and target models [52, 57]. It is an extension to Essential MOF (EMOF) with an action language. The language is designed based on two existing languages, namely Xion, and MTL. Therefore many features of these two languages, such as the action language (Xion), and multiple inheritance (MTL), can be re-expressed in KerMeta [84].

This action language is an imperative OO language used for defining operations and constraints [55] in metamodels. It is also used for implementing executable metamodels to provide all model management tasks [57, 65], and for performing transformations [15]. KerMeta [52] supports many OO language features such as static typing, and multiple inheritance, and also it supports more specific concepts to be suitable for modelling and model transformation tasks. For example, many OCL [85] operations such as *collect, select,* and each are available in KerMeta [52] to apply such a condition and dealing with a certain collection (Listing 2.4).

Listing 2.4: An example of the representation of the transformation rules using KerMeta

```
getAllClasses(inputModel)
    .select{c| c.is_persistent}
    .each{c| var table:Table init Table.new
          Table.name:= c.name
          Class2Table.storeTrace(c,table)
          Result.table.add{table}
    }
```

#### 2.3.1.4 Model transformation approaches for Hybrid rules

**AtlanMod Transformation Language (ATL)**

ATL [33] is a MOF-based model transformation language that supports both imperative and declarative representation (hybrid) of mapping models (Figure 2.5). This allows users to express a complex transformation that is hard represent declaratively using an ATL [33] imperative structure. Its representation of mapping rules forms a transformation model (program) that describes how to create the output model of transformation [15]. ATL transformation models are implemented as ATL Modules that consist of a number of transformation rule definitions (rules), and some helper methods. Listing 2.5 demonstrates a snapshot of an ATL module for defining the transformation between OO Classes to Relational Tables.

The ATL [33] approach delivers only part of what MDE promises, in that it supports only the model-to-model transformation [33]. ATL [33] is not directly based on the mathematical foundation of graph transformation techniques [59]. ATL [33] has abstract and concrete syntax that are expressed textually [15, 41], and a particular compiler and virtual machine [110]. It is essential to make a clear distinction between source and target models, since ATL [33] cannot use the same model for both. The input model here is considered a read-only model. The navigation task on the input model is achieved using ATL query units. These queries define the navigation over one or more source models to produce a value. Conversely, the output model is not navigated and considered a write-only model [123].

Listing 2.5: An example of the representation of the transformation rules using ATL

```
module Class2Relational;
create OUT : Relational from IN : Class;

rule ClassAttribute2Column {
    from
        a : Class!Attribute (
        a.type.oclIsKindOf(Class!Class) and not
            a.multiValued)
    to
        out : Relational!Column (
        name <- a.name + 'Id',
        type <- thisModule.objectIdType)
}
```

### Epsilon Transformation Language (ETL)

Epsilon [34] provides a framework called the Epsilon Model Management Infrastructure that consists of a family of task-specific modelling languages such as the Epsilon Merging Language (EML), Epsilon Comparison Language (ECL), and Epsilon Transformation Language (ETL) [34]. The ETL [66] is another hybrid model-to-model transformation language that is designed in such a way to be integrated smoothly with other task-specific languages within Epsilon [34, 100]. Unlike ATL [33] that is designed to perform many model management tasks such as model validation and merging, ETL [66] is a DSL for implementing transformations [67].

Listing 2.6: An example of the representation of the transformation rules using ETL

```

rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.name = t.label;

  // If t is not the top tree create an edge connecting n
  // with the Node created from t's parent

  if (t.parent.isDefined()) {
    var e : new Graph!Edge;
    e.source ::= t.parent;
    e.target = n;
    }
}
```

Epsilon has an OCL-based core language, *Epsilon Object Language (EOL)* [64], that is used in developing the Epsilon family of task-specific modelling languages. The ETL [66] uses the navigational features and other imperative features of EOL [64], such as accessibility to multiple models, model modifications, programming constructs (e.g. loops, branches) to express complex transformations [66]. Similar to ATL [33], transformations in ETL [66] are implemented as a number of *modules*, containing a set of *rules* and *operations* to support the imperative side of the rule [66]. Listing 2.6 above illustrates a snapshot of an ETL [66] module for defining the transformation from a *Tree* node to a *Graph* node.

#### 2.3.1.5 Rule Scheduling Mechanisms

The order of rules execution can be specified using two styles: *implicit* and *explicit* forms. In the implicit style, rules call another rule without referencing explicitly the rule name; that is, in which the calling mechanism is based on rule dependencies. The transformation engine has a scheduling algorithm that determines the order. In general, a rule A is executed when another rule B is waiting for the result produced by the rule A. This mechanism is common in declarative model transformation languages such as QVT-Relations discussed above [69].

In contrast, the explicit style might use control flow structure within a rule including explicit rule invocations (*internal*). It also can be *external* by using logic to separate transformation rules without a direct invoking of each other. This style is usually found in imperative and hybrid model transformation languages such as ATL [69] and ETL [66].

#### 2.3.1.6 Template-based Approach For Code Generation

According to [15, 25], a template-based approach is widely adopted in the majority of MDA [89] tools such as AndroMDA [8], OptimalJ [19], and Acceleo [3], for supporting the model-to-text transformation. A template of a target model that contains a meta-code is accessed, mapped, and filled from the source model. The structure of the template is closely related to the generated code [26].

**AndroMDA**

AndroMDA [8] is an open-source Model-Driven Architecture (MDA) framework. It takes one or more graphical UML models [70] and produces a target component in one of the J2EE technologies such as Java, PhP, and EJB. In the MDA [89] process, AndroMDA [8] takes the Platform-Independent Model (PIM), resulting from the analysis phase, and refines it to construct a Platform-Specific Model (PSM) for a target technology with a template for producing the final code.

AndroMDA is regarded as a template-based approach for code generation that is able to generate different code based on the language of the template [8] using language-specific cartridges. The user can customise these template files to produce a source code in any programming language. The metamodel in AndroMDA [8] is based on MOF [87]. The cartridges are designed to get the information necessary to generate a target code from MOF models inside a MOF repository. The AndroMDA Cartridges, primary plug-ins in the framework, are designed to perform this process by parsing the underlying XMI representation of UML diagrams collaborating with transformation libraries. These cartridges consist of code templates that are written using the Apache Velocity template engine [8].

**Acceleo**

Acceleo [3] is a template-based MDA [89] code generator approach for generating code for various platforms such as Java, C++, C#, PHP, JEE, and Python. It is fully integrated with Eclipse and the EMF framework (GMP) in which it offers a user-friendly template editor for organising the template of target code. The framework consists of a number of separate technology-specific modules for code generation. Each module is created from templates that express the information required to generate executable code from a metamodel. Within each template, several scripts allows the user to customise the generator accurately [3]. The basic idea of this approach is that the generation program (script) is applied to a source model, which conforms to an EMF based metamodel, to fill the target template.

The modularity supported in the Acceleo architecture enables adding new generators as plug-ins without any problem, which accelerates the implementation process in general [3]. It could be argued that Acceleo can deliver a complete MDA [89] solution or implementation when it is integrated with other Eclipse-based tools such as ATL that supports model-to-model transformation.

**ZOOM**

Z-based Object-Oriented Modelling notation (ZOOM) is a model-driven engineering framework that is based on a formal modelling notation. It aims at improving the quality and productivity of software development processes by handling existing drawbacks, such as incomplete modelling notations, and the lack of an effective model transformation mechanism. It is a template-based model transformation approach that is supported by a CASE tool, called Hierarchical Relational Metamodel Transformation (HRMT) [56]. Transformation templates (cartridges) are considered a collection of transformation rules that define the mapping between source and target models. The framework consists of a set of templates; each one is used for performing a specific transformation task on a target platform [76]. End users are able to add their preference only at user interface generation [56].

ZOOM has a simplified metamodel called Hierarchical Relational Metamodel (HRM) that maintains a tree structure and relationships representation between PIM elements. The modelling notation, which is consistent with UML 2 [70], has a textual syntax defined by BNF and a similar internal representation of models to programming languages, which is an Abstract Syntax Tree (AST). It uses mathematical collections to depict such complicated modelling language constructs as associations [76].

The functional requirements in the framework derive the structural, behavioural and UI models. ZOOM provides a pre-defined event model, which is processed by an event-driven framework, to bind the structural, behavioural, and UI models together. The integrated ZOOM model will be processed by the Knowledge-based Model Compilation Tools resulting in different implementations of the software system based on the specific platform and knowledge base. To start the MDE process, the developer needs to build a platform-independent model and other models are mostly generated using several steps. Firstly, the textual representation of ZOOM notation is parsed into an abstract syntax tree (AST). Then a post-order traversing mechanism is used to visit all elements in the AST and applies mapping rules to generate the code [56].

### 2.3.2 Composition of Model Transformations

Model transformation plays a significant role for supporting and achieving the automation of various Model-Driven Engineering (MDE) tasks, such as creating, refining and refactoring models, as well as synchronizing and merging and weaving. As a result of its critical roles within various MDE tasks that are focused on the mapping, checking and validation of models, the transformation units are becoming more complex and harder maintain. Therefore, the composition of model transformations has emerged as an important research topic in its own right, allied to model transformation in general. Tasks include designing a suitable orchestrating mechanism for controlling the execution of the decomposed transformations, in order to produce a single consistent result.

Many composition techniques have appeared to handle this issue. For instance, one technique is achieved by simply linking several pre-designed model transformations (external) whether they are expressed using one or different languages and executed by one or several tools. On the other hand, the other technique is based on decomposing the rules of transformation into independent units (internal) to collaborate as a big transformation unit to perform a complete transformation [46]. Here, we discuss some of the limitations that exist in the current model transformation approaches, particularly, the rule-based and direct manipulation approach, and review an example of a composition technique in the graph-based and rule-based, model transformation paradigm.

#### 2.3.2.1 Composition Techniques

Hidaka et al. [50] developed a composition strategy for a graph-based model transformation approach, inspired by the Unstructured Data Query Language (UnQL), the compositional graph querying language. UnQL is a powerful query language that is based on a pattern-matching technique to express queries in a structural recursion style. This style enables the composition of two or more queries that are designed in a structural recursion way to be expressed as a single query [17].

Hidaka et al. [50] extended the UnQL by adding three editing structures that support the direct specification of graph transformation, namely, deleting, extending, and replacing a subgraph. At the end, they came up with a graph transformation language UNQL+ that supports the composition of graph-based model transformation. This homogenous composition approach is implemented as a framework that supports the development of model transformations in-the-large. In this instance, a large model transformation can be systematically designed by gluing simpler model transformation units together via efficient intermediate models, as illustrated in the Class-to-RBDMS transformation case study [50].

When transforming attributes to columns of a particular table, there is a need to gather all information from directly and indirectly associated classes that have relationships with that table. Therefore, designing an intermediate model to associate directly all indirect associated classes to a particular table as a simple transformation (query) is an example of efficient intermediate models. From that query result, it is possible to generate directly primary/foreign keys for the specific table.

Wagelaar [123] proposes a technique for rule-based model transformations which he calls Module Superimposition. It is implemented using the ATL transformation language [33]. Module Superimposition is an internal composition style that allows for the constructing of smaller maintainable and reusable transformation modules, based on transformation rules, to perform together a mega transformation.

The basic idea of this technique aims at splitting up transformation modules into modules of manageable size, which each consists of a number of rules that are superimposed on top of each other and are executed as one rule. It is equivalent to applying a union with override operation to transformation rules that have the same names in a pair of transformation modules, in which the original rule is replaced (overridden) by the new one. The rule then will be executed once. Therefore, the approach is about a rule overriding technique that operates at the ATL helper methods and helper attributes levels [33]. It can also work in the QVT Relations [95] language by considering the transformation rules (relations) as the atomic level of modularity [123].

Goknil et al. [46] provide a composition approach based on two levels of granularity using two transformation languages (ATL [33] and Tefkat [73]). The first level is a traditional rule-based composition supported by ATL [33], whereas the second one is an operation-based composition for handling complex structures in the source model, supported by Tefkat [73]. The external level of composition aims to improve the quality of transformation rules by reducing the number of rules used in each transformation module. It considers the basic transformation operations such as *Create, Read, Update,* and *Delete* operation as atomic parts of transformation instead of whole transformation rules.

The idea behind this approach is to avoid the problem of scattering changes on several mapping rules when a change has occurred on the source pattern. This problem affects negatively the consistency and modifiability of transformation rules. Their solution, proposed here, is to consider the source's complex structure (pattern) as a distinct construct and then map it into a single module provided by a language that supports the finer grained decomposition (e.g. Tefkat [73]); then it will be executed as a single transformation rule. In rule-based model transformation languages such as ATL, the transformation rule consists of three patterns, namely, *source pattern (SP), target pattern (TP)* and *action*.

$$TR = SP, TP, Action$$

The source and target patterns are elements in the input and output models, which might be simple or complex constructs, whereas the *action* is an atomic part of the rule, which might include one or more operations, such as the basic CRUD operations. These operations might be composed into a single transformation rule, which is not supported in ATL (*action* part,which cannot be composed of multiple operations). However, using another model transformation language such as Tefkat [73] enables having multiple operations to be composed as one *action* part in a single rule. The *action* performs one or more *operations*, which part have three patterns, namely, *source pattern (SP), target pattern (TP)* and operation type (OpTy). The types of composition operations are specified in the *operation* type part.

$$Action = (Op1 + Op2 + \quad + Opn)$$
$$Operation = SP, TP, OpTy$$

From that we achieve two compositions at different levels of granularity. The first composition is between the transformation module (specification) and rules, whereas the second level is between the *action* part of the rule and basic operations.

### 2.3.3 Limitations in the Current Model Transformation Approaches

According to [119], the rule-based model transformation approach suffers from several deficiencies that prevent a full implementation of reusable transformation components forming as a transformation chain. For instance, in the Relational approach, e.g. MTF, where declarative mapping rules express the relationships between source and target elements, the transformation can be initiated from any of these rules resulting in (producing) different outputs. Therefore, there is a need to determine an initial rule and also the direction of transformation (forward, or backward).

Furthermore, in the hybrid model transformation approach (ATL) where the transformations might be considered metamodel-independent, there is a need to select concrete metamodels when executing a transformation. This requires studying the implementations in order to identify a set of valid metamodels for each mapping rule. From these it can be said that the problem of having incomplete knowledge of the transformation's implementation exists in various types of rule-based approaches.

The direct-manipulation approach also has limitations in developing reusable transformations; particularly where there is a demand to specify the entry point of the transformation and to have complete implementation knowledge in order to use the designed transformations. As seen in SiTra [9] and ReMoDeL [101], each framework adopts a specific way to use models and to implement the mapping rules using the general-purpose programming language, Java. There is no standard technique to develop, construct, manipulate and specify source and target models. Therefore, extra specifications of a transformation are required in order to design reusable transformation components.

Unlike the declarative rule-based approach, the entry point (initial rule) of a transformation and the order of rule execution are imperatively determined in the old version of the ReMoDeL Database Generator [108] by the transformation engineer during the creation of the transformation. The framework handles this issue using an execution algorithm implemented imperatively as a series of method invocations in the top class of the translators/generators hierarchy, and controlled invocations within other methods in other sub- translators/generators classes. The translation algorithm is generic for all database normalisation scenarios, in which we follow logical normalisation steps taken by database designers, whereas it might be varied in generating constraints based on the target database vendor (e.g. generating triggers instead of field constraints).

### 2.3.4 Summary

This part of the survey discussed various model transformation paradigms and languages. In the direct manipulation approach, two frameworks are considered, namely SiTra [9] and JaMDA [105]. For both approaches, the use of Java for encoding the transformation rules, as well as the design of transformers (Java classes) are discussed.

Regarding the rule-based transformation style, the *imperative, declarative* and *hybrid* approaches are demonstrated with some transformation language examples. For each approach, the policy for rule definition, rule ordering and rule invocation are discussed and supported by small examples to show the ability of the language for expressing the transformation.

In relation to the above, the work developed in this thesis adopts the *hybrid* approach for expressing transformation rules using *Java*. The related works are reviewed in this part of the survey. Any rule may be a top-rule; and there is no distinction (as with some other MT approaches) between top-rules and dependent rules. The following table (2.1) summarises general model transformation features (reviewed in the literature) that are adopted in the proposed method in the thesis.

| **Approach** | **Comments** |
|---|---|
| **Modularity** | A vertical modularisation (composition) strategy is adpoted to minimise to bridge the semantic gap between high-level requirement models and low-level design ones, then generating code dirctly from them. |
| **Reusability** | Each transformation component (agent or rule) is independent. Any rule might be applied in any order (e.g. top or dependent) |
| **Rule Scheduling** | A transformation algorithm is specified by a designer to determine the order of execution. This is based on dependencies between rules. |
| **MT language & Technology** | The proposed method aims at using simple and available technologies, Java and XML, for expressing features of the hybrid transformation style and model concepts. |

Table 2.1: Features of Model Transformations

## 2.4 Software System Modelling Techniques

Whereas the above considered approaches used in model transformation, below we consider general modelling approaches in software engineering. We select those approaches which bear some affinity with model-driven engineering.

### 2.4.1 Object-Process Methodology (OPM)

Object-Process Methodology (OPM) [31] is regarded as a holistic approach for system development. It is supported by a CASE tool (OPCAT) [90] for enabling the generation of complete systems from specifications. It integrates the object-oriented and process-oriented paradigms to introduce a *single-view* MDE approach. This approach produces a size-wise (fewer lines) code in contrast to other approaches based on UML models. This is demonstrated in the comparison between the UML CASE tool *Rhapsody* [51], invented by *ILogix*, and *OPCAT* [90] tool proposed in [96].

In a multi-view approach, the generated code from each view is limited and represents a partial code that needs be filled by further details generated from other views. This might cause inconsistencies in the code output by different generators, as well as causing incompetence of generated behaviour. OPM [31] tries to tackle this issue due to the crosscutting representation of the system specifications using multiple views. This is achieved by providing a complete model of a system in a single model (view) [96].

The OPM [31] model consists of a set of interconnected OPM [31] concepts or entities, represented graphically via a workflow-like Object-Process (graph) Diagrams (OPD), and textually via a dual-purpose and natural-like Object-Process Language (OPL) based on context-free grammar [60]. The metamodel introduces two general concepts (*things* with states, and *links*). The *thing* concept is to represent an object or a process, and the *link* is used to express structural or procedural connections between *things* to connect objects to processes. Processes are considered as transformation behaviours for translating objects [60]. Therefore, it can be argued that OPM [31] has a notation that combines static and dynamic aspects of the system in one view, which facilitates generating a complete code at the end rather than code skeletons [96].

OPM [31] has its refinement/abstraction mechanisms that enable the representation of a complete system at different levels of abstraction without losing the overall consistency of its internal components. These techniques are applied to OPM [31] entities as follows:

- Folding/unfolding mechanism for refining/abstracting the structure hierarchy of OPM [31] objects.
- In-zooming/out-zooming for hiding/exposing the inner details of OPM processes.
- State expressing/suppressing for hiding/exposing the state of OPM objects.

As the OPM methodology [31] is supported by a CASE tool (OPCAT) [90], the OPM-GCG is the generator component for generating generic code. It consists of two main parts, namely, Template for Implementation Programming (OPCAT TIP), and Implementation Generator [96].

The OPCAT TIP enables users to encode transformation rules in a language-specific template (Template and Translation DB). The translation is expressed imperatively as an ordered set of operations that contains conditions and actions (Event-Condition-Action (ECA) paradigms). The TIP then generates corresponding XML [122] files. These XML [122] documents are used as inputs to the Implementation Generator [96].

The system uses the underlying representation of the OPM model, which is an OPL-XML [122] script (automatically generated from the graphical notation OPD), with a template to create the actual system, including UI, code, and database schema. The Implementation Generator takes a language-specific OPL template and OPL-XML [122] scripts as inputs, and captures the match between corresponding elements. It executes required operations, and then generates programming languages encoded into XML [122] format. The next step is parsing these output XML [122] files of the system, which is based on user preferences to select transformation options: code (e.g. Java), mark-up (e.g. HTML), or comment [96, 31].

### 2.4.2 Pure Object-Oriented Method

It is a widely-recognised that a multi-view modelling approach is adopted by various development methodologies and tools, such as the Model-Driven Architecture (MDA) [89]. Generally, in the UML-based approach, each UML [70] model is used for representing a system from a different angle; for example, the approach proposed earlier by Chow et al. [20]. is regarded as a two-stage multi-view MDE approach for generating Java code based on five UML models [70], namely, *Class, Component, Statechart, Sequence,* and *Activity diagrams*. The approach captures the static structure of a system at the first stage and then adds the behaviour in the second one [20].

Rhapsody [51] by ILogix is another example that is able to generate partial behaviour of a system. It uses the *UML Class* and *Statechart* [49] with translation rules for generating the system, the UML Interaction diagrams to define objects and their interconnections, and the *UML Use case* and *Activity diagrams* for analysing and documenting the system [96]. In both examples, it can be argued that maintaining the consistency between these UML views (models) is not a trivial task.

#### ZOOM Approach

ZOOM separates software modelling into three components: the structural, behavioural and User Interface (UI) models. This separation of concerns allows each aspect of the system to be specified separately, and makes it possible for us to use an appropriate formal specification language to specify each of these unique aspects. The Structural model is defined formally, and visually represented by UML 2 class diagrams. It is completely separated from the other aspects.

Besides this, the Behaviour model is regarded as a communication component that links the structural model with the UI model. It is formalised using state diagrams. It is represented visually using a *UML 2 Statechart diagram*, and textually using a *UML 2 Finite State Machine (FSM)* that contains rich syntactical grammar with formal semantics. Therefore, the behaviour model will consist of a number of FSMs for different user behaviours. The changes (transitions) from one state to another are based on user inputs and commands.

ZOOM uses a separate model for representing the UI. As a result, any change of the interface specification will lead to a change in the UI models only without any crosscutting effect in other types of model. The UI model is formally defined using pre-defined XML [122] schemas ZOOM-UIDL as a textual and concrete syntax.

### 2.4.3 Integrated Methods

The examples mentioned above are based only on the OO method for modelling system specifications. Several attempts have been made to integrate the OO method with the Process-Oriented and Structured (Functional) Methods to enable better and more comprehensive capturing of the system specifications at the design level. Here, we consider some of them to show how this integration contributes to the MDE strategy.

#### FOOM Approach

Functional and Object-Oriented Methodology (FOOM) is an analysis and design methodology for developing Information Systems (IS). It combines two well-known software engineering paradigms, namely, the OO approach and the Functional approach. FOOM aims to cover the structural and behaviour representations of ISs [60].

FOOM adopts the ADISSA methodology (*Architectural Design of Information Systems*) that extends the *Structured Analysis Methodology*, during the analysis and design stages. ADISSA is used to extract: (a) menu-tree interface, as an external view of the system to users, and (b) system basic transactions, as an internal view to represent the user interaction with the system via different events. In addition to this, (c) database schema, and (d) system Input/Outputs are also obtained from above using the ADISSA method, which is supported by a CASE tool [60].

The analysis phase of FOOM produces a data model and a functional model. The data model is noted as an initial class diagram without methods. It represents only real-world data entities derived from the requirement analysis stage. The detailed methods will be added later in the design phase. The functional model is considered a hierarchical *Object-Oriented Data Flow Diagram (OO-DFD)* that includes initial classes instead of data stores (external entities). It is used to specify the functional requirements of a system. Like traditional DFD, the OO-DFD consists of a number of functions (decomposable and elementary), and external entities (e.g. user, time entities) [60].

The design phase produces a complete class diagram and user interfaces (UIs) based on the ADISSA methodology. The class diagram contains detailed descriptions of methods, including input and output screens. This phase is organised in sub-levels, one for extracting system transactions from the OO-DFD model. Each transaction contains a chain of functions and external entities to perform a particular process of the system. This will produce the detailed descriptions of the transaction-methods (high-level descriptions) attached to classes. The process logic of transaction at this level is expressed textually using standard structured programming such as pseudo code, or visually using message charts [60].

The next step is constructing a menu-tree interface that is derived from the OO-DFD (the second level according to ADISSA methodology). A menu item is created for each function connected to a user entity and attached in a parent menu forming a hierarchal menu tree. A generic class Menu will be added to the Class diagram, and all extracted menus become objects of it [60].

Input and output screens are designed in the third step. For each input/output command in the high-level descriptions of transaction, there is a form and report screen respectively. Generic *form* and *report* classes will be added to the class diagram and all form and report screens become become instances of these classes. Then behaviour methods are designed in the last (fourth) step by converting the high-level transaction descriptions into a detailed description of methods, such as CRUD operations [60].

### MOSYS Approach

In addition to this, MOSYS is another integrated approach that is aimed at the construction of Distributed Real-Time Systems (DRTS). It combines the functional and OO methodology and uses UML [70] with an extended DFD (E-DFD) in order to enable the automatic identification of objects and classes from high-level specifications. The components of E-DFD are weighted processes and weighted attributes. The weights are used to model the time constraints and complexity of processes [13].

After identifying the entities that interact with the system, a use case model is created to describe the system functionality. Then activity diagrams or E-DFDs are used to model the use case functionality. Then the functional model (e.g. E-DFD) is mapped into a graph that consists of edges and tasks (nodes). Based on clustering criteria, different object identifications might be obtained [13].

Besides that, [38] a practical technique in integrating notations of UML [70] and the DFD is proposed that aimed at the development of embedded systems. The combined approach is introduced as a part of an MDE process in which a system is modelled using different views; model transformations then force the integration rules between these views. In this approach, the designed DFDs can be transformed into UML object or class diagrams according to the transformation algorithm.

The development process consists of a series of mapping between different models (views). It starts decomposing each use case of an embedded system into three objects: *data, control* and *interface*. Therefore, a transformation step is performed on the use case model to produce an initial object diagram (IOD) including refactoring processes on it. Then the data flow model is obtained from the initial object diagram representing different details of the system. This step involves specifying data stores and processes, and identifying the connection between them [38]. Then an object-oriented model is derived from the constructed DFD using one of the following approaches:

- Applying a direct mapping to an object diagram, in which processes and data store will be mapped into objects.
- Applying advanced mapping into a class diagram, which classifies processes and data stores within the DFD based on their type. Processes become logical methods, whereas each data store is translated into a separate class.

### DFD net Approach

Another novel approach has been introduced by [113] to complement the OO method with functional decomposition for realising uses case using extended DFDs (DFD net) at the analysis stage. This means that all processes and data flows are transformed into OO classes, including attributes and operations, at the design stage.

In DFD net, a distinction between various data flows has been considered. For instance, distinctions between main-input, inter-process, and non-inter-process data flows are made using double-solid, single-solid, and single-open arrowhead respectively. The process starts by specifying a main-input data flow for a collection of primitive processes in the lowest-level DFD net and connecting them with a local buffer to form a single group. Similarly, processes that share the same data as input or output are grouped together, then classes are generated for each separate group. The development process at the design stage is automatable and supported by a tool to perform transformations [113].

## 2.5 Metamodelling approaches for Automatic Generation of Information Systems

This section aims to highlight different metamodelling approaches for generating information systems. It considers system aspects that are taken into account during different modelling stages (analysis and design), as well as the degree of automation for generating a complete executable code, user interface and business logic. A domain-specific language framework (MOD4J) [77] for developing administrative enterprise applications, and a framework for developing Data-Driven Applications (XPage) are not included in this survey.

Different metamodelling approaches are proposed for developing web-based information systems (IS) such as [28, 118]. These approaches are normally based on capturing the structural aspects of ISs (entities, relationships, and constraints) using metamodels or appropriate domain-specific languages. They allow the automatic generation of user interfaces (UI) and in other cases basic CRUD operations. These approaches have remarkable limitations in which they suffer from the lack of business logic support and UI development [27].

### 2.5.1 Enterprise Information Systems (EIS) Metamodel

With regard to these limitations, several approaches have appeared to overcome the issues mentioned above. For example, the EIS approach [27, 28] for developing information systems that integrate business domain, processes and user interface descriptions is proposed to enable automatic code generation. Three separate metamodels for each aspect are introduced, namely the, Business Domain, Human-Interaction, and Business Process Metamodel.

The Business Domain Metamodel (BDM) uses the specialisation strategy instead of using logic at another modelling level to solve data modelling problems when a given entity type participates in an association type with various roles. On the other hand, the Human-Computer Interaction (HCI) metamodel expresses the appearance and behaviour of UI concepts at a higher level of abstraction. Each behaviour triggered by the UI is treated by either a business rule or application functionality generated to connect to one or more UI rules. This information is specified in the metamodel as application rules for providing the integration of UI with the information system [27, 28].

The framework contributes to integrating concepts of HCI patterns with business data aspects, for forming an Information System metamodel. Thus, the integration between these aspects is promoted via metamodels. It also supports the usage of OCL [85] for generating stored procedures in the backend database system that implements the modelled business rules. It does not offer any end users customisation. [27].

### 2.5.2 Metamodel to support End user Development of Web-based Business Information Systems

Another metamodelling approach has been introduced for enabling end-users to contribute in web-based information systems development [29]. The semantics of the proposed metamodel helps end-users, using their logical thinking and domain knowledge, to start the development process with little knowledge in the technical domain.

The proposed concepts in the hierarchical metamodel are similar to those in the metamodel of the UML-based Web Engineering (UWE) approach. The main difference is the three levels of abstraction in [29] approach. They indeed grouped related common aspects at a separate level of abstraction and produced a three-level hierarchal metamodel for describing information systems, namely, Shell, Application, and Function level. Common aspects in all web applications such as user, object, access control and navigation are four models that are expressed at the shell level. Access control and other specific aspects (inherited from shell level), such as workflow (business rules) are modelled at the application level. The detailed implementation of functions is modelled at the function level [29].

End users might contribute in the development at a particular stage or level; for example, to configure a specific web application they only need the knowledge presented in the shell metamodel. Moreover, in order to specify the application level, users need to use the domain knowledge presented in the application metamodel.

### 2.5.3 Using Weaving Models to Automate Model-Driven Web Engineering

In [120], a model-driven approach for the development of Service-Oriented Web applications (SOD-M) the use of model weaving is proposed. It is noted as a part of the MIDAS framework for developing Web Information Systems (WIS) to use the MDE approach. Unlike the promise of MDD to enable a full automation of the whole development process, SOD-M aims to include designer-decisions during the development stages and consider it before executing each model transformation (customisation). This is done in order to reduce the complexity of model transformation resulting from the nature of the behavioural model specified at an early stage.

Modelling PIMs consists of constructing and mapping two models, namely, Extended Use Case (EUC), and Service Process models. The first model is used to capture the service functionalities of the system at a lower granularity, whereas the second model is a type of *Activity Diagram* used to represent processes. The mapping between these PIMs is defined from the EUC to the Service Process one. The Weaving Model is used as an annotation model to introduce design decisions during executing transformations. It is a container for the extra data absent from the behavioural metamodels, but required for transformation execution. For instance, it helps in mapping the order of executing *include* relationships in the case of having a use case with more than one *include-case* [120].

The annotated model is also used to establish and handle the links between model elements using ATLAS Model Weaver (AMW). Both it and a source model are then used as an input to the transformation to generate the target model. The annotation in the source model (e.g. EUC) is used to fill the missing data that is required to execute transformation. It also allows the customisation of model transformation [120].

The development process starts normally by transforming a high-level business model into a service composition model using several layers of transformation. As a common task in MIDAS, the rules of transformation are expressed with natural language at the initial stage, then it is formalised using graph transformation rules. ATL [33] is used to implement the formalised rules to be used in expressing the mapping between PIM and PSM within the framework [120].

### 2.5.4 UWE and MDUWE Metamodelling Approaches

Koch [117] proposes a model-driven engineering framework in UML-based Web Engineering (UWE), which is supported by a CASE tool integrated with Eclipse. In UWE [117], aspects of web applications are captured using different models (UML profiles[70]). These models are then integrated and transformed to produce a business logic code, web pages and configuration files. In MDA [89] terms, these models are CIMs (UWE profile use case models), PIMs, and PSMs (other UML profiles). The MagicUWE tool and any tool that supports UML modelling might be used to design web applications using UWE in order to provide a semi-automatic generation of web software. A UME approach employs various transformation languages such as ATL [33], QVT-P, and Java for implementing model transformations [117].

Kraus, Knapp, and Koch in [68], also proposed a complementary approach, called the Model-Driven UWE (MDUWE) approach, which is a based on transformations and metamodels. On the one hand, the transformation rules are implemented using the ATL transformation language [33]. A number of transformations are designed to obtain different aspects of the web systems, such as *Requriement2Content* for deriving the content model from the use cases.

On the other hand, the metamodel of UWE is structured into requirements, content, navigation, process, presentation and more packages or sub- metamodels using UML 2.0 profiles [70]. For example, the presentation metamodel defines elements that are used to specify the layout and user interface UI elements on the web page reflecting underplaying processes and navigation [68]. From that, it can be said that end-users still need to learn the UWE modelling language and some technical (designing) skill to model the different views of the web system.

### 2.5.5 Model-Driven Web Engineering (WebML)

WebML [124] is a domain-specific language for expressing the concepts and mechanisms of the domain of web engineering. It allows the specification of the conceptual model of web applications, such as data, service, navigation, and processes. It has its own formalism and abstraction levels of models. WebML [124] is supported by the WebRatio [4] tool for enabling automatic code generation for the J2EE platform [16].

The WebML [124] metamodel consists of several views, namely, the data model, the hypertext model, and the presentation model. The data model is used for presenting the data schema including entities, attributes and relationships. It is based on the standard ER model (Entity-Relationship). The hypertext model is regarded as a graph of linked pages that represent different information (view of the site) and also the navigation path between these pages. It establishes the overall structure of the domain as a collection of units, such as views, areas, pages, and content units. These units are connected together forming the WebML [124] hypertext model. The presentation is concerned with the visual representations and corresponding styles of the pages on screen [83].

Authors describe in [16] a transformation approach, integrated with Eclipse, for transforming WebML [124] to MDA [89] representations (MOF metamodelling layers [87]). The approach consists of three transformation stages, namely, Metamodel Generation, model Generation, and Higher Order Transformation. In Metamodel Generation stage, the mapping between the DSL metamodel and MOF M2 [87] is considered by mapping DTD and EBNF representations of the DSL metamodel into Ecore. The generation starts by parsing the concrete syntax of DTD and EBNF in order to derive the instance of the defined metamodel (DSL Injection). The next step is applying a model transformation that is implemented using ATL [33] between the DSL Metamodel and the Ecore one [16].

In the following stage, Model Generation, a transformation from an XML [122] representation of a WebML [124] project to an instance of the WebML [124] metamodel (generated at previous stage) is performed. Similar to the Metamodel Generation stage, this phase consists of an injection and transformation step. A Java injector is used in the DMS Injection step to convert the XML [122] document to an instant XML [122] metamodel. In the next step, an ATL [33] transformation is used to map an XML [122] and EBNF model to an Ecore model (MOF M1 [87]), which is an instance of the generated DSM metamodel [16].

### 2.5.6 Model driven, Existence dependency Relationship, Object oriented Development (MERODE)

MERODE is a model-driven engineering approach for developing enterprise systems. It adopts the MDA strategy, in which it aims at the creation of a complete platform-independent domain model (PIM) that is translated into a platform-specific one (PSM) and executable code [75, 21]. The method tackles the problem of UML semantics ambiguity by using a limited number of formally defined and semantically clear UML models (*Class* and *State Machine*), supplying them with an *existence dependency graph* and *Object-Event Table* [21]. Moreover, it uses the *Process Diagrams* from the BPMN as a behavioural business domain model to capture topmost concepts of the system functional requirements at the early stage of development (requirement/analysis) [21, 103].

MERODE has several modelling activities during the development process of IS, namely, *data modelling, interaction modelling* and *life cycle modelling* [103]. The *data modelling* step in MERODE is a process of creating structural aspects of business objects. A refined UML *Class Diagram* is used to construct the *existence dependency graph* using *MERODE specific* notation [21, 103]. This diagram has a key benefit to MERODE, in which it performs better consistency checking between the three MERODE diagrams rather than other modelling approaches [103].

Furthermore, the interaction modelling step in MERODE is achieved via the creation of the *Object-Event Table (OET).* It is a tabular representation that consists of all business objects with related business event types. The type of interaction can be either *create (C), modify (M)* or *end (E).* In addition, *life cycle modelling* is drawn according to each object in the domain to prevent events occurring in a wrong or random order during the life cycle of that object [103].

It is worth saying that the MERODE method is supported by an object-oriented tool (JMermaid) that consists of graphical editors for constructing and modifying specific system artifacts consistently. The tool aims to help software engineers to construct enterprise systems using a number of formal models [74]. The*JMermaid* adopts a template-based code generator mechanism that is able to successfully generate a multi-layer system that consists of a *GUI, event handling (session beans)* and *persistence layer* [104].

### 2.5.7 Reactive System Development Support (UML-RSDS)

The Reactive System Development Support (UML-RSDS) is a subset of UML with a precise semantics [71]. It aims at generating executable code from high-level specifications, in which it is supported by an MDE tool [71] to provide a rapid automatic creation of software systems from high-level standard UML artefacts, namely, *Class, Use case Diagram* and OCL [85] language.

The targeted multi-tier EIS applications, which are developed from UML-RSDS MDE tool, consist of five tiers, namely, *Client, Presentation, Business, Integration* and *Resource tier*. The *Client tier* consists of a number of HTML files as interfaces to the end-users. Moreover, the *Presentation tier* contains Java servlets classes and JSP files. In addition, the *Business tier* has some *session, entity beans* and *value objects* to represent the business services. Furthermore, the *Integration* and *Resource tier* consist of database interface, as well as web services, and the back-end data stores of the system respectively [71].

In UML-RSDS, two forms of *use case* can be constructed and used, *EIS and General*, using the standard mechanisms of structuring and composing use cases (via *extend, include, inheritance* and more). The EIS use case represents a simple straightforward operation (e.g. CRUD), whereas, the *General* ones are used to specify model transformations [71].

The UML-RSDS approach provides a formal specification approach for model transformation [72]. The transformation is defined in a hybrid style using *OCL constraints* that express the relationships between system models. In the approach, OCL is used in various places, such as, *class invariants, class diagram constraints, operation preconditions and postconditions, use case pre-, postconditions and invariants, state invariant and transition guards* [72, 71].

In order to generate more efficient code, the *constraints* are defined explicitly as *operations*, specified by pre- and postconditions, determining how a source element is translated to the target one in the target model [72]. The precondition of each rule identifies when this rule is applicable to which source element in the source model, whereas, the postcondition identifies what utilisations to elements and links should be constructed between the source and the target element [71].

### 2.5.8 Summary

This part of the literature review discussed some general modelling approaches in software engineering that are supported by MDE CASE tools. From the modelling perspective, we considered single-view modelling approaches and multi-view modelling approaches. It is noticed that many pure OO approaches employ, or adopt the UML [70] *Class Diagram* as a central artefact for capturing the structural view of systems, while, the UML [70] *State Machine Diagram* is used for capturing the behaviour side of the systems. On the other hand, in the integrated approaches, a modified version of *Data Flow Diagram* are used to capture system's behaviour and then mapped to OO models, such as the UML *Class Diagram* and other models.

Besides this, examples of DSL-based approaches are also considered in this part of the literature. We tried to identify what are essential and minimal views to be used at early phases of development. Furthermore, we tried to justify how complex the artefact(s) are, in both types of approaches, with respect to their suitability to business end-users. The following table (2.2) summarises drawbacks of the above approaches that are covered and improved in the proposed method in the thesis.

| Approach | Artefacts | Comments |
|---|---|---|
| **WebML** | (DSL) Data, Hypertext & Presentation Model | Complete design details of web applications are expressed explicitly, manually by developers, in the related models, such as navigation links, GUI control types & specifications, pages structure and back-end data that appears in each page. |
| **MERODE** | UML Class & State and BPMN Process Diagram | Business objects are defined in the Class Diagram, including their internal structure, relationships between objects (with correct multiplicities) and *association* classes. This diagram is equivalent to the Class Diagram at the *Design level* in the traditional sotware development methodologies. Additionally a default finite state machine is specified for each obejct using a State Diagram. These technical specifications might be error-prone. |
| **MDUWE** | Use case, Class, & Activity | Different UML Profiles are used to define CIM, PIM, PSM, and UI. These complicated diagrams require technical details, expressed using OCL. This does not suit business users knowledge. Furthermore, the lack of Use case formal semantics might lead to have different interpretations for a same construct. The way of expressing *include* and *extend* dependency may raise a confusion when using them. |
| **UML-RSDS** | UML Class, Use case, State Machine and OCL | A developer writes system specifications in UML and OCL. It can be said that specifying constraints in OCL is not a simple task from a business user perspective. Additionally, defining transformations using Use cases may raise similar arguments mentioned in **MDUWE** regarding the semantics-free zones and expressing dependencies. |
| **SOD-M** | Extended Use case, Service Process Model (UML Activity) | Even after extending the Use case Diagram, it still expresses the dependencies between use cases in the traditional way. Arguments in **MDUWE** & **UML-RSDS** for Use cases can be used. Additionally, a detailed Activity model (complicated) is constructed for realising and capturing the control flow of each Use case. |
| **EIS** | UML Profiles & Business Process Diagram (BPMN) | A number of UML Profiles are used to express different IS layers. OCL is used to specify constraint on these profiles. Argument for expressing constraints using OCL mentioned in **UML-RSDS** can be used in this approach. |

| | | |
|---|---|---|
| **MoSys** | Extended Data Flow Diagram, UML Use case & Activity | Modelling a system behaviour requires a number of Activity Diagrams for specifying the behaviour for each system function modelled using Use case Diagram. This raises the issue related to ambiguty of Use case semantics and semantics-free zones, arguments mentioned in **MDUWE**, as well as designing complicated Activity models. |
| **ED-UWE** | Use case, Class, & Activity | All important design details must be expressed by end-users, such as UI appearance, navigation links, process work flows. Arguments raised in **MDUWE** for using UML Diagrams can be used. |
| **FOOM** | OO-Data Flow, UML Class, Transaction Diagram, message chart & pseudo-code | Three types of Class Diagrams are desgined to define system data (initial and complete) and UI. The approach requires further detail about methods in the complete Class Diagram to represent system behaviour. This is captured using message chart or pseudo-code. |
| **ZOOM** | UML Class, Activity, Statechart & Finite State Machine | Detailed UML artefacts are required to represent the system. Modelling superstates and their interanl states and transitions in Statechart Diagram [49] requires design knowledge (low-level), which might be error-prone from the end-user perspective. |
| **MOD4J** | DSLs: Business, Data Contract, Service & Presentation | It is designed for developing Data-Centric Applications that support the basic CRUD operations. No control flow of business process (logic) is supported. Additionally, a developer constructs all models manually. |
| **Rhapsody** | UML Class, Statechart, Use case & Activity | It generates a partical behaviour of a system. The Use case semantics and notation arguments **UML-RSDS** & **MDUWE** are issues arise in this approach. |
| **OPM** | Object-Process Diagram (Workflow-like) | No separation of concerns, the structure and behaviours of a system are expressed in one complex model. This increases the number of elements, notation types and constructs in the model. A complete OPD can be expressed as five UML Diagrams: Sequence, Class, State, Activity, Use case and Deployment diagram. Additionally, developers has to consider the state of each object before, after and during a process occurence. |

Table 2.2: Drawbacks of other MDE approaches

## 2.6 Overview of End-User Development (EUD) With Respect To MDE

The End-User Development (EUD) [106] is a development technique that aims to empower end-users, who have limited technical knowledge, to become involved in the process of designing and/or customising their systems to increase their productivity and satisfaction [107]. It also aims to translate accurately and comprehensively the informal description of domain problems to reduce the gap between what the exact user desires and what functionalities the implemented system has [30].

### 2.6.1 Component Based EBusiness Application Development and Deployment Shell (CBEADS)

Producing end-user tools for constructing web applications, such as *DEMIN* and *Mashups*, *CBEADS* [45] is a widely-known example of applying EUD [106] in the real-world to tackle the problem of the lack of web developing skills for non-programmers [97]. This kind of tool considers the business-users perspectives or mental model [97], encodes developers knowledge as rules [92] and enables users to easily tailor software to meet their individual needs [107].

Ginige and De Silva, in [45], have introduced a meta-modelling approach for enabling end-users to be involved in the continuous development process, side by side with developers, end users with little technical knowledge. It also aims to enable them to employ effectively the meta-models in such as customisable environment [45]. The concepts in the meta-model are similar to those in the UWE approach (UWE, 2011). However, all common aspects are grouped at a separate level of abstraction for describing information systems, namely, Shell, Application and Function level [30]. These models are embedded into a component-based Shell introduced as developers templates for end-users to instantiate a meta-model instance and populate it at one or more levels. The *CBEADS* and other related (SMART) tools are used to generate business objects as well as functions, users interfaces and SQL queries [45].

### 2.6.2 Model-Driven Development for End-User development (MDD4EU)

This approach provides a mechanism to allow end-users to collaborate with developers at the early development stage, rather than transform end-users description of their needs into developers. It is supported by a modelling language (DSL) called Pervasive Modeling Language (PervML). PervML is used on the construction of pervasive services in the context of smart home systems [91].

MDD4EU is mastered by developers, in which they determine what knowledge (*properties*) end-users are allowed to supply to models during the development process. After that, developers also determine an appropriate language that is familiar to end-users for enabling them to supply and edit models using the language. At the end of the story, a model that holds user preferences is created with a degree of quality [91].

In order to achieve this, the MDD4EU approach is introduced to end-users via a toolkit that is specially designed in such a way that it suits end-users. The toolkit is designed to be domain-independent, in which it has an interface that might be reusable in other domains when adopting the MDD4EU [91].

### 2.6.3 Summary

This part of the literature discussed some model-driven engineering approaches that allow end-user participation in the development process. Both approaches provide mechanisms for allowing end-user to participate and work with developers. In contrast, the thesis aims at going beyond this and allow business end-user to lead and master the development process of information systems using less technical knowledge. The movement from this business knowledge to more technical one is performed and managed by transformation rules.

## 2.7 Outlook on the Chapter

This chapter discussed the related literature, including a brief comparison between alternative methodologies such as, *AOSD, MDA, xUML* and *Software Factory*. From this, it is possible to realised where the proposed approach fits, in contrast to other related model-based approaches, which is a multi-view MDE framework for developing enterprise information systems. The approach employs a user-friendly and semantically clear UML-like modelling language to describe system specifications at a higher level of abstraction than existing UML approaches.

Furthermore, the second part of the literature investigated model transformations, including the languages, paradigms, composition techniques for model transformations and code generation strategies. This formulated the adopted model transformation mechanism and language in the proposed MDE method in the thesis, which is a hybrid model transformation style, encoded using *Java* programming language.

Additionally, a number of model-driven engineering approaches are covered in the rest of this chapter. The discussion focused, mainly, on the complexity of metamodels, modelling languages and notation, used for defining and expressing domain concepts, and the ability to generate complete executable code from abstracted models. Core concepts of the proposed modelling language are defined using a metamodel rather than an UML profile. Moreover, end-users participation, in some approaches, is highlighted and compared in order to introduce a development process led by business end-users.

# 3
# Framework Overview and Analysis

"Make everything as simple as possible, but not simpler"

Albert Einstein

## 3.1 Context

The chapter presents a general overview of the proposed solution showing how it supports Business-user Model-Driven Engineering. It introduces, in brief, the notion of a simplified user-friendly modelling language, which we call the Micro Modelling Language ($\mu$ML); and, it discusses briefly the purposes and engineering activities in each development phase of the proposed Business-User Information-Led Development (BUILD) framework.

In each stage, the chapter motivates the idea of adopting one or more $\mu$ML model(s), and shows how this suits the conceptual (natural) thinking of business users. It also demonstrates how the $\mu$ML models collaborate for capturing critical aspects of the system, which lead to significant design decisions. Each model is exemplified by a small portion of a real-world business process.

The adopted structure of model transformations throughout the development lifecycle is also reviewed, including modularisation, reusability and composition. The chapter also formalises the basic foundation of the language metamodel that will be used frequently in the following chapters.

## 3.2 BUILD: Business-User Information-Led Development

The proposed Business-User Information-Led Development (BUILD) framework is a model driven engineering approach to enterprise information systems development. It aims to derive detailed implementations of information system layers from high-level business-user specifications using a lightweight modelling language called the Micro Modelling Language ($\mu$ML). The inspiration for this methodology is the Reusable Modelling Design Language (ReMoDeL) approach[101], which is based on a multi-layered, forward-transformation strategy that turns multiple abstract representations of the requirements into concrete design models and generated code.

As discussed in earlier chapters, end-users understand their business well in a whole, but it is difficult for them to articulate, systematically, the software system that they require using their limited technical skills. BUILD encourages end-users to employ their business knowledge to construct a number of user-friendly $\mu$ML models, and let the transformation rules evolve an executable prototype system. This strategy avoids the mistake of asking users to construct complicated models that are full of technical design detail and formal constraints.

In contrast, end-users express business processes, entities and the impact of each process on one or more business entities, using three simple requirements models with semantically clear notation (*Task, Impact* and *Information* Model). The following figure (Figure 3.1) presents a wider picture of BUILD using a traditional flowchart. The development process within the framework is divided into four main phases (illustrated by horizontal dashed lines), namely, *Requirements Sketching, Analysis, Design* and *Code Generation* (Section 3.2.2). The development progress, from one stage to another, is handled and proceeds by model transformation rules (steps) that are responsible for the evolution of the model artefacts representing the system, from initial requirements through to generated executable code.

In Figure 3.1, the $\mu$ML models used in each phase of the BUILD method are depicted as Document nodes in the flowchart notation. These models are sorted as ASTs in memory and XML files to express various system views to be used during the model transformation steps. Additionally, all forward translations from a single aspect (model) to a new one, merging two aspects into a new one and performing in-place modification of a particular aspect, are depicted in Figure 3.1 using the *Arrow* flowchart notation. External human participation, which is needed to construct models or supply additional data, is also depicted in the figure using *manual input* notation.

### 3.2.1 What is Micro Modelling Language ($\mu$ML)?

The Micro Modelling Language ($\mu$ML) is a user-friendly modelling language that is used to express basic structural and behavioural aspects of enterprise information systems. It has graphical UML-like representations that correspond to underlying XML parse trees. It aims to simplify the modelling activity using simpler notation with cleaner semantics than existing UML-based, or Model Driven Architecture (*MDA*) approaches. It accelerates the development process by using effectivily business-users' knowledge of their desired system.

It can be argued that the simplicity and clarity are achieved in the proposed $\mu$ML when each notation has the same meaning in each model, but in appropriate way that suits the context of each model. For instance, $\mu$ML unifies the visual notation of the generalisation and compostion relationship between elements in each model it appears in. This issue is discussed in more detail in chapters 4, 5 and 6, where the notation and its semantics is described.

In BUILD, $\mu$ML models are used to represent various information system views. There are three major groups of models, with each group being associated with a specific development stage in order to capture the essential characteristics of the system. The models in the top group are employed by model transformation to produce further detail and new aspects (models) of the following group to be used in the next stage of development. $\mu$ML concepts, notations and their formal foundation are discussed in detail in chapters 4, 5 and 6, according to the BUILD

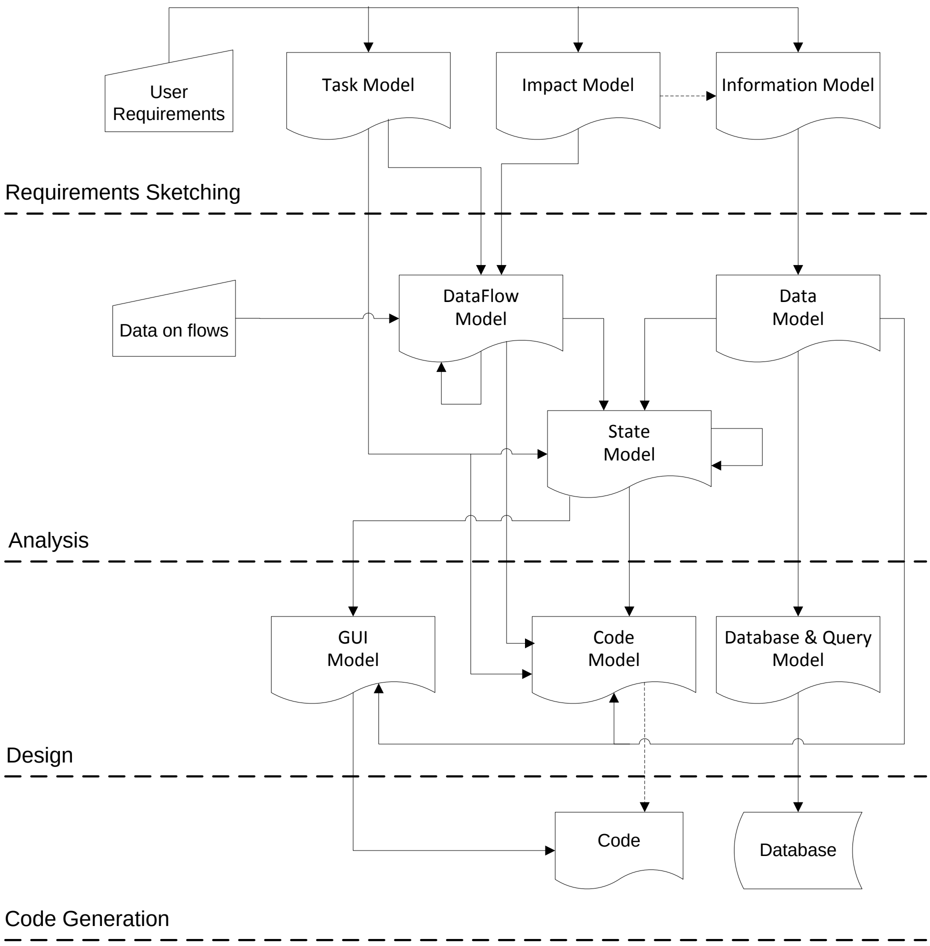



Figure 3.1: BUILD. A wider picture

stage that the model belongs to.

#### 3.2.1.1 The Conceptual $\mu$ML Metamodel Hierarchy

Conceptually, the $\mu$ML main concepts are organised in a hierarchical structure of model nodes (elements), which form a core metamodel for the language. The following Figure (3.2) illustrates the main generalisation and composition relationships between $\mu$ML elements. It consists of higher level concepts of the metamodel that express the core elements of the language. These concepts are: *Element, Node, Edge* and *Model.*

The $\mu$ML metamodel includes a number of intermediate nodes that group some terminal nodes based on their common features. For instance, the kinds of *Arcs* subdivide into the *Flow, Transition* and *Participation* arcs. Additionally, the kinds of *Relationships* subdivide into: *Association, Generalisation, Composition* and *Dependency* relationships. The metaclass *Model* consists of a collection of the kinds *Element.*

In $\mu$ML model(s), all *Arcs* and *Relationships* are defined as instances of one of their descendant (metaclasses). Each of the leaf (terminal) nodes, appears in the metamodel (Figure 3.2), corresponds to a concept in one or some $\mu$ML model(s), which has a specific meaning and

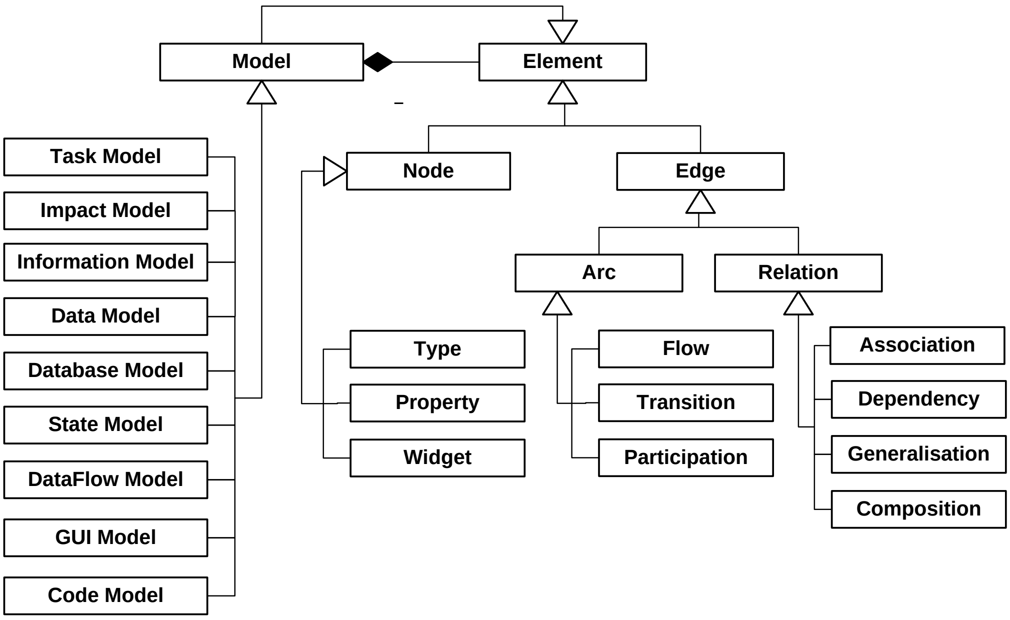



Figure 3.2: The Metamodel for $\mu$ML

a graphical notation. Furthermore, there is an element in the underlying representation (AST) of the models to represent each terminal concept.

### 3.2.2 Development Phases in BUILD

The overall structure of BUILD can be divided into four main phases: *Requirements Sketching, Analysis, Design* and *Code Generation*. Each phase in BUILD focuses on a variety of abstract system views to present the system on several levels of detail and encapsulates various model transformation activities that occur during the process of software development.

#### 3.2.2.1 Requirement Sketching Phase

In this context, requirements engineering can be introduced as common practice in the software development lifecycle. It refers to the elicitation of stakeholders' desires and all other activities involved in discovering the requirements for a SE process. This stage involves the initiation of communication and collaboration between end-users and the developer team in order to map system requirements and objectives onto a collection of formally designed software characteristics and functions.

Engaging different stakeholders in the initial phase of the development process is considered critical in all traditional software development approaches, such as *Waterfall Model, V-Model, Incremental Model* and *Spiral Model* [94]. The functional requirements have to be explained clearly and transferred precisely from the business users to the developer team. This happens via holding a number of formal meetings to discuss requirements and the ability to accommodate them within the system. In the end, all details are stated textually in the contract.

By contrast, the current trend in new approaches to software development, e.g. Model Driven Software Development (MDSD) and Domain Specific Language (DSL), is to assign the task of expressing the functional requirements and system structure to skilful designers. These designers must have enough knowledge to deal with complex modelling language and models rich in detail. They are responsible for modelling the system to meet the functional requirements provided by the users. From there, the initial step of passing the end-users requirements to the designers must also be taken.

In BUILD, the business-user engagement is different from that found in traditional approaches. During the Requirement Sketching Phase, the business analyst, who is aware of the exact requirements of the organisation, establishes formal agreements by describing the functional and structural requirements of an information system, in models, using clean and simple modelling language. The adopted $\mu$ML notations are lightweight and tailored to capture, visually, the intellectual logical thinking of end-users about their business. The artifacts produced are evolved directly through a chain of model transformations.

Practically, business-users describe the real world business data in terms of entities and relationships, making the entities organised, structured and easy to understand. Moreover, they describe a skeleton structure of the system that contains business tasks, goals and real world interactions with stakeholders. Additionally, users also express how the task affects the business entities in terms of the impact on the system object during the execution.

A number of system aspects must be prepared in order to progress to the second stage (*Analysis*). The proposed $\mu$ML models, Task, Impact and Information Model, can adequately cover the end-users specifications that act as a starting point for the first collection of transformation steps to satisfy the requirements of the analysis phase.

### Task Model

The *Task Model* is a ***manually-constructed*** structural model that describes the structure of all significant enterprise activities. It typically captures the wider context of the business, including interactions with external stakeholders. By using the conceptual (natural) thinking of business-users to express the functional requirements, in terms of the business process and stakeholder interactions (Figure 3.3), the *Task Model* can be constructed naturally in a straightforward way, stating the business tasks and the stakeholders involved in them. The notation and concepts of the *Task Model* are presented and discussed in detail in Chapter 4.

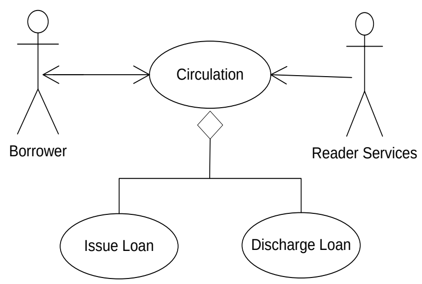



Figure 3.3: Task Model. Library Circulation System example

Figure 3.3 above illustrates a *Task Model* example of a small portion of *Library Circulation System*. It indicates that the *Circulation* consists of two business tasks: *Issue Loan* and *Discharge Loan*, modelled using a *part-of* relationship. Furthermore, a *Borrower* and *Service Reader* participate by supplying the system with some information and/or receiving back some responses from it.

### Impact Model

The *Impact Model* is a kind of behavioural model, ***manually-constructed*** by business users, that describes the impact of business tasks on the internal objects portion of the system data. Exploiting the end-users awareness and understanding of the business to sketch the interconnection between tasks and objects enables the capturing of the internal behaviour of these tasks. This is extracted by model transformations to estimate the control flow between tasks (the order of execution). The following figure demonstrates an example of the *Impact Model* of a *Library Circulation System*. The notation is discussed and formalised in Chapter 4.

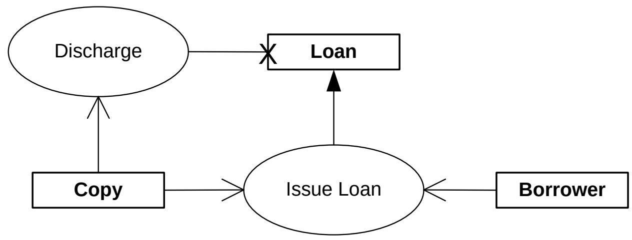



Figure 3.4: Impact Model. Library Circulation System example

Figure 3.4 above demonstrates an *Impact Model* example of a small portion of *Library Circulation System*. It shows the internal behaviour of the *Circulation* task components: *Issue Loan* and *Discharge Loan*, collaborating with the logical business objects: *Copy*, *Loan* and *Borrower*. The figure indicates that the *Issue Loan* task reads *Copy* and *Borrower* objects to create a *Loan*, whereas the *Discharge Loan* reads only a *Copy* to destroy the related *Loan*.

### Information Model

The *Information Model* is a ***manually-constructed*** structural (abstracted) representation that describes real-world business aspects and the relationships that might exist between them. This model is obtained through two different methods: a partial model is derived (automatically) from a pre-defined *Impact Model* or a complete one is provided (manually) by end-users. The first method is considered a simpler approach for Information Model construction because the framework takes responsibility for manufacturing the required relationships between entities, and leaves the task of specifying their end-role multiplicities to the end-users.

The structural relationships between objects in the *Information Model* can automatically be extracted from the *Impact Model* in the first approach. A number of mapping rules are applied to achieve this shift between these system views. This leads to the construction of *Information Model Aggregation* and *Association* relationships between the captured business entities. Then the properties of the entities (attributes) and the end-role multiplicities are added as part of a separate design activity.

On the other hand, in the second method, the end-users are in charge of maintaining consistency during the construction of each model to avoid any clash of inter-relationships between objects. They must model manually all business entities and their relationships. The objects that appear in the Information Model might be physical objects and documents, logical information about stakeholders or other physical items used in the business. The inter-relationships between these objects are presented in an un-normalised form. It is worthwhile emphasising that the current implementation of BUILD supports this construction approach.

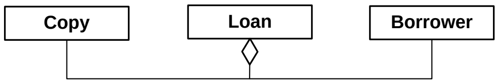



Figure 3.5: Information Model. Library System example

The above figure (Figure 3.5)shows an example of the *Information Model* of a *Library Circulation System*. It illustrates a conceptual whole/parts relationship (*Composition*) between three objects: *Loan*, *Copy* and *Borrower*. The *Loan* object represents the whole side of the relationship, whereas *Copy* and *Borrower* are its parts. The notation is discussed and formalised in Chapter 4.

#### 3.2.2.2 Analysis Phase

Several intermediate models that contain extended detail about the required system are constructed in the *Analysis Phase*. This stage is inspired by the early *Structured Systems Analysis and Design Methodology* [43]. Our method attempts to specify the requirements in terms of initial business tasks that need to be decomposed into a number of atomic ones, based on a functional decomposition technique and the use of a data flow diagram.

As the outcome of this stage, a number of artifacts are produced in an attempt to reveal more detailed aspects of the system. The internal behaviour of each business task and the state of the system are clearly identified to the designers in terms of data flow (detailed *DataFlow Model*) and state-transition (*State Model*). In addition to this, the business entities captured previously during the *Requirement Sketching Phase* are refined and re-described in the notion of the *Data (Dependency) Model*. All artefacts at this stage are ***automatically-generated*** by rules, except the initial DFD model that needs end-users engagement via annotating data on flows.

It is worthwhile noting that during the shift from the *Requirement Phase*, the number of complex transformations varies for each model, for instance, the creation of the detailed *DataFlow* requires two forward translating steps; the initial *DataFlow* (DF) is created first and then the detailed DF is derived from it. On the other hand, the compilation of the *State Model* construction requires one forward translating step from the detailed DF model and one in-place modification for generating states of failure business scenarios. In contrast, the *Data Dependency Model* is constructed directly using a single translating step from the *Information* and/or *Impact Model(s)*.

Indeed, the *Model Transformation Framework* takes *Task* and *Impact Models* as source models and then applies translation rules to fold their concepts and generate the initial *DataFlow Model*. For each task that appears in both source models, new components in the target model arise after combining the external interactions provided/received by actors and the internal ones with objects that lie within a logical boundary representing this task. The resulting boundaries introduce a number of independent event-driven business process transactions of the system.

To exemplify this, new *Input/Output* subtasks are generated after extracting the system interactions with the actors. Furthermore, CRUD subtasks are produced after translating the internal interactions with the system objects (back-end entities). Both kinds of subtasks are linked and grouped within a logical boundary of the original business task. It is important to mention that the boundaries in the initial *DataFlow* have no knowledge about the data on flows and there are no interconnections between subtasks. These missing parts are considered in the next transformation step.

The existence of data on flows is fundamental to the approach in order to estimate the possible order of task execution. Thus, the manual engagement of business end-users is required to specify the kind of data flow on the initial *DataFlow* artifact. As a consequence of visualising the *DataFlow Model* using $\mu$ML notations, users can add data to related flows on the model in a straightforward way. These extra details are used simultaneously with the type of CRUD operation assigned to tasks during the task decomposition step in order to construct the detailed *DataFlow Model*.

Accordingly, boundaries in the detailed model express the data flow between linked subtasks. This version of *DataFlow* is used as a source in a second translating step in order to generate a *State Model*, after prioritising the atomic tasks in the previous translating step, whereas the *Data (Dependency) Model* is derived directly from the predefined Information Model. It is worth emphasising that the *Analysis Phase* ends after decomposing all the business tasks that require a number of atomic actions into tasks with a single action that are presented with a logical order of execution.

### Data (Dependency) Model

The *Data Model* is an intermediate view, ***automatically-generated*** from a pre-defined *Information Model*, that describes the dependency of the logical data in the system, and supports the development to a point where a logical database schema may be generated. The model consists of logical objects linked by a number of dependency relationships, representing the direction of data dependency.


Figure 3.6: Generated Data Model. Library Circulation System example

Figure 3.6 illustrates a part of a Data (Dependency) Model of a *Library Circulation System*. It demonstrates dependency relationships between three objects: *Loan*, *Copy* and *Borrower*. The *Loan* object depends on both *Copy* and *Borrower*.

### DataFlow Model

The *DataFlow Model* is an ***automatically-generated*** intermediate model that describes how information data is transformed in tasks, flows via system components and is manipulated into data sources at a particular level of abstraction. The model consists of a number of entities and participants (agents) that are connected to some tasks via a variety of flows. It provides a detailed description of the business tasks and their aggregated subtasks that supply and perform operations on data in terms of data flows.

It is worth mentioning that the creation of a complete *DataFlow Model* requires two independent mapping steps. The first one performs automatic merging and gathers concepts from both *Task* and *Impact Models* to derive a number of business tasks that are connected by some (typed) flows with system stakeholders and internal objects. The second step starts by a manual engagement from end-users who annotate the generated data flow model by data on flows. Then, the automatic task decomposition step is applied to the annotated intermediate DFD to produce the complete DFD. The detailed description of these construction steps is presented later in chapter 7.

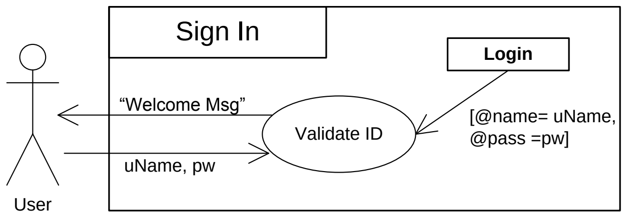



Figure 3.7: Generated DFD Model. Library Circulation System example

Figure 3.7 above shows a portion of a *DataFlow Model* of a *Library Circulation System*. It illustrates the contents of the *Sign In* business task. It can clearly be seen that the task consists of only one task, *Validate ID*, that is connected to three flows: *Input, Read* and *Output*. A stakeholder, *User*, interacts with the system by supplying login details to be checked via the task against the data stored in the login entity. The *Validate ID* then responds back to the user with a suitable message.

### Screen State Model

The *Screen State Model* is an ***automatically-generated*** intermediate representation, derived from a pre-generated detailed *DataFlow Model*, to demonstrate, in an abstract way, the behaviour of the system in terms of screens and their navigations. It is mainly based on the notion of a *State Machine*[116] in which each screen appears as a state and associated navigations appear as transitions.

*DataFlow Boundaries* and *Tasks* are mapped into boundaries that contain *States* (each represents a systems state after executing an action) and *Transitions*, which trigger their actions, respectively. Any flow connected to an object (either source or target) is translated into a *Ready* state, and any *Input/Output* flow connected to an actor is translated into a *Waiting* state. The *Waiting* state indicates that the system is waiting for some external activity related to stakeholders, such as input or output information. On the other hand, the *Ready* state indicates that the system is ready to execute a back-end operation (e.g. a database action), such as an update, insert or delete of a record.

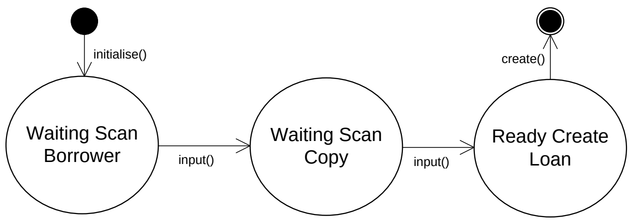



Figure 3.8: Generated State Model. Library System (Issue Loan) example

The above figure (Figure 3.8) demonstrates the content of the *Issue Loan* business process. It consists of two *Waiting* states for inputting copy and borrower detail and one *Ready* state for (creating) inserting a new *Loan record*.

It is worth mentioning that the creation of a complete *State Model* requires an in-place model modification step, which it considered to be a separate transformation step. A number of *Error* states are generated to handle failure cases of the business task, such as invalid/null inputs or database connection failure. The following figure (Figure. 3.9) illustrates a complete *State Model* of the *Issue Loan* task after applying the in-place modification step.

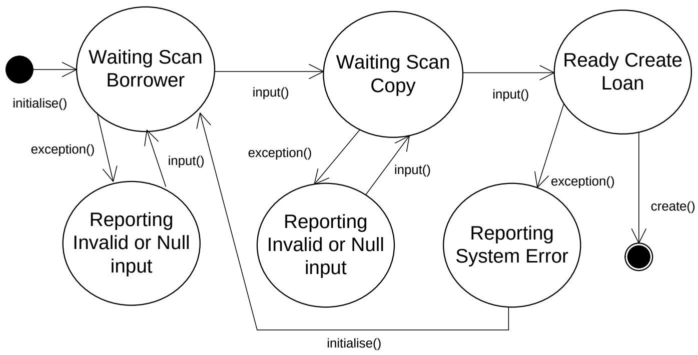



Figure 3.9: Generated State Model. Library System (Issue Loan) example

### 3.2.2.3 Design Phase

The *Design Phase* is considered to be a stage that is led by model transformations to construct a number of low-level design models. This phase aims to reach an adequate level of detail to enable a full code generation of an implemented solution. This involves the creation of the technical *Database and Query (DBQ), Graphical User Interface (GUI)* and *Code Model.* All constructed models are ***automatically-generated***, platform-independent and are ready for the *Automatic Code Generation Phase* to produce a platform-specific code.

The *Database and Query Model* is derived by translating business entities that appear in the pre-constructed *Data (Dependency) Model* and by consulting the type of *CRUD* effects for each entity in the detailed *DataFlow* of the previous stage. As an outcome, a complete definition of relational database tables with clear information about their keys is presented in the model. Moreover, a number of predefined stored procedures and triggers associated with the tables also appear, including their implementations. The resulting artifact is generic in that it can be used as an entry point for generating executable relational database schemas, such as *MySQL* and *Oracle.*

Additionally, the *Screen State Model* is used as a source in the translating step to construct a *GUI Model*. The produced artifact consists of multiple connected event-driven screens that perform the business, triggered by the system end-users. Furthermore, states of the possible failure scenarios are also translated into screens linked to the related screen in the successful scenarios or to the termination of the process.

The resulting *GUI Model* is considered to be platform-independent and complete enough to be employed in the next *Code Generation Phase*. For each generated screen, minimal component features, e.g. size, name and text, are inferred in the model as a consequence of crosschecking with object specifications and types in the *Data Dependency Model*. Moreover, tracing the type of *CRUD* operation and its carried data indicates the most appropriate type of generated *GUI* control. For example, in the case of Insert action, the possible type of generated *GUI* control on the screen is *TextField*.

Apart from this, there is a room for a separate folding transformation step that is applied to a number of analysis models: *Task, DataFlow,* and *State* Model, to construct the (System) *Code Model*. This step is independent and not part of the main IS development process (optional), as the current version of BUILD aims at generating 2-tier applications. This model is optionally used to represent the *business logic* layer in the 3-tier applications, because the business logic is by default translated into stored procedure logic in the database schema directly. The generated *Code Model* is also crosschecked with entities and attributes in the *Data Model* in order to maintain consistency in the object-relational mapping process when required.

### Database and Query (DBQ) Model

The *Database and Query Model* is an ***automatically-generated*** lowest level representation that describes the normalised design of the logical schema in a relational database, as opposed to a conceptual information model. A model is considered platform-independent that includes the generic data schema definitions, query expressions, and logical representation of tables and fields. The main objective is that a comprehensive level of abstraction for representing all generic specifications, required for any database generation, is achieved and expressed at this level of detail. This low-level model is used as a source model in the code generation approach. Figure (3.10) illustrates an example of the generated DBQ model, taken from the Library System example.

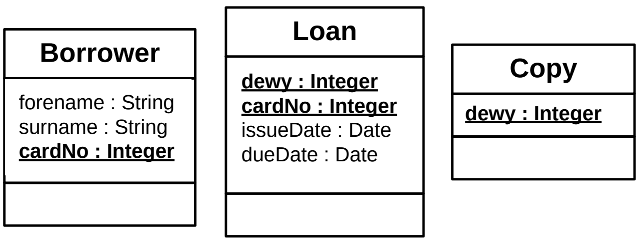



Figure 3.10: The Generated DBQ Model. Library System example

### Graphical User Interface (GUI) Model

The *Graphical User Interface Specification Model* is an ***automatically-generated*** lowest level textual representation AST (XML format) that describes the structure and the components of information system screens, in terms of windows with some controls and events. It is an abstracted platform-independent model in which there is no implementation detail presented in the model. This model does not have a graphical representation in the current version of the proposed $\mu$ML ver.1.0.

**Code Model**

The *Code Model* is an ***automatically-generated*** lowest level AST (XML) representation that describes the abstract specification for Object-Oriented (OO) code of the system in terms of business entities and tasks (classes), their methods, attributes, and expressions. It aims at representing a comprehensive business logic layer that determines how to manipulate the data in a separate layer, as part of an (optional) three-tier architecture. In the Current version of BUILD, the model can be constructed independently using a separate folding step, as it is not a mandatory part of the main development process of IS. Similar to the *GUI Model*, this model does not have a graphical representation in the current version of the proposed $\mu$ML ver.1.0.

### 3.2.3 Model Transformation Strategy in BUILD

The transformation strategy in BUILD consists of a number of independent translation steps, where each step contains a collection of *Translator* agents. It takes the given knowledge from one level and evolves it to a new more detailed level. As an outcome, several intermediate and lower level design artifacts are constructed based on end-user specifications. Generally, the strategy attempts to reach an adequate level of detail that enables full code generation, which is a completely independent phase.

The transformation approach relies on a *hybrid* (*imperative* - *declarative*) style. Each rule is designed separately, in which it does not know how other rules work or how they are implemented. There are dependency relationships between rules. Each rule fires selectively, if its input matches a conditional guard, and may employ further rules to execute part of the transformation. The design of the rule-tree typically follows the compositional structure of the model. While rules are always invoked in some given order on collections of artefacts, this order is not significant, in the sense that rule-dependency is the only constraint, and multiple firings of the same rule on the same inputs are idempotent. The rules of transformation are directly encoded in *Java* classes to define the map between input elements and output ones. Each set of classes represents a Java-translator that is responsible for orchestrating the translation rules in order to derive target models from source ones.

At each development stage, these transformations are capable of initiating an automatic shift from the current phase to the next one, introducing more detailed artifacts without the direct engagement of business users. However, on a specific layer of the *Analysis Phase*, the designer supplies manual data flow detail to be considered in the next step of the transformation. It is worth mentioning that the BUILD framework has two model-to-model transformation layers: *Requirement-to-Analysis* and *Analysis-to-Design*, whereas the *Code Generation Phase* is a completely independent layer and is totally automatic.

The *Translator* agents resemble the *Visitor* and *Composite design patterns*. All models can be regarded as a kind of content containing other similarly-structured content. This feature makes it possible to apply transformations as a kind of *Composite design pattern*, in which each rule delegates to other rules to continue the transformation, recursively. In addition, each translating agent has knowledge (rules) embedded within it to be applied to a source model.

Additionally, the translators behave like the *Visitor pattern* in the way they traverse a model. Each translator is responsible for a scope of the source model (parse tree). It focuses only on a particular part and applies translation rules to generate a part of the target model. The agent names are based on the source and target mappings. The following list addresses the translation steps within the *Requirement-to-Analysis* transformation step:

- *Information to Data Dependency Model:* This one-to-one translator derives the dependency relationships, between business entities, from binary association, generalisation, and aggregation relationships that appear in the sketched Information Model.
- *Task and Impact to DataFlow Model:* This two-to-one translator merges concepts from given Task and Impact Models and produces output elements in the (initial) DataFlow Model.
- *Initial DataFlow to Detailed DataFlow Model:* Based on the business-user supplying extra information about the nature of the data on each flow, this transformation takes the initial *DataFlow* with the additional information, provided by users, and construct the detailed *DataFlow Model* that contains the interconnections between atomic tasks.
- *Task, DataFlow and Data Dependency to State Model:* A merge of *DataFlow* and *Data Dependency Model* and a one-to-one translation from *Task Model* are considered together to produce to produce a *Screen State Model*.

As an outcome of the transformations described above, the artefacts of the Analysis phase are completely constructed and ready to act as source models for the next transformation step (Analysis-to-Design). This step of transformation adopts the identical structure to the previous one, in which it consists of a number of *Java-translators* for producing richer detail artefacts as follows:

- *State to GUI Model:* This translator is responsible for constructing GUI detail from the generated *State Model*, including actions and widgets controls based on the screen type.
- *Data Dependency to Database and Query Model:* The translator employs the dependency information, expressed in the source model to manufacture foreign keys, for the final form the translated business entities (normalised database tables). For each table, all fields are formally typed and specified in the way that it is interoperable by the relevant code generator to produce complete table definition syntax in SQL.
- *DataFlow to Database and Query Model:* The translator considers tasks attached to objects in the source model in order to create equivalent stored procedures.

As a result of the above transformations, all required design models are constructed and ready for the *Code Generation* phase. This stage is presented with further detail in the next section.

### 3.2.4 Code Generation in BUILD

The *Code Generation Phase*, the final stage of development, performs model-to-code generation in a particular running environment. It consists of a collection of generators that take the responsibility for generating executable code from detailed models. The overall architecture of the *Code Generation Phase* in BUILD also exemplifies the *Visitor* and *Composite design* *patterns*, in that it consists of a number of independent Java-generators. These generators are designed to support code generation for different specific platforms.

Unlike the overall structure of the *Java-translators* in the previous model-to-model transformation stage, we assign a *Java-generator* to produce the implementation for each information system tier: *Presentation, Business processes* and *Data*. These generators are:

- *Database Schema Generator:* The Database Schema Generator takes concepts from the design *DBQ Model* to generate executable dump files that contain *SQL* scripts. In this thesis, *MySQL Java-generator* is presented as a proof of concept, Chapter 7.
- *GUI Generator:* This generator creates Windows application screens with their widgets to visualise the designed system. A Java-generator for Java Swing Application with JDBC is presented as a proof of concept, Chapter 7.
- *OOP Code (System) Generator (Optional):* This generator is responsible for constructing a separate business logic layer that consists of a number of business entity classes, if required, Chapter 7.

## 3.3 Foundation of $\mu$ML

It is worth emphasising that, in formalising $\mu$ML, we are not looking for a degree of automated analysis and verification, all we are seeking is to add a formal flavour to the semantics of $\mu$ML concepts and notation. This section presents an approach to formalise $\mu$ML using standard First-Order Predicate Logic (FOPL) with extensions for subtyping ($<:$), equality ($=$), membership ($\in$), exclusive disjunction ($\oplus$) and uniqueness quantification ($!$).

According to previous work in this field [14, 58, 82, 99, 125], it can be noted that using FOPL is a widely adopted approach for formalising the constructs of UML diagrams and other DSLs. It mainly used for describing the language semantics and the interrelationships between its elements via mapping the concepts to a number of FOPL predicates and symbols.

The spirit behind the proposed strategy for formally specifying the semantics of $\mu$ML concepts is similar to approaches introduced in [99, 125] particularly. In [99], for instance, a formal approach is presented to define the descriptive semantics of modelling languages by mapping metamodels concepts to a number of FOPL sentences, representing the core concepts of the language via predicates and functions. In the same context, [125] proposes mappings from UML models and metamodels to FOPL statements for describing their semantics. As an outcome, models creation policies and mapping rules can be expressed formally using FOPL.

### In Comparison with OCL

OCL is based on FOPL and set theory, but it uses a syntax similar to programming languages and closely related to the syntax of UML. OCL is intended to be a light-weight formal specification language for annotation purposes e.g. invariants, guards, pre- and post-conditions for methods or UML diagrams, rather than to be used as the basis for extensive automated behaviour analysis [78].

### 3.3.1 Preliminary Principles

This thesis represents the basic concepts of $\mu$ML through predicate logic. The predicates are derived from the concepts appear in the $\mu$ML metamodel, such as, $Model, Node$, and *Arc*. Thus, the description of each concept in $\mu$ML models is provided with respect to its equivalent concept that appears in the metamodel. For instance, given $m$ is a $Node$, a unary predicate $Node(x)$ denotes *$x$ is an instance of Node.*

Similarly at the model level, concepts like *Task* and *Object* are both considered *Nodes* at the metamodel level. By applying the notion of subtyping, we can express this in our logic as: given $Task(x) \longrightarrow Node(x)$, $Object(x) \longrightarrow Node(x)$ denotes that *if $x$ is a Task then it is also a Node*, this can be expressed in our logic as $(Task <: Node)$, or *if $x$ is an Object then it is also a Node*, this can be written in logic as $(Object <: Node)$.

Two syntactic sugar declarations are used for these predicates. The unary predicate for assigning the type, e.g $Model(x)$, can be shorten and re-expressed as $x : Model$. In addition, the symbol (,) is also used to replace the logical *And* ($\wedge$) operator some occasions. For example, the statement $n1, n2 : Node$ is equivalent to $(n1 : Node) \wedge (n2 : Node)$.

On some occasions, we need to reason about the logical value of properties. For example, one of the main features of $\mu$ML language is that *each Element in the Model has a unique Identifer.* This means the distinction of *Elements* is based on the value of their *Identifiers.* From that, if we want to say that *Node (a) and Node (b), in such a Model (m), are distinct*, we need to prove logically that their *Identifiers* are not the same. As a consequence, the need for *equality* $(=, \neq)$ emerges to add this expressive feature to our logic. In order to add *equality* to our logic, three axioms must be declared:

- Reflexivity: $\forall\ x\ \bullet (x = x)$ (a variable x is equal to itself).
- Function substitution: $\forall\ x, y\ \bullet\ (x = y) \longrightarrow (f(..., x, ...) = f(..., y, ...))$ (if you substitute y for x in function $f$, the results are equal).
- Formula substitution: $\forall\ x, y\ \bullet\ (x = y) \longrightarrow (p(..., x, ...) \longrightarrow p(..., y, ...))$ (if you substitute y for x in the predicate $p$, the truth-value is the same.

From that, other properties of equality can be expressed as follows:

- Commutativity: $x = y \longrightarrow y = x$.
- Transitivity: $(x = y) \wedge (y = z) \longrightarrow x = z$.

In our context, based on the axioms mentioned above, given *id1* and *id2* are *Identifiers*, $id1 = id2$ denotes that $id1$ and $id2$ are equal. Similarly the statement $id1 \neq id2$ denotes that that $id1$ and $id2$ are different (not equal). Moreover, in order to say that $y$ has the name of $x$, a simple logic function $nameOf(x, y)$ is declared.

Besides this, a binary predicate is used to express the notion of membership in our approach. For instance, given $Model(x)$ and $Node(y)$, $Member(x, y)$ denotes that *element y is a member of the model x*. This predicate can be re-written using ($\in_m$) symbol as $y \in_m x$. Therefore, the statement says that *the node y is a member of the model x* can be expressed as: $x : Model, y : Node \bullet (y \in_m x)$

In the same context, a binary predicate is used to express the notion of property. For instance, given $Node(x)$ and $Property(y)$, $PropertyOf(x, y)$ denotes that *property y belongs to the node x*. This predicate can be re-written using ($\in_p$) symbol as $y \in_p x$. Therefore, the statement says that *y is a property of the node x* can be expressed as: $x : Node, y : Property \bullet (y \in_p x)$.

Furthermore, another binary predicate is used to express the notion of widget. For instance, given $Node(x)$ and $Widget(y)$, $WidgetIn(x, y)$ denotes that *widget y is a control in node x*. The node x must be a GUI window. This predicate can be re-written using ($\in_c$) symbol as $y \in_c x$. Therefore, the statement says that *y is a GUI control in the node x* can be expressed as: $x : Node, y : Widget \bullet (y \in_c x)$.

To abbreviate this, the symbol $\in$ without any subscript character is used for indicating *Member*, *WidgetIn* and *PropertyOf* predicates. The interpretation of $\in$ is obtained based on the types of elements appearing in the logic statement.

Additionally, in our formalisation appraoch, there is a need for the use of an exclusive disjunction operator ($\oplus$) operation to emphisize that only one of the operand predicates is *true* **but not both**. For instance, any *Element* in the *Model* can be either a *Node* or an *Edge*. This cannot be expressed adequately using the direct logical ($\vee$), in which it produces *true* output when one or both operand predicates are *true*. The statement $Node(x) \oplus Edge(x)$ can be bootstrapped using logical *And* ($\wedge$), *Or* ($\vee$), and *Negation* ($\neg$) operations as: $(Node(x) \wedge \neg Edge(x)) \vee (\neg Node(x) \wedge Edge(x))$.

Moreover, the uniqueness quantifier (!) is used, here, to indicate that exactly **one and only one** unique property exists in a certain *Element*. For instance, the phrase *a Node has a unique Identifier* cannot be expressed without (!) to emphisize that the *Identifier* is unique and only one. The uniqueness quantifier $\exists!\ id : Identifier \bullet Id(n, id)$ appears in the statement: $\forall\ n : Node \bullet (\exists!\ id : Identifier \bullet Id(m, id))$ can be bootstrapped using basic *existential* ($\exists$) and *universal* ($\forall$) quantifiers with the logical *And* ($\wedge$), *Or* ($\vee$), and *Negation* ($\neg$) operations as: $\exists\ id1 \bullet (Id(n, id1) \wedge \neg \exists\ id2 \bullet (Id(n, id2) \wedge id2 \neq id1))$.

Regarding the transformation rule definition, it is required to consider, for each rule, what is the next-level-down translation issue in order to specify exactly what goes where. Therfore, it is assumed that a general polymorphic translation function *tr* is defined: $\forall\ Source, Target \bullet tr : Source \rightarrow Target$. Each of subsequent translation rules, presented later in Chapter 8, is a particular type-instantiation of this function, with a different overloaded rule. This means that the function $tr(x)$ can be used in a rule-body to refer to the translation of the next bit of sub-structure x. Section 8.3.2.1 in Chapter 8 is an example of translating $DEntity \rightarrow Table$.

## 3.4 Specifications of the $\mu$ML Metamodel

This section introduces a set of common policies that are applicable to all $\mu$ML model elements. The presented laws are used to specify the concepts of $\mu$ML metamodel using FOPL. In order to construct valid instance models, the following laws must not be violated. The following table (3.1) summaries all predicates used in describing $\mu$ML Metamodel.

| Predicate | Meaning | Syntactic Suger |
|---|---|---|
| *Model(x)* | $x$ is a Model | *x:Model* |
| *Node(x)* | $x$ is a Node | *x:Node* |
| *Edge(x)* | $x$ is an Edge | *x:Edge* |
| *Arc(x)* | $x$ is an Arc | *x:Arc* |
| *Flow(x)* | $x$ is a Flow | *x:Flow* |
| *Transition(x)* | $x$ is a Transition | *x:Transition* |
| *Relationship(x)* | $x$ is a Relationship | *x:Relationship* |
| *Identifier(x)* | $x$ is an Identifier | *x:Identifier* |
| *Id(x, m, y)* | $y$ is an Id of $x$ in the model $m$ | |
| *Member(x, y)* | $y$ is a member of $x$ | $x \in_m y$ |
| *PropertyOf(x, y)* | $y$ is a property of $x$ | $x \in_p y$ |
| *WidgetIn(x, y)* | $y$ is a GUI widget in $x$ | $x \in_c y$ |
| *NameOf(x, y)* | $y$ has the name of $x$ | |
| *Source(x, y)* | $y$ is a source of the Flow $x$ | |
| *Target(x, y)* | $y$ is a target of the Flow $x$ | |
| *Connects(x, y, z)* | An Arc $x$ connects a source $y$ to a target $z$ | |
| *Links(x, y, z)* | A Relationship $x$ links $y$ and $z$ | |

Table 3.1: Predicates for $\mu$ML Metamodel

**Law 1.** Each model $m$ has one identifier $id$.

$$\begin{aligned} &\forall\, m : Model \\ &\qquad \longrightarrow \\ &\qquad\quad \exists!\, id : Identifier \;\bullet\; Id(m, id) \end{aligned} \tag{3.1}$$

**Law 2.** Each model $m$ has some nodes.

$$\begin{aligned}&\forall\ m : Model\\&\qquad\longrightarrow\\&\qquad\quad \exists\ n : Node\ \bullet (n\ \in\ m)\end{aligned} \tag{3.2}$$

**Law 3.** Each node $n$ in a model $m$ has one unique identifier $id$.

$$\begin{aligned}&\forall\ m : Model, n : Node\\&\quad\bullet\ (n\ \in_m\ m)\\&\qquad\longrightarrow\\&\qquad\quad \exists!\ id : Identifier\ \bullet\ Id(n, m, id)\end{aligned} \tag{3.3}$$

**Law 4.** The node can be either a task, goal, object, entity, actor, state, or window.

$$\begin{aligned}&\forall\ n : Node\\&\quad\longrightarrow\\&\qquad Task(n)\ \oplus\ Object(n)\ \oplus\ Entity(n)\ \oplus\ Actor(n)\\&\qquad\oplus\ Goal(n)\ \oplus\ State(n)\ \oplus\ Window(n)\end{aligned} \tag{3.4}$$

**Law 5.** Each model $m$ has some edges.

$$\begin{aligned}&\forall\ m : Model\\&\qquad\longrightarrow\\&\qquad\quad \exists\ e : Edge\ \bullet\ (e\ \in\ m)\end{aligned} \tag{3.5}$$

**Law 6.** Any edge can be either an arc or a relationship.

$$\begin{aligned}&\forall\ a : Edge\\&\quad\longrightarrow\\&\qquad Arc(a)\ \oplus\ Relationship(a)\end{aligned} \tag{3.6}$$

**Law 7.** Any arc can be either a flow or a transition or a stakeholder interaction (participation).

$$\begin{aligned}&\forall\ a : Arc\\&\quad\longrightarrow\\&\qquad Flow(a)\ \oplus\ Transition(a)\ \oplus\ Participation(a)\end{aligned} \tag{3.7}$$

**Law 8.** A relationship can be a generalisation, composition, dependency or an association.

$$\begin{aligned} \forall\ r : \mathit{Relationship} \\ \longrightarrow \\ & \mathit{Generalisation}(r) \ \oplus \ \mathit{Composition}(r) \\ & \oplus \ \mathit{Dependency}(r) \ \oplus \ \mathit{Association}(r) \end{aligned} \tag{3.8}$$

**Law 9.** Each Node $n$ has one unique identifier $id$.

$$\begin{aligned} \forall\ n : \mathit{Node} \\ \longrightarrow \\ & \exists!\ id : \mathit{Identifier} \ \bullet \ \mathit{Id}(n, id) \end{aligned} \tag{3.9}$$

**Law 10.** In the model, each edge connects two nodes

$$\begin{aligned} \forall\ m : \mathit{Model}, a : \mathit{Edge}, (n1, n2) : \mathit{Node} \\ \bullet\ (a, n1, n2 \ \in \ m),\ \mathit{Connects}(a, n1, n2) \end{aligned} \tag{3.10}$$

**Law 11.** In the *Model*, each *Edge* connects a *Source Node* to a *Target Node*

$$\begin{aligned} \forall\ m : \mathit{Model}, a : \mathit{Edge}, (n1, n2 : \mathit{Node}) \\ \bullet\ (a, n1, n2 \ \in \ m) \\ \wedge\ \mathit{Connects}(a, n1, n2) \\ \longrightarrow \\ & \mathit{Source}(a, n1) \ \wedge \ \mathit{Target}(a, n2) \end{aligned} \tag{3.11}$$

**Law 12.** In the *Model*, each *Dependency* connects a *Source Node* to a *Target Node*

$$\begin{aligned} \forall\ m : \mathit{Model}, d : \mathit{Dependency}, (n1, n2 : \mathit{Node}) \\ \bullet\ (d, n1, n2 \ \in \ m) \\ \wedge\ \mathit{Connects}(d, n1, n2) \\ \longrightarrow \\ & \mathit{Source}(d, n1) \ \wedge \ \mathit{Target}(d, n2) \end{aligned} \tag{3.12}$$

**Law 13.** In the *Model*, each relationship links two nodes $n1$ and $n2$, when it connects $n1$ to $n2$ and connects $n2$ to $n1$.

$$\begin{aligned} \forall\ m : \mathit{Model}, r : \mathit{Relationship}, (n1, n2 : \mathit{Node}) \\ \bullet\ ((r, n1, n2 \ \in \ m) \\ \wedge\ \mathit{Links}(r, n1, n2)) \\ \longrightarrow \\ & (\mathit{Connects}(r, n1, n2) \ \vee \ \mathit{Connects}(r, n2, n1)) \end{aligned} \tag{3.13}$$

**Law 14.** In the *Model*, each association links two nodes.

$$\begin{aligned} &\forall\ m : Model, a : Association, (n1, n2 : Node) \\ &\quad \bullet\ ((a, n1, n2\ \in\ m)\ \wedge\ Links(a, n1, n2)) \end{aligned} \tag{3.14}$$

According to the $\mu$ML metamodel (Figure 3.2), the concept *Relationship* serves to declare directed and undirected relationships between two nodes or a node with itself. *End-roles* of this kind of *Edge* hold some additional features and accept further boolean predicates than those *roles* appear in *Arc*, such as multiplicities. Based on the type of relationship, the end-role might be, for instance, a *whole/part*, as it appears in *Composition* relationships, and a *parent/child* as it occurs in *Generalisation* relationships.

It worth saying that the predicate $Links(x, y, z)$ represents the undirected relationship (connection) between the node $y$ and the node $z$, whereas, the $Connects(a, b, c)$ expresses the directed relationship between the node $a$ and the node $b$, in which the relationship $a$ connects the source $b$ to the target $c$.

## 3.5 Outlook on the Chapter

The chapter introduced an overview of the proposed solution for the research problem, idenfitied earlier in Chapter 1. More specifically, the overall structure of the BUILD method is discussed, including the related system views, example of the graphical notation of models and critical transformation steps at each stage of developement. Development activities that are mastered by end-users or are led automatically by rules are also highlighted for each phase.

Furthermore, the overall structure and components of model transformation and code generation phase, within BUILD, are presented. This includes the internal architecture of transformation steps, the style of designing mapping rules (*hybrid*), and programming language that is used to implement them (*Java*).

Moreover, the modelling language ($\mu$ML), which is used in BUILD, for expressing required system specifications is presented briefly in this chapter. The core elements of the language were defined using metamodelling strategy in order to express all critical concepts of the information system domain. Semantics of the metamodel elements are formalised using First-Order Predicate Logic (FOPL) with a number of selected extensions, namely, subtyping, equality, membership, exclusive disjunction and uniqueness quantification.

# 4

# $\mu$ML Concepts and Notations

*In the Requirement Sketching Phase*

"The most difficult part of requirements gathering is not the act of recording what the user wants, it is the exploratory development activity of helping users figure out what they want"

Steve McConnell

## 4.1 Context

This chapter describes, in detail, the Micro Modelling Language ($\mu$ML) models appearing in the Requirement Sketching Phase, namely, Task, Impact and Information Model. For each model, the chapter presents how the model satisfies its purpose within the framework. Additionally it discusses each model concept and graphical notation for that concept, also providing a formally defined semantics using a set of FOPL policies.

## 4.2 Overview of the Task Model

A *Task Model* is the highest level, ***manually-constructed***, diagrammatic representation of a business in terms of stakeholders, the tasks they perform and the goals to be achieved. It is a structural model that captures the wider context and the initial requirements of the business in a straightforward way by linking every task with a purpose or goal that is significant and worth recording in the business domain.

The participation of stakeholders, human or external system, in tasks is depicted in a clear way to be modelled naturally by the business end-user. Stakeholders might be specified / generalised to appear at different levels of abstraction. Precise types of participation and task occurrence, namely, multiplicity and optionality, are adopted to reflect the designer's natural intuitions about the ways in which they interact with the system and how the business is performed. These basic specifications are developed during the transformation stages into further design decisions used for implementation, such as iterations and conditional branches.

Apart from this, business tasks in the *Task Model* are fully compositional, and they may be aggregated or decomposed as desired to reveal their *part/whole* structure. Additionally, they may be presented at different degrees of abstraction to reveal their *general/specialised* structure. There is no restriction on the ideal granularity of a task; a stakeholder might perceive tasks arbitrarily at different levels of detail.

The possible ways in which parts collaborate in order to perform the main task and meet the intended goal are considered from the end-users perspective. One part, for instance, might be executed independently to perform all aspects of the original task. Consequently, the parts are noted as options or choices. In some cases, internal flows between the parts might conceptually exist, each part performing an aspect of the main task. Both features can be captured by end-users using a simple notation.

Each group of business tasks that falls within the scope of a business practice might be modelled as a boundary that represents a separate part of the system. This feature can be adopted in order to document and organise the incremental delivery phases if planned. Technically, boundaries are used to describe the details of a composed task, including subtasks and their interactions with stakeholders.

The *Task Model*, in the proposed metamodelling hierarchy, is considered a kind of $Model$. Thus, concepts of the *Task Model* can be defined using FOPL and relate to the corresponding $\mu$ML metamodel concepts.

**Law 15.** The *TaskModel* is a kind of the *Model*. Thus, every *TaskModel* is a *Model*.

$$
\begin{aligned}
& TaskModel \ <: \ Model \\
& \forall \ m \ \bullet \ TaskModel(m) \\
& \qquad\qquad\qquad \longrightarrow \\
& \qquad\qquad\qquad\quad Model(m)
\end{aligned}
$$

### 4.2.1 Notation and Semantics of the Task Model

In this section, each concept that appears in the *Task Model* is introduced with some details about its usage and how it is visualised graphically in the model. It is worth emphasising that a number of UML notations in the *Use Case* and *Class Diagram* are adopted and reused with some differences in their semantics. The differences allow us to apply the same notations in more modelling contexts, with a well-defined interpretation. This means our concepts might have a slightly different meaning from standard UML[70]. The following table (4.1) summaries all predicates used in describing $\mu$ML *Task Model*.

In order to motivate the list of policies that specifies the *Task Model*, some definitions must be identified first. According to the $\mu$ML metamodel hierarchy, the core elements of the *Task Model* are subtypes of either *Node* or *Arc*:

$$
\begin{aligned}
(Actor \ <: \ Node) \ \wedge \ (Task \ <: \ Node) \ \wedge \ (Goal \ <: \ Node) \wedge \ (Participation \ <: \ Arc) \\
\wedge \ (Input \ <: \ Participation) \ \wedge \ (Output \ <: \ Participation)
\end{aligned}
$$

| Predicate | Meaning | Syntactic Sugar |
|---|---|---|
| *TaskModel(x)* | $x$ is a Task Model | *x:TaskModel* |
| *Task(x)* | $x$ is a Business Task | *x:Task* |
| *Goal(x)* | $x$ is a Business Goal | *x:Goal* |
| *Actor(x)* | $x$ is an Actor | *x:Actor* |
| *Human(x)* | $x$ is (Human) user | |
| *ExSystem(x)* | $x$ is an external system | |
| *Participation(x)* | $x$ is a Participation | *x:Participation* |
| *Input(x)* | $x$ is an Input participation | *x:Input* |
| *Output(x)* | $x$ is a Output participation | *x:Output* |
| *Composition(x)* | $x$ is a Composition | *x:Composition* |
| *Total(x)* | $x$ is a Total Composition | |
| *Generalisation(x)* | $x$ is an Generalisation | *x:Generalisation* |
| *Disjoint(x)* | $x$ is a Disjoint Generalisation | |
| *Role(x)* | $x$ is an end-role | *x:Role* |
| *General(x, y)* | $x$ is general in *Generalisation* $y$ | |
| *Specific(x, y)* | $x$ is specific in *Generalisation* $y$ | |
| *Whole(x, y)* | $x$ is whole in *Composition* $y$ | |
| *Part(x, y)* | $x$ is part in *Composition* $y$ | |

Table 4.1: Predicates for $\mu$ML Task Model

From that, any *Node* in the model can be either a *Task, Goal* or an *Actor*. This can be expressed in logic as:

$$\begin{aligned} &\forall\ n : Node,\ \forall\ m : TaskModel \\ &\quad \bullet\ (n\ \in_m\ m) \\ &\qquad \longrightarrow \\ &\qquad\quad Actor(n)\ \oplus\ Task(n)\ \oplus\ Goal(n) \end{aligned}$$

In the same context, any *Arc* appears in the *Task Model* is a *Participation*:

$$\begin{aligned} &\forall\ a : Arc,\ \forall\ m : TaskModel \\ &\quad \bullet\ (a\ \in_m\ m) \\ &\qquad \longrightarrow \\ &\qquad\quad Participation(a) \end{aligned}$$

#### 4.2.1.1 Business Goal

The notion of a business goal is used to express some high-level aim, which is not directly measurable. It is an essential concept in the model from business management prespective. Some business tasks might be restructured during transformation steps. Relating some business tasks to a goal, at the early stage, enables the achievement of that goal to be measured, in the later design.

Because it is more abstract than a business task, we use a dashed outline ellipse to represents abstract aims in the model, as it illustrated in Figure 4.1. The semantics of Goals can be formalised using the following *FOPL* policies:

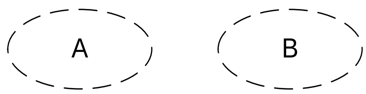



Figure 4.1: Task model. Goals

According to the $\mu$ML metamodel, *Business Goal* is defined as a subtype of the *Node* element ($Goal \ <: \ Node$). Thus, it is identified using a unique identifier. This can be written formally in logic as:

$$\forall\ m : TaskModel,\ \forall\ g : Goal\ \bullet\ (a\ \in_m\ m), \\ \exists!\ id : Identifier\ \bullet\ Id(g, m, id)$$

#### 4.2.1.2 Business Task

To depict tasks, we reuse the ellipse notation for a UML use case, but do not restrict this to mean a single, small-scale interaction, or use case. This allows for larger, aggregated tasks to be described, as well as single use cases. *Tasks* always achieve measurable objectives, within the business domain. Figure 4.2 illustrates the graphical notation of tasks in *Task model.*

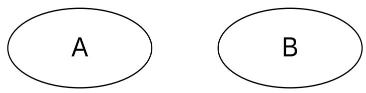



Figure 4.2: Task model. Tasks

Similar to *Business Goal, Business Task* is defined as a subtype of the *Node* element ($Task \ <: \ Node$). Thus, it is identified using a unique identifier that formalised in FOPL as:

$$\forall\ m : TaskModel,\ \forall\ t : Task\ \bullet\ (t\ \in_m\ m), \\ \exists!\ id : Identifier\ \bullet\ Id(t, m, id)$$

#### 4.2.1.3 Participant

The system stakeholders are represented as UML actors that appear in the model as Human or an external system participant $\forall\ a : Actor\ \longrightarrow\ Human(a)\ \oplus\ ExSystem(a)$. Human actors are depicted as stick figures, Figure 4.3 (left), whereas external systems are depicted as box nodes, reused from the UML deployment diagram, Figure 4.3 (right).

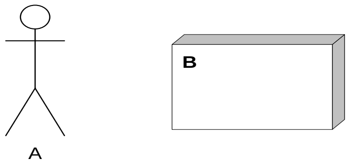



Figure 4.3: Task model. Actors

As it described in the $\mu$ML metamodel, *Actors* in the *Task Model* are *Nodes*, ($Actor\ <:\ Node$), each participant (actor) has only one unique identifier:

$$\begin{aligned} &\forall\ m : TaskModel,\ \forall\ a : Actor\ \bullet\ (a\ \in_m\ m), \\ &\qquad \exists!\ id : Identifier\ \bullet\ Id(a, m, id) \end{aligned}$$

#### 4.2.1.4 Participation

*Participation* arrows capture system interactions with stakeholders, which can be represented as *Input* or *Output* participation. Both types are considered subtypes of *Participation* in the $\mu$ML metamodel ($(Input, Output)\ <:\ Participation$). Therefore, any *Participation* arrow appears in the *Task Model* either a *Input* or an *Output* arrow:

$$\begin{aligned} &\forall\ p : Participation,\ \forall\ m : TaskModel \\ &\qquad \bullet\ (p\ \in_m\ m) \\ &\qquad\qquad \longrightarrow \\ &\qquad\qquad\quad Input(p)\ \oplus\ Output(p) \end{aligned}$$

The input arrow indicates that an actor participates in a task by supplying some external inputs (e.g information). This is determined by the direction of the arrow from the actor toward the task. Similarly, the output arrow indicates that an actor participates in a task by receiving some information from the system. Figure 4.4 illustrates the graphical notation of participation in the *Task model.*

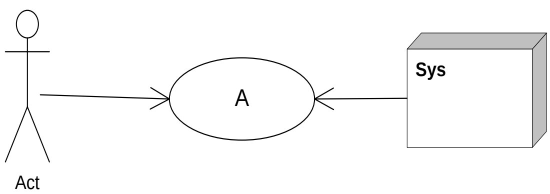



Figure 4.4: Task model. Participation

The *Input* participation can be defined in the following theorem that follows law (axiom) 10, substituting variously *Task* or *Actor* for *Node*, and *Input* for *Edge*. This can be written as:

$$\begin{aligned}\forall\ p : Input,\ \forall\ t : Task,\ \forall\ a : Actor \\ \bullet\ (Connects(p, a, t) \\ \wedge\ Source(p, a)\ \wedge\ Target(p, t))\end{aligned}$$

Following the similar way of defining *Input* participation, It can be said that any *Output* connects a task, as *Source*, to an actor, as *Target*. This can be written as:

$$\begin{aligned}\forall\ p : Output,\ \forall\ t : Task,\ \forall\ a : Actor \\ \bullet\ (Connects(p, t, a) \\ \wedge\ Source(p, t)\ \wedge\ Target(p, a))\end{aligned}$$

#### 4.2.1.5 Generalisation

Tasks may be described in general, or specific terms, in order to express the "is-a" relationship. In consequence, a task is regarded as an ancestor if it has a specialised task that inherits from it. In the same context, actors are treated likewise. We reuse the UML white triangle arrowhead notation for generalisation (inheritance), pointing to the general task (Figure 4.5) . The following figure illustrates that the Task A is a task generalisation of the Task B and C.

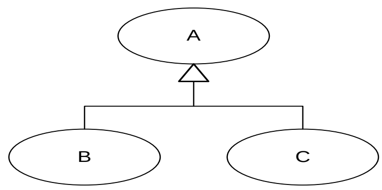



Figure 4.5: Task model. Generalisation

According to the $\mu$ML metamodel, generalisation is considered a kind of *Relationship* (*Generalisation* <: *Relationship*) in which it connects the more specific task to the general one. The following logic expression represents the notion of *Generalistion* in the *TaskModel*:

$$\begin{aligned}&\forall\ t1, t2 : Task,\ \forall\ g : Generalisation,\ \forall\ m : TaskModel \\ &\quad \bullet\ ((t1, t2, g\ \in_m\ m)\ \wedge\ Connects(g, t1, t2)) \longrightarrow \\ &\qquad\qquad General(g, t2)\ \wedge\ Specific(g, t1)\end{aligned}$$

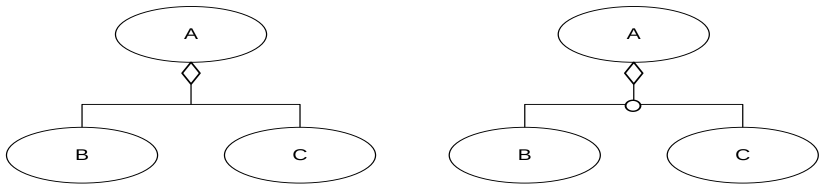
#### 4.2.1.6 Composition

Aggregate tasks may be composed of other tasks to represent the "is part of" relationship. The notion of composition is a primary concept in compositional systems. We therefore prefer this term to the alternative term aggregation. We reuse the UML white diamond arrowhead and disjoint notations for whole/parts structures, pointing to the composed task (as a consequence, the include-dependency is redundant).

According to the $\mu$ML metamodel, composition is considered a kind of *Relationship* ($Composition$ <: *Relationship*) thatconnects the part tasks to the whole one. This relationship appears in the *Task Model* in two styles: total or partial composition $\forall\ c : Composition, m : TaskModel\ \bullet\ ((c\ \in_m\ m)\ \wedge\ (Total(c)\ \oplus\ \neg\ Total(c)))$. The following logic expression represents the notion of *Composition* in the *TaskModel*:

$$\begin{aligned}&\forall\ t1, t2 : Task,\ \forall\ c : Composition,\ \forall\ m : TaskModel\\&\quad\bullet\ ((t1, t2, c\ \in_m\ m)\ \wedge\ Connects(c, t1, t2))\\&\qquad\longrightarrow\\&\qquad Whole(c, t2)\ \wedge\ Part(c, t1)\end{aligned}$$

From the business-user perspective, all "part" tasks involved in a white diamond (with *disjoint*) arrowhead composition represent options, or choices. In each execution, one part may perform independently the complete job of the ancestor task. This feature can be expressed using a unary predicate as: $c : Composition\ \wedge\ Disjoint(c)$. Figure 4.6 (a) below illustrates the visual representation of task optionality feature in the *Task Model*.




(a) Task model. White diamond Composition

(b) Task model. White diamond (with *disjoint*) Composition

Figure 4.6: Task Model. Compositions

Besides, *part* tasks involved in a white diamond arrowhead composition indicate that there are such data flows and sequence interconnections between them, that they must be considered as all collaborating together to perform the complete job of their ancestor. Technically, there are internal variables that hold and pass data from a task to another to complete the ancestor mission. This feature can be expressed using the negation of *Disjoint* predicate as: $c : Composition\ \wedge\ \neg\ Disjoint(c)$. Figure 4.6 (b) demonstrates this.

As previously discussed, it is worth saying that the *Task Model* takes a different position from the use case model, where composition would create something larger than a use case. The current UML Use Case Diagram, does not formally compose use cases (the composition would no longer be a use case). The *Composition* relationship used here obviates the need for extra "include dependency" relationships. UML[70] uses composition to denote a specific flavour of aggregation, which unfairly limits the use of the term *composition*, which is used more generally in English to describe composed relationships.

#### 4.2.1.7 Boundary

A rectangular system boundary may be drawn around groups of tasks, to indicate that these tasks fall within the scope of the system to be developed, while other tasks fall outside. Therefore, the *Task Model* might contain a number of logical *Boundary* nodes: $\forall\ m : TaskModel,\ \exists\ b : Boundary\ \bullet\ (b\ \in_m\ m)$.

### 4.2.2 The Significance of Task Model

The importance of *Task Model* emerges through its use of subtle concepts that allow the model transformation process to take different design decisions. These features are highlighted below and discussed later in detail in Chapter 4.

- Differentiating between the two types of stakeholders at early stage contributes to critical design decisions of user interfaces or system inter-operations, which will later be required.
- Differentiating between the two types of participations, *Input* and *Output*, at early stage allows the automatic prediction of the Graphical User Interface (GUI) components and controls required in systems' screens.
- Differentiating between the two types of compositions at this stage automatically determines the control flow between the part tasks. Each subtask, produced at later stages, might be independently executed to perform the main task, or a particular flow might exist between them based on their atomic CRUD operations.

## 4.3 Overview of the Impact Model

The *Impact Model* is the highest level, ***manually-constructed***, diagrammatic representation of a business task's interaction with the logical data within a system. It does not describe the time or order of processing but rather what information is produced and consumed by each task. It is assumed that each task has an effect on objects of reading, writing or updating (both reading and writing). These influences appear in the model via several kinds of arcs, each with a distinct meaning, that represent each *CRUD* (*create, read, update* and *delete*) operation.

The model might be compositional, expanding the level of detail at which a task is described. Like the *Task Model*, this feature is captured using system boundaries that contain the parts of the decomposed task with their interactions with the data. It is essential that the tasks in both the *Task* and *Impact Model* remain at the same level of granularity in order to generate a consistent layer in our layered model transformation approach.

The business data appears in the *Impact Model* as objects, corresponding to physical objects and documents, or logical data used by the business. It can be said that any object holds useful information about stakeholders or physical business items. Each object in the model has a lifespan that is determined by tracing the effects of *CRUD* on it.

The notions of object multiplicity and alternatives are also presented in the *Impact Model* to reflect end-user logical thoughts about how an object might participate in a task. This is determined from the perspective of the tasks using simple and easily-adopted notations. For each execution of the task, we identify how many other objects are affected. In addition, alternative kinds of data may be acceptable as inputs to or outputs from certain tasks. This is captured using a simple notation that does not complicate the model.

**Law 16.** The *ImpactModel* is a kind of *Model*.

$$
\begin{aligned}
&ImpactModel \ <: \ Model \\
&\forall\, m \ \bullet \ ImpactModel(m) \\
&\qquad\qquad\qquad\longrightarrow \\
&\qquad\qquad\qquad\quad Model(m)
\end{aligned}
$$

### 4.3.1 Notation and Semantics of the Impact Model

In this section, each concept that occurs in the *Impact Model* is introduced with some details about its usage and how it is visualised graphically in the model. Similar to our strategy in adopting UML for the *Task Model* notation, a number of UML notations taken from the *Class* and *Use Case Diagram* are reused with slightly different meanings than the usual UML semantics. The semantics of the proposed concepts are expressed in the following sub-sections. The following table (4.2) summaries all predicates used in describing $\mu$ML Impact Model.

| Predicate | Meaning | Syntactic Suger |
|---|---|---|
| *ImpactModel(x)* | $x$ is an Impact Model | *x:ImpactModel* |
| *ImpBoundary(x)* | $x$ is an Impact Boundary | *x:ImpBoundary* |
| *ImpTask(x)* | $x$ is an Impact Task | *x:ImpTask* |
| *ImpObject(x)* | $x$ is an Impact Object | *x:ImpObject* |
| *Flow(x)* | $x$ is an Impact Flow | *x:Flow* |
| *InputFlow(x)* | $x$ is an Input Flow | *x:InputFlow* |
| *OutputFlow(x)* | $x$ is a Output Flow | *x:OutputFlow* |
| *CreateFlow(x)* | $x$ is an Create Flow | *x:CreateFlow* |
| *ReadFlow(x)* | $x$ is a Read Flow | *x:ReadFlow* |
| *UpdateFlow(x)* | $x$ is an Update Flow | *x:UpdateFlow* |
| *DeleteFlow(x)* | $x$ is a Delete Flow | *x:DeleteFlow* |
| *ImpConjunction(x)* | $x$ is a disjoint impact combinator | *x:ImpConjunction* |
| *ImpRole(x)* | $x$ is an end-role | *x:ImpRole* |

Table 4.2: Predicates for $\mu$ML Impact Model

In order to motivate the list of policies that specifies the *Impact Model*, two basic laws must be identified first as follows:

**Law 17.** Any node in the *ImpactModel* can be either a task or an object or a conjunction.

$$
\begin{aligned}
&(ImpTask \ <: \ Node) \ \wedge \ (ImpObject \ <: \ Node) \\
&\forall\, n : Node, \ \forall\, m : ImpactModel \ \bullet \ (n \ \in_m \ m) \\
&\qquad\qquad\qquad\longrightarrow \\
&\qquad\qquad\qquad\quad ImpTask(n) \ \oplus \ ImpObject(n) \ \oplus \ ImpConjunction(n)
\end{aligned}
$$

**Law 18.** Any flow in the *ImpactModel* can be either create, read, update, delete, write, input, or output flow.

$$
\begin{aligned}
(Flow <: Arc) \wedge (InputFlow <: Flow) \wedge \\
(OutputFlow <: Flow) \wedge (CreateFlow <: Flow) \wedge \\
WriteFlow <: Flow) \wedge (ReadFlow <: Flow) \wedge \\
(UpdateFlow <: Flow) \wedge (DeleteFlow <: Flow)
\end{aligned}
$$

$$
\begin{aligned}
\forall\, f : Flow,\ \forall\, m : ImpactModel \bullet (f \in_m m) \\
\longrightarrow \\
& CreateFlow(f) \oplus ReadFlow(f) \oplus UpdateFlow(f) \\
& \oplus DeleteFlow(f) \oplus WriteFlow(f) \oplus InputFlow(f) \\
& \oplus OutputFlow(f)
\end{aligned}
$$

**Law 19.** Any node in the *ImpactModel* is involved in at least one flow as a source or a target.

$$
\begin{aligned}
\forall\, n : Node,\ \forall\, m : ImpModel \\
\bullet (n \in_m m) \\
\longrightarrow \\
\exists\, f \bullet Flow(f) \bullet (f \in_m m) \wedge (Source(f,n) \oplus Target(f,n))
\end{aligned}
$$

#### 4.3.1.1 Impact Task

Tasks are depicted as in the task model, using the ellipse node. In this model, the emphasis is on the impact that tasks have upon objects at a given level of granularity, so no task structure is shown. Likewise, no actors or external systems are depicted. Figure 4.7 below demonstrates the A and B are two Impact tasks.

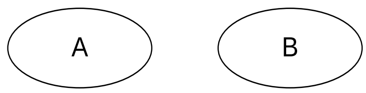



Figure 4.7: Impact model. Tasks

In regard to the $\mu$ML metamodel, Impact Task is considered a subtype of the metaclass *Node* element ($ImpTask <: Node$). Therefore, it is identified using a unique identifier. This can be written formally in logic as:

$$
\begin{aligned}
\forall\, m : ImpactModel,\ \forall\, t : ImpTask \bullet (t \in_m m), \\
\exists!\, id : Identifier \bullet Id(t, m, id)
\end{aligned}
$$

Given ($ImpTask$ <: $Node$) from the defined metamodel, it can be deduced that any *ImpTask* in the *ImpactModel* can be either a source or a target of such a flow. This is a theorem that follows from the earlier law 19, given previously in Chapter 3, by substituting of *Node*.

$$\begin{aligned}
&\forall\, t : ImpTask,\ \forall\, m : ImpModel \ \bullet\ (t\ \in_m\ m)\\
&\qquad\longrightarrow\\
&\qquad\quad \exists\, f\ \bullet Flow(f)\ \bullet\ (f\ \in_m\ m)\ \wedge\ (Source(f,t)\ \oplus\ Target(f,t))
\end{aligned}$$

#### 4.3.1.2 Impact Object

The UML rectangular nodes, taken from the UML class diagram, are used to represent information objects, corresponding to physical business objects (entities) and documents, or logical data used by the business in the real-world (Figure 4.8). Any object that bears useful information may be modelled. This may include objects storing logical information about human actors, or other physical things.

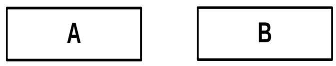



Figure 4.8: Impact model. Objects

Similar to the way of declaring *Impact Task* according to the $\mu$ML metamodel, Impact Object is also considered a subtype of the *Node* element ($ImpObject$ <: $Node$). Therefore, it is identified using a unique identifier. This can be written formally in logic as:

$$\begin{aligned}
&\forall\, m : ImpactModel,\ \forall\, obj : ImpObject\ \bullet\ (obj\ \in_m\ m),\\
&\qquad \exists!\, id : Identifier\ \bullet\ Id(obj, m, id)
\end{aligned}$$

Similar to the *ImpTask*, given ($ImpObject$ <: $Node$) from the defined metamodel, it can be deduced that any *ImpObject* in the *ImpactModel* can be either a source or a target of such a flow. This is a theorem that follows from the earlier law 19, by substitution of *Node*.

$$\begin{aligned}
&\forall\, obj : ImpObject,\ \forall\, m : ImpModel\ \bullet\ (obj\ \in_m\ m)\\
&\qquad\longrightarrow\\
&\qquad\quad \exists\, f\ \bullet\ Flow(f)\ \bullet\ (f\ \in_m\ m)\\
&\qquad\quad \wedge\ (Source(f, obj)\ \oplus\ Target(f, obj))
\end{aligned}$$

#### 4.3.1.3 Impacts (flows)

The impacts that occur on objects are shown visually by diverse-headed arrows. The style of arrowhead indicates the direction of information flow, or the impact, between a task and an object. Figure 4.9[1] below illustrates the various types of impacts in the *Impact Model*.

(a) Create (b) Read/Write (c) Update (d) Destroy

Figure 4.9: Impact model. Impacts

Given ($Flow$ <: $Arc$ <: $Edge$), ($ImpModel$ <: $Model$) from the constructed metamodel, it can be deduced that each $Flow$ connects two distinct nodes. This is a theorem that follows from Law 10, by substituting for $Model$ and $Edge$.

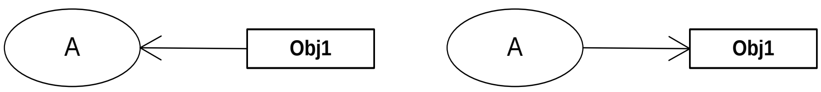
$$\forall\ (n1, n2) : Node,\ \forall\ m : ImpactModel,\ \forall f : Flow$$
$$\bullet\ ((f\ \in_m\ m)\ \land\ Connects(f, n1, n2))$$

In order to define formally the various types of impacts in the model, a basic law is presented:

**Law 20.** Any flow, in the *Impact Model*, that connects an object to a task is a *read* flow, whereas any flow connects a task to an object can be any flow out of a *create, delete* or *write* flow. This can be expressed in FOPL as:

$$\begin{aligned}
&\forall\ m : ImpactModel,\ \forall\ f : Flow,\ \forall\ e : ImpObject,\\
&\forall\ t : ImpTask\ \bullet\ (e, t, f\ \in_m\ m)\ \land\ Connects(f, e, t)\\
&\qquad\longrightarrow\\
&\qquad ReadFlow(f)\\
&\land\ Connects(f, t, e)\\
&\qquad\longrightarrow\\
&\qquad CreateFlow(f)\ \oplus\ WriteFlow(f)\ \oplus\ DeleteFlow(f)
\end{aligned}$$

Visually, the open-headed arrows are utilised to represent the impact on objects, which may be read, or written, Figure 4.9(B). Figure 4.10a below demonstrates that the *Object Obj1* is read by the the *Task A*, whereas, Figure 4.10b shows that the *Obj1* is written by a *Task A*.




(a) Impact model. Read (b) Impact model. Write

Figure 4.10: Impact Model. Read/Write Object

[1]The interpretation of Read/Write flows depends on whether the source or target is an object.

In addition to this, a double-ended arrow indicates both reading and writing from the same object, which presents an updating effect, Figure 4.9(C). The following law expresses the decomposition of an *Update* into separate *Read* and *Write* flows.

**Law 21.** Any flow, in the Impact Model, that connects bi-directionally an object to a task is an update flow. This can be expressed in FOPL as:

$$\begin{aligned} &\forall\ m : ImpactModel,\ \forall\ f : Flow,\ \forall\ e : ImpObject,\ \forall\ t : ImpTask \\ &\quad \bullet\ (e, t, f\ \in_m\ m)\ \wedge\ Connects(f, e, t)\ \wedge\ Connects(f, t, e) \\ &\quad\quad \longrightarrow \\ &\quad\quad\quad UpdateFlow(f) \end{aligned}$$

The basic interpretation of this decomposition step means that each object, involved in this type of impact, must be read first, before it can be updated or written. This interpretation comes into play later on, when timing becomes significant, since the current model is simply concerned with flow structure. The following figure (4.11) shows that the *Obj1* is updated by a *Task B*.

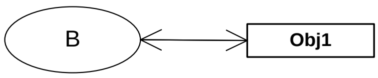
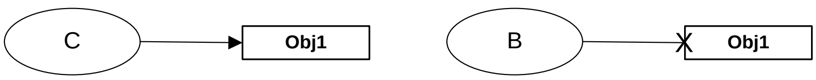



Figure 4.11: Impact model. Update

Furthermore, two additional types of arrows are used to express the creation and deletion of objects, as illustrated in Figure 4.9(a) and 4.9(d) respectively. These Impacts are presented via the following figures. Figure 4.12(a) demonstrates that the object Obj1 is created by the the *Task C*, whereas, Figure 4.12(b) illustrates that the *Obj1* is destroyed (deleted) by a *Task B*. It is significant to note that objects must already exist (created) first to participate in any other type of dataflow.




(a) Impact model. Create (b) Impact model. Delete

Figure 4.12: Impact Model. Create/Delete Object

#### 4.3.1.4 Multiplicity and Optionality of Impacts

The proposed $\mu$ML language adopts the *UML* style for expressing cardinality to represent the number of elements involved in such a relationship or flow. It generally expresses the statement: *at least m but no more than n objects*, in the *UML* multiplicity adornment as $(m..n)$. The notion of *upper bound* and *lower bound* are also exists to specify the range of elements. Each bound might be an exact positive integer number or unlimited number of elements, denoted by asterisk symbol $(*)$. The following table (4.3) exemplifies multiplicity:

| Multiplicity | Meaning |
|---|---|
| 0..1 | Optional (no instances or one instance) |
| 0..* (or *) | Zero or more instances |
| 1..* | At least one instance |
| 2 (or 2..2) | Exactly two instances |

Table 4.3: Examples of multiplicity

As a default interpretation of the model, any *Impact* represents a one-to-one correspondence. This means that one instance of each datum, modelled as an *Impact Object*, read or written for each execution of the task. For simplicity, there is no need to attach (*1..1*) multiplicity for each flow unless the impact correspondence is not one-to-one.

#### 4.3.1.5 Disjoint Impact Combinator (Alternative)

The disjoint flow combinator joins two or more optional flows, where these alternate exclusively. This *exclusive alternation* concept is shown in the *Impact Model* by placing a clear circle symbol between a number of flows. Figure 4.13 below illustrates this.

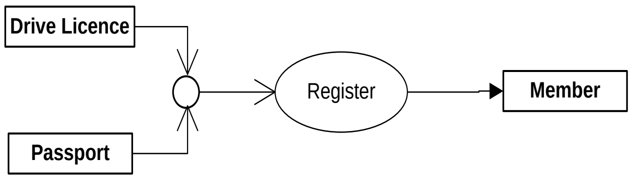



Figure 4.13: Impact model. Disjoint Impact Combinator

In the foundation of the *Impact Model*, a unary predicate (*ImpConjunction*) is defined to determine *Nodes* that are disjoint flow combinators in the model. Following the similar way of declaring *Impact Task* and *Object* according to the $\mu$ML metamodel, Disjoint flow combinator is also considered a subtype of the *Node* element ($ImpConjunction <: Node$). Therefore, it is identified using a unique identifier. This is a theorem that follows from the earlier law 9, by substituting for *Node*.

$$\begin{aligned} &\forall\, m : ImpactModel,\ \forall\, c : ImpConjunction \bullet (c \in_m m),\\ &\qquad \exists!\, id : Identifier \bullet Id(c, m, id) \end{aligned}$$

Because disjoint flow combinators link some flows in *ImpactModel*, it can be either a source or a target similar to the *ImpTask*, any *ImpObject*. This is a theorem that follows from the earlier law 19, by substituting for *Node*.

$$\begin{aligned} &\forall\, c : ImpConjunction,\ \forall\, m : ImpModel \bullet (c \in_m m)\\ &\qquad \longrightarrow\\ &\qquad\quad \exists\, f \bullet Flow(f) \bullet (f \in_m m)\\ &\qquad\quad \wedge\ (Source(f, c) \oplus Target(f, c)) \end{aligned}$$

#### 4.3.1.6 Impact Boundary

Similar to the previously presented *Task Model*, a rectangular system boundary may be drawn around groups of tasks, to indicate that these tasks fall within the scope of the system to be developed, while other tasks fall outside. From that, the *Impact Model* might contain a number of logical *ImpBoundary* nodes: $\forall\ m : ImpactModel,\ \exists\ b : ImpBoundary\ \bullet\ (b\ \in_m\ m)$.

### 4.3.2 The Significance of Impact Model

The *Impact Model* aims to generate a partial ordering of tasks by realising data dependency captured via data participation in tasks. This contributes to predicting and constraining the possible orders of task execution in later design stages. The execution of a task may depend on the prior existence of certain data, and it may in turn enable other tasks to be performed due to the data it produces.

Therefore, the order of execution can in part be determined by tracing the *CRUD* effects of tasks on a single object or multiple objects that fall in the same business scope (system boundary). By considering each task in Figure 3.4, shown previously in Chapter 3, the creation of the *Loan* object, for instance, depends on the completion of reading the *Copy* and *Borrower* objects. Likewise, the deletion of *Loan* requires reading *Copy* first.

Furthermore, partial object life histories can also be determined by tracing the *CRUD* effects of tasks on single objects. Each creation and deletion event occurs only once, but other events may occur multiple times in any unspecified order. As a consequence, data dependency, the fundamental property of the *Data Model*, may be determined directly via analysis of the object lifetimes in the impact model. When more than one obejct is involved in the same task, with different types of *impacts*, this means that there exist dependency relationships between these objects, in which shorter-lived objects always depend on longer-lived ones.

Tracing the *CRUD* effects of a single task execution on individual objects (viz. a set of individuals) also provides useful information for the *Data Model* since knowing the relative extent (life expectancy) of each object also informs the notion of data dependency. From this, we can assume that it is possible to predict target concepts to generate a partial *Data Model* to express the interdependency of the business entities.

## 4.4 Overview of the Information Model

The *Information Model* is the highest level, ***manually-constructed***, diagrammatic representation of the structure of business objects. The main purpose of the *Information Model* is to capture sufficient information on business data to generate a complete *Data Model*. We are following the standard approach taken in software engineering in which the model is presented in terms of conceptual entities (objects), attributes and inter-relationships. As a basic inspiration, we view conceptual modelling as a separate activity from database design, performed by business analysts or users in a very abstract way.

*Entities* in the model represent physical objects and documents or logical data used by the business that may, or may not, be in normal form. Any object that bears useful information may be modelled, including those storing logical information about stakeholders or other physical items. It is common for these kinds of objects to require further logical decomposition, especially if they contain repeating groups of data.

An entity consists of properties, modelled as attributes, which describe that entity (object). It is important for a business-user who is a domain expert to test the properties of each object in order to ensure that they are atomic and that their values depend fully on the object instance owning them. For each object instance, the attribute may take on a distinct value.

Apart from this, in the real-world, business objects relate and interact to present the domain of the business. The *Information Model* represents this via undirected conceptual associations, which link objects together, indicating that the related objects are acquainted in some fashion. Additionally, relationships that require grouping sets of objects that are treated together or share common properties, such as composition and generalization, respectively, can be expressed as other types of association to imply further semantics.

The various relationships provide clean and precise semantics about the notation of the model elements for the business-user. They show details, such as multiplicity constraints, which hide technically significant information to determine data dependency, using our promoting and qualifying policies until relationships are all of the *many-to-one* kind, with the many always depending on the one.

The *Information Model*, in the proposed metamodelling hierarchy, is considered a kind of $Model$. As a consequence, concepts of the Information Model can be defined using FOPL and relate to the corresponding $\mu$ML metamodel concepts.

**Law 22.** The *InformationModel* is a kind of *Model*.

$$(InformationModel \ <: \ Model)$$
$$\forall\ m \ \bullet \ InformationModel(m) \longrightarrow Model(m)$$

### 4.4.1 Notation and Semantics of the Information Model

In this section, each concept that appears in the *Information Model* is introduced with some details about its usage and how it is visualised graphically in the model. The following table (Table 4) summaries all predicates used in describing $\mu$ML Information Model.

In order to motivate the list of policies that specifies the Information Model, two basic laws must be identified first as follows:

**Law 23.** Any node in the *InformationModel* is an object.

$$(InfEntity \ <: \ Node)$$
$$\forall\ n : Node,\ \forall\ m : InformationModel \ \bullet \ (n \ \in_m \ m) \longrightarrow InfEntity(n)$$

| Predicate | Meaning | Syntactic Suger |
|---|---|---|
| *InformationModel(x)* | $x$ is an Information Model | *x:InformationModel* |
| *InfEntity(x)* | $x$ is an Information Object (Entity) | *x:InfEntity* |
| *InfAttribute(x)* | $x$ is an Information Attribute | *x:InfAttribute* |
| *Relationship(x)* | $x$ is a Relationship | *x:Relationship* |
| *Association(x)* | $x$ is an Association | *x:Association* |
| *Generalisation(x)* | $x$ is a Generalisation | *x:Generalisation* |
| *Parent(x)* | $x$ is a Parent | |
| *Child(x)* | $x$ is a Child | |
| *Disjoint(x)* | $x$ is Disjoint | |
| *Composition(x)* | $x$ is a Composition | *x:Composition* |
| *Whole(x)* | $x$ is a Whole | |
| *Part(x)* | $x$ is a Part | |
| *Total(x)* | $x$ is a Total Composition | |
| *InfRole(x)* | $x$ is an end-role | |

Table 4.4: Predicates for $\mu$ML Information Model

**Law 24.** Any edge in the *Information Model* is a relationship.

$$
\begin{aligned}
&(\mathit{Relationship} \ <: \ \mathit{Edge}) \\
&\forall\, a : \mathit{Edge},\ \forall\, m : \mathit{InformationModel} \\
&\quad \bullet\ (a \ \in_m \ m) \\
&\quad \longrightarrow \\
&\quad \mathit{Relationship}(a)
\end{aligned}
$$

**Law 25.** Any relationship in the *Information Model* can only be one out of: a generalisation or composition or association.

$$
\begin{aligned}
&(\mathit{Generalisation} \ <: \ \mathit{Relationship} \ \wedge \\
&\ \mathit{Composition} \ <: \ \mathit{Relationship} \ \wedge \\
&\ \mathit{Association} \ <: \ \mathit{Relationship}) \\
&\forall\, r : \mathit{Relationship},\ \forall\, m : \mathit{InformationModel} \\
&\quad \bullet\ (r \ \in_m \ m) \\
&\quad \longrightarrow \\
&\quad \mathit{Generalisation}(r) \ \oplus \ \mathit{Association}(r) \ \oplus \ \mathit{Composition}(r)
\end{aligned}
$$

#### 4.4.1.1 Object

Similar to the object's notation used in the *Impact model*, the UML rectangular nodes, taken from the UML class diagram are adopted to present information objects in the model (Figure 4.14). The information objects correspond to physical business entites and documents, or logical data used by the business. Any object that bears useful information may be modelled. This may include objects storing logical details about human actors, or other physical things.

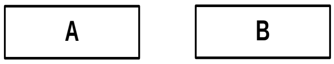



Figure 4.14: Information model. Objects (Entities)

It is worth saying, physical objects are often more complex than their logical counterparts, because they may contain repeating groups of data that should eventually be split. According to the $\mu$ML metamodel, *Information Object* is also considered a subtype of the *Node* element ($InfEntity$ <: $Node$). Therefore, it is identified using a unique identifier. This can be defined in FOPL as:

$$\forall\ m : InfModel,\ \forall\ obj : InfEntity\ \bullet\ (obj\ \in_m\ m),\\ \exists!\ id : Identifier\ \bullet\ Id(obj, m, id)$$

Each object in the model might involved in one or more than one relationship with other objects. This can be expressed formally as:

$$\forall\ m : InformationModel,\ \forall\ r : Relationship,\\ \exists\ obj1, obj2 : InfEntity\ \bullet\ ((obj1, obj2, r\ \in_m\ m)\\ \longrightarrow\\ (Connects(r, obj1, obj2)\ \vee\ Connects(r, obj2, obj1)))$$

#### 4.4.1.2 Association

*Associations* express conceptual "has a" relationships between objects, between objects of any class, expressed formally as *Association* <: *Relationship*. The UML straight lines are used to represent the links, that are usually bounded by events that involve the objects concerned. These relationships are undirected, meaning that the dependency of one object upon another is not yet known or understood. Figure 4.15 below, A is an object associated with another object B. *Associations* may optionally be named.

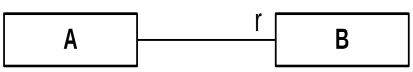



Figure 4.15: Impact model. Associations

The object might involved in one or more than one association with other objects, or it might refer to itself (self-relationship). This theorem follows from Law 13 by substituting for *Model, Node* and *Relationship*:

$$\forall\ m : InformationModel,\ \forall\ r : Association,\\ \exists\ obj1, obj2 : InfEntity\ \bullet\ ((obj1, obj2, r\ \in_m\ m)\\ \longrightarrow\\ (Connects(r, obj1, obj2)\ \vee\ Connects(r, obj2, obj1)))$$

#### 4.4.1.3 Composition

*Composition* is a conceptual whole/parts relationship, or "is part of", in the domain, expressed formally as: *Composition*
<: *Relationship*. It groups sets of objects that are treated together, indicating data dependency of the whole on the parts, which is assumed by default. The parts may exist independently of the whole. *Composition* is shown using the white UML diamond arrowhead notation, with the arrow pointing to the whole concept (Figure 4.16). The notion of composition is a primary concept in compositional systems. We therefore prefer this term to the alternative term aggregation. UML uses composition to denote a specific flavour that we call total composition (section 4.4.1.4 below).

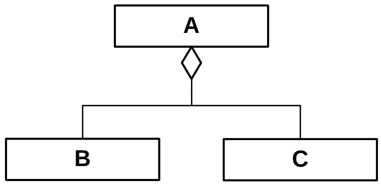



Figure 4.16: Information model. Composition

The following policy defines the concepts of *Composition* in the *Information Model* and the notion of its *Parts* and *Whole*.

**Law 26.** Any compostion, in the Information Model connects *Part* objects to the *Whole* one. This can be expressed in FOPL as:

$$\begin{aligned}&\forall\ m : InformationModel,\ \forall\ c : Composition,\ \forall\ obj1, obj2 : InfEntity\\ &\qquad \bullet\ ((c, obj1, obj2\ \in_m\ m)\ \wedge\ Connects(c, obj2, obj1))\\ &\qquad\qquad \longrightarrow\\ &\qquad\qquad\quad Whole(obj1)\ \wedge\ Part(obj2)\end{aligned}$$

#### 4.4.1.4 Total Composition

A total composition is one in which the parts belong entirely to the whole, in the sense that they cannot exist independently without the whole. Total compositions are significant, because the direction of data dependency is reversed: the parts depend on the whole. For instance, deletion of the whole must result in a cascading deletion of the parts. This is shown by placing a filled circle1 over the composition, to indicate total ownership of the parts. The following figure (4.17) indicates that $A$ is composed of $B$ and $C$, which cannot exist without the $A$. It can be declared using Law 28 above, as *Total* is irrelevant to deducing the whole and part.

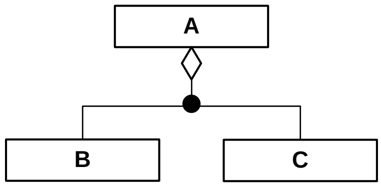



Figure 4.17: Information model. Total Composition

#### 4.4.1.5 Generalisation

*Generalisation* is a conceptual general/specific, or "is a", relationship in the domain. It groups sets of objects that share some common properties, indicating data dependency of the specific on the general. *Generalisation* is expressed using the UML triangle arrowhead notation, with the arrow pointing towards the more general concept. The following figure (4.18) illustrates that $B$ and $C$ are specialised cases of $A$.

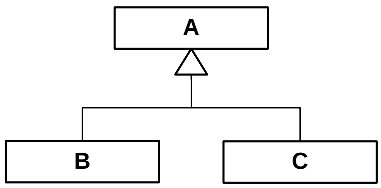



Figure 4.18: Information model. Generalisation

**Law 27.** In the *InformationModel*, each *Generalisation* Connects *Child* objects to a *Parent* one. This can be formally expressed as:

$$\forall\ m : InformationModel,\ \forall\ g : Generalisation,\ \forall\ obj1, obj2 : InfEntity \\ \bullet\ ((f, obj1, obj2\ \in_m\ m)\ \wedge\ Connects(g, obj2, obj1)) \\ \longrightarrow \\ Parent(obj1)\ \wedge\ Child(obj2))$$

#### 4.4.1.6 Disjoint Generalisation

A *disjoint* generalisation is one in which the specific concepts are mutually exclusive, in the sense that no object from the domain could be an instance of both concepts simultaneously. This is shown by placing a clear circle over the generalisation, to indicate that the specialisations are disjoint. Figure (4.19) indicates that $A$ may be either $B$, or $C$, but not both in the given domain.

Using the same law for defining the (overlapping) generalisation (law 29), the disjoint generalisation also connects *Child* objects to a *Parent* one.

$$\forall\ m : InformationModel,\ \forall\ g : Generalisation,\ \forall\ obj1, obj2 : InfEntity \\ \bullet\ ((f, obj1, obj2\ \in_m\ m)\ \wedge\ Disjoint(g) \wedge\ Connects(g, obj2, obj1)) \\ \longrightarrow \\ Parent(obj1)\ \wedge\ Child(obj2))$$

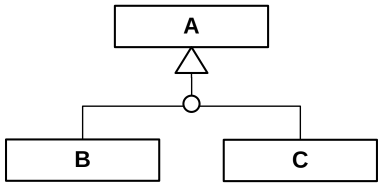



Figure 4.19: Information model. Generalisation

#### 4.4.1.7 Attribute

*Attributes* represents the atomic properties of entities (objects), in which each entity has one or more attribute ($\forall\ obj : InfEntity,\ \exists a : Attribute \bullet (a \in_p obj)$). Similar to the UML class diagram, they are visually located in the usual drop-down partition of the Information Model obejcts the following Figure (4.20) illustrates the representation of properties, including the name and the data type, for two information entities: *A* and *B*.

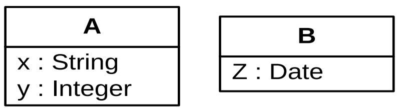



Figure 4.20: Information model. Attributes

#### 4.4.1.8 Multiplicity

In the *Information Model*, objects are related to each other with a given multiplicity. For each instance of object at one end of the association, there exist zero, one or more instances of the second object at the opposite end. The usual UML multiplicity adornments, examplified previously in table 4.3, are adopted to mark various kinds of multiplicity, namely, *one-to-many*, *zero-to-many*, *many-to-many*, and optional.

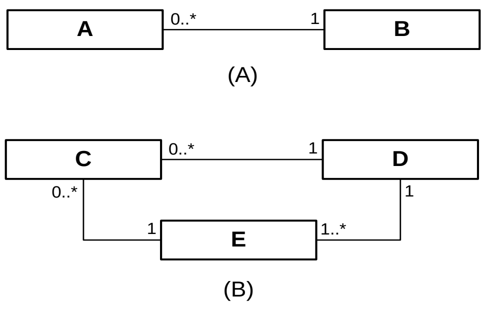



Figure 4.21: Information model. Multiplicity

Figure 4.21 (A) indicates that there are *zero-to-many* A(s) for each entity B. However, binary associations in Figure 4.21 (B) represents the decomposition of a ternary or higher-arity association between three entities: C, D, and E. Each N-arity association must be broken down into a set of binary associations in order to be analysed in a later stage.

### 4.4.2 The Significance of The Information Model

- Associations are analysed later to reveal data dependencies, using rules based on multiplicity. This analysis is later used to normalise tables ready for database implementation.
- Properties (Attributes) are captured for each entity. The attributes later become the columns of database tables. Some attributes may be marked as uniquely identifying the owning entity.
- In circumstances where it is impossible to find a unique identifier for an entity, this may be manufactured automatically.
- The Information Model may still capture atomic data at a high level of abstraction, for example, using types such as Currency, Text, Time, which may need translation in later stages.

## 4.5 Outlook on the Chapter

This chapter discussed, in-depth, the concepts, adopted graphical notations, and formal semantics for each Micro Modelling Language ($\mu$ML) model appearing in the *Requirement Sketching* phase. It covered all critical elements, their usage, and their all possible interpretations within the various contexts of the $\mu$ML Task, Impact and Information Model.

Additionally, the chapter discussed how the selected system views contribute to the main goals of the thesis. More specifically, it showed how the proposed user-friendly modelling language, which has simpler and cleaner semantics than current UML-based approaches, captured the most critical aspects of the system using business user knowledge. Moreover, the chapter discussed how the core elements of this $\mu$ML notation succeed in raising the level of abstraction, compared to UML, at which business-users may express their model specifications.

The semantics and notation for each model and element are compared to similar or closely related UML concepts and notation, showing the simplicity and clarity of the proposed language. A number of unary and n-ary predicates were introduced for defining concepts and their interrelationships within each model in respect of the previously defined metamodel.

# 5

# $\mu$ML Concepts and Notations

*In the Analysis Phase*

"The Analyst wants to get at the simplest form of the system which has the features they are interested in"

Analysis Wisdom - ultradark

## 5.1 Context

This chapter describes in detail the Micro Modelling Language models used in the Analysis Phase, namely, Data (Dependency), DataFlow (initial and detailed) and (Screen) State Model. For each model, the chapter presents how the model satisfies its purpose within the framework. The chapter also discusses each model concept and its corresponding graphical notation, defining the concept formally using FOPL.

## 5.2 Overview of the Data Dependency Model

The *Data Dependency Model* is an ***automatically-generated*** intermediate level diagrammatic representation that is intended to describe the logical data of the system and support the development to a point where a logical database schema may be generated. The model consists of a number of logical objects linked by a number of dependency relationships, representing the direction of data dependency. This also determines which additional attributes will be used for references between objects and handled in the next stage of the layered transformation to generate the *Database Schema Model*. In other words, this determines where the forthcoming *foreign keys* will be located. The following figure (Figure 5.1) demonstrates the visual representation of the *Data Model* elements.

The *entities* represent logical, rather than physical, data used by the business in the real-world from which all repeating groups have been isolated. Any entity (object) bearing useful logical information, including stakeholders or other business items, is modelled. It contains

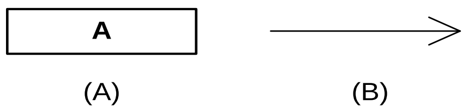



Figure 5.1: Data model. (A) Entity, (B) Dependency

attributes corresponding to the named columns of tables in a relational database that will be generated in the next stage of transformation.

*Attributes* are the atomic properties of objects. They are wholly functionally dependent on their owning object for their value; for each object instance, the attribute may possibly take on a distinct value. Only simple properties that are not further decomposable are modelled as attributes. Decomposable properties must be modelled as distinct objects instead, unless the target database can treat them in the same way as a basic type. At a higher level, the business-user specifies one or more attributes that eventually identify the object uniquely. This is taken for granted in the generated *Data Model*. Otherwise, if the identifier cannot be found, an artificial identifier is created.

In the current version of BUILD, a complete *Data Dependency Model* is constructed automatically from a multiplicity analysis of conceptual associations in the pre-designed *Information Model*. All associations are either promoted or translated directly into directed dependencies in which the many-sided object depends on the one-sided one. In worth mentioning that the *Data Dependency Model* may be, alternatively, constructed partially from an impact analysis of events in the *Impact Model*, an object being created from a pre-existing one. Either or both of these prior business user predefined models may be used as a source for the data model and may be crosschecked for consistency.

The *Data Dependency Model*, in the proposed metamodelling hierarchy, is considered a kind of $Model$. Thus, conepts of the *Data Dependency Model* can be defined using FOPL and relate to the corresponding $\mu$ML metamodel concepts.

**Law 28.** The *DataModel* is a kind of *Model*.

$$
\begin{aligned}
&DataModel \ <: \ Model \\
&\forall\ m \ \bullet \ DataModel(m) \\
&\qquad\qquad\qquad\qquad \longrightarrow \\
&\qquad\qquad\qquad\qquad\quad Model(m)
\end{aligned}
$$

### 5.2.1 Notation and Semantics of the Data Model

In this section, each concept that appears in the *Data Model* is discussed in detail. This includes its usage and how it is visualised graphically in the model. It is worth emphasising that a number of UML notations in the *Class Diagram* are adopted and reused with slight different meaning from its original semantics. For instance, *underlining attributes* in the UML Class Diagram is used to express *static attributes*. The meaning of *underlining attributes* is changed here to express *primary keys* The following table (5.1) summaries all predicates used in describing $\mu$ML Data Dependency Model.

| Predicate | Meaning | Syntactic Sugar |
|---|---|---|
| *DataModel(x)* | $x$ is a Data Model | *x:DataModel* |
| *DEntity(x)* | $x$ is a Data Model Entity | *x:DEntity* |
| *DAttribute(x)* | $x$ is a Data Model Attribute | *x:DAttribute* |
| *Dependency(x)* | $x$ is a Dependency | *x:Dependency* |
| *DependsOn(x, y)* | $x$ depends on $y$ | |
| *DRole(x)* | $x$ is a Data Model Role | *x:DRole* |

Table 5.1: Predicates for $\mu$ML Data Dependency Model

In order to motivate some policies that specifies *Data (Dependency) Model*, basic declarations of core elements must be identified first. According to the $\mu$ML metamodel hierarchy where *Data Model* entity is a subtype of *Node* and *Dependency* is a subtype of *Arc*, expressed as: ($DEntity\ <:\ Node\ \wedge\ Dependency\ <:\ Arc$), then any *Node* and *Arc* in the *Data Model* represents an entity and a dependency respectively. This can be expressed formally as:

$$\begin{aligned} &\forall\ n : Node,\ \forall\ a : Arc, m : DataModel \\ &\quad \bullet\ (n, a\ \in_m\ m) \\ &\quad\quad \longrightarrow \\ &\quad\quad\quad DEntity(n) \wedge\ Dependency(a) \end{aligned}$$

#### 5.2.1.1 Entity and Attributes

Similar to Information Model objects and their properties, entities in the Data Model are drawn as rectangular nodes and their attributes are treated likewise in the previously described model, in which each entity has some attributes $\forall\ e : DEntity,\ \exists\ a : DAttribute\ \bullet\ (a\ \in_p\ e)$. The representation of Entity and Attribute are adopted from the well-known UML class diagram. This can be shown in Figure 5.1 (A) above, Figures 5.3 and 5.4 below.

#### 5.2.1.2 Dependency

As illustrated in Figure 5.1 (B) above, dependencies in the Data Model are drawn as a directed edge, with an open arrowhead pointing toward the target from the dependent source on which it depends. Figure (5.2) demonstrates that the object A is dependent upon the object B, which can be formalised in FOPL as:

$$\begin{aligned} &(DEntity\ <:\ Node\ \wedge\ Dependency\ <:\ Arc) \\ &\forall\ m : DataModel, d : Dependency, (e1, e2 : DEntity) \\ &\quad \bullet\ ((d, e1, e2\ \in_m\ m)\ \wedge\ Connects(d, e1, e2)) \\ &\quad\quad \longrightarrow \\ &\quad\quad\quad DependsOn(e1, e2) \end{aligned}$$

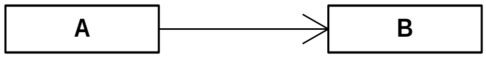



Figure 5.2: Generated Data model. Dependencies

#### 5.2.1.3 Identifier

In the Data Model, each object (entity) must have at least a single attribute that identifies it uniquely, or several attributes, whose values taken together uniquely identify the object.

$$\begin{aligned} (Identifier\ <:\ DAttribute) \\ \forall\ m : DataModel,\ \forall\ e : DEntity, \\ \exists!\ id : Identifier\ \bullet\ ((Id(obj, m, id))\ \wedge\ id\ \in_p\ e) \end{aligned}$$

Under all circumstances, if no identifier can be found in such an entity, an artificial identifier is created. In the model, identifiers are shown by underlining the attributes concerned.

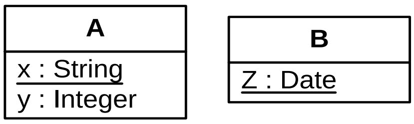



Figure 5.3: Generated Data model. Primary Key

This contrasts with the usual meaning of underlining in UML notation, which denotes a static (shared or class) attribute. From a data modelling perspective, a static attribute is not wholly functionally dependent on its owning entity, so logically should belong to a separate single entity, representing the constants of the class, on which the current entity depends by association.

#### 5.2.1.4 References

For each dependency between one object and another, certain additional attributes must be chosen for the source object to uniquely refer to the target object. The references of an object correspond exactly to the identifiers of the objects to which they refer by association. The references are simple, if the identifier of the target is simple, and compound if the identifier is compound. From that, it can be said that the structure of each *reference* is identical to the structure of the *identity* of the obejct it refers to.

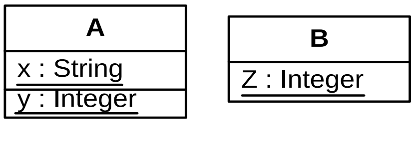



Figure 5.4: Generated Data model. Foreign Key

References are implicit in the dependency structure, but may optionally be visually shown in a second drop-down box (Figure 5.4). This replaces the use of the second drop-down box in the UML class diagram, which represents the methods of a class.

## 5.3 Overview of the DataFlow Models

The *DataFlow Model* is an ***automatically-generated*** intermediate level diagrammatic representation that that shows how data flows between the internal tasks and data stores, and external entities at a particular level of abstraction. The model consists of a number of entities and participants (agents) that are connected to some tasks via various flows. It provides a detailed description of the business tasks and aggregated subtasks that s perform operations on data in terms of data flow.

Unlike the *Impact Model*, which demonstrates how a single object is impacted by tasks, the *DataFlow Entity* represents a collection of objects that are directly matched to a formalised entity appearing in the *Data Model*. This allows the representation of crosschecked instances and attributes that are carried on the flows linked to an entity.

In addition, data on flows is expressed using formal expressions in terms of attributes, instances of entities (objects), local variables and values. This textual notation also represents filtering, projecting and assigning values when read from or written to an entity. It is worth saying that the design of the textual statements considers the business-users capability to express the data flows and related operations without any difficulty. As a result, the produced *DataFlow Model* holds readable data with more accurate interpretations of the flows.

Much like the previously introduced *Impact Model*, it does not describe the time-order of processing but rather what data passes between tasks, users and entities. A partial graph of flows between the system components of each part of the system is automatically derived by merging concepts from the matched boundaries that appear in the pre-constructed *Task* and *Impact* models. This produces a number of artifacts that are divided into boundaries equivalent to those existing in the requirement models, forming an initial *DataFlow Model*. It is worth mentioning that flows between tasks are not allowed, at this level of abstraction, in which there is insufficient information about task ordering that prevents predicting these additional flows.

The initial *DataFlow Model* is considered to be a top level partial model with no precise information about the data on flows. Business users, who are aware of the kinds of data transferred between the system components, are responsible for specifying the data and expressing basic operations meant by flows, using restricted statements recognized by the model. Using this external information enables the complete evolution of the detailed *DataFlow Model*. Figure 3.7, Chapter 3, demonstrates the contents of DataFlow Model, including, a *task, flow, entity* and *boundary* concept.

The detailed *DataFlow Model* is considered to be a bottom level comprehensive model that consists of a number of boundaries of decomposed tasks (atomic tasks) based upon their atomic actions. Each subtask, which is also a task, has only one associated flow and one atomic action. This new collection of boundaries forms the final *DataFlow Model* that contains comprehensive information regarding data flows and related operations. Figure 9.21, Chapter 9, illustrates a detailed DFD model including its concepts similar to the initial DFD model.

The *DataFlow Model*, in the proposed metamodelling hierarchy, is considered a kind of *Model*. Thus, concepts of the *DataFlow Model* can be defined using FOPL and relate to the corresponding $\mu$ML metamodel concepts.

**Law 29.** The *DataFlowModel* is a kind of *Model*.

$$\begin{array}{l} DataFlowModel \; <: \; Model \\ \forall \, m \; \bullet \; DataFlowModel(m) \\ \qquad\qquad\qquad\qquad \longrightarrow \\ \qquad\qquad\qquad\qquad\quad Model(m) \end{array}$$

### 5.3.1 Notation and Semantics of the DataFlow Model

In this section, each concept that appears in the *DataFlow Model* is introduced with some details about its usage and how it is visualised graphically in the model. It is worth mentioning that the notation introduced in the previously defined *Task, Impact* and *Information Model* are resused in the *DataFlow* to indicate slightly different meanings. The following table (5.2) summaries all predicates used in describing $\mu$ML DataFlow Model.

| **Predicate** | **Meaning** | **Syntactic Suger** |
|---|---|---|
| *DataFlowModel(x)* | $x$ is a DataFlowModel | *x:DataFlowModel* |
| *DfBoundary(x)* | $x$ is a boundary | *x:DfBoundary* |
| *DfTask(x)* | $x$ is a business task | *x:DfTask* |
| *DfActor(x)* | $x$ is an actor | *x:DfActor* |
| *DfEntity(x)* | $x$ is an entity | *x:DfEntity* |
| *Instance(x)* | $x$ is an instance of an entity | *x:Instance* |
| *Datum(x)* | $x$ is a piece of data | *x:Datum* |
| *DfInputFlow(x)* | $x$ is an input flow | *x:DfInput* |
| *DfOutputFlow(x)* | $x$ is a output flow | *x:DfOutput* |
| *DfCreateFlow(x)* | $x$ is an create flow | *x:DfCreate* |
| *DfReadFlow(x)* | $x$ is a read flow | *x:DfRead* |
| *DfUpdate(x)* | $x$ is an update flow | *x:DfUpdate* |
| *DfDelete(x)* | $x$ is a delete flow | *x:DfDelete* |
| *DfWriteFlow(x)* | $x$ is an write flow | *x:DfWriteFlow* |
| *DfRole(x)* | $x$ is an end-role | *x:DfRole* |
| *InstanceOf(x, y)* | $y$ is an instance of $x$ | |
| *FlowTypeOf(x, y)* | $y$ has flow type of $x$ | |
| *CreatedBy(x, y)* | $y$ is created by $x$ | |

| | |
|---|---|
| *DeletedBy(x, y)* | $y$ is deleted by $x$ |
| *ReadBy(x, y)* | $y$ is read by $x$ |
| *UpdatedBy(x, y)* | $y$ is updated by $x$ |
| *Assigns(x, y, z)* | $x$ assigns $y$ value to $z$ |
| *Transmits(x, y)* | $x$ transmits datum $y$ |
| *AtomicTaskOf(x, y)* | $y$ is an atomic subtask of $x$ |

Table 5.2: Predicates for $\mu$ML DataFlow Model

In order to motivate the list of policies that describes DataFlow Model, two basic laws must be declared first as follows:

**Law 30.** Using the notion of subtyping in our FOPL approach, any *Node* in the *DataFlowModel* can be either a *DfTask* , a *DfEntity* or a *DfActor*.

$$(DfActor <: Node) \wedge (DfTask <: Node) \wedge (DfEntity <: Node)$$

$$\forall\ n : Node, m : DataFlowModel \bullet (n \in_m m) \longrightarrow DfActor(n) \oplus DfTask(n) \oplus DfEntity(n)$$

**Law 31.** Using the notion of subtyping in our FOPL approach, any *Flow* in the *DataFlowModel* can be either *DfCreateFlow*, *DfReadFlow*, *DfUpdateFlow*, *DfDeleteFlow*, *DfWriteFlow*, *DfInputFlow*, or *DfOutputFlow*. In the following sections, each concept that appears in the *DataFlow Model* is introduced with some details about its usage and how it is visualised graphically in the model.

$$\begin{gathered}(Flow <: Arc) \wedge (DfInputFlow <: Flow) \wedge \\ (DfOutputFlow <: Flow) \wedge (DfCreateFlow <: Flow) \wedge \\ (DfWriteFlow <: Flow) \wedge (DfReadFlow <: Flow) \wedge \\ (DfUpdateFlow <: Flow) \wedge (DfDeleteFlow <: Flow)\end{gathered}$$

$$\forall\ f : Flow, m : DataFlowModel \bullet (f \in_m m) \longrightarrow DfCreateFlow(f) \oplus DfReadFlow(f) \oplus DfUpdateFlow(f) \oplus DfDeleteFlow(f) \oplus DfWriteFlow(f) \oplus DfInputFlow(f) \oplus DfOutputFlow(f)$$

#### 5.3.1.1 Task

Similar to the notation adopted for tasks in the *Task* and *Impact model, DataFlow Tasks* are depicted using the ellipse node. In this model, the emphasis is on the actual data produced/consumed by business tasks from/to system entities at a given level of granularity. The external data usage and production with actors or external systems are also depicted. It is worth mentioning that during the development process, the decomposed DataFlow Tasks are also presented in the detailed DFD using the same notation (ellipse node). Figure 5.5 below depicts the two Data Flow tasks, A and B.

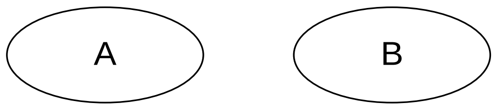



Figure 5.5: Generated DataFlow model. Tasks

According to the previously defined $\mu$ML metamodel hierarchy ($DfTask$ <: $Node$), each dataflow task has only one unique identifier. This can be expressed using FOPL predicate as: $\forall\ m : DataFlow,\ \forall\ t : DfTask\ \bullet\ (t\ \in_m\ m),\ \exists!\ i : Identifier\ \bullet\ Id(t, m, i)$

#### 5.3.1.2 Entity

Similar to the notation for objects in the *Impact, Information* and *Data model*, the UML rectangular nodes ar used to depict *DataFlow Entity*. The *Entity* is totally derived from the *Impact Model* and might be cross-checked with the *Data/Information Models*. In this model, it refers to a collection of objects, or a type, which is a distinct meaning compared to its meaning in other models. Figure 5.6 below depicts two Data Flow entities A and B.


Figure 5.6: Generated DataFlow model. Entites

A *DataFlow Entity* is considered subtype of *Node* in the $\mu$ML metamodel hierarchy ($DfEntity$ <: $Node$), therefore, it has only one unique identifier $\forall\ m : DataFlow,\ \forall\ e : DfEntity\ \bullet\ (e\ \in_m\ m),\ \exists!\ i : Identifier\ \bullet\ Id(e, m, i)$. They replace the notion of data sources in the traditional DFD, in which each entity consists of a number of instances $\forall\ e : DfEntity,\ \exists\ x : Instance\ \bullet\ (InstanceOf(e, x))$. It worth mentioning that entities in the *DataFlow Model* have no attributes shown in the diagram, but theythey are presumed to correspond to identically-named objects (with attributes) that appear in other $\mu$ML models.

#### 5.3.1.3 Actor

*Actors* in the *DataFlow Model* are totally derived from *Participant* concept in the *Task Model*. It is used to represent the same concept here. Therefore, to maintain the consistency between models, an *Actor* is presented in the DFD using the same notation as the *Task Model Participant*, that is, a human stick figure, or a 3D box to represent a human, or system actor, respectively $\forall\ a : DfActor\ \longrightarrow\ Human(a)\ \oplus\ ExSystem(a)$. A stick figure node is used to represent a human actor, as it seen in Figure 5.7 (left), whereas external systems are depicted as box nodes, taken from the UML deployment diagram, Figure 5.7 (right).

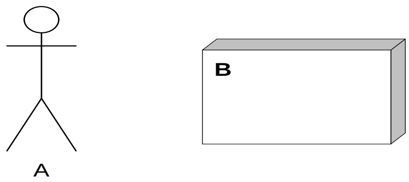
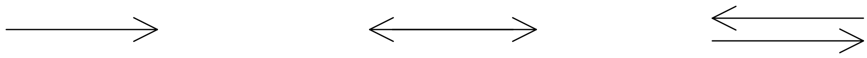



Figure 5.7: Generated DataFlow model. Actors

Each *DfActor*, in *DataFlow Model*, has only one unique identifier $\forall\ m : DataFlow,\ \forall a : DfActor \bullet (a \in_m m),\ \exists!\ i : Identifier \bullet Id(a, m, i)$.

#### 5.3.1.4 Data Flows

*Data flows* describe the movement of information (data drift) within a system, between its *Tasks, Actors* and *Entities*. The direction of such a flow is indicated by an arrow-head at one end (unidirectional) to represent (*Read, Write, Create, Destroy (Delete), Input* and *Output*), or at both ends (bi-directional) to represent (*Update*) at the initial *DataFlow Diagram*.



(a) Unidirectional flow

(b) Bidirectional flow

(c) Bidirectional flow (breakdown)

Figure 5.8: DataFlow model. Flows

Figure 5.8 (a,b) above illustrates the notation of the unidirectional and the bi-directional flow respectively, whereas (c) shows the way to express bi-directional update flows at the detailed *DataFlow Diagram*.

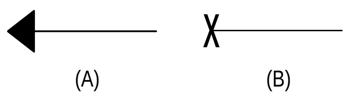



Figure 5.9: Generated DataFlow model. Create and Delete Flow

Figure 5.9 above shows the notation of two types unidirectional flows that are used in the DFD model to express *Create* (left) and *Delete* (right) flow.

**Law 32.** Any flow, in the *DataFlow Model*, that connects an actor to a task is an input flow, whereas any flow that connects a task to an actor is an output flow. It can be expressed as:

$$\begin{aligned}
&\forall\ m : DataFlowModel, f : Flow,\\
&a : DfActor, t : DfTask \ \bullet\ (a, t, f\ \in_m\ m)\\
&\wedge\ Connects(f, a, t) \longrightarrow\\
&\quad DfInputFlow(f)\\
&\wedge\ Connects(f, t, a)\\
&\longrightarrow\\
&\quad DfOutputFlow(f)
\end{aligned}$$

**Law 33.** Any flow, in the *DataFlow Model*, that connects a task to an entity can be either a create, a write, or a delete flow. On the other hand, any flow that connects an entity to a task is a read flow. This can be expressed in FOPL as:

$$\begin{aligned}
&\forall\ m : DataFlowModel, f : Flow, e : DfEntity,\\
&t : DfTask \ \bullet\ ((e, t, f\ \in_m\ m)\ \wedge\\
&Connects(f, e, t)\\
&\longrightarrow\\
&\quad DfCreateFlow(f)\ \oplus\ DfDeleteFlow(f)\ \oplus\ DfWriteFlow(f)\\
&\wedge\ Connects(f, t, e)\\
&\longrightarrow\\
&\quad DfReadFlow(f)
\end{aligned}$$

**Law 34.** In the *DataFlowModel*, each *Flow* links two distinct Nodes.

$$\begin{aligned}
&\forall\ m : DataFlowModel, f : Flow \ \bullet\ (f\ \in_m\ m)\\
&\longrightarrow\\
&\quad \exists\ (n1, n2 : Node)\ \bullet\ Connects(f, n1, n2)\\
&\quad \wedge\ (n1\ \neq\ n2)
\end{aligned}$$

**Law 35.** Any *Flow* in the *DataFlowModel* transmits some data.

$$\begin{aligned}
&\forall\ m : DataFlowModel, f : Flow \ \bullet\ (f\ \in_m\ m)\\
&\longrightarrow\\
&\quad \exists\ d1 : Datum\ \bullet\ (Transmits(f, d1))
\end{aligned}$$

**Law 36.** Every *Node* in the *DataFlowModel* is involved in at least one *Flow* as a *Source* or a *Target*.

$$\begin{aligned}
&\forall\ n : Node, m : DataFlowModel\ \bullet\ (n\ \in_m\ m)\\
&\longrightarrow\\
&\quad \exists\ f\ \bullet\ Flow(f)\ \bullet\ (f\ \in_m\ m)\ \wedge\ (Source(f, n)\ \oplus\ Target(f, n))
\end{aligned}$$

**Law 37.** Every *DfTask*, *DfEntity* and *DfActor* in the *DataFlowModel* can be either a *Source* or a *Target*.

$$\forall\, t : DfTask, m : DataFlowModel \bullet (t \in_m m) \longrightarrow \exists\, f \bullet Flow(f) \wedge (Source(f,t) \oplus Target(f,t))$$

$$\forall\, e : DfEntity, m : DataFlowModel \bullet (e \in_m m) \longrightarrow \exists\, f \bullet Flow(f) \wedge (Source(f,e) \oplus Target(f,e))$$

$$\forall\, ac : DfActor, m : DataFlowModel \bullet (ac \in_m m) \longrightarrow \exists\, f \bullet Flow(f) \wedge (Source(f,ac) \oplus Target(f,ac))$$

**Law 38.** Each create flow creates an instance of a target entity

$$\forall\, m : DataFlowModel, cf : CreateFlow, t : DfTask, e : DfEntity \bullet ((t,e,cf \in_m m) \wedge Connects(f,t,e)) \longrightarrow \exists\, x : Instance \bullet (InstanceOf(e,x) \wedge CreatedBy(f,x))$$

**Law 39.** Each create flow assigns some carried data into some attributes of the target entity's instance

$$\forall\, m : DataFlowModel, cf : CreateFlow, t : DfTask, e : DfEntity, d1 : Datum \bullet ((cf,t,e \in_m m) \wedge Connects(cf,t,e) \wedge Transmits(cf,d1) \longrightarrow \exists\, x : Instance, att1 : DfAttribute \bullet ((att1 \in_p x) \wedge InstanceOf(e,x) \wedge CreatedBy(cf,x) \wedge Assigns(cf,d1,att1)$$

**Law 40.** Each delete flow deletes some instances (one or more) of a target entity

$$\forall\, m : DataFlowModel, df : DeleteFlow, t : DfTask, e : DfEntity \bullet ((t,e,df \in_m m) \wedge Connects(df,t,e)) \longrightarrow \exists\, x : Instance \bullet (InstanceOf(e,x) \wedge DeletedBy(df,x))$$

**Law 41.** Each flow, in the *DataFlow Model*, that connects an entity to a task is a read flow

$$\forall\, m : DataFlowModel, f : Flow, e : DfEntity, t : DfTask \bullet ((e,t,f \in_m m) \wedge Source(f,e) \wedge Target(f,t)) \longrightarrow DfReadFlow(f) \wedge Connects(f,e,t)$$

**Law 42.** Each *read* flow reads one or more instance(s) of a target entity

$$\begin{aligned} &\forall\ m : DataFlowModel, rf : ReadFlow,\\ &t : DfTask, e : DfEntity \bullet ((t, e, rf \in_m m)\\ &\wedge Connects(rf, t, e))\\ &\longrightarrow\\ &\exists\ x, y : Instance \bullet (InstanceOf(e, x)\\ &\wedge\ InstanceOf(e, x) \wedge ReadBy(rf, x)\\ &\wedge\ ReadBy(rf, x) \wedge (x \neq y)) \end{aligned}$$

**Law 43.** Each flow, in the *DataFlow Model*, that connects bi-directionally an entity to a task is an update flow

$$\begin{aligned} &\forall\ m : DataFlowModel, f : Flow, e : DfEntity,\\ &t : DfTask \bullet ((e, t, f \in_m m) \wedge (Source(f, e)\\ &\wedge Target(f, t)) \wedge (Source(f, t) \wedge Target(f, e)))\\ &\longrightarrow\\ &DfUpdateFlow(f) \wedge Connects(f, e, t) \wedge Connects(f, t, e) \end{aligned}$$

**Law 44.** Each *update* flow that connects bi-directionally an entity $e$ to a task $t$, in the *DataFlow Model*, consists of a read flow that connects $e$ to $t$ and a *write* flow that connects $t$ to $e$.

$$\begin{aligned} &\forall\ m : DataFlowModel, uf : UpdateFlow,\\ &e : DfEntity, t : DfTask \bullet (e, t, uf \in_m m)\\ &\longrightarrow\\ &\exists\ rf : ReadFlow, wf : WriteFlow\\ &\bullet ((rf, wf \in_m m) \wedge Breakdown(uf, rf, wf)\\ &\wedge\ Connects(rf, e, t) \wedge Connects(wf, t, e)) \end{aligned}$$

**Law 45.** Each write flow updates an instance of a target entity

$$\begin{aligned} &\forall\ m : DataFlowModel, wf : WriteFlow,\\ &t : DfTask, e : DfEntity \bullet ((t, e, wf \in_m m)\\ &\wedge Connects(wf, t, e))\\ &\longrightarrow\\ &\exists\ x : Instance \bullet (InstanceOf(e, x) \wedge UpdatedBy(wf, x)) \end{aligned}$$

**Law 46.** Each write flow modifies some attibute values of the target entity's instance by some carried data.

$$\begin{aligned}
\forall\ m : \mathit{DataFlowModel}, rf : \mathit{ReadFlow}, wf : \mathit{WriteFlow},\\
t : \mathit{DfTask}, e : \mathit{DfEntity}, d1, d2 : \mathit{Datum}\\
\bullet\ ((rf, wf, t, e\ \in_m\ m)\ \wedge\ \mathit{Connects}(rf, e, t)\\
\wedge\ \mathit{Connects}(wf, t, e)\ \wedge\ \mathit{Transmits}(wf, d1)\\
\wedge\ \mathit{Transmits}(wf, d2)\ \wedge\ (d1\ \neq\ d2)\\
\longrightarrow\\
\exists\ x : \mathit{Instance}, att1, att2 : \mathit{DfAttribute}\\
\bullet\ ((att1, att2\ \in_p\ x)\ \wedge\ \mathit{InstanceOf}(e, x)\\
\wedge\ \mathit{ReadBy}(rf, x)\ \wedge\ \mathit{Assigns}(wf, d1, att1)\\
\wedge\ \mathit{Assigns}(wf, d2, att2)\ \wedge\ (att1\ \neq\ att2))
\end{aligned}$$

#### 5.3.1.5 Boundary

The final generated *DataFlow* artefact is the detailed *DataFlow Model*. It consists of a number of logical boundaries: $\forall m : DataFlowModel, \exists b : DFBoundary \bullet (b \in_m m)$. The *DataFlow DfBoundary* represents a business task that is decomposed into its atomic actions (tasks) within this boundary. The rectangular system boundary notation is drawn around groups of tasks and entities, to indicate that these element fall within the scope of the original business task.

#### 5.3.1.6 Textual Expressions For Describing Data of Flows

Business users are able to annotate the generated initial DFD model using a structured textual expressions on flows. The following table (5.3) describes this:

| **Concept** | **Textual expression** | **Comment** |
|---|---|---|
| Single variable | $x$ | Java naming convention of primitive datatypes |
| Multiple variables | $x, y, z$ | separated by commas |
| Single object | $Y$ | Java naming convention of objects |
| Value (number) | 665 | numbers only without quotation |
| Value (text) | "*value*" | any valid value within quotation |
| Condition expression | [...] | a logical operator between brackets |
| Multiple conditions | [... *AND* ... *AND* ...] | logical operators between brackets separated by "*AND, OR*" or comma |
| Logical operator | $>, <=, <>, AND, OR$ | |
| Attributes | @$x$, @$y$ | "@" symbol for indicating attribute |
| Projection | [...] @$x$ | the attribute "@x" is projected |
| Assignment | @$y\ =\ 5$ | assignment operator without brackets |

Table 5.3: Textual Expressions for Describing Data on Flows

For each flow connected to an entity, the type of the return value on that flow is the type of entity, unless a projection expression exists. For instance, suppose we have an entity called *Student* and we want to retrieve the name of the one who has *id = 10012*. This is written as:

$$Student[@id \;=\; 10012]\;@name$$

The following expression represents how the status of a particular student is updated after completing their degree.

$$Student[@id = 10012]@status = "graduated"$$

## 5.4 Overview of the (Screen) State Model

The *Screen State Model* is the lowest level, ***automatically-generated***, analysis representation that demonstrates, in very abstract detail, the behaviours of the system in terms of states and transitions. It is mainly based on the notion of the UML State Machine Diagram[116], which contains characteristics from both Mealy and Moore finite state machines[10]. Each state is used to indicate a current system status (displayed screen), and when an event occurs, the system will transit to the next state due to a transition triggered (labelled) by an event and/or conditions.

The model is regarded as navigational, the specifications of the GUI widgets in each screen being out of its scope. The purpose of the model is to describe, in an abstract way, the legal and illegal scenarios involved in business processes and the effects of actions within each process. As a consequence, the complete *State Model* expresses the most successful scenario of a process as well as all possible unsuccessful ones in order that any errors that occur during their execution can be reported and dealt with. Those scenarios occurring in a particular task are grouped together and modelled inside a logical boundary. As a result, a collection of boundaries that represent the internal behaviours of business tasks jointly form the content of the *State Model*.

Two kinds of errors are captured in the model, one being regarded as user-input validation and the other as related to failures caused by data source operations. The user-input error might be raised when feeding the system with values that have an unexpected data type or a null value. This issue is handled by reporting the error to the user and going back to the current state (screen). On the other hand, errors related to the back-end database system, such as connection failure, denial of access, or query/update failure, raise exceptions that notify the user about the type of error and give the option of terminating or resetting the task again.

Achieving a complete *State Model* leads to a comprehensive picture of the behaviours of the information system. This model can be understood by business users as being organised within a set of boundaries. Figure 9.11, 9.12 and other figures, Chapter 9, demonstrate some examples of the generated *State Model* including its core concepts. Each boundary represents the business task with the internal behaviours of its subtasks.

Additionally, transition labels, containing an action's name and precondition (if applicable) only, with the descriptive states name, showing whether the system is waiting for the users input or is ready to perform a database action, produce a more a readable model from an analysis perspective. The *State Model*, in the metamodelling, is considered a *Model*. Their concepts are defined using FOPL and relate to the corresponding $\mu$ML metamodel concepts.

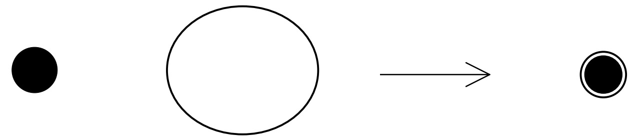
**Law 47.** The *StateModel* is a kind of *Model*.

$$StateModel \ <: \ Model$$

$$\forall \ m \ \bullet \ StateModel(m) \longrightarrow Model(m)$$

### 5.4.1 Graphical Notations of the State Model

In this section, each concept that appears in the *State Model* is introduced with some details about its usage and how it is visualised graphically in the model.



(a) Start (b) State (c) Transition (d) End

Figure 5.10: Generated State model. Core Elements

This model can be similar to state machine diagrams, employed in other approachs, in which each boundary of the system starts start off by an *initial* state (pseudostate) and end up by an *end* state. Each boundary has at most one initial and one end pseudostate. These states show only the start or end of the main business task, which do not have any internal behaviour.

| **Predicate** | **Meaning** | **Syntactic Suger** |
|---|---|---|
| *StateModel(x)* | $x$ is a State Model | *x:StateModel* |
| *State(x)* | $x$ is a Screen State | *x:State* |
| *Waiting(x)* | $x$ is a Waiting State | |
| *Ready(x)* | $x$ is a Ready State | |
| *Transition(x)* | $x$ is a Transition | *x:Transition* |
| *Variable(x)* | $x$ is a local Variable | *x:Variable* |
| *Start(x)* | $x$ is an initial pseudostate | *x:Start* |
| *End(x)* | $x$ is a final pseudostate | *x:End* |
| *StBoundary(x)* | $x$ is a logical boundary (region) | *x:StBoundary* |

Table 5.4: Predicates for $\mu$ML State Model

In order to motivate the list of policies that specifies *State Model*, two basic laws must be identified first as follows:

**Law 48.** Any *Node* in the *StateModel* can be either a *State* , *Start*, or *End*.

$$
\begin{aligned}
&(State \;<:\; Node) \wedge (Start \;<:\; Node) \wedge (End \;<:\; Node)\\
&\qquad \forall\, n : Node, m : StateModel \bullet (n \in_m m)\\
&\qquad\qquad \longrightarrow\\
&\qquad\qquad\quad State(n) \oplus Start(n) \oplus End(n)
\end{aligned}
$$

**Law 49.** Any *Arc* in the *StateModel* is a *Transition*.

$$
\begin{aligned}
&(Transition \;<:\; Arc)\\
&\forall\, a : Arc, m : StateModel \bullet (a \in_m m)\\
&\qquad \longrightarrow\\
&\qquad\quad Transition(a)
\end{aligned}
$$

**Law 50.** Any boundary in the *StateModel* contains one *Start* and one *End* pseudostate, and some *States* and *Transitions*.

$$
\begin{aligned}
&\forall\, m : StateModel,\ \forall\, b : StBoundary \bullet (b \in_m m)\\
&\qquad \longrightarrow\\
&\qquad\quad \exists\, t : Transition,\ \exists\, s : State,\ \exists!\, init : Start,\\
&\qquad\quad \exists!\, e : End \bullet (t, s, init, e \in_m b)
\end{aligned}
$$

#### 5.4.1.1 State

A *State* in the *State Model* is considered a ***simple state***, in respect to the UML terminology, in which it does not have substates. A *State* is depicted in the model, using the circle node, to represent a *current status* of the system at a time. Figure 5.10(b) illustrates the graphical notation of *State*. It is worth mentioning that there is no difference between the visual representation of *waiting*, *error* and *ready* state. Every *State* that appears in the model can be either *waiting*, *error* or *ready*. This can be formalised as:

$$
\begin{aligned}
&\forall\, m : StateModel,\ \forall\, s : State \bullet (s \in_m m)\\
&\qquad \longrightarrow\\
&\qquad\quad Waiting(s) \oplus Ready(s) \oplus Error(s)
\end{aligned}
$$

*State* is defined as a subtype of the *Node* element ($State \;<:\; Node$). Thus, it is identified using a unique identifier that formalised in FOPL as:

$$
\begin{aligned}
&\forall\, m : StateModel,\ \forall\, s : State \bullet (s \in_m m),\\
&\qquad \exists!\, id : Identifier \bullet Id(s, m, id)
\end{aligned}
$$

#### 5.4.1.2 Transition

A *Transition* is a directed arrow drawn from the current state to the next state of a system. The UML notation of transition is adopted, an open-headed arrow, to represent the *Transitions* in the *State Model*. Figure 5.10(c) demonstrates the graphical notation of *Transition*. The *Transition* can be defined in the following theorem that follows axiom (law 10) by substituting *Node* with *State*, as well as substituting *Edge* with *Transition*. This can be written as:

$$\begin{aligned}
&\forall\, m : StateModel,\ \forall\, t : Transition\ \forall\, s1, s2 : State\\
&\quad \bullet\ ((s, t\ \in_m\ m)\ \wedge\ Connects(t, s1, s2))\\
&\qquad \longrightarrow\\
&\qquad\quad Source(s1)\ \wedge\ Target(s2)
\end{aligned}$$

#### 5.4.1.3 Start and End Pseudostate

Two types of pseudostate are used in the *State Model*, namely *Start* and *End*. The *Start* one is placed within a logical boundary to identify the starting state of the internal behaviour of a business task. The UML notation of the initial pseudostate is used, Figure 5.10(a) above.

On the other hand, The *End* pseudostate is placed within a logical boundary to identify the end of the execution of a state machine. Similar to the *Start* pseudostate, the UML notation of the final pseudostate is used, Figure 5.10(d) above.

#### 5.4.1.4 State Boundary

Similar to the previously presented *boundary* concepts in other $\mu$ML models, such as *Task, Impact* and *DataFlow* models, the *StBoundary* element in the *State Model* is used to bind a complete simple state machine diagram that contains *States, Transitions* and *Pseudostates.* This captures a complete behaviour of a part of the system that falls within such a subsystem scope. *StBoundary* can be defined using FOPL as: $\forall\, m : StateModel,\ \exists\, b : StBoundary\ \bullet\ (b\ \in\ m)$.

## 5.5 Outlook on the Chapter

This chapter presented, in detail, the concepts, adopted graphical notations, and formal semantics for each Micro Modelling Language ($\mu$ML) model appearing in the *Analysis* phase of BUILD. It covered all significant concepts, their utilisation, and their all possible interpretations within the various contexts of the $\mu$ML Data Dependency, DataFlow and State Model.

In addition to this, the chapter represented how the generated intermediate system views contribute to the main goals of the thesis. More specifically, it shows how the generated models together hold a comprehensive detailed view of the system using small and semantically clean notation. Moreover, the chapter discussed how notations of the designed core elements, in each model, were reused to appear in $\mu$ML analysis models.

The semantics and notation for each model and element are compared to similar or closely related UML concepts and notation, showing the simplicity and clarity of the proposed language. A number of unary and n-ary predicates were introduced for defining concepts and their interrelationships within each model in respect of the previously defined metamodel.

# 6

# $\mu$ML Concepts and Notations

*In the Design Phase*

"Adding manpower to a late software project makes it later!" Brooks Law

## 6.1 Context

This chapter describes those Micro Modelling Language ($\mu$ML) models that appear within the Design Phase of BUILD, namely, Database and Query (DBQ), Graphical User Interface (GUI) and Code Model. For each model, the chapter introduces, in detail, concepts, underlying ASTs and graphical notations with formally defined semantics using a set of FOPL policies and discusses how the model satisfies its purpose within the framework.

## 6.2 Overview of the Database and Query Model

A *Database and Query Model (DBQ)*, shortened to a *Database Model*, is the lowest level, ***automatically-generated***, model that describes, in a generic way, the common concepts and behaviours that exist in various relational database systems, such as *MySQL, Oracle* and *Microsoft SQL*. The model consists of one or more data schemas that hold all data and query definitions. It uses familiar database terminologies to describe the core concepts of relational database systems.

Concepts in DBQ fall into two categories, namely, *Data Definition* and *Data Manipulation and Query* concepts. The *Data Definition* concepts are used to describe the structure of tables, constraints that are applied to fields and special fields (keys) that represent relationships between tables. A schema element is used to define the logical database schema that consists of a number of persistent tables that contain some atomic columns, defined using a *DBQ Table* and *Column* concept, respectively.

The *Table* element reflects the definition of a database table, that is, the table type definition with its associated column specifications, as found in a real database system. It has a number of columns and special columns (a primary key and possible foreign keys) represented by *Column, PrimaryKey* and *ForeignKey* concepts, respectively. *DBQ Columns* are considered to be atomic elements in the model that cannot be further decomposed. Together they define the type of each instance (new row) of the table they belong to.

It is worth mentioning that relationships between business entities are also expressed at this level either by foreign keys or separate tables (*Linkers*). The decision to convert relationships into either a table or a special column is made according to the application of a transformation rule. This is discussed separately in Chapter 7 and 8.

In addition to the *Data Definition* concepts, *Data Manipulation and Query* concepts, which are used for constructing appropriate queries about a domain, are also described in a declarative way. For example, *Trigger* and *Procedure* elements are used to define, respectively, triggers and pre-defined queries that exist in a particular schema. The trigger is used to capture *range constraints* applied to some columns of a particular table. As a result, the generated *SQL* script includes a *BEFORE INSERT* trigger attached to the table to enforce constraints that prevent or allow new data to be inserted.

The format for query expressions is regarded sufficiently general in order to be translated into relational model (*SQL*) for various database vendors. The declarative style of describing database queries, including the *Projection* ($\pi$), *Selection* ($\sigma$) and *Join* ($\bowtie$) operations, is inspired by *Functional Algebra*, as it discussed later in section 6.2.2. The following example illustrates the expression of a query using the DBQ query syntax, *Relational Algebra* syntax, and natural language (English).

The query: "*Retrieve the name of a module that has a code equals to COM6006*" can be expressed declaratively using Relational Algebra as: $\pi_{name}(\sigma_{code='COM6006'}(Module))$;. The following listing (6.1) illustrates the equivalent DBQ query expression. *Query* as a higher-order function is defined. It takes *Project* as a first argument and a *Relation* as a second argument, to which the *Project* is applied.

Listing 6.1: The structure of the DBQ Query Expression

```
<dbs:Query id="14" name="Reteive_Module_Name">
    <dbs:Project id="15">
        <dbs:Column id="15" name="name" size="0"  prefix="Module">
            <mod:Type id="16" name="Integer"/>
        </dbs:Column>
    </dbs:Project>
    <dbs:Relation id="17" name="Module">
        <dbs:Filter id="18">
            <dbs:Relation id="19" name="Module"/>
            <dbs:Operator id="20" type="boolean" symbol="equals">
                <dbs:Column id="21" name="code" size="20">
                    <mod:Type id="22" name="INTEGER"/>
                </dbs:Column>
                <dbs:Literal id="23" type="VARCHAR"  value="COM6006"/>
            </dbs:Operator>
        </dbs:Filter>
    </dbs:Relation>
</dbs:Query>
```

The query language starts from the assumption that the results of all operations on tables, e.g. *Filter* (listing 6.1 above) and *Join*, are treated as *Relations*. It can be said that the Relation element is an abstracted term of *Table*. This means that any *Relation* in DBQ refers to either an actual database table or else filtered or joined tables.

As a consequence of using the notion of *Relation* in the model, direct navigation through the data is clearly expressed in a unified way for all operations. The concrete path to data within *Relations* is used without the necessity to be computed during transformation.

The *Database Model*, in the proposed metamodelling hierarchy, is considered a kind of $Model$. Thus, conepts of the *DBQ* model can be defined using FOPL and relate to the corresponding $\mu$ML metamodel concepts.

**Law 51.** The *DatabaseModel* is a kind of *Model*.

$$DatabaseModel <: Model$$

$$\forall\, m \bullet DatabaseModel(m) \longrightarrow Model(m)$$

### 6.2.1 Notation and Semantics of the Database and Query Model

In this section, each concept that appears in the *Database Model* is discussed in detail. This includes its usage and how it is visualised graphically in the model. It is worth emphasising that a number of UML notations from the *Class Diagram* are adopted and reused with a slightly different meaning compared to the usual UML semantics. The following table (6.1) summaries all predicates used in describing $\mu$ML Database and Query Model.

| Predicate | Meaning | Syntactic Sugar |
|---|---|---|
| *DatabaseModel(x)* | $x$ is a Database & Query (DBQ) Model | *x:DatabaseModel* |
| *Schema(x)* | $x$ is a DBQ Schema | *x:Schema* |
| *Table(x)* | $x$ is a DBQ Table | *x:Table* |
| *Relation(x)* | $x$ is a DBQ Relation | *x:Relation* |
| *Column(x)* | $x$ is a DBQ Column | *x:Column* |
| *Type(x,y)* | a column $x$'s value has a $y$ datetype where $y \in \{INTEGER, VARCHAR, DATE\}$ | |
| *Child(x,y)* | the node $x$ is a child of the node $y$ | |
| *Trigger(x)* | $x$ is a DBQ Trigger | *x:Trigger* |
| *Procedure(x)* | $x$ is a DBQ Stored Procedure | *x:Procedure* |
| *PrimaryKey(x)* | $x$ is a DBQ Primary Key | *x:PrimaryKey* |

| | | |
|---|---|---|
| *CompositePK(x)* | $x$ is a DBQ Composite Primary Key | *x:CompositePK* |
| *CompositeFK(x)* | $x$ is a DBQ Composite Foreign Key | *x:CompositeFK* |
| *ForeignKey(x)* | $x$ is a DBQ Foreign Key | *x:ForeignKey* |
| *PartOfKey(x, y)* | $x$ is a part (column) of a Composite Key $y$ | |
| *Create(x)* | $x$ is a DBQ Insert operation | *x:Create* |
| *Query(x)* | $x$ is a DBQ Select operation | *x:Query* |
| *Update(x)* | $x$ is a DBQ Update operation | *x:Update* |
| *Delete(x)* | $x$ is a DBQ Delete operation | *x:Delete* |
| *Defines(x, y)* | a column $x$ defines the characteristics of each row uniquely in table $y$ | |
| *Refers(x, y, z)* | a column $x$ refers table $y$ to table $z$ | |
| *Refers(x, y)* | the relation $x$ refers to the table $y$ | |
| *Join(x)* | $x$ is a join operation | *x:Join* |
| *Joins(x, y, z)* | $x$ is a join operation between relation $y$ and relation $z$ | |
| *Filter(x)* | $x$ is a filter function | *x:Filter* |
| *Project(x)* | $x$ is a project function | *x:Project* |

Table 6.1: Predicates for $\mu$ML Database and Query Model

In order to motivate some policies that specify the *DBQ Model*, basic declarations of core elements must be identified first. According to the $\mu$ML metamodel hierarchy where *DBQ Model* table is a subtype of *Node* as: ($Table\ <:\ Node$), then any *Node* in the *Database Model* or schema represents a logical table: $\forall\ n : Node, m : Schema\ \bullet\ (n\ \in_m\ m)\ \longrightarrow\ Table(n)$.

In addition to this, the *Procedure* element in the *DBQ Model* is declared in the $\mu$ML metamodel as a *Function* element, in which the DBQ *Procedure* is a subtype of *Function*, ($Procedure\ <:\ Function\ <:\ Expression$). Therefore, any *Function* in the model represents a stored procedure: $\forall\ p : Function, s : Schema\ \bullet\ (p\ \in_m\ s)\ \longrightarrow\ Procedure(n)$.

#### 6.2.1.1 Schema

A *Schema* is the metadata for a database, consisting of data and query definitions. According to the $\mu$ML metamodel, the DBQ *Schema* element is also considered a subtype of the *Node* element ($Schema \ <: \ Node$). Therefore, it is identified using a unique identifier. This can be defined in FOPL as:

$$\begin{aligned}&\forall\ s : Schema,\ \forall\ d : DatabaseModel\ \bullet\ (s\ \in_m\ d),\\ &\qquad \exists!\ id : Identifier\ \bullet\ Id(s, id)\end{aligned}$$

A *Schema* is constructed by considering the internal structure of business entities (*Table*), e.g. named and typed columns, and the relationships between entities that are represented via key columns (*PrimaryKey* and *ForeignKey*), and finally the various kinds of pre-defined queries (*Procedure* and *Trigger*) that will be executed over the data.

**Law 52.** In the *DatabaseModel*, each node in the schema is a table.

$$\begin{aligned}&\forall\ s : Schema, d : DatabaseModel,\ \forall\ n : Node\\ &\qquad \bullet\ ((n\ \in_m\ s)\ \wedge\ (s\ \in_m\ d))\\ &\qquad\qquad \longrightarrow\\ &\qquad\qquad Table(n)\end{aligned}$$

**Law 53.** In the *DatabaseModel*, each function in the schema can be either a trigger or a stored procedure.

$$\begin{aligned}&\forall\ s : Schema, d : DatabaseModel,\ \forall\ p : Function\\ &\qquad \bullet\ ((p\ \in_m\ s)\ \wedge\ (s\ \in_m\ d))\\ &\qquad\qquad \longrightarrow\\ &\qquad\qquad Procedure(n)\ \oplus\ Trigger(n)\end{aligned}$$

#### 6.2.1.2 Relation

Based on the concept of a *Relation* in *Relational Databases* and also in the *Relational Algebra*, in which everything is considered a relation, the DBQ *Relation* element serves the same purpose. Whereas *Table* represents the type of a database table, *Relation* represents the actual table (the collection of rows), and, recursively, the result of performing relational algebra operations on relations. It is used for binding some internal database operations (functions), such as *filter* and *join* and presenting their returned result as a *Relation* with a given name.

$$\begin{aligned}&\forall\ s : Schema,\ \forall\ r : Relation\ \bullet\ (r\ \in_m\ s),\\ &\exists!\ ((f : Filter\ \bullet\ (Child(r, f))\ \oplus\ (j : Join\ \bullet\ (Child(r, j)))\end{aligned}$$

In addition to this, the DBQ Relation is also used to refer to the collection of rows in a table, defined in another DBQ *Table* element represented within the same database schema. This interpretation is intended when when the *Relation* node has no *descendants*.

$$\begin{aligned}&\forall\ s : Schema,\ \forall\ r : Relation\ \bullet\ ((r\ \in_m\ s)\ \wedge\ (Child(r, \varnothing))),\\ &\qquad \exists!\ t : Table\ \bullet\ ((t\ \in_m\ s)\ \wedge\ (Refers(r, t)))\end{aligned}$$

#### 6.2.1.3 Table

As the intention for DBQ is to express low-level logical schemas that can be directly implemented in a relational database system, business entities that appeared previously at the *Requirement Sketching* and *Analysis* phases are declared at this level as *Table* types. Each *Table* reflects the usual notion of a database table and its properties as found in real-world database systems. Figure 6.1 illustrates the structure of the DBQ table.

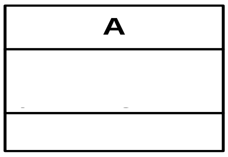



Figure 6.1: Generated Database and Query model. Structure of a Table

According to the $\mu$ML metamodel, DBQ *Table* is also considered a subtype of the *Node* element ($Table \;<:\; Node$). Therefore, it is identified using a unique identifier. This can be defined in FOPL as:

$$\forall\ s : Schema,\ \forall\ t : Table\ \bullet\ (t\ \in_m\ s),$$
$$\exists!\ id : Identifier\ \bullet\ Id(t, id)$$

DBQ Tables always contains one or more columns that describe the shape of the data stored in the table: $\forall\ t : Table,\ \forall\ c : Column\ \bullet\ (c\ \in_p\ t)$. Additionally, tables have some key columns: $\forall\ t : Table,\ \forall\ fk : ForeginKey\ \bullet\ (fk\ \in_p\ t)$. The description of *Column, PrimaryKey, CompositePk* and *ForeginKey* are presented in the following sections.

**Law 54.** In the *DatabaseModel*, each table has a primary key.

$$\begin{aligned} &\forall\ s : Schema,\ \forall\ t : Table,\ c : Column \\ &\bullet\ ((c\ \in_p\ t)\ \wedge\ (t\ \in_m\ s)\ \wedge\ Defines(c, t)) \\ &\longrightarrow \\ &PrimaryKey(c) \end{aligned}$$

**Law 55.** In the *DatabaseModel*, a table has only one composite key.

$$\begin{aligned} &\forall\ s : Schema,\ \forall\ t : Table,\ (c1, c2) : Column \\ &\bullet\ (((c1, c2)\ \in_p\ t)\ \wedge\ (t\ \in_m\ s) \\ &\wedge\ (Defines(c1, t)\ \wedge Defines(c2, t))\ \wedge\ (c1\ \neq\ c2) \\ &\longrightarrow \\ &\exists!\ ck : CompositeKey\ \bullet\ (ck\ =\ c1 \cup c2) \end{aligned}$$

#### 6.2.1.4 Column

The *Column* element is used to define the actual columns of a table at DBQ. Additional features exist, which include the ability to declare a recognised datatype for each column's value, such as *VARCHAR, INTEGER* and *DATE*:

$$\forall\, c : Column,\ \forall\, t : Table\ \bullet\ (c\ \in_p\ t), Type(c, VARCHAR)\ \oplus Type(c, INTEGER)\ \oplus\ Type(c, DATE)$$

Moreover, at this level, a column can be defined whether it is special or not. Some columns are also *foreign keys* that represent relationships or *primary keys* that represent a structure of a table:

$$\begin{aligned}&\forall\, c : Column,\ \forall\, t : Table\ \bullet\ (c\ \in_p\ t)\\ &\longrightarrow\\ &(PrimaryKey(c)\ \vee\ ForeignKey(c))\ \vee\ Column(c)\end{aligned}$$

#### 6.2.1.5 Primary Key

A Primary Key is regarded an essential feature of database tables. Each DBQ *Table* has one that might be either manufactured (automatically generated) or specified manually by a user. This can be defined using FOPL as: $\forall\, pk : PrimaryKey,\ \forall\, t : Table\ \bullet\ ((pk\ \in_p\ t)\ \wedge\ (Auto(pk)\ \oplus\ \neg\ Auto(pk)))$. The following figure (6.2) shows the visual representation of *Primary Key* within a table, which is the bold underlined attribute.

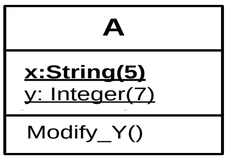



Figure 6.2: Generated Database and Query model. Table's Structure

**Law 56.** In the *Schema* (DBQ) Model, each primary key in such a table defines each row of that table.

$$\begin{aligned}&s : Schema,\ \forall\, pk : Column,\ \forall\, t : Table\\ &\bullet\ ((pk\ \in_p\ t)\ \wedge\ (t\ \in_m\ s)\ \wedge\ Defines(pk, t))\\ &\longrightarrow\\ &PrimaryKey(pk)\end{aligned}$$

In some cases, the table might contain two or more columns that identify the uniqueness of its rows. Therefore, these columns are captured and treated as a composite primary key, (*CompositeKey*), of that table.

**Law 57.** In the *DatabaseModel*, a composite key defines a table if that table has more than one primary key.

$$\begin{aligned}&s : Schema,\ \forall\, c1, c2 : Column,\ \forall t : Table\\ &\bullet\ ((c1, c2\ \in_p\ t)\ \wedge (t\ \in_m\ s)\\ &\wedge\ (Defines(c1, t)\ \wedge\ Defines(c2, t))\\ &\wedge\ (c1\ \neq\ c2))\\ &\longrightarrow\\ &\exists!\ ck : CompositeKey\ \bullet\ (PartOfKey(c1, ck)\\ &\wedge\ PartOfKey(c2, ck)\ \wedge\ Defines(ck, t))\end{aligned}$$

#### 6.2.1.6 Foreign Key

Relationships among business objects that are previously expressed, using graphical notation, in Information and Data Dependency model appear here as a number of *Foreign Key* columns attached to *Tables*. It refers to the other table in the relationship. Figure 6.2 above illustrates the visual representation of *foreign keys*, which is the underlined attribute.

**Law 58.** In the *Schema* (DBQ) Model, each foreign key in such a table refers to another table.

$$\begin{aligned} & s : DatabaseModel,\ \forall\ fk : ForeignKey,\ \forall\ t1, t2 : Table \\ & \qquad \bullet\ ((t1, t2\ \in_m\ s)\ \wedge\ (fk\ \in_p\ t1)) \\ & \qquad\qquad \longrightarrow \\ & \qquad\qquad\qquad Refers(fk, t1, t2) \end{aligned}$$

### 6.2.2 The Query Language and Functional Algebra

The query language in DBQ is inspired by two types of *Algebra*, namely, *Relational Algebra* and *Functional Algebra*. *Relational Algebra* is a procedural query language that applies particular operators to one or more relations. Additionally, *Functional Algebra* implies a higher-order functional style, in which functions accept functions as arguments. The following subsections discuss every database operation (function) taking into account how each one relates to *Algebra* concepts.

#### 6.2.2.1 Filter

A *Filter* function is one of the main operations of the algebra that maps a predicate over a set and returns a subset of values which pass the predicate test. This is similar to a filter function in *Functional Algebra*, which accepts a predicate and a list as arguments, returning a filtered list as the result. *Filter* can be defined using the following signature:

$$filter(pred : T\ \rightarrow\ Bool, Set : Set[T]) : Set[T].$$

Regarding to the DBQ query language, a *Filter* element, which is equivalent to *filter* in a *Functional Algebra*, is considered *parent* of two *child* elements, namely, *Relation* and *Operator*:

$$\begin{gathered} \forall\ m : DatabaseModel,\ \forall\ f : Filter\ \bullet\ (f\ \in_m\ m), \\ \exists\ r : Relation, op : Operator\ \bullet\ (Child(r, f)\ \wedge\ Child(op, f)) \end{gathered}$$

The *Operator* is noted as a boolean function (*Operator* <: *Function*) that has comparison operators, such as greater than (>), less than (<) and equals (=), and two arguments to be examined. The arguments are expressed as *child* nodes of *Operator* element in DBQ, which might be *Column, Variable* or *Literal.* Listing 4.1 examplifies the utilisation of *Filter*, in which a substantial predicate is provided to return a subset of Module records (filtered).

#### 6.2.2.2 Project

The second core function in the proposed DBQ query language is the *Project* function that projects out a set of column values, determined by a function, from a set of records. Similar to the *Filter*, the *Project* function corresponds to a *map* function in *Functional Algebra*, which accepts a function over every element in the input list. *Project* can be defined using the following signature:

$$project(func : T \rightarrow U, Set : Set[T]) : Set[U].$$

In the DBQ query language, a *Project* element is utilised to express the *project* opration that is equivalent to *project* in a *Functional Algebra*. The DBQ *Project* node consists of either some *Columns* or *Relations* nodes as *descendants*. This makes *Project* is not a complete query expression in the DBQ language, as the relation argument is missing, but *Project* is considered a function to be applied to a relation in a higher-order way, by a DBQ *Query* operator. However, the *prefix* name, which is an *attribute* of the *Column* element, is used to label the expected type of the second (implicit) argument.

$$\forall\ m : DatabaseModel,\ \forall\ p : Project\ \bullet\ (p\ \in_m\ m),$$
$$\exists\ c : Column, r : Relation\ \bullet\ (Child(c, f)\ \oplus\ Child(r, f))$$

On one hand, when *Project* has a number of *Columns* as *descendants*, it means these columns are projected from the set declared in their *prefix* attribute. On the other hand, when the *child* of *Project* is a *Relation* node, it means that whole columns (tuples) are projected, which is equivalent to $(*)$ in SQL. Listing 6.1 examplifies the utilisation of *Project* element, in which it has only one *child* element (*Column*) to be projected from a set of *Modules* as declared in its *prefix* attribute.

#### 6.2.2.3 Query

The DBQ *Project* and *Filter* functions are often used together to to return data of interest after a search. Therefore, another high-order function might be introduced, *query* that takes a *project* function and a set and returns a selected set. The following signature defines the *Query* function:

$$query(func : T \rightarrow U, Set : Set[T]) : Set[U]$$

The *search* operation is represented in DBQ via the *Query* element that consists of a *Project* and *Relation* element. In the case that the *Relation* is filtered, it will contain a *Filter* element as *child*. Listing 6.1 examplifies the use of *Query* element, where as the following FOPL specifies the *Query* element:

$$\forall\ m : DatabaseModel,\ \forall\ q : Query\ \bullet\ (p\ \in_m\ m),$$
$$\exists\ p : Project, r : Relation\ \bullet\ (Child(p, q)\ \wedge\ Child(r, q))$$

#### 6.2.2.4 Create

*Create* is a usual set- theoretic operation for adding new instances (rows) into a dataset (*insert*). It is a destructive operation that modifies the base tables (dataset) by instantiating a new distinct instance (object) of that table and inserting a new record into it. It is obvious that there is a predicate to prevent inserting duplicated records. The uniqueness of records is based on the values of all fields and equality is judged on the basis of the declared primary keys.

From this, the *Create* function can be defined as a function that takes two arguments: an original dataset (*base*) of some type (*T*), and a new tuple (*extra*) of the same type *T* that is going to be inserted into the *base*. The function returns the number of inserted tuples(*nat*), which is a Natural number ($nat \in \mathbb{N}$), and has the side-effect of a modified dataset (*base'*). The following signature defines the *Create* function:

$$create(base : Set[T], extra : Tuple[T]) : nat, base'$$

In the DBQ *Create* element, a number of assignment operator statements, *Operator* elements with *symbol="assign"*, are used, as *descendant* nodes of *Create*, for each item in the new tuple that is going to be adding to the dataset. The following listing (6.2) demonstrates the structure of the DBQ *Create* element. It is clear to see that *Create* has a list of *Operator* elements that are assignment instructions (*symbol = "assign"*) to insert every item of the new tuple, supplied as *Variables*, into the equivalent column in the dataset.

Listing 6.2: The structure of the DBQ Create Expression

```
<dbs:Create id="28" name="INSERT_Module">
    <dbs:Operator id="29" type="boolean" symbol="assign">
        <dbs:Column id="30" name="code" size="6">
            <mod:Type id="31" name="INTEGER"/>
        </dbs:Column>
        <dbs:Variable id="32" type="INTEGER" name="code"/>
    </dbs:Operator>
    <dbs:Operator id="33" type="boolean" symbol="assign">
        <dbs:Column id="34" name="title" size="20">
            <mod:Type id="35" name="VARCHAR"/>
        </dbs:Column>
        <dbs:Variable id="36" type="VARCHAR" name="title"/>
    </dbs:Operator>
    <dbs:Operator id="37" type="boolean" symbol="assign">
        <dbs:Column id="38" name="credit" size="2">
            <mod:Type id="39" name="VARCHAR"/>
        </dbs:Column>
        <dbs:Variable id="40" type="VARCHAR" name="credit"/>
    </dbs:Operator>
</dbs:Create>
```

#### 6.2.2.5 Update

The *Update* function performs an update operation on every instance of a dataset. In algebra, we assume that it is possible to have functions with side effects. The *Update* function iterates over every row of a dataset and applies an arbitrary re-assignment operation to each record that probably satisfies a predicate. It is assumed that the dataset is modified as a side-effect of *Update*. The following signature defines the *Update* function:

$$update(pred : T \rightarrow U, Set : Set[T]) : nat.$$

It is common in various database systems that the *SQL UPDATE* statement returns a Natural number, indicating the number of records that were modified. In the proposed DBQ *Update*, a number of assignment operator statements are used to express modification to express modifications to column data without considering an expression to present the return value of *Update*.

The following listing (6.3) demonstrates the structure of the DBQ *Update*. It is clear to see that *Update* has an *Operator* of the type assignment (*symbol = "assign"*) to modify the value of the column *title* by the value of the variable *title*.

Listing 6.3: The structure of the DBQ Update Expression

```
<dbs:Update id="40">
    <dbs:Operator id="41" type="Boolean" symbol="assign">
        <dbs:Column id="42" name="title" size="20">
            <mod:Type id="43" name="VARCHAR"/>
        </dbs:Column>
        <dbs:Variable id="44" type="VARCHAR" name="title"/>
    </dbs:Operator>
</dbs:Update>
```

#### 6.2.2.6 Delete

The *Delete* is another common data manipulation operation for removing existing records from a dataset. Similar to *Create*, the *Delete* operation is defined as a destructive operation that modifies a base tables (dataset) by removing records and reducing its size. As *Delete* function removes some records from a dataset, it is clear that there is a predicate that must be applied to every record in the dataset. Only records that pass the predicate are removed. That is why *Delete* is often used with a independant sub-operation *Query* with a *Filter* to achieve this.

From this, the *Delete* operation can be defined as a function that takes two arguments, an original dataset (*base*) of the type *T*, and a projected tuple, of the same type *T* from the performed *filter* (sub-operation). The function returns the number of removed tuples(*nat*), which is a Natural number ($nat \in \mathbb{N}$), has the side-effect of a modified dataset (*base'*). The following signature defines the *Delete* function:

$$delete(base : Set[T], togo : Tuple[T]) : nat, base'$$

Regarding the DBQ *Delete* element, a *child* element *Relation* is used to wrap the *Filter* function (sub-operation) and represents its return result set to the *Delete* function. This filtered *Relation* will be deleted (extracted and remove) from the original *Relation* defined inside the *Filter* element. The following listing (6.4) demonstrates the structure of the DBQ *Delete*.

Listing 6.4: The structure of the DBQ Delete Expression

```
<dbs:Delete id="43" name="DELETE_Code">
    <dbs:Relation id="44" name="Delete␣Module">
        <dbs:Filter id="45">
            <dbs:Relation id="46" name="Module"/>
            <dbs:Operator id="47" type="boolean"
                symbol="equals">
                <dbs:Column id="48" name="code" size="6">
                    <mod:Type id="49" name="INTEGER"/>
                </dbs:Column>
                <dbs:Variable id="50" type="INTEGER"
                    name="code"/>
            </dbs:Operator>
        </dbs:Filter>
    </dbs:Relation>
</dbs:Delete>
```

#### 6.2.2.7 Join

The *join* is a database operation ($\bowtie$) that computes the inner join of two tables, resulting in a merged table, based on a constraint between the values of two columns, one from each table. It is considered a binary *cartesian product* operation ($\times$) with a selection condition ($\theta$). The *join* concept can be expressed using *algebra* as: $\sigma_\theta(p \times q)$, where $p$ and $q$ are two relations. Therefore, the resulting *join* outcome is another relation that consists of columns from both relations $p$ and $q$. The following signature defines the *Join* function:

$$join(Set : Set[T], Set : Set[U]) : Set : JSet[W]$$

### 6.2.3 The Significance of the Database and Query Model

The *Database and Query Model* is a key model in $\mu$ML because it describes a comprehensive representation all generic data schema specifications, required for any relational database generation. This platform-independent model is used as a source model in the code generation approach to produce executable database schema code in various target database systems.

## 6.3 Overview of the Graphical User Interface (GUI) Model

A *Graphical User Interface Model (GUI)* is the lowest level,***automatically-generated***, structural model that describes, in an abstracted way, the detailed GUI architecture that is common in various enterprise system environments. The model contributes to the development of system GUIs by representing the concrete structure of their controls (widgets) that can later generate simple and user-friendly interfaces for their system. Familiar terminologies for describing basic concepts of GUI are used to form the declarative GUI language. The model is considered to be platform- independent as no implementation details are presented in the model.

Much like the modelling strategy adopted in $\mu$ML, each group of application windows that falls within a scope of a business practice is modelled as a boundary to represent a separate part of the system. Thus, it can be said that the *GUI Model* consists of a number of independent boundaries expressed via *Boundary* nodes. As this is considered during the early development phases in BUILD, a business task may, by this stage, already have been broken down into atomic tasks (each task having an atomic action). It is assumed here that each atomic task requires a user interface to support human-computer interaction throughout the execution of a business process. As a consequence, an appropriate screen design appears in the model for each atomic task that appeared previously in the *(Screen) State Model*.

It can be said that the *Boundary* contains a collection of *Window* nodes that represent IS screens. Each window in a system boundary is prioritised to be linked to another one, forming an ordered sequence or branches (choices) of screens. The screen order is based on the priority number attached to each *Window* node, the window with the highest priority score appearing before those with lower ones to reflect the internal behaviour (logic) of a business process. It is worth mentioning that the priority score is allocated for each window, except error ones, by a previous model transformation step for generating the *State Model*, in which the priority for each *state* is passed to the corresponding *window*, as discussed later in Chapter 7.

Generally, information system screens are classified into two major types: *entry form* and *output report* screen. The *entry form* is a screen that is waiting for an external input/event from a system actor. This appears in the *GUI Model* as *Waiting Windows* with a main button that triggers an *input* action when a *click* event occurs. The *output report* is a ready screen for executing an atomic business task, such as retrieving or deleting information from the system. It appears in the model as *Ready Windows* with a main button that initiates either *create, read, update, delete* or *output (display)* actions. It simply notifies the user that an action is going to be initiated or displays some output message to them.

In addition to this, a failure of a business task is captured by presenting an appropriate *Error Reporting Window* that links to each window in a successful scenario. An *Error Reporting Window* can be defined as a special kind of *Reporting Window* that is used to display (report) a meaningful error message to the user. An *Exception* event that is triggered when any error occurs is attached to every window. As a result, an error reporting window may appear during any step in a scenario.

In fact, the *GUI model* has one kind of node to represent the window concept, the *Window* node. The type of window, whether it is *Waiting* or *Ready*, is extracted from the generated name of each *Window*. A particular naming convention is adopted whereby the name of a window must include its type, such as *Input_title_Waiting* or *Input_title_Waiting_error*.

In typical enterprise applications, business-users visualise and gain access to the backend database of the system by interacting with a number of GUIs to execute some business actions in order to retrieve or manipulate data. This is known as a presentation layer in the traditional architecture of any information system. The widgets in the *GUI Model* hold the required information about the target database tables and fields, with their properties and constraints, along with corresponding data that is shown via these screens. This offers a consistent mapping between the presentation and database layers and leads to the correct visualisation of a window.

The *GUI Model*, in the produced metamodelling hierarchy, is considered a kind of $Model$. Thus, concepts of the *GUI Model* can be defined using FOPL and relate to the corresponding $\mu$ML metamodel concepts.

**Law 59.** The *GUIModel* is a kind of *Model*.

$$GUIModel \ <: \ Model$$

$$\forall\, m \ \bullet \ GUIModel(m) \longrightarrow Model(m)$$

### 6.3.1 Notation and Semantics of the Graphical User Interface (GUI) Model

In this section, each concept that appears in the *GUI Model* is introduced with some details about its usage and how it is visualised graphically in the model. The following table (6.2) summaries all predicates used in describing $\mu$ML GUI Model.

| **Predicate** | **Meaning** | **Syntactic Sugar** |
|---|---|---|
| *GuiModel(x)* | $x$ is a Graphical User Interface (GUI) Model | *x:GuiModel* |
| *GuiBoundary(x)* | $x$ is a IS boundary | *x:GuiBoundary* |
| *Window(x)* | $x$ is an IS Screen | *x:Window* |
| *Label(x)* | $x$ is a label control | *x:Label* |
| *Button(x)* | $x$ is a button control | *x:Button* |
| *Textfield(x)* | $x$ is a Textfield control | *x:Textfield* |
| *Widget(x, y)* | $x$ is a widget in a window $y$ | |

Table 6.2: Predicates for $\mu$ML GUI Model

In order to motivate some policies that specify the *GUI Model*, basic declarations of core elements must be identified first. According to the $\mu$ML metamodel hierarchy where *GUI Model* window is a subtype of *Node* as: ($Window \ <: \ Node$), any *Node* in the *GUI Model* represents a GUI window:

$$\forall\, n : Node, m : GUIModel \ \bullet \ (n \ \in_m \ m) \longrightarrow Window(n)$$

Additionally, various GUI controls are considered *Widget* concept in the metamodel. According to the $\mu$ML metamodel hierarchy, the *Widget* element is denoted a subtype of *Node* as: ($Widget \ <: \ Node$), any *Node* in the GUI window represents a GUI control:

$$\forall\, n : Node, w : Window \ \bullet \ (n \ \in_c \ w) \longrightarrow Widget(n)$$

**Law 60.** In the *GUIModel*, each widget in any window can be either a button, textfield, or label.

$$\begin{aligned} &\forall\ c : Widget,\ \forall\ w : Window, m : GUIModel \\ &\qquad \bullet\ ((w\ \in_m\ m)\ \wedge\ (c\ \in_c\ w)) \\ &\qquad\qquad \longrightarrow \\ &\qquad\qquad\qquad Textfield(c)\ \oplus\ Button(c)\ \oplus\ Label(c) \end{aligned}$$

#### 6.3.1.1 GUI Window

The *Window* element is used to capture every information system screen. According to the $\mu$ML metamodel, GUI *Window* is considered a subtype of the *Node* element ($Window\ <:\ Node$). Therefore, it is identified using a unique identifier. This can be defined in FOPL as:

$$\begin{aligned} &\forall\ m : GUIModel,\ \forall\ w : Window\ \bullet\ (w\ \in_m\ m), \\ &\qquad \exists!\ id : Identifier\ \bullet\ Id(w, id) \end{aligned}$$

Every *Window* cosists of many GUI controls; these are all *Widget* elements. This can be expressed as:

$$\begin{aligned} &\forall\ w : Window, m : GUIModel\ \bullet\ (w\ \in_m\ m), \\ &\qquad \exists\ c : Widget\ \bullet\ (c\ \in_c\ w) \end{aligned}$$

Extra features are added to specify each window. For instance, the priority score of a window is declared, for each *Window* element, using the *order* attribute. Moreover, the declaration of *Error Window* is determind by the boolean attribute *error* attached the *Window* node.

Listing 6.5: The structure of the GUI Window node

```
<gui:Window id="22" name="Input_code_Waiting" order="4">
    <gui:Textfield id="23" name="code" size="6"/>
    <gui:Button id="24" name="Exception" event="Exception"
         exit="false" hidden="true"/>
    <gui:Button id="25" name="Input" event="Input"
        exit="false"/>
</gui:Window>
```

The previous listing (9.8) illustrates a structure of a window for receiving *code* value from the user that consists of a *Textfield* and a *Button* to read the input value. However, it is worth mentioning that the button that fires the *Exeption* is invisible (*hidden =true*).

#### 6.3.1.2 GUI Textfield

The *Textfield* element is used to capture every user interface text field control that appears in a window. It allows the actor to input textual information to be used by the system. The additional specifications of each text field are determined by some attributes attached to a

*Textfield* node, such as the *name* and *size* attributes. Listing 9.8 above exemplifies the GUI *Textfield* element including its attributes. According to the $\mu$ML metamodel, GUI *Textfield* is considered a subtype of the *Widget* element ($Textfield\ <:\ Widget$). Therefore, it is identified using a unique identifier. This can be defined in FOPL as:

$$\forall\, tf : Textfield,\ \forall\, w : Window\ \bullet\ (tf\ \in_c\ w),$$
$$\exists!\ id : Identifier\ \bullet\ Id(tf, id)$$

#### 6.3.1.3 GUI Label

The label is a user interface control that is used to display textual information on a window. It is represented in the GUI Model via the *Label* element that is used to capture every label appears in a screen. *Label* is also specified by some attributes attached to it, such as the *name* and *text* attribute. The *text* attribute the actual text that appears on a window. According to the $\mu$ML metamodel, GUI *Label* is considered a subtype of the *Widget* element ($Label\ <:\ Widget$). Therefore, it is identified using a unique identifier. This can be defined in FOPL as:

$$\forall\, leb : Label,\ \forall\, w : Window\ \bullet\ (leb\ \in_c\ w),$$
$$\exists!\ id : Identifier\ \bullet\ Id(lebf, id)$$

#### 6.3.1.4 GUI Button

The button is another kind of user interface control that enables the user a direct way to trigger an event. It is appeared in a window as a rounded rectangle with longer width than its hight, and a text in its middle. Every button in such a window is expressed in the *GUI Model* through the *Button* element. The GUI *Button* describes many features in regard to the button using a number of attributes, such as, *name* and *event* attribute that define the button's name and the type of event triggered by this button. Listing 9.8 above examplifies the GUI *Button* element including its attributes. According to the $\mu$ML metamodel, GUI *Button* is considered a subtype of the *Widget* element ($Button\ <:\ Widget$). Therefore, it is identified using a unique identifier. This can be defined in FOPL as:

$$\forall\, b : Button,\ \forall\, w : Window\ \bullet\ (b\ \in_c\ w),$$
$$\exists!\ id : Identifier\ \bullet\ Id(b, id)$$

#### 6.3.1.5 GUI Boundary

Similar to the previously presented *boundary* concepts in other $\mu$ML, such as *Task, Impact, DataFlow* and *State*, the *GUIBoundary* node in the *GUI Model* to bind one or more *Window* elements. This indicates that some windows fall within a particular subsystem or (scope) to be developed, while other windows fall within another scope. It can be defined using FOPL as: $\forall\, m : GUIModel,\ \exists\, b : GUIBoundary\ \bullet\ (b\ \in\ m)$.

### 6.3.2 The Significance of the GUI Model

As the Micro Modelling Language ($\mu$ML) seeks simplicity for designers, business-users are relieved from the task of designing the kinds of screens used, since the decision to create a *waiting* or *ready* window is determined automatically, based on information provided in the *Screen State Model.* Additionally, some database tables might might be automatically assigned as data sources, when required, in some screens without any end-user manual modification. Consequently, reasonable chunks of boilerplate code for managing the database connection and exception handling may be manufactured directly at the code generation stage. Moreover, the fact that this is all done automatically provides type crosschecking to preserve consistency between the information system layers.

## 6.4 Overview of the Code Model

The *Code Model* is the lowest level, ***automatically-generated***, AST representation that describes the abstract specification for the Object-Oriented (OO) code of the system in terms of business entities and tasks (classes), their methods, attributes and expressions. It aims to present a comprehensive business logic layer that determines how to manipulate the data in either a separate layer or one combined with the user interfaces, that is, either in a 3-tier or 2-tier architecture, respectively. At the current version of BUILD, it is worth mentioning that the model is **NOT** a part of the transformation chain for generating complete 2-tier applications. It might be used in later versions for developing 3-tier IS applications.

Following the code structure in the OOP languages, the *Code Model* consists of a number of *Clazz*[1] elements that represent the actual OO classes of a system. Each *Clazz* node typically has some *Constructor, Attribute* and *Method* nodes as descendants to express the actual class attributes, constructors and methods at this level of detail.

Apart from this, this model is platform independent and intended to capture sufficient details to generate idiomatic constructs of executable code for a target OOP language, e.g. *Java*, along with adequate chunks of *boilerplate* code to establish and manage consistent protocols between the invocation of methods and database predefined queries in the system layers.

Additionally, the model is completely derived from both the detailed *Database and Query Model* and the *DataFlow Model* without any direct contribution from the end-user. This achieves one of the goals of BUILD, by avoiding end-users from modelling these technical specifications about their system. All modelling activities are done at a higher level of abstraction. As a consequence, one goal of our approach is achieved by relieving users from writing these technical specifications users from these technical specifications regarding their system and producing it at a higher level of abstraction. Business-users, the designers, make no contribution to the detailed architecture of their software system as our approach adopts a common architecture and is able to generate all kinds of systems.

The classes are divided into two main categories, namely, entity classes and process (task) classes. The entity classes are translated from the equivalent tables and exist in the *Database and Query Model*, whereas the process classes are derived from boundaries that appear in the *DataFlow Model.* Relationships between classes, such as *composition* and *inheritance*, are also extracted from the *Database and Query Model* and might be implemented in the *Code Model* to represent a consistent business logic layer.

[1]The name *Clazz* is chosen for this node to avoid a name-clash with the built-in type "Class" in Java.

The *Code Model*, in the produced metamodelling hierarchy, is considered a kind of *Model*. Thus, concepts of the *Code Model* can be defined using FOPL and relate to the corresponding $\mu$ML metamodel concepts.

**Law 61.** The *CodeModel* is a kind of *Model*.

$$CodeModel \ <: \ Model$$
$$\forall \ m \ \bullet \ CodeModel(m) \longrightarrow Model(m)$$

### 6.4.1 Notation and Semantics of the Code Model

In this section, each concept that appears in the *Code Model* is introduced with some details about its usage and how it is expressed in the model. The following table (6.3) summaries all predicates used in describing $\mu$ML Code Model.

| Predicate | Meaning | Syntactic Suger |
|---|---|---|
| *CodeModel(x)* | $x$ is an Code Model | *x:CodeModel* |
| *Clazz(x)* | $x$ is an OOP class | *x:Clazz* |
| *CAttribute(x)* | $x$ is a class attribute | *x:CAttribute* |
| *CMethod(x)* | $x$ is a class method | *x:CMethod* |
| *Constructor(x, y)* | $y$ is a class $x$ Constructor | |
| *Invokes(x, y)* | a method $x$ calls a database stored procedure $y$ | |

Table 6.3: Predicates for $\mu$ML Code Model

In order to motivate the list of policies that specifies the Code Model, two basic laws must be identified first as follows:

**Law 62.** Any node in the CodeModel is a class.

$$CodeModel \ <: \ Model$$
$$\forall \ m : CodeModel, n : Node \ \bullet \ (n \ \in_m \ m) \longrightarrow Clazz(n)$$

**Law 63.** Any function in the CodeModel is a method in a class.

$$\begin{aligned} CodeModel \ &<: \ Model \\ \forall\ m : CodeModel, f : Function, c : Clazz \bullet (c \ \in_m \ m) \\ &\longrightarrow \\ & Method(f) \ \wedge \ (f \ \in_p \ c) \end{aligned}$$

#### 6.4.1.1 Class

The *Clazz* element, in the Code Model, is used to define object-oriented class types. According to the $\mu$ML metamodel, *Clazz* is considered a subtype of the *Node* element ($Clazz <: Node$). Therefore, it is identified using a unique identifier. This can be defined in FOPL as:

$$\forall\ m : CodeModel,\ \forall\ c : Clazz \bullet (c \ \in_m \ m), \exists!\ id : Identifier \bullet Id(c, id)$$

A *Clazz* element consists of a number of attributes to describe its features and methods to describe its operations, expressed using *CAttribute* and *CMethod* nodes as descendants of *Clazz*. This can be formalised in FOPL as:

$$\begin{aligned} &\forall\ m : CodeModel,\ \forall\ c : Clazz \bullet (c \ \in_m \ m), \\ &\exists\ ((a : CAttribute \bullet (a \ \in_p \ c) \ \wedge \ Child(c, a)) \\ &\wedge (f : CMethod \bullet (f \ \in_p \ c) \ \wedge \ Child(c, f))) \end{aligned}$$

An extra feature to specify the accessibility of a class (*public, private* and *protected*) is considered in the Code Model language using an *XML* attribute attached to the *Clazz* element (*visible="public"*).

#### 6.4.1.2 Attribute

A *CAttribute* element is used to declare every fields of a class in the Code Model. It is an atomic node in the model that has no further *childern.* Additional features that specify the attribute are attached, as *XML* attributes, to the *CAttribute* node, such as *name, visible* and *type.* The *CAttribute* elements appear in the model as *descendants* of a *Clazz* node, indicating that these attributes are properties of that class. This can be expressed in logic as:

$$\forall\ m : CodeModel,\ \forall\ c : Clazz \bullet (c \ \in_m \ m), \exists\ att : CAttribute \bullet (att \ \in_p \ c)$$

#### 6.4.1.3 Method

A *CMethod* element is utilised to define the structure of every method in a class.These act as wrappers for binding internal database operations, in which they declare syntax for calling a database stored procedure (pre-defined query) and/or for managing the database connectivity, such as *JDBC* application in *Java.*

*CMethod* has two important kinds of *child* nodes, namely, an *CArgument* element to represent its input parameter and a *Call* element to express a procedure invoked:

$$\forall\ c : Clazz,\ \forall\ f : method\ \bullet\ (f\ \in_p\ c), \exists\ a : CArgument\ \wedge\ Child(f, a)$$

The *Call* element is used to declare a method invocation expression, which is typically a *child* element of *Method*. It refers to a syntax (expression), which is a part of the body of a method, that invokes a stored procedure located in a database schema, expressed previously in DBQ. The name of the invoked pre-defined query is declared in the *Call* node via its *name* attribute. The notion of *Call* can be defined using the following policy:

**Law 64.** Any method, in a class, might call a stored procedure, in a database schema.

$$\begin{aligned} &\forall\ s : DatabaseModel,\ \forall\ m : CodeModel,\ \forall\ f : Method,\\ &\forall\ c : Clazz\ \bullet\ ((c\ \in_m\ m)\ \wedge\ (f\ \in_p\ c)),\\ &\exists\ p : procedure\ \bullet\ ((p\ \in_m\ s)\ \wedge\ Invokes(f, p)) \end{aligned}$$

In the same context, a *Constructor* element, which is used to declare a construction (initialising) of a class, has a number of *Variable* elements and some assignment *Operators*, as its descendants, to represent its input parameters and initialise the attributes of the class.

## 6.5 Outlook on the Chapter

This chapter represented, in-depth, the concepts, underlying ASTs (XML), and formal semantics for each Micro Modelling Language ($\mu$ML) model appearing in the *Design* phase of BUILD. It covered all critical concepts, their usage, and all their possible interpretations within the various contexts of the $\mu$ML Database and Query (DBQ), Graphical User Interface (GUI) and Code Model.

The Code Model was introduced to be used in later versions of BUILD for developing 3-tier ISs, whereas models may be used alone for generating complete 2-tier business applications. Business logics were expressed using predefined stored procedures as a part of database schema. Based on *Relational* and *Functional Algebra*, a query modelling language was introduced for expressing a number of database operations in the DBQ model.

Apart from this, the chapter showed how to reuse the graphical notation from the *Data Dependency* Model for expressing DBQ concepts. There is no graphical representation in the current version of $\mu$ML for the GUI and Code model. The formal semantics and notation for models and their elements are compared to similar or closely related UML concepts and notation, showing the simplicity and clarity of the proposed language. A number of unary and n-ary predicates were introduced for defining concepts and their relationships in each model in respect of the previously defined metamodel.

# 7
# The Architecture of the Model-Transformation Approach

"If something is worth doing once, it's worth building a tool to do it "

Unknown

## 7.1 Context

This chapter is divided into two main sections. The first one provides an in depth description of the overall structure and mechanism of the model transformation approach, highlighted briefly in Chapter 3. The approach encompasses a two level transformation framework, namely, top level and concrete level.

The first sections below discuss the generic design of the top level framework. This level is an ancestor of all actual transformation rules used to generate several $\mu$ML models via the different BUILD development stages. The top level is adopted since it is the predecessor of all actual transformation rules used to generate a variety of $\mu$ML model(s) via the different BUILD development stages, and the architecture of the concrete (actual) model transformation framework.

In addition, the second part of the chapter clarifies, in depth, the overall structure and mechanism of the code generation approach, mentioned briefly in Chapter 3. The current version consists of two platform specific generators. Each generator has two levels of generator agents. Throughout later sections, the overall and the detailed design of the code generation framework is discussed.

## 7.2 Brief Overview of the Model Transformation Framework

The transformation architecture developed for BUILD is simple and flexible. Every model manipulation is classified either as a kind of *translation, transformation, folding (merging)* and *in-place modification*. These four types of model transformation rules are supported by the current version of the framework. A *translation* rule translates a concept from a source model into an equivalent concept in a different type of target model (*exogenous*). A *transformation* rule transforms between models expressed in the same language (*endogenous*). In the case of optimising a model (where only one model is involved in a rule as both its source and target), this rule is considered *in-place modification*. In contrast, when concepts from more than one model are involved in a rule as source, the rule is regarded as a*folding (merging)* transformation rule.

It is worth mentioning that the proposed architecture of transformations in BUILD differs from the previous ReMoDeL [101] version that was used for the MySQL Database Generator, presented in [109] and [108]. That code generation framework encodes transformation rules as methods in each generator responsible for generating a part of the target (*imperative*). On the other hand, the current version of BUILD encodes each rule of transformations as an independent *java* class, representing the *declarative* side of the language. Chapter 8 explained how this leads to a hybrid declarative/imperative approach, where individual transformations are both independent and idempotent, while some order of execution is eventually imposed.

The overall architecture of BUILD is divided into two main layers: *abstract* (generic) and *specific*. Two types of translation rules are used in our transformation approach, namely, (*one-to-one translating) Simple* and (*two-to-one merging) Merging* rules. Our *Simple* rules take an element from a model and generate a new element in another model. *Translation, transformation* and *in-place modification* rules are implemented in the framework as *Simple* rules. On the other hand, our *Merging* rules, in the current version of BUILD, take two elements from different models and generate a new element in a target model.

A simple forward transformation (refinement or code generation) occurs when a source model belongs to a different metamodel type than the target. This activity is frequently seen in the majority of transformation components in BUILD. It enables the shift from a higher development phase to a lower one, introducing new and richer knowledge. The following section discusses these kinds of transformations in more detail.

## 7.3 The Top Level Architecture of the Model Transformation Framework

The top level architecture of the model transformation framework is an Object-Oriented (OO) approach that consists of five main classes, namely *Rule, MergeRule, Translation, MergeTranslation* and *Context*, illustrated in Figure 7.1. The root classes of the framework are the *Rule* and *MergeRule* class.

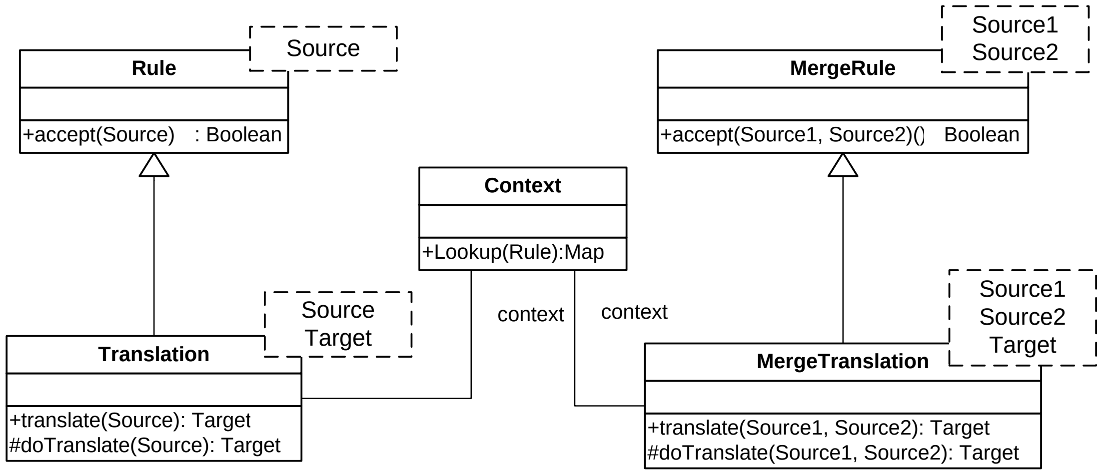



Figure 7.1: Model Transformation in BUILD. The Top Level Framework

The *Rule* class is considered the ancestor of all rules applied to a single source that is regarded a super class of a single subclass (so far), called *Translation*. Based on the type of source and target, this subclass acts as a *model translator* that translates a source model to a target one with a different metamodel, a *model transformer* that translates a source model to a target one within the same metamodel, and an *in-place modifier* that modifies and evolves a source model.

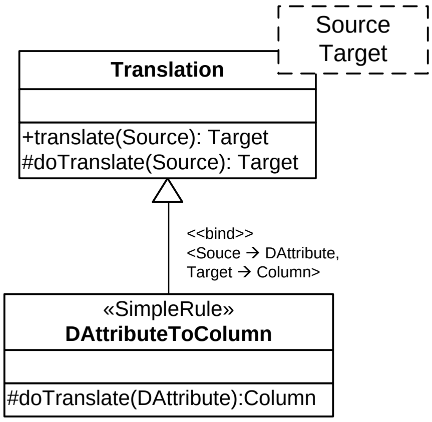



Figure 7.2: Translation Rule Structure

The above Figure (7.2) exemplifies the *Translation* rule structure. It includes a simple rule (class) from the concrete level, called *DAttributeToColumn* class. that is used to translate every Data Dependency attribute (*DAttribute*) into a DBQ column (*Column*).

In additon to this, *MergeRule* is considered a general class of a subclass (so far), called *MergeTranslation*. It takes two source elements and translates (merges) them to produce a target. The following Figure (7.3) exemplifies the *MergeTranslation* rule structure. It includes a simple rule (class) from the concrete level, called *DDiagramToSchema* class. that takes DataFlow and Data Model element to translate them into a DBQ Schema.

Both *Rule* and *MergeRule* have an appropriate *accept* method when the rule has particular restrictions on the valid source elements required to be examined and accepted. This method is effectively the precondition of the rule, which returns *true* when the source element is valid for

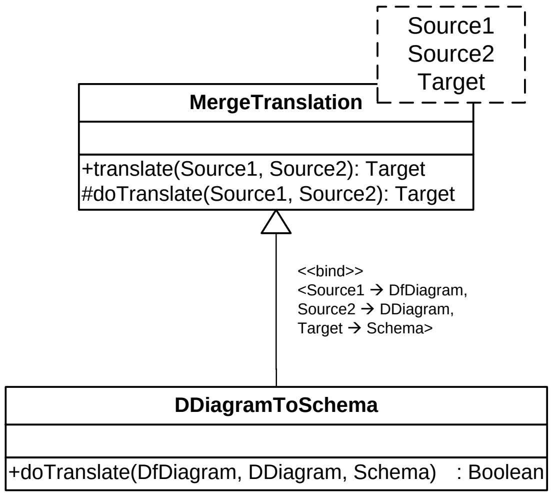



Figure 7.3: Translation Rule Structure

the rule. An overriden version of *accept* may appear in any subclass of *Rule* and *MergeRule* if it has particular restrictions on the valid source elements in the subclass.

The *Context* class is the fifth class in the top level framework. It is a record of the work done by a tree of *Translation* rules. A new *Context* is created whenever a *Translation* rule is created at the top level, to perform a translation. The same *Context* is shared by every descendant rule created by the top level Translation rule. The *Context* indexes all transformations performed by the tree of Transformation rules, under the name of each rule. This allows individual rules to access any existing translation and avoids repeated work.

## 7.4 Framework Architecture at The Concrete Level

The architecture of all translator components at the concrete level is directly influenced by the hierarchical structure of elements in the source model of each. A naming convention for each agent is specified using the following format: $< sourceelementname > To < targetelementname >$.

For instance, considering the structure of nodes in the *Information Model*, the step of translating the *Information Model into the Data Model* (section 7.5.2) consists of *Diagrams*, containing *Nodes* (e.g. *DEntity*), which in turn contain *Properties* (e.g. *DAttribute* and *Identifier*). Additionally, it contains *Relationship* (e.g. *Association, Generalisation* and *Composition*), which in turn contains *Properties* (*Roles*).

Thus, the internal structure of each translator is based upon the concept distribution strategy. This means that each concept in a source model is handled separately by a particular agent (sub-translator), represented via a java class in the framework. Each agent is responsible for applying appropriate transformation rules to create the corresponding target concepts.

Agents present in a hierarchical series of layers delegate to other agents at the next finer level of abstraction. Sometimes these sub-translators refer back to information previously processed by the higher-level (parent) translator. Each translator, which might consist of a set of either *Translation* or *MergeTranslation* or even both rules, behaves in the same manner as the *Composite* and *Visitor design pattern*[42] as some agents, which are containers of others, have a command *translate()* to delegate to their parts (sub-translators), and each agent traverses a part of the model. It is worth mentioning that the *translate()* method is the top-level method. It may either look up a cached result or invoke doTranslate() to compute the result for the first time.

In addition to this, all mapping algorithms are encoded as methods of the framework. The algorithms are imperative, using an ordered collection of transformations on *Task* and *Impact Models*, represented as (as abstract syntax trees). The order of rules execution is controlled and implemented through the body of the *doTranslate()* method in each agent.

## 7.5 Requirement-to-Analysis Model Transformation Approach

As previously presented in Chapter 3, the *Requirement to Analysis Model Transformation* step is regarded as a first forward mapping step that shifts the end-users requirement models towards the analysis phase. In order to construct the required models, which are conceptually located within the BUILD Analysis phase, a number of Java-translator components are designed to implement the (hybrid) mapping rules for each translator agent. These translators are: (*Information-to-DataDependency, Task-Impact-to-DataFlow, DataFlow-to-DetailedDataFlow, DetailedDataFlow-to-State*). Initially, the XML files representing the business user's input are parsed, creating source models. Then these rules are applied to the source models to generate in-memory target models, which in turn will be used for the source models in the next stage.

### 7.5.1 Translating Task and Impact Models into (initial) DataFlow

Translating *Task* and *Impact Models* into (initial) *DataFlow* is considered to be a forward merge translation step. This takes two elements as its *source* from a *Task* and *Impact Model*, producing an equivalent target element in the *DataFlow* artefact. This translation step consists of 15 translating agents that are responsible for traversing both *Task* and *Impact Model*, applying a set of transformation rules to perform the mapping between their elements and the target elements in the *DataFlow Model*. The following figure (7.4) demonstrates the internal structure of the translator.

From this, some agents are considered to be subclasses of the *MergeTranslation* class, at the top level of the framework. These translators are:

- **DiagramToDfDiagram:** The agent takes a *Task Model Diagram* and an *Impact Model Diagram* as source and produces an equivalent *DataFlow Model DfDiagram*. This rule consists of one subrule (*BoundaryToDfBoundary*).
- **BoundaryToDfBoundary:** The agent takes an (equivalent) *Task Model Boundary* and an *Impact Model Boundary* as source and produces an equivalent *DataFlow Model DfBoundary*. This significant translator also exemplifies the merging process of two source elements to obtain a single target. It consists of nine subrules that are responsible for mapping other concepts from the source to corresponding ones in the target *DataFlow Model*.
- **MergeTaskToDfTask:** The agent takes a *Task Model Task* and an *Impact Model Task* as source and produces an equivalent *DataFlow Model Task*. This agent maps two source elements, extracted from different models, and produces a single target element.

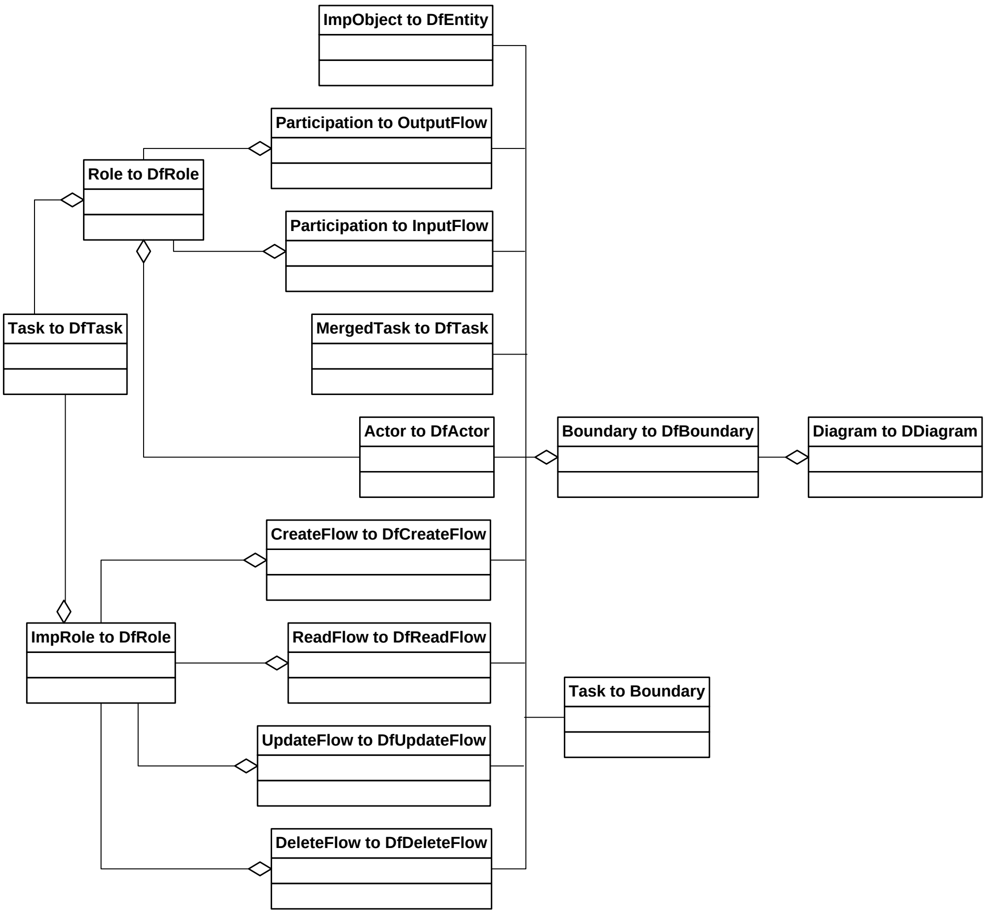



Figure 7.4: Req-to-Analysis: Task & Impact to (initial) DataFlow Model

The rest of the agents are subclasses of the *Translation* class (*Simple* rule), at the top level framework. These translators are:

- **TaskToBoundary:** The agent takes a *Task Model* composite *Task* as source and translates it into an equivalent *DataFlow DfBoundary*. This agent is interesting because it is controlled by a precondition that must be satisfied (*a task is composite*) to be translated into a dataflow boundary.

- **ParticipationToInputFlow:** The agent takes a *Task Model Participation* as source and translates it into an equivalent *DataFlow InputFlow*. The type of user interaction must be an *input* to satisfies the precondition of this rule. *ParticipationToInputFlow* consists of one subrule to translate endroles of each *Participation*, *RoleToDfRole* rule.

- **ParticipationToOutputFlow:** The agent takes a *Task Model Participation* as source and translates it into an equivalent *DataFlow OutputFlow*. The type of user interaction must be an *output* to satisfies the precondition of this rule. *ParticipationToOutFlow* consists of one subrule to translate the *Participation* endroles, namely, *RoleToDfRole* rule.

- **RoleToDfRole:** The agent takes a *Task Model* endrole *Role* of every *Participation* as source and translates it into an equivalent *DataFlow DfRole* of either *InputFlow* or *OutputFlow*. Any source *task* from the *Task Model* must be simple (not composite) in order to satisfy the precondition of this rule. *RoleToDfRole* consists of two subrules: *TaskToDfTask* and *ActorToDfActor*.
- **TaskToDfTask:** The agent takes the Task referenced by an end *Role* either in the *Task Model* or *Impact Model* and translates it into equivalent *DataFlow Model DfTask*.
- **ActorToDfActor:** The agent takes a *Task Model Actor* as source and translates it into an equivalent *DataFlow DfActor*.
- **CreateFlowToDfCreateFlow:** The agent takes a *Impact Model CreateFlow* as source and translates it into an equivalent *DataFlow CreateFlow*. This rule consists of one subrule to translate the endroles, *RoleToDfRole* rule.
- **DeleteFlowToDfDeleteFlow:** The agent takes a *Impact Model DeleteFlow* as source and translates it into an equivalent *DataFlow DeleteFlow*. This rule consists of one subrule to translate the endroles, *RoleToDfRole* rule.
- **UpdateFlowToDfUpdateFlow:** The agent takes a *Impact Model UpdateFlow* as source and translates it into an equivalent *DataFlow UpdateFlow*. This rule consists of one subrule to translate the endroles, *RoleToDfRole* rule.
- **ReadFlowToDfReadFlow:** The agent takes a *Impact Model ReadFlow* as source and translates it into an equivalent *DataFlow ReadFlow*. This rule consists of one subrule to translate the endroles, *RoleToDfRole* rule.
- **ImpObjectToDfEntity:** The agent takes a refered *Impact Model ImpObject* by either any *Role* in the *Impact Model* as source and translates it into an equivalent *DataFlow DfEntity*.
- **ImpRoleToDfRole:** The agent takes a *Task Model* endrole *Role* of every *Flow* as source and translates it into an equivalent *DataFlow DfRole* of the equivalent *DataFlow Flow*.

### 7.5.2 Translating the Information Model into the Data (Dependency) Model

Translating an *Information Model* into a *Data (Dependency) Model* is regarded as a *one-to-one* forward translation step. It takes one element as its source from an *Information Model*, producing an equivalent target element in the *Data Model*. This mapping step consists of 10 translating agents that are responsible for traversing nodes in the *Information Model*, applying a set of rules to perform the transformation. The following figure (7.5) shows the internal structure of the translator.

For this step, all agents are considered subclasses of the *Translation* class, at the top level framework. These translators are:

- **InfDiagramToDDiagram:** The agent takes an *Information Model Diagram* as source and produces an equivalent *Data Model DDiagram*. It is considered a root rule of this transformation step that consists of seven subrules.

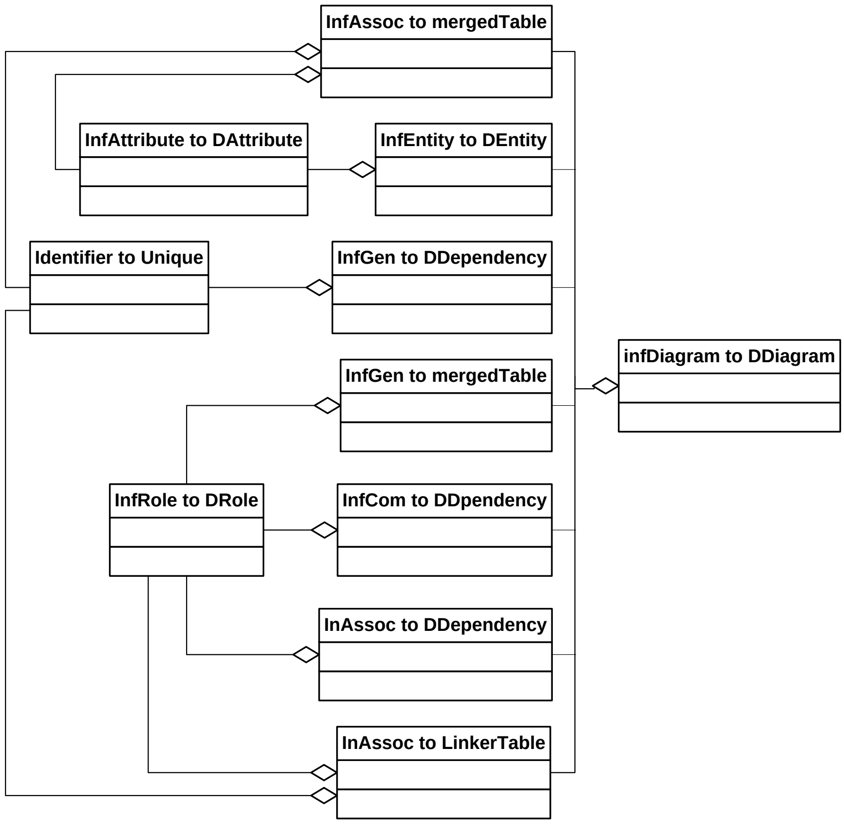



Figure 7.5: Req-to-Analysis: Information to Data Dependency Model

- **InfEntityToDEntity:** This agent provides a direct mapping as it takes an *Information Model InfEntity* as source and produces an equivalent *Data Model DEntity*. This agent is interesting because it can produce knew knowledge in the target model (e.g. manufactured identity), if needed.

- **InfAttributeToDAttribute:** The agent takes an *Information Model Attribute* as source and produces an equivalent *Data Model DAttribute*. Data types and additional specifications (e.g. size and null) might be generated precisely, if needed.

- **InfAssocToDDependency:** The agent takes an *Information Model Association* as source and produces an equivalent *Data Model Dependency*. This rule is restricted by a precondition, in which the source association must be *many-to-one*. It consists on only one subrule for translating endroles (*InfRoleToDRole*) of that association.

- **InfAssocToMergedDEntity:** The agent takes an *Information Model Association* as source and produces an equivalent *Data Model DEntity*. This rule is applicable for translating *many-to-many* associations only (precondition). It consists of two subrules to proceed the translation of attributes and identifiers for the generated target element. These subrules are: *InfAttributeToDAttribute* and *IdentifierToUnique*.

- **InfCompToDDependency:** The agent takes an *Information Model Composition* as source and produces an equivalent *Data Model Dependency*. A precondition is used to control this rule to be applied to **n**ot *Total Composition*. The *InfCompToDDependency* rule consists of one subrule for translating endroles (*InfRoleToDRole*) of that composition.

- **InfGenToDDependency:** The agent takes an *Information Model Generalisation* (*overlapping*) as source and produces an equivalent *Data Model Dependency*. This rule consists of one subrule for translating endroles (*InfRoleToDRole*) of that generalisation.

- **InfGenToMergedDEntity:** The agent takes an *Information Model Generalisation* as source and produces an equivalent *Data Model DEntity*. This rule is restricted to *disjoint Generalisation* only, via a precondition. It consists of two subrules to complete the mapping of attributes and identifiers for the generated entities. These subrules are: *InfAttributeToDAttribute* and *IdentifierToUnique*.

- **IdentifierToUnique:** The agent takes an *Information Model Identifier* as source and produces an equivalent *Data Model Unique* concept.

- **InfRoleToDRole:** The agent takes an *Information Model Role* as source and produces an equivalent *Data Model DRole*.

### 7.5.3 Translating (initial) DataFlow Model into Detailed DataFlow Model

Translating the (initial) *DataFlow Model* into the detailed one can be described as a *one-to-one* forward translation step. In particular, it is regarded as a model optimisation and evolving step that aims to decompose each business task that appears in the initial *DataFlow* into its collection of atomic sub-tasks after applying a number of decompositional rules to each *DfTask*. It is worth mentioning that the rule of transformation is presented in more detail further in Chapter 8.

This mapping step consists of 8 translating agents that are responsible for traversing the nodes in the initial *DataFlow Model*, applying a set of rules to perform the transformation. The following figure (7.6) shows the internal structure of the translator. For this step, all agents are considered to be subclasses of the *Translation* class, at the top level of the framework. These translators are:

- **ArcToArc:** The agent takes one of the (initial) DataFlow *Flow* elements, where its type is either *create, read, delete, input* or *output*, as source and produces an equivalent (detailed) DataFlow Model *Flow*.

- **ArcToArcs:** The agent takes an (initial) DataFlow *Flow* element, where its type is *update*, as source and produces an equivalent pair of *read* and *write* flows in the (detailed) DataFlow Model. This rule is applicable for *update* flows only.

- **ArcToDfActor:** The agent takes one of the (initial) DataFlow *Flow* elements, where its type is either *input* or *output*, as source and produces an equivalent (detailed) DataFlow Model *DfActor*.

- **ArcToDfObject:** The agent takes one of the (initial) DataFlow *Flow* elements, where its type is either *create, read, update* or *delete*, as source and produces an equivalent (detailed) DataFlow Model *DfObject*.

- **ArcToDfTask:** The agent takes one of the (initial) DataFlow *Flow* elements, where its type is either *create, read, delete, input* or *output*, as source and produces an equivalent (detailed) DataFlow Model *DfTask*.

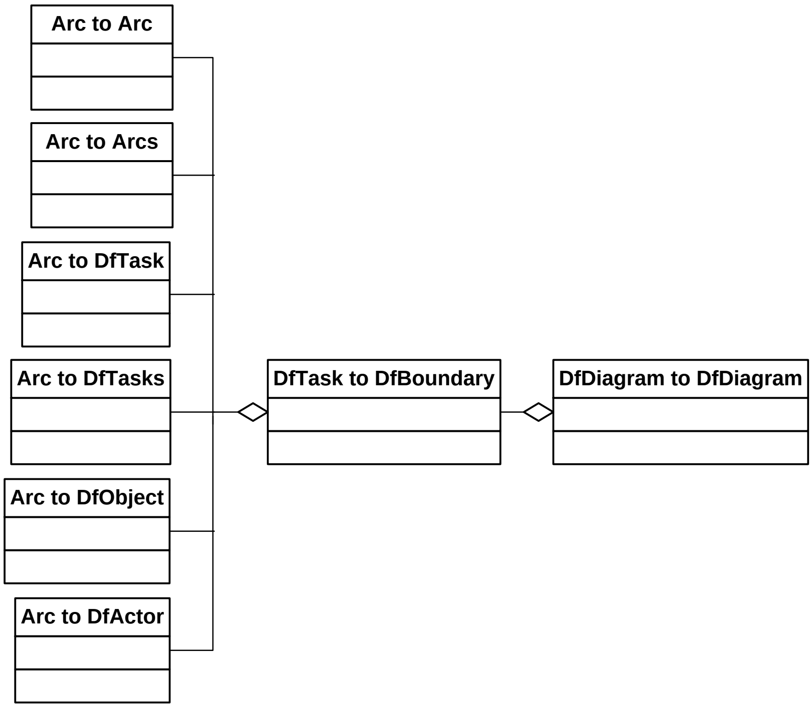



Figure 7.6: Req-to-Analysis: (initial) DataFlow to Detailed DataFlow Model

- **ArcToDfTasks:** The agent takes an (initial) DataFlow *Flow* elements, where its type is *update*, as source and produces equivalent (detailed) DataFlow Model *DfTask* elements.
- **DfdDiagramToDfdDiagram:** The agent takes a DataFlow Model *DfdDiagram* as source and produces an equivalent DataFlow Model *DfdDiagram*.
- **DfTaskToDfBoundary:** The agent takes a DataFlow Model *DfTask* as source and produces an equivalent DataFlow Model *DfBoundary*.

### 7.5.4 Translating DataFlow Model into (Screen) State Model

Translating a (detailed) *DataFlow Model* into a *State Model* can be defined as a forward translation step that takes an element as a source from a *DataFlow*, constructing an equivalent target element in the *Screen State* artefact. This step consists of 6 translating agents that traverse the tree structure of the *DataFlow Model*, applying a set of transformation rules to map a source element to a target one in the *State Model*. The following figure (7.7) illustrates the internal structure of the translator.

For this step, all translating agents are considered subclasses of the *Translation* class, at the top level framework. These translators are:

- **DfdDiagramToStDiagram:** The agent takes a DataFlow Model DfdDiagram as source and produces an equivalent *State Model StDiagram*. This root rule consists of two main subrules: DfdBoundaryToStBoundary and StBoundaryToStBoundary.
- **DfdBoundaryToStBoundary:** The agent takes a DataFlow Model *DfBoundary* as

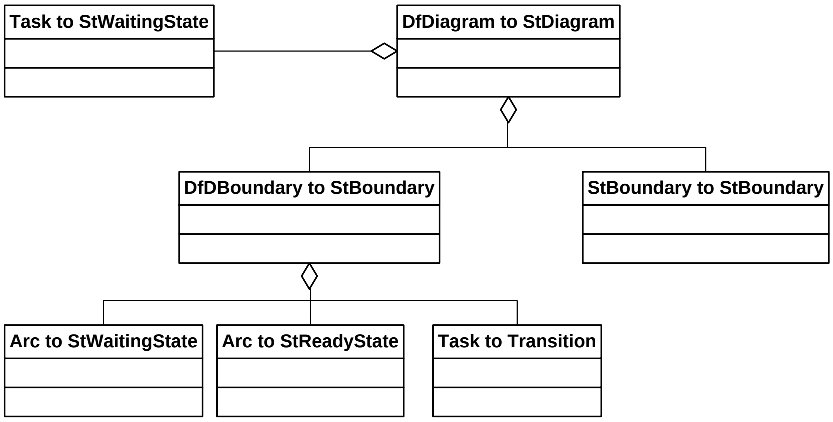



Figure 7.7: Req-to-Analysis: DFD to State Model

source and produces an equivalent *State Model StBoundary*. This rule consists of three subrules for translating the content the source model into equivalent *States* and *Transitions* (*ArcToStReadyState, ArcToStWaitingState* and *DfTaskToTransition*).

- **StBoundaryToStBoundary:** This is an *in-place* model modification step. The agent takes a *State Model StBoundary* as source and produces generate a number of *Error States*, which capture the failure scenarios of the business process, to be added to the current boundary *StBoundary*.

- **ArcToStReadyState:** The agent takes a *DataFlow Model Flow* as source and produces an equivalent *State Model ReadyState*. Each acceptable arc must be one of the CRUD flows only.

- **ArcToStWaitingState:** The agent takes a *DataFlow Model Flow* as source and produces an equivalent State Model *WaitingState*. Each acceptable arc must be either *input* or *output* flow. Otherwise, this rule is dismissed.

- **DfTaskToTransition:** The agent takes a *DataFlow Model DfTask* as source and produces an equivalent *State Model Transition*.

- **TaskToStWaitingState:** This rule takes every *composite Task* in the *Task Model* to produce a *waiting state*.

## 7.6 Analysis-to-Design Model Transformation Approach

According to the overall structure of the BUILD framework, presented in chapter 3, the *Analysis to Design Model Transformation* step is considered to be a second forward mapping step that translates some intermediate analysis models, resulting from the previous *Requirement to Analysis Model Transformation* process, into lower level design artefacts that describe the system at the design phase.

Similar to the previously discussed *Requirement to Analysis Model Transformation* step, a number of Java-translator programs are designed to implement the (hybrid) transformation rules for each agent. These translators are: *State-to-GUI* and *Data-DFD-to-DataBaseQuery*.

They parse the *XML* representation of the $\mu$ML analysis models, namely, DataFlow, Data Dependency, and State Model and produce a number of design models: GUI, DBQ and (optional) Code artefacts ready for the code generation phase.

### 7.6.1 Translating the State Model into the GUI Model

Translating *State Model* into a *GUI Model* is considered a direct forward translation step. It takes an element from a *State Model*, as its *source*, producing an equivalent target element in the *GUI Specification Model*. The internal structure of the translator is simple, in which the current version consists of 6 translating agents that are responsible for traversing *State Model*, applying transformation rules to perform the mapping to target elements in the *GUI Model*. The following figure (7.8) demonstrates the internal structure of the translator.

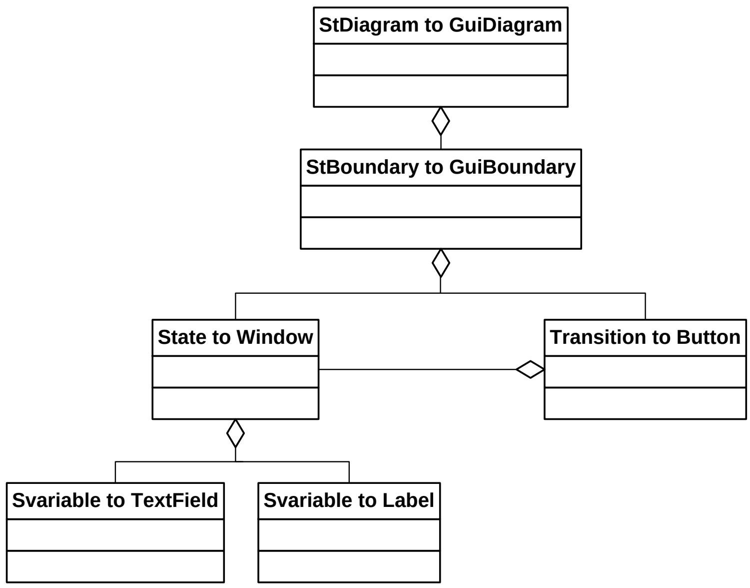



Figure 7.8: Analysis-to-Design: State to GUI Model

From this, all translating agents are considered subclasses of the *Translation* class, at the top level framework. These translators are:

- **StDiagramToGuiDiagram:** The agent takes a *State Model StDiagram* as source and produces an equivalent *GUI Model GuiDiagram*. This rule consists of one subrule for translating *boundaries*.
- **StBoundaryToGuiBoundary:** The agent takes a *State Model StBoundary* as source and produces an equivalent *GUI Model GuiBoundary*.
- **StateToWindow:** The agent takes a *State Model State* as source and produces an equivalent *GUI Model Window*. This rule consists of two subrules for translating widget controls for that window, namely, *SvariableToLabel* and *SvariableToTextfield*.
- **SvariableToLabel:** The agent takes a *State Model Variable* as source and produces an equivalent *GUI Model Label*.

- **SvariableToTextfield:** The agent takes a *State Model Variable* as source and produces an equivalent *GUI Model Textfield.*

- **TransitionToButton:** The agent takes a *State Model Transition* as source and produces an equivalent *GUI Model Button.*

### 7.6.2 Translating the DataFlow and Data Model into the DBQ Model

Described previously in Chapter 6, the DBQ model consists of two kinds of concepts: *data definition* and *query expression.* "This translation step has two source models, the DataFlow model and the Data Dependency model.

All agents that aim to produce data definition concepts in the target model are actually designed and implemented as *one-to-one* forward translation components. Each of these takes one element as its *source* from either a *DataFlow* or *Data Dependency Model*, producing an equivalent target element in the *Database Model* (DBQ).

On the other hand, agents that aim to produce query expression concepts in the target model are designed as *two-to-one* merging and *one-to-one* translation components. Each of these takes elements from one or both the *DataFlow* and *Data Model* as source and translates them into a target DBQ concept.

This mapping step consists of 12 translating agents, in total, that are responsible for traversing nodes in either *DataFlow* or *Data Dependency* or even both models, and applying a set of rules to perform the transformation. The following figure (7.9) shows the internal structure of the translator.

From this, some agents are considered to be subclasses of the *MergeTranslation* class, at the top level framework. These translators are:

- **CreateFlowToStoredProcedure:** The agent takes a *DataFlow Model CreateFlow* and an *Data Model DDiagram* as source and produces an equivalent *DBQ Model Procedure.* The argument of the target procedure is translated by *CulomnToArgument* subrule.

- **ReadFlowToStoredProcedure:** The agent takes a DataFlow Model *ReadFlow* and an Data Model *DDiagram* as source and produces an equivalent DBQ Model *Procedure.* This rule consists of one subrule for translating required arguments of that procedure.

- **DeleteFlowToStoredProcedure:** The agent takes a *DataFlow Model DeleteFlow* and an *Data Model DDiagram* as source and produces an equivalent *DBQ Model Procedure.* The argument of the target procedure is translated by *CulomnToArgument* subrule.

- **WriteFlowToStoredProcedure:** The agent takes a *DataFlow Model WriteFlow* and an *Data Model DDiagram* as source and produces an equivalent *DBQ Model Procedure.* This rule consists of two subrules for translating required arguments and local variables of that procedure.

The rest of agents are subclasses of the one-to-one *Translation* class, at the top level framework. These translators are:

- **DDiagramToSchema:** The agent takes a *Data Model DDiagram* as source and translates it into a DBQ equivalent *Schema.* This rule consists of seven subrules to proceed

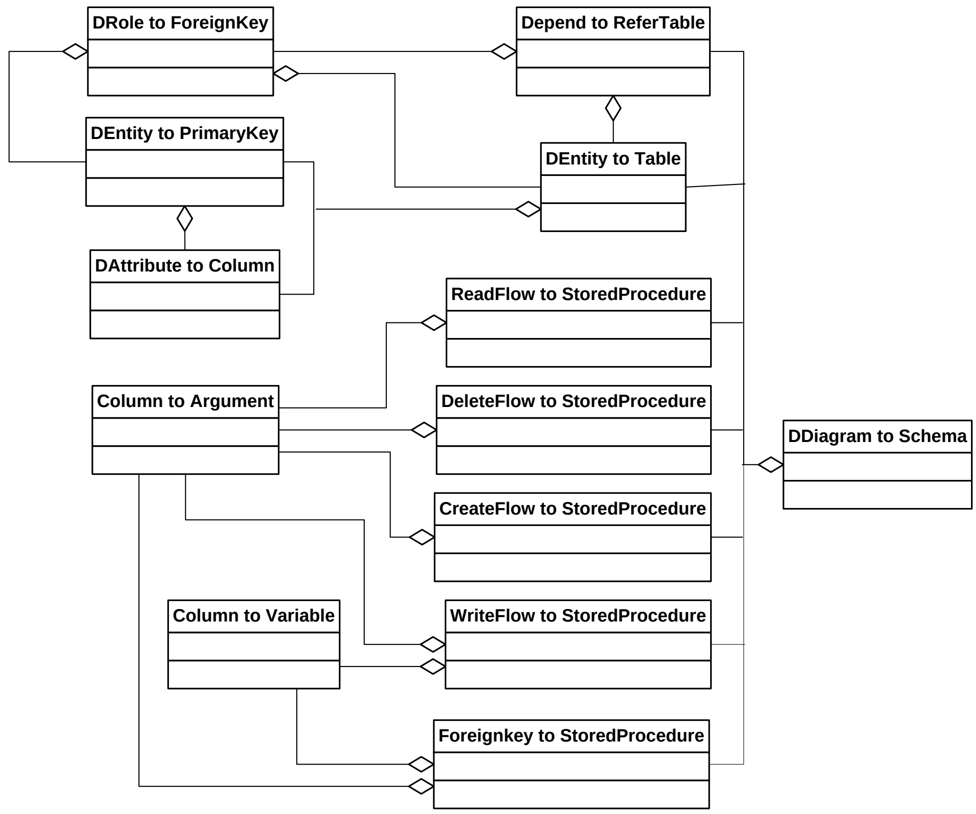



Figure 7.9: Analysis-to-Design: Data Dependency to Database and Query Model

with this translation step.

- **DEntityToTable:** The agent takes a *Data Model DEntity* as source and translates it into a DBQ equivalent *Table*. This rule consists of two subrules for translating attributes and primary keys of that generated table.
- **DEntityToPrimaryKey:** The agent takes a *Data Model DEntity* as source and translates it into a DBQ equivalent *PrimaryKey*.
- **DAttributeToColumn:** The agent takes a *Data Model DAtribute* as source and translates it into a DBQ equivalent *Column*.
- **DependToReferTable:** The agent takes a *Data Model Dependency* as source and translates it into a DBQ equivalent *Table*. This rule consists of one subrule for translating a foreignkey that referred to the generated table.
- **DRoleToForeignKey:** The agent takes a *Data Model DRole* as source and translates it into a DBQ equivalent *DfEntity*.
- **ForeignKeyToStoredProcedure:** The agent takes a *DBQ Model ForeignKey* as source and translates it into a DBQ equivalent *Procedure*. This rule consists of two subrules for translating required arguments and local variables of that procedure.
- **OopToDbType:** The agent takes an OOP data type and translates it into an equivelant target database datatype.

# 7.7 Alternative Model Transformation Steps

In this section, the structure of the optional translation steps are discussed. The current version of BUILD has two additional translation steps that are not a part of the information systems development process, as currently implemented. These steps are: translating the *Impact Model* into the (initial) *Information Model* and translating *DataFlow and DBQ Models* into the *Code Model*.

## 7.7.1 Translating the Impact Model into the (Initial) Information Model

Translating an *Impact Model* into an (initial) *Information Model* in is an alternative *one-to-one* translation step toward generating a complete data model of a system. It takes *object* and *impact* elements from an *Impact Model*, as its *source*, generating an equivalent *Information Model Entity* and *Associations* as target elements. This translation step consists of 6 *Java* agents, in total, that are responsible for traversing nodes in an *Impact Model*, and applying a set of rules to perform the dirct translation between elements and predict the association between them. The following figure (7.10) shows the internal structure of this translator.

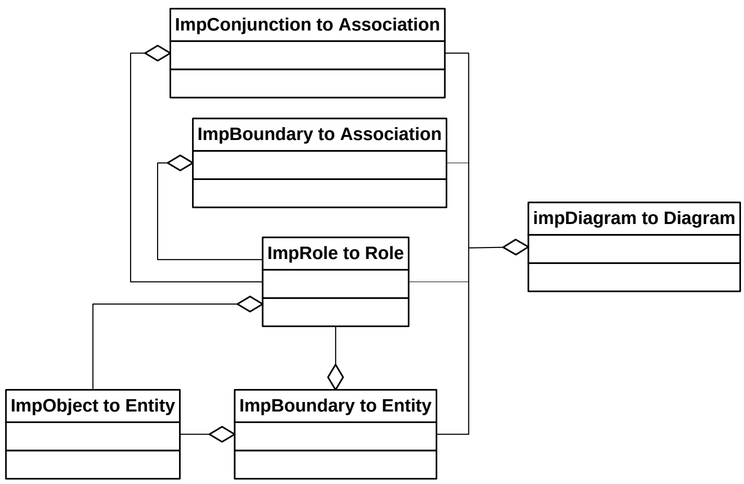



Figure 7.10: Alternative Transformation: Impact to Information Model

From this, all translating components are regarded subclasses of the *Translation* class, at the top level framework. These translators are:

- **ImpBoundaryToDiagram:** The agent takes a *Impact Model ImpDiagram* as source and produces an equivalent *Information Model Diagram.*
- **ImpBoundaryToEntity:** The agent takes a boundary (*ImpBoundary*) from the *Impact Model* as source; each enitity, within that boundary, is mapped into an equivalent enitity (*Entity*) in the *Information Model.* This rule consists of one subrule (*ImpObjectToEntity*) that continues the translation.
- **ImpBoundaryToAssociation:** The agent takes a *Impact Model ImpBoundary* as source; each flow, in that boundary, is translated into an equivalent *Information Model Association.* This rule consists of a single subrule (*ImpRoleToRole*) that is responsible for map-

ping the types of endroles of each flow into the quivalent ones in the target *Information Model*.

- **ImpConjunctionToAssociation:** The agent takes a *Impact Model ImpBoundary* that has one or more conjunction *ImpConjunction* as source; for each conjunction within that boundary, the rule generates equivalent *Information Model Associations*. This rule also consists of one subrule (*ImpRoleToRole*) that is responsible for mapping endroles of each conjunction into the quivalent ones in the *Information Model*.

- **ImpObjectToEntity:** The agent takes a *Impact Model ImpObject* as source and produces an equivalent *Information Model Entity*.

- **ImpRoleToRole:** The agent takes a *Impact Model ImpRole* as source and produces an equivalent *Information Model Role*.

It is worth saying that this translation agent is implemented and used later in Chapter 9 for generating the *Information Model* in a case study. The rule of transformation is presented later with further detail in Chapter 8.

### 7.7.2 Translating the DataFlow and DBQ Model into the Code Model

Translating a *Database and Query* and *DataFlow Model* into an (OO) *Code Model* is considered to be a forward merging step. It takes two elements as its *source* from a *DBQ* and *DataFlow Model*, producing an equivalent target element in the *Code* model. Finalising the clean version of this translation agent, which supports all possible OO code features, is not implemented completely at the date of writing this thesis.

The agent consists of 15 translating agents that are responsible for traversing both *Database and Query* and *DataFlow Models*, applying a set of transformation and merging rules to perform the mapping between their elements and the target elements in the *Code Model*. For this step, some agents are considered to be subclasses of the *MergeTranslation* class, at the top level framework. These translators are:

- **DDiagramToCDiagram:** The agent takes a *DBQ Schema* and DFD Diagram as source and produces an equivalent *Code Model Diagram* element. This rule consists of three main subrules: *TableToClass, DfBoundaryToClass* and *StoredProcedureToMethod* for generating entity classes, process classes and methods respectively.

The rest of the agents are subclasses of the one-to-one *Translation* class, at the top level framework. These translators are:

- **DfBoundaryToClass:** The agent takes a *DataFlow DfBoundary* as source and translates it into a *Code Model* equivalent *Clazz*. The generated class represents a process class (business task class). This rules consists of two main subrules: *DfObjectToAttribute* and *DfActorToAttribute* for generating all required attributes (fields) of that process class.

- **ColumnToCArgument:** The agent takes a *DBQ Model Column* as source and translates it into a *Code Model* equivalent *Argument* element.

- **ColumnToVariable:** The agent takes a *DBQ Model Column* as source and translates it into a *Code Model* equivalent *Variable*.

- **DbToOopType:** The agent takes a *DBQ Model* datatype of every element as source and translates it into an acceptable OO datatype that is equivalent to the source one.
- **OopToDbType:** The agent takes a *Code Model* datatype of every element as source and translates it into an acceptable relational database datatype.
- **DfActorToAttribute:** The agent takes a *DataFlow DfActor* as source and translates it into a *Code Model* equivalent *Attribute*. The generated attributes are a member of a class that is represent a business task (not a business entity).
- **DfObjectToAttribute:** The agent takes a *DataFlow DfObject* as source and translates it into a *Code Model* equivalent *Attribute*. The generated attributes are a member of a class that is represent a business task (not a business entity).
- **AttributeToIdentifier:** The agent takes a recently generated *Code Model Attribute* as source and translates it into a Code Model equivalent *Identifier*, which might be used in such a method or a constructor.
- **StoredProcedureToMethod:** The agent takes a *DBQ Model Procedure* as source and translates it into an equivalent *Code Mode's Method*. This rule consists of two subrules: *proArgumentToMethArgument* and *proArgumentToMethResult* for translating its input arguments and the type of its return result.
- **ProArgumentToMethArgument:** The agent takes an argument of *DBQ Model Procedure* as source and translates it into a *Code Mode's Method* equivalent *Argument*.
- **ProArgumentToMethResult:** The agent takes an result (return value) of *DBQ Model Procedure* as source and translates it into a *Code Mode's Method* equivalent *return* value.
- **TableToArgument:** The agent takes a *DBQ Model Table* as source and translates it into an equivalent *Code Model's Argument*. This generated argument is used to pass *object* to method and/or constructor.
- **TableToClass:** The agent takes a *DBQ Table* as source and translates it into a *Code Model* equivalent *Clazz*. The generated class represents a business entity class. It is consists of two subrules: *ColumnToAttribute* and *TableToConstructor* for translating attributes and constructor for that entity class.
- **TableToConstructor:** The agent takes a *DBQ Table* as source and translates it into a *Code Model* equivalent *Constructor* for each generated business entity class.
- **VariableToIdentifier:** The agent takes a recently generated *Code Model Variable* as source and translates it into a *Code Model* equivalent *Identifier*, which might be used in such a method or a constructor.

## 7.8 The Implementation of $\mu$ML Models

*Java* is adopted both for constructing the model and for the transformation technology the model and the transformation technology in BUILD. It is a widely known OO programming language and does not force any further conceptual load on developers, unlike rule-based transformation languages that require a a steep learning curve before introducing a new translation agent or a new system aspect into the framework. Models in BUILD are abstract syntax trees (ASTs), built in the same programming language (*Java*).

Two types of *Java Packages* are used to implement $\mu$ML models, namely, *a core package* and several *concrete packages*. The current version of BUILD has one *core package* that contains several *Java* classes for representing core implementation of the $\mu$ML metamodel elements (see Table 7.1). On the other hand, the framework contains seven *concrete packages* for representing language concepts of each $\mu$ML model (information system view), each package has a number of classes for defining these concrete concepts (Table 7.2).

| **Package** | **Classes** |
|---|---|
| **mde.model** | Element, Node, Model, Type, Arc, Expression, Widget |

Table 7.1: Java Package of the Core $\mu$ML Elements

At the core metamodel level, a concept might inherit from another one. The following listing (7.1) illustrates the representation of *Node.java* class. It interits from the core *Element* class, and has constructors, attributes and a number of get/set methods. The remaining of concepts are defined using the same strategy.

Listing 7.1: Construction of Node Concept

```
public class Node extends Element {

    protected String name;

    public Node() {
    }

    public Node(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public Node setName(String name) {
        this.name = name;
        return this;
    }
}
```

As it is mentioned before, a number of *concrete packages* are used to represent the concept of each $\mu$ML model. The following table (Table 7.2) demonstrates $\mu$ML concepts for each *concrete package* that are defined using *Java classes*.

| **Package** | **Classes** |
|---|---|
| **mde.task.model** | Task, Actor, Boundary Diagram, Generalisation, Composition, Participation, Role |

| | |
|---|---|
| **mde.impact.model** | ImpTask, ImpObject, ImpBoundary, ImpDiagram, ImpDeleteFlow ImpConjunction, ImpCreateFlow, ImpReadFlow, ImpUpdateFlow, , ImpRole |
| **mde.information.model** | Entity, Association, Attribute, Role Diagram, Generalisation, Composition |
| **mde.state.model** | State, Action, Variable, StRole, Transition, StDiagram, StBoundary |
| **mde.gui.model** | Window, GuiDiagram, GuiBoundary, Button, Variable, GuiRole, Event, Label, Textfield |
| **mde.data.model** | DEntity, DDependency, DAttribute, DDiagram, DRole |
| **mde.dataflow.model** | DfEntity, DfActor, DfTask, DfDiagram, DfBoundary, DfCreateFlow, DfReadFlow, DfUpdateFlow, DfDeleteFlow, DfInputFlow, DfOutputFlow, DfWriteFlow, DfRole |

Table 7.2: Java Package of the main $\mu$ML Concrete Elements

Elements of memory models are indexed by their *names*. A *Java LinkedHashMap* object is used to store memory elements in the same order as they appear in the model, because some transformation rules need to check the order of occurrence of some elements to make a mapping decision. The following listing (7.2) demonstrates the implementation of the Information Model *Entity.java* concrete class:

Listing 7.2: Construction of the Entity element in the Information Model

```
public class Entity extends Type {

    private Map<String, Attribute> attributes;

    public Entity() {
        attributes = new LinkedHashMap<String, Attribute>();
    }

    public Entity(String name) {
        super(name);
        attributes = new LinkedHashMap<String, Attribute>();
    }

    public List<Attribute> getAttributes() {
        return new ArrayList<Attribute>(attributes.values());
    }

    public Entity addAttribute(Attribute datt) {
        attributes.put(datt.getName(), datt);
        return this;
    }

    public Attribute getAttribute(String name) {
        return attributes.get(name);
    }
}
```

## 7.9 The Implementation of the Transformation Rules

In BUILD, the transformation rules are classified into two types, *top level* and *concrete* rule. The top level rules are implemented as **abstract** classes in the top level framework. The current version of BUILD has two **abstract** classes, namely, *Rule* and *MergeRule*. At the concrete level, each rule is implemented as a concrete *java* class that inherits either from *Translation* or *MergeTranslation* **abstract** classes at the top level framework. The following sections present examples of an abstract and concrete transformation rule.

### 7.9.1 Example of Top-Level Rule Implementation

The **abstract** *Translation* class is the ancestor of all translation rules, implemented in the concrete level, mapping from one *Source* element to one *Target* element. Every concrete forward translation rule must inherit from *Translation*, which is responsible for maintaining the shared Context of completed work. Figure 7.3 demonstrates the content of the *Translation.java* **abstract** class that expresses the generic translation rule at the top-level framework.

As mentioned earlier, the method *translate()* first checks whether a translation already exists for the given source and, if so, it returns the corresponding target for that source. Otherwise it invokes the **abstract** method *doTranslate()* and stores the translated target element in the *context*.

The shared Context is used by the *hasTranslation ()* method to check whether a translation exists for a particular source element. When such a translation exists, the *getTranslation()* method is used to retrieve the translated target element for a given source, otherwise the method returns the *null* value. In contrast, when such a translation does not exist, the method *putTranslation()* stores a target element as the translation for a given source one and returns it.

Listing 7.3: Abstract root Translation class

```java
public abstract class Translation<Source, Target> extends Rule<Source> {

    private Context context;

    public Translation(Context context) {
        this.context = context;
    }

    public boolean hasTranslation (Source source) {
        Map<Object, Object> map = context.lookup(this);
        return map.containsKey(source);
    }

    protected Target getTranslation(Source source) {
        Map<Object, Object> map = context.lookup(this);
        return (Target) map.get(source);
    }

    public Target putTranslation(Source source, Target target) {
        Map<Object, Object> map = context.lookup(this);
        map.put(source, target);
        return target;
    }

    protected abstract Target doTranslate(Source source);
```

```

    public Target translate(Source source) {
        if (hasTranslation(source))
            return getTranslation(source);
        else
            return putTranslation(source, doTranslate(source));
    }

}
```

### 7.9.2 Example of Concrete Rule Implementation

The transformation rule for translating *Data Model's DEntity* into *DBQ's Table* is considered in this section to exemplify its implementation and illustrate how the rule invokes another one in an imperative style. Figure 7.4 represents the content of the *DEntityToTable.java* class that expresses the translation rule for generating *DBQ table* from a generated *Data Model*.

Listing 7.4: Construction of the Entity element in the Information Model

```java
public class DEntityToTable extends Translation<DEntity, Table> {

    private DAttributeToColumn dAttributeToColumn;
    private DEntityToPrimaryKey dEntityToPrimKey;

    public DEntityToTable() {
        this(new Context());
    }

    public DEntityToTable(Context context) {
        super(context);
        dAttributeToColumn = new DAttributeToColumn(context);
        dEntityToPrimKey = new DEntityToPrimaryKey(context);
    }

    public boolean accept(DEntity dentity) {
        return true;
    }

    public Table doTranslate(DEntity dentity) {
        Table table = new Table();
        table.setName(dentity.getName());
        if (dEntityToPrimKey.accept(dentity)) {
            PrimaryKey primary = dEntityToPrimKey.translate(dentity);
            if (primary.isSynthetic())
                table.addColumn(primary.getColumn());
            table.addPrimaryKey(primary);
        }
        for (DAttribute dattr : dentity.getDAttributes()) {
            if (dAttributeToColumn.accept(dattr))
                table.addColumn(dAttributeToColumn.translate(dattr));
        }
        return table;
    }
}
```

The concrete *DEntityToTable* class inherits from the *Translation* class at the top level. The rule has no preconditions to be fired, as it seen in the body of *accept()* method. Two subrules are declared declaratively, without taking into account their order of execution, namely, *DAttributeToColumn* and *DEntityToPrimaryKey*.

The body of the *DEntityToTable*, which represents exactly how the target element is developed, is expressed imperatively as a body of the concrete *doTranslate(...)* method. The following algorithm demonstrates how to translate *Data Model DEntities* into *DBQ Tables*:

```
Result: Table
initialise DAttributeToColumn object;
initialise DEntityToPrimaryKey object;
while DEntityToTable accepts a new source element do
    translate the source element (DEntity);
    if DEntityToPrimaryKey accept the source then
        generate a primary key from the source;
        add the generated key to the generated table;
    end
    forall the DAttribute in the current DEntity do
        if DAttributeToColumn accept the source then
            translate the current DAttribute from the source;
            add the generated Column to the generated table;
        end
    end
    go back to translate the next DEntity in the source model;
end
```

**Algorithm 1:** Translating *Data Model DEntities* into *DBQ Tables* (Default)

In order to illustrate the degree of modularity of the designed agents (*DAttributeToColumn* and *DEntityToPrimaryKey*), they can be reused in the opposite order to produce the same output result. From that it can be concluded that these two agents are completely independent. The following algorithm shows an alternative order to invoking the subrules than the order shown in algorithm 1.

```
Result: Table
initialise DAttributeToColumn object;
initialise DEntityToPrimaryKey object;
while DEntityToTable accepts a new source element do
    translate the source element (DEntity);
    forall the DAttribute in the current DEntity do
        if DAttributeToColumn accept the source then
            translate the current DAttribute from the source;
            add the generated Column to the generated table;
        end
    end
    if DEntityToPrimaryKey accept the source then
        generate a primary key from the source;
        add the generated key to the generated table;
    end
    go back to translate the next DEntity in the source model;
end
```

**Algorithm 2:** Translating *Data Model DEntities* into *DBQ Tables* (Alternative)

## 7.10 Brief Overview of the Code Generation Framework

As previously presented in Chapter 3, the *Code Generation* step is considered the final forward mapping step, within BUILD framework, that generates executable code from a number of low-level platform-independent design models. The input models for this phase are: the *GUI* and *DBQ Model*, whereas, the output is a platform-specific executable OOP code and a relational SQL schema script. The current version of BUILD is able to generate a *Java Swings* application with a *JDBC* connection to a MySQL back-end database system is also generated from BUILD, producing a runnable 2-Tier Information System.

As a proof of concept, two domain-specific generators, namely, a *Java Swing and MySQL* generator, were designed. The overall architecture of the BUILD Code Generation Framework is described as a two levels of code generator agents. The top-level framework consists of a number of *abstract* classes, or Java agents, for holding common features of all sub-generators that are platform-specific for a target environment, such as *MySQL*.

## 7.11 Code Generation Approach (Design-to-Code)

In order to achieve a complete code generation facility, a number of Java-generator components are designed to implement the (imperative) mapping rules for each generator agent. These generators are: *State-to-GUI* and *DataFlow-DataModel-to-Database*. They accept as source the in-memory models produced by earlier transformation steps and produce executable code of each input artefact providing an executable code for a target system.

As the approach aims to generate code for different working environments, two layers of java classes are considered in the internal structure of each generator: abstract and concrete classes. The layer of abstract classes is constructed on the top of all groups of sub-generators, for a particular IS tier, to hold and share the common behaviours of the widely-known relational database vendors and OOP languages, which might be duplicated within different kinds of generators. Figure 7.11 demonstrates the sub-generators of the *Java Swing GUI* code generation framework.

For instance, when constructing the data tier, generating back-end databases in MySQL, Oracle, and other database system is possible when a relevant version of database generator exists. The following figure (Figure 7.12) illustrates sub-generators for generating a MySQL database system. These specific generators are implemented separately and grouped as concrete classes in *Java*.

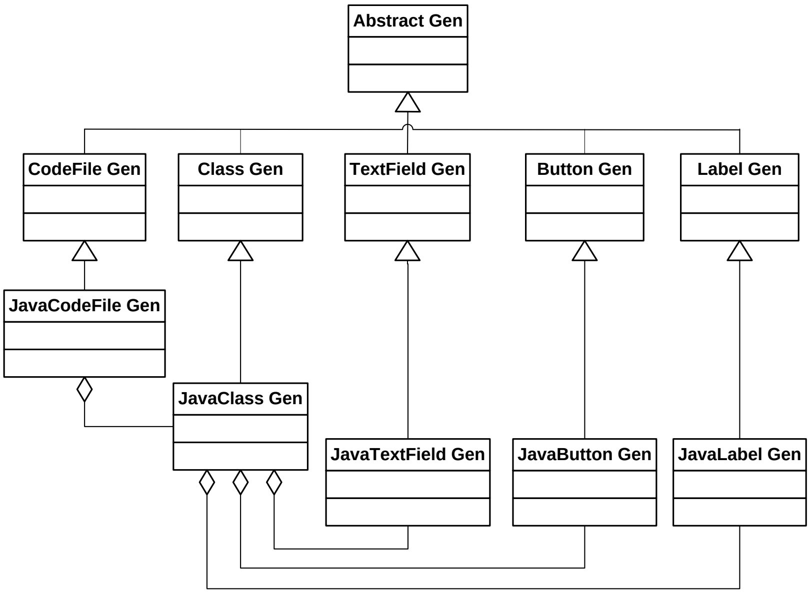



Figure 7.11: The Architecture of the Java Swings UI Generator

For each concept in the target model, a specific agent (Java generator) is introduced. The transformation rules are implemented imperatively as methods within each agent. The following listing (7.5) shows an example of a method in *MySQLFieldGenerator* class for generating fields of a particular *MySQL* table.

Listing 7.5: A method for generating fields of a MySQL table

```
public void writeField() throws TreeException
{
    if(reservedWords.contains(((Column) model).getName()))
        write("  "+((Column) model).getName()+"1");
    else
        write("  "+((Column) model).getName());
    write(" " + convertType(getType()));

    if(!(convertType(getType()).equalsIgnoreCase("BOOLEAN") ||
            convertType(getType()).equalsIgnoreCase("DATE") ||
              convertType(getType()).equalsIgnoreCase("DOUBLE")))

        write("("+((Column) model).getSize()+")");

    Table table = (Table) getOwner().getModel();
    PrimaryKey pk = table.getPrimaryKey();

    for(Column col : pk.getColumns()) {
        if(col.getName().equalsIgnoreCase(getName()))
            write(" NOT NULL");
    }   }
```

## 7.12 Outlook on the Chapter

The chapter has discussed the overall architecture and mechanism of the transformation approach in BUILD. The transformation framework consists of two levels, an abstract *top level* and *concrete bottom level*. In the top level, two root rules (*Java classes*) are designed to implement the various types of mapping rules within the approach (*e.g. one-to-one translation, two-to-one merging rule*). Furthermore, for each compositional layer of transformation, the architecture of all translation agents at the *concrete level* is discussed in-depth in this chapter. Independent rules for each translation step are highlighted, showing their roles in the transformation approach.

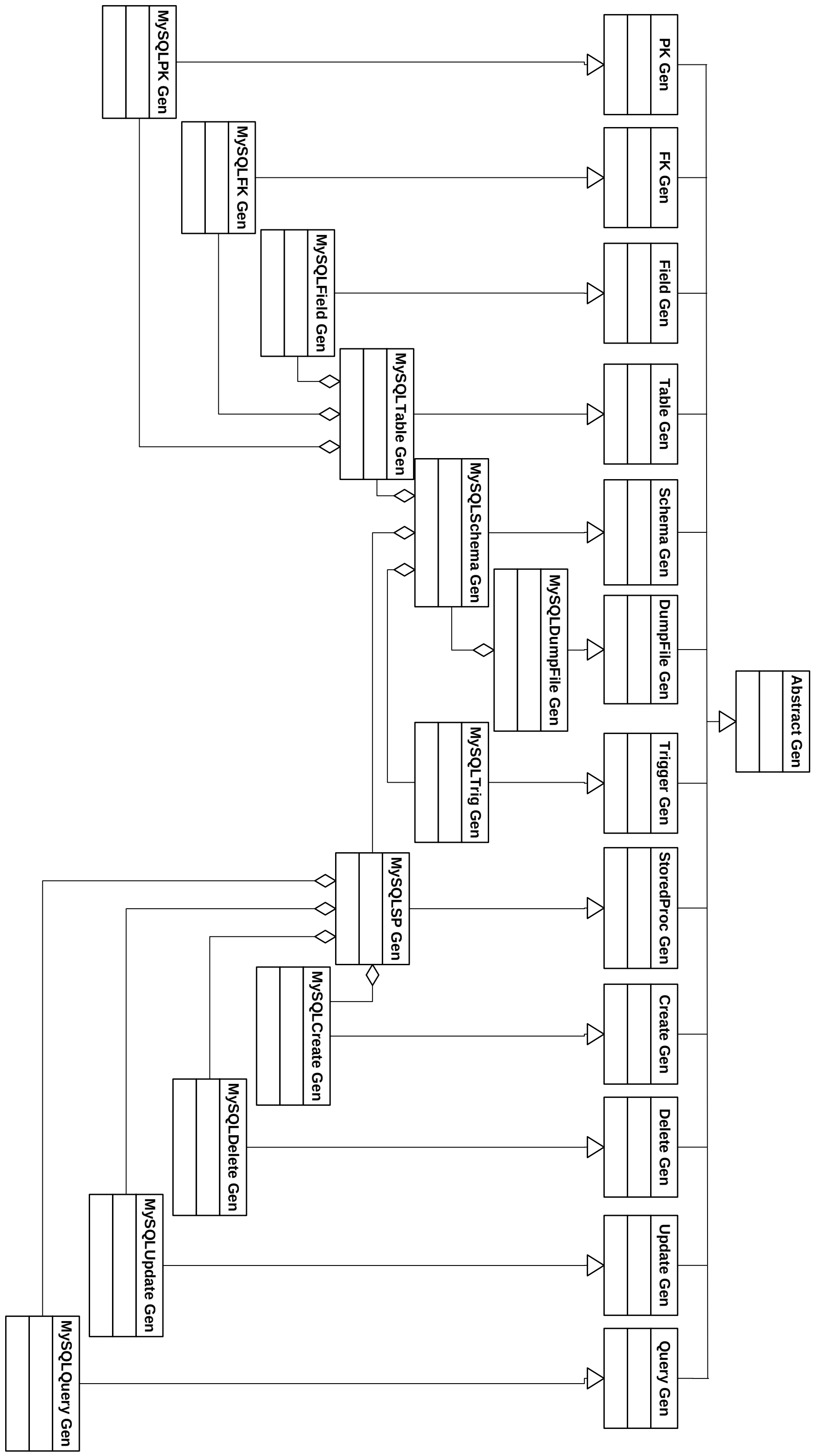



Figure 7.12: The Architecture of the (MySQL) Database Generator

# 8

# The Rules of Model Transformation

"Simplicity is the soul of efficiency"

Austin Freeman

## 8.1 Context

This chapter presents and discusses the transformation rule for each translation step at the concrete level of the framewok. The sections are divided based on the transformations between the different development phases in BUILD, which mainly are *Requirement to Analysis phase* and *Analysis to Design phase*. For each target model, the rule that is used for deriving each target element is presented. The First-Order Predicate Logic (FOPL) statements are used to express and formalise the transformation rules.

## 8.2 Transformation Rules at the Requirement to Analysis Phase

### 8.2.1 Translating Task and Impact Models into (initial) DataFlow

While each external interaction between tasks and actors is expressed in the *Task Model*, and the internal interaction between the same tasks and system objects is described in the *Impact Model*, the *DataFlow Model* brings together both interactions in one diagram as a result of merging equivalent task from the *Task* and *Impact Model*.

#### 8.2.1.1 DataFlow Diagram

Equivalent *Diagram* elements in the *Task* and *Impact Model* are mapped into a *DataFlow* one.

$$mr : Diagram \times ImpDiagram \rightarrow DfDiagram$$

$$\begin{aligned}
&\forall\ d1 : Diagram,\ \forall\ b1 : Boundary, m1 : TaskModel \\
&\forall\ d2 : ImpDiagram,\ \forall\ b2 : ImpBoundary, \\
&m2 : ImpactModel \bullet ((b1 \in d1) \wedge \\
&(d1 \in_m m1) \wedge (b2 \in d2) \wedge (d2 \in_m m2) \\
&\wedge NameOf(d1, d2) \wedge NameOf(b1, b2)) \\
&\longrightarrow \\
&\exists!\ d3 : DfDiagram,\ b3 : DfBoundary, \\
&m3 : DataFlowModel \bullet ((d3 \in_m m3) \wedge (b3 \in d3) \\
&\wedge (NameOf(d1, d3) \wedge NameOf(d2, d3)) \\
&\wedge (b3 = mr(b1, b2)))
\end{aligned}$$

#### 8.2.1.2 DataFlow Task

Each *DfTask* corresponds to the merge of both a *Task Model Task* and it is equivalent *ImpTask* in the *Impact Model*. This translation is performed by the *MergeTaskToDfTask* agent. Therefore, the rule can be stated as: *for each two equivalent tasks in both Task and Impact Model, there exist a task (DfTask) in the generated DataFlow Model.* The main translation rule for this translator can be expressed in logic as:

$$tr : Task, ImpTask \rightarrow DfDTask$$

$$\begin{aligned}
&\forall\ t1 : Task, m1 : TaskModel, \\
&\forall\ t2 : ImpTask, m2 : ImpactModel \\
&\bullet ((t1 \in_m m1) \wedge (t2 \in_m m2) \wedge NameOf(t1, t2)) \\
&\longrightarrow \\
&\exists!\ t3 : DfTask, m3 : DataFlowModel \\
&\bullet ((t3 \in_m m3) \wedge (NameOf(t1, t3) \\
&\wedge NameOf(t2, t3))
\end{aligned}$$

#### 8.2.1.3 DataFlow Boundary

Composite tasks in the *Task Model* are treated differently, in that they are translated into a *DataFlow Boundary*, containing its sub-tasks. This is handled by the TaskToBoundary agent. Thus, the transformation rule is: *For each composite task in the Task Model, there exists a logical boundary in model of the DataFlow Model.*

$$tr : Task \rightarrow DfBoundary$$

$$\begin{aligned}
&\forall\ t : Task, c : Composition, m1 : TaskModel \\
&\bullet ((t, c \in_m m1) \wedge (Whole(c, t))) \\
&\longrightarrow \\
&\exists!\ b : DfBoundary, m2 : DataFlowModel \\
&\bullet ((b \in_m m2) \wedge NameOf(t, b))
\end{aligned}$$

In addition to this, a direct mapping rule exists: *For each equivalent boundary in the Task Model and Impact Model, there exists a logical boundary in the DataFlow Model.* This can be expressed in FOPL as:

$$mr : Boundary, ImpBoundary \;\rightarrow\; DfBoundary$$

$$\begin{aligned}
&\forall\, b1 : Boundary, \forall\, p : Participation,\\
&\forall\, t1 : Task,\ \forall\, a1 : Actor, m1 : TaskModel\\
&\bullet\ ((t1, p, a1\ \in_m\ b1)\ \wedge\ (b1\ \in_m\ m1)),\\
&\forall\, b2 : ImpBoundary, \forall\, ar : Arc, \forall\, obj : ImpObject,\\
&\forall\, t2 : ImpTask, m2 : ImpactModel\\
&\bullet\ ((t2, obj, ar\ \in_m\ b2)\ \wedge\ (b2\ \in_m\ m2))\\
&\longrightarrow\\
&\exists!\ b3 : DfBoundary, t3 : DfTask, a2 : DfActor,\\
&f : Flow, e : DfEntity, m3 : DataFlowModel\\
&\bullet\ ((t3, a2, f, e\ \in_m\ b3)\ \wedge\ (b3\ \in_m\ m3)\\
&\wedge\ (NameOf(b1, b3)\ \wedge\ NameOf(b2, b3)\ \wedge\\
&(t3\ =\ mr(t1, t2))\ \wedge\ (a2\ =\ tr(a1))\ \wedge\\
&(f\ =\ tr(p))\ \wedge\ (f\ =\ tr(ar))))
\end{aligned}$$

#### 8.2.1.4 DataFlow Entity

Each *DfEntity* is mapped directly to an *Impact Model* object (*ImpObject*). This translation is performed by the *ImpObjectToDfEntity* agent. The rule of this translation is: *for each object in the Impact Model, there exist a DataFlow Entity* in the generated DataFlow Model. It can be expressed in FOPL as:

$$tr : ImpObject \;\rightarrow\; DfEntity$$

$$\begin{aligned}
&\forall\, obj : ImpObject, m1 : ImpactModel\ \bullet\ (obj\ \in_m\ m1)\\
&\longrightarrow\\
&\exists!\ ent : DfEntity, m2 : DataFlowModel\\
&\bullet\ ((ent\ \in_m\ m2)\ \wedge\ NameOf(obj, ent))
\end{aligned}$$

#### 8.2.1.5 DataFlow Actor

Each *DfActor* is translated simply to a *Task Model* actor (*Actor*). This translation is performed by the *ActorToDfActor* agent. The rule of this translation is: *for each actor in the Task Model, there exist a DataFlow Entity* in the generated DataFlow Model. It can be expressed in FOPL as:

$$tr : Actor \;\rightarrow\; DfActor$$

$$\begin{aligned}
&\forall\, a1 : Actor, m1 : TaskModel\ \bullet\ (a1\ \in_m\ m1)\\
&\longrightarrow\\
&\exists!\ a2 : DfActor, m2 : DataFlowModel\\
&\bullet\ ((a2\ \in_m\ m2)\ \wedge\ NameOf(a1, a2))
\end{aligned}$$

#### 8.2.1.6 DataFlow Flows

Each *Task Model Participation* is checked first to see whether or not it is an *Input* or an *Output*. This check is carried out by *BoundaryToDfBoundary* merge translation agent. Based on the kind of *Participation*, the decision to translate it into *DataFlow DfInputFlow* or *DfOutputFlow* is made. The translation is perform by either *ParticipationToDfInputFlow* and *ParticipationToDfOutputFlow* component. From that, two basic rules can be extracted:

1- *For each Input participation in the Task Model, there exist an Input flow in the DataFlow Model.*

$$
\begin{array}{rl}
(DfInputFlow \ <: \ Arc) & \\
tr : Participation \ \rightarrow \ DfInputFlow & \\
\forall \ p : Participation, (r1, r2) : Role, m1 : TaskModel & \\
\bullet \ ((p, r1, r2 \ \in_m \ m1) \ \wedge \ (Input(p))) & \\
\longrightarrow & \\
& \exists! \ f : DfInputFlow, (r3, r4) : DfRole, \\
& m2 : DataFlowModel \ \bullet \ ((r3 \ = \ tr(r1)) \\
& \wedge \ (r4 \ = \ tr(r2)) \ \wedge \ (f, r3, r4 \ \in_m \ m2))
\end{array}
$$

2- *For each Output participation in the Task Model, there exist an Output flow in the DataFlow Model.*

$$
\begin{array}{rl}
(DfOutputFlow \ <: \ Arc) & \\
tr : Participation \ \rightarrow \ DfOutputFlow & \\
\forall \ p : Participation, (r1, r2) : Role, m1 : TaskModel & \\
\bullet \ ((p, r1, r2 \ \in_m \ m1) \ \wedge \ (Output(p))) & \\
\longrightarrow & \\
& \exists! \ f : DfOutputFlow, (r3, r4) : DfRole, \\
& m2 : DataFlowModel \ \bullet \ ((r3 \ = \ tr(r1)) \\
& \wedge \ (r4 \ = \ tr(r2)) \ \wedge \ (f, r3, r4 \ \in_m \ m2)
\end{array}
$$

Besides this, the kind of *Impact* in the *Impact Model* is also checked by the *BoundaryToDfBoundary* merge translation agent that traverse the *Impact Model XML* tree handling each kind of *Impact* separately. Each *Impact* is mapped directly to an equivalent *Flow* in the *DataFlow Model*. The translation is perform by either *ParticipationToDfInputFlow* and *ParticipationToDfOutputFlow* component. From that, four basic rules can be extracted:

1- *For each create impact in the Impact Model, there exist an create flow in the DataFlow Model.*

$$
\begin{aligned}
&(DfCreateFlow \;<:\; Arc)\\
&tr : CreateFlow \;\rightarrow\; DfCreateFlow\\
&\forall\, cf1 : CreateFlow, (r1, r2) : ImpRole,\\
&m1 : ImpactModel \bullet (cf1, r1, r2 \;\in_m\; m1)\\
&\longrightarrow\\
&\exists!\; cf2 : DfCreateFlow, (r3, r4) : DfRole,\\
&m2 : DataFlowModel \;\bullet\; ((r3 \;=\; tr(r1))\\
&\wedge\; (r4 \;=\; tr(r2)) \;\wedge\; (cf2, r3, r4 \;\in_m\; m2)
\end{aligned}
$$

2- *For each read impact in the Impact Model, there exist an read flow in the DataFlow Model.*

$$
\begin{aligned}
&(DfReadFlow \;<:\; Arc)\\
&tr : ReadFlow \;\rightarrow\; DfReadFlow\\
&\forall\, rf1 : ReadFlow, (r1, r2) : ImpRole,\\
&m1 : ImpactModel \;\bullet\; (rf1, r1, r2 \;\in_m\; m1)\\
&\longrightarrow\\
&\exists!\; rf2 : DfReadFlow, (r3, r4) : DfRole,\\
&m2 : DataFlowModel \;\bullet\; ((r3 \;=\; tr(r1))\\
&\wedge\; (r4 \;=\; tr(r2)) \;\wedge\; (rf2, r3, r4 \;\in_m\; m2)
\end{aligned}
$$

3- *For each update impact in the Impact Model, there exist an update flow in the DataFlow Model.*

$$
\begin{aligned}
&(DfUpdateFlow \;<:\; Arc)\\
&tr : UpdateFlow \;\rightarrow\; DfUpdateFlow\\
&\forall\, uf1 : UpdateFlow, (r1, r2) : ImpRole,\\
&m1 : ImpactModel \;\bullet\; (uf1, r1, r2 \;\in_m\; m1)\\
&\longrightarrow\\
&\exists!\; uf2 : DfUpdateFlow, (r3, r4) : DfRole,\\
&m2 : DataFlowModel \;\bullet\; ((r3 \;=\; tr(r1))\\
&\wedge\; (r4 \;=\; tr(r2)) \;\wedge\; (uf2, r3, r4 \;\in_m\; m2)
\end{aligned}
$$

4- *For each delete impact in the Impact Model, there exist an delete flow in the DataFlow Model.*

$$
\begin{aligned}
&(DfDeleteFlow \;<:\; Arc)\\
&tr : DeleteFlow \;\rightarrow\; DfDeleteFlow\\
&\forall\, df1 : DeleteFlow, (r1, r2) : ImpRole,\\
&m1 : ImpactModel \;\bullet\; (df1, r1, r2 \;\in_m\; m1)\\
&\longrightarrow\\
&\exists!\; df2 : DfDeleteFlow, (r3, r4) : DfRole,\\
&m2 : DataFlowModel \;\bullet\; ((r3 \;=\; tr(r1))\\
&\wedge\; (r4 \;=\; tr(r2)) \;\wedge\; (df2, r3, r4 \;\in_m\; m2)
\end{aligned}
$$

It is important to highlight that these agents use the *ImpRoleToDfRole* and *RoleToDfRole* agents to translate the end-roles for each flow and participation, respectively, into correspondence ones in the *DataFlow Model*.

$$tr : Role \rightarrow DfRole$$

$$\begin{aligned} &\forall\ r1 : Role, m1 : ImpactModel \bullet (r1 \in_m m1) \\ &\longrightarrow \\ &\exists!\ r2 : DfRole, m2 : DataFlowModel \\ &\bullet ((r2 \in_m m2) \land NameOf(r1, r2)) \end{aligned}$$

## 8.2.2 Translating Information Model into Data (Dependency) Model

### 8.2.2.1 Data Diagram (Model)

A direct *one-to-one* mapping exists between a given *Information Model* and the generated *Data Model*, in which the *Information Diagram* element in the *Information Model* is translated directly into a *Data Dependency Diagram* node. This is expressed in the following rule:

$$tr : InfDiagram \rightarrow DDiagram$$

$$\begin{aligned} &\forall\ d1 : InfDiagram,\ \forall\ e1 : InfEntity, \\ &\forall\ a : Relationship,\ \forall\ (r1, r2) : InfRole, \\ &m1 : InformationModel \bullet ((e1, a, r1, r2 \in_m d1) \\ &\land (d1 \in_m m1)) \\ &\longrightarrow \\ &\exists!\ d2 : DDiagram, (r3, r4) : DRole, d : Dependency, \\ &e2 : DEntity, m2 : DataModel \bullet ((e2 = tr(e1)) \\ &\land (d = tr(a)) \land (r3 = tr(r1)) \land (r4 = tr(r2)) \\ &\land (e2, d, r3, r4 \in_m d2) \land (d2 \in_m m2) \\ &\land NameOf(d1, d2)) \end{aligned}$$

### 8.2.2.2 Data Entity

Deriving *DEntity* is done by mapping an entity in *Information Model* into a corresponding one in the *Data Dependency Model*. This translation is carried out by *InfEntityToDEntity* agent. The rule of this translation is: *For each entity in the Information Model, there exists an entity in the Data Model.* It can be expressed in FOPL as:

$$tr : InfEntity \rightarrow DEntity$$

$$\begin{aligned} &\forall\ ent1 : InfEntity, \forall\ att1 : InfAttribute, \\ &m1 : InformationModel \bullet ((att1 \in_p ent1) \\ &\land (ent1 \in_m m1)) \\ &\longrightarrow \\ &\exists!\ ent2 : DEntity, att2 : DAttribute, m2 : DataModel \\ &\bullet ((att2 \in_p ent2) \land (ent2 \in_m m2) \\ &\land NameOf(ent1, ent2) \land (att2 = tr(att1))) \end{aligned}$$

In addition to this, *DEntity* may also correspond to an *Information Model* many-to-many *Association*. This kind of association is promoted to a full entity in the *Data Model*. The end-roles are converted into named attributes storing the identities of the *InfEntities* of the association as extra *DAttribute* model the old end *Roles*.

This translation is performed by *InfAssocToMergedTable* agent. It searches in the source model tree for many-to-many *Association* nodes, and then generates a new *DEntity* element in the target model. This includes translating the *Identifiers* of each *InfEntity* into unique *DAttributes* within the created entity. These sub-translations are carried out by *IdentifierToUnique* agent. From above, two main roles can be expressed as:

1- *For each many-to-many association in the Information Model, there exists an entity in the Data Model.*

$$
\begin{aligned}
tr : Association &\rightarrow DEntity \\
\forall\ a : Association, (obj1, obj2) &: InfEntity, \\
(id1, id2) : Identifier, m1 &: InformationModel \\
\bullet\ ((a, obj1, obj2\ \in_m\ m1) &\wedge\ ManyToMany(a) \\
\wedge\ Links(a, obj1, obj2)\ &\wedge\ Id(obj1, id1) \\
&\wedge\ Id(obj2, id2)) \\
&\longrightarrow \\
&\quad \exists!\ e : DEntity, (att3, att4) : DAttribute, \\
&\quad m2 : DataModel\ \bullet\ ((e\ \in_m\ m2) \\
&\quad \wedge (att3, att4\ \in_p\ e)\ \wedge\ NameOf(a, ent) \\
&\quad \wedge\ (att3\ =\ tr(id1))\ \wedge\ (att4\ =\ tr(id2))
\end{aligned}
$$

2- *For each identifier of an entity involved in a many-to-many association, in the Information Model, there exists an equivalent unique attribute in the corresponding entity of that association in the Data Model.*

$$
\begin{aligned}
tr : Identifier &\rightarrow DAttribute \\
\forall\ a : Association,\ \forall\ e &: InfEntity, \\
\forall\ id : Identifier, m1 &: InformationModel \\
\bullet\ ((a, e\ \in_m\ m1)\ &\wedge\ ManyToMany(a) \\
\wedge\ Id(e, id)\ \wedge\ (Source(a, e)\ &\oplus\ Target(a, e))) \\
&\longrightarrow \\
&\quad \exists!\ ent : DEntity,\ \exists!\ att : DAttribute, m2 : DataModel \\
&\quad \bullet\ ((ent\ \in_m\ m2)\ \wedge\ (att\ \in_p\ ent) \\
&\quad \wedge\ Unique(att)\ \wedge\ NameOf(a, ent)))
\end{aligned}
$$

It is worth emphasising an interesting case in regard to *disjoint Generalisation*, where it is assumed that *Child* entities consist of the attributes of the *Parent* Entity, added to its original attributes. At the end the *Parent* entity is deleted from the model and a new version of its *Child* entities appear in the *Data Dependency Model*.

From this, a new rule might be stated: *For each disjoint generalisation, which connects child entities to a parent one, in the Information Model, there exists an equivalent entity to the child entities that have the attributes of the parent.* This can be written in logic as:

$$tr : Generalisation \rightarrow DEntity$$

$$\begin{aligned}
&\forall\ g : Generalisation,\ \forall\ e1, e2 : InfEntity,\\
&\forall\ a1, a2 : InfAttribute, m1 : InformationModel\\
&\bullet\ ((g, e1, e2\ \in_m\ m1)\ \wedge\ (a1\ \in_p\ e1)\\
&\wedge\ (a2\ \in_p\ e2)\ \wedge\ Connects(g, e2, e1)\\
&\wedge\ Disjoint(g)\ \wedge\ (e1\ \neq\ e2)\ \wedge\ (a1\ \neq\ a2))\\
&\longrightarrow\\
&\exists!\ ent : DEntity,\ \exists\ att1, att2 : DAttribute, m2 : DataModel\\
&\bullet\ ((ent\ \in_m\ m2)\ \wedge\ (att1, att2\ \in_p\ ent)\\
&\wedge\ NameOf(e2, ent)\\
&\wedge\ (att1\ =\ tr(a1))\ \wedge\ (att2\ =\ tr(a2)))
\end{aligned}$$

#### 8.2.2.3 Data Attribute

A straightforward mapping exists between *Information Model Attribute* to generate equivalent *Data Dependency DAttribute*, in which each *Attribute* element is directly translated into a *DAttribute*. The following logic formula expresses this:

$$tr : InfAttribute \rightarrow DAttribute$$

$$\begin{aligned}
&\forall\ att1 : InfAttribute, m1 : InformationModel\\
&\bullet\ (att1\ \in_m\ m1)\\
&\longrightarrow\\
&\exists!\ att2 : DAttribute, m2 : DataModel\\
&\bullet\ ((att2\ \in_m\ m2)\ \wedge\ NameOf(att1, att2))
\end{aligned}$$

#### 8.2.2.4 Data Dependency

A *Dependency* in the *Data Model* might be derived by translating one of the three *Information Model* concepts: *many-to-one* association, (overlapping) generalisation, and composition. In regards to *many-to-one* association, the *InfAssocToDDependency* rule (agent) is responsible for translating each detected *Association* into a *Dependency* in which the many always depend on the one. Any *many-to-many* association must be promoted into a into a pair of *many-to-one* associations first, then the multiplicity constraints must also be translated (see earlier rule above).

From the above, the first translating rule can be summarised as: *For each many-to-one association between two entities in the Information Model, there exists an equivalent dependency in between the equivalent entities the Data Dependency Model where the entity on the many-side depends on the entity on the one-side.* This can be expressed in FOPL as:

$$tr : Association \rightarrow Dependency$$

$$\begin{aligned}
&\forall\ e1, e2 : InfEntity,\ \forall\ a : Association,\\
&m1 : InformationModel \bullet ((a, e1, e2 \in_m m1)\\
&\wedge\ ManyToOne(a)\ \wedge\ Connect(a, e1, e2)\ \wedge\ (e1 \neq e2))\\
&\longrightarrow\\
&\exists!\ d : Dependency, e3, e4 : DEntity, m2 : DataModel\\
&\bullet\ ((d, e3, e4 \in_m m2)\ \wedge\ Connects(d, e3, e2)\\
&\wedge\ NameOf(e1, e3)\ \wedge\ NameOf(e2, e4)\ \wedge\\
&Depends(e3, e4)\ \wedge\ (e3 = tr(e1))\ \wedge\ (e4 = tr(e2)))
\end{aligned}$$

In addition to this, every (overlapping) *Generalisation* in the *Information Model* is resolved by making the specific entities depend on the general one. This mapping is carried out by the *InfGenToDDependency* agent. The second rule of translation can be defined as: *For each overlapping generalisation between entities in the Information Model, there exists an equivalent dependency relationship between the equivalent entities in the Data Dependency Model.* This can be written in logic as:

$$tr : Generalisation \rightarrow Dependency$$

$$\begin{aligned}
&\forall\ e1, e2 : InfEntity,\ \forall\ g : Generalisation,\\
&m1 : InformationModel \bullet ((g, e1, e2 \in_m m1)\\
&\wedge \neg\ Disjoint(g)\ \wedge\ Connect(g, e1, e2)\ \wedge\ (e1 \neq e2))\\
&\longrightarrow\\
&\exists!\ d : Dependency, e3, e4 : DEntity, m2 : DataModel\\
&\bullet\ ((d, e3, e4 \in_m m2)\ \wedge\ Connects(d, e3, e4)\\
&\wedge\ (NameOf(e1, e3)\ \wedge\ NameOf(e2, e4)\ \wedge\\
&Depends(e3, e4)\ \wedge (e3 = tr(e1))\ \wedge\ (e4 = tr(e2)))
\end{aligned}$$

Furthermore, every (***not total***) *Composition* in the *Information Model* is resolved by making the whole depend on the parts. The *InfComToDDependency* rule is responsible for proceeding with this mapping from *Composition* into *Dependency*. From this, the third rule for deriving *Dependency* can be defined as: *For each composition between entities in the Information Model, there exists an equivalent dependency relationship between the equivalent entities in the Data Dependency Model.* This can be written in logic as:

$$tr : Composition \rightarrow Dependency$$

$$\begin{aligned}
&\forall\ e1, e2 : InfEntity,\ \forall\ c : Composition,\\
&m1 : InformationModel \bullet ((c, e1, e2 \in_m m1)\\
&\wedge \neg\ Total(c)\ \wedge\ Connect(c, e1, e2)\ \wedge\ (e1 \neq e2))\\
&\longrightarrow\\
&\exists!\ d : Dependency, e3, e4 : DEntity, m2 : DataModel\\
&\bullet\ ((d, e3, e4 \in_m m2)\ \wedge\ Connects(d, e3, e4)\\
&\wedge\ NameOf(e1, e3)\ \wedge\ NameOf(e2, e4)\ \wedge\\
&Depends(e3, e4)\ \wedge\ (e3 = tr(e1))\ \wedge\ (e4 = tr(e2)))
\end{aligned}$$

### 8.2.3 Transforming (initial) DataFlow Model into Detailed DataFlow Model

#### 8.2.3.1 Detailed DataFlow Diagram

This one-to-one mapping aims at applying a task decomposition strategy to *DataFlow* tasks that are connceted to more than one flow. From that, *For each DataFlow Model, there exists a detailed DataFlow one generated from it.* This can be formalised as follows:

$$tr : DfDiagram \rightarrow DfDiagram$$

$$\begin{aligned}
&\forall\ d1 : DfDiagram, \forall\ b1 : Boundary,\\
&m1 : DataFlowModel \bullet ((b1 \in_m d1)\\
&\wedge (d1 \in_m m1)\\
&\longrightarrow\\
&\exists!\ d2 : DfDiagram, b2 : Boundary,\\
&m2 : DataFlowModel \bullet ((b2 \in_m d2)\\
&\wedge (d2 \in_m m2) \wedge (b2 = tr(b1)))
\end{aligned}$$

#### 8.2.3.2 Detailed DataFlow Boundary

Each *DfTask* that is connected to more than one *Flow* is translated into a *DfBoundary* in the detailed *DataFlow Model*. This can be defined as: *for each dataflow task that is connected to more than one flow, there exists one equivalent boundary in the detailed dataflow model.* The following FOPL expresses this formally:

$$tr : DfTask \rightarrow DfBoundary$$

$$\begin{aligned}
&\forall\ t1 : DfTask, f1, f2 : Flow, a1 : DfActor,\\
&m1 : DataFlowModel \bullet ((t1, f1, f2, a1 \in_m m1)\\
&\wedge(f1 \neq f2) \wedge (Source(f1, t1) \oplus\\
&Target(f1, t1)) \wedge (Source(f2, t1)\\
&\oplus Target(f2, t1)))\\
&\longrightarrow\\
&\exists!\ b : DfBoundary, t2 : DfTask, f3, f4 : Flow,\\
&a2 : DfActor, m2 : DataFlowModel \bullet ((b \in_m m2)\\
&\wedge NameOf(t1, b) \wedge (f3 = tr(f1)) \wedge\\
&(f4 = tr(f2)) \wedge (t2 = tr(t1)) \wedge (a2 = tr(a1)))
\end{aligned}$$

#### 8.2.3.3 Detailed DataFlow Task

As a general rule, each *DfTask* in an initial DFD model is translated into an equivalent *DfTask* in the detailed DFD one. This mapping is applicable when the *DfTask* is the source of either *create, delete* or *update*, or when it is target of either *input* or *read.* The following rule expresses the cases when the flow is either a *create, delete* or *update* flow:

$$tr : DfTask \rightarrow DfTask$$

$$\begin{aligned}
&\forall\ f1 : Flow, t1 : DfTask, m1 : DataFlowModel \\
&\bullet ((t1, f1\ \in_m\ m1) \wedge (DfCreate(f1) \\
&\oplus DfDelete(f1) \oplus DfUpdate(f1)) \wedge (Source(f1, t1))) \\
&\longrightarrow \\
&\exists!\ t2 : DfTask, m2 : DataFlowModel \\
&\bullet\ ((t2\ \in_m\ m2) \wedge NameOf(t1, t2))
\end{aligned}$$

In addition to this, the following rule expresses the mapping when the flow is either an *input* and a *read* flow.

$$tr : DfTask \rightarrow DfTask$$

$$\begin{aligned}
&\forall\ f1 : Flow, t1 : DfTask, m1 : DataFlowModel \\
&\bullet ((t1, f1\ \in_m\ m1) \wedge (DfRead(f1) \\
&\oplus DfInput(f1)) \wedge (Target(f1, t1))) \\
&\longrightarrow \\
&\exists!\ t2 : DfTask, m2 : DataFlowModel \\
&\bullet\ ((t2\ \in_m\ m2) \wedge NameOf(t1, t2))
\end{aligned}$$

Any *DfTask* that is connected to more than one *Flow* will be decomposed into a number of atomic tasks based on the number of flows to which it is connected. The following rule expresses this mapping when the task is a source in more than one flow:

$$tr : DfTask \rightarrow DfTask$$

$$\begin{aligned}
&\forall\ f1, f2 : Flow, t1 : DfTask, m1 : DataFlowModel \\
&\bullet ((t1, f1, f2\ \in_m\ m1) \wedge (f1\ \neq\ f2) \\
&\wedge (Source(f1, t1) \oplus Source(f2, t1))) \\
&\longrightarrow \\
&\exists\ t1a, t1b : DfTask, m2 : DataFlowModel \\
&\bullet\ ((t1a, t1b\ \in_m\ m2) \wedge AtomicTaskOf(ta, t1a) \\
&\wedge\ AtomicTaskOf(t1, t1b) \\
&\wedge\ (Source(f1, t1a) \oplus Source(f2, t1b)))
\end{aligned}$$

It is worth saying that the rule expressed above might be rewritten by considering the task to be target in more than a flow, or when it is source in one flow and target in another one. The following rule expresses the second case:

$$tr : DfTask \rightarrow DfTask$$

$$\begin{aligned}
&\forall\ f1, f2 : Flow, t1 : DfTask, m1 : DataFlowModel \\
&\bullet ((t1, f1, f2\ \in_m\ m1) \wedge (f1\ \neq\ f2) \\
&\wedge (Target(f1, t1) \oplus Target(f2, t1))) \\
&\longrightarrow \\
&\exists\ t1a, t1b : DfTask, m2 : DataFlowModel \\
&\bullet\ ((t1a, t1b\ \in_m\ m2) \wedge AtomicTaskOf(t1, t1a) \\
&\wedge\ AtomicTaskOf(t1, t1b) \\
&\wedge\ (Target(f1, t1a) \oplus Target(f2, t1b)))
\end{aligned}$$

A priority scoring mechanism is introduced for prioritising each atomic task in the detailed DFD. This technique is applied during decomposing initial DFD tasks into their atomic ones.

#### 8.2.3.4 Detailed DataFlow Object

A *DfObject* is passed directly to the detailed *DataFlow Model* during the transformation step. This mapping is carried out by the *ArcToDfObject* agent which maps the *DfObject* at both ends both ends of each *Flow*. The following rule demonstrates the transformation in the case of *create, delete* and *update* flows only.

$$tr : DfObject \rightarrow DfObject$$

$$\begin{aligned}
&\forall f1 : Flow, obj1 : DfObject, m1 : DataFlowModel \\
&\bullet((obj1, f1 \in_m m1) \wedge (DfCreate(f1) \oplus DfDelete(f1) \\
&\oplus DfUpdate(f1)) \wedge (Target(f1, obj1)) \\
&\longrightarrow \\
&\exists!\ obj2 : DfObject, m2 : DataFlowModel \\
&\bullet\ ((obj2 \in_m m2) \wedge NameOf(obj1, obj2))
\end{aligned}$$

By considering *DfReadFlow*:

$$tr : DfObject \rightarrow DfObject$$

$$\begin{aligned}
&\forall\ f1 : Flow, obj1 : DfObject, m1 : DataFlowModel \\
&\bullet\ ((obj1, f1 \in_m m1) \wedge (DfRead(f1) \wedge (Source(f1, obj1)) \\
&\longrightarrow \\
&\exists!\ obj2 : DfObject, m2 : DataFlowModel \\
&\bullet\ ((obj2 \in_m m2) \wedge NameOf(obj1, obj2))
\end{aligned}$$

#### 8.2.3.5 Detailed DataFlow Actor

A *DfActor* is passed directly to the detailed *DataFlow Model* during the transformation step. This mapping is carried out by the *ArcToDfActor* agent by detecting *DfActor* at the both ends of each *Flow*. The following formula illustrates the rule in the case of *DfInputFlow*:

$$tr : DfActor \rightarrow DfActor$$

$$\begin{aligned}
&\forall\ f1 : DfInputFlow, a1 : DfActor, m1 : DataFlowModel \\
&\bullet\ ((a1, f1 \in_m m1) \wedge (Source(f1, a1)) \\
&\longrightarrow \\
&\exists!\ a2 : DfActor, m2 : DataFlowModel \\
&\bullet\ ((a2 \in_m m2) \wedge (NameOf(a1, a2)))
\end{aligned}$$

by considering the *DfOutputFlow*:

$$tr : DfActor \rightarrow DfActor$$

$$\begin{aligned}
&\forall\ f1 : DfOutputFlow, a1 : DfActor, m1 : DataFlowModel \\
&\quad \bullet\ ((a1, f1\ \in_m\ m1)\ \wedge\ (Target(f1, a1)) \\
&\quad \longrightarrow \\
&\quad\quad \exists!\ a2 : DfActor, m2 : DataFlowModel \\
&\quad\quad \bullet\ (a2\ \in_m\ m2)
\end{aligned}$$

#### 8.2.3.6 Detailed DataFlow Flows

All *Flow* types are passed directly to the detailed *DataFlow Model* the transformation step except *DfUpdateFlow*. This can be expressed in the following general FOPL law:

$$tr : Flow \rightarrow Flow$$

$$\begin{aligned}
&\forall\ f1 : Flow, m1 : DataFlowModel\ \bullet\ ((f1\ \in_m\ m1) \\
&\quad \wedge\ (DfCreateFlow(f1)\ \oplus\ DfReadFlow(f1) \\
&\quad \oplus\ DfDeleteFlow(f1)\ \oplus\ DfInputFlow(f1) \\
&\quad \oplus\ DfOutputFlow(f1)) \\
&\quad \longrightarrow \\
&\quad\quad \exists!\ f2 : Flow, m2 : DataFlowModel \\
&\quad\quad \bullet\ ((f2\ \in_m\ m2) \wedge \\
&\quad\quad FlowTypeOf(f1, f2))
\end{aligned}$$

In the case of *DFUpdateFlow*, it is decomposed into *DfReadFlow* and *DfWriteFlow* and attached to the decomposed *DfTask* connected to that *DfUpdateFlow*. This can be expressed as:

$$tr : UpdateFlow \rightarrow DfReadFlow \times DfWriteFlow$$

$$\begin{aligned}
&\forall\ f1 : Flow, m1 : DataFlowModel \\
&\quad \bullet\ ((f1\ \in_m\ m1)\ \wedge\ (UpdateFlow(f1)) \\
&\quad \longrightarrow \\
&\quad\quad \exists!\ f2 : DfReadFlow, f3 : DfWriteFlow, \\
&\quad\quad m2 : DataFlowModel\ \bullet\ (f2, f3\ \in_m\ m2)
\end{aligned}$$

#### Priority Scoring Algorithm of Tasks

Two prioritising steps are used to extract (predict) the internal behaviour of each *DfBoundary* in the detailed DFD: *analysing types of tasks* and *analysing data on flows* . In the first step, *Partial Dependency* from the task execution perspective is considered a strategy for prioritising all atomic tasks at the detailed DFD model. For each DFD boundary, tasks are classified into four levels of priority based on their types. The task that has a high priority score is executed before the lower ones. It is assumed that the *input* tasks have the highest priority in a boundary with a score of 4. The *read* tasks come next with a score of 3. Whereas, *create, delete* and *write* come after with a score of 2. The *output* task, at the end, has a score of 1 (see Table 8.1).

| **Score** | **Type(s) of Tasks** |
|---|---|
| **4** | Input |
| **3** | Read |
| **2** | Create, Delete, and Write |
| **1** | Output |

Table 8.1: Priority ranking of different types of task

In the second step, data on flows is checked in order to be sure in which order tasks might be executed. For example, suppose we have a piece of data ($x$) on an *input* flow and the same ($x$) on a *create* flow, it can be predicted (extracted) that *$x$ has to be input before it can be used in a create task*. As a result of these two steps, additional *DfReadFlows* are generated to connect tasks that have higher priority scores to the lower ones, for each detailed DFD boundary.

### 8.2.4 Translating DataFlow Model into (Screen) State Model

#### 8.2.4.1 State Diagram (model)

A *StDiagram* element in translated from an existing *DfDiagram* node in the detailed *DataFlow Model*, in which *For each detailed DFD, there exists a diagram in the State Model.* This can be expressed as:

$$
\begin{aligned}
tr : DfDiagram \;\rightarrow\; StDiagram \\
\forall\ d1 : DfDiagram, b1 : DfBoundary, \\
m1 : DataFlowModel \;\bullet\; ((b1 \;\in_m\; d1) \\
\wedge\ (d1 \;\in_m\; m1)) \\
\longrightarrow \\
& \exists!\ d2 : StDiagram, b2 : StBoundary, m2 : StateModel \\
& \bullet\ ((b2 \;\in_m\; d2) \;\wedge\; (d2 \;\in_m\; m2) \\
& \wedge\ NameOf(d1, d2) \;\wedge\; (b2 \;=\; tr(b1)))
\end{aligned}
$$

#### 8.2.4.2 Boundary

Each *StBoundary* corresponds to a *DfBoundary* in the *DataFlow Model.* This mapping is acheived using the *DfdBoundaryToStBoundary* agent that provides a one-to-one translation step. The main rule of this mapping is: *For each boundary in the DataFlow Model, there exists a boundary in the State Model.* The logic expression of this rule is:

$$tr : DfBoundary \rightarrow StBoundary$$

$$\begin{aligned}
&\forall\ m1 : DataFlowModel, t : DfTask, f : Flow, \\
&b1 : DfBoundary \bullet ((t, f \in_m b1) \wedge (b1 \in_m m1)) \\
&\longrightarrow \\
&\exists!\ m2 : StateModel, b2 : StBoundary, s : State, \\
&ts : Trainsition \bullet ((s, ts \in_m b2) \wedge \\
&(b2 \in_m m2) \wedge NameOf(b1, b2) \wedge \\
&(s = tr(f)) \wedge (ts = tr(t)))
\end{aligned}$$

Is it worth saying that each generated boundary has a number of states that describe the successful case of the business scenario. The step of adding *Error States* for handling possible failure scenarios is a separate *one-to-one* translation step. The *StBoundaryToStBoundary* is responsible for this by traversing each *State* in the *StBoundary* to produce and attach a related error reporting state to the boundary.

#### 8.2.4.3 State

*Waiting State* in the *State Model* corresponds to an *DfInputFlow* element in the *DataFlow Model*, in which each input flow is translated into a *State* that is waiting for receiving some inputs. These inputs, which are equivalent to the data on *DfInputFlow*, are translated into internal variables (*Variable* ) for that state. This mapping is achieved by the *ArcToWaitingState* translator for each accepted source element in the *DataFlow Model*.

One the other hand, *Ready State* is derived by translating the other kinds of flows, namely: *DfCreateFlow, DfReadFlow, DfUpdate, DfOutputFlow* and *DfWriteFlow*. This translation is done by the *ArcToReadyState* agent for each accepted *Flow* in the *DataFlow Model*. From the above, a number of basic transformation rules are formed:

1- *For each Input flow in the DataFlow Model, there exists a State in the State Model that is waiting for receiving some external inputs.*

$$tr : DfInputFlow \rightarrow State$$

$$\begin{aligned}
&\forall\ m1 : DataFlowModel, f : DfInputFlow \bullet (f \in_m m1) \\
&\longrightarrow \\
&\exists\ s : State, m2 : StateModel \\
&\bullet ((s \in_m m2) \wedge Waiting(s))
\end{aligned}$$

2- *For each flow that is either Create, Read, Delete, Write or Output flow in the DataFlow Model, there exists a State in the State Model that is ready to fire an internal system operation or funtion.*

$$tr : Flow \rightarrow State$$

$$\begin{aligned}
\forall\, m1 : DataFlowModel, f : Flow \bullet ((f \in_m m1) \\
\wedge\, (DfCreateFlow(f) \oplus DfReadFlow(f) \oplus \\
DfDeleteFlow(f) \oplus DfWriteFlow(f) \oplus DfOutputFlow(f))) \\
\longrightarrow \\
\exists\, s : State, m2 : StateModel \\
\bullet ((s \in_m m2) \wedge Ready(s))
\end{aligned}$$

#### 8.2.4.4 Transition

In order to obtain *Transition* elements, a complex merge translation step is required. Thus, a *DfTaskToTransition* agent is designed to fit this need. This translator must be fired after deriving all *States* in the *State Model*. *DfTaskToTransition* takes each flow with its attached end *Roles* to create a new *Transition*.

Therefore, the mapping rule of this agent is: *For each source in a flow in the DataFlow Model and its translated state in the State Model, there exists a transition in the State Model, where the source of the transition is the state that is pased as an argument of this agent.* This can be expressed in FOPL as:

$$tr : DfTask \rightarrow Transition$$

$$\begin{aligned}
\forall\, m1 : DataFlowModel, t : DfTask, \\
m2 : StateModel, f : Flow, s : State \bullet ((f \in_m m1) \\
\wedge\, (s \in_m m2) \wedge Source(t, f) \wedge (s = tr(f)) \\
\longrightarrow \\
\exists!\, t : Transition \bullet ((t \in_m m2) \wedge Source(t, s))
\end{aligned}$$

#### 8.2.4.5 Action

Each transition has an *action* attribute refering to its type. For example, if the *Flow* was a *DfCreateFlow*. then the action type will be *create*. This *Action* takes some arguments that are eqivalent to the internal *Variable* of the *State* passed to the *DfTaskToTransition* agent. The body of the action might have *assignment* statements if the *Flow* is either *DfCreateFlow* or *DfWriteFlow*, whereas it may have *filter* statements if the *Flow* is *DfReadFlow*.

## 8.3 Transformation Rules at the Analysis to Design Phase

### 8.3.1 Translating State into GUI Model

#### 8.3.1.1 GUI Diagram

A *GuiDiagram* element in translated from an existing *StDiagram* node in the *State Model*, in which *for each State diagram, there exists a diagram in the GUI Model.* This can be expressed as:

$$tr : StDiagram \rightarrow GuiDiagram$$

$$\begin{aligned}\forall\ d1 : StDiagram, b1 : StBoundary,\\ m1 : StateModel \bullet ((b1 \in_m d1)\\ \wedge (d1 \in_m m1))\\ \longrightarrow\end{aligned}$$
$$\begin{aligned}&\exists!\ d2 : GuiDiagram, b2 : GuiBoundary, m2 : GUIModel\\ &\bullet\ ((b2 \in_m d2) \wedge (d2 \in_m m2)\\ &\wedge\ NameOf(d1, d2)) \wedge (b2 = tr(b1))\end{aligned}$$

#### 8.3.1.2 GUI Window

Each *Window* corresponds directly to a *State* in the *State Model*. This straightforward mapping is implemented in *StateToWindow* agent. This can be expressed in logic as:

$$tr : State \rightarrow Window$$

$$\begin{aligned}\forall\ s : State,\ \forall\ v : Variable, m1 : StateModel\\ \bullet ((s \in_m m1) \wedge (c \in_p s))\\ \longrightarrow\end{aligned}$$
$$\begin{aligned}&\exists!\ w : Window, c : Widget, m2 : GUIModel\\ &\bullet\ ((d2 \in_m m2) \wedge (c \in_c w)\\ &\wedge\ NameOf(s, w) \wedge (c = tr(v)))\end{aligned}$$

*Variables* in the *State* is translated into *Widget* by an agent related to the type of the state. If the state is *waiting*, then the variables will be handled by *svariableToTextfield* that translates these variables into entry text fields. On the other hand, If the state is *ready*, then the variables will be handled by *svariableToLabel* that translates these variables into displaying labels.

#### 8.3.1.3 GUI Button Widget

*Button* control appears in all types of screens. The basic mapping rule for obtaining *Button* can be defined: *Each generated window has a single button.* Unlike other widgets, deriving the *Button* control requires such a complex translation step than other widgets, because it has an event that triggers (invokes) a particular stored procedure in the database. The following FOPL statement express the main rule for deriving *Button*:

$$tr : State \rightarrow Window$$

$$\begin{aligned}\forall\ s : State, m1 : StateModel \bullet ((s \in_m m1)\\ \longrightarrow\end{aligned}$$
$$\begin{aligned}&\exists!\ w : Window, b : Button, m2 : GUIModel\\ &\bullet\ ((w \in_m m2) \wedge (b \in_c w) \wedge NameOf(s, w))\end{aligned}$$

#### 8.3.1.4 GUI Label Widget

*Label* widget appears in all type of windows, displaying the title, error message, names of textfields and more. The basic mapping rule for obtaining *Label* can be defined: *Each generated window has some label.* This can be expressed as:

$$
\begin{aligned}
&(Label \ <: \ Widget) \\
&tr : Variable \ \rightarrow \ Label \\
&\forall \ s : State, \ \forall \ v : Variable, m1 : StateModel \\
&\quad \bullet ((s \ \in_m \ m1) \ \wedge \ (v \ \in_p \ s)) \\
&\quad \longrightarrow \\
&\quad\quad \exists! \ w : Window, l : Label, m2 : GUIModel \\
&\quad\quad \bullet \ ((w \ \in_m \ m2) \ \wedge \ (l \ \in_c \ w) \\
&\quad\quad \wedge \ NameOf(s, w))
\end{aligned}
$$

#### 8.3.1.5 GUI Textfield Widget

*Textfield* widget appears in *waiting* windows only for receiving external inputs. The transformation rule for deriving *Textfield* can be expressed in english as: *Each generated wating window has some text field.* The associated FOPL formula can be expressed as:

$$
\begin{aligned}
&(Textfield \ <: \ Widget) \\
&tr : Variable \ \rightarrow \ Textfield \\
&\forall \ s : State, \ \forall \ v : Variable, m1 : StateModel \\
&\quad \bullet ((s \ \in_m \ m1) \ \wedge (v \ \in_p \ s)) \\
&\quad \longrightarrow \\
&\quad\quad \exists! \ w : Window, t : Textfield, m2 : GUIModel \\
&\quad\quad \bullet \ ((w \ \in_m \ m2) \ \wedge \ (t \ \in_c \ w) \\
&\quad\quad \wedge \ NameOf(s, w))
\end{aligned}
$$

### 8.3.2 Translating DataFlow and Data Model into the DBQ Model

#### 8.3.2.1 Table

Deriving *Table* is achieve directly by translating an entity, in an *Data Dependency Model*, into a corresponding logical tabel, in the *Database and Query Model.* This translation is carried out by *DEntityToTable* agent. The rule of this translation is: *For each entity in the Data Model, there exists a table in the Database Model.* It can be expressed in FOPL as:

$$
\begin{aligned}
&tr : DEntity \ \rightarrow \ Table \\
&\forall \ e : DEntity, \ \forall \ a : DAttribute, m1 : DataModel \\
&\quad \bullet ((e \ \in_m \ m1) \ \wedge \ (a \ \in_p \ e)) \\
&\quad \longrightarrow \\
&\quad\quad \exists! \ t : Table, c : Column, m2 : DatabaseModel \\
&\quad\quad \bullet \ ((t \ \in_m \ m2) \ \wedge \ (c \in_p t) \ \wedge \ NameOf(e, t) \\
&\quad\quad \wedge \ (c \ = \ tr(a)))
\end{aligned}
$$

#### 8.3.2.2 Column

A *Column* corresponds to a *DAttribute* of a *Data Model DEntity*. This mapping is performed by the *DAttributeToColumn* agent that translate every attribute into a column of a relational table. The main rule of this translation can be stated as: *For each attribute in such an entity in the Data Dependency Model, there exists a corresponding column in the equivalent table to that entity.* this can be written in FOPL as:

$$
\begin{aligned}
&tr : DAttribute \ \rightarrow \ Column \\
&\forall\ e : DEntity,\ \forall\ a : DAttribute, m1 : DataModel \\
&\forall\ t : Table, m2 : DatabaseModel \\
&\bullet\ ((e\ \in_m\ m1)\ \wedge\ (a\ \in_p\ e)\ \wedge\ (t\ =\ tr(e))) \\
&\longrightarrow \\
&\quad \exists\ c : Column \\
&\quad \bullet\ \ ((c \in_p t)\ \wedge\ NameOf(t, c))
\end{aligned}
$$

It is significant to mention that some column might be ranged with upper and lower values. This is translated into *SQL Triggers* at the code generation step. There is no element in the DBQ model that refers directly to *triggers*, this because the variation of handling *range constraint* in different database vendors.

#### 8.3.2.3 Foreign Key

In order to construct a *Table* with a foreign key to represent its dependency on another table, the agent *DependToReferTable* is used to do this task. It invokes the *DEntityToTable* agent, described above to translate the entity at the *many* side end-role of the dependency into a table. Additionally it invokes *DRoleToForeignKey* aget to translate the PK of the entity at the *one* side end-role of the dependency into a foreign key attached to the table.

#### 8.3.2.4 Stored Procedure

A *Procedure* is derived as a result of merging concepts from *DataFlow* and *Data Model*. As the DBQ procedure is designed to bind a *SQL* statement (*SELECT, UPDATE, DELETE* and *INSERT*), there are four kinds of agents each one is responsible for translating a particular type of *Flow* in a DataFlow Model into a *Procedure*. These translators are: *CreateFlowToStoredProcedure, DeleteFlowToStoredProcedure, WriteFlowToStoredProcedure, ReadFlowToStoredProcedure.*

The *CreateFlowToStoredProcedure* agent takes a *DfCreateFlow* along with a *DataModel* as arguments to produce a *Procedure* that consists of a *Create* element, and possibly a *Query* one. The Data Model is used to access all *Columns* of the *DEntity* that matches the one appears at the target end of that *DfCreateFlow*. Then, each piece of data on the *DfCreateFlow* with its related *Columns* in the *DEntity* forms a single *assignment statement* using the element *Operator*.

In the same context, the *ReadFlowToStoredProcedure* agent takes a *DfReadFlow* and a *DataModel* as parameters of this translator. It then produces a *Procedure* that contains a *Query* and *Project* node. The Data Model is used to access all *Columns* of the *DEntity* that matches the one appears at the source end of that *DfReadFlow*. Then, each data item on the *DfReadFlow* is matched to its associated *Columns* in the *DEntity* to be considered a *child* of the *Project* element. (filter)

Regarding the *DfDeleteFlow*, the *DeleteFlowToStoredProcedure* agent, which is responsible for this translation, takes a *DfDeleteFlow* with a *DataModel* as arguments in order to produce a *Procedure* that has a *Delete* element, and possibly a *Query* one. The Data Model is used to access all *Columns* of the *DEntity* that matches the one appears at the target end of that *DfDeleteFlow*.

Similarly, the *WriteFlowToStoredProcedure* agent is responsible for translating *DfWriteFlow* into the related DBQ *Procedure*. It takes a *DfWriteFlow* and a *Data Model* to construct a stored procedure *Procedure* with an *Update* and *Query* element. Data on the *DfWriteFlow*, which is interpreted as data to be stored, is converted into an assignment statement.

## 8.4 Alternative Translation Step For Generating the Information Model

Apart from the chain of model transformations that are designed to produce basic information systems, there is a separate translation step that might be applied independently for constructing intermediate artefacts. The following section presents the rules for translating a pre-defined *Impact Model* into a partial *Information Model*.

### 8.4.1 Translating the Impact Model into an (initial) Information Model

#### 8.4.1.1 Information Diagram (Model)

A *Diagram* element is translated from an existing *ImpDiagram* node in the *Impact Model*, in which *for each Impact Diagram, there exists an initial Information Model (diagram)*. This can be expressed as:

$$\begin{aligned} tr : ImpactModel \;\rightarrow\; & InformationModel \\ \forall\, m1 : ImpactModel & \\ \longrightarrow & \\ & \exists!\, m2 : InformationModel \end{aligned}$$

#### 8.4.1.2 Information Entity

*Entities* in the *Information Model* are derived automatically by a direct one-to-one mapping rule. This can be stated as: *For each object in every boundary in the Impact Model, there exists an entity in the Information Model.* This can be expressed in FOPL as:

$$tr : ImpBoundary \rightarrow InfEntity$$

$$\begin{aligned}\forall\ b : ImpBoundary,\ \forall\ obj : ImpObject,\\ m1 : ImpactModel \bullet ((obj \in_m b) \wedge (b \in_m m1))\\ \longrightarrow\\ \exists!\ e : InfEntity, m2 : InformationModel\\ \bullet ((e \in_m m2) \wedge (obj = tr(e)))\end{aligned}$$

#### 8.4.1.3 Information Attribute

*Attributes* in the *Information Model* are not derived automatically from the *Impact Model*.

#### 8.4.1.4 Information Association

*Association* types are derived directly from pre-defined *Flow* types in the *Impact Model*. This can be expressed in the following general FOPL law:

$$\begin{aligned}\forall\ b : ImpBoundary,\ \forall\ f : Flow, m1 : ImpactModel\\ \forall\ (obj1, obj2) : ImpObject \bullet (Connect(f, obj1, obj2)) \wedge\\ ((f, obj1, obj2 \in_m b) \wedge (b \in_m m1) \wedge (obj1 \neq obj2))\\ \longrightarrow\\ \exists\ a : Association,\ \forall\ (e1, e2) : InfEntity,\\ m2 : InformationModel \bullet ((a, e1, e2 \in_m m2)) \wedge\\ (obj1 = tr(e1)) \wedge (obj2 = tr(e2)) \wedge \wedge\\ (a = tr(f))\ Connect(a, e1, e2))\end{aligned}$$

## 8.5 Outlook on the Chapter

The chapter has presented, in-depth, the rules of transformations that are applied to source models in each tranformation step, within the three development phases of BUILD. These rules were formalised and expressed using First-Order Predicate Logic. For each rule, source elements and the generated target ones are highlighted, including constraints that must be satisfied, in the form of unary and n-ary predicates, in order to complete the execution of that rule.

# 9
# Case Studies

"The best time to plan an experiment is after you've done it"

R. A. Fisher

## 9.1 Context

This chapter illustrates the results of the work presented in the previous chapters by introducing a real-world enterprise information system case study (*University Administration System*). Three experiments are designed in which each involves a particular part of the system. The construction of system models, expressed using the $\mu$ML notation, at each development level of BUILD is highlighted. Then, significant translation decisions, which are interesting, are highlighted and discussed for each development phase.

It is also worth mentioning that each $\mu$ML model is represented through BUILD as an *abstract syntax tree (AST)*, written in *Java* node classes, that mimics the structure of that model, allowing the creation of a variety of memory models representing different systems.

In technical detail, the current version of BUILD does not include a graphical editor to enable the automatic shift from the graphical notation of models, sketched by business end-users, into their corresponding underlying XML representations (AST). Therefore, in each experiment, all $\mu$ML requirements models are constructed directly in memory, using the builder-API supplied by the metamodel classes. This is just for the purposes of conducting these experiments; in live usage, the initial models would be supplied as *XML* files, to be unmarshalled by the*JAST* parser.

## 9.2 Overview of the University Administration System

The University Administration System is a common information system that might be developed within academic institutions. From the domain user perspective, the system can be divided into two major parts (sub-systems), namely, *Module Management System* and *Student Enrolment System*. The following subsections introduce each part of the system separately, including its evolution during the BUILD development stages.

## 9.3 Overview of the Module Management Sub-system

The Module Management System enables the users to perform real-world tasks, such as obtaining the full description of a module, updating some details of a module, adding a new module to the system, and removing an existing module. The system has two main types of actor, *Staff* and *Student*, where each actor role has different tasks to perform. Staff are responsible for adding, removing, and modifying modules in the system, one at a time, while *Students* are able to search by a module *code* to retrieve the full description of that module only. Each business task here is performed on a single module at a time; for example, staff can add one module to the system during the execution of the *Add Module* task, and the rest is likewise.

### 9.3.1 Information System Representation at the Requirements Sketching Phase

In this section, the $\mu$ML requirements models that represent the Module Management System are demonstrated graphically. The modelling (construction) activities are then discussed, highlighting critical concepts that are captured in each model. The complete underlying XML representation of all models at this stage is listed in Appendix A.

The development process starts when a business-user expresses the structure of the *Module Management System* using the $\mu$ML *Task Model*. The functions of the system can be categorised into two main jobs which are *managing modules* and *displaying modules description*. Thus, they can be expressed as two main Tasks called *Manage Module* and *See Description* respectively. The *Manage Module* consists of three independent business tasks namely, *Add Module, Delete Module* and *Modify Module*. A white diamond is used to express the *part of* relationship that has independent subtasks.

The human-computer interaction is represented via placing some *Actors* connected to business tasks they are involved in. The *Student* interacts with the *See Description* task by supplying it with a module code to get the full detail of the corresponding module, whereas, *Staff* might interact with all business tasks within the *Manage Module* task. The following *Task Model*, Figure 9.1 represents the structure of the *Module Management System*.

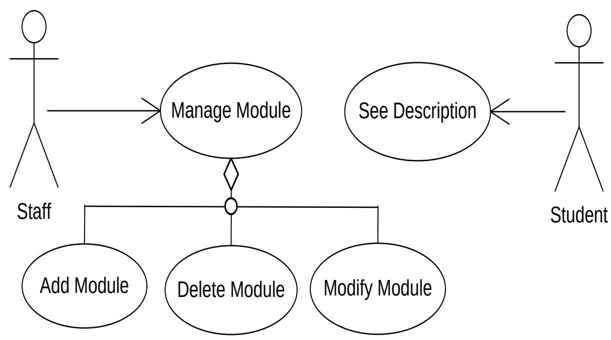



Figure 9.1: Task model. Module Management System

With regard to the behaviour of the system, the $\mu$ML *Impact Model* is used to capture the internal interaction between the business tasks and the system objects. Each task, captured in the *Task Model*, has a different kind of *impact* on the *Module* entity. The *Add Module* task connects to the *Module* by a *create* flow, the *Delete Module* task connects to the object by a *delete* flow, the *Modify Module* task connects to the object by a bi-directional *update* flow, and finally, the *Read Module* task connects to the object by a *read* flow. The following figure, Figure 9.2 is the *Impact Model* of the *Module Management System.*

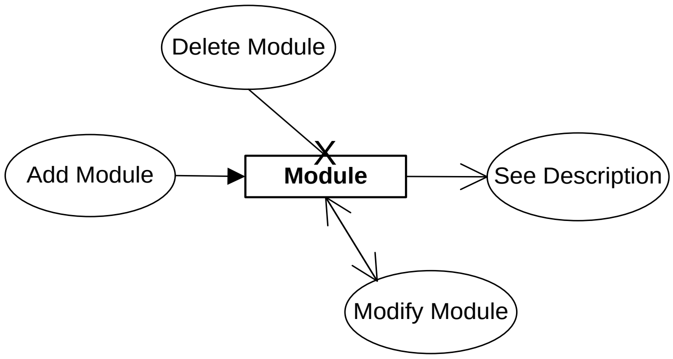



Figure 9.2: Impact model. Module Management System

The third source model at this stage of development is the *Information Model.* The model consists of an entity called *Module* that has four attributes, namely, *code, title, desc* and *credit.* The following figure (Figure 9.3) illustrates this.

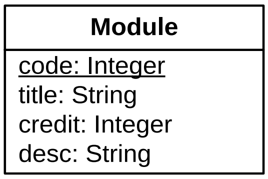



Figure 9.3: Information model. Module Entity.

### 9.3.2 Running the Experiment on the BUILD Framework (1)

**Purpose:** Generate a JDBC Java Swing application with MySQL backend database.
**Critical Feature:** Adopting appropriate decisions for constructing a menu that handles the optional functions of the system.
**Input:** Three $\mu$ML models: Task, Impact and Information Model.
**Output:** Java classes (*.java) and a MySQL script file (*.sql).
**Running Environment:** Eclipse.

#### 9.3.2.1 Construction of Requirements Models

The first requirements model considered by business-users in the development process is the *Task Model.* As there is a custom AST representing *Task Model* in BUILD, the memory model of the system *Task Model* is constructed out of the corresponding *Java* node classes. For example, the *Diagram* node class is used to initialise a new model, and the *Task* node class is used to define new all business tasks of that model. Listing 9.1 below illustrates a snapshot of the manual creation of the *Task Model* of the *Module Management System.* The complete code is presented in Appendix A.

Listing 9.1: Construction of the Task Model

```
Diagram taskModel = new Diagram();
Boundary boundary = new Boundary("Module Management");

Task manage = new Task("Manage Module");

Task add = new Task("Add Module");

Actor actor1 = new Actor("Staff");

Composition comp = new Composition();
comp.addRole(new mde.task.model.Role("manage", manage));

Participation link1 = new Participation();
link1.addRole(new mde.task.model.Role("staff",actor1));
link1.addRole(new mde.task.model.Role("add", add));
```

Similar to the creation of *Task Model*, business users design the *Impact Model* of the system using node classes in the *Impact Model Package*. The *ImpDiagram* class is used to initialise a new *Impact Model* in memory; *ImpTask* and *ImpObject* are used to define all business tasks and actors for their system respectively, and the rest in likewise. Listing 9.2 below demonstrates a snapshot of the construction of *Impact Model*, where the complete code is presented later in Appendix A. It can be seen that names of boundaries and tasks are equivalent to previously designed boundaries and tasks in the *Task Model*, Listing 9.1 above.

Listing 9.2: Construction of the Impact Model

```
ImpDiagram ImpactModel = new ImpDiagram();
ImpBoundary impboundary = new ImpBoundary("Module Management");

ImpTask impManage = new ImpTask("Manage Modules");
ImpTask impAdd = new mde.impact.model.ImpTask("Add Module");

ImpObject impObj1 = new ImpObject("Module");

ImpCreateFlow cf = new ImpCreateFlow();
ImpRole impcf1 = new ImpRole("module", impObj1);
ImpRole impcf2 = new ImpRole("add", impAdd);
cf.addImpRole(impcf2);
cf.addImpRole(impcf1);
```

The final model at this stage, created manually by user, is the *Information Model*. The following listing (Listing. 9.3) illustrates a snapshot of the related part of the *Module Management System*. It is designed using the *Information Model Package* of the BUILD framework. The class *Entity* is used to define the back-end entities of the system.

Listing 9.3: Construction of the Information Model

```
mde.information.model.Diagram informationModel =
    new mde.information.model.Diagram();

Entity moduleEntity = new Entity("Module");
Attribute attr12 = new Attribute("code",
     new Type("Integer")).setIdentifier(true);
Attribute attr13 = new Attribute("title", new Type("String"));
Attribute attr14 = new Attribute("credit", new Type("Integer"));
Attribute attr15 = new Attribute("desc", new Type("String"));
```

Once the initial requirements models have been constructed in memory, the class ASTWriter, the class *ASTWriter* is used for marshalling each abstract syntax tree, built in memory to an XML file. The method *writeDocument* is used to serialise each memory model into its related XML file. The idea behind having all models stored into XML files is to apply a *XML model inspection* as a strategy of evaluating the quality of produced artifacts, as well as verifying and checking the correctness and completeness of translation results. This is discussed later in Chapter 9.

#### 9.3.2.2 Information System Representation at the Analysis Phase

The *Requirement-to-Analysis* model transformation step leads to creating a number of intermediate analysis models of the Module Management System. These models are: *DataFlow, State Model* and *Data Dependency Model.* In this section, the $\mu$ML analysis models that express the *Module Management System* are demonstrated graphically, including some key automatic transformation decisions made by the approach. The complete underlying XML representation of all models at this stage is listed in Appendix A.

While the *Module Management System* deals with a single business entity, there is no significant transformation decision, from *Information* to *Data Dependency Model*, in this case study. The following figure (Figure 9.4) demonstrates the produced *Data Dependency Model* at the *Analysis* Phase of BUILD.

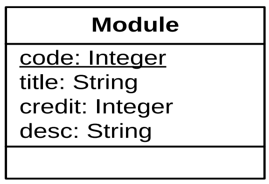



Figure 9.4: Data model. Module Entity

Furthermore, terminal tasks in both *Task* and *Impact Model* appear in the initial *DataFlow Model*, as a result of the translation shift from the requirements phase to the analysis one. These tasks are: *See Descriptions, Add Module, Modify Module* and *Delete Module.* Moreover, business entities and system actors in both *Impact* and *Task Model*, respectively, also appear in the *DataFlow Model*, in which each complete business transaction from an actor to object is expressed in the model (Figure 9.5). These transactions can be listed as:

- The *See Description* task is connected to a *Student* actor by an *input* flow and to a *Module* object by a *read* flow.
- The *Add Module* task is connected to a *Staff* actor by an *input* flow and to a *Module* object by a *create* flow.
- The *Modify Module* task is connected to a *Staff* actor by an *input* flow and to a *Module* object by an *update* flow.
- The *Delete Module* task is connected to a *Staff* actor by an *input* flow and to a *Module* object by a *delete* flow.

From that, each generated task is either connected to an actor or an object, or to both. There are no interconnections between tasks within the boundary and no data on flows. The generated *DataFlow Model* can be visualised using $\mu$ML notation, and interpreted by the user using the clear distinction between its types of flows, as well as the life (cycle) history of an object. One possible example is: *a Staff inputs some data to the Add Module task, then the task creates an object of the type Module and stores it in the Module datastore.* The following figure (Figure 9.5) demonstrates the initial *DataFlow Model* of the *Module Management System.*

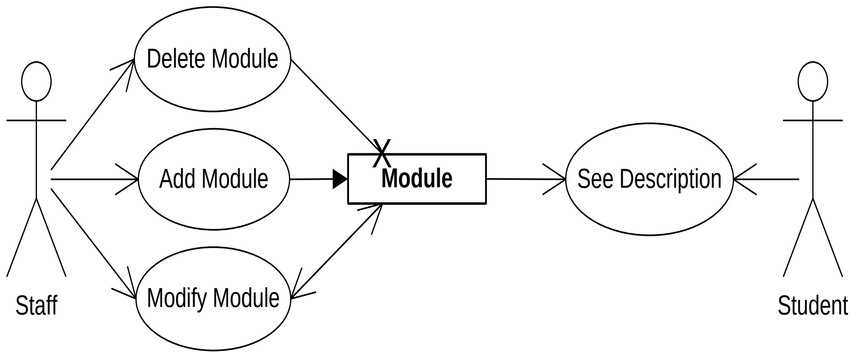



Figure 9.5: (initial) DFD model. Manage Module System

At this stage of the development, a manual engagement, provided by business-users, is required for supplying the created *DataFlow Model* with data on flows. Then, the next translation step is fired by the user in order to complete the development process. The following figure (Figure 9.8) illustrates an initial *DataFlow* with data on flows, supplied.

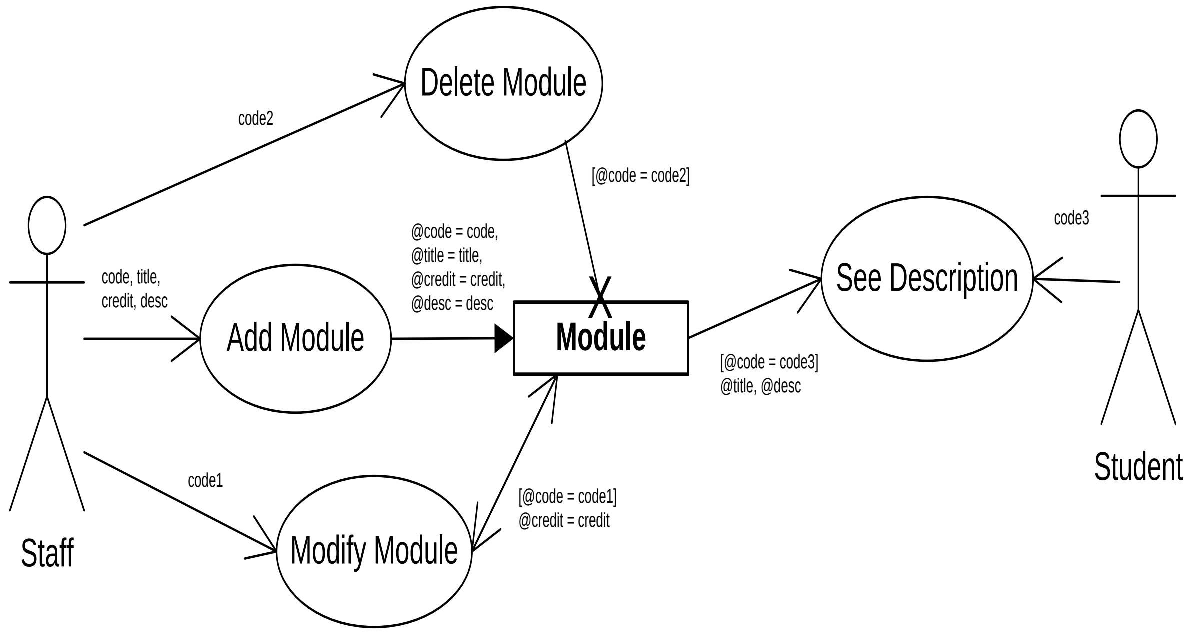



Figure 9.6: DFD model. Manage Module Sys. (with data on flows)

From the generated *DataFlow Model*, it can be seen that each task is connected by two flows, which means each involves two atomic tasks. For instance, *Add Module* task receives the detail of a new module from the user and then (creates) inserts a new module instance into the collection of the modules in the system. Therefore, these tasks must be decomposed into their atomic tasks by translating the initial *DataFlow* into the detailed *DataFlow Model*.

### Deriving Detailed DataFlow Model

A further transformation step is carried out in order to derive a complete *DataFlow Model* that contains atomic tasks. This transformation is known in BUILD as *intitial DataFlow to detailed DataFlow Model*, in which each task in the initial model is broken down into its atomic subtasks that are grouped into a logical boundary. The following figures: Figure 9.7, Figure 9.8, Figure 9.9 and Figure 9.10 demonstrate the generated logical boundaries that form a detailed *DataFlow Model* resulting from this transformation step.

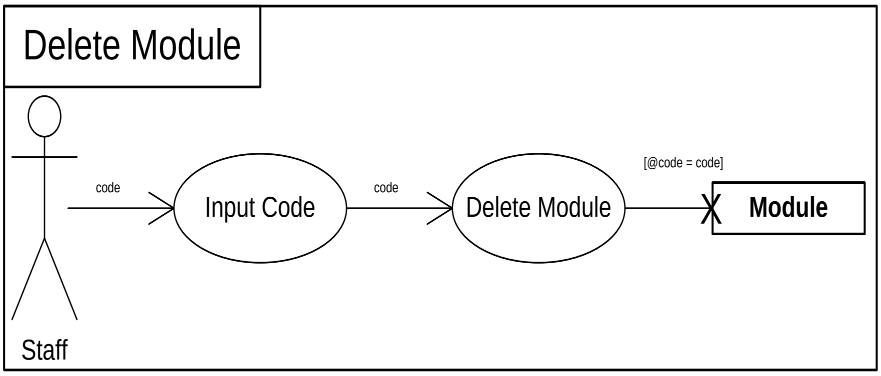



Figure 9.7: DDFD model. Delete Module subtask.

Figure 9.7 above illustrates the internal data flow representation of the *Delete Module* task in the initial DFD model. The task is expressed, in the detailed DFD representation, as a logical boundary that consists of two atomic subtasks. The first one is a subtask to receive data from a user (*Input*), whereas the second task is a task to *delete* the corresponding module from the system. These subtasks are resulting from the decomposition step of the original *Delete Module* task expressed in the source *DataFlow Model*.

An extra *Read* flow between these generated tasks is added after analysing their priority scores. The source of the flow is the subtask that has a higher priority score. In this case, the *Input Code* task that is connected to an input flow is higher priority than the *Delete Module* task.

Figure 9.8 shows a logical detailed DFD *Boundary* that represents the initial DFD *Add Module* task at this level of abstraction. It consists of two decomposed atomic subtasks, a subtask to receive data from a user (*Input*) and another one to *create* a new module (insert it into the system).

Similar to the *Delete Module* task, presented before in Figure 9.7, an additional *read* flow between these subtasks is added to connect the subtask that has a higher priority score to the one that has a lower score. In this case, it is the *Input Code* task that is connected to the *Create Module* one.

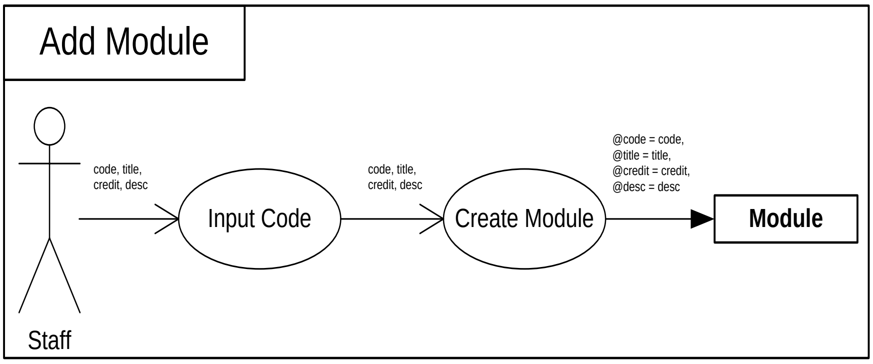



Figure 9.8: DDFD model. Manage Add Module subtask.

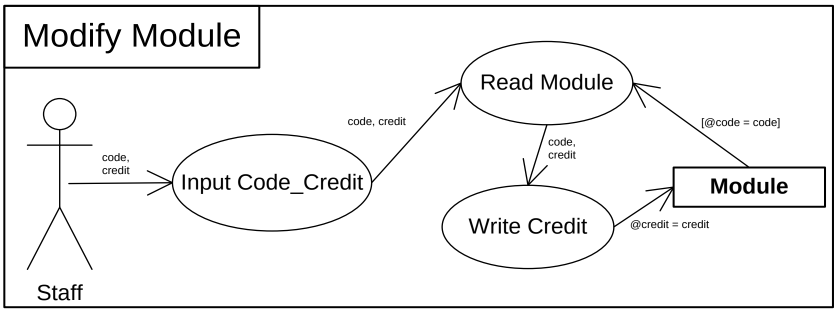



Figure 9.9: DDFD model. Manage Modify Module subtask.

Figure 9.9 illustrates the internal data flow representation of the Modify Module task. The task is expressed, in the detailed DFD model, as a data flow boundary that contains three main subtasks. The first task is introduced for receiving data from a user (*Input*), the second and third task correspond to the broken down *Update* task, and the *read* and *write* subtask. The read task is used to retrieve a module that required to be updated, while the *write* one is used to update particular columns of the retrieved module.

There is a generated *read* flow between the decomposed *Update Module* subtasks (*read* and *write*). It is assumed that this additional flow connects the *Read Module* task to the *Write Module* task. On the other hand, another *read* flow is added that connects the *Input Code* task to the *Read Module* task based on their priority scores.

Figure 9.10) shows the data flow representation within the *See Description* task. A data flow boundary, with an equivalent name to the task name, is utilised at this level of abstraction to present the internal behaviour of each business task, in terms of data flow. It consists of a subtask to receive data from a user (*Input*), a task to *read* a new module and insert it into the system, and a third subtask to display (*output*) the retrieved result to a user.

At the analysis phase, an independent mapping step is carried to derive the *State Model* from the previously generated detailed DFD model. The *Add Module* DFD boundary, for instance, is mapped directly to an *Add Module* boundary in the *State Model*, containing two states (*input* and *create*) that are derived from the *DataFlow Input* and *Create* flow respectively. Moreover, a new transition, with a particular action and/or condition(s), is added for each generated state

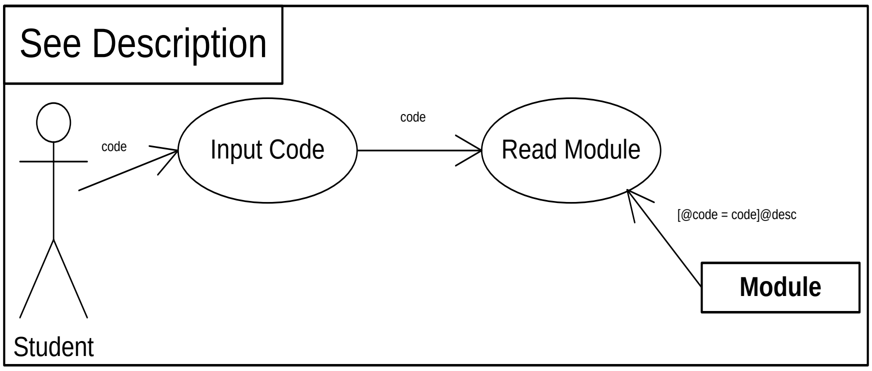



Figure 9.10: DDFD model. See Module Description subtask.

*input_code_title_credit_desc_Waiting* and *Create_Module_Ready* as a result of translating DFD *Tasks*.

In addition to this, a separate in-place modification is applied to the constructed *Add Module* boundary in order to generate an error handling (reporting) state for each *State* in the boundary. In this case, the *input_code_title_credit_desc_Waiting_Error* and *Create_Module_Ready_Error* are added to the *Add Module* boundary. Figure 9.11 below demonstrates the content of the *Add Module* boundary in the *State Model*.

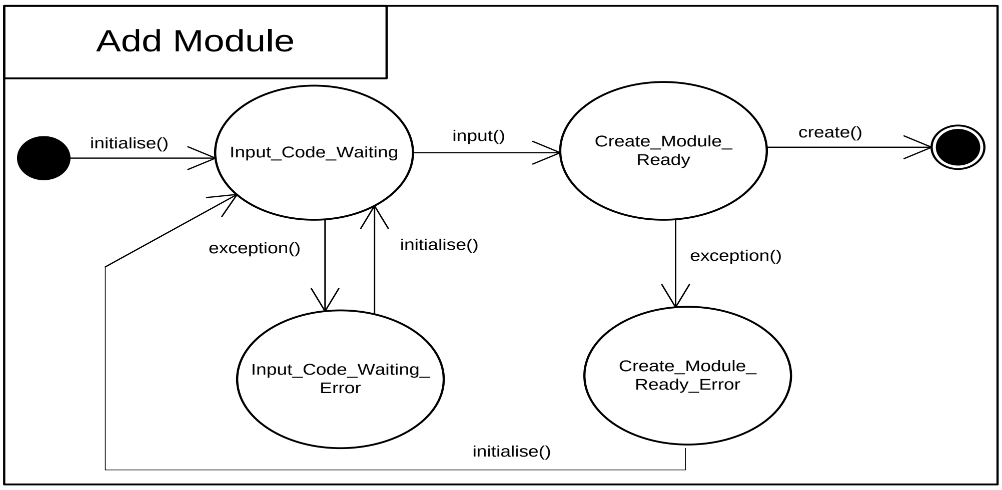



Figure 9.11: State model. Add Module subtask.

Similar to translation steps and decisions mentioned previously for constructing the *Add Module* boundary, the content of the *Delete Module* and *Modify Module* are constructed likewise. The following figures (Figure 9.12 and Figure 9.13) illustrate the content of the *Delete Module* and *Modify Module* boundary respectively.

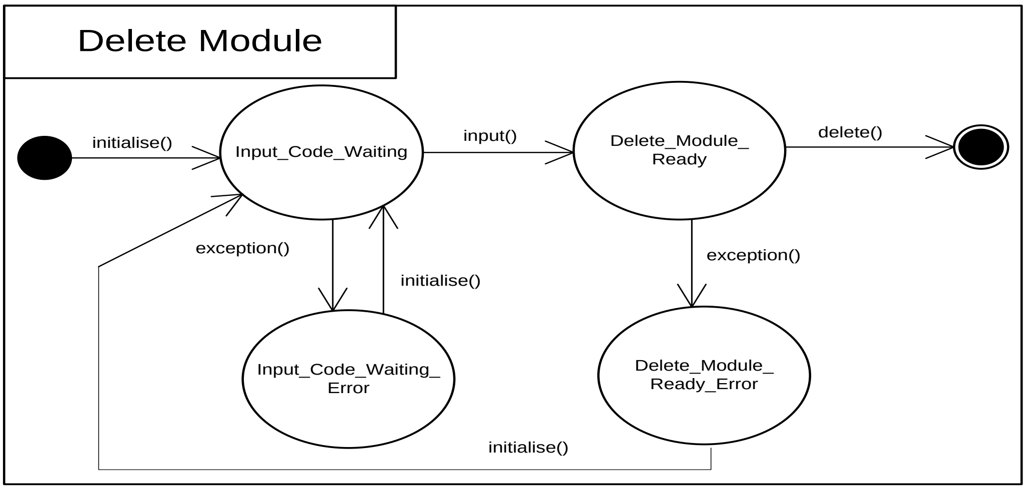



Figure 9.12: State model. Delete Module subtask.

At this stage, the *Data Dependency*, *DataFlow* and *State Model* are completely derived to represent a detailed picture of the system. All business tasks have been broken down into their atomic parts and relate to each other and other tasks in a particular control flow.

#### 9.3.2.3 Information System Representation at the Design Phase

The intermediate artifacts that are generated from the analysis phase, namely, *Data Dependency, DataFlow* and *State Model* are used as source models in this stage of development. The Design phase in BUILD aims at producing two detailed models: *Database and Query Model* and *GUI Model*. The following subsections represent both models at the design level of abstraction.

While the *Module Management System* deals with a single business entity, there is no significant transformation decision, from *Data* to *Database and Query Model*, in this case study. The following figure (Figure 9.14) demonstrates the produced DBQ model at the Analysis Phase of BUILD. The *Module* entity expresses the structure of a relational database table in a platform-independent way rather than specialised for a particular target environment.

In addition, the Graphical User Interface (GUI) Model is the last detailed design model at this stage. Each state from the analysis *State Model* appears in the *GUI Model* as a window. For example, the *Input_regNumber_Waiting* state is translated into a window that has the same name. The generated GUI control elements (widgets) for each *Window* is based on the type of the *State*, either *waiting, ready* or *error*. The number of controls is determined by the number of local variables in each state. The following listing (Listing. 9.4) demonstrates a snapshot of the *GUI Model*, including the specifications of three windows of the models, namely, *Input_regNumber_Waiting, Input_regNumber_Waiting_Error* and *Read_Code_Ready*.

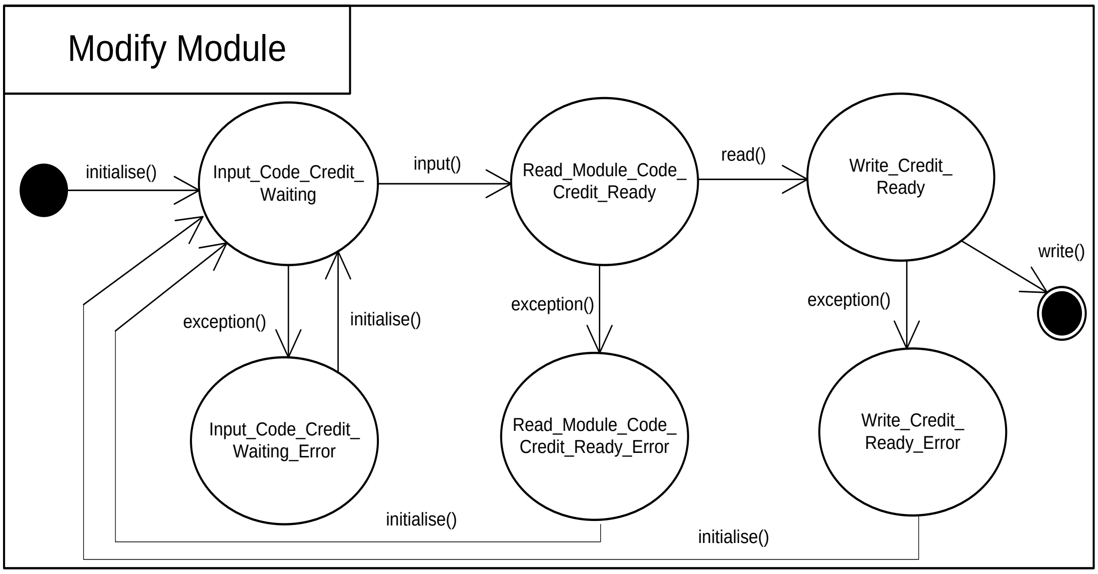



Figure 9.13: State model. Modify Module subtask.

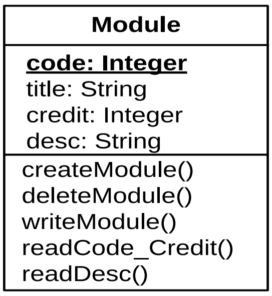


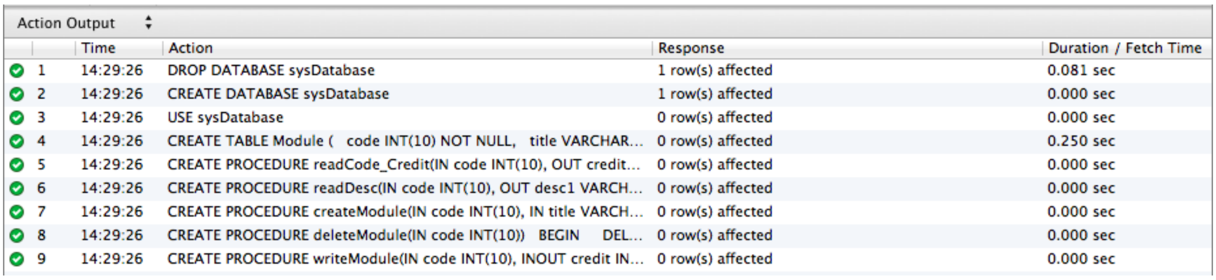

Figure 9.14: DBQ model. Module Entity

Listing 9.4: GUI Model

```
<gui:GuiBoundary id="1" name="Enrol">
    <gui:Window id="2" name="Input_regNumber_Waiting" order="4">
       <gui:Textfield id="3" name="regNumber"/>
       <gui:Button id="4" name="Exception" event="Exception" exit="false"/>
       <gui:Button id="5" name="Input" event="Input" exit="false"/>
    </gui:Window>
    <gui:Window id="6" name="Input_regNumber_Waiting_Error" order="0" error="true">
       <gui:Label id="7" name="input_regNumber_Waiting_Error_warning"
            text="Null value not accepted"/>
       <gui:Button id="8" name="initialise" event="initialise" exit="false"/>
    </gui:Window>
    <gui:Window id="9" name="Read Code_Ready" order="3">
       <gui:Label id="10" name="regNumber" text="regNumber"/>
       <gui:Label id="11" name="code" text="code"/>
       <gui:Button id="12" name="Read" event="Read" exit="false"/>
       <gui:Button id="13" name="Exception" event="Exception" exit="false"/>
    </gui:Window>
</gui:GuiBoundary>
```

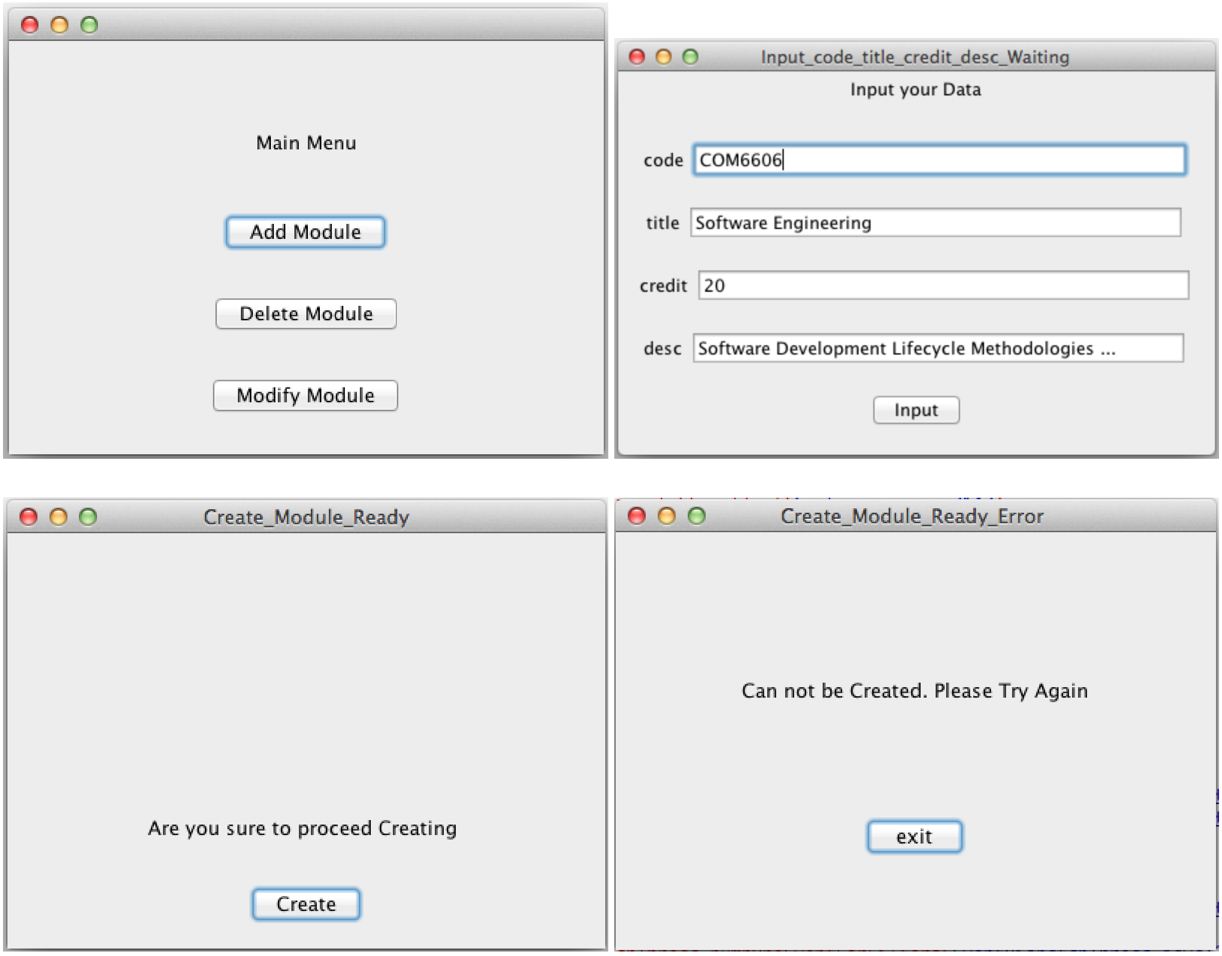
#### 9.3.2.4 Executable Code from the Student Enrolment System

The framework of BUILD generates a number of *Swing Java classes* (*.java) files, and a single *MySQL* dump file, (*.sql) file. A full representation of code is presented in Appendix B. The following images demonstrate the running system screens under Eclipse, as well as the MySQL Workbench 6.1 compiling report after executing the generated *MySQL* script file.



Action Output

| | Time | Action | Response | Duration / Fetch Time |
|---|---|---|---|---|
| 1 | 14:29:26 | DROP DATABASE sysDatabase | 1 row(s) affected | 0.081 sec |
| 2 | 14:29:26 | CREATE DATABASE sysDatabase | 1 row(s) affected | 0.000 sec |
| 3 | 14:29:26 | USE sysDatabase | 0 row(s) affected | 0.000 sec |
| 4 | 14:29:26 | CREATE TABLE Module ( code INT(10) NOT NULL, title VARCHAR... | 0 row(s) affected | 0.250 sec |
| 5 | 14:29:26 | CREATE PROCEDURE readCode_Credit(IN code INT(10), OUT credit... | 0 row(s) affected | 0.000 sec |
| 6 | 14:29:26 | CREATE PROCEDURE readDesc(IN code INT(10), OUT desc1 VARCH... | 0 row(s) affected | 0.000 sec |
| 7 | 14:29:26 | CREATE PROCEDURE createModule(IN code INT(10), IN title VARCH... | 0 row(s) affected | 0.000 sec |
| 8 | 14:29:26 | CREATE PROCEDURE deleteModule(IN code INT(10)) BEGIN DEL... | 0 row(s) affected | 0.000 sec |
| 9 | 14:29:26 | CREATE PROCEDURE writeModule(IN code INT(10), INOUT credit IN... | 0 row(s) affected | 0.000 sec |



## 9.4 Overview of the Student Enrolment Sub-system

The Student Enrolment System enables the users to complete the enrolment process in university. The system has one main actor (*Student*). The actor is able to enrol in a module, using their *registration number* and the module *code* (for simplicity). The task is performed (executed) on a single enrolment at a time.

### 9.4.1 Information System Representation at the Requirements Sketching Phase

In this section the $\mu$ML requirements models that express the enrolment system is presented graphically. The translation activities are then discussed with the highlighting of some key transformation decisions made by the transformation approach. The complete underlying XML representation of all models at this stage is listed in Appendix B.

The development process starts when a business-user expresses the structure of the system using the $\mu$ML *Task Model*, as seen in Figure 9.15 below. The system does one main job, namely, enrolling a student in a module. This business process can be expressed as a single main *Task* called *Enrol*. The human-computer interaction is represented via placing a *Student* actor connected to the *Enrol* business task.

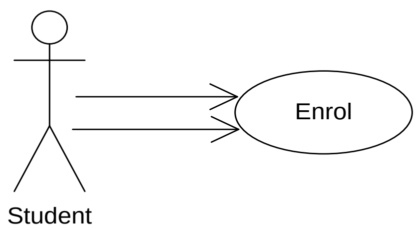



Figure 9.15: Task model. Enrol a Student sub-Sys.

Furthermore, the behaviour of the enrolment system is captured using the $\mu$ML *Impact Model*. The model consists of a task that is equivalent to a business task captured in the *Enrol a Student* in the *Task Model*. The task, in this model, interacts internally with one object namely, *Enrolment*, in which the *Enrol* task is connected to the *Enrolment* entity by a *create* flow. The following figure (Figure 9.16) is the *Impact Model* of the *Enrol a Student* sub-system.

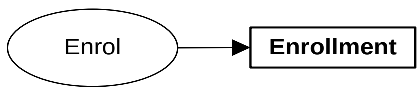



Figure 9.16: Impact model. Enrol a Student sub-Sys.

The following figure (Figure 9.17) is the *Information Model* of the *Enrol a Student*:

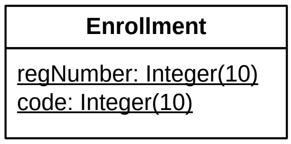



Figure 9.17: Information model. Enrolment Entity

### 9.4.2 Running the Experiment on the BUILD Framework (2)

**Purpose:** Generate a Java Swing information system with MySQL backend database.
**Critical Feature:** Adopting appropriate decisions for handling multiple steps of inputing data to the system.
**Input:** Three $\mu$ML models: Task, Impact and Information Model.
**Output:** Java classes (*.java) and a MySQL script file (*.sql).
**Running Environment:** Eclipse.

#### 9.4.2.1 Construction of Requirements Models

Similar to the previous example, a memory model of the system *Task Model* is constructed first manually by business users. The *Task* class is used to initialise the *Enrol* business task in memory, while the *Actor* and *Paricipation* are used to define the *Student* actor and its interactions to the system respectively. Listing 9.5 below demonstrates the Java code for the construction of the *Task Model* of the *Student Enrolment System* to be used in BUILD.

Listing 9.5: Construction of the Student Enrolment Task Model

```
Diagram taskModel = new Diagram();
Boundary boundary = new Boundary("Enrol a Student");

Actor actor1 = new Actor("Student");
Task enrolStd = new Task("Enrol");

Participation link4 = new Participation();
link4.addRole(new mde.task.model.Role("student",actor1));
link4.addRole(new mde.task.model.Role("enrol", enrolStd));
Participation link3 = new Participation();
link3.addRole(new mde.task.model.Role("student",actor1));
link3.addRole(new mde.task.model.Role("enrol", enrolStd));
```

In a similar way, the *Impact Model* of the system is created manually in memory, using node classes in the *Impact Model Package*. The *ImpTask* class is used to initialise the *Enrol* (*Impact*) task in memory, whereas the *ImpCreateFlow* and *ImpObject* are used to define the create impact and the *Enrolment* business object respectively. Listing 9.6 below demonstrates the construction of the *Impact Model*. It can be seen that names of boundaries and tasks are equivalent to previously designed boundaries and tasks in the *Task Model*, Listing 9.5 above.

Listing 9.6: Construction of the Student Enrolment Impact Model

```
ImpDiagram ImpactModel = new ImpDiagram();
ImpBoundary impboundary = new ImpBoundary("Enrol a Student");

ImpTask impEnrolStd = new mde.impact.model.ImpTask("Enrol");

ImpObject impObj3 = new ImpObject("Enrollment");

ImpCreateFlow cf = new ImpCreateFlow();
ImpRole impcf2 = new ImpRole("enrol", impEnrolStd);
ImpRole impcf1 = new ImpRole("enrollment", impObj3);
cf.addImpRole(impcf2);
cf.addImpRole(impcf1);
```

Furthermore, the *Information Model* is the final model of this stage. The following listing (Listing. 9.7) demonstrates a snapshot of the related part of the *Student Enrolment System*. It is designed using the *Information Model Package* of the BUILD framework. The class *Entity* is used to define the back-end entities of the system.

Listing 9.7: Construction of the Student Enrolment Information Model

```
mde.information.model.Diagram informationModel = new mde.information.model.Diagram();

Entity enrolEntity = new Entity("Enrollment");
Attribute attr10 = new Attribute("regNumber", new Type("Integer"))
.setIdentifier(true);
Attribute attr11 = new Attribute("code", new Type("String")).setIdentifier(true);
enrolEntity.addAttribute(attr10);
enrolEntity.addAttribute(attr11);
informationModel.addEntity(enrolEntity);
```

#### 9.4.2.2 Information System Representation at the Analysis Phase

Similar to the experiment (1), the requirements models of the system (*Task, Impact* and *Information Model*) are produced manually at the requirement stage. A series of translations at this development layer aims at producing three analysis models, namely, the *Data Dependency, DataFlow* and *State Models*. The following sections present the creation of these models.

As this sub-system interacts with only a single entity, the constructed *Data Model* consists of one entity, *Enrolment*. There is no significant decision in this example. The following figure (Figure 9.18) demonstrates the structure of *Enrolment* entity at the *Data Dependency Model*.

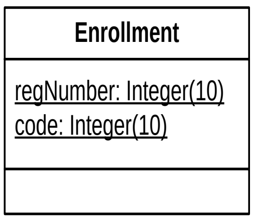



Figure 9.18: Data model. Enrollment Entity

Besides this, as a result of merging *Task* and *Impact Model*, the initial *DataFlow Model* is constructed, containing a single business transaction involving a task, *Enrol*, to receive users inputs and for creating a new enrolment object. It can be noticed that there are two input flows from the *Student* actor to the *Enrol* task. This expresses the multiple steps of entering data into the system without any information about their order. The following figure (Figure 9.19) shows the representation of the initial DFD.

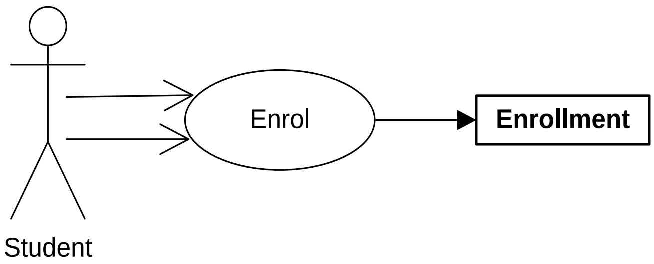



Figure 9.19: (initial) DFD model. Enrol a Student sub-Sys.

After the end-users engagement to supply the model with data on flows, the following figure (Figure 9.20) represents the initial DFD after adding appropriate data on flows.

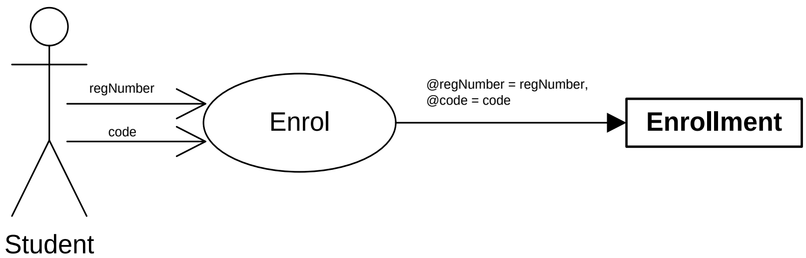



Figure 9.20: (initial) DFD model. Enrol a Student sub-Sys. with data of flows

From the generated DataFlow Model (Figure 9.20), it can be seen that the *Enrol* task is connected by two *input* flows and one *create* flow. This requires a further decompositional step in order to generate a number of atomic tasks that perform the original *Enrol* one.

### Deriving Detailed DataFlow Model

The *intital DataFlow to detailed DataFlow Model* transformation step is applied in order to derive a complete *DataFlow Model* that contains atomic tasks. The interesting transformation decision, in this case, is made to construct a logical boundary to represent the *Enrol* task that contains two *Input* tasks and one *Create* task. These three subtasks are connected to each other by a number of *read* flows.

The direction of flows, between subtasks, is determined by the type of flow that is connected to each task. For example, an extra *read* flow is generated from a task that is connected to the *input* flow (e.g. *Input Code*) An extra read flow is generated to connect *InputCode* to *CreateEnrollment* (the former handles the initial input flow and the latter handles the final create flow; so these must communicate). However, the transformation step that is responsible for generating flows between tasks has a default rule applied to the model when they have two or more *input* flows. Tasks are ordered in the DataFlow Model, based on the order in which the input flows were entered in the Task Model. In this case, the task Input RegNumber precedes the task InputCode. The following figure (Figure 9.21) demonstrates the detailed DFD:

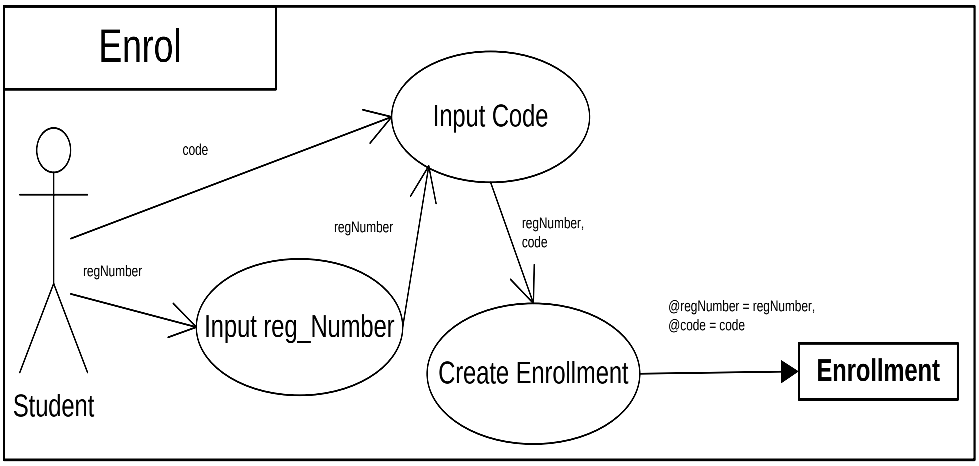



Figure 9.21: (detailed) DFD model. Manage Module Sys.

Similar to the previous case study (section 9.3), the *State Model* is derived from the detailed DFD model. The *Enrol* DFD boundary is mapped directly to an equivalent state boundary, called *Enrol*, in boundary within the target model. It contains three states (two are *waiting* states and one is *ready*) that are derived from the *DataFlow Inputs* and *Create* flow respectively.

Furthermore, a number of transitions, with appropriate actions and/or condition(s), are added to the model, reflecting the right sequence of navigation of system screens to perform the original task. Similar to a previous case study (section 9.3), a separate in-place modification is applied to the constructed *State Model* to generate error handling (reporting) states for each *State* in the boundary, see Figure 9.22 below.

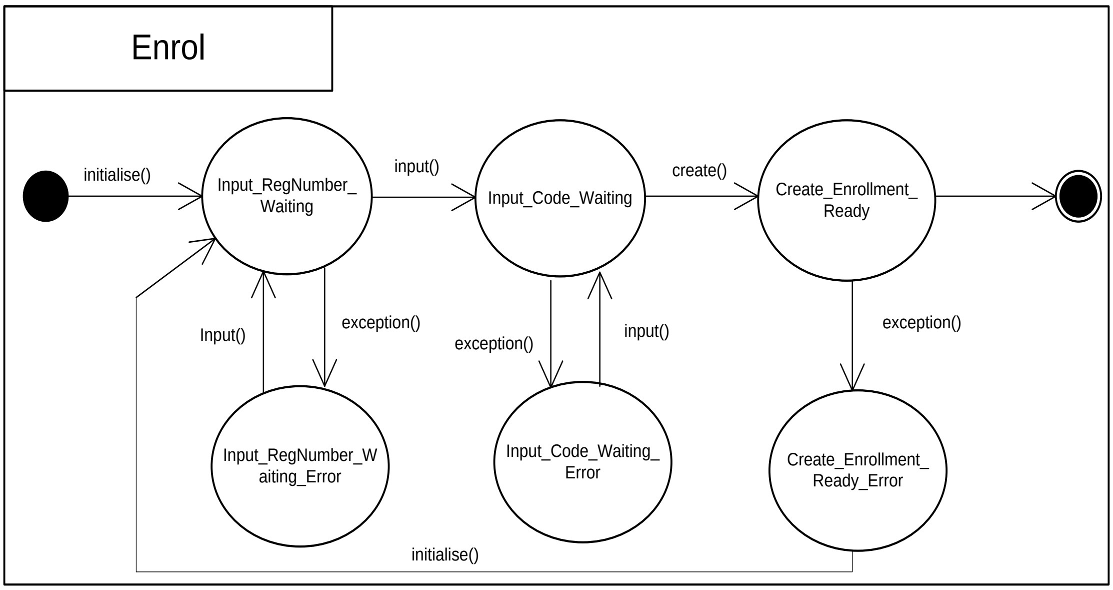



Figure 9.22: State model. Manage Module Sys.

From the above transformations, the construction of all required analysis artifacts (*Data, DataFlow* and *State Model*) is accomplished. These models are ready to be utilised as source models in the next *Design Phase* of the framework.

### 9.4.3 Information System Representation at the Design Phase

The *Design Phase* in BUILD aims at producing the *Database and Query and GUI Model* from the intermediate analysis models, constructed above. The following subsections represent both models at the design level of abstraction.

While the system deals with a single business entity, there is no significant transformation decision, from *Data* to *Database and Query Model*, in this case study. The following figure (Figure 9.23) demonstrates the produced DBQ model at the *Analysis Phase* of BUILD. The *Enrolment* entity expresses the structure of a relational database table in a platform-independent way.

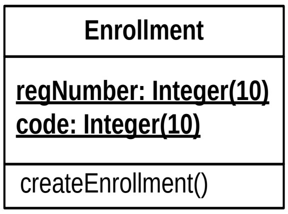



Figure 9.23: DBQ model. Enrollment Entity

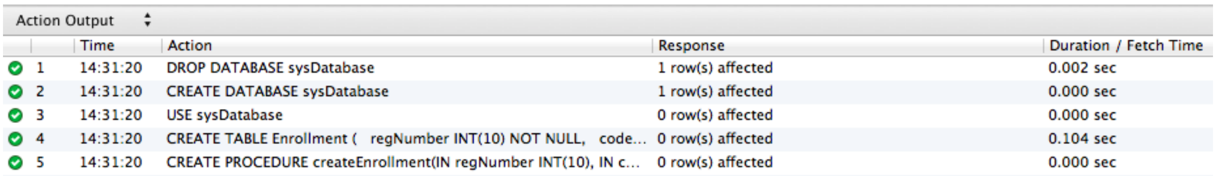
In a similar way to the previous experiment, this model is constructed mainly from the analysis *State Model*. The following listing (Listing. 9.8) demonstrates a snapshot of the *GUI Model*, including the specifications of two windows, namely, *Create_Enrollment_Ready* and *Create_Enrollment_Ready_Error*.

Listing 9.8: GUI Model

```
<gui:GuiBoundary id="1" name="Enrol">
  <gui:Window id="16" name="Create Enrollment_Ready" order="2">
        <gui:Label id="17" name="regNumber" text="regNumber"/>
        <gui:Label id="18" name="code" text="code"/>
        <gui:Button id="19" name="Create" event="Create" exit="false"/>
        <gui:Button id="20" name="exit" event="exit" exit="true"/>
        <gui:Button id="21" name="Exception" event="Exception" exit="false"/>
  </gui:Window>
  <gui:Window id="22" name="Create Enrollment_Ready_Error" order="0" error="true">
        <gui:Label id="23" name="Create Enrollment_Ready_Error_warning"
             text="Connection to the Data source is fail"/>
        <gui:Button id="24" name="exit" event="exit" exit="false"/>
  </gui:Window>
  ...
</gui:GuiBoundary>
```

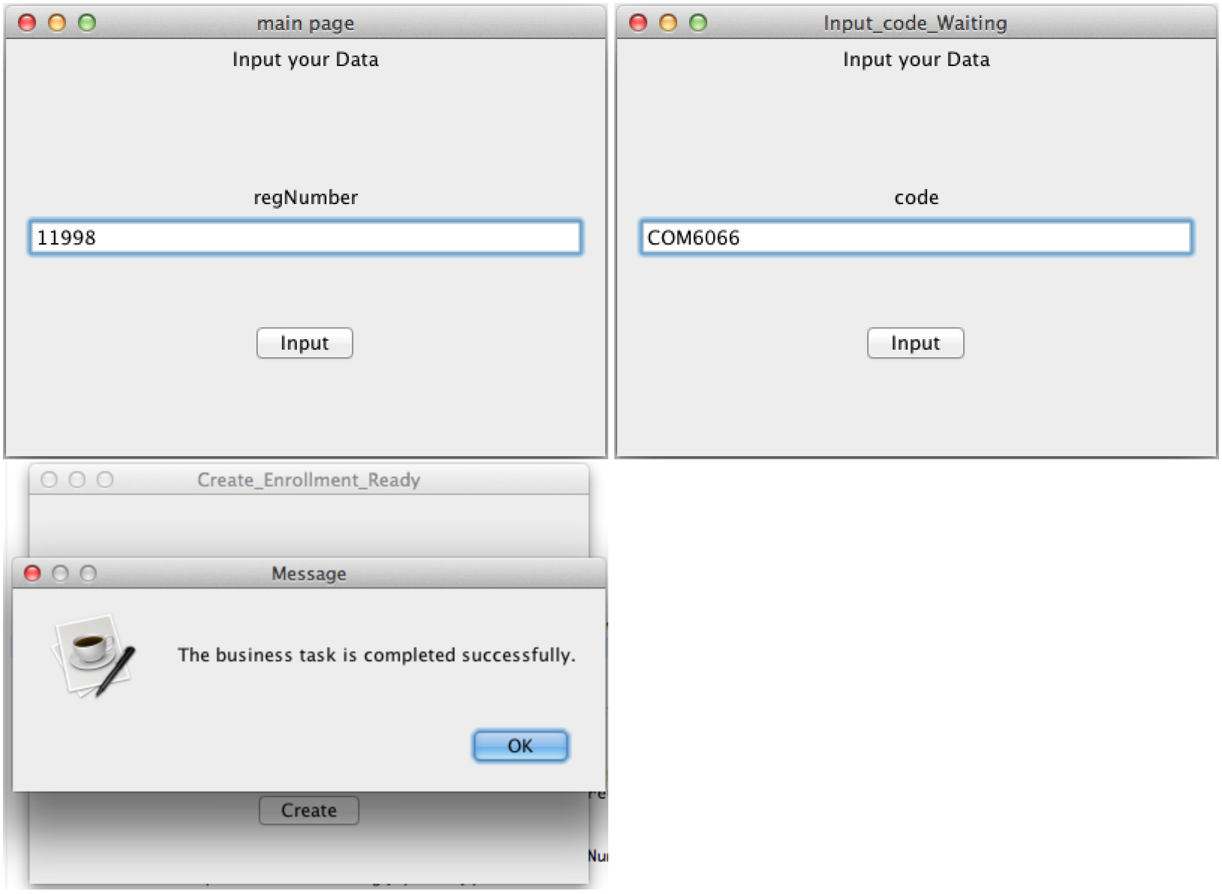
#### 9.4.3.1 Executable Code from the Student Enrolment System

The framework of BUILD generates a number of *Swing Java classes*, (*.java) files, and a single *MySQL* dump file, (*.sql) file. A full representation of code is presented in Appendix B. The following images demonstrate the running system screens under Eclipse, as well as the MySQL Workbench 6.1 compiling report after executing the generated *MySQL* script file.



Action Output

| | Time | Action | Response | Duration / Fetch Time |
|---|---|---|---|---|
| 1 | 14:31:20 | DROP DATABASE sysDatabase | 1 row(s) affected | 0.002 sec |
| 2 | 14:31:20 | CREATE DATABASE sysDatabase | 1 row(s) affected | 0.000 sec |
| 3 | 14:31:20 | USE sysDatabase | 0 row(s) affected | 0.000 sec |
| 4 | 14:31:20 | CREATE TABLE Enrollment ( regNumber INT(10) NOT NULL, code... | 0 row(s) affected | 0.104 sec |
| 5 | 14:31:20 | CREATE PROCEDURE createEnrollment(IN regNumber INT(10), IN c... | 0 row(s) affected | 0.000 sec |




## 9.5 Overview of the Extended Information Model of the University Administration System

This section presents a fairly complex conceptual data model for the University Administration System case study adopted in this chapter. The model consists of a number of information entities and various types of associations, generalisations, and one kind of composition. In addition, a *range constraint* is applied to an attribute in order to illustrate how the transformation may react and makes alternative decisions compared to unrestricted attributes.

Two ways for constructing the *Data Dependency* and *DBQ Models* are followed and the results are compared. Starting from the $\mu$ML *Information Model* is one possible way for starting the transformation chain till the *Data* and *DBQ Model* is achieved. The second way is by deriving automatically the initial *Information Model* from the *Impact Model* that is provided manually by business users. The following figure (Figure 9.24) illustrates the extended *Information Model* of the *University Administration System*.

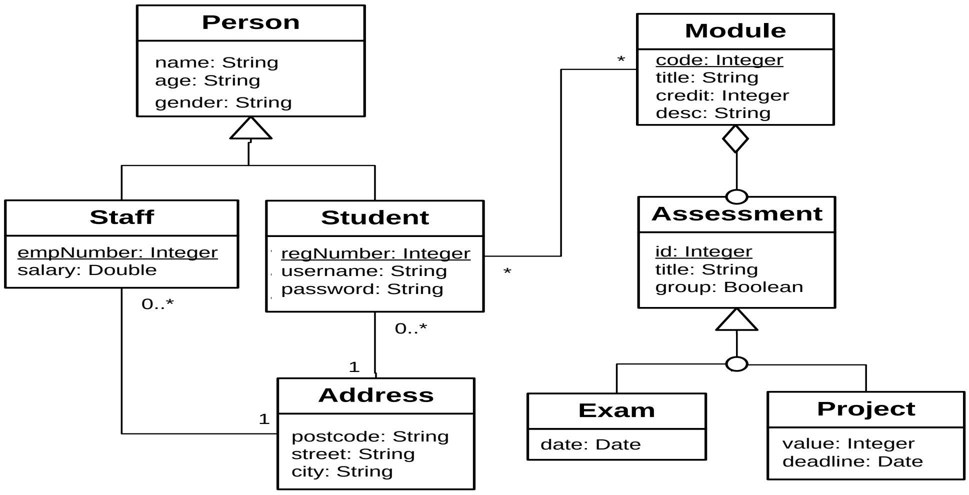



Figure 9.24: Information model. University Administration Sys.

### 9.5.1 Running the Experiment on the BUILD Framework (3a)

**Purpose:** Generate a MySQL database system.
**Input:** one $\mu$ML model: Information Model.
**Output:** a MySQL script file (*.sql).
**Running Environment:** MySQL.

The *Information Model* of the *University Administration System* is used to demonstrate the capability of the BUILD framework to generate a complete executable *MySQL* script from a very abstracted information model. In order to achieve this, two steps of model-to-model translation are required, namely, *Information to Data Dependency Model* and *Data Dependency to Database and Query (DBQ) Model.* Moreover, a further code generation step is also applied to generate the final *MySQL* code from the $\mu$ML DBQ schema.

#### 9.5.1.1 Information Model Representation at the Requirements Sketching Phase

Similar to previous experiments, the *Information model* is constructed manually in BUILD as an Abstract Syntax Tree (AST). A snapshot of the specification of the entities and attributes are expressed in the Listing 9.9 below (Person entity only), which represents the Java code used to create the model in memory (see Appendix A for the full code). The *Entity* and *Attribute* classes are used to construct the structure of the business entities and their properties, whereas the *Association, Generalisation* and *Composition* classes are used to define a variety of relationships between entities. The *range constraint* is specified using the *setLowerbound()* and *setUpperbound()* methods of the *Attribute* object.

Listing 9.9: Construction of the Complex Information Model (Person Entity)

```
mde.information.model.Diagram informationModel = new
    mde.information.model.Diagram();

Entity personEntity = new Entity("Person");
Attribute attr2 = new Attribute("name", new Type("String"));
Attribute attr3 = new Attribute("age", new Type("String"))
    .setLowerbound(18).setUpperbound(35);
Attribute attr4 = new Attribute("gender", new Type("String"));

personEntity.addAttribute(attr2);
personEntity.addAttribute(attr3);
personEntity.addAttribute(attr4);

informationModel.addEntity(personEntity);
```

#### 9.5.1.2 Information Model Representation at the Analysis Phase

At the analysis phase of BUILD, the *Data Model* is constructed (Figure 9.25). It includes some significant translation decisions made by the *Information to Data Model* translator. The many-to-many association between *Student* and *Module* is promoted to an entity *Enrolment* which depend on its related entities. In addition to this, the many-to-one association between *Student* and *Address* is resolved in the direction from the many side (*Student*) to the one side (*Address*) in the *Date Model*.

Furthermore, the *Generalisation* relationship between *Person, Staff* and *Student* is resolved by making *Student* and *Staff* (the specific entities) depends on the *Person* (the general entity). On the other hand, the *disjoint Generalisation* between *Assessment* and *Assessment_Exam* is resolved by merging the general entity (*Assessment*) into each specific one (*Assessment_Exam* and *Assessment_Project*). Moreover, the *Total Composition* relationship between *Module* and *Assessment* is resolved by making the part entity (*Assessment*) dependant on the whole one (*Module*).

#### 9.5.1.3 Data Dependency Representation at the Design Phase

The evolution of the *Data Dependency Model* at the *Design Phase* in BUILD is the *Database and Query (DBQ) Model*, visualised in Figure 9.26 below. The generated DBQ model is considered a final detailed model that expresses a relational database schema in such a generic representation in a higher level of abstraction. All *Dependency* relationships between entities are translated into *Foreign Keys* (*references*).

There is a standard rule in the *Data Model to DBQ Schema* that translate any *range constraint* applied to a particular attribute into a *Trigger* associated to the table that holds that attribute. This is expressed in the following DBQ schema (Figure 9.26) in the third partition of the *Person* entity.

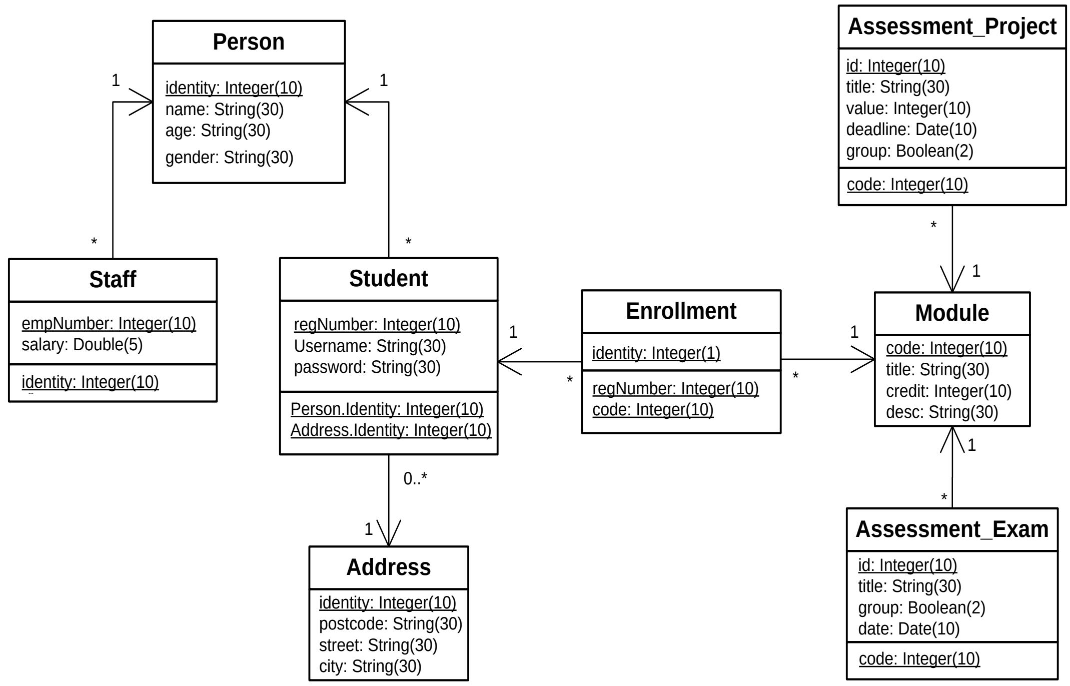



Figure 9.25: Data model. University Administration Sys.

#### 9.5.1.4 Executable MySQL Code from the Database and Query (DBQ) Schema

Unlike previous experiments, the code in this example is only a single (*.sql) file. The current version of BUILD has a domain-specific code generation framework (*MySQL*). The following code presents a snapshot of the final database schema in *MySQL*, representing the structure of the *Person* table and its associated *Trigger*. The complete schema is presented in Appendix C.

Listing 9.10: *database_MySQL.sql*

```
CREATE DATABASE sysDatabase;
USE sysDatabase;

-- Structure for table 'Person'

CREATE TABLE Person (
  identity INT(10) NOT NULL,
  name VARCHAR(30),
  age VARCHAR(30),
  gender VARCHAR(30),
  PRIMARY KEY(identity));

-- Trigger: Applying Checking Constraints on table 'Person'

DELIMITER //

DROP TRIGGER IF EXISTS 'personCheck' //
CREATE TRIGGER 'personCheck' BEFORE INSERT ON Person
    FOR EACH ROW
      IF (NEW.age < 18 OR NEW.age > 35) THEN
        SET msg = 'INVALID DATA IN age';
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = msg;
      END IF; //
```

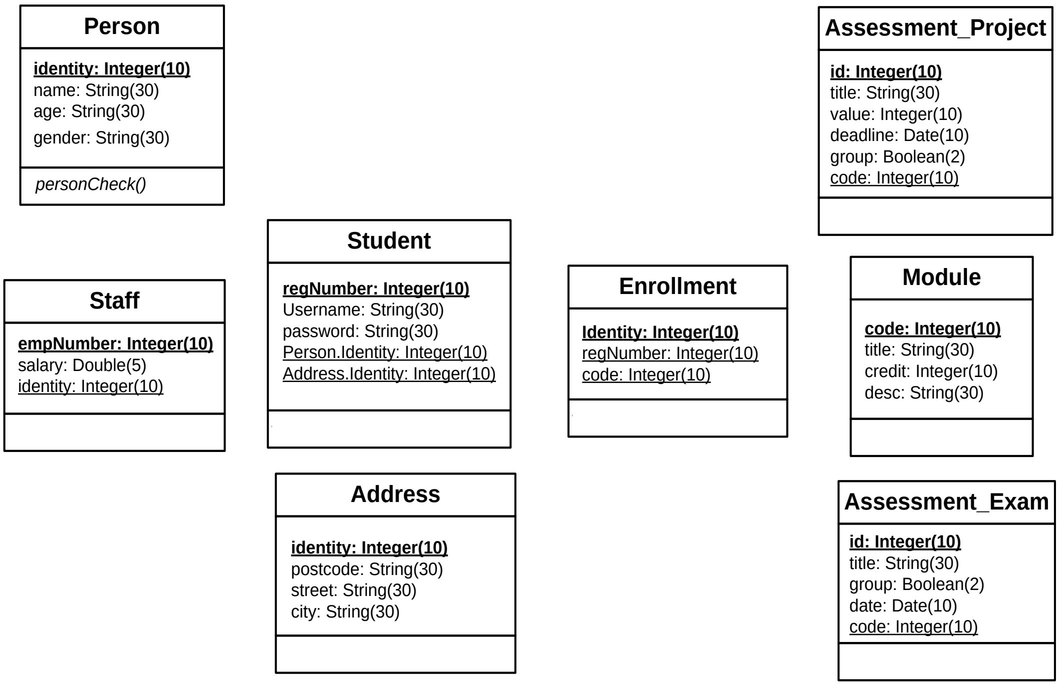



Figure 9.26: DBQ model. University Administration System

#### 9.5.1.5 Running the Experiment on the BUILD Framework (3b)

**Purpose:** Generate a MySQL database system.
**Input:** one $\mu$ML model: Impact Model.
**Output:** a MySQL script file (*.sql).
**Running Environment:** MySQL.

A possible *Impact Model* of the *University Administration System* is designed to be used for deriving an initial *Information* and *Data Model* and then generating *MySQL* code. In order to achieve this, a further model-to-model translating step is required, which is *Impact-to-Information Model*.

#### 9.5.1.6 Information Model Representation at the Requirements Sketching Phase

The *Impact Model* is provided manually, by users, in BUILD as an AST. The suggested model consists of a number of tasks that interact with system entities. Appendix C contains the complete *Java* code for creating the *Impact Models* using *JAST*[102].

**Deriving the Initial Information Model**

At the requirement gathering level, the initial *Information Model* might be derived from the pre-defined *Impact Model*. The model consists of a number of information *Entities* and their *predicted* relationships. This translation step is carried by an *Impact-to-Information Model* translation agent (rule). The rules are discussed previously in Chapter 7. Figure 9.27 below illustrates the generated initial *Information Model*.

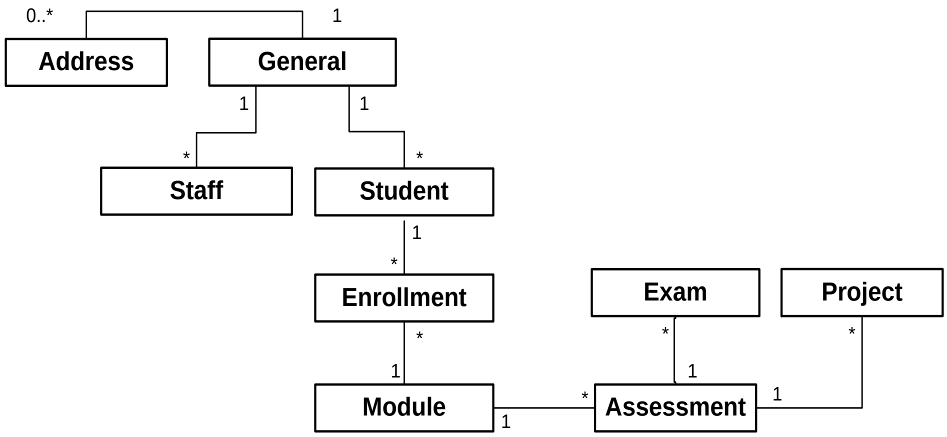



Figure 9.27: The Initial Information Model

According to the *Impact-to-Information Model* step, each object in the *Impact Model* is used to generate a corresponding *Entity* in the *Information Model*, without any predicted details about its attributes. Business users might supply this information later to the generated model in an independent step, similar to the step of annotating the initial DFD model. Besides this, after tracing the CRUD effect for each task in the *Impact Model*, the relationships with appropriate multiplicities are predicted.

It is useful to compare the *Information Model* provided in experiment (3a) with the *Information Model* provided in experiment (3b) to examine the similarities and differences between them. All entities appearing in model (3a) are generated in the second one in (3b), except the *Enrolment* entity in which the *many-to-many Association* between *Student* and *Module* is captured as two *many-to-one* relationships from the *Impact Model*. This difference is in fact the result of normalising the *many-to-many* relationship, which would happen in the next step anyway. This shows that it is possible to derive normalised data models directly from the *Impact Model*.

Furthermore, a possible *Generalisation* relationship is predicted between the *Student* and *Staff* object, in the *Impact Model*. Consequently, a *general* entity is manufactured and a number of many-to-one associations between the specific entities (many sides) and the general one (*one side*) are generated directly between them. As before, the *Total Composition* relationship between the *Model* and its parts *Exam* and *Project* entity are also predicted as a number of *many-to-one Associations*, where the parts are connected on the *many-side* of these associations.

#### 9.5.1.7 Information Model Representation at the Analysis Phase

At the *Analysis* stage, the typical translation rules are applied by the *Information-to-Data Model* agent to produce the *Data Dependency Model*, (Figure 9.28) below. To avoid the repetition, the generated *DBQ Model* and *MySQL* script are described in Appendix C containing the representation of the (3b) *DBQ Model* and its generated *MySQL* database schema.

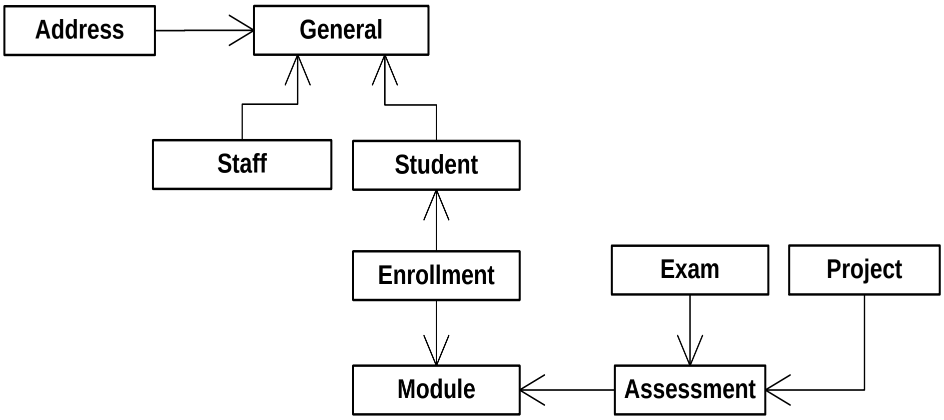


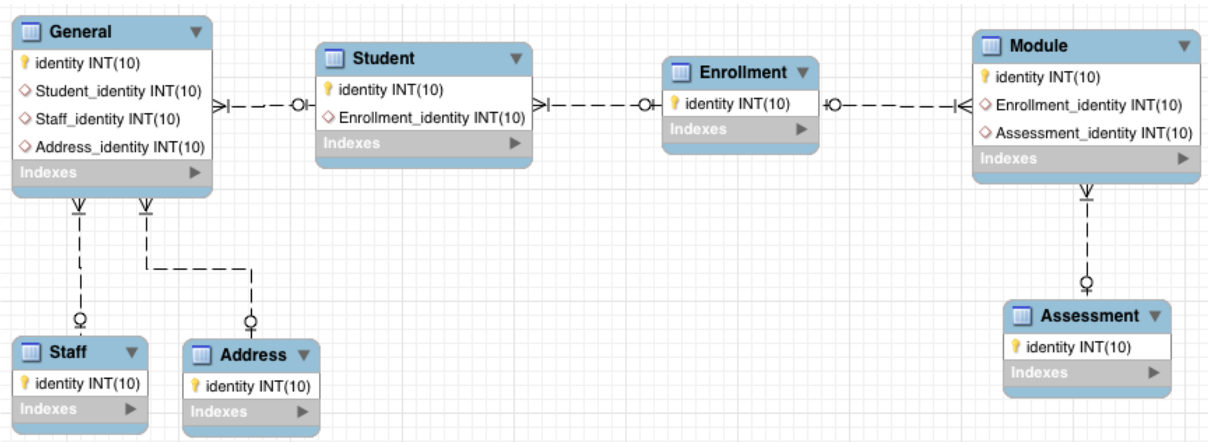

Figure 9.28: The derived Data Dependency Model

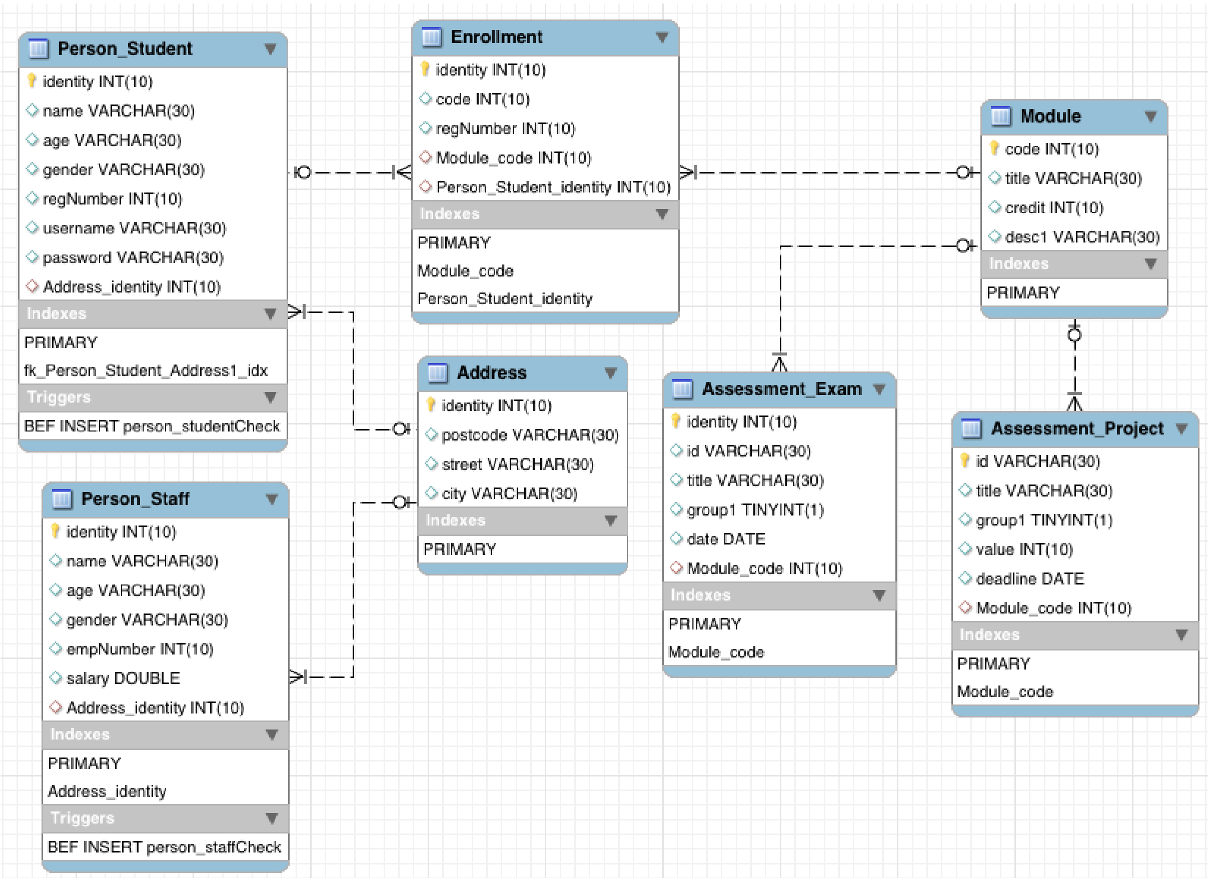
After compiling the generated *MySQL* script file into the MySQL Workbench 6.1 server, the following result is obtained:

Action Output

| | Time | Action | Response | Duration / Fetch Time |
|---|---|---|---|---|
| 1 | 21:46:34 | DROP DATABASE sysDatabase | 7 row(s) affected | 0.053 sec |
| 2 | 21:46:34 | CREATE DATABASE sysDatabase | 1 row(s) affected | 0.000 sec |
| 3 | 21:46:34 | USE sysDatabase | 0 row(s) affected | 0.000 sec |
| 4 | 21:46:34 | CREATE TABLE Module ( code INT(10) NOT NULL, title VARCHAR... | 0 row(s) affected | 0.074 sec |
| 5 | 21:46:34 | CREATE TABLE Address ( identity INT(10) NOT NULL, postcode... | 0 row(s) affected | 0.198 sec |
| 6 | 21:46:34 | CREATE TABLE Person_Student ( identity INT(10) NOT NULL, na... | 0 row(s) affected | 0.145 sec |
| 7 | 21:46:34 | CREATE TABLE Person_Staff ( identity INT(10) NOT NULL, name... | 0 row(s) affected | 0.135 sec |
| 8 | 21:46:34 | CREATE TABLE Assessment_Exam ( identity INT(10) NOT NULL,... | 0 row(s) affected | 0.158 sec |
| 9 | 21:46:34 | CREATE TABLE Assessment_Project ( id VARCHAR(30) NOT NULL,... | 0 row(s) affected | 0.155 sec |
| 10 | 21:46:34 | CREATE TABLE Enrollment ( identity INT(10) NOT NULL, code IN... | 0 row(s) affected | 0.168 sec |
| 11 | 21:46:35 | DROP TRIGGER IF EXISTS person_studentCheck | 0 row(s) affected, 1 warning(s): 1360 Trigger does n... | 0.000 sec |
| 12 | 21:46:35 | CREATE TRIGGER person_studentCheck BEFORE INSERT ON Person... | 0 row(s) affected | 0.329 sec |
| 13 | 21:46:35 | DROP TRIGGER IF EXISTS person_staffCheck | 0 row(s) affected, 1 warning(s): 1360 Trigger does n... | 0.000 sec |
| 14 | 21:46:35 | CREATE TRIGGER person_staffCheck BEFORE INSERT ON Person_St... | 0 row(s) affected | 0.278 sec |

Using the *reverse engineering* facility in the MySQL Workbench 6.1 server, we can reconstruct the database schema of the generated and executed database system we run. From that, the following diagrams demonstrate the reverse engineered data model derived from the Impact model and the user-defined information model respectively.






### Business User Made Information Model v.s Generated (initial) Information Model

When considering the user-defined *Information Model* (Figure 9.24) and the derived one, from the *Impact Model* (Figure 9.27), we can notice that the generated model contains all entities appearing in the manually constructed one. However, unlike the first *Information Model*, entities in the second model (derived) have no detail about their attributes. This occurs because there is insufficient information in the *Impact Model* to enable translation agents to translate or predict and then manufacture appropriate attributes for each entity. An additional entity appears in the generated *Information Model*, from the *Impact* one, as a result of capturing the *many-to-many* association via two impacts on the *Module* and *Student* object, in the *Impact Model*. Each one represents a single *many-to-one* association. In addition to this, the parts *Exam* and *Project* are not expressed in the *Impact Model*, see Appendix C.

Furthermore, *aggregation* relationships are predicted as *many-to-one* associations between the whole entity and its associated parts. Besides this, it can be seen that *generalisation* relationships are also captured via the existence of the *Conjunction* concept in the model and directly resolved into *many-to-one* associations between the entities involved.

After completing the whole development process that produces the final MySQL schema, the number of tables in the user-defined model is fewer than the number of tables in the derived model, due to the accuracy of resolving the *disjoint generalisation* relationship, in which the *general* and *specific* table will be merged.

## 9.6 Outlook on the Chapter

A real-world enterprise information system case study (*University Administration System*) was presented in this chapter. Three experiments were designed, each involving a particular part of the system. Requirements models, namely, *Task, Impact* and *Information Models* were expressed, manually, using the graphical notation of $\mu$ML.

For each case study, the chapter represents the development stages of each model from the requirements sketching level to the code generation one. All significant transformation decisions are highlighted, showing how the approach generates an independent menu screen in the first experiment, whereas it manufactures a sequence of input forms (screen) for allowing a user to insert inputs step by step.

Throughout the demonstrated results, it can be seen that the designed chain of transformations is able to generate complete basic information systems that are connected to backend relational databases. The generated system is platform-specific and is tested within Eclipse. The produced information system is implemented in *Java* and the related database is generated in *MySQL.* The generated *Java* code includes a *JDBC* connection statement for establishing the required connection to the backend database. Full listings of these experiments are given in Appendices A, B and C.

# 10

# Evaluation and Testing

"Discovering the unexpected is more important than confirming the known"

George Box (1919- 2013)

## 10.1 Context

The main theme of this thesis is to introduce a simple and semantically cleaned modelling language ($\mu$ML), and a MDE framework, which is suitable for end-users, to ease and accelerate the capability of developing small/medium scale information systems. In order to achieve this ambition, the proposed language must be able to express, in a simple and clear way for business end-users, all critical concepts in the information systems domain. In addition to this, the transformation approach must be able to lead the development of information systems from requirements to the final executable code. As a result, $\mu$ML and BUILD are introduced in the previous chapters of this thesis.

This chapter tries to emphasise the completeness and correctness of the output results from BUILD. To make sure $\mu$ML is capable of modelling the most common enterprise system concepts we have chosen to evaluate it against the generated objects and chunks of code that appear in the final results. Model translation and code generation for producing *JDBC Java Swing Applications* with a *MySQL* backend database, in the BUILD Framework, are used to verify and inspect the evolving of models from the requirement phase to the final one that generates the executable code.

Furthermore, the *time efficiency* of transformation algorithms is evaluated to measure the quality and scalability of the proposed framework. The Big-O notation technique is adopted, in this chapter, for articulating how long an algorithm takes to run, and to examine whether or not there exist some factors or limits on inputs that affect the growth rate of time in some transformation steps. Apart from that, the suitability of the proposed $\mu$ML is also evaluated by conducting an experiment with selected university students from different disciplines. It aims at evaluating $\mu$ML in terms of its simplicity and ease to adopt by end-users as a modelling language for expressing basic specifications of information systems.

## 10.2 Evaluating The Generated Results from BUILD

### 10.2.1 Assuring the Determinism of the Transformation Rules

This criterion emphasises the capability of the framework to restrict transformations to the appropriate type of input element, and to produce one specific type of result. There are two types of transformation rules provided in BUILD, namely, one-to-one (forward) rule and two-to-one (merge) mapping rule. When a particular concept appears in a model, it will be accepted only by one or more approved rules to produce a particular target concept(s). This means that the mapping is controlled by a precondition(s) that must be met before executing any translation rule, via the (*accept()*) method (see Chapter 7).

This strategy also ensures the right treatment of concepts (based on their types and semantics), in which it is impossible to misinterpret any concept chosen by a business user, and produce an unexpected target element. However, it leads to increasing the number of translation agents, as there is an agent for each concept to ensure that a correct decision is made.

For example, handling composed business tasks, in the *Task Model*, is different from the atomic ones, in which each task type is determined by checking whether it is attached to the head of any kind of *Composition* relationship or not. The composed task is normally ignored at the first translation step (constructing the initial DFD model), whereas atomic tasks are merged with the equivalent tasks that appear in the *Impact Model*, and converted into *DataFlow* tasks, and then to *State Model* transitions. Table 10.1 shows translation rules that are applicable for each *Task Model* concept.

| **Task Model Concept** | **Translator Agents** |
|---|---|
| **Diagram** | DiagramToDfDiagram |
| **Boundary** | BoundaryToDfBoundary |
| **Role** | RoleToDfRole |
| **Task (atomic)** | MergeTaskToDfTask |
| **Actor** | ActorToDfActor |
| **Input Participation** | ParticipationToDfInputFlow |
| **Output Participation** | ParticipationToDfOutputFlow |
| **Composition** | - - |

Table 10.1: Task Model concepts and Related Agents

In the *Impact Model*, for example, each type of *impact* is translated differently into its corresponding type of *flows* in the *DataFlow Model*. For example, *Create Impact* is converted into a *DfCreateFlow* that has more specific features (e.g. *assignment*) than the *DfReadFlow*, for instance, which has *filters*. Table 10.2 shows translation rules that are applicable for each *Impact Model* concept.

| Impact Model Concept | Translator Agents |
|---|---|
| **ImpDiagram** | DiagramToDfDiagram |
| **ImpBoundary** | BoundaryToDfBoundary |
| **ImpRole** | ImpRoleToDfRole |
| **ImpTask** | MergeTaskToDfTask |
| **Create Impact** | CreateFlowToDfCreateFlow |
| **Read Impact** | ReadFlowToDfReadFlow |
| **Update Impact** | UpdateFlowToDfUpdateFlow |
| **Delete Impact** | DeleteFlowToDfDeleteFlow |

Table 10.2: Impact Model concepts and Related Agents

When considering the translation of the *Association* concept in *Information Model*, for instance, it can be noticed that it might be treated in three different ways based on its specific meaning (m-to-1, m-to-n, or 1-to-1). The *m-to-1 Association* is translated into a *Dependency* relationship in *Data Model*. Whereas *m-to-n* and *1-to-1 Associations* are converted into *Linker* and *Merged Entity* respectively. *Generalisation* and *Compostion* are treated likewise. Table 10.3 shows translation rules that are applicable for each *Information Model* concept.

| Information Model Concept | Translator Agents |
|---|---|
| **InfDiagram** | InfDiagramToDDiagram |
| **Entity** | InfEntityToDEntity |
| **Association (m-to-1)** | InfAssocToDDependency |
| **Association (1-to-1)** | InfAssocToMergedTable |
| **Association (m-to-n)** | InfAssocToLinkerTable |
| **Generalisation (overlapping)** | InfGenToDDependency |
| **Generalisation (disjoint)** | InfGenToMergedTable |
| **Composition (total)** | InfComToMergedTable |
| **Composition** | InfCompToDDependency |

Table 10.3: Information Model concepts and Related Agents

These *Dependency* relationships are resolved in the next translation step into a number of *Foreign Keys* in the *Database and Query (DBQ) Model*. The generated FKs represents the relationship between the business entity at the *Design* phase for generating a relational database system. Therefore, the only meaning of *Foreign keys* is to represent a relationship between the entity that holds the key and the one it *refers* to. Table 10.4 shows translation rules that are applicable for each *Data Dependency Model* concept.

| Data Model Concept | Translator Agents |
|---|---|
| **DDiagram** | DDiagramToSchema2 |
| **DEntity** | DEntityToTable, DEntityToPrimaryKey |
| **DAttribute** | DAttributeToColumn |
| **Dependency** | DependToReferTable |
| **DRole** | DRoleToForeignKey |

Table 10.4: Data Dependency Model concepts and Related Agents

Regarding the construction of *Procedure* concept in the *DBQ Model*, it can be seen how the importance of the type of corresponding *Flows* in the detailed DFD model. For example, any DFD *create* flow is translated into a DBQ representation of a *Procedure* that performs *insert* database operation. There is no other way to produce *insert (create)* DBQ procedure. Table 10.5 shows translation rules that are applicable for each *DataFlow Model* concept.

| DataFlow Model Concept | Translator Agents |
|---|---|
| **DfdDiagram** | DfDiagramToDfDiagram, DfDiagramToStDiagram |
| **DfdBoundary** | DfdBoundaryToStBoundary2 |
| **Flow** | ArcToArc, ArcToArcs, ArcToDfActor, ArcToDfObject, ArcToDfTask, ArcToDfTasks, ArcToStReadyState, ArcToStWaitingState, CreateFlowToStoredProcedure, ReadFlowToStoredProcedure, UpdateFlowToStoredProcedure, DeleteFlowToStoredProcedure |
| **DfTask** | DfTaskToDfBoundary, DfTaskToTransistion |

Table 10.5: DataFlow Model concepts and Related Agents

In addition to this, concepts in the *State Model* are treated in a direct way, in which the internal architecture of its relevant translation agent is simple. Based on the type of *State*, *Ready* or *Waiting*, the decision of firing either the *SvariableToLabel* or *SvariableToTextfield* is made to translate the variables of this state into suitable GUI controls. Table 10.6 shows translation rules that are applicable for each *State Model* concept.

| State Model Concept | Translator Agents |
|---|---|
| **StDiagram** | StDiagramToGuiDiagram |
| **StBoundary** | StBoundaryToStBoundary, StBoundaryToGuiBoundary |
| **State** | StateToWindow |
| **Transition** | TransitionToButton |
| **State Variable** | SvariableToLabel, SvariableToTextfield |

Table 10.6: State Model concepts and Related Agents

In regard to the *GUI Model* that is used at the *Code Generation* phase, each type of widget element is translated into the corresponding *Swing Java* control for that element, by a particular code generator agent. For example, the specification of *Java Labels* are generated from *GUI Label* elements in the *GUI Model* by the *JavaLabelGenerator*, using approved specifications by the transformation approach (e.g. font type, size, colour, and text), and the rest is likewise.

Furthermore, as each generated system screen, there is a main button that performs that main task associated with that screen. The *Java* code of the button is generated via the *JavaButtonGenerator*, in which the agent fills the body of the *actionPerformed* method of that screen by ***boilerplate code***. This is approved by the generator agent, to establish a suitable JDBC connection, invoke the related stored procedure that performs a particular database operation, or presents user interactions by passing their inputs to the system. This strategy is applicable for all types of screens that might be generated using BUILD.

Additionally, the ***boilerplate code*** for error handling is also considered as a part of the *actionPerformed* method, using the *try-catch* expression and a number of appropriate *Exceptions* to avoid unexpected behaviour of the generated system. Table 10.7 shows generation rules that are applicable for each *GUI Model* concept.

| GUI Model Concept | Generator Agents |
|---|---|
| **GuiBoundary** | JavaCodeFileGenerator |
| **Window** | JavaClassGenerator |
| **Textfield** | JavaTextFieldGenerator |
| **Label** | JavaLabelGenerator |
| **Button** | JavaButtonGenerator |

Table 10.7: GUI Model concepts and Related Agents

Besides generating *JDBC Swing Java* code, the *Code Generation* framework also generates suitable *MySQL* scripts from the DBQ model in BUILD. Each concept appearing in the DBQ model has a particular generator agent that is responsible for generating a specific portion of the *MySQL* Data Definition Language (DDL) or Data Manipulation Language (pre-defined queries). Table 10.8 below illustrates this.

*Stored Procedures* in *MySQL* can be generated from either a DBQ *Create, Update, Delete* or *Query* concept. The difference between the procedure derived from the DBQ *create* and the one generated from the DBQ *Query* for instance, is in the *MySQL* statement (command). Therefore, stored procedures that bind the *MySQL INSERT* statement are generated only from the DBQ procedures that have a *create* concept, in contrast to the ones that bind *MySQL SELECT* query that are derived from the DBQ procedures that have a *Query* element. The rest are likewise.

| DBQ Model Concept | Generator Agents |
|---|---|
| **Schema** | MySQLDumpFileGenerator, MySQLSchemaGenerator |
| **Table** | MySQLTableGenerator |
| **Column** | MySQLFieldGenerator |
| **PrimaryKey** | MySQLPKGenerator |
| **ForeignKey** | MySQLFKGenerator |
| **Procedure** | MySQLStoredProcGenerator |
| **Range constraint** | MySQLTriggerGenerator |
| **Filter** | MySQLQueryGenerator, MySQLUpdateGenerator & MySQLDeleteGenerator |
| **Project** | JavaButtonGenerator |
| **Query** | MySQLQueryGenerator |
| **Create** | MySQLCreateGenerator |
| **Update** | MySQLUpdateGenerator |
| **Delete** | MySQLDeleteGenerator |

Table 10.8: DBQ Model concepts and Related Agents

### 10.2.2 Was it Possible to trace the Development of Models during the Development Stages?

This criterion demonstrates the capability to trace each intermediate artifact and its evolution after completing every translation and/or transformation step. One of the main claims of the $\mu$ML is its suitability for business users, in which it is designed to enable them, with their limited technical knowledge, to express their system functionalities in a very generic and abstracted way. These semantically cleaned $\mu$ML models are serialised into XML files, during each development stage in BUILD, resulting in a number of platform-independent views of the system. This This supports a validation strategy by Model Inspection (or Code Inspection) to identify defects in each produced artifact. Any unexpected defect might be determined by tracing the evolution of each requirement-level concept throughout the development stages, and comparing it with what concept ***we expect to see*** at the final code.

The first column in tables 10.9 and 10.10 demonstrate each visualised concept that appears in $\mu$ML requirement models, before commencing any translating step. The rest of the columns show the evolution of each concept throughout the development stages with regard to what concept ***we expect to see*** at each level of detail till reaching the final code. The *Code Generation Framework* in BUILD produces two types of domain-specific code, namely, *JDBC/Swing Java* code and *MySQL* scripts. Table 10.9 considers the main requirements model concepts and traces its development till generating the final *JDBC Swing Java* code. On the other hand, Table 10.10 considers the development of the concepts till generating the executable database schema script in *MySQL*.

For instance, the *create* impact, in the *Impact Model* is developed to be a *Ready State*, at the end of the *Analysis* phase. This state, next, is represented as a system *Window*, at the *Design* stage. Then, each developed screen is converted into an equivalent *Swing Java class*, (Table 10.9). At the same time, the *create* impact is also developed to be a *create* flow, in the *DataFlow Model* supplied with some data on it, at the end of the Analysis stage. This flow is converted into a DBQ Procedure concept and then into a MySQL *Stored Procedure* at the *Design* and final *MySQL* script respectively (Table 10.10).

In addition to this, it is worth mentioning that for each item of data on each flow in the *DataFlow*, we *expect to see* a state *variable* equivalent to this datum. At the *Design* phase, this *variable* is converted into an *argument* in such a DBQ *Procedure* and in a local GUI control *widget* of the system screen. At the end, it becomes an *argument* in a *constructor* of a related *Swing Java class*, as well as an *argument* in a relevant *MySQL Stored Procedure*.

| **Initial Concept** | **Analysis Phase** | **Design Phase** | **Java Code** |
|---|---|---|---|
| **Business Task, Impact Task** | Boundary | Boundary | Package or *actionPer formed* Method in the Swing Java Class |
| **Subtask** | Atomic Task(DFD), Transition | Button with Action | *actionPerformed* Method in the Swing Java Class, JDBC connection and call procedure |
| **Input Participation** | Input flow (DFD), State | Window (GUI) | Swing Java Class |
| **Output Participation** | Output flow (DFD), State | Window (GUI) | Swing Java Class |
| **Impact Create** | Create flow (DFD), State | Window (GUI) | Swing Java Class |
| **Impact Read** | Read flow (DFD), State | Window (GUI) | Swing Java Class |
| **Impact Update** | Update flow (DFD), Read/Write flows (DDFD), 2 States | Window (GUI) | Swing Java Class |
| **Impact Delete** | Delete flow (DFD), State | Window (GUI) | Swing Java Class |
| **White Diamond Composition** | - - | Boundary and main menu Window (GUI) | Swing Java Class |
| **- -** | Datum on flow (DFD) | GUI Wedget in Window (GUI) | parameter of Swing Java Classs constructor |

Table 10.9: Evolution of Requirement concepts to Java

Furthermore, there is a step to ensure that each DBQ *Table* has a *Primary Key* in the *DBQ model*. This step is taken by the *DEntityToTable* translation rule (table 10.9). If there is no *identifier* for such a *DEntity*, there will be a manufactured *Primary Key* attached to the target DBQ *Table*.

*Evolution of Range constraints* is another example that shows an interesting transformation decision. It can be evaluated by detecting the generated MySQL schema file looking for an equivalent *Trigger* associated with the table that holds the *ranged* attributes. This decision is relied on by the domain-specific code generator, in this case *MySQL Code Generator* because it is known that *range constraints* might be handled differently by another relational data vendor, such as *Oracle*. In the *Oracle* version of *SQL*, this constraint is handled by the *CHECK* command as part of a column declaration. Therefore, the evaluation of this concept might be distinct based on the target environment of each code generator.

| **Initial Concept** | **Analysis Phase** | **Design Phase** | **MySQL Script** |
|---|---|---|---|
| **Business Actor** | Actor (DFD), Entity (Data) | Table (DBQ) | MySQL Table |
| **Impact Object** | Object (DFD), Entity (Data) | Table (DBQ) | MySQL Table |
| **Identifier** | Identifier (Data) | Primary Key (DBQ) | MySQL Primary Key |
| **Attribute** | Attribute (Data) | Attribute(DBQ) | MySQL Column |
| **Range constraint** | range constraint (Data) | range constraint (DBQ) | MySQL Trigger *BEFORE INSERT* |
| **Upper and Lower Bounds** | Upper and Lower Bounds (Data) | Upper and Lower Bounds (DBQ) | MySQL Trigger |
| **Association (1-to-1), (m-to-n)** | Entity (Data) | Table (DBQ) | MySQL Table |
| **Association (m-to-1)** | Dependency (Data) | Foreign Key (DBQ) | MySQL Foreign Key |
| **Generalisation, Composition** | Dependency or Entity (Data) | Foreign Key or Table (DBQ) | MySQL Foreign Key or Table |
| **Impact Create** | Create flow (DFD) | Create Procedure (DBQ) | MySQL Stored Procedure |
| **Impact Read** | Read flow (DFD) | Query Procedure (DBQ) | MySQL SP, MySQL Stored Procedure |
| **Impact Update** | Update flow (DFD), Read/Write flows (DDFD) | Update Procedure (DBQ) | MySQL SP, MySQL Stored Procedure |
| **Impact Delete** | Delete flow (DFD) | Delete Procedure (DBQ) | MySQL Stored Procedure |
| **- -** | Datum on flow (DFD) | Argument of Procedure (DBQ) | Argument MySQL Stored Procedure |

Table 10.10: Evolution of Requirement concepts to MySQL

### 10.2.3 Was the Generated Code Complete with Respect to Initial Inputs at the Requirement Sketching Phase?

This criterion examines some quality dimensions of the models employed within BUILD during the whole development. This includes the completeness, consistency and clarity of the content of each model. This helps to know whether the transformation approach takes correct mapping decisions and produces complete outputs with an acceptable level of quality. All models used within BUILD must satisfy the proposed evaluation criteria in this section. It is represented, in table 10.11, as a number of goals that must be met.

According to the criteria provided, table 10.11, it can be seen that both the *State* and *GUI Model* have not satisfied the goal: *Name of elements look professional*. This occurs because of the strategy we follow to form names of *States*, in which some names might be long. This leads to having long names for both the generated *Windows*, and *Java classes*. A proper naming convention has to be considered at this level of transformation.

| **Goal** | Task | Impact | Information | DFDs | State | GUI | Data | DBQ | Code Model |
|---|---|---|---|---|---|---|---|---|---|
| Each model is represented via a well-formed XML document. | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Every element has an appropriate namespace. | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Every element has a unique *name* and *id*. | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Inter-refering relationships appear between elements belong to the same model. | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Elements that are instances of a model belong to the associated metamodel of that it or the *core* $\mu$ML. | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| The structure of the model is logically correct. | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Each *descendant* of *Arc* has a source and a target. | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Each *typed* element has a recognised datatype. | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Each *Named* element has a meaningful name. | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Name of elements look professional. | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ |
| Appropriate Naming convention is applied to each elements (e.g. camelCase). | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| No dublicated concepts in models | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| No null elements in models | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

Table 10.11: $\mu$ML Evaluation Criteria

### 10.2.4 Could We Generate All That We Want?

In this section, a broader picture of the generated enterprise systems is considered, as well as some of its detailed component specifications. System *windows, navigation, type of widgets, business logics* and *database schema* are examples of main components that must be produced completely and correctly from the transformation approach. Any good (valid) system must have these components to be able to perform real-world business processes.

The BUILD framework is able to produce consistent screens with business tasks described at the *Requirement Phase*. This means that for each atomic task, there will be a screen (*Swing Java class*), that extends the *JFrame class* to perform this task and a screen to handle (report) its failure scenario. For instance, according to the *Student Enrolment System* case study presented previously in Chapter 9, three system screens are generated in correspondence to the subtasks of the system, as well as another three error reporting ones. Table 10.12 below describes this.

| Atomic Task | System Screen |
|---|---|
| **Input Code** | Input_code_Waiting, Input_code_Waiting_Error |
| **Input reg_Number** | Input_regNumber_Waiting, Input_regNumber_Waiting_Error |
| **Create Enrollment** | Create_Enrollment_Ready, Create_Enrollment_Ready_Error |

Table 10.12: Atomic Business Tasks and their Corresponding Windows

According to the introduced priority scoring strategy, which is based on the impact of each task on system entities and the *participation* of actors on each task, the order of executing these subtasks is determined to form the internal sequence of the main business process. As a consequence, the navigation between system windows is established, reflecting the order of subtasks execution, calculated by the transformation approach. Table 10.13 and 10.14 below demonstrate the generated sequences in both presented case studies.

| Flow Type | Engaged Task | Priority Score | Screen Order |
|---|---|---|---|
| **Input** | Input Code | 4 | 2 |
| **Input** | Input reg_Number | 4 | 1 (default rule) |
| **Create** | Create Enrollment | 2 | 3 |

Table 10.13: Order of Windows (Student Enrolment)

| Flow Type | Engaged Task | Priority Score | Screen Order |
|---|---|---|---|
| **Input** | Input code_credit | 4 | 1 |
| **Input** | Read Module Code_Credit | 3 | 2 |
| **Update** | Write Module | 2 | 3 |

Table 10.14: Order of Windows (Module Management - Modify Module)

Furthermore, the framework is also able to generate a system with a number of business tasks that might be executed optionally in an independent way. It generates an appropriate menu screen to enable the end-user to select which process is required to be performed. For instance, in the *Module Management System* case study, chapter 9, *Manage_Module_Main_Menu_Waiting* is a root menu for the system that allows the selection between a number of atomic business tasks, such as *Add Module, Modify Module* and more.

The current version of BUILD is able to generate limited navigations (links between screens). This limitation is discussed in detail in the following section (10.8). The following table (10.15) summarises the number of completely generated links between windows in both case studies presented previously in chapter 9.

| **Case Study** | **Number of Windows** | **Number of Links** |
|---|---|---|
| Module Management System | 19 | 30 |
| Student Enrolment System | 6 | 9 |

Table 10.15: Atomic Business Tasks and their Corresponding Windows

In addition to this, establishing a proper *JDBC* connection between the *presentation layer* and the *data layer* is critical. This part of the generated code is regarded ***boilerplate***, which is present in every Java class that requires preparing *MySQL* database connectivity statements. Besides this, the part of the code for catching internal errors, such as passing invalid (null) inputs to the system, is also considered ***boilerplate***, in which all screens involved in the successful business scenario must have this exception handling chunk of code to report any error to the user.

According to the structure of system windows, this version of the framework is able to generate four types of *Swing Java GUI* controls (widgets), namely, *JTextField, JButton* and *JLabel*, all placed within the *JPanel* container. The *JButton* and *JLabel* controls are common for each type of screen, in which each window has a label to display its title and a button to perform its specific action. However, the *JTextField* widget appears only when passing external user inputs to the system.

Control properties, such as *size, text* and *data type* are extracted from both the *DBQ* and *GUI Model*. From that, it is possible to generate very basic GUI controls that enable a simple business process to be completed. It is worth mentioning that additional controls might be generated from the *DBQ Model*, but it is not supported yet in this version of BUILD. See next section (10.8) for further details.

With regard to the generated relational database connected to the system, it is essential to use a normalised database in order to provide an information system that works at an acceptable level of performance (Table 10.16). By using BUILD, it can be assured that the final *MySQL* database schema satisfies the requirements of the third-normal form. This degree of normalisation is achieved according to the translation chain started from the *Information Model*, which contains unnormalised relationships (e.g. *generalisations, compositions*, and *many-to-many associations*), and ending up with the normalised *Database and Query Model*, which contains resolved relationships.

| Relationship | Engaged Entities | Rule Applied | Final Table(s) |
|---|---|---|---|
| Association (m-to-1) | Student, Address | Generate FKs | Student, Address |
| Association (m-to-n) | Student, Module | Generate Linker table and FKs | Student, Module, Enrollment |
| Generalisation (disjoint) | Assessment, Assessment_Exam | Merge | Assessment_Exam |
| Generalisation (overlapping) | Person, Student, Staff | Generate FKs | Person, Student, Staff |
| Composition (total) | Module, Assessment | Generate FKs | Module, Assessment_Exam |

Table 10.16: Data Model Normalisation

### 10.2.5 Are Transformations Adequate to Fill the Gap in Implementation?

On some occasions, during the chain of model transformations, new concepts are inferred due to applying particular mapping decisions. For instance, a unique *Identity* in the *Data Dependency Model* or an automatic *Primary Key* in the *Database and Query Model* is manufactured according to the translation agent decision when such an entity in the source model has no information about its *Identity*. Furthermore, the size of some attributes, that have no *size* detail given in the source model, is also manufactured based on the data type.

In addition to this, generating a menu screen that holds alternative functions of the system is another example of the design decision, taken by the transformation/code generation rules, to fill the implementation gap. *The Module Management* case study is an example of this. Moreover, the *Code Generation* approach is able to fill the implementation gap by filling the generated classes by ***boilerplate*** code for arranging the database connectivity and embedding commands that are used for executing and calling *SQL* stored procedures.

### 10.2.6 Can We Execute the Generated Code?

This criterion aims at examining the generated code/database from BUILD after running them under the related environments. This helps to answer some code evaluation questions to know whether the generated code has compiled, executed and done its desired job successfully without mistakes. As the current *Code Generation Framework* of BUILD produces only *Java* code and *MySQL* script, a decision was made to compile the generated *Java* files using the *Eclipse Framework*, and execute the final *MySQL* dump file using *MySQL Server 6.1*.

The following Table (10.17) demonstrates the criteria for evaluating the generated system and its related database. It can be seen that the generated *Java* code has not satisfied the goal: *The generated system is able to receive multiple selected record from the database*. This occurs because the code generators in BUILD are currently designed as a proof of concept, in a way that supports only passing back a single record, or part of a record. This issue might be overcome by considering retrieving multiple records in a later version of the code generation framework.

Indeed, this goal is related to another one that the generated *MySQL* schema has not satisfied, which is: *Query stored procedures are able to pass multiple selected records to the system.* This happens because the stored procedures are designed, as a simple proof of concept, in a way that they bind a single *SQL* statement that is stored in a single-value *out* parameter to be passed back to the code. This issue might be also overcome by considering retrieving multiple records in a later version of the code generation framework.

| **Goal** | **Java** | **MySQL** |
|---|---|---|
| The generated system (*Java* classes) is compiled. | ✓ | - |
| The generated system (*Java* classes) is executed. | ✓ | - |
| Each window is displayed correctly. | ✓ | - |
| Each window consists of correct GUI controls. | ✓ | - |
| Each window contains consistent GUI controls. | ✓ | - |
| The generated system passes local variables, between screens, correctly. | ✓ | |
| Windows are connected correctly to the database with all access permissions required. | ✓ | - |
| Each button fires an action according to *click* even. | ✓ | - |
| The generated system/database represents real-world business. | ✓ | ✓ |
| The generated system invokes a correct stored procedure. | ✓ | |
| The generated system passes correct arguments (*input parameters*) to the invoked procedure. | ✓ | |
| The generated system is able to receive multiple selected records from the database. | ✗ | - |
| The generated system is able to receive a single selected record from the database. | ✓ | - |
| *null* exception is handled correctly. | ✓ | |
| Failure of the database conncetion is handled correctly. | ✓ | ✓ |
| The generated database schema (**.sql file*) is executed without errors. | - | ✓ |
| The generated database schema (**.sql file*) consists of complete database tables structure (specified attributes). | - | ✓ |
| The generated schema (**.sql file*) consists of correct foreign keys in tables. | - | ✓ |
| Every table in generated schema (**.sql file*) has a primary key (original or manufactured). | - | ✓ |
| The generated database schema (**.sql file*) consists of all required triggers (BEFORE INSERT) to enforce check constraints. | - | ✓ |
| The generated database schema (**.sql file*) consists of all required stored procedures to represent the business logics. | - | ✓ |
| Every generated stored procedure has one SQL statement to perform a single database operation | - | ✓ |
| Each *update* stored procedure updates a database record successfully. | - | ✓ |
| Each *create* stored procedure inserts a new database record successfully. | - | ✓ |

| | | |
|---|---|---|
| The *delete* stored procedure removes an existing database record successfully. | - | ✓ |
| The *query* stored procedure selects and projects a database record successfully. | - | ✓ |
| *Query* stored procedures are able to retrieve multiple selected records. | - | ✓ |
| *Query* stored procedures are able to pass multiple selected records to the system. | - | ✗ |
| There is a *Trigger BEFORE INSERT* for each table that has a field with a range constraint. | - | ✓ |

Table 10.17: Criteria for Evaluating the Generated Information System

### 10.2.7 Things Were Wrong or Missing and Suggested Repairs

Expanding the generated type of widgets, to cover more GUI controls, is a main issue that has to be considered in later versions of BUILD, as the type of widget is based on the type of screen, whether it is for inputting or displaying information. The current translation rules provide two types of GUI control to handle this, namely, *JLabel* and *JTextField*. It is possible to improve the current version of BUILD by filling the implementation gap with more accurate widgets than the current one. For example, a *JComboBox* control might be used to restrict the input data to a particular field. This can be done by allowing only one (or many) selected value(s) to be valid inputs, from *selection constraints* on some attributes in the *Data* or *DBQ Model*.

In addition to this, adding the ability to display a collection of results (items) in such a *JTable* control is essential. The current version of the BUILD Code Generator is able to generate executable code for retrieving and displaying a single result only. In the case that the query procedure returns a collection of records, the code will present the first record only to be the retrieved result. In order to solve this issue, we must expand our $\mu$ML notation at the requirement level to be able to express a collection of objects.

Furthermore, in the case of generating a system with multiple *input* steps, the current version of BUILD produces a window for each step. These windows form a chain to perform the whole input process. Adding an extra design choice to allow merging these *windows* into a compound one, which holds a longer *entry form* to receive all user inputs at once, offers an optimal system with a fewer number of screens. This can be achieved by enhancing the (*in-place*) transformation agent that optimises the generated *State Model* from the detailed *DataFlow* one.

Moreover, there is an issue (limitation) in the developed algorithm for translating the detailed DFD into *State Model*, in which the current version is allowed to translate a short sequence of atomic tasks that must start (be activated) by a user input only. We are aware that some business tasks might be fired automatically, before any user interaction, when the system starts (pre-processing).

The pattern for extracting this from the detailed Data Flow Model is not yet supported. In order to overcome this dilemma, the algorithm must be improved to be capable to cover all missing patterns.

Besides this, there are a variety of business patterns (control flows) that might be covered in later versions of the proposed $\mu$ML and related BUILD translator agents. To solve this problem without expanding the graphical constructs of the modeling language (keeping $\mu$ML lightweight), we suggest adding the multiplicity annotation to the *Task Model*, particularly to the *Composite* relationship and improve the extraction (prediction) of its possible interpretations. Then, the relevant translator agent generates a possible control-flow construct, similar to the way we follow for dealing with the optionality of subtasks (see the related case study in chapter 9).

## 10.3 Analysis of Time Complexity

Analysing the *time efficiency* of transformation algorithms is considered to measure the quality and scalability of the proposed framework, including its architecture and implementation. This section aims at evaluating the performance of the model transformation approach within BUILD. The Big-O notation technique is adopted, here, for articulating how long an algorithm takes to run, and to examine whether or not there exist some factors or limits on inputs that affect the growth rate of time in some transformation steps. This section discusses the *time efficiency* of the framework from three perspectives: *rule complexity, transformation step complexity* and *overall complexity*. The section also describes how the complexity in such a translation step may be reduced by caching the results of repeated transformations in the *Context*.

### 10.3.1 Big-O Analysis of Rules

As previously presented in the thesis, the current version of BUILD consists of two types of transformation rule, namely, *one-to-one* translation and *two-to-one* merging (folding) rules. The following subsections discuss the complexity of these types. The time complexity varies from one rule to another even within a transformation step. The worst case scenario is considered for each rule, and later for each step (Section 10.3.2). This decision assures that any possible alternative scenario has less complexity than the worst case one.

For all transformation rule classes in the framework, the main computation of each rule is performed by the *doTranslate* method that belongs to that rule. This method runs once for each distinct execution of the rule on distinct arguments. Each rule is constructed once inside the dominant rule that calls it. The graph of all applicable rules is constructed prior to rule execution, so does not contribute to the time complexity of execution. Construction time is negligible, the linear in the number of rules constructed.

The body of the *doTranslate* method in all rules contains one or more conditions for checking any precondition of that rule, implemented using *if* or *if..else* statements. This chunk of code (construct) is executed once at each run of the *doTranslate* method. From that, the time complexity of this construct is $O(1)$. On some occasions, calling other methods, e.g. *translate*, is required, the time complexity of this call or any further method invocation is also $O(1)$.

In non-terminal rules, when translating further properties of a target element, or invoking some subrules for translating its descendants, a kind of *iteration* is required to complete the whole translation step. The time complexity of this part of the code is $O(N)$, where $N$ is a number of properties that are ready for translation. Even if the loop has some *if* or *if..else* statements, object creation and method invocations, the worst case is still $O(N)$ based on the number of properties/descendants of the target element.

| **Complexity** | **Number of** *one-to-one Rules* | **Number of** *two-to-one Rules* |
|---|---|---|
| $\mathbf{O}(1)$ | 31 | 2 |
| $\mathbf{O}(N)$ | 17 | 1 |
| $\mathbf{O}(N * M)$ | 0 | 2 |
| $\mathbf{O}(N * M^2)$ | 1 | 0 |
| $\mathbf{O}(N * M * K)$ | 3 | 5 |
| $\mathbf{O}(N * M * K * P * Q)$ | 1 | 0 |
| $\mathbf{O}(N^2)$ | 0 | 3 |

Table 10.18: Different Complexity Types in the Transformation Approach

When nested *iterations* occur within *doTranslate* method, the computation time is increased to be $O(N * M)$, where $N$ represents the number of target elements and $M$ represents the number of a particular type of properties/descendants for each target element. For each type of rule in the framework, Table 1 demonstrates all possible worst case complexities and the number of rules that represents each one.

#### 10.3.1.1 One-to-One Rule Complexity Analysis

The *one-to-one* rule complexity analysis is applicable for analysing *time efficiency* for all concrete rules that extend *Translation* class in the top-level framework. This includes *one-to-one translation, transformation* and *in-place modification*. It is worth emphasising that the time complexity of most terminal rules is $O(1)$, as the body of their *doTranslate* methods consists of two main parts: checking preconditions ($O(1)$) and performing the mapping ($O(1)$).

However, there are a few terminal rules that have a loop to handle a collection of properties within that same rule. The time complexity of these rules is $O(N)$, where $N$ is number of properties/descendants of the source element. According to the implementation, one or more *foreach* constructs are used in some rules to perform this job. The majority of rules, in the framework, have **at most** one loop. As seen in Table 1, the time complexity of those rules can be either $O(1)$ or $O(N)$.

Apart from this, the usual complexity of a rule reduces to $O(1)$ if the same rule is invoked again on the same argument(s). This will happen if multiple higher-level rules depend on the same lower-level rule, which is executed multiple times. The time saving is achieved by caching the result of every rule execution in a *Context*, so that it need not be recomputed. However, the overall reduction in complexity depends on the degree of overlap and depth of nesting in how the rules are designed, so it cannot be systematically quantified in general.

#### 10.3.1.2 Two-to-One Rule Complexity Analysis

The *two-to-one* rule complexity analysis is applicable for analysing *time efficiency* for all concrete rules that extend the *MergeTranslation* class in the top-level framework. The majority of rules have some *iterations*. These loops are mostly used to traverse elements in the second source model for checking or collecting further information required for generating a target element.

The complexity of these rules depends on how deeply the iteration is nested. It is found in the designed rules that the number of nested loops are either three or two. The time complexity of these constructs are either $O(N * M)$, $O(N * M * K)$ or $O(N^2)$. It is worth mentioning that a few rules have **at most** one loop. The time complexity of these rules can be either $O(1)$ or $O(N)$ (see Table 10.18.).

Figure 1(a) summarises the remarkable changes in growth rates of time complexity in four possible worst case scenarios: $O(N), O(N*M), O(N*M*K), O(N*M^2), O(N*M*K*P*Q)$ and $O(N^2)$. The *x axis* represents the number of $N$, while the *y axis* represents the computation time. It is assumed that all calculations are based on a single run. It can be seen how the scalability becomes poorer in the cases that have a larger number of nested and linear *iterations*.

### 10.3.2 Big-O Analysis of Translation Step

As previously presented in the thesis, the current version of BUILD consists of a number of model transformation steps, implemented independently in a number of *Java Packages*. These packages contain several rules (*Java classes*) to perform such a complete mapping step. This part of *time efficiency* analysis focuses on the complexity of the transformation step itself, by considering the Big-O notation of the worst case scenario of all included rules in each step. For each transformation, the total number of rules that have similar complexity are grouped together and are represented in Table 10.19.

| **Transformation** | **O(1)** | $\mathbf{O}(N)$ | $\mathbf{O}(N*M)$ | $\mathbf{O}(N*M^2)$ | $\mathbf{O}(N*M*K)$ | $\mathbf{O}(N*M*K*P*Q)$ | $\mathbf{O}(N^2)$ |
|---|---|---|---|---|---|---|---|
| **Information to Data** | 7 | 5 | 0 | 0 | 0 | 0 | 0 |
| **Data Model to DBQ** | 2 | 6 | 0 | 1 | 0 | 0 | 0 |
| **Task & Impact to DFD** | 12 | 0 | 0 | 0 | 0 | 0 | 2 |
| **DFD to detailed DFD** | 7 | 0 | 0 | 0 | 0 | 1 | 0 |
| **DFD to DBQ** | 0 | 0 | 0 | 0 | 4 | 0 | 0 |
| **DFD to State** | 1 | 2 | 2 | 0 | 0 | 0 | 1 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Task to State** | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| **State to GUI** | 2 | 3 | 0 | 0 | 1 | 0 | 0 |
| **Impact to Information** | 2 | 1 | 0 | 1 | 2 | 0 | 0 |

Table 10.19: Complexity of Each Transformation Step

Table 10.19 summarises the total number of each type of rule involved in each translation step. In the step of translating the detailed *DataFlow Model* into the *DBQ Model*, for instance, there are four rules that have $O(N*M*K)$ time complexity. For each *flow* in the DFD, $N$ is the number of flows, $M$ is the number of data variables on that flow and $K$ is the number of attributes in the target end-role of that flow. The complexity of each translation step is calculated as the worst-case complexity of any rule group involved in that step. From that, the time complexity of *Information to Data* is $O(N)$, *Data to DBQ* is $O(N*M^2)$, *Task & Impact to DBQ* is $O(N^2)$, *DFD to DDFD* is $O(N*M*K*P*Q)$, *DFD to DBQ* is $O(N*M*K)$, *DFD to State* is $O(N^2)$, *Task to State* is $O(N)$, *State to GUI* and *Impact to Information* is $O(N*M*K)$ time.

### 10.3.3 The Overall Time Complexity Analysis of BUILD

This sections aims at calculating the exact time performance of BUILD, rather than the big-O estimate. As the BUILD transformation strategy is designed in a linear way, in which each step runs only once. Transformation rules within each step are designed in a way that a rule calls another rule one or more than one time (e.g. to translate an *entity* and its corresponding *attributes*). The time complexity for each transformation step is calculated in the previous section (Section 10.3.2) and summerised in Table 10.19. The highest time complexity of each step is considered the worst case (Big-O notation) of that step.

In order to calculate the overall time complexity, we need to multiply the time complexity by the number of transformation rules that has this complexity. For instance, according to Table 10.19, the time complexity of the *DFD to DBQ* step is $O(N*M*K)$, in which there are four rules that have this complexity. From that, the exact time performance of this transformation step is $N*M*K*4$. The overall complexities of the rest of steps are calculated likewise.

## 10.4 Evaluation Experiment of $\mu$ML Notation

In order to evaluate the suitability of $\mu$ML, an experiment has been conducted with selected university students from different faculties and departments such as, Electrical and Communication Engineering, Information Management, Chemistry and Medicine. The main objective of this experiment is to evaluate $\mu$ML in terms of its simplicity and ease of adoption by end-users as a modelling language for expressing basic functional requirements and specifications of a real-world and commonly used information systems. It is worth noting that none of the participants were Computer Scientists, who might have been exposed to UML notations. This was important, in order to get the responses of non-specialist business users.

Students are interviewed and the general overall idea about the task and the modelling language is explained to them before commencing the task. After that, an incomplete solved business case study is given to them, showing what exactly is expected from them. Then, students were given a task to draw several μML diagrams of given business scenarios, taken from the same case study. The time taken to draw each diagram was measured manually by the students.

### 10.4.1 Design of the Experiment

The experiment involves an *Online Hospital Booking System (OHBS)* case study, exemplifying how a registered patient books an appointment to see a registered doctor in the hospital. A *Task, Impact* and *Information Model* to this part of the system is given to students showing how the notation of each model looks like. Then, students will be asked three questions to draw *Task, Impact and Information Models* for some business activities within OHBS and record the time they needed for each question (see Appendix D). In order to analyze the results of the experiment, criteria are designed, including the following points:

- The average time for drawing diagrams is less than 5 mins, between 5 mins and 8 mins or greater than 8 mins.
- Business entities is captured with correct names and notation.
- Business activities (tasks) are captured with correct names and notation.
- Business structure of task are drawn with correct notation.
- HCIs between stakeholders and business activities are captured with correct notation.
- Impacts between entities and business activities are drawn with correct notation and direction.

### 10.4.2 Analysis of the Results

As mentioned before, three questions are provided to draw three types of μML requirement models, namely, *Task, Impact* and *Information* Model. The following subsections analyse the results for each task provided.

#### 10.4.2.1 Results Analysis for the Task Model

The expected answer has 9 elements: 6 *tasks*, 2 *compositions* and 1 *actor*. Figure 10.1 below demonstrates a sample of the correct solution, sketched by a student. The assumed time for completing this question is three and a half minutes.

It was observed, from student answers, that all students were able to identify the stakeholder of the system. They also used the correct notation (ellipse) for expressing business activities. 60% of the students selected the correct (*composition*) type for representing the right *whole/part* relationship between business tasks, whereas 20% had not completed the model and 20% failed in using the accurate notation for representing a *sequence of subtasks*, which is one (*whole*) task and some (*parts*) subtasks that are connected to the *whole* one by a *Composition* relationship, but instead they used a chain of *Composition* relationships from the *whole* task to the last *part* subtask. This gives a specific order of subtasks' execution that is not a part of the givn scenario. The average duration time of all students to complete this task was 4.6 minutes.

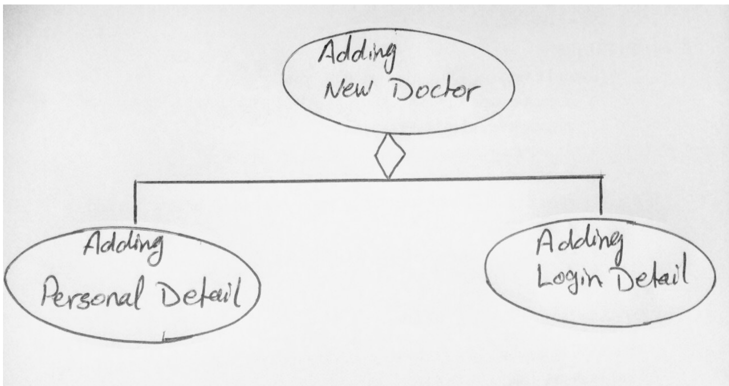



Figure 10.1: Student attempt at drawing a Task Model

Table 10.20 illustrates (summarised) all answers provided by the participants for the first question.

| **Item** | **Std 1** | **Std 2** | **Std 3** | **Std 4** | **Std 5** |
|---|---|---|---|---|---|
| **Number of Element** | 9 | 9 | 6 | 10 | 9 |
| **Drawing Time (mins)** | 5 | 4 | 5 | 4 | 5 |

Table 10.20: Summary of All Student Answers (Question 1)

#### 10.4.2.2 Results Analysis for the Information Model

The expected answer has 3 elements only: 2 *objects (entities)* and 1 *association.* Figure 10.2 illustrates an example of a complete solution, drawn by a student. The assumed time for the completion of this question is just one minute.

It was observed, from the provided answers, that all students were able to identify the correct business objects (entities) of the system. 100% of the students succeeded in drawing the correct notation (rectangle) for expressing business entities. Furthermore, all of them were able to extract the relationship between these entities. It is worth mentioning that determining multiplicities is out of the scope of this task.

However, 20% of the students sketched duplicate entities (e.g. New Doctor and Existing Doctor), and added extra entities out of the scope of the given case study. The average duration time of all students to complete this task was 3.8 minutes.

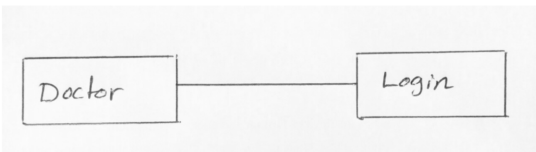



Figure 10.2: Student attempt at drawing an Information Model

Table 10.21 demonstrates all answers provided by the participants for the first question.

| Item | Std 1 | Std 2 | Std 3 | Std 4 | Std 5 |
|---|---|---|---|---|---|
| **Number of Element** | 3 | 3 | 3 | 7 | 3 |
| **Drawing Time (mins)** | 5 | 2 | 5 | 4 | 3 |

Table 10.21: Summary of All Student Answers (Question 2)

#### 10.4.2.3 Results Analysis for the Impact Model

The assumed answer has 12 elements: 4 *tasks* and 4 *objects* and 4 *impacts*. Figure 10.3 below exemplifies a partial solution, answered by a student. The expected time for completing this question is 6 minutes.

One of the main observations was that 80% of the students were able to distinguish the notation used for capturing *tasks* from the one that is used for capturing *entities* and draw it correctly without any confusion. Only one sample (20% of the whole number of participants) used a wrong notation (ellipse) for drawing *entities* and (rectangle) for drawing *tasks* and did not complete the model.

In addition to this, 60% of the participants used correct *impacts* for representing internal interactions between tasks and entities, but 40% of them have at most either one wrong *impact* direction or wrong *impact* type.

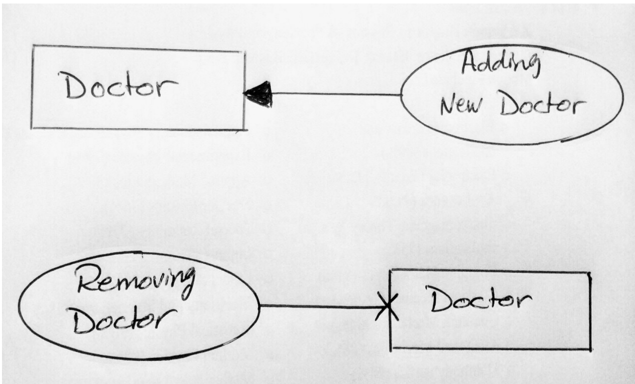



Figure 10.3: Student attempt at drawing an Impact Model

Table 10.22 summarised all answers provided by the participants for the first question.

| Item | Std 1 | Std 2 | Std 3 | Std 4 | Std 5 |
|---|---|---|---|---|---|
| **Number of Element** | 11 | 12 | 6 | 12 | 12 |
| **Drawing Time (mins)** | 5 | 6 | 5 | 7 | 7 |

Table 10.22: Summary of All Student Answers (Question 3)

#### 10.4.2.4 Overall Conclusion

After analysing the experiment results, it can be noticed that about 67% of the drawn models are complete and answers are correct. Non-specialist students were able to differentiate one concept from another. Reasons for failure could include that they did not learn the notation well enough; or they failed to understand the business scenario. Better-trained users might improve this statistic; and domain-experts might understand how to express their business more effectively. This is because they are domain experts who clearly understand all business activities. From that, our strategy in constructing small models that have clean semantics and simple notation is worthwhile to be adopted by non professional developers.

## 10.5 Outlook on the Chapter

The overall work presented in this chapter can be divided into three main evaluation dimensions, namely, evaluating the completeness and correctness of the rules, time efficiency analysis and suitability of $\mu$ML concepts and notations. In the first dimension, criteria are introduced for evaluating whether or not the generated code is complete with respect to initial specifications expressed at the *Requirements Sketching* phase. Furthermore, the ability to execute the generated system (code) and its backend database without errors is also evaluated. It shows how the transformations approach fills the gap in implementation by taking critical design decisions and introducing appropriate chunks of *boilerplate* code.

In addition to this, the Big-O notation technique is utilised for evaluating the *time efficiency* of the model transformation algorithms in BUILD. For each type of transformation, the worst case complexity is calculated and explained within the chapter. At the end an experiment is conducted for evaluating the suitability of $\mu$ML in terms of its simplicity and ease of adoption by end-users as a modelling language. The majority of results shows positive feedback from participants.

# 11
# Conclusion and Future Work

"If you don't work on important problems, it's not that you'll do important work."

Richard Hamming

## 11.1 Context

This chapter summarises the overall contributions conducted by the research into the Model-Driven Engineering for enterprise information system development with respect to the following aspects:

(1) Business users engagement contributions related to how the thesis expands the end-users role in the development lifecycle by enabling them to act as designers and lead the development processes, using less technical knowledge to construct their desired system.

(2) Modelling language contributions related to achieving a lightweight language that is able to capture end-users logical thinking about their information system using less technical knowledge, with a higher level of abstraction than other existing UML and other DSL approaches.

(3) Development approach contributions relate to the proposed MDE method that adopts a forward model transformation strategy and employs the proposed lightweight modelling language, to produce an executable information system derived from its initial requirements.

## 11.2 In Support of the Thesis

By observing the current state of art in Model-Driven approaches for web applications and information systems development in Chapter 2, we noticed, on the one hand, that most of the approaches target developers who have the ability to construct rich detailed models with technical specifications, such as UML-based approaches. On the other hand, in DSL-based approaches, such as WebML[124], that aim at raising the abstraction level further, we noticed that developers must fill those models with a lot of detail in order to generate a complete code at the end. For instance, the components of each screen must be specified, as well as the navigation links between pages.

From the above, it can be concluded that both approaches rely on modelling technical concepts at a particular level of abstraction. Based on this given knowledge, the transformation rules directly map the concepts, which appear in source models, to those equivalent concepts in the target. This section highlights the contributions of this thesis that improve the development process of information systems from the perspective of business end-users.

With respect to the current analysis of the state of art, Chapter 3 introduced a novel Model-Driven Engineering approach (BUILD) that includes a more intelligent forward model transformation strategy than current approaches. It allows rules of transformation to predict and provide new knowledge at each translation step carried on within the proposed method. Unlike the existing approaches, mentioned in the survey, the BUILD method starts with simple and semantically cleaned models ($\mu$ML); the transformation mechanism then evolves these models and introduces new and more detailed concepts.

Besides this, the literature review revealed how little attention is paid to having an appropriate modelling language that supports End-User MDE for information systems development. To address this, a lightweight modelling language ($\mu$ML) was introduced to enable the end-user to specify desired systems using their conceptual and business knowledge about the domain. The proposed language achieved an acceptable degree of clarity from the perspective of end-users.

The same notation was employed in different $\mu$ML models, giving different meanings in respect to the context of that model. Chapter 4, 5 and 6 discuss in depth the notation and semantics of each model with regard to the development stage, in BUILD, which it belongs to. A formal flavour, using First Order Predicate Logic FOPL with some extentions, mentioned previously in Chapter 3, has been added to the definition of each concept.

Chapter 7, reflected the three-phase linear composition of model translations, within BUILD, that shifts the requirement models from the *Requirements Sketching* phase to the *Analysis* phase, and from the intermediate analysis models, used in the *Analysis* phase to the *Design* phase, and lastly from design models, used in the *Design* phase, to the code generation stage. This illustrates the advantages of the intermediate layers, in which new knowledge, supplied by business users or by the transformation rules, are allowed at each layer. As a result, some of the final outcomes are more heavily influenced by useful, generated intermediate concepts, than by the initial user input, illustrating the usefulness of a multilayered approach.

In addition, Chapter 7 also discussed the efficiency of the proposed overall architecture of the transformation approach in BUILD. It demonstrates how the structure successfully orchestrates the hybrid transformation rules that are implemented using Java, and which are styled as a number of Java translation agents (class). In the main rule for each translation step, As a result, control over the execution order is achieved, by natural interpretation of the rule dependency structure. Rules are also idempotent, in that a duplicate application maps to the same result.

Chapter 9 discussed, in depth, how the evolution of models is achieved. A full list of the detailed transformation rules was described for each translation step in the framework, showing how each rule (translation agent) produces a target concept from the source, infers new knowledge that is inserted in the translated elements, and makes a critical design decision to the structure or the behaviour of the system. It is worth saying that the designed rules are formalised using FOPL in terms ($Forall$ $(\forall), Exists$ $(\exists)$) to ensure that all rules are deterministic with a known type of target.

At the end a number of translation, transformation, and in-place modification steps are assembled together, forming a complete chain of model transformations that start from the requirements and end with the final code.

## 11.3 Discussion of Research Questions

***Question 1*** *What is the required interaction mechanism between the end-user and the transformation approach to capture their desired system specifications?*

The interaction is achieved via the manual creation (sketching) of the requirements models using the proposed $\mu$ML notation that is able to capture functional requirements, using end-users logical thinking and less technical knowledge, via three system views (Chapter 3 and 4).

***Question 2*** *What kind of abstract system views do we need to capture in order to have comprehensive knowledge and behaviour of a system?*

Domain experts have conceptual knowledge about their business from three perspectives. The structure of business processes and how system actors might interact with each process, the structure of business entities and how these entities might relate to each other, and finally, it is required to express how to supply/retrieve external data into/from the system. In other words, the interaction between system actor(s) and each process must be considered. Furthermore, the types of interactions (effects) between business processes and entities must be modelled (Chapter 3 and 4).

***Question 3*** *What transformation rules are required to fold and optimise high-level views and introduce extra detailed design information into a system?*

Merging the rule of concepts that appear in the *Task Model* with the equivalent ones that appear in the *Impact Model* is a significant step to producing the core DataFlow Model. It is designed in BUILD via the *Task-Impact-to-DataFlow* agent. Furthermore, inferring an initial *Information Model* from a pre-defined *Impact Model* is a mapping step that predicts new details regarding the relationships between business entities in the *Information Model* (Chapter 7 and 8).

***Question 4*** *What transformation rules are required to refine high-level models and introduce richer design information into a system?*

At the *Analysis* phase, the step of deriving the detailed *DataFlow Model* from the initial one is one refinement step. Task decomposition approaches are applied to tasks appearing in the initial DFD to produce atomic tasks that perform a single CRUD operation. Furthermore, the in-place model modification step on the generated *State Model* to generate appropriate *error* states for handling possible failure cases of business processes is another example of this kind of transformation.

In addition to this, the translation step from *Information Model* to the normalised *Data Dependency Model* is a step that performs data normalisation on the source model. It might add a manufactured *identity* for each entity that has no user-defined *identifier*. Not only this, This step also derives the detailed *Database and Query Model* by merging concepts from both the *Data and DataFlow Model*, which consists of a complete database schema that holds business logic as well as table definitions.

Besides this, a direct one-to-one mapping between the *Information Model* and the intermediate *Data Dependency Model* is another translation step that analyses the types of relationships (e.g. *generalisations* and *compositions*), as well as multiplicities on *associations* to produce the *dependency* relationships between *Data Model* entities (Chapter 7 and 8).

***Question 5*** *At which level of development is end-user engagement required to supply new knowledge to be considered by model translators at the next translation step?*

Users manual engagement is performed by annotating the generated (initial) *DataFlow Model* with data on each flow. This occurs during the *Analysis* phase as a critical step in order to generate the detailed *DataFlow Model* (Chapter 3 and 4).

***Question 6*** *To what extent are end-users able to generate a complete system for their demand?*

As a proof-of-concept, the current version of BUILD is able to generate executable information systems that are *JDBC Java Swing* applications running in an Eclipse environment, that connect via *JDBC* to a *MySQL* backend database (Chapter 9 and 10).

***Question 7*** *With respect to code generation, to what extent are we able to construct and link the IS layers from generic and less technical specifications?*

As the current version of BUILD successfully produces two-tier information systems, its database connection is prepared and established as a *boilarplate* code generated by the relevant code generator within the framework. This strategy links the generated classes (screens) at the *presentation layer* to the back-end database. As a result, a complete connection is achieved.

## 11.4 Summary of Findings

### 11.4.1 Semi-Automated Way For Producing a Rich Detail Data Dependency Model

According to the BUILD method, the *Data Dependency Model* is only derived from the predesigned *Information Model*. The *Information Model* can be constructed manually by business users who sketch all model contents from scratch, including entities, attributes and relationships. The other way is by a particular translation agent that derives automatically an initial *Information Model* from scratch, using the knowledge provided previously in the *Impact Model* to predict entities and resolved relationships only.

An interesting exploration is concluded from the similarity between both generated *Data Dependency Models* resulting from experiment 3a and 3b, Chapter 9. The default transformation rules within the *Impact-to-Information Model* translator successfully produced a number of *many-to-one* associations that lead to derive identical dependencies between entities in the generated *Data Model*, from the user-defined *Information Model*. Under few circumstances, the translator agent cannot do more due to the lack of knowledge in expressing whether the predicted generalisation is *overlapping* or *disjoint*. Therefore, a default rule that ignores this issue treats any possible *generalisation* as an *overlapping* one.

This finding raises a clear idea for introducing an alternative approach for constructing a rich detailed data model that combines benefits from both development ways mentioned above. It aims to achieve new generated *Data Dependency Models* from an *Information Model* that contains both the manually specified entities and their attributes, and automatically inferred and resolved relationships between entities, from an *Impact Model.*

This approach is beneficial because it narrows and defines the manual engagement carried out by end-users for constructing *Information Models.* This significant idea reduces possible manual errors, performed by business users, in designing relationships between entities. This current work can be continued to provide extra support to information system development approaches, led by end-users. This is described later in the future work section 11.5.

### 11.4.2 Alternative Strategy For Modelling Rich Information Models at the Requirement Phase

After the successful generation of *MySQL set constraints* and the *PRIMARY KEY* from requirement level concepts, we discovered that handling constraints from that level of abstraction is possible in a straightforward way. It does not require vast end-users technical knowledge, and substantial computation by the related translators. Unlike the manufactured and generated constraints, such as *FOREIGN KEY, UNIQUE,* and *NOT NULL* that requires checking other intermediate elements in the source and/or target models, during the creation of the *Database and Query Model*, these trivial constraints are directly passed from the source to the target till the development approach generates an equivalent script at the code generation stage.

Given the above, if emphasis is laid more on supplying trivial attribute constraints in an initial Information Model, this will lead to producing trivial attribute constraints at this level, which enriches the *Information Model*, alongside predicting associations automatically, with less end-user engagement, leads to producing a more sophisticated data model. The detailed description of entities and attributes, including structure and constraints may be extracted from the *Information Model*, whereas the knowledge about relationships between business entities may be predicted from the *Impact Model.* This strategy avoids the repetition of concepts or information in both the *Impact* and *Information Model.*

## 11.5 Future Work

It can be said that the introduced Micro-Modelling Language ($\mu$ML) and the proposed Model-Driven Engineering approach (BUILD) improve the information systems development process from the perspective of business end-users. But there are a number of ways in which the current work can be extended.

### 11.5.1 Designing Additional Types of Generic Transformation Rules At the Top-Level Architecture

The current version of BUILD has two types of model transformation rules at its top-level framework. Other model transformations such as, *in-place model modification* and *one-to-two forward translation* must be added to the family of generic translation rules designed in BUILD. Currently, the *in-place modification* is treated as a *one-to-one* model transformation, in which the source and target have the same type, whereas the *one-to-two translation* is implemented via two one-to-one transformation rules.

### 11.5.2 Developing a Merging Translation Agent for Data Dependency Model Construction

This piece of future work is related to the findings discussed in section 11.4; it aims at enhancing the approach for developing the *Data Model* by enabling the BUILD method to avoid possible accidental end-user mistakes during designing relationships between entities or during annotating (assigning) their multiplicities. It is assumed that defining the structure of entities is considered a straightforward task for domain experts. In contrast, constructing accurate types of relationships, with correct multiplicities might be error-prone from the perspective of naïve business users, which requires a further treatment.

This issue can be tackled by minimising the usage, and possibly notation, of the *Information Model* to cover only business entities and their structure (attributes and some constraints on them), and letting the relationships between these entities be predicted from the *Impact Model*. A slightly alternative approach might be adopted here, by including all partial relationships, about which the designer (end-user)has total confidence, in the initial information model.

In both suggested solutions, the knowledge in an (initial) *Information Model* is accumulated together with the predicted one from the Impact Model in an independent merging step. From that, designing the *Information-Impact-To-Data Model* translation agent is a sensible solution to be in charge of carrying out this translation step.

### 11.5.3 Enrich the Information Model by Supporting More Types of Constraints

In data modelling, various types of constraints are applied to fields in order to enforce particular input, restrict values, prevent empty values, and more. Some of these constraints are naturally known by domain experts without the demand to have additional technical data modelling skills. From their prespective, it is assumed that *not null, unique, set constraint, range constraint, and default value* are examples of trivial constraints that might be determined by business users at the requirement level.

*Set constraint*, for instance, is a type of constraint that applies to values of attributes within an *entity/table*. It occurs when acceptable values of a particular field are restricted by a limited known set of values. This constraint is expressed in *MySQL* using an *Enumerator*, appearing as an enum field within the generated table, which rejects any attempt to input any invalid data. As we successfully generated an equivalent *MySQL* code for our early work in ReMoDeL[101] DBQ model and Database Generation Framework, introduced in[108] and[109].

As we have not paid adequate attention to these concepts during the design of the *Information Model*, adding extra concepts that are able to express these constraints on attributes is, therefore, worth considering. In real-world business, this information is regarded as trivial from the perspective of domain experts (end-users). The evolution strategy of these concepts must be determined to enable passing these specifications throughout development stages of BUILD.

### 11.5.4 Covering More Business Workflow Patterns

The current version of BUILD is able to extract knowledge from various $\mu$ML models to construct two types of business workflow patterns, namely, *Sequence and Exclusive Choice*. These supported patterns were demonstrated previously throughout Chapter 9 case studies. Covering additional workflow patterns is an interesting future research question raised by this thesis. It includes expanding the concepts and notation of some $\mu$ML models, and designing extra translation agents to deal with these new concepts.

One possible suggestion is expanding the *Task Model* concepts to allow the expression of multiplicities in *Composition* relationships between tasks. This indicates the number of executions for each subtask involved in a *Composition*. As a result of this, an iteration control flow pattern might be detected from this concept. Besides this, adding *conditions (filters)* on the branches of a *Composition* allows for the possible detection of sophisticated *conditional branching* of business tasks. For both suggestions, an appropriate translation chain has to be defined in order to get the benefits of these patterns later in the design and code generation phases.

### 11.5.5 Optimising the DataFlow-to-State-Model Translation Algorithm

As mentioned in the limitation of the work in Chapter 10, section 10.8, the current version of the *DataFlow-to-State-Model* agent only supports the translation of a particular business process pattern that is expressed in the detailed DFD model into the associated *State Model*. It is desirable to enhance the current algorithm in order to construct more complex business tasks (such as a task that consists of a collection of *input and output* subtasks along with one or more subtasks that performs a single CRUD operation).

To achieve different pattern coverage, a further consideration might be required in the proposed *priority scoring* algorithm, in order to enable having more detailed scores to cover any possible repetition of task type. For example, in the current version, the score of *read* is 3 and the score of *input* is 4. On some occasions, when having a system that requires more than one *input* step, the *scoring algorithm* gives both (input) steps 4, and a default rule is applied to consider the first input task appearing in the model to be the first one in the sequence of input and the rest is likewise. An additional step for analysing the data on flow together with the flow-type is needed to extract complex (longer) sequences.

### 11.5.6 Constructing a Separated Business Logic Layer

The current version of BUILD is able to construct only a 2-tier information systems design, which consists of a *presentation* and *data storage (database)* tier. The specifications of these tiers are expressed via a number of $\mu$ML models. In the generated systems from BUILD, the *business logic* is embedded in the *data storage* tier, and invoked from UIs, the *presentation* tier.

The logic is implemented via a number of stored procedures, each of which performs a specific CRUD operation on the database. This would not be applicable for *computational* business tasks that perform different jobs rather than CRUD operations.

From that, on some occasions, having a separate *business logic* tier brings benefits to the design, in which it enables dealing with some types of *computational* business tasks. As a consequence, the proposed method (BUILD) might be used on a large scale, dealing with systems that require a further security and performance of the overall information system design.

Constructing a 3-tier architecture might be achieved by modelling business logic in a separate model apart from the DBQ model that represents pure OO code. The $\mu$ML *Code Model* serves this purpose. Chapter 6 introduces the notation and semantics of some of its core concepts and the rules-structure and definition of the translation step for deriving the *Code Model* from the *DataFlow* and *DBQ model*.

In order to continue the work in this direction, the notation of the *Task Model* must be extended to support new types of business activities, e.g. *arithmetic* in addition to the existing types of tasks. Choosing the right level of abstraction is important; for example, high-level tasks representing predefined accounting operations may be preferable to low-level arithmetic. Moreover, appropriate transformation rules must be designed to derive the equivalent code from related models.

### 11.5.7 Integrating BUILD with Eclipse Graphical Modeling Framework (GMF)

In order to improve the interoperability of the BUILD framework, designing a graphical editor tool is an essential improvement of the work. Eclipse, via the Graphical Modeling Project[37] offered a set of generative components and runtime infrastructures for creating graphical editors based on the Eclipse EMF[36] artefact and the GEF[35] framework. Using the technology provided by GEF[35] on the top of our approach, to invent rich graphical editors, would allow end-users to sketch their system models in a convenient way. In the literature survey (Chapter 2), many model transformation approached and development methodologies support their work with an Eclipse-based CASE tool, such as, ATL[33], Epsilon[34] and WebML[124] via its WebRation[4] CASE tool.

## 11.6 Final Remarks

In summary, the thesis introduced a user-friendly modelling language, called Micro-Modelling Language ($\mu$ML), aimed at supporting end-user model-driven engineering by raising the level of abstraction rather than using other *UML* approaches, which requires more technical knowledge. The main purpose of $\mu$ML is to tackle some issues regarding the ambiguity of *UML* semantics and the complexity of its models. This is because the Models in *UML* are too complex and eclectic to be given a single, clear interpretation, while paradoxically not covering all of the views that are needed to completely specify a software system.

Designing this lightweight language ($\mu$ML) that has a simpler notation with cleaner semantics enabled capturing the functional requirements of systems in a generic way, using familiar concepts to a business domain. This was demonstrated via examples and case studies throughout the thesis. The individual $\mu$ML model is smaller and more restricted; but more types of models were used together to cover the different interlinking aspects (views) of an information system.

In addition, possible total and partial forward transformations, between different kinds of model, were identified. The rules of transformation are designed in a hybrid style, gathering benefits from the known *declarative* and *imperative* approaches. Each rule is implemented as a *Java* class (agent), forming a layered forward model transformation strategy to generate an executable 2-tier information system from requirements models.

From that, a novel model-driven engineering approach (BUILD) was established. The BUILD method introduced a development methodology that contributes in the area of model-driven information systems engineering. The method introduces a development technique that allows domain experts to lead the development process, and participate more at various development stages, by constructing initial MDE artifacts, and annotating some of the intermediate models at a later phase.

The engagement of business users is achieved via the proposed $\mu$ML. Through BUILD, the development lifecycle starts with models, which hold business knowledge, defined manually by end-users. More views of the desired system are constructed automatically during the development process, in which the transformation rules evolve the captured requirements to predict, and produce, more detailed concepts, at the design and code generation phases.

# List of References


[1] Abstract Solutions: Complexity Simplified. Xuml :: Action specification languages. `http://www.kc.com/XUML/asl.php`, 2014.

[2] Abstract Solutions: Complexity Simplified. Xuml :: Executable uml. `http://www.kc.com/XUML/executableumldescription.php`, 2014.

[3] Acceleo. Planet acceleo. `http://www.acceleo.org/pages/planet-acceleo`, February 2010.

[4] Roberto Acerbis, Aldo Bongio, Marco Brambilla, and Stefano Butti. Webratio 5: An eclipse-based case tool for engineering web applications. In Luciano Baresi, Piero Fraternali, and Geert-Jan Houben, editors, *Web Engineering*, volume 4607 of *Lecture Notes in Computer Science*, pages 501–505. Springer Berlin Heidelberg, 2007.

[5] David H Akehurst, Behzad Bordbar, Michael J Evans, J Howells, W Gareth, and Klaus D McDonald-Maier. Sitra: Simple transformations in java. In *Model Driven Engineering Languages and Systems*, pages 351–364. Springer, 2006.

[6] Joao Paulo Almeida, Remco Dijkman, Marten Van Sinderen, and Luis Ferreira Pires. On the notion of abstract platform in MDA development. In *Enterprise Distributed Object Computing Conference, 2004. EDOC 2004. Proceedings. Eighth IEEE International*, pages 253–263. IEEE, 2004.

[7] Zaid Altahat, Tzilla Elrad, and Didier Vojtisek. Using aspect oriented modeling to localize implementation of executable models. In *Models and Aspects workshop, at ECOOP 2007*, 2007.

[8] AndroMDA. Generate components quickly with andromda. `http://www.andromda.org/docs/index.html`, January 2011.

[9] Bordbar B. Sitra: Simple transformer. `http://www.cs.bham.as.uk/~bxb/Sitra/index.html`, 2014.

[10] Dave Bacon. An introduction to digital design: Moore and mealy machines. University of Washington, Lecture. `https://courses.cs.washington.edu/courses/cse370/06sp/pdfs/lecture18.pdf`, 2006.

[11] Elisa L. A. Baniassad and Siobhán Clarke. Theme: An approach for aspect-oriented analysis and design. In *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, pages 158–167, 2004.

[12] Luciano Baresi and Reiko Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In *Graph Transformation*, pages 402–429. Springer, 2002.

[13] Leandro Buss Becker, Carlos Eduardo Pereira, Octávio Páscoa Dias, JP Teixeira, and IM Teixeira. Mosys: a methodology for automatic object identification from system specification. In *Object-Oriented Real-Time Distributed Computing, 2000.(ISORC 2000) Proceedings. Third IEEE International Symposium on*, pages 198–201. IEEE, 2000.

[14] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on uml class diagrams. *Artif. Intell.*, 168(1):70–118, October 2005.

[15] Matthias Biehl. Literature study on model transformations. *Royal Institute of Technology, Tech. Rep. ISRN/KTH/MMK*, 2010.

[16] Marco Brambilla, Piero Fraternali, and Massimo Tisi. A metamodel transformation framework for the migration of webml models to mda. In *MDWE, CEUR Workshop Proceedings*, volume 389, pages 91–105. Citeseer, 2008.

[17] Peter Buneman, Mary Fernandez, and Dan Suciu. Unql: a query language and algebra for semistructured data based on structural recursion. *The VLDB JournalThe International Journal on Very Large Data Bases*, 9(1):76–110, 2000.

[18] Andrew Carton, Cormac Driver, Andrew Jackson, and Siobhán Clarke. Model-driven theme/uml. *T. Aspect-Oriented Software Development VI*, 6:238–266, 2009.

[19] Case-Tools. Optimalj case tool: Leading mda/uml tool generates j2ee/ejb code from uml models, reads xmi. `http://www.case-tools.org/tools/optimalj.html`, August 2010.

[20] Grant Wing Fai Chan. Model-based generation of java code. Master's thesis, 2003.

[21] Karina Chong, María Verónica Macías Mendoza, and Monique Snoeck. Experiences with the use of MERODE in the development of a web based application. In *VII Jornadas Iberoamericanas de Ingeniería de Software e Ingeniería del Conocimiento 2008, Guayaquil, Ecuador, January 30 - February 1, 2008. Proceedings*, pages 421–426, 2008.

[22] Tony Clark, Andy Evans, Paul Sammut, and James Willans. Transformation language design: A metamodelling foundation. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 3256 of *Lecture Notes in Computer Science*, pages 13–21. Springer Berlin Heidelberg, 2004.

[23] Tony Clark, Paul Sammut, and James Willans. Applied metamodelling: a foundation for language driven development. 2008.

[24] Krzysztof Czarnecki. Overview of generative software development. In *Unconventional Programming Paradigms*, pages 326–341. Springer, 2005.

[25] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17. Citeseer, 2003.

[26] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[27] Sofia Larissa da Costa, VV Graciano Neto, Luiz Fernando Batista Loja, and Juliano Lopes de Oliveira. A metamodel for automatic generation of enterprise information systems. In *Anais do I Congresso Brasileiro de Software: Teoria e Prática-I Workshop Brasileiro de Desenvolvimento de Software Dirigido por Modelos*, volume 8, pages 45–52, 2010.

[28] Alexandre Cláudio de Almeida, Glauber Boff, and Juliano Lopes de Oliveira. A framework for modeling, building and maintaining enterprise information systems software. In *Software Engineering, 2009. SBES'09. XXIII Brazilian Symposium on*, pages 115–125. IEEE, 2009.

[29] Buddhima De Silva and Athula Ginige. Meta-model to support end-user development of web based business information systems. In *Web Engineering*, pages 248–253. Springer, 2007.

[30] Buddhima De Silva and Athula Ginige. Meta-model to support end-user development of web based business information systems. In Luciano Baresi, Piero Fraternali, and Geert-Jan Houben, editors, *Web Engineering*, volume 4607 of *Lecture Notes in Computer Science*, pages 248–253. Springer Berlin Heidelberg, 2007.

[31] Dov Dori and Edward F. Crawley. *Object-Process Methodology: A Holistic Systems Paradigm*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[32] Eclipse. Viatra2: Project overview. `http://www.eclipse.org/gmt/VIATRA2`, 2010.

[33] Eclipse. Atl: A model transformation technology. `http://www.eclipse.org/atl/`, 2011.

[34] Eclipse. Epsilon framework. `http://www.eclipse.org/gmt/epsilon`, May 2011.

[35] Eclipse. Gef (graphical editing framework). `http://www.eclipse.org/gef/`, 2012.

[36] Eclipse. Eclipse modeling framework project (emf). `http://www.eclipse.org/modeling/emf`, 2013.

[37] Eclipse. Graphical modeling project. `http://www.eclipse.org/modeling/gmp`, 2013.

[38] João M Fernandes, Johan Lilius, and Dragos Truscan. Integration of dfds into a uml-based model-driven engineering approach. *Software & Systems Modeling*, 5(4):403–428, 2006.

[39] Robert Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit. *Aspect-oriented Software Development*. Addison-Wesley Professional, first edition, 2004.

[40] Frédéric Fondement and Raul Silaghi. Defining model driven engineering processes. In *Third International Workshop in Software Model Engineering (WiSME), held at the 7th International Conference on the Unified Modeling Language (UML)*. Citeseer, 2004.

[41] Juan Pedro Silva Gallino. Composing models with six different tools: a comparative study. 1st International Workshop on Model Transformation with ATL, Nantes, France, July 2009.

[42] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[43] Chris Gane and Trish Sarson. *Structured Systems Analysis: Tools and Techniques*. McDonnell Douglas Systems Integration Company, 1977.

[44] Miguel Garcia. Formalization of qvt-relations: Ocl-based static semantics and alloy-based validation. In *Proceedings of the Second Workshop on MDSD Today*, pages 21–30, 2008.

[45] Athula Ginige and Buddhima De Silva. Cbeads: A framework to support meta-design paradigm. In Constantine Stephanidis, editor, *Universal Acess in Human Computer Interaction. Coping with Diversity*, volume 4554 of *Lecture Notes in Computer Science*, pages 107–116. Springer Berlin Heidelberg, 2007.

[46] Arda Göknil, N Yasemin Topaloglu, and KG Van Den Berg. Operation composition in model transformations with complex source patterns. Technical report, Centre for Telematics and Information Technology, University of Twente, 2008.

[47] Jack Greenfield and Keith Short. Software factories: assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 16–27. ACM, 2003.

[48] Distributed System Group. Aspect-oriented architectures. `http://www.dsg.cs.tcd.ie/aspects/themeUML`, 2009.

[49] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[50] Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. Towards a compositional approach to model transformation for software development. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 468–475. ACM, 2009.

[51] IBM. Rational rhapsody family. `http://www-03.ibm.com/software/products/en/ratirhapfami`, 2011.

[52] IRISA. Kermeta: Breathe life into your metamodels. `http://www.kermeta.org`, 2010.

[53] ISIS. Great: Gaph rewriting and transformation. `http://www.isis.vanderbilt.edu/tools/GReAT`, May 2010.

[54] ISIS. Model integrated computing. `http://www.isis.vanderbilt.edu/MIC/`, 2011.

[55] Andrew Jackson, Jacques Klein, Benoit Baudry, Siobhán Clarke, et al. Executable aspect oriented models for improved model testing. In *ECMDA workshop on Integration of Model Driven Development and Model Driven Testing.*, 2006.

[56] X Jai. zoom. `http://www.se.cs.depaul.edu/ise/zoom/zoom.html/`, 2011.

[57] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. Model driven language engineering with kermeta. In *Generative and Transformational Techniques in Software Engineering III*, pages 201–221. Springer, 2011.

[58] Tao Jiang and WenYun Zheng. Research on formalization of domain-specific metamodeling language based on first-order logic. In *Computer Science and Automation Engineering (CSAE), 2011 IEEE International Conference on*, volume 4, pages 170–174, June 2011.

[59] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of computer programming*, 72(1):31–39, 2008.

[60] Judith Kabeli and Peretz Shoval. Comprehension and quality of analysis specifications: a comparison of foom and opm methodologies. *Information and Software Technology*, 47(4):271–290, 2005.

[61] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[62] Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.

[63] E. Kindler and R. Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical report, Technical Report tr-ri-07-284, University of Paderborn, 2007.

[64] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The epsilon object language (eol. In *In: Proceedings European Conference in Model Driven Architecture (EC-MDA) 2006*, pages 128–142. Springer, 2006.

[65] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The epsilon object language (eol). In *Model Driven Architecture–Foundations and Applications*, pages 128–142. Springer, 2006.

[66] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. The epsilon transformation language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin Heidelberg, 2008.

[67] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The epsilon transformation language. In *Theory and practice of model transformations*, pages 46–60. Springer, 2008.

[68] Andreas Kraus, Alexander Knapp, and Nora Koch. Model-driven generation of web applications in uwe. *MDWE*, 261, 2007.

[69] Ivan Kurtev, Klaas Van Den Berg, and Frédéric Jouault. Evaluation of rule-based modularization in model transformation languages illustrated with atl. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1202–1209. ACM, 2006.

[70] Unified Modelling Language. Uml resource page. `http://www.uml.org`, 2011.

[71] Kevin Lano. The UML-RSDS manual. Technical report, Department of Informatics, King's College London, 2014.

[72] Kevin Lano and Shekoufeh Kolahdouz-Rahimi. Specification and verification of model transformations using uml-rsds. In Dominique Mry and Stephan Merz, editors, *Integrated Formal Methods*, volume 6396 of *Lecture Notes in Computer Science*, pages 199–214. Springer Berlin Heidelberg, 2010.

[73] Michael Lawley and Jim Steel. Practical declarative model transformation with tefkat. In *Satellite Events at the MoDELS 2005 Conference*, pages 139–150. Springer, 2006.

[74] KU Leuven. Jmermaid: Merode modeling aid. `http://merode.econ.kuleuven.ac.be/mermaid.aspx`, 2011.

[75] KU Leuven. Merode: Is merode something for you? `http://merode.econ.kuleuven.ac.be/merodefaq.aspx`, 2011.

[76] Hongming Liu, Xiaoping Jia, et al. Model transformation using a simplified metamodel. *Journal of Software Engineering and Applications*, 3(07):653, 2010.

[77] Vincent Lussenburg, Tijs Van Der Storm, Jurgen Vinju, and Jos Warmer. Mod4j: a qualitative case study of model-driven software development. In *Model Driven Engineering Languages and Systems*, pages 346–360. Springer, 2010.

[78] William E. McUmber and Betty H. C. Cheng. A general framework for formalizing uml with formal languages. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 433–442, Washington, DC, USA, 2001. IEEE Computer Society.

[79] Tom Mens. Model transformation: A survey of the state of the art. *Model-Driven Engineering for Distributed Real-Time Systems: MARTE Modeling, Model Transformations and their Usages*, pages 1–19, 2010.

[80] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. 04101 discussion - a taxonomy of model transformations. In *Language Engineering for Model-Driven Software Development*, 2004.

[81] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[82] Shui ming Ho and Kung kiu Lau. Catalysis framework in first-order logic. In *Proc. of FME03 Workshop on Formal Aspects of Component Software*, 2003.

[83] Nathalie Moreno, Piero Fraternali, and Antonio Vallecillo. Webml modelling in uml. *IET software*, 1(3):67–80, 2007.

[84] Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer, Jean-Marc Jézéquel, et al. On executable meta-languages applied to model transformations. In *Model Transformations In Practice Workshop*, 2005.

[85] OMG. Object constraint language. `http://www.omg.org/spec/OCL/2.2/`.

[86] OMG. Object management group (omg). `http://www.omg.org/`, 1997.

[87] OMG. Meta object facility (mof) core specification version 2.0. `http://www.omg.org/spec/MOF/2.0/HTML/`, 2006.

[88] OMG. Query/view/transformation specification version 1.1. `http://www.omg.org/spec/QVT/1.1/PDF/`, 2011.

[89] OMG. Model driven architecture (mda). `http://www.omg.org/mda/`, 2014.

[90] OPCAT. Rapid conceptual design for complex systems. `http://www.opcat.com`, 2005.

[91] F. Perez, P. Valderas, and J. Fons. Allowing end-users to participate within model-driven development approaches. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 187–190, Sept 2011.

[92] Jakob Pinggera, Stefan Zugal, Barbara Weber, Dirk Fahland, Matthias Weidlich, Jan Mendling, and Hajo A. Reijers. How the structuring of domain knowledge helps casual process modelers. In *Proceedings of the 29th International Conference on Conceptual Modeling*, ER'10, pages 445–451, Berlin, Heidelberg, 2010. Springer-Verlag.

[93] John D Poole. Model-driven architecture: Vision, standards and emerging technologies. In *Workshop on Metamodeling and Adaptive Object Models, ECOOP*, volume 2001, 2001.

[94] Roger Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, Inc., New York, NY, USA, 7 edition, 2010.

[95] Sreedhar Reddy, R Venkatesh, and Zahid Ansari. A relational approach to model transformation using qvt relations. *Tata Research Development and Design Centre, Pune, India.–2006./Internet: http://www. iist. unu. edu/˜ vs/wiki-files/QVT-TRDCC. pdf.*

[96] Iris Reinhartz-Berger and Dov Dori. Object-process methodology (opm) vs. uml-a code generation perspective. In *CAiSE Workshops (1)*, pages 275–286. Citeseer, 2004.

[97] Jochen Rode, Mary Beth Rosson, and Manuel A. Pérez-Quiñones. End-users' mental models of concepts critical to web application development. In *2004 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2004), 26-29 September 2004, Rome, Italy*, pages 215–222, 2004.

[98] Sergio Segura, David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. Automated merging of feature models using graph transformations. In *Generative and Transformational Techniques in Software Engineering II*, pages 489–505. Springer, 2008.

[99] Lijun Shan and Hong Zhu. Semantics of metamodels in uml. In *Proceedings of the 2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering*, TASE '09, pages 55–62, Washington, DC, USA, 2009. IEEE Computer Society.

[100] Juan Pedro Silva, Miguel de Miguel, Javier F Briones, and Alejandro Alonso. Composing models with six different tools: a comparative study. *CEUR-WS MtATL*, pages 103–118, 2009.

[101] Anthony J.H. Simons. Remodel: Reusable model design languages: Generating software systems by model transformation and adaptation. `http://staffwww.dcs.shef.ac.uk/people/A.Simons/remodel/`, 2011.

[102] Anthony J.H. Simons. Jast: Java abstract syntax trees. `http://staffwww.dcs.shef.ac.uk/people/A.Simons/jast/`, 2012.

[103] Monique Snoeck. *Object-Oriented Enterprise Modelling with Merode.* Leuven University Press, 1999.

[104] Monique Snoeck, Raf Haesen, Herman Buelens, Manu De Backer, and Geert Monsieur. Computer aided modelling exercises. *Informatics in education*, 6(1):231–248, 2007.

[105] SourceForge. The jamda project. `http://www.jamda.sourceforge.net`, 2003.

[106] Michael Spahn, Christian Dorner, and Volker Wulf. End user development: approaches towards a flexible software design. Presented in ECIS 2008, 9th 11th June 2008, Galway, Ireland, June 2008.

[107] Michael Spahn and Volker Wulf. End-user development of enterprise widgets. In *Proceedings of the 2nd International Symposium on End-User Development*, IS-EUD '09, pages 106–125, Berlin, Heidelberg, 2009. Springer-Verlag.

[108] Ahmad F. Subahi. ReMoDeL Database Generator. Master's thesis, University of Sheffield, Department of Computer Science, Sheffield, United Kingdom, 2010.

[109] Ahmad F. Subahi and Anthony J.H. Simons. A multi-level transformation from conceptual data models to database scripts using java agents. In *Proc. in the Workshop on Composition and Evaluation of Model Transformations*, 2011.

[110] Eugene Syriani and Hans Vangheluwe. Matters of model transformation. *Relatório Técnico SOCS-TR-2009.2, School of Computer Science, McGill University*, 2009.

[111] Janos Sztipanovits and Gabor Karsai. Generative programming for embedded systems. In *Generative Programming and Component Engineering*, pages 32–49. Springer, 2002.

[112] Gabriele Taentzer. Agg: A graph transformation environment for modeling and validation of software. In *Applications of Graph Transformations with Industrial Relevance*, pages 446–453. Springer, 2004.

[113] Hee Beng Kuan Tan, Yong Yang, and Lei Bian. Systematic transformation of functional analysis model into oo design and implementation. *Software Engineering, IEEE Transactions on*, 32(2):111–135, 2006.

[114] F.T.S.D Team. Fujaba tool suite. `http://www.fujaba.de/no_cache/home.html`, 2011.

[115] Laurence Tratt. Model transformations and tool integration. *Software & Systems Modeling*, 4(2):112–122, 2005.

[116] uml diagrams.org. State machine diagrams. `http://www.uml-diagrams.org/state-machine-diagrams.html`, 2014.

[117] UWE. Uml-based web engineering. `http://uwe.pst.ifi.lmu.de/aboutUwe.html`, 2011.

[118] Francisco Valverde, Ignacio Panach, and Oscar Pastor. An abstract interaction model for a mda software production method. In *Tutorials, posters, panels and industrial contributions at the 26th international conference on Conceptual modeling-Volume 83*, pages 109–114. Australian Computer Society, Inc., 2007.

[119] Bert Vanhooff, Dhouha Ayed, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers. Uniti: A unified transformation infrastructure. In *Model Driven Engineering Languages and Systems*, pages 31–45. Springer, 2007.

[120] Juan M Vara, Maria Valeria De Castro, Marcos Didonet Del Fabro, and Esperanza Marcos. Using weaving models to automate model-driven web engineering proposals. *International Journal of Computer Applications in Technology*, 39(4):245–252, 2010.

[121] Gergely Varró. Towards incremental graph transformation in fujaba. In *Proc. of the 2nd International Fujaba Days*, pages 3–6, Darmstadt, Germany, September 2004.

[122] W3C. Extensible markup language (xml). `http://www.w3.org/XML/`, 2014.

[123] Dennis Wagelaar. Composition techniques for rule-based model transformation languages. In *Theory and Practice of Model Transformations*, pages 152–167. Springer, 2008.

[124] WebML.org. The web modelling language. `http://www.webml.org/webml/page1.do`, 2004.

[125] Hong Zhu, Lijun Shan, Ian Bayley, and Richard Amphlett. *Formal Descriptive Semantics of UML and Its Applications*, pages 95–123. John Wiley & Sons, Inc., 2009.

# Appendices

# A

# Appendix: Models and Executable Code of Experiment (1)

## A.1 Complete Models and Full results of Experiment (1)

This Appendix presents the XML representations of all $\mu$ML models of the Module Management System case study, and the completed Java and MySQL code generated from these models via BUILD. The experiment presented previously in chapter 10.

### Experiment (1): Requirement Models Construction

Listing A.1: CaseStudy.java

```
package mde.example;

import java.io.File;
import java.io.IOException;
import mde.data.model.DDiagram;
import mde.database.gen.DumpFileGenerator;
import mde.mysql.gen.MySQLDumpFileGenerator;
import mde.database.gen.TreeException;
import mde.dataflow.model.DfDiagram;
import mde.dbs.model.Schema;
import mde.dfd2ddfd.rule.DfDiagramToDfDiagram;
import mde.dfd2state.rule.DfdDiagramToStDiagram;
import mde.dm2schem.rule.DDiagramToSchema2;
import mde.dm_dfd_state2code.rule.DDiagramToCDiagram2x;
import mde.gui.gen.CodeFileGenerator;
import mde.gui.model.GuiBoundary;
import mde.gui.model.GuiDiagram;
import mde.impact.model.ImpBoundary;
import mde.impact.model.ImpDeleteFlow;
import mde.impact.model.ImpDiagram;
import mde.impact.model.ImpCreateFlow;
import mde.impact.model.ImpObject;
import mde.impact.model.ImpReadFlow;
import mde.impact.model.ImpRole;
import mde.impact.model.ImpTask;
import mde.impact.model.ImpUpdateFlow;
import mde.inf2dm.rule.InfDiagramToDDiagram;
import mde.information.model.Association;
import mde.information.model.Entity;
import mde.information.model.Attribute;
import mde.information.model.Role;
import mde.javagui.gen.JavaCodeFileGenerator;
import mde.model.Type;
import mde.state.model.StDiagram;
import mde.state2win.rule.StDiagramToGuiDiagram;
import mde.task.model.Actor;
import mde.task.model.Boundary;
```

```
import mde.task.model.Composition;
import mde.task.model.Diagram;
import mde.task.model.Participation;
import mde.task.model.Task;
import mde.taskimpact2dataflow.rule.DiagramToDfDiagram;
import org.jast.ast.ASTWriter;

public class CaseStudy {

    public static void main(String[] args) throws IOException,
        TreeException, mde.gui.gen.TreeException {
        // TODO Auto-generated method stub

        //Construct the Task Model
        Diagram taskModel = new Diagram();
        Boundary boundary = new Boundary("Module Management");

        Task manage = new Task("Manage Module");
        Task add = new Task("Add Module");
        Task delete = new Task("Delete Module");
        Task modify = new Task("Modify Module");
        Task see = new Task("See Description");
        Actor actor1 = new Actor("Staff");
        Actor actor2 = new Actor("Student");

        Composition comp = new Composition();
        comp.addRole(new mde.task.model.Role("manage", manage));
        comp.addRole(new mde.task.model.Role("add", add));
        comp.addRole(new mde.task.model.Role("delete", delete));
        comp.addRole(new mde.task.model.Role("modify", modify));

        Participation link1 = new Participation();
        link1.addRole(new mde.task.model.Role("staff",actor1));
        link1.addRole(new mde.task.model.Role("add", add));
        Participation link2 = new Participation();
        link2.addRole(new mde.task.model.Role("staff",actor1));
        link2.addRole(new mde.task.model.Role("modify", modify));
        Participation link3 = new Participation();
        link3.addRole(new mde.task.model.Role("staff",actor1));
        link3.addRole(new mde.task.model.Role("delete", delete));
        Participation link4 = new Participation();
        link4.addRole(new mde.task.model.Role("student",actor2));
        link4.addRole(new mde.task.model.Role("see", see));
        Participation link5 = new Participation();
        link5.addRole(new mde.task.model.Role("see", see));
        link5.addRole(new mde.task.model.Role("student",actor2));

        boundary.addActor(actor1);
        boundary.addActor(actor2);
        boundary.addTask(manage);
        boundary.addTask(add);
        boundary.addTask(delete);
        boundary.addTask(modify);
        boundary.addTask(see);

        boundary.addParticipation(link1);
        boundary.addParticipation(link2);
        boundary.addParticipation(link3);
        boundary.addParticipation(link4);
        boundary.addParticipation(link5);
        boundary.addComposition(comp);

        taskModel.addBoundary(boundary);

        ASTWriter writer = new
            ASTWriter(new File("Manage_Module_taskModel.xml"));
        writer.usePackage("mde.task.model", "xmlns:task");
        writer.writeDocument(taskModel);
        writer.close();

        System.out.println("(1) Task Model is Created by user.");
        // ------------------------------------------ //

      // Construct the Impact Model
      ImpDiagram ImpactModel = new ImpDiagram();
      ImpBoundary impboundary = new ImpBoundary("Module Management");
      ImpTask impManage = new ImpTask("Manage Modules");
      ImpTask impAdd = new mde.impact.model.ImpTask("Add Module");
      ImpTask impDelete = new mde.impact.model.ImpTask("Delete Module");
      ImpTask impModify = new mde.impact.model.ImpTask("Modify Module");
      ImpTask impSee = new mde.impact.model.ImpTask("See Description");

        ImpObject impObj1 = new ImpObject("Module");

        ImpCreateFlow cf = new ImpCreateFlow();
        ImpRole impcf1 = new ImpRole("module", impObj1);
        ImpRole impcf2 = new ImpRole("add", impAdd);
        cf.addImpRole(impcf2);
        cf.addImpRole(impcf1);

        ImpDeleteFlow df = new ImpDeleteFlow();
        ImpRole impdf1 = new ImpRole("module", impObj1);
        ImpRole impdf2 = new ImpRole("delete", impDelete);
        df.addImpRole(impdf2);
```

```
        df.addImpRole(impdf1);
        ImpUpdateFlow uf = new ImpUpdateFlow();
        ImpRole impuf1 = new ImpRole("module", impObj1);
        ImpRole impuf2 = new ImpRole("modify", impModify);
        uf.addImpRole(impuf2);
        uf.addImpRole(impuf1);
        ImpReadFlow rf = new ImpReadFlow();
        ImpRole imprf1 = new ImpRole("module", impObj1);
        ImpRole imprf2 = new ImpRole("see", impSee);
        rf.addImpRole(imprf1);
        rf.addImpRole(imprf2);
        impboundary.addImpTask(impManage);
        impboundary.addImpTask(impAdd);
        impboundary.addImpTask(impDelete);
        impboundary.addImpTask(impModify);
        impboundary.addImpTask(impSee);
        impboundary.addImpObject(impObj1);
        impboundary.addImpCreateFlow(cf);
        impboundary.addImpDeleteFlow(df);
        impboundary.addImpUpdateFlow(uf);
        impboundary.addImpReadFlow(rf);

        ImpactModel.addImpBoundary(impboundary);

        ASTWriter writer1 = new
            ASTWriter(new File("Manage_Module_impactModel.xml"));
        writer1.usePackage("mde.impact.model", "xmlns:imp");
        writer1.writeDocument(ImpactModel);
        writer1.close();

        System.out.println("(2) Impact Model is Created by user.");
        // ---------------------------------------- //

        // Construct the Information Model
        mde.information.model.Diagram informationModel = new
            mde.information.model.Diagram();

        Entity moduleEntity = new Entity("Module");
        Attribute attr12 = new Attribute("code", new Type("Integer"))
            .setIdentifier(true);
        Attribute attr13 = new Attribute("title", new Type("String"));
        Attribute attr14 = new Attribute("credit", new Type("Integer"));
        Attribute attr15 = new Attribute("desc", new Type("String"));

        moduleEntity.addAttribute(attr12);
        moduleEntity.addAttribute(attr13);
        moduleEntity.addAttribute(attr14);
        moduleEntity.addAttribute(attr15);

        informationModel.addEntity(moduleEntity);

        informationModel.addAssociation(new Association()
        .addRole(new Role("module", informationModel
            .getEntity("Module")).setMultiple(true))
        .addRole(new Role("student", informationModel
            .getEntity("Student")).setMultiple(true)));

        ASTWriter writer2 = new
            ASTWriter(new File("Manage_Module_informationModel.xml"));
        writer2.usePackage("mde.information.model", "xmlns:inf");
        writer2.writeDocument(informationModel);
        writer2.close();

        System.out.println("(3) Information Model is Created by user.");
        // ---------------------------------------------- //

        //Generate DataFlow Model
        DiagramToDfDiagram topRule = new DiagramToDfDiagram();
        DfDiagram  dataflowModel = topRule.translate(taskModel, ImpactModel);

        ASTWriter writer3 = new
            ASTWriter(new File("Manage_Module_DataFlowModel.xml"));
        writer3.usePackage("mde.dataflow.model", "xmlns:dfd");
        writer3.writeDocument(dataflowModel);
        writer3.close();

        System.out.println("(4) DataFlow Model is Created"+
            "(Task + Impact --> DataFlow).");
        // ---------------------------------------- //

        //Adding some data on flows in dataflow model.

        // input to add
        dataflowModel.getDfBoundaries().get(0).getDfInputFlows()
            .get(0).addDataonflow("code");
        dataflowModel.getDfBoundaries().get(0).getDfInputFlows()
            .get(0).addDataonflow("title");
        dataflowModel.getDfBoundaries().get(0).getDfInputFlows()
            .get(0).addDataonflow("credit");
        dataflowModel.getDfBoundaries().get(0).getDfInputFlows()
            .get(0).addDataonflow("desc");
        // input to update credit
        dataflowModel.getDfBoundaries().get(0).getDfInputFlows()
            .get(1).addDataonflow("code");
```

```
        dataflowModel.getDfBoundaries().get(0).getDfInputFlows()
            .get(1).addDataonflow("credit");
        // input to delete
        dataflowModel.getDfBoundaries().get(0).getDfInputFlows()
            .get(2).addDataonflow("code");
        // input to see
        dataflowModel.getDfBoundaries().get(0).getDfInputFlows()
            .get(3).addDataonflow("code");
        // create assignment
        dataflowModel.getDfBoundaries().get(0).getDfCreateFlows()
            .get(0).addAssignment("@code = code");
        dataflowModel.getDfBoundaries().get(0).getDfCreateFlows()
            .get(0).addAssignment("@title = title");
        dataflowModel.getDfBoundaries().get(0).getDfCreateFlows()
            .get(0).addAssignment("@credit = credit");
        dataflowModel.getDfBoundaries().get(0).getDfCreateFlows()
            .get(0).addAssignment("@desc = desc");
        // create
        dataflowModel.getDfBoundaries().get(0).getDfCreateFlows()
            .get(0).addDataonflow("code");
        dataflowModel.getDfBoundaries().get(0).getDfCreateFlows()
            .get(0).addDataonflow("title");
        dataflowModel.getDfBoundaries().get(0).getDfCreateFlows()
            .get(0).addDataonflow("credit");
        dataflowModel.getDfBoundaries().get(0).getDfCreateFlows()
            .get(0).addDataonflow("desc");
        // delete filter
        dataflowModel.getDfBoundaries().get(0).getDfDeleteFlows()
            .get(0).setFilter("[@code = code] @code");
        // delete
        dataflowModel.getDfBoundaries().get(0).getDfDeleteFlows()
            .get(0).addDataonflow("code");
        // read filter
        dataflowModel.getDfBoundaries().get(0).getDfReadFlows()
            .get(0).setFilter("[@code = code] @desc");
        // read
        dataflowModel.getDfBoundaries().get(0).getDfReadFlows()
            .get(0).addDataonflow("desc");
        // update filter
        dataflowModel.getDfBoundaries().get(0).getDfUpdateFlows()
            .get(0).setFilter("[@code = code] @code, @credit");
        // update assignment
        dataflowModel.getDfBoundaries().get(0).getDfUpdateFlows()
            .get(0).addAssignment("@credit = credit");
        // update
        dataflowModel.getDfBoundaries().get(0).getDfUpdateFlows()
            .get(0).addDataonflow("code");
        dataflowModel.getDfBoundaries().get(0).getDfUpdateFlows()
            .get(0).addDataonflow("credit");

        System.out.println("(4) Data on flows are added in the DataFlow model.");
        // ------------------------------------------------ //

        // Generate Detailed DatFlow Model
        DfDiagramToDfDiagram topRule6 = new DfDiagramToDfDiagram();
        DfDiagram detailed_dataflowModel = topRule6.translate(dataflowModel);

        ASTWriter writer6 = new ASTWriter(new
            File("Manage_Module_Detailed_DataFlowModel.xml"));
        writer6.usePackage("mde.dataflow.model", "xmlns:dfd");
        writer6.writeDocument(detailed_dataflowModel);
        writer6.close();

        System.out.println("(5) Detailed DataFlow Model is Created"+
            "(DataFlow Model --> DataFlow Model).");
        // ------------------------------------------------ //

        //Generate Data Model
        InfDiagramToDDiagram topRule4 = new InfDiagramToDDiagram();
        DDiagram  dataModel = topRule4.translate(informationModel);

        ASTWriter writer4 = new ASTWriter(new
            File("Manage_Module_DataModel.xml"));
        writer4.usePackage("mde.data.model", "xmlns:data");
        writer4.usePackage("mde.model", "xmlns:mod");
        writer4.writeDocument(dataModel);
        writer4.close();

        System.out.println("(5) Data Model is Created"+
            "(Information Model --> Data Model).");
        // ------------------------------------------------------ //

        //Generate State Model for Screens Navigation
        //(DataFlow Model ---> State Model)
        DfdDiagramToStDiagram topRule7 = new DfdDiagramToStDiagram();
        StDiagram  stateModel=topRule7.translate(detailed_dataflowModel, dataModel);

        ASTWriter writer7 = new
            ASTWriter(new File("Manage_Module_StateModel.xml"));
        writer7.usePackage("mde.state.model", "xmlns:state");
        writer7.writeDocument(stateModel);
        writer7.close();

        System.out.println("(6) State Model is Created"+
```

```
                "(Detailed DataFlow Model --> State Model).");
            // -------------------------------------------------- //

            //Generate Gui Description Model
            //(State Model ---> Gui Specification Model)
            StDiagramToGuiDiagram topRule9 = new StDiagramToGuiDiagram();
            GuiDiagram  GuiModel = topRule9.translate(stateModel);

            ASTWriter writer9 = new
                ASTWriter(new File("Manage_Module_GuiModel.xml"));
            writer9.usePackage("mde.gui.model", "xmlns:gui");
            writer9.writeDocument(GuiModel);
            writer9.close();

            System.out.println("(7) Gui Specification Model is Created"+
                "(State Model --> Gui Model).");
            // ---------------------------------------------- //

            //generate Database Schema Model
            //(Detailed_DFD + Data Model ---> Schema)
            DDiagramToSchema2 topRule5 = new DDiagramToSchema2();
            Schema schemaModel = topRule5.translate(detailed_dataflowModel, dataModel);

            System.out.println(schemaModel.getName());

            ASTWriter writer5 = new
                ASTWriter(new File("Manage_Module_SchemaModel.xml"));
            writer5.usePackage("mde.dbs.model", "xmlns:dbs");
            writer5.usePackage("mde.model", "xmlns:mod");
            writer5.writeDocument(schemaModel);
            writer5.close();

            System.out.println("(9) Database Schema Model is Created"+
                "(DataFlow + Data Model --> Database Schema Model).");
            // --------------------------------------------- //

            //Generate Code Model
            //(Database Schema + DataFlow Model ---> Code Model)
            DDiagramToCDiagram2x topRule8 = new DDiagramToCDiagram2x();
            mde.code.model.Diagram  codeModel = topRule8
                .translate(schemaModel, detailed_dataflowModel);

            ASTWriter writer8 = new ASTWriter(new File("CodeModel.xml"));
            writer8.usePackage("mde.code.model", "xmlns:code");
            writer8.usePackage("mde.model", "xmlns:mod");
            writer8.writeDocument(codeModel);
            writer8.close();

            System.out.println("(10) Code Model is Created"+
                "(Database Scehma Model --> Code Model).");
            System.out.println("\nModel Transformation System is Completed ...");

            // Code generation
            DumpFileGenerator MySQLGenerator = new
                MySQLDumpFileGenerator(schemaModel);
            System.out.println("calling generate in mysql package generator");
            MySQLGenerator.generate();
            System.out.println("Finished generating MySQL Schema OK");

            for(GuiBoundary bound : GuiModel.getGuiBoundaries())
            {
                CodeFileGenerator JavaGenerator = new JavaCodeFileGenerator(bound);
                System.out.println("calling generate in java code file generator");
                JavaGenerator.generate();
                System.out.println("Finished generating Java Gui code");
            }
        }
}
```

## Task model: Module Management System

Listing A.2: *Manage_Module_taskModel.xml*

```
<task:Diagram xmlns:task="mde.task.model" id="0">
  <task:Boundary id="1" name="Module Management">
    <task:Task id="2" name="Manage Module"/>
    <task:Task id="3" name="Add Module"/>
    <task:Task id="4" name="Delete Module"/>
    <task:Task id="5" name="Modify Module"/>
    <task:Task id="6" name="See Description"/>
    <task:Actor id="7" name="Staff"/>
    <task:Actor id="8" name="Student"/>
    <task:Participation id="9">
      <task:Role id="10" name="staff">
        <task:Actor ref="7"/>
      </task:Role>
      <task:Role id="11" name="add">
        <task:Task ref="3"/>
```

```
        </task:Role>
      </task:Participation>
      <task:Participation id="12">
        <task:Role id="13" name="staff">
          <task:Actor ref="7"/>
        </task:Role>
        <task:Role id="14" name="modify">
          <task:Task ref="5"/>
        </task:Role>
      </task:Participation>
      <task:Participation id="15">
        <task:Role id="16" name="staff">
          <task:Actor ref="7"/>
        </task:Role>
        <task:Role id="17" name="delete">
          <task:Task ref="4"/>
        </task:Role>
      </task:Participation>
      <task:Participation id="18">
        <task:Role id="19" name="student">
          <task:Actor ref="8"/>
        </task:Role>
        <task:Role id="20" name="see">
          <task:Task ref="6"/>
        </task:Role>
      </task:Participation>
      <task:Participation id="21">
        <task:Role id="22" name="see">
          <task:Task ref="6"/>
        </task:Role>
        <task:Role id="23" name="student">
          <task:Actor ref="8"/>
        </task:Role>
      </task:Participation>
      <task:Composition id="24">
        <task:Role id="25" name="manage">
          <task:Task ref="2"/>
        </task:Role>
        <task:Role id="26" name="add">
          <task:Task ref="3"/>
        </task:Role>
        <task:Role id="27" name="delete">
          <task:Task ref="4"/>
        </task:Role>
        <task:Role id="28" name="modify">
          <task:Task ref="5"/>
        </task:Role>
      </task:Composition>
    </task:Boundary>
  </task:Diagram>
```

## Impact model: Module Management System

Listing A.3: *Manage_Module_impactModel.xml*

```
<imp:ImpDiagram xmlns:imp="mde.impact.model" id="0">
  <imp:ImpBoundary id="1" name="Module Management">
    <imp:ImpTask id="2" name="Manage Modules"/>
    <imp:ImpTask id="3" name="Add Module"/>
    <imp:ImpTask id="4" name="Delete Module"/>
    <imp:ImpTask id="5" name="Modify Module"/>
    <imp:ImpTask id="6" name="See Description"/>
    <imp:ImpObject id="7" name="Module"/>
    <imp:ImpReadFlow id="8">
      <imp:ImpRole id="9" name="module">
        <imp:ImpObject ref="7"/>
      </imp:ImpRole>
      <imp:ImpRole id="10" name="see">
        <imp:ImpTask ref="6"/>
      </imp:ImpRole>
    </imp:ImpReadFlow>
    <imp:ImpCreateFlow id="11">
      <imp:ImpRole id="12" name="add">
        <imp:ImpTask ref="3"/>
      </imp:ImpRole>
      <imp:ImpRole id="13" name="module">
        <imp:ImpObject ref="7"/>
      </imp:ImpRole>
    </imp:ImpCreateFlow>
    <imp:ImpDeleteFlow id="14">
      <imp:ImpRole id="15" name="delete">
        <imp:ImpTask ref="4"/>
      </imp:ImpRole>
      <imp:ImpRole id="16" name="module">
        <imp:ImpObject ref="7"/>
      </imp:ImpRole>
    </imp:ImpDeleteFlow>
    <imp:ImpUpdateFlow id="17">
```

```
      <imp:ImpRole id="18" name="modify">
        <imp:ImpTask ref="5"/>
      </imp:ImpRole>
      <imp:ImpRole id="19" name="module">
        <imp:ImpObject ref="7"/>
      </imp:ImpRole>
    </imp:ImpUpdateFlow>
  </imp:ImpBoundary>
</imp:ImpDiagram>
```

## Inofrmation model: Module Management System

Listing A.4: *Manage_Module_informationModel.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<inf:Diagram xmlns:inf="mde.information.model" id="0">
  <inf:Entity id="1" name="Module">
    <inf:Attribute id="2" name="code" identifier="true" size="0">
      <Type id="3" name="Integer"/>
    </inf:Attribute>
    <inf:Attribute id="4" name="title" size="0">
      <Type id="5" name="String"/>
    </inf:Attribute>
    <inf:Attribute id="6" name="credit" size="0">
      <Type id="7" name="Integer"/>
    </inf:Attribute>
    <inf:Attribute id="8" name="desc" size="0">
      <Type id="9" name="String"/>
    </inf:Attribute>
  </inf:Entity>
</inf:Diagram>
```

## DataFlow model (initial): Module Management System

Listing A.5: *Manage_Module_DataFlowModel.xml*

```
<dfd:DfDiagram xmlns:dfd="mde.dataflow.model" id="0">
  <dfd:DfBoundary id="1" name="Module Management">
    <dfd:DfTask id="2" name="Add Module"/>
    <dfd:DfTask id="3" name="Delete Module"/>
    <dfd:DfTask id="4" name="Modify Module"/>
    <dfd:DfTask id="5" name="See Description"/>
    <dfd:DfActor id="6" name="Staff"/>
    <dfd:DfActor id="7" name="Student"/>
    <dfd:DfObject id="8" name="Module"/>
    <dfd:DfOutputFlow id="9">
      <dfd:DfRole id="10" name="see">
        <dfd:DfTask id="11" name="See Description"/>
      </dfd:DfRole>
      <dfd:DfRole id="12" name="student">
        <dfd:DfTask id="13" name="Student"/>
      </dfd:DfRole>
    </dfd:DfOutputFlow>
    <dfd:DfInputFlow id="14">
      <dfd:DfRole id="15" name="staff">
        <dfd:DfTask id="16" name="Staff"/>
      </dfd:DfRole>
      <dfd:DfRole id="17" name="add">
        <dfd:DfTask id="18" name="Add Module"/>
      </dfd:DfRole>
    </dfd:DfInputFlow>
    <dfd:DfInputFlow id="19">
      <dfd:DfRole id="20" name="staff">
        <dfd:DfTask ref="16"/>
      </dfd:DfRole>
      <dfd:DfRole id="21" name="modify">
        <dfd:DfTask id="22" name="Modify Module"/>
      </dfd:DfRole>
    </dfd:DfInputFlow>
    <dfd:DfInputFlow id="23">
      <dfd:DfRole id="24" name="staff">
        <dfd:DfTask ref="16"/>
      </dfd:DfRole>
      <dfd:DfRole id="25" name="delete">
        <dfd:DfTask id="26" name="Delete Module"/>
      </dfd:DfRole>
    </dfd:DfInputFlow>
    <dfd:DfInputFlow id="27">
      <dfd:DfRole id="28" name="student">
        <dfd:DfTask ref="13"/>
      </dfd:DfRole>
```

```
        <dfd:DfRole id="29" name="see">
          <dfd:DfTask ref="11"/>
        </dfd:DfRole>
      </dfd:DfInputFlow>
      <dfd:DfReadFlow id="30">
        <dfd:DfRole id="31" name="module">
          <dfd:DfTask id="32" name="Module"/>
        </dfd:DfRole>
        <dfd:DfRole id="33" name="see">
          <dfd:DfTask id="34" name="See Description"/>
        </dfd:DfRole>
      </dfd:DfReadFlow>
      <dfd:DfCreateFlow id="35">
        <dfd:DfRole id="36" name="add">
          <dfd:DfTask id="37" name="Add Module"/>
        </dfd:DfRole>
        <dfd:DfRole id="38" name="module">
          <dfd:DfTask ref="32"/>
        </dfd:DfRole>
      </dfd:DfCreateFlow>
      <dfd:DfUpdateFlow id="39">
        <dfd:DfRole id="40" name="modify">
          <dfd:DfTask id="41" name="Modify Module"/>
        </dfd:DfRole>
        <dfd:DfRole id="42" name="module">
          <dfd:DfTask ref="32"/>
        </dfd:DfRole>
      </dfd:DfUpdateFlow>
      <dfd:DfDeleteFlow id="43">
        <dfd:DfRole id="44" name="delete">
          <dfd:DfTask id="45" name="Delete Module"/>
        </dfd:DfRole>
        <dfd:DfRole id="46" name="module">
          <dfd:DfTask ref="32"/>
        </dfd:DfRole>
      </dfd:DfDeleteFlow>
  </dfd:DfBoundary>
</dfd:DfDiagram>
```

## DataFlow model (detailed): Module Management System

Listing A.6: $Manage_{M}odule_{D}etailed_{D}ataFlowModel.xml$

```
<dfd:DfDiagram xmlns:dfd="mde.dataflow.model" id="0">
  <dfd:DfBoundary id="1" name="Add Module">
    <dfd:DfTask id="2" name="Input Code and more"/>
    <dfd:DfTask id="3" name="Create Module"/>
    <dfd:DfActor id="4" name="Staff"/>
    <dfd:DfObject id="5" name="Module"/>
    <dfd:DfInputFlow id="6">
      <dfd:DfRole id="7" name="input_actor">
        <dfd:DfActor ref="4"/>
      </dfd:DfRole>
      <dfd:DfRole id="8" name="input_task">
        <dfd:DfTask ref="2"/>
      </dfd:DfRole>
    </dfd:DfInputFlow>
    <dfd:DfReadFlow id="9">
      <dfd:DfRole id="10" name="input_task">
        <dfd:DfTask ref="2"/>
      </dfd:DfRole>
      <dfd:DfRole id="11" name="create_task">
        <dfd:DfTask ref="3"/>
      </dfd:DfRole>
    </dfd:DfReadFlow>
    <dfd:DfCreateFlow id="12">
      <dfd:DfRole id="13" name="create_task0">
        <dfd:DfTask ref="3"/>
      </dfd:DfRole>
      <dfd:DfRole id="14" name="create_object0">
        <dfd:DfObject ref="5"/>
      </dfd:DfRole>
    </dfd:DfCreateFlow>
  </dfd:DfBoundary>
  <dfd:DfBoundary id="15" name="Modify Module">
    <dfd:DfTask id="16" name="Input Code_Credit"/>
    <dfd:DfTask id="17" name="Read Module Code_Credit"/>
    <dfd:DfTask id="18" name="Write Module"/>
    <dfd:DfActor id="19" name="Staff"/>
    <dfd:DfObject id="20" name="Module"/>
    <dfd:DfInputFlow id="21">
      <dfd:DfRole id="22" name="input_actor">
        <dfd:DfActor ref="19"/>
      </dfd:DfRole>
      <dfd:DfRole id="23" name="input_task">
        <dfd:DfTask ref="16"/>
      </dfd:DfRole>
    </dfd:DfInputFlow>
```

```
    <dfd:DfReadFlow id="24" filter="[@code = code] @code, @credit" update="true">
      <dfd:DfRole id="25" name="update_object">
        <dfd:DfObject ref="20"/>
      </dfd:DfRole>
      <dfd:DfRole id="26" name="readu_task">
        <dfd:DfTask ref="17"/>
      </dfd:DfRole>
    </dfd:DfReadFlow>
    <dfd:DfReadFlow id="27">
      <dfd:DfRole id="28" name="read_task">
        <dfd:DfTask ref="16"/>
      </dfd:DfRole>
      <dfd:DfRole id="29" name="readu_task">
        <dfd:DfTask ref="17"/>
      </dfd:DfRole>
    </dfd:DfReadFlow>
    <dfd:DfReadFlow id="30" update="true" update="true">
      <dfd:DfRole id="31" name="readu_task">
        <dfd:DfTask ref="17"/>
      </dfd:DfRole>
      <dfd:DfRole id="32" name="write_task">
        <dfd:DfTask ref="18"/>
      </dfd:DfRole>
    </dfd:DfReadFlow>
    <dfd:DfWriteFlow id="33" filter="[@code = code] @code, @credit">
      <dfd:DfRole id="34" name="write_task">
        <dfd:DfTask ref="18"/>
      </dfd:DfRole>
      <dfd:DfRole id="35" name="update_object">
        <dfd:DfObject ref="20"/>
      </dfd:DfRole>
    </dfd:DfWriteFlow>
  </dfd:DfBoundary>
  <dfd:DfBoundary id="36" name="Delete Module">
    <dfd:DfTask id="37" name="Input Code"/>
    <dfd:DfTask id="38" name="Delete Module"/>
    <dfd:DfActor id="39" name="Staff"/>
    <dfd:DfObject id="40" name="Module"/>
    <dfd:DfInputFlow id="41">
      <dfd:DfRole id="42" name="input_actor">
        <dfd:DfActor ref="39"/>
      </dfd:DfRole>
      <dfd:DfRole id="43" name="input_task">
        <dfd:DfTask ref="37"/>
      </dfd:DfRole>
    </dfd:DfInputFlow>
    <dfd:DfReadFlow id="44">
      <dfd:DfRole id="45" name="input_task">
        <dfd:DfTask ref="37"/>
      </dfd:DfRole>
      <dfd:DfRole id="46" name="delete_task">
        <dfd:DfTask ref="38"/>
      </dfd:DfRole>
    </dfd:DfReadFlow>
    <dfd:DfDeleteFlow id="47" filter="[@code = code] @code">
      <dfd:DfRole id="48" name="delete_task">
        <dfd:DfTask ref="38"/>
      </dfd:DfRole>
      <dfd:DfRole id="49" name="delete_object">
        <dfd:DfObject ref="40"/>
      </dfd:DfRole>
    </dfd:DfDeleteFlow>
  </dfd:DfBoundary>
  <dfd:DfBoundary id="50" name="See Description">
    <dfd:DfTask id="51" name="Input Code"/>
    <dfd:DfTask id="52" name="Read Desc"/>
    <dfd:DfActor id="53" name="Student"/>
    <dfd:DfObject id="54" name="Module"/>
    <dfd:DfInputFlow id="55">
      <dfd:DfRole id="56" name="input_actor">
        <dfd:DfActor ref="53"/>
      </dfd:DfRole>
      <dfd:DfRole id="57" name="input_task">
        <dfd:DfTask ref="51"/>
      </dfd:DfRole>
    </dfd:DfInputFlow>
    <dfd:DfReadFlow id="58" filter="[@code = code] @desc">
      <dfd:DfRole id="59" name="read_object">
        <dfd:DfObject ref="54"/>
      </dfd:DfRole>
      <dfd:DfRole id="60" name="read_task">
        <dfd:DfTask ref="52"/>
      </dfd:DfRole>
    </dfd:DfReadFlow>
    <dfd:DfReadFlow id="61">
      <dfd:DfRole id="62" name="input_task">
        <dfd:DfTask ref="51"/>
      </dfd:DfRole>
      <dfd:DfRole id="63" name="read_task">
        <dfd:DfTask ref="52"/>
      </dfd:DfRole>
    </dfd:DfReadFlow>
  </dfd:DfBoundary>
</dfd:DfDiagram>
```

## Data Dependency model: Module Management System

Listing A.7: *Manage_Module_DataModel.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<data:DDiagram xmlns:data="mde.data.model" id="0">
  <data:DEntity id="1" name="Module">
    <data:DAttribute id="2" name="code" identifier="true" size="10">
      <mod:Type xmlns:mod="mde.model" id="3" name="Integer"/>
    </data:DAttribute>
    <data:DAttribute id="4" name="title" size="30">
      <mod:Type id="5" name="String"/>
    </data:DAttribute>
    <data:DAttribute id="6" name="credit" size="10">
      <mod:Type id="7" name="Integer"/>
    </data:DAttribute>
    <data:DAttribute id="8" name="desc" size="30">
      <mod:Type id="9" name="String"/>
    </data:DAttribute>
  </data:DEntity>
</data:DDiagram>
```

## Screen State model: Module Management System

Listing A.8: *Manage_Module_StateModel.xml*

```
<state:StDiagram xmlns:state="mde.state.model" id="0">
  <state:StBoundary id="1" name="Add Module">
    <state:State id="2" name="Start" priority="5"/>
    <state:State id="3" name="End" priority="0"/>
    <state:State id="4" name="input_code_title_credit_desc_Waiting" priority="4"/>
    <state:State id="5" name="input_code_title_credit_desc_Waiting_Error" priority="0"/>
    <state:State id="6" name="Create_Module_Ready" priority="2"/>
    <state:State id="7" name="Create_Module_Ready_Error" priority="0"/>
    <state:Transition id="8" exit="false" action="initialise">
      <state:StRole id="9" name="start">
        <state:State ref="2"/>
      </state:StRole>
      <state:StRole id="10" name="start_task">
        <state:State ref="4"/>
      </state:StRole>
    </state:Transition>
    <state:Transition id="11" exit="false" action="Exception">
      <state:StRole id="12" name="state">
        <state:State ref="4"/>
      </state:StRole>
      <state:StRole id="13" name="error">
        <state:State ref="5"/>
      </state:StRole>
    </state:Transition>
    <state:Transition id="14" exit="false" action="initialise">
      <state:StRole id="15" name="error">
        <state:State ref="5"/>
      </state:StRole>
      <state:StRole id="16" name="state">
        <state:State ref="4"/>
      </state:StRole>
    </state:Transition>
    <state:Transition id="17" label="create()" exit="false" action="Create">
      <state:StRole id="18" name="Create_Module_Ready">
        <state:State ref="6"/>
      </state:StRole>
      <state:StRole id="19" name="End">
        <state:State ref="3"/>
      </state:StRole>
    </state:Transition>
    <state:Transition id="20" label="input()" exit="false" action="Input">
      <state:StRole id="21" name="input_code_title_credit_desc_Waiting">
        <state:State ref="4"/>
      </state:StRole>
      <state:StRole id="22" name="Create_Module_Ready">
        <state:State ref="6"/>
      </state:StRole>
    </state:Transition>
    <state:Transition id="23" exit="true" action="exit">
      <state:StRole id="24" name="end_task">
        <state:State ref="6"/>
      </state:StRole>
      <state:StRole id="25" name="end">
        <state:State ref="3"/>
      </state:StRole>
    </state:Transition>
    <state:Transition id="26" exit="false" action="Exception">
      <state:StRole id="27" name="state">
        <state:State ref="6"/>
      </state:StRole>
```

```
        <state:StRole id="28" name="error">
          <state:State ref="7"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="29" exit="false" action="exit">
        <state:StRole id="30" name="error">
          <state:State ref="7"/>
        </state:StRole>
        <state:StRole id="31" name="state">
          <state:State ref="6"/>
        </state:StRole>
      </state:Transition>
    </state:StBoundary>
    <state:StBoundary id="32" name="Modify Module">
      <state:State id="33" name="Start" priority="5"/>
      <state:State id="34" name="End" priority="0"/>
      <state:State id="35" name="input_code_credit_Waiting" priority="4"/>
      <state:State id="36" name="input_code_credit_Waiting_Error" priority="0"/>
      <state:State id="37" name="Read_Module_Code_Credit_Ready" priority="3"/>
      <state:State id="38" name="Read_Module_Code_Credit_Ready_Error" priority="0"/>
      <state:State id="39" name="Write_Module_Ready" priority="2"/>
      <state:State id="40" name="Write_Module_Ready_Error" priority="0"/>
      <state:Transition id="41" exit="false" action="initialise">
        <state:StRole id="42" name="start">
          <state:State ref="33"/>
        </state:StRole>
        <state:StRole id="43" name="start_task">
          <state:State ref="35"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="44" exit="false" action="Exception">
        <state:StRole id="45" name="state">
          <state:State ref="35"/>
        </state:StRole>
        <state:StRole id="46" name="error">
          <state:State ref="36"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="47" exit="false" action="initialise">
        <state:StRole id="48" name="error">
          <state:State ref="36"/>
        </state:StRole>
        <state:StRole id="49" name="state">
          <state:State ref="35"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="50" label="input()" exit="false" action="Input">
        <state:StRole id="51" name="input_code_credit_Waiting">
          <state:State ref="35"/>
        </state:StRole>
        <state:StRole id="52" name="Read_Module_Code_Credit_Ready">
          <state:State ref="37"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="53" label="read()" exit="false" action="Read">
        <state:StRole id="54" name="Read_Module_Code_Credit_Ready">
          <state:State ref="37"/>
        </state:StRole>
        <state:StRole id="55" name="Write_Module_Ready">
          <state:State ref="39"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="56" exit="false" action="Exception">
        <state:StRole id="57" name="state">
          <state:State ref="37"/>
        </state:StRole>
        <state:StRole id="58" name="error">
          <state:State ref="38"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="59" exit="false" action="exit">
        <state:StRole id="60" name="error">
          <state:State ref="38"/>
        </state:StRole>
        <state:StRole id="61" name="state">
          <state:State ref="37"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="62" label="update()" exit="false" action="Write">
        <state:StRole id="63" name="Write_Module_Ready">
          <state:State ref="39"/>
        </state:StRole>
        <state:StRole id="64" name="End">
          <state:State ref="34"/>
        </state:StRole>
      </state:Transition>
      <state:Transition ref="53"/>
      <state:Transition id="65" exit="true" action="exit">
        <state:StRole id="66" name="end_task">
          <state:State ref="39"/>
        </state:StRole>
        <state:StRole id="67" name="end">
          <state:State ref="34"/>
        </state:StRole>
```

```
      </state:Transition>
      <state:Transition id="68" exit="false" action="Exception">
        <state:StRole id="69" name="state">
          <state:State ref="39"/>
        </state:StRole>
        <state:StRole id="70" name="error">
          <state:State ref="40"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="71" exit="false" action="exit">
        <state:StRole id="72" name="error">
          <state:State ref="40"/>
        </state:StRole>
        <state:StRole id="73" name="state">
          <state:State ref="39"/>
        </state:StRole>
      </state:Transition>
    </state:StBoundary>
    <state:StBoundary id="74" name="Delete Module">
      <state:State id="75" name="Start" priority="5"/>
      <state:State id="76" name="End" priority="0"/>
      <state:State id="77" name="input_code_Waiting" priority="4"/>
      <state:State id="78" name="input_code_Waiting_Error" priority="0"/>
      <state:State id="79" name="Delete_Module_Ready" priority="2"/>
      <state:State id="80" name="Delete_Module_Ready_Error" priority="0"/>
      <state:Transition id="81" exit="false" action="initialise">
        <state:StRole id="82" name="start">
          <state:State ref="75"/>
        </state:StRole>
        <state:StRole id="83" name="start_task">
          <state:State ref="77"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="84" exit="false" action="Exception">
        <state:StRole id="85" name="state">
          <state:State ref="77"/>
        </state:StRole>
        <state:StRole id="86" name="error">
          <state:State ref="78"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="87" exit="false" action="initialise">
        <state:StRole id="88" name="error">
          <state:State ref="78"/>
        </state:StRole>
        <state:StRole id="89" name="state">
          <state:State ref="77"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="90" label="input()" exit="false" action="Input">
        <state:StRole id="91" name="input_code_Waiting">
          <state:State ref="77"/>
        </state:StRole>
        <state:StRole id="92" name="Delete_Module_Ready">
          <state:State ref="79"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="93" label="delete()" exit="false" action="Delete">
        <state:StRole id="94" name="Delete_Module_Ready">
          <state:State ref="79"/>
        </state:StRole>
        <state:StRole id="95" name="End">
          <state:State ref="76"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="96" exit="true" action="exit">
        <state:StRole id="97" name="end_task">
          <state:State ref="79"/>
        </state:StRole>
        <state:StRole id="98" name="end">
          <state:State ref="76"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="99" exit="false" action="Exception">
        <state:StRole id="100" name="state">
          <state:State ref="79"/>
        </state:StRole>
        <state:StRole id="101" name="error">
          <state:State ref="80"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="102" exit="false" action="exit">
        <state:StRole id="103" name="error">
          <state:State ref="80"/>
        </state:StRole>
        <state:StRole id="104" name="state">
          <state:State ref="79"/>
        </state:StRole>
      </state:Transition>
    </state:StBoundary>
    <state:StBoundary id="105" name="See Description">
      <state:State id="106" name="Start" priority="5"/>
      <state:State id="107" name="End" priority="0"/>
      <state:State id="108" name="input_code_Waiting" priority="4"/>
```

```
      <state:State id="109" name="input_code_Waiting_Error" priority="0"/>
      <state:State id="110" name="Read_Desc_Ready" priority="3"/>
      <state:State id="111" name="Read_Desc_Ready_Error" priority="0"/>
      <state:Transition id="112" exit="false" action="initialise">
        <state:StRole id="113" name="start">
          <state:State ref="106"/>
        </state:StRole>
        <state:StRole id="114" name="start_task">
          <state:State ref="108"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="115" exit="false" action="Exception">
        <state:StRole id="116" name="state">
          <state:State ref="108"/>
        </state:StRole>
        <state:StRole id="117" name="error">
          <state:State ref="109"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="118" exit="false" action="initialise">
        <state:StRole id="119" name="error">
          <state:State ref="109"/>
        </state:StRole>
        <state:StRole id="120" name="state">
          <state:State ref="108"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="121" label="input()" exit="false" action="Input">
        <state:StRole id="122" name="input_code_Waiting">
          <state:State ref="108"/>
        </state:StRole>
        <state:StRole id="123" name="Read_Desc_Ready">
          <state:State ref="110"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="124" label="read()" exit="false" action="Read">
        <state:StRole id="125" name="Read_Desc_Ready">
          <state:State ref="110"/>
        </state:StRole>
        <state:StRole id="126" name="End">
          <state:State ref="107"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="127" exit="true" action="exit">
        <state:StRole id="128" name="end_task">
          <state:State ref="110"/>
        </state:StRole>
        <state:StRole id="129" name="end">
          <state:State ref="107"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="130" exit="false" action="Exception">
        <state:StRole id="131" name="state">
          <state:State ref="110"/>
        </state:StRole>
        <state:StRole id="132" name="error">
          <state:State ref="111"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="133" exit="false" action="exit">
        <state:StRole id="134" name="error">
          <state:State ref="111"/>
        </state:StRole>
        <state:StRole id="135" name="state">
          <state:State ref="110"/>
        </state:StRole>
      </state:Transition>
    </state:StBoundary>
    <state:StBoundary id="136" name="Manage Module_Menu">
      <state:State id="137" name="Start" priority="5"/>
      <state:State id="138" name="End" priority="0"/>
      <state:State id="139" name="Manage_Module_Main_Menu_Waiting" priority="5"/>
      <state:Transition id="140" exit="false" action="initialise"
          boundary="Add Module">
        <state:StRole id="141" name="Manage_Module_Main_Menu_Waiting">
          <state:State ref="139"/>
        </state:StRole>
        <state:StRole id="142" name="main_task_Add Module">
          <state:State ref="4"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="143" exit="false" action="initialise"
          boundary="Delete Module">
        <state:StRole id="144" name="Manage_Module_Main_Menu_Waiting">
          <state:State ref="139"/>
        </state:StRole>
        <state:StRole id="145" name="main_task_Delete Module">
          <state:State ref="77"/>
        </state:StRole>
      </state:Transition>
      <state:Transition id="146" exit="false" action="initialise"
          boundary="Modify Module">
        <state:StRole id="147" name="Manage_Module_Main_Menu_Waiting">
          <state:State ref="139"/>
```

```
      </state:StRole>
      <state:StRole id="148" name="main_task_Modify Module">
        <state:State ref="35"/>
      </state:StRole>
    </state:Transition>
  </state:StBoundary>
</state:StDiagram>
```

## GUI model: Module Management System

Listing A.9: *Manage_Module_GuiModel.xml*

```
<gui:GuiDiagram xmlns:gui="mde.gui.model" id="0">
  <gui:GuiBoundary id="1" name="Add Module">
    <gui:Window id="2" name="Input_code_title_credit_desc_Waiting" order="4">
      <gui:Textfield id="3" name="code"/>
      <gui:Textfield id="4" name="title"/>
      <gui:Textfield id="5" name="credit"/>
      <gui:Textfield id="6" name="desc"/>
      <gui:Button id="7" name="Exception" event="Exception" exit="false"/>
      <gui:Button id="8" name="Input" event="Input" exit="false"/>
    </gui:Window>
    <gui:Window id="9" name="Input_code_title_credit_desc_Waiting_Error" order="0" error="true">
      <gui:Label id="10" name="input_code_title_credit_desc_Waiting_Error_warning"
        text="Null value not accepted"/>
      <gui:Button id="11" name="initialise" event="initialise" exit="false"/>
    </gui:Window>
    <gui:Window id="12" name="Create_Module_Ready" order="2">
      <gui:Label id="13" name="code" text="code"/>
      <gui:Label id="14" name="title" text="title"/>
      <gui:Label id="15" name="credit" text="credit"/>
      <gui:Label id="16" name="desc" text="desc"/>
      <gui:Button id="17" name="Create" event="Create" exit="false"/>
      <gui:Button id="18" name="exit" event="exit" exit="true"/>
      <gui:Button id="19" name="Exception" event="Exception" exit="false"/>
    </gui:Window>
    <gui:Window id="20" name="Create_Module_Ready_Error" order="0" error="true">
      <gui:Label id="21" name="Create_Module_Ready_Error_warning"
         text="Connection to the Data source is fail"/>
      <gui:Button id="22" name="exit" event="exit" exit="false"/>
    </gui:Window>
  </gui:GuiBoundary>
  <gui:GuiBoundary id="23" name="Modify Module">
    <gui:Window id="24" name="Input_code_credit_Waiting" order="4">
      <gui:Textfield id="25" name="code"/>
      <gui:Textfield id="26" name="credit"/>
      <gui:Button id="27" name="Exception" event="Exception" exit="false"/>
      <gui:Button id="28" name="Input" event="Input" exit="false"/>
    </gui:Window>
    <gui:Window id="29" name="Input_code_credit_Waiting_Error" order="0" error="true">
      <gui:Label id="30" name="input_code_credit_Waiting_Error_warning"
         text="Null value not accepted"/>
      <gui:Button id="31" name="initialise" event="initialise" exit="false"/>
    </gui:Window>
    <gui:Window id="32" name="Read_Module_Code_Credit_Ready" order="3">
      <gui:Label ref="13"/>
      <gui:Label ref="15"/>
      <gui:Button id="33" name="Read" event="Read" exit="false"/>
      <gui:Button id="34" name="Exception" event="Exception" exit="false"/>
    </gui:Window>
    <gui:Window id="35" name="Read_Module_Code_Credit_Ready_Error" order="0" error="true">
      <gui:Label id="36" name="Read_Module_Code_Credit_Ready_Error_warning"
         text="Connection to the Data source is fail"/>
      <gui:Button id="37" name="exit" event="exit" exit="false"/>
    </gui:Window>
    <gui:Window id="38" name="Write_Module_Ready" order="2">
      <gui:Label ref="13"/>
      <gui:Label ref="15"/>
      <gui:Button id="39" name="Write" event="Write" exit="false"/>
      <gui:Button id="40" name="exit" event="exit" exit="true"/>
      <gui:Button id="41" name="Exception" event="Exception" exit="false"/>
    </gui:Window>
    <gui:Window id="42" name="Write_Module_Ready_Error" order="0" error="true">
      <gui:Label id="43" name="Write_Module_Ready_Error_warning"
         text="Connection to the Data source is fail"/>
      <gui:Button id="44" name="exit" event="exit" exit="false"/>
    </gui:Window>
  </gui:GuiBoundary>
  <gui:GuiBoundary id="45" name="Delete Module">
    <gui:Window id="46" name="Input_code_Waiting" order="4">
      <gui:Textfield id="47" name="code"/>
      <gui:Button id="48" name="Exception" event="Exception" exit="false"/>
      <gui:Button id="49" name="Input" event="Input" exit="false"/>
    </gui:Window>
    <gui:Window id="50" name="Input_code_Waiting_Error" order="0" error="true">
      <gui:Label id="51" name="input_code_Waiting_Error_warning"
         text="Null value not accepted"/>
      <gui:Button id="52" name="initialise" event="initialise" exit="false"/>
```

```xml
      </gui:Window>
      <gui:Window id="53" name="Delete_Module_Ready" order="2">
        <gui:Label ref="13"/>
        <gui:Button id="54" name="Delete" event="Delete" exit="false"/>
        <gui:Button id="55" name="exit" event="exit" exit="true"/>
        <gui:Button id="56" name="Exception" event="Exception" exit="false"/>
      </gui:Window>
      <gui:Window id="57" name="Delete_Module_Ready_Error" order="0" error="true">
        <gui:Label id="58" name="Delete_Module_Ready_Error_warning"
           text="Connection to the Data source is fail"/>
        <gui:Button id="59" name="exit" event="exit" exit="false"/>
      </gui:Window>
    </gui:GuiBoundary>
    <gui:GuiBoundary id="60" name="See Description">
      <gui:Window id="61" name="Input_code_Waiting" order="4">
        <gui:Textfield id="62" name="code"/>
        <gui:Button id="63" name="Exception" event="Exception" exit="false"/>
        <gui:Button id="64" name="Input" event="Input" exit="false"/>
      </gui:Window>
      <gui:Window id="65" name="Input_code_Waiting_Error" order="0" error="true">
        <gui:Label id="66" name="input_code_Waiting_Error_warning"
           text="Null value not accepted"/>
        <gui:Button id="67" name="initialise" event="initialise" exit="false"/>
      </gui:Window>
      <gui:Window id="68" name="Read_Desc_Ready" order="3">
        <gui:Label ref="13"/>
        <gui:Button id="69" name="Read" event="Read" exit="false"/>
        <gui:Button id="70" name="exit" event="exit" exit="true"/>
        <gui:Button id="71" name="Exception" event="Exception" exit="false"/>
      </gui:Window>
      <gui:Window id="72" name="Read_Desc_Ready_Error" order="0" error="true">
        <gui:Label id="73" name="Read_Desc_Ready_Error_warning"
           text="Connection to the Data source is fail"/>
        <gui:Button id="74" name="exit" event="exit" exit="false"/>
      </gui:Window>
    </gui:GuiBoundary>
    <gui:GuiBoundary id="75" name="Manage Module_Menu">
      <gui:Window id="76" name="Manage_Module_Main_Menu_Waiting" order="5" menu="true">
        <gui:Label id="77" name="Main Menu" text="Main Menu"/>
        <gui:Button id="78" name="Add Module" event="initialise" exit="false"/>
        <gui:Button id="79" name="Delete Module" event="initialise" exit="false"/>
        <gui:Button id="80" name="Modify Module" event="initialise" exit="false"/>
      </gui:Window>
    </gui:GuiBoundary>
  </gui:GuiDiagram>
```

## Database and Query model: Module Management System

Listing A.10: *Manage_Module_SchemaModel.xml*

```xml
<?xml version="1.0" encoding="UTF-8"?>
<dbs:Schema xmlns:dbs="mde.dbs.model" id="0" name="database">
  <dbs:Table id="1" name="Module">
    <dbs:Column id="2" name="code" size="10">
      <mod:Type xmlns:mod="mde.model" id="3" name="INTEGER"/>
    </dbs:Column>
    <dbs:Column id="4" name="title" size="30">
      <mod:Type id="5" name="VARCHAR"/>
    </dbs:Column>
    <dbs:Column id="6" name="credit" size="10">
      <mod:Type id="7" name="INTEGER"/>
    </dbs:Column>
    <dbs:Column id="8" name="desc" size="30">
      <mod:Type id="9" name="VARCHAR"/>
    </dbs:Column>
    <dbs:PrimaryKey id="10">
      <dbs:Column ref="2"/>
    </dbs:PrimaryKey>
  </dbs:Table>
  <dbs:Procedure id="11" name="readCode_Credit" table="Module">
    <dbs:Argument id="12" type="Integer" name="code" in="true"/>
    <dbs:Argument id="13" type="Integer" name="credit" out="true"/>
    <dbs:Query id="14" name="SELECT_Code,_Credit">
      <dbs:Project id="15" name="proj">
        <dbs:Column id="16" name="credit" size="10" prefix="Module">
          <mod:Type id="17" name="Integer"/>
        </dbs:Column>
      </dbs:Project>
      <dbs:Relation id="18" name="Module">
        <dbs:Filter id="19" name="filter1">
          <dbs:Relation id="20" name="Module"/>
          <dbs:Operator id="21" type="boolean" symbol="equals">
            <dbs:Column id="22" name="code" size="10">
              <mod:Type id="23" name="INTEGER"/>
            </dbs:Column>
            <dbs:Variable id="24" type="INTEGER" name="code"/>
          </dbs:Operator>
        </dbs:Filter>
```

```
        </dbs:Relation>
      </dbs:Query>
    </dbs:Procedure>
    <dbs:Procedure id="25" name="readDesc" table="Module">
      <dbs:Argument id="26" type="String" name="desc" out="true"/>
      <dbs:Argument id="27" type="Integer" name="code" in="true"/>
      <dbs:Query id="28" name="SELECT_Desc">
        <dbs:Project id="29" name="proj">
          <dbs:Column id="30" name="desc" size="30" prefix="Module">
            <mod:Type id="31" name="String"/>
          </dbs:Column>
        </dbs:Project>
        <dbs:Relation id="32" name="Module">
          <dbs:Filter id="33" name="filter1">
            <dbs:Relation id="34" name="Module"/>
            <dbs:Operator id="35" type="boolean" symbol="equals">
              <dbs:Column id="36" name="code" size="10">
                <mod:Type ref="23"/>
              </dbs:Column>
              <dbs:Variable id="37" type="INTEGER" name="code"/>
            </dbs:Operator>
          </dbs:Filter>
        </dbs:Relation>
      </dbs:Query>
    </dbs:Procedure>
    <dbs:Procedure id="38" name="createModule" table="Module">
      <dbs:Argument id="39" type="Integer" name="code" in="true"/>
      <dbs:Argument id="40" type="String" name="title" in="true"/>
      <dbs:Argument id="41" type="Integer" name="credit" in="true"/>
      <dbs:Argument id="42" type="String" name="desc" in="true"/>
      <dbs:Create id="43" name="INSERT_Module">
        <dbs:Operator id="44" type="boolean" symbol="Assign">
          <dbs:Column id="45" name="code" size="10">
            <mod:Type id="46" name="INTEGER"/>
          </dbs:Column>
          <dbs:Variable id="47" type="INTEGER" name="code"/>
        </dbs:Operator>
        <dbs:Operator id="48" type="boolean" symbol="Assign">
          <dbs:Column id="49" name="title" size="30">
            <mod:Type id="50" name="VARCHAR"/>
          </dbs:Column>
          <dbs:Variable id="51" type="VARCHAR" name="title"/>
        </dbs:Operator>
        <dbs:Operator id="52" type="boolean" symbol="Assign">
          <dbs:Column id="53" name="credit" size="10">
            <mod:Type id="54" name="INTEGER"/>
          </dbs:Column>
          <dbs:Variable id="55" type="INTEGER" name="credit"/>
        </dbs:Operator>
        <dbs:Operator id="56" type="boolean" symbol="Assign">
          <dbs:Column id="57" name="desc" size="30">
            <mod:Type id="58" name="VARCHAR"/>
          </dbs:Column>
          <dbs:Variable id="59" type="VARCHAR" name="desc"/>
        </dbs:Operator>
      </dbs:Create>
    </dbs:Procedure>
    <dbs:Procedure id="60" name="deleteModule" table="Module">
      <dbs:Argument id="61" type="Integer" name="code" in="true"/>
      <dbs:Delete id="62" name="DELETE_Code">
        <dbs:Relation id="63" name="Delete Module">
          <dbs:Filter id="64">
            <dbs:Relation id="65" name="Module"/>
            <dbs:Operator id="66" type="boolean" symbol="equals">
              <dbs:Column id="67" name="code" size="10">
                <mod:Type id="68" name="INTEGER"/>
              </dbs:Column>
              <dbs:Variable id="69" type="INTEGER" name="code"/>
            </dbs:Operator>
          </dbs:Filter>
        </dbs:Relation>
      </dbs:Delete>
    </dbs:Procedure>
    <dbs:Procedure id="70" name="writeModule" table="Module">
      <dbs:Argument id="71" type="INTEGER" name="code" in="true"/>
      <dbs:Argument id="72" type="INTEGER" name="credit" inout="true"/>
      <dbs:Query id="73" name="SELECT_Code, _Credit">
        <dbs:Project id="74">
          <dbs:Column id="75" name="credit" size="10" prefix="Module">
            <mod:Type id="76" name="INTEGER"/>
          </dbs:Column>
          <dbs:Relation id="77" name="Module"/>
        </dbs:Project>
        <dbs:Relation id="78" name="Module">
          <dbs:Filter id="79">
            <dbs:Relation id="80" name="Module"/>
            <dbs:Operator id="81" type="boolean" symbol="equals">
              <dbs:Column id="82" name="code" size="10">
                <mod:Type id="83" name="INTEGER"/>
              </dbs:Column>
              <dbs:Variable id="84" type="INTEGER" name="code"/>
            </dbs:Operator>
          </dbs:Filter>
        </dbs:Relation>
```

```
    </dbs:Query>
    <dbs:Update id="85">
      <dbs:Operator id="86" type="boolean" symbol="assign">
        <dbs:Column id="87" name="credit" size="10">
          <mod:Type id="88" name="INTEGER"/>
        </dbs:Column>
        <dbs:Variable id="89" type="INTEGER" name="credit"/>
      </dbs:Operator>
    </dbs:Update>
  </dbs:Procedure>
</dbs:Schema>
```

## OOP Code model: Module Management System

Listing A.11: *Manage_Module_CodeModel.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<code:Diagram xmlns:code="mde.code.model" id="0">
  <code:Clazz id="1" name="Module" visible="public">
    <code:Attribute id="2" name="code" visible="private">
      <mod:Type xmlns:mod="mde.model" id="3" name="int"/>
    </code:Attribute>
    <code:Attribute id="4" name="title" visible="private">
      <mod:Type id="5" name="String"/>
    </code:Attribute>
    <code:Attribute id="6" name="credit" visible="private">
      <mod:Type id="7" name="int"/>
    </code:Attribute>
    <code:Attribute id="8" name="desc" visible="private">
      <mod:Type id="9" name="String"/>
    </code:Attribute>
    <code:Method id="10" name="readCode_Credit" visible="public">
      <mod:Type id="11" name="void"/>
      <code:Argument id="12" type="int" name="code"/>
      <code:Call id="13" name="readCode_Credit"/>
    </code:Method>
    <code:Method id="14" name="readDesc" visible="public">
      <mod:Type id="15" name="void"/>
      <code:Argument id="16" type="int" name="code"/>
      <code:Call id="17" name="readDesc"/>
    </code:Method>
    <code:Method id="18" name="createModule" visible="public">
      <mod:Type id="19" name="void"/>
      <code:Argument id="20" type="int" name="code"/>
      <code:Argument id="21" type="String" name="title"/>
      <code:Argument id="22" type="int" name="credit"/>
      <code:Argument id="23" type="String" name="desc"/>
      <code:Call id="24" name="createModule"/>
    </code:Method>
    <code:Method id="25" name="deleteModule" visible="public">
      <mod:Type id="26" name="void"/>
      <code:Argument id="27" type="int" name="code"/>
      <code:Call id="28" name="deleteModule"/>
    </code:Method>
    <code:Method id="29" name="writeModule" visible="public">
      <mod:Type id="30" name="void"/>
      <code:Argument id="31" name="code"/>
      <code:Call id="32" name="writeModule"/>
    </code:Method>
    <code:Constructor id="33" name="Module" visible="public">
      <code:Operator id="34" type="boolean" symbol="assign">
        <code:Identifier id="35" type="int" name="code" scope="Object"/>
        <code:Identifier id="36" type="int" name="cod"/>
      </code:Operator>
      <code:Operator id="37" type="boolean" symbol="assign">
        <code:Identifier id="38" type="String" name="title" scope="Object"/>
        <code:Identifier id="39" type="String" name="tit"/>
      </code:Operator>
      <code:Operator id="40" type="boolean" symbol="assign">
        <code:Identifier id="41" type="int" name="credit" scope="Object"/>
        <code:Identifier id="42" type="int" name="cred"/>
      </code:Operator>
      <code:Operator id="43" type="boolean" symbol="assign">
        <code:Identifier id="44" type="String" name="desc" scope="Object"/>
        <code:Identifier id="45" type="String" name="des"/>
      </code:Operator>
    </code:Constructor>
  </code:Clazz>
  <code:Clazz id="46" name="Add Module" visible="public">
    <code:Attribute id="47" name="Module" visible="private">
      <mod:Type id="48" name="Module"/>
    </code:Attribute>
    <code:Attribute id="49" name="Staff" visible="private">
      <mod:Type id="50" name="Staff"/>
    </code:Attribute>
    <code:Method id="51" name="Run" visible="public" staticm="true">
      <mod:Type id="52" name="void"/>
    </code:Method>
```

```
      <code:Constructor id="53" name="Add Module"/>
    </code:Clazz>
    <code:Clazz id="54" name="Modify Module" visible="public">
      <code:Attribute id="55" name="Module" visible="private">
        <mod:Type id="56" name="Module"/>
      </code:Attribute>
      <code:Attribute id="57" name="Staff" visible="private">
        <mod:Type id="58" name="Staff"/>
      </code:Attribute>
      <code:Method id="59" name="Run" visible="public" staticm="true">
        <mod:Type id="60" name="void"/>
      </code:Method>
      <code:Constructor id="61" name="Modify Module"/>
    </code:Clazz>
    <code:Clazz id="62" name="Delete Module" visible="public">
      <code:Attribute id="63" name="Module" visible="private">
        <mod:Type id="64" name="Module"/>
      </code:Attribute>
      <code:Attribute id="65" name="Staff" visible="private">
        <mod:Type id="66" name="Staff"/>
      </code:Attribute>
      <code:Method id="67" name="Run" visible="public" staticm="true">
        <mod:Type id="68" name="void"/>
      </code:Method>
      <code:Constructor id="69" name="Delete Module"/>
    </code:Clazz>
    <code:Clazz id="70" name="See Description" visible="public">
      <code:Attribute id="71" name="Module" visible="private">
        <mod:Type id="72" name="Module"/>
      </code:Attribute>
      <code:Attribute id="73" name="Student" visible="private">
        <mod:Type id="74" name="Student"/>
      </code:Attribute>
      <code:Method id="75" name="Run" visible="public" staticm="true">
        <mod:Type id="76" name="void"/>
      </code:Method>
      <code:Constructor id="77" name="See Description"/>
    </code:Clazz>
</code:Diagram>
```

## MySQL Script: Module Management System

Listing A.12: *database_MySQL.sql*

```
-- Database Creation

CREATE DATABASE sysDatabase;
USE sysDatabase;

-- Structure for table 'Module'

CREATE TABLE Module (
  code INT(10) NOT NULL,
  title VARCHAR(30),
  credit INT(10),
  desc VARCHAR(30),
  PRIMARY KEY(code));

-- Structure of Procedure 'readCode_Credit'

DELIMITER //

CREATE PROCEDURE readCode_Credit(IN code INT(10), OUT credit INT(10))
  BEGIN
    SELECT Module.credit INTO credit FROM Module
    WHERE
     Module.code = code;
  END //

DELIMITER //

-- Structure of Procedure 'readDesc'

DELIMITER //

CREATE PROCEDURE readDesc(IN code INT(10), OUT desc VARCHAR(30))
  BEGIN
    SELECT Module.desc INTO desc FROM Module
    WHERE
     Module.code = code;
  END //

DELIMITER //

-- Structure of Procedure 'createModule'

DELIMITER //

```

```
CREATE PROCEDURE createModule(IN code INT(10), IN title VARCHAR(30), IN credit INT(10), IN desc VARCHAR(30))
  BEGIN
    INSERT INTO Module VALUES (code, title, credit, desc);
  END //

DELIMITER //

-- Structure of Procedure 'deleteModule'

DELIMITER //

CREATE PROCEDURE deleteModule(IN code INT(10))
  BEGIN
    DELETE FROM Module
    WHERE
     code = code;
  END //

DELIMITER //

-- Structure of Procedure 'writeModule'

DELIMITER //

CREATE PROCEDURE writeModule(IN code INT(10), INOUT credit INT(10))
  BEGIN
    UPDATE Module
    SET
        Module.credit = credit
    WHERE
        Module.code = code;
  END //

DELIMITER //
```

## Java Swing Source Code: Module Management System

## Class: Manage_Module_Main_Menu_Waiting

Listing A.13: *Manage_Module_Main_Menu_Waiting.java*

```java
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Manage_Module_Main_Menu_Waiting extends JFrame {
    public Manage_Module_Main_Menu_Waiting(String winTitle) {

        super(winTitle);

        // Create Swing Components
        JPanel main_menu_panel = new JPanel();
        final JLabel main_menuLab = new JLabel();
        main_menu_panel.add(main_menuLab);
        JPanel message_panel = new JPanel();
        final JLabel messageLab = new JLabel("Main Menu");
        message_panel.add(messageLab);
        JPanel add_module_panel = new JPanel();
        JButton add_module = new JButton("Add Module");
        add_module_panel.add(add_module);

         // Add Action to the button
        add_module.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                    Input_code_title_credit_desc_Waiting nextWindow = new
                        Input_code_title_credit_desc_Waiting("Input_code_title_
                            credit_desc_Waiting");
                    nextWindow.setSize(400, 300);
                    nextWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    nextWindow.setLocationRelativeTo(null);
                    nextWindow.setVisible(true);
                    dispose();
            }
        });

        JPanel delete_module_panel = new JPanel();
        JButton delete_module = new JButton("Delete Module");
        delete_module_panel.add(delete_module);
```

```java

        // Add Action to the button
        delete_module.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                    Input_code_Waiting nextWindow = new
                        Input_code_Waiting("Input_code_Waiting");
                    nextWindow.setSize(400, 300);
                    nextWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    nextWindow.setLocationRelativeTo(null);
                    nextWindow.setVisible(true);
                    dispose();
            }
        });

        JPanel modify_module_panel = new JPanel();
        JButton modify_module = new JButton("Modify Module");
        modify_module_panel.add(modify_module);

        // Add Action to the button
        modify_module.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                    Input_code_credit_Waiting nextWindow = new
                        Input_code_credit_Waiting("Input_code_
                            credit_Waiting");
                    nextWindow.setSize(400, 300);
                    nextWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    nextWindow.setLocationRelativeTo(null);
                    nextWindow.setVisible(true);
                    dispose();
            }
        });
        // Set Layout Manager
        setLayout(new GridLayout(5, 0));

        // Add Panels to the Window
        this.add(main_menu_panel);
        this.add(message_panel);
        this.add(add_module_panel);
        this.add(delete_module_panel);
        this.add(modify_module_panel);
        }
    } // END OF CLASS
```

## Input_code_title_credit_desc_Waiting

Listing A.14: *Input_code_title_credit_desc_Waiting.java*

```java
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Input_code_title_credit_desc_Waiting extends JFrame {
    public Input_code_title_credit_desc_Waiting(String winTitle) {

        super(winTitle);

        // Create Swing Components
        JPanel message_panel = new JPanel();
        final JLabel messageLab = new JLabel("Input your Data");
        message_panel.add(messageLab);
        JPanel code_panel = new JPanel();
        JLabel code_label = new JLabel("code");
        final JTextField codeTxt = new JTextField(30);
        code_panel.add(code_label);
        code_panel.add(codeTxt);
        JPanel title_panel = new JPanel();
        JLabel title_label = new JLabel("title");
        final JTextField titleTxt = new JTextField(30);
        title_panel.add(title_label);
        title_panel.add(titleTxt);
        JPanel credit_panel = new JPanel();
        JLabel credit_label = new JLabel("credit");
        final JTextField creditTxt = new JTextField(30);
        credit_panel.add(credit_label);
        credit_panel.add(creditTxt);
        JPanel desc_panel = new JPanel();
        JLabel desc_label = new JLabel("desc");
        final JTextField descTxt = new JTextField(30);
        desc_panel.add(desc_label);
        desc_panel.add(descTxt);
```

```
        JPanel input_panel = new JPanel();
        JButton input = new JButton("Input");
        input_panel.add(input);

         // Add Action to the button
        input.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                    String code = codeTxt.getText();
                    String title = titleTxt.getText();
                    String credit = creditTxt.getText();
                    String desc = descTxt.getText();

                if((codeTxt.getText().isEmpty()) || (titleTxt.getText().isEmpty())
                    || (creditTxt.getText().isEmpty()) || (descTxt.getText().isEmpty())) {
                    Input_code_title_credit_desc_Waiting_Error nextErrWindow = new
                         Input_code_title_credit_desc_Waiting_Error("Input_code_
                             title_credit_desc_Waiting_Error ",
                                 "Invalid Input. Please Try Again" );
                    nextErrWindow.setSize(400, 300);
                    nextErrWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    nextErrWindow.setLocationRelativeTo(null);
                    nextErrWindow.setVisible(true);
                    dispose();
                    }

                else {
                    Create_Module_Ready nextWindow = new
                        Create_Module_Ready("Create_Module_Ready", code, title,
                            credit, desc);
                    nextWindow.setSize(400, 300);
                    nextWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    nextWindow.setLocationRelativeTo(null);
                    nextWindow.setVisible(true);
                    dispose();
                    }
            }
        });

        // Set Layout Manager
        setLayout(new GridLayout(6, 0));

        // Add Panels to the Window
        this.add(message_panel);
        this.add(code_panel);
        this.add(title_panel);
        this.add(credit_panel);
        this.add(desc_panel);
        this.add(input_panel);
        }
    } // END OF CLASS
```

## Input_code_title_credit_desc_Waiting_Error

Listing A.15: *Input_code_title_credit_desc_Waiting_Error.java*

```
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Input_code_title_credit_desc_Waiting_Error extends JFrame {
    public Input_code_title_credit_desc_Waiting_Error(String winTitle, String err) {

        super(winTitle);

        // Create Swing Components
        JPanel input_code_title_credit_desc_waiting_error_
            warning_panel = new JPanel();
        final JLabel input_code_title_credit_desc_waiting_error_
            warningLab = new JLabel();
        input_code_title_credit_desc_waiting_error_warning_panel
            .add(input_code_title_credit_desc_waiting_error_warningLab);

        JPanel message_panel = new JPanel();
        final JLabel messageLab = new JLabel(err);
        message_panel.add(messageLab);
        JPanel initialise_panel = new JPanel();
        JButton initialise = new JButton("initialise");
        initialise_panel.add(initialise);

         // Add Action to the button
```

```java
        initialise.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                    Input_code_title_credit_desc_Waiting backWindow = new
                        Input_code_title_credit_desc_
                            Waiting("Input_code_title_credit_desc_Waiting");
                    backWindow.setSize(400, 300);
                    backWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    backWindow.setLocationRelativeTo(null);
                    backWindow.setVisible(true);
                    dispose();
            }
        });

        // Set Layout Manager
        setLayout(new GridLayout(3, 0));

        // Add Panels to the Window
        this.add(input_code_title_credit_desc_waiting_error_warning_panel);
        this.add(message_panel);
        this.add(initialise_panel);
        }
    } // END OF CLASS
```

## Input_code_credit_Waiting

Listing A.16: *Input_code_credit_Waiting.java*

```java
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Input_code_credit_Waiting extends JFrame {
    public Input_code_credit_Waiting(String winTitle) {

        super(winTitle);

        // Create Swing Components
        JPanel message_panel = new JPanel();
        final JLabel messageLab = new JLabel("Input your Data");
        message_panel.add(messageLab);
        JPanel code_panel = new JPanel();
        JLabel code_label = new JLabel("code");
        final JTextField codeTxt = new JTextField(30);
        code_panel.add(code_label);
        code_panel.add(codeTxt);
        JPanel credit_panel = new JPanel();
        JLabel credit_label = new JLabel("credit");
        final JTextField creditTxt = new JTextField(30);
        credit_panel.add(credit_label);
        credit_panel.add(creditTxt);
        JPanel input_panel = new JPanel();
        JButton input = new JButton("Input");
        input_panel.add(input);

         // Add Action to the button
        input.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                    String code = codeTxt.getText();
                    String credit = creditTxt.getText();

                if((codeTxt.getText().isEmpty()) || (creditTxt
                    .getText().isEmpty())) {
                    Input_code_credit_Waiting_Error nextErrWindow = new
                        Input_code_credit_Waiting_Error("Input_code_credit_
                        Waiting_Error ", "Invalid Input. Please Try Again" );
                    nextErrWindow.setSize(400, 300);
                    nextErrWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    nextErrWindow.setLocationRelativeTo(null);
                    nextErrWindow.setVisible(true);
                    dispose();
                    }

                else {
                    Read_Module_Code_Credit_Ready nextWindow = new
                        Read_Module_Code_Credit_Ready("Read_Module_Code_
                        Credit_Ready", code, credit);
                    nextWindow.setSize(400, 300);
                    nextWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    nextWindow.setLocationRelativeTo(null);
                    nextWindow.setVisible(true);
```

```
                        dispose();
                        }
                }
            });

            // Set Layout Manager
            setLayout(new GridLayout(4, 0));

            // Add Panels to the Window
            this.add(message_panel);
            this.add(code_panel);
            this.add(credit_panel);
            this.add(input_panel);
            }
        } // END OF CLASS
```

## Input_code_credit_Waiting_Error

Listing A.17: *Input_code_credit_Waiting_Error.java*

```
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Input_code_credit_Waiting_Error extends JFrame {
    public Input_code_credit_Waiting_Error(String winTitle, String err) {

        super(winTitle);

        // Create Swing Components
        JPanel input_code_credit_waiting_error_warning_panel = new JPanel();
        final JLabel input_code_credit_waiting_error_warningLab = new JLabel();
        input_code_credit_waiting_error_warning_panel
            .add(input_code_credit_waiting_error_warningLab);
        JPanel message_panel = new JPanel();
        final JLabel messageLab = new JLabel(err);
        message_panel.add(messageLab);
        JPanel initialise_panel = new JPanel();
        JButton initialise = new JButton("initialise");
        initialise_panel.add(initialise);

         // Add Action to the button
        initialise.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                    Input_code_credit_Waiting backWindow = new
                        Input_code_credit_Waiting("Input_code_credit_Waiting");
                    backWindow.setSize(400, 300);
                    backWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    backWindow.setLocationRelativeTo(null);
                    backWindow.setVisible(true);
                    dispose();
            }
        });

        // Set Layout Manager
        setLayout(new GridLayout(3, 0));

        // Add Panels to the Window
        this.add(input_code_credit_waiting_error_warning_panel);
        this.add(message_panel);
        this.add(initialise_panel);
        }
    } // END OF CLASS
```

## Input_code_Waiting

Listing A.18: *Input_code_Waiting.java*

```
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
```

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Input_code_Waiting extends JFrame {
    public Input_code_Waiting(String winTitle) {

        super(winTitle);

        // Create Swing Components
        JPanel message_panel = new JPanel();
        final JLabel messageLab = new JLabel("Input your Data");
        message_panel.add(messageLab);
        JPanel code_panel = new JPanel();
        JLabel code_label = new JLabel("code");
        final JTextField codeTxt = new JTextField(30);
        code_panel.add(code_label);
        code_panel.add(codeTxt);
        JPanel input_panel = new JPanel();
        JButton input = new JButton("Input");
        input_panel.add(input);

         // Add Action to the button
        input.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                    String code = codeTxt.getText();

                 if((codeTxt.getText().isEmpty())) {
                     Input_code_Waiting_Error nextErrWindow = new
                         Input_code_Waiting_Error("Input_code_Waiting_Error ",
                              "Invalid Input. Please Try Again" );
                    nextErrWindow.setSize(400, 300);
                    nextErrWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    nextErrWindow.setLocationRelativeTo(null);
                    nextErrWindow.setVisible(true);
                    dispose();
                     }

                 else {
                     Read_Desc_Ready nextWindow = new Read_Desc_Ready("Read_Desc_
                         Ready", code);
                     nextWindow.setSize(400, 300);
                     nextWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                     nextWindow.setLocationRelativeTo(null);
                     nextWindow.setVisible(true);
                     dispose();
                     }
            }
        });

        // Set Layout Manager
        setLayout(new GridLayout(3, 0));

        // Add Panels to the Window
        this.add(message_panel);
        this.add(code_panel);
        this.add(input_panel);
        }
    } // END OF CLASS
```

## Input_code_Waiting_Error

Listing A.19: *Input_code_Waiting_Error.java*

```java
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Input_code_Waiting_Error extends JFrame {
    public Input_code_Waiting_Error(String winTitle, String err) {
        super(winTitle);

        // Create Swing Components
        JPanel input_code_waiting_error_warning_panel = new JPanel();
        final JLabel input_code_waiting_error_warningLab = new JLabel();
        input_code_waiting_error_warning_panel
            .add(input_code_waiting_error_warningLab);
```

```
        JPanel message_panel = new JPanel();
        final JLabel messageLab = new JLabel(err);
        message_panel.add(messageLab);
        JPanel initialise_panel = new JPanel();
        JButton initialise = new JButton("initialise");
        initialise_panel.add(initialise);

         // Add Action to the button
        initialise.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                     Input_code_Waiting backWindow = new
                        Input_code_Waiting("Input_code_Waiting");
                    backWindow.setSize(400, 300);
                    backWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    backWindow.setLocationRelativeTo(null);
                    backWindow.setVisible(true);
                    dispose();
            }
        });

        // Set Layout Manager
        setLayout(new GridLayout(3, 0));

        // Add Panels to the Window
        this.add(input_code_waiting_error_warning_panel);
        this.add(message_panel);
        this.add(initialise_panel);
        }
    } // END OF CLASS
```

## Create_Module_Ready

Listing A.20: *Create_Module_Ready.java*

```
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Create_Module_Ready extends JFrame {
    public Create_Module_Ready(String winTitle, final String code,
        final String title, final String credit, final String desc) {
        super(winTitle);

        // Create Swing Components
        JPanel code_panel = new JPanel();
        final JLabel codeLab = new JLabel();
        code_panel.add(codeLab);
        JPanel title_panel = new JPanel();
        final JLabel titleLab = new JLabel();
        title_panel.add(titleLab);
        JPanel credit_panel = new JPanel();
        final JLabel creditLab = new JLabel();
        credit_panel.add(creditLab);
        JPanel desc_panel = new JPanel();
        final JLabel descLab = new JLabel();
        desc_panel.add(descLab);
        JPanel message_panel = new JPanel();
        final JLabel messageLab = new
             JLabel("Are you sure to proceed Creating ");
        message_panel.add(messageLab);
        JPanel create_panel = new JPanel();
        JButton create = new JButton("Create");
        create_panel.add(create);

         // Add Action to the button
        create.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                    boolean recordInserted = false;

                // Declare Connection properties
                Connection conn = null;
                Statement stmt = null;
                int recAffected = 0;

                // Declare Connection properties
                try {
                    Class.forName("com.mysql.jdbc.Driver").newInstance();
                    String connectionUrl = "jdbc:mysql://localhost:3306/testdb1";
                    String connectionUser = "root";
                    String connectionPassword = "";
```

```java
                        conn = DriverManager.getConnection(connectionUrl,
                            connectionUser, connectionPassword);
                        stmt = conn.createStatement();

                        // Execute a query statement or a stored procedure
                        CallableStatement procnone =
                            conn.prepareCall("{call createModule(?, ?, ?, ?)}");

                        // Sets the input parameter
                        procnone.setString(2, title);
                        procnone.setString(4, desc);
                        procnone.setInt(1, Integer.parseInt(code));
                        procnone.setInt(3, Integer.parseInt(credit));

                        // Registers the out parameters
                        recAffected =  procnone.executeUpdate();

                        // Check if there is an updated record
                        if (recAffected > 0)
                            recordInserted = true;
                        else
                            recordInserted = false;

                    } catch (Exception e1) {
                      e1.printStackTrace();
                    } finally {
                      try { if (stmt != null) stmt.close(); } catch (SQLException e1)
                          { e1.printStackTrace(); }
                      try { if (conn != null) conn.close(); } catch (SQLException e1)
                          { e1.printStackTrace(); }
                    }
                if(recordInserted) {
                        JOptionPane.showMessageDialog(null,
                            "The business task is completed successfully.");
                        System.exit(0);
                }
                else {
                         Create_Module_Ready_Error nextErrWindow = new
                              Create_Module_Ready_Error("Create_Module_Ready_Error ",
                            "Can not be Created. Please Try Again" );
                        nextErrWindow.setSize(400, 300);
                        nextErrWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                        nextErrWindow.setLocationRelativeTo(null);
                        nextErrWindow.setVisible(true);
                        dispose();
                       }
              }
          });

          // Set Layout Manager
          setLayout(new GridLayout(6, 0));

          // Add Panels to the Window
          this.add(code_panel);
          this.add(title_panel);
          this.add(credit_panel);
          this.add(desc_panel);
          this.add(message_panel);
          this.add(create_panel);
          }
    } // END OF CLASS
```

## Create_Module_Ready_Error

Listing A.21: *Create_Module_Ready_Error.java*

```java
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Create_Module_Ready_Error extends JFrame {
    public Create_Module_Ready_Error(String winTitle, String err) {
        super(winTitle);

        // Create Swing Components
        JPanel create_module_ready_error_warning_panel = new JPanel();
        final JLabel create_module_ready_error_warningLab = new JLabel();
        create_module_ready_error_warning_panel
            .add(create_module_ready_error_warningLab);
        JPanel message_panel = new JPanel();
```

```java
        final JLabel messageLab = new JLabel(err);
        message_panel.add(messageLab);
        JPanel exit_panel = new JPanel();
        JButton exit = new JButton("exit");
        exit_panel.add(exit);

         // Add Action to the button
        exit.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                    Input_code_title_credit_desc_Waiting backWindow = new
                        Input_code_title_credit_desc_Waiting("Input_code_title_credit_desc_Waiting");
                    backWindow.setSize(400, 300);
                    backWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    backWindow.setLocationRelativeTo(null);
                    backWindow.setVisible(true);
                    dispose();
            }
        });

        // Set Layout Manager
        setLayout(new GridLayout(3, 0));

        // Add Panels to the Window
        this.add(create_module_ready_error_warning_panel);
        this.add(message_panel);
        this.add(exit_panel);
        }
    } // END OF CLASS
```

## Delete_Module_Ready

Listing A.22: *Delete_Module_Ready.java*

```java
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Delete_Module_Ready extends JFrame {
    public Delete_Module_Ready(String winTitle, final String code) {
        super(winTitle);

        // Create Swing Components
        JPanel code_panel = new JPanel();
        final JLabel codeLab = new JLabel();
        code_panel.add(codeLab);
        JPanel message_panel = new JPanel();
        final JLabel messageLab = new JLabel("Are you sure to proceed Deleting ");
        message_panel.add(messageLab);
        JPanel delete_panel = new JPanel();
        JButton delete = new JButton("Delete");
        delete_panel.add(delete);

         // Add Action to the button
        delete.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                    boolean recordDeleted = false;

                // Declare Connection properties
                Connection conn = null;
                Statement stmt = null;
                int recAffected = 0;

                // Declare Connection properties
                try {
                    Class.forName("com.mysql.jdbc.Driver").newInstance();
                    String connectionUrl = "jdbc:mysql://localhost:3306/testdb1";
                    String connectionUser = "root";
                    String connectionPassword = "";
                    conn = DriverManager.getConnection(connectionUrl, connectionUser,
                        connectionPassword);
                    stmt = conn.createStatement();

                    // Execute a query statement or a stored procedure
                    CallableStatement procnone =
                        conn.prepareCall("{call deleteModule(?)}");

                    // Sets the input parameter
                    procnone.setInt(1, Integer.parseInt(code));

                    recAffected =  procnone.executeUpdate();
```

```

                    // Check if there is an updated record
                    if (recAffected > 0)
                        recordDeleted = true;
                    else
                        recordDeleted = false;

                } catch (Exception e1) {
                    e1.printStackTrace();
                } finally {
                    try { if (stmt != null) stmt.close(); } catch (SQLException e1)
                        { e1.printStackTrace(); }
                    try { if (conn != null) conn.close(); } catch (SQLException e1)
                        { e1.printStackTrace(); }
                }
            if(recordDeleted) {
                    JOptionPane.showMessageDialog(null,
                        "The business task is completed successfully.");
                    System.exit(0);
            }
            else {
                     Delete_Module_Ready_Error nextErrWindow = new
                         Delete_Module_Ready_Error("Delete_Module_Ready_Error ",
                         "Can not be Deleted. Please Try Again" );
                    nextErrWindow.setSize(400, 300);
                    nextErrWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    nextErrWindow.setLocationRelativeTo(null);
                    nextErrWindow.setVisible(true);
                    dispose();
                   }
            }
        });

        // Set Layout Manager
        setLayout(new GridLayout(3, 0));

        // Add Panels to the Window
        this.add(code_panel);
        this.add(message_panel);
        this.add(delete_panel);
        }
    } // END OF CLASS
```

## Delete_Module_Ready_Error

Listing A.23: *Delete_Module_Ready_Error.java*

```
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Delete_Module_Ready_Error extends JFrame {
    public Delete_Module_Ready_Error(String winTitle, String err) {
        super(winTitle);

        // Create Swing Components
        JPanel delete_module_ready_error_warning_panel = new JPanel();
        final JLabel delete_module_ready_error_warningLab = new JLabel();
        delete_module_ready_error_warning_panel
            .add(delete_module_ready_error_warningLab);
        JPanel message_panel = new JPanel();
        final JLabel messageLab = new JLabel(err);
        message_panel.add(messageLab);
        JPanel exit_panel = new JPanel();
        JButton exit = new JButton("exit");
        exit_panel.add(exit);

         // Add Action to the button
        exit.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                     Input_code_Waiting backWindow = new
                        Input_code_Waiting("Input_code_Waiting");
                    backWindow.setSize(400, 300);
                    backWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    backWindow.setLocationRelativeTo(null);
                    backWindow.setVisible(true);
                    dispose();
            }
        });

```

```
        // Set Layout Manager
        setLayout(new GridLayout(3, 0));

        // Add Panels to the Window
        this.add(delete_module_ready_error_warning_panel);
        this.add(message_panel);
        this.add(exit_panel);
        }
    } // END OF CLASS
```

## Read_Desc_Ready

Listing A.24: *Read_Desc_Ready.java*

```
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Read_Desc_Ready extends JFrame {
    public Read_Desc_Ready(String winTitle, final String code) {
        super(winTitle);

        // Create Swing Components
        JPanel code_panel = new JPanel();
        final JLabel codeLab = new JLabel();
        code_panel.add(codeLab);
        JPanel message_panel = new JPanel();
        final JLabel messageLab = new JLabel("Are you sure to proceed Reading ");
        message_panel.add(messageLab);
        JPanel read_panel = new JPanel();
        JButton read = new JButton("Read");
        read_panel.add(read);

         // Add Action to the button
        read.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                    boolean recordReturned = false;

                // Declare Connection properties
                Connection conn = null;
                Statement stmt = null;
                ResultSet rs = null;

                // Declare Connection properties
                try {
                    Class.forName("com.mysql.jdbc.Driver").newInstance();
                    String connectionUrl = "jdbc:mysql://localhost:3306/testdb1";
                    String connectionUser = "root";
                    String connectionPassword = "";
                    conn = DriverManager.getConnection(connectionUrl, connectionUser,
                        connectionPassword);
                    stmt = conn.createStatement();

                    // Execute a query statement or a stored procedure
                    CallableStatement procnone =
                        conn.prepareCall("{call readDesc(?, ?)}");

                    // Sets the input parameter
                    procnone.setInt(1, Integer.parseInt(code));

                    // Registers the out parameters
                    procnone.registerOutParameter(2, Types.VARCHAR);

                    procnone.executeUpdate ();

                    // Declare variable(s) to stored the result(s) of the query
                    String descVar = procnone.getString(2);
                     System.out.println(descVar +" ");

                    // Check if there is a returned record
                    if (descVar != null)
                        recordReturned = true;
                    else
                        recordReturned = false;

                 } catch (Exception e1) {
                     e1.printStackTrace();
                 } finally {
                     try { if (rs != null) rs.close(); } catch (SQLException e1)
                         { e1.printStackTrace(); }
```

```
                        try { if (stmt != null) stmt.close(); } catch (SQLException e1)
                            { e1.printStackTrace(); }
                        try { if (conn != null) conn.close(); } catch (SQLException e1)
                            { e1.printStackTrace(); }
                    }
                if(recordReturned) {
                       JOptionPane.showMessageDialog(null,
                           "The business task is completed successfully.");
                       System.exit(0);
                }
                else {
                        Read_Desc_Ready_Error nextErrWindow = new
                            Read_Desc_Ready_Error("Read_Desc_Ready_
                               Error ", "Can not be retrieved. Please Try Again" );
                       nextErrWindow.setSize(400, 300);
                       nextErrWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                       nextErrWindow.setLocationRelativeTo(null);
                       nextErrWindow.setVisible(true);
                       dispose();
                       }
                }
        });

        // Set Layout Manager
        setLayout(new GridLayout(3, 0));

        // Add Panels to the Window
        this.add(code_panel);
        this.add(message_panel);
        this.add(read_panel);
        }
    } // END OF CLASS
```

## Read_Desc_Ready_Error

Listing A.25: *Read_Desc_Ready_Error.java*

```
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Read_Desc_Ready_Error extends JFrame {
    public Read_Desc_Ready_Error(String winTitle, String err) {
        super(winTitle);

        // Create Swing Components
        JPanel read_desc_ready_error_warning_panel = new JPanel();
        final JLabel read_desc_ready_error_warningLab = new JLabel();
        read_desc_ready_error_warning_panel.add(read_desc_ready_error_warningLab);

        JPanel message_panel = new JPanel();
        final JLabel messageLab = new JLabel(err);
        message_panel.add(messageLab);

        JPanel exit_panel = new JPanel();
        JButton exit = new JButton("exit");
        exit_panel.add(exit);

         // Add Action to the button
        exit.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                        Input_code_Waiting backWindow = new
                            Input_code_Waiting("Input_code_Waiting");
                       backWindow.setSize(400, 300);
                       backWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                       backWindow.setLocationRelativeTo(null);
                       backWindow.setVisible(true);
                       dispose();
            }
        });

        // Set Layout Manager
        setLayout(new GridLayout(3, 0));

        // Add Panels to the Window
        this.add(read_desc_ready_error_warning_panel);
        this.add(message_panel);
        this.add(exit_panel);
        }
    } // END OF CLASS
```

## Read_Module_Code_Credit_Ready

Listing A.26: *Read_Module_Code_Credit_Ready.java*

```
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Read_Module_Code_Credit_Ready extends JFrame {
    public Read_Module_Code_Credit_Ready(String winTitle, final String code,
         final String credit) {
        super(winTitle);

        // Create Swing Components
        JPanel code_panel = new JPanel();
        final JLabel codeLab = new JLabel();
        code_panel.add(codeLab);
        JPanel credit_panel = new JPanel();
        final JLabel creditLab = new JLabel();
        credit_panel.add(creditLab);
        JPanel message_panel = new JPanel();
        final JLabel messageLab = new JLabel("Are you sure to proceed Reading ");
        message_panel.add(messageLab);
        JPanel read_panel = new JPanel();
        JButton read = new JButton("Read");
        read_panel.add(read);

         // Add Action to the button
        read.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                    boolean recordReturned = false;

                // Declare Connection properties
                Connection conn = null;
                Statement stmt = null;
                ResultSet rs = null;

                // Declare Connection properties
                try {
                    Class.forName("com.mysql.jdbc.Driver").newInstance();
                    String connectionUrl = "jdbc:mysql://localhost:3306/testdb1";
                    String connectionUser = "root";
                    String connectionPassword = "";
                    conn = DriverManager.getConnection(connectionUrl, connectionUser,
                         connectionPassword);
                    stmt = conn.createStatement();

                    // Execute a query statement or a stored procedure
                    CallableStatement procnone =
                        conn.prepareCall("{call readCode_Credit(?, ?)}");

                    // Sets the input parameter
                    procnone.setInt(1, Integer.parseInt(code));

                    // Registers the out parameters
                    procnone.registerOutParameter(2, Types.INTEGER);

                    procnone.executeUpdate ();

                    // Declare variable(s) to stored the result(s) of the query
                    int creditVar = procnone.getInt(2);
                     System.out.println(creditVar +" ");

                    // Check if there is a returned record
                    if (creditVar != 0)
                        recordReturned = true;
                    else
                        recordReturned = false;

                 } catch (Exception e1) {
                     e1.printStackTrace();
                 } finally {
                     try { if (rs != null) rs.close(); } catch (SQLException e1)
                         { e1.printStackTrace(); }
                     try { if (stmt != null) stmt.close(); } catch (SQLException e1)
                         { e1.printStackTrace(); }
                     try { if (conn != null) conn.close(); } catch (SQLException e1)
                         { e1.printStackTrace(); }
                 }

            if(recordReturned) {
                     Write_Module_Ready nextWindow = new
                         Write_Module_Ready("Write_Module_Ready", codeLab.getText(),
                             creditLab.getText());
                    nextWindow.setSize(400, 300);
```

```
                    nextWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    nextWindow.setLocationRelativeTo(null);
                    nextWindow.setVisible(true);
                    dispose();

                }
                else {
                     Read_Module_Code_Credit_Ready_Error nextErrWindow = new
                         Read_Module_Code_Credit_Ready_Error("Read_Module_Code_
                             Credit_Ready_Error ",
                                 "Can not be retrieved. Please Try Again" );
                    nextErrWindow.setSize(400, 300);
                    nextErrWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    nextErrWindow.setLocationRelativeTo(null);
                    nextErrWindow.setVisible(true);
                    dispose();
                    }
                }
        });

        // Set Layout Manager
        setLayout(new GridLayout(4, 0));

        // Add Panels to the Window
        this.add(code_panel);
        this.add(credit_panel);
        this.add(message_panel);
        this.add(read_panel);
        }
    } // END OF CLASS
```

## Read_Module_Code_Credit_Ready_Error

Listing A.27: *Read_Module_Code_Credit_Ready_Error.java*

```
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Read_Module_Code_Credit_Ready_Error extends JFrame {
    public Read_Module_Code_Credit_Ready_Error(String winTitle, String err) {
        super(winTitle);

        // Create Swing Components
        JPanel read_module_code_credit_ready_error_warning_panel = new JPanel();
        final JLabel read_module_code_credit_ready_error_warningLab = new JLabel();
        read_module_code_credit_ready_error_warning_panel
            .add(read_module_code_credit_ready_error_warningLab);
        JPanel message_panel = new JPanel();
        final JLabel messageLab = new JLabel(err);
        message_panel.add(messageLab);
        JPanel input_panel = new JPanel();
        JButton input = new JButton("Input");
        input_panel.add(input);

         // Add Action to the button
        input.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                     Input_code_credit_Waiting backWindow = new
                        Input_code_credit_Waiting("Input_code_credit_Waiting");
                    backWindow.setSize(400, 300);
                    backWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    backWindow.setLocationRelativeTo(null);
                    backWindow.setVisible(true);
                    dispose();
                }
        });

        // Set Layout Manager
        setLayout(new GridLayout(3, 0));

        // Add Panels to the Window
        this.add(read_module_code_credit_ready_error_warning_panel);
        this.add(message_panel);
        this.add(input_panel);
        }
    } // END OF CLASS
```

## Write_Module_Ready

Listing A.28: *Write_Module_Ready.java*

```
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Write_Module_Ready extends JFrame {
    public Write_Module_Ready(String winTitle, final String code,
        final String credit) {
        super(winTitle);

        // Create Swing Components
        JPanel code_panel = new JPanel();
        final JLabel codeLab = new JLabel();
        code_panel.add(codeLab);
        JPanel credit_panel = new JPanel();
        final JLabel creditLab = new JLabel();
        credit_panel.add(creditLab);
        JPanel message_panel = new JPanel();
        final JLabel messageLab = new JLabel("Are you sure to proceed Updating ");
        message_panel.add(messageLab);
        JPanel write_panel = new JPanel();
        JButton write = new JButton("Write");
        write_panel.add(write);

         // Add Action to the button
        write.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                    boolean recordUpdated = false;

                // Declare Connection properties
                Connection conn = null;
                Statement stmt = null;
                int recAffected = 0;

                // Declare Connection properties
                try {
                    Class.forName("com.mysql.jdbc.Driver").newInstance();
                    String connectionUrl = "jdbc:mysql://localhost:3306/testdb1";
                    String connectionUser = "root";
                    String connectionPassword = "";
                    conn = DriverManager.getConnection(connectionUrl,
                        connectionUser, connectionPassword);
                    stmt = conn.createStatement();

                    // Execute a query statement or a stored procedure
                    CallableStatement procnone =
                         conn.prepareCall("{call writeModule(?, ?)}");

                    // Sets the input parameter
                    procnone.setInt(1, Integer.parseInt(code));
                    procnone.setInt(3, Integer.parseInt(credit));

                    // Registers the out parameters
                    procnone.registerOutParameter(3, Types.INTEGER);

                    recAffected =  procnone.executeUpdate();

                    // Check if there is an updated record
                    if (recAffected > 0)
                        recordUpdated = true;
                    else
                        recordUpdated = false;

                 } catch (Exception e1) {
                     e1.printStackTrace();
                 } finally {
                     try { if (stmt != null) stmt.close(); } catch (SQLException e1)
                         { e1.printStackTrace(); }
                     try { if (conn != null) conn.close(); } catch (SQLException e1)
                         { e1.printStackTrace(); }
                 }
              if(recordUpdated) {
                    System.out.println(" YES YES YES YES YES YES");
                    JOptionPane.showMessageDialog(null,
                        "The business task is completed successfully.");
                    System.exit(0);
              }
              else {
                    System.out.println(" NO NO NO NO NO NO NO NO");
                     Write_Module_Ready_Error nextErrWindow = new
                         Write_Module_Ready_Error("Write_Module_Ready_Error ",
                             "Can not be Updated. Please Try Again" );
```

```java
                        nextErrWindow.setSize(400, 300);
                        nextErrWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                        nextErrWindow.setLocationRelativeTo(null);
                        nextErrWindow.setVisible(true);
                        dispose();
                        }
                }
            });

            // Set Layout Manager
            setLayout(new GridLayout(4, 0));

            // Add Panels to the Window
            this.add(code_panel);
            this.add(credit_panel);
            this.add(message_panel);
            this.add(write_panel);
            }
    } // END OF CLASS
```

## Write_Module_Ready

Listing A.29: *Write_Module_Ready.java*

```java
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Write_Module_Ready_Error extends JFrame {
    public Write_Module_Ready_Error(String winTitle, String err) {
        super(winTitle);

        // Create Swing Components
        JPanel write_module_ready_error_warning_panel = new JPanel();
        final JLabel write_module_ready_error_warningLab = new JLabel();
        write_module_ready_error_warning_panel
            .add(write_module_ready_error_warningLab);
        JPanel message_panel = new JPanel();
        final JLabel messageLab = new JLabel(err);
        message_panel.add(messageLab);
        JPanel exit_panel = new JPanel();
        JButton exit = new JButton("exit");
        exit_panel.add(exit);

         // Add Action to the button
        exit.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                     Input_code_credit_Waiting backWindow = new
                        Input_code_credit_Waiting("Input_code_credit_Waiting");
                    backWindow.setSize(400, 300);
                    backWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    backWindow.setLocationRelativeTo(null);
                    backWindow.setVisible(true);
                    dispose();
            }
        });

        // Set Layout Manager
        setLayout(new GridLayout(3, 0));

        // Add Panels to the Window
        this.add(write_module_ready_error_warning_panel);
        this.add(message_panel);
        this.add(exit_panel);
        }
    } // END OF CLASS
```

# B

# Appendix: Models and Java Code of Experiment (2)

## B.1 Complete Models and Full results of Experiment (2)

This Appendix presents the XML representations of all $\mu$ML models of the Student Enrollment Sub-system case study, and the completed Java and MySQL code generated from these models via BUILD. The experiment presented previously in chapter 10.

### Experiment (2): Requirement Models Construction

Listing B.1: CaseStudy2.java

```
package mde.example;

import java.io.File;
import java.io.IOException;
import mde.data.model.DDiagram;
import mde.database.gen.DumpFileGenerator;
import mde.mysql.gen.MySQLDumpFileGenerator;
import mde.database.gen.TreeException;
import mde.dataflow.model.DfDiagram;
import mde.dbs.model.Schema;
import mde.dfd2ddfd.rule.DfDiagramToDfDiagram;
import mde.dfd2state.rule.DfdDiagramToStDiagram;
import mde.dm2schem.rule.DDiagramToSchema2;
import mde.dm_dfd_state2code.rule.DDiagramToCDiagram2x;
import mde.gui.gen.CodeFileGenerator;
import mde.gui.model.GuiBoundary;
import mde.gui.model.GuiDiagram;
import mde.impact.model.ImpBoundary;
import mde.impact.model.ImpDiagram;
import mde.impact.model.ImpCreateFlow;
import mde.impact.model.ImpObject;
import mde.impact.model.ImpRole;
import mde.impact.model.ImpTask;
import mde.inf2dm.rule.InfDiagramToDDiagram;
import mde.information.model.Association;
import mde.information.model.Entity;
import mde.information.model.Attribute;
import mde.information.model.Role;
import mde.javagui.gen.JavaCodeFileGenerator;
import mde.model.Type;
import mde.state.model.StDiagram;
import mde.state2win.rule.StDiagramToGuiDiagram;
import mde.task.model.Actor;
import mde.task.model.Boundary;
import mde.task.model.Diagram;
import mde.task.model.Participation;
import mde.task.model.Task;
```

```
import mde.taskimpact2dataflow.rule.DiagramToDfDiagram;
import org.jast.ast.ASTWriter;

public class CaseStudy2 {
    public static void main(String[] args) throws IOException,
        TreeException, mde.gui.gen.TreeException {
        // TODO Auto-generated method stub

        //Construct the Task Model
        Diagram taskModel = new Diagram();
        Boundary boundary = new Boundary("Enrol a Student");
        Actor actor1 = new Actor("Student");
        Task enrolStd = new Task("Enrol");
        Participation link4 = new Participation();
        link4.addRole(new mde.task.model.Role("student",actor1));
        link4.addRole(new mde.task.model.Role("enrol", enrolStd));
        Participation link3 = new Participation();
        link3.addRole(new mde.task.model.Role("student",actor1));
        link3.addRole(new mde.task.model.Role("enrol", enrolStd));
        boundary.addActor(actor1);
        boundary.addParticipation(link4);
        boundary.addParticipation(link3);
        taskModel.addBoundary(boundary);

        ASTWriter writer = new ASTWriter(new File("Stud_Enrol_taskModel.xml"));
        writer.usePackage("mde.task.model", "xmlns:task");
        writer.writeDocument(taskModel);
        writer.close();

        System.out.println("(1) Task Model is Created by user.");
        // ---------------------------------------- //

        // Construct the Impact Model
        ImpDiagram ImpactModel = new ImpDiagram();
        ImpBoundary impboundary = new ImpBoundary("Enrol a Student");
        ImpTask impEnrolStd = new mde.impact.model.ImpTask("Enrol");
        ImpObject impObj3 = new ImpObject("Enrollment");
        ImpCreateFlow cf = new ImpCreateFlow();
        ImpRole impcf2 = new ImpRole("enrol", impEnrolStd);
        ImpRole impcf1 = new ImpRole("enrollment", impObj3);
        cf.addImpRole(impcf2);
        cf.addImpRole(impcf1);
        impboundary.addImpObject(impObj3);
        impboundary.addImpCreateFlow(cf);
        ImpactModel.addImpBoundary(impboundary);

        ASTWriter writer1 = new ASTWriter(new File("Stud_Enrol_impactModel.xml"));
        writer1.usePackage("mde.impact.model", "xmlns:imp");
        writer1.writeDocument(ImpactModel);
        writer1.close();

        System.out.println("(2) Impact Model is Created by user.");
        // ---------------------------------------------- //

        // Construct the Information Model
        mde.information.model.Diagram informationModel =
            new mde.information.model.Diagram();
        Entity studentEntity = new Entity("Account");
        Attribute attr5 = new Attribute("regNumber", new Type("Integer")).setIdentifier(true);
        Attribute attr2 = new Attribute("name", new Type("String"));
        Attribute attr3 = new Attribute("age", new Type("String"));
        Attribute attr4 = new Attribute("gender", new Type("String"));
        Attribute attr7 = new Attribute("username", new Type("String"));
        Attribute attr8 = new Attribute("password", new Type("String"));
        studentEntity.addAttribute(attr5);
        studentEntity.addAttribute(attr2);
        studentEntity.addAttribute(attr3);
        studentEntity.addAttribute(attr4);
        studentEntity.addAttribute(attr7);
        studentEntity.addAttribute(attr8);

        Entity enrolEntity = new Entity("Enrollment");
        Attribute attr10 = new Attribute("regNumber", new Type("Integer")).setIdentifier(true);
        Attribute attr11 = new Attribute("code", new Type("String")).setIdentifier(true);
        enrolEntity.addAttribute(attr10);
        enrolEntity.addAttribute(attr11);

        Entity moduleEntity = new Entity("Module");
        Attribute attr12 = new Attribute("code", new Type("Integer")).setIdentifier(true);
        Attribute attr13 = new Attribute("title", new Type("String"));
        Attribute attr14 = new Attribute("credit", new Type("Integer"));
        Attribute attr15 = new Attribute("desc", new Type("String"));
        moduleEntity.addAttribute(attr12);
        moduleEntity.addAttribute(attr13);
        moduleEntity.addAttribute(attr14);
        moduleEntity.addAttribute(attr15);

        informationModel.addEntity(studentEntity);
        informationModel.addEntity(moduleEntity);
        informationModel.addEntity(enrolEntity);
        informationModel.addAssociation(new Association()
        .addRole(new Role("module", informationModel
            .getEntity("Module")).setMultiple(false))
        .addRole(new Role("enrol", informationModel
```

```
            .getEntity("Enrollment")).setMultiple(true)));
        informationModel.addAssociation(new Association()
        .addRole(new Role("student", informationModel
            .getEntity("Account")).setMultiple(false))
        .addRole(new Role("enrol", informationModel
            .getEntity("Enrollment")).setMultiple(true)));

        ASTWriter writer2 = new ASTWriter(new File("Stud_Enrol_informationModel.xml"));
        writer2.usePackage("mde.information.model", "xmlns:inf");
        writer2.writeDocument(informationModel);
        writer2.close();

        System.out.println("(3) Inf. Model is Created by user.");
        // ------------------------------------------------ //

        //Generate DataFlow Model
        DiagramToDfDiagram topRule = new DiagramToDfDiagram();
        DfDiagram  dataflowModel = topRule.translate(taskModel, ImpactModel);

        ASTWriter writer3 = new ASTWriter(new File("Stud_Enrol_DataFlowModel.xml"));
        writer3.usePackage("mde.dataflow.model", "xmlns:dfd");
        writer3.writeDocument(dataflowModel);
        writer3.close();

        System.out.println("(4) DataFlow Model is Created"+
            "(Task + Impact Model --> DataFlow Model).");
        // -------------------------------------------- //

        //Adding some data on flows in dataflow model.
        // input to login task
        dataflowModel.getDfBoundaries().get(0).getDfInputFlows()
            .get(0).addDataonflow("regNumber");
        dataflowModel.getDfBoundaries().get(0).getDfInputFlows()
            .get(1).addDataonflow("code");
        // create assignment
        dataflowModel.getDfBoundaries().get(0).getDfCreateFlows()
            .get(0).addAssignment("@regNumber = regNumber");
        dataflowModel.getDfBoundaries().get(0).getDfCreateFlows()
            .get(0).addAssignment("@code = code");
        // create
        dataflowModel.getDfBoundaries().get(0).getDfCreateFlows()
            .get(0).addDataonflow("regNumber");
        dataflowModel.getDfBoundaries().get(0).getDfCreateFlows()
            .get(0).addDataonflow("code");

        System.out.println("(4) Data on flows are added"+
            " by user in the DataFlow model.");
        // ------------------------------------------------ //

        // Generate Detailed DatFlow Model
        DfDiagramToDfDiagram topRule6 = new DfDiagramToDfDiagram();
        DfDiagram detailed_dataflowModel = topRule6.translate(dataflowModel);

        ASTWriter writer6 = new ASTWriter(new File("Stud_Enrol_Detailed_DataFlowModel.xml"));
        writer6.usePackage("mde.dataflow.model", "xmlns:dfd");
        writer6.writeDocument(detailed_dataflowModel);
        writer6.close();

        System.out.println("(5) Detailed DataFlow Model is Created"+
            "(DataFlow Model --> DataFlow Model).");
        // ------------------------------------------- //

        //Generate Data Model
        InfDiagramToDDiagram topRule4 = new InfDiagramToDDiagram();
        DDiagram  dataModel = topRule4.translate(informationModel);

        ASTWriter writer4 = new ASTWriter(new File("Stud_Enrol_DataModel.xml"));
        writer4.usePackage("mde.data.model", "xmlns:data");
        writer4.usePackage("mde.model", "xmlns:mod");
        writer4.writeDocument(dataModel);
        writer4.close();

        System.out.println("(5) Data Model is Created"+
            "(Information Model --> Data Model).");
        // --------------------------------------------- //

        //Generate State Model for Screens Navigation
        // (DataFlow Model ---> State Model)
        DfdDiagramToStDiagram topRule7 = new DfdDiagramToStDiagram();
        StDiagram  stateModel = topRule7.translate(detailed_dataflowModel, dataModel);

        ASTWriter writer7 = new ASTWriter(new File("Stud_Enrol_StateModel.xml"));
        writer7.usePackage("mde.state.model", "xmlns:state");
        writer7.writeDocument(stateModel);
        writer7.close();

        System.out.println("(6) State Model is Created (Detailed DataFlow Model --> State Model).");
        // ------------------------------------------------ //

        //Generate Gui Description Model
        //(State Model ---> Gui Specification Model)
        StDiagramToGuiDiagram topRule9 = new StDiagramToGuiDiagram();
        GuiDiagram  GuiModel = topRule9.translate(stateModel);

```

```
        ASTWriter writer9 = new ASTWriter(new File("Stud_Enrol_GuiModel.xml"));
        writer9.usePackage("mde.gui.model", "xmlns:gui");
        writer9.writeDocument(GuiModel);
        writer9.close();

        System.out.println("(7) Gui Specification Model is Created (State Model --> Gui Model).");
        // ---------------------------------------------------------------- //

        //generate Database Schema Model
        // (Detailed_DFD + Data Model ---> Schema)
        DDiagramToSchema2 topRule5 = new DDiagramToSchema2();
        Schema schemaModel = topRule5.translate(detailed_dataflowModel, dataModel);

        System.out.println(schemaModel.getName());

        ASTWriter writer5 = new ASTWriter(new File("Stud_Enrol_SchemaModel.xml"));
        writer5.usePackage("mde.dbs.model", "xmlns:dbs");
        writer5.usePackage("mde.model", "xmlns:mod");
        writer5.writeDocument(schemaModel);
        writer5.close();

        System.out.println("(9) Database Schema Model is Created"+
            " (DataFlow + Data Model --> Database Schema Model).");
        // ---------------------------------------------------------------- //

        //Generate Code Model
        // (Database Schema + DataFlow Model ---> Code Model)
        DDiagramToCDiagram2x topRule8 = new DDiagramToCDiagram2x();
        mde.code.model.Diagram  codeModel = topRule8
            .translate(schemaModel, detailed_dataflowModel);

        ASTWriter writer8 = new ASTWriter(new File("CodeModel.xml"));
        writer8.usePackage("mde.code.model", "xmlns:code");
        writer8.usePackage("mde.model", "xmlns:mod");
        writer8.writeDocument(codeModel);
        writer8.close();

        System.out.println("(10) Code Model is Created"+
            " (Database Scehma Model --> Code Model).");
        System.out.println("\nModel Transformation System is Completed ...");

        // Code Generation
        DumpFileGenerator MySQLGenerator = new MySQLDumpFileGenerator(schemaModel);
        System.out.println("calling generate in mysql package generator");
        MySQLGenerator.generate();
        System.out.println("Finished generating MySQL Schema OK");

        for(GuiBoundary bound : GuiModel.getGuiBoundaries())
        {
            CodeFileGenerator JavaGenerator = new JavaCodeFileGenerator(bound);
            System.out.println("calling generate in java code file generator");
            JavaGenerator.generate();
            System.out.println("Finished generating Java Gui code");
        }
    }
}
```

## Task model: Student Registration System (Enrol a Student)

Listing B.2: *Stud_Enrol_taskModel.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<task:Diagram xmlns:task="mde.task.model" id="0">
  <task:Boundary id="1" name="Enrol a Student">
    <task:Actor id="2" name="Student"/>
    <task:Participation id="3">
      <task:Role id="4" name="student">
        <task:Actor ref="2"/>
      </task:Role>
      <task:Role id="5" name="enrol">
        <task:Task id="6" name="Enrol"/>
      </task:Role>
    </task:Participation>
    <task:Participation id="7">
      <task:Role id="8" name="student">
        <task:Actor ref="2"/>
      </task:Role>
      <task:Role id="9" name="enrol">
        <task:Task ref="6"/>
      </task:Role>
    </task:Participation>
  </task:Boundary>
</task:Diagram>
```

## Impact model: Student Registration System (Enrol a Student)

Listing B.3: *Stud_Enrol_impactModel.xml*

```xml
<imp:ImpDiagram xmlns:imp="mde.impact.model" id="0">
  <imp:ImpBoundary id="1" name="Enrol a Student">
    <imp:ImpObject id="2" name="Enrollment"/>
    <imp:ImpCreateFlow id="3">
      <imp:ImpRole id="4" name="enrol">
        <imp:ImpTask id="5" name="Enrol"/>
      </imp:ImpRole>
      <imp:ImpRole id="6" name="enrollment">
        <imp:ImpObject ref="2"/>
      </imp:ImpRole>
    </imp:ImpCreateFlow>
  </imp:ImpBoundary>
</imp:ImpDiagram>
```

## Inofrmation model: Student Registration System (Enrol a Student)

Listing B.4: *Stud_Enrol_InformationModel.xml*

```xml
<inf:Diagram xmlns:inf="mde.information.model" id="0">
  <inf:Entity id="23" name="Enrollment">
    <inf:Attribute id="24" name="regNumber" identifier="true" size="0">
      <Type id="25" name="Integer"/>
    </inf:Attribute>
    <inf:Attribute id="26" name="code" identifier="true" size="0">
      <Type id="27" name="String"/>
    </inf:Attribute>
  </inf:Entity>
</inf:Diagram>
```

## DataFlow model (initial): Student Registration System (Enrol a Student)

Listing B.5: *Stud_Enrol_DataFlowModel.xml*

```xml
<dfd:DfDiagram xmlns:dfd="mde.dataflow.model" id="0">
  <dfd:DfBoundary id="1" name="Enrol a Student">
    <dfd:DfActor id="2" name="Student"/>
    <dfd:DfObject id="3" name="Enrollment"/>
    <dfd:DfInputFlow id="4">
      <dfd:DfRole id="5" name="student">
        <dfd:DfActor ref="2"/>
      </dfd:DfRole>
      <dfd:DfRole id="6" name="enrol">
        <dfd:DfTask id="7" name="Enrol"/>
      </dfd:DfRole>
    </dfd:DfInputFlow>
    <dfd:DfInputFlow id="8">
      <dfd:DfRole id="9" name="student">
        <dfd:DfActor ref="2"/>
      </dfd:DfRole>
      <dfd:DfRole id="10" name="enrol">
        <dfd:DfTask ref="7"/>
      </dfd:DfRole>
    </dfd:DfInputFlow>
    <dfd:DfCreateFlow id="11">
      <dfd:DfRole id="12" name="enrol">
        <dfd:DfTask id="13" name="Enrol"/>
      </dfd:DfRole>
      <dfd:DfRole id="14" name="enrollment">
        <dfd:DfTask id="15" name="Enrollment"/>
      </dfd:DfRole>
    </dfd:DfCreateFlow>
  </dfd:DfBoundary>
</dfd:DfDiagram>
```

## DataFlow model (detailed): Student Registration System (Enrol a Student)

Listing B.6: *Stud_Enrol_Detailed_DataFlowModel.xml*

```
<dfd:DfDiagram xmlns:dfd="mde.dataflow.model" id="0">
<dfd:DfBoundary id="1" name="Enrol">
    <dfd:DfTask id="2" name="Input RegNumber"/>
    <dfd:DfTask id="3" name="Create Enrollment"/>
    <dfd:DfTask id="4" name="Input Code"/>
    <dfd:DfActor id="5" name="Student"/>
    <dfd:DfObject id="6" name="Enrollment"/>
    <dfd:DfInputFlow id="7">
      <dfd:DfRole id="8" name="input_actor">
        <dfd:DfActor id="9" name="Student"/>
      </dfd:DfRole>
      <dfd:DfRole id="10" name="input_task">
        <dfd:DfTask ref="2"/>
      </dfd:DfRole>
    </dfd:DfInputFlow>
    <dfd:DfInputFlow id="11">
      <dfd:DfRole id="12" name="input_actor">
        <dfd:DfActor ref="5"/>
      </dfd:DfRole>
      <dfd:DfRole id="13" name="input_task">
        <dfd:DfTask ref="4"/>
      </dfd:DfRole>
    </dfd:DfInputFlow>
    <dfd:DfReadFlow id="14">
      <dfd:DfRole id="15" name="input_task">
        <dfd:DfTask ref="2"/>
      </dfd:DfRole>
      <dfd:DfRole id="16" name="Input Code">
        <dfd:DfTask ref="4"/>
      </dfd:DfRole>
    </dfd:DfReadFlow>
    <dfd:DfReadFlow id="17">
      <dfd:DfRole id="18" name="input_task">
        <dfd:DfTask ref="4"/>
      </dfd:DfRole>
      <dfd:DfRole id="19" name="create_task">
        <dfd:DfTask ref="3"/>
      </dfd:DfRole>
    </dfd:DfReadFlow>
    <dfd:DfCreateFlow id="20">
      <dfd:DfRole id="21" name="create_task0">
        <dfd:DfTask ref="3"/>
      </dfd:DfRole>
      <dfd:DfRole id="22" name="create_object0">
        <dfd:DfObject ref="6"/>
      </dfd:DfRole>
    </dfd:DfCreateFlow>
    <dfd:DfCreateFlow ref="20"/>
  </dfd:DfBoundary>
</dfd:DfDiagram>
```

## Data Dependency model: Student Registration System (Enrol a Student)

Listing B.7: *Stud_Enrol_DataModel.xml*

```
<data:DDiagram xmlns:data="mde.data.model" id="0">
  <data:DEntity id="23" name="Enrollment">
    <data:DAttribute id="24" name="regNumber" identifier="true" size="10">
      <mod:Type id="25" name="Integer"/>
    </data:DAttribute>
    <data:DAttribute id="26" name="code" identifier="true" size="30">
      <mod:Type id="27" name="String"/>
    </data:DAttribute>
  </data:DEntity>
</data:DDiagram>
```

## Screen State model: Student Registration System (Enrol a Student)

Listing B.8: *Stud_Enrol_StateModel.xml*

```
<state:StDiagram xmlns:state="mde.state.model" id="0">
  <state:StBoundary id="1" name="Enrol">
    <state:State id="2" name="Start" priority="5"/>
    <state:State id="3" name="End" priority="0"/>
    <state:State id="4" name="input_regNumber_Waiting" priority="4"/>
    <state:State id="5" name="input_regNumber_Waiting_Error" priority="0"/>
    <state:State id="6" name="input_code_Waiting" priority="4"/>
    <state:State id="7" name="input_code_Waiting_Error" priority="0"/>
    <state:State id="8" name="Create Enrollment_Ready" priority="2"/>
```

```xml
    <state:State id="9" name="Create Enrollment_Ready_Error" priority="0"/>
    <state:Transition id="10" exit="false" action="initialise">
      <state:StRole id="11" name="start">
        <state:State ref="2"/>
      </state:StRole>
      <state:StRole id="12" name="start_task">
        <state:State ref="4"/>
      </state:StRole>
    </state:Transition>
    <state:Transition id="13" exit="false" action="Exception">
      <state:StRole id="14" name="state">
        <state:State ref="4"/>
      </state:StRole>
      <state:StRole id="15" name="error">
        <state:State ref="5"/>
      </state:StRole>
    </state:Transition>
    <state:Transition id="16" exit="false" action="initialise">
      <state:StRole id="17" name="error">
        <state:State ref="5"/>
      </state:StRole>
      <state:StRole id="18" name="state">
        <state:State ref="4"/>
      </state:StRole>
    </state:Transition>
    <state:Transition id="19" label="input()" exit="false" action="Input">
      <state:StRole id="20" name="input_regNumber_Waiting">
        <state:State ref="4"/>
      </state:StRole>
      <state:StRole id="21" name="input_code_Waiting">
        <state:State ref="6"/>
      </state:StRole>
    </state:Transition>
    <state:Transition id="22" exit="false" action="Exception">
      <state:StRole id="23" name="state">
        <state:State ref="6"/>
      </state:StRole>
      <state:StRole id="24" name="error">
        <state:State ref="7"/>
      </state:StRole>
    </state:Transition>
    <state:Transition id="25" exit="false" action="Input">
      <state:StRole id="26" name="error">
        <state:State ref="7"/>
      </state:StRole>
      <state:StRole id="27" name="state">
        <state:State ref="6"/>
      </state:StRole>
    </state:Transition>
    <state:Transition id="28" label="input()" exit="false" action="Input">
      <state:StRole id="29" name="input_code_Waiting">
        <state:State ref="6"/>
      </state:StRole>
      <state:StRole id="30" name="Create Enrollment_Ready">
        <state:State ref="8"/>
      </state:StRole>
    </state:Transition>
    <state:Transition id="31" label="create()" exit="false" action="Create">
      <state:StRole id="32" name="Create Enrollment_Ready">
        <state:State ref="8"/>
      </state:StRole>
      <state:StRole id="33" name="End">
        <state:State ref="3"/>
      </state:StRole>
    </state:Transition>
    <state:Transition id="34" exit="true" action="exit">
      <state:StRole id="35" name="end_task">
        <state:State ref="8"/>
      </state:StRole>
      <state:StRole id="36" name="end">
        <state:State ref="3"/>
      </state:StRole>
    </state:Transition>
    <state:Transition id="37" exit="false" action="Exception">
      <state:StRole id="38" name="state">
        <state:State ref="8"/>
      </state:StRole>
      <state:StRole id="39" name="error">
        <state:State ref="9"/>
      </state:StRole>
    </state:Transition>
    <state:Transition id="40" exit="false" action="exit">
      <state:StRole id="41" name="error">
        <state:State ref="9"/>
      </state:StRole>
      <state:StRole id="42" name="state">
        <state:State ref="8"/>
      </state:StRole>
    </state:Transition>
  </state:StBoundary>
</state:StDiagram>
```

## GUI model: Student Enrollment System (Enrol a Student)

Listing B.9: *Stud_Enrol_GuiModel.xml*

```
<gui:GuiDiagram xmlns:gui="mde.gui.model" id="0">
  <gui:GuiBoundary id="1" name="Enrol">
    <gui:Window id="2" name="Input_regNumber_Waiting" order="4">
      <gui:Textfield id="3" name="regNumber"/>
      <gui:Button id="4" name="Exception" event="Exception" exit="false"/>
      <gui:Button id="5" name="Input" event="Input" exit="false"/>
    </gui:Window>
    <gui:Window id="6" name="Input_regNumber_Waiting_Error" order="0" error="true">
      <gui:Label id="7" name="input_regNumber_Waiting_Error_warning"
        text="Null value not accepted"/>
      <gui:Button id="8" name="initialise" event="initialise" exit="false"/>
    </gui:Window>
    <gui:Window id="9" name="Input_code_Waiting" order="4">
      <gui:Textfield id="10" name="code"/>
      <gui:Button id="11" name="Exception" event="Exception" exit="false"/>
      <gui:Button id="12" name="Input" event="Input" exit="false"/>
    </gui:Window>
    <gui:Window id="13" name="Input_code_Waiting_Error" order="0" error="true">
      <gui:Label id="14" name="input_code_Waiting_Error_warning" text="Null value not accepted"/>
      <gui:Button id="15" name="Input" event="Input" exit="false"/>
    </gui:Window>
    <gui:Window id="16" name="Create Enrollment_Ready" order="2">
      <gui:Label id="17" name="regNumber" text="regNumber"/>
      <gui:Label id="18" name="code" text="code"/>
      <gui:Button id="19" name="Create" event="Create" exit="false"/>
      <gui:Button id="20" name="exit" event="exit" exit="true"/>
      <gui:Button id="21" name="Exception" event="Exception" exit="false"/>
    </gui:Window>
    <gui:Window id="22" name="Create Enrollment_Ready_Error" order="0" error="true">
      <gui:Label id="23" name="Create Enrollment_Ready_Error_warning"
        text="Connection to the Data source is fail"/>
      <gui:Button id="24" name="exit" event="exit" exit="false"/>
    </gui:Window>
  </gui:GuiBoundary>
</gui:GuiDiagram>
```

## Database and Query model: Student Enrollment System (Enrol a Student)

Listing B.10: *Manage_Module_SchemaModel.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<dbs:Schema xmlns:dbs="mde.dbs.model" id="0" name="database">
  <dbs:Table id="1" name="Enrollment">
    <dbs:Column id="2" name="regNumber" size="10">
      <mod:Type xmlns:mod="mde.model" id="3" name="INTEGER"/>
    </dbs:Column>
    <dbs:Column id="4" name="code" size="30">
      <mod:Type id="5" name="VARCHAR"/>
    </dbs:Column>
    <dbs:PrimaryKey id="6">
      <dbs:Column ref="2"/>
      <dbs:Column ref="4"/>
    </dbs:PrimaryKey>
  </dbs:Table>
  <dbs:Procedure id="7" name="createEnrollment" table="Enrollment">
    <dbs:Argument id="8" type="Integer" name="regNumber" in="true"/>
    <dbs:Argument id="9" type="String" name="code" in="true"/>
    <dbs:Create id="10" name="INSERT_Enrollment">
      <dbs:Operator id="11" type="boolean" symbol="Assign">
        <dbs:Column id="12" name="regNumber" size="10">
          <mod:Type id="13" name="INTEGER"/>
        </dbs:Column>
        <dbs:Variable id="14" type="INTEGER" name="regNumber"/>
      </dbs:Operator>
      <dbs:Operator id="15" type="boolean" symbol="Assign">
        <dbs:Column id="16" name="code" size="30">
          <mod:Type id="17" name="VARCHAR"/>
        </dbs:Column>
        <dbs:Variable id="18" type="VARCHAR" name="code"/>
      </dbs:Operator>
    </dbs:Create>
  </dbs:Procedure>
</dbs:Schema>
```

## OOP Code model: Student Enrollment System (Enrol a Student)

Listing B.11: *Stud_Enrol_CodeModel.xml*

```xml
<?xml version="1.0" encoding="UTF-8"?>
<code:Diagram xmlns:code="mde.code.model" id="0">
  <code:Clazz id="1" name="Enrollment" visible="public">
    <code:Attribute id="2" name="regNumber" visible="private">
      <mod:Type xmlns:mod="mde.model" id="3" name="int"/>
    </code:Attribute>
    <code:Attribute id="4" name="code" visible="private">
      <mod:Type id="5" name="String"/>
    </code:Attribute>
    <code:Method id="6" name="createEnrollment" visible="public">
      <mod:Type id="7" name="void"/>
      <code:Argument id="8" type="int" name="regNumber"/>
      <code:Argument id="9" type="String" name="code"/>
      <code:Call id="10" name="createEnrollment"/>
    </code:Method>
    <code:Constructor id="11" name="Enrollment" visible="public">
      <code:Operator id="12" type="boolean" symbol="assign">
        <code:Identifier id="13" type="int" name="regNumber" scope="Object"/>
        <code:Identifier id="14" type="int" name="regNu"/>
      </code:Operator>
      <code:Operator id="15" type="boolean" symbol="assign">
        <code:Identifier id="16" type="String" name="code" scope="Object"/>
        <code:Identifier id="17" type="String" name="cod"/>
      </code:Operator>
    </code:Constructor>
  </code:Clazz>
  <code:Clazz id="18" name="Enrol" visible="public">
    <code:Attribute id="19" name="Enrollment" visible="private">
      <mod:Type id="20" name="Enrollment"/>
    </code:Attribute>
    <code:Attribute id="21" name="Student" visible="private">
      <mod:Type id="22" name="Student"/>
    </code:Attribute>
    <code:Method id="23" name="Run" visible="public" staticm="true">
      <mod:Type id="24" name="void"/>
    </code:Method>
    <code:Constructor id="25" name="Enrol"/>
  </code:Clazz>
</code:Diagram>
```

## MySQL Script: Student Registration System (Enrol a Student)

Listing B.12: *database_MySQL.sql*

```sql
-- Database Creation

CREATE DATABASE sysDatabase;
USE sysDatabase;

-- Structure for table 'Enrollment'

CREATE TABLE Enrollment (
  regNumber INT(10) NOT NULL,
  code VARCHAR(30) NOT NULL,
  PRIMARY KEY(regNumber, code));

-- Structure of Procedure 'createEnrollment'

DELIMITER //

CREATE PROCEDURE createEnrollment(IN regNumber INT(10), IN code VARCHAR(30))
  BEGIN
    INSERT INTO Enrollment VALUES (regNumber, code);
  END //

DELIMITER //
```

## Java Swing Source Code: Student Enrollment System

## Input_code_Waiting

Listing B.13: *Input_code_Waiting.java*

```java
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
```

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Input_code_Waiting extends JFrame {
    public Input_code_Waiting(String winTitle, final String regNumber) {
        super(winTitle);

        // Create Swing Components
        JPanel message_panel = new JPanel();
        final JLabel messageLab = new JLabel("Input your Data");
        message_panel.add(messageLab);
        JPanel code_panel = new JPanel();
        JLabel code_label = new JLabel("code");
        final JTextField codeTxt = new JTextField(30);
        code_panel.add(code_label);
        code_panel.add(codeTxt);
        JPanel input_panel = new JPanel();
        JButton input = new JButton("Input");
        input_panel.add(input);

         // Add Action to the button
        input.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                    String code = codeTxt.getText();

                if((codeTxt.getText().isEmpty())) {
                    Input_code_Waiting_Error nextErrWindow = new
                        Input_code_Waiting_Error("Input_code_Waiting_Error ",
                            "Invalid Input. Please Try Again" );
                    nextErrWindow.setSize(400, 300);
                    nextErrWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    nextErrWindow.setLocationRelativeTo(null);
                    nextErrWindow.setVisible(true);
                    dispose();
                    }
                else {
                    Create_Enrollment_Ready nextWindow = new
                        Create_Enrollment_Ready("Create_Enrollment_Ready", regNumber, code);
                    nextWindow.setSize(400, 300);
                    nextWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    nextWindow.setLocationRelativeTo(null);
                    nextWindow.setVisible(true);
                    dispose();
                    }
            }
        });
        // Set Layout Manager
        setLayout(new GridLayout(3, 0));

        // Add Panels to the Window
        this.add(message_panel);
        this.add(code_panel);
        this.add(input_panel);
        }
    } // END OF CLASS
```

## Input_code_Waiting_Error

Listing B.14: *Input$_c$ode_Waiting_Error.java*

```
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Input_code_Waiting_Error extends JFrame {
    public Input_code_Waiting_Error(String winTitle, String err) {
        super(winTitle);

        // Create Swing Components
        JPanel input_code_waiting_error_warning_panel = new JPanel();
        final JLabel input_code_waiting_error_warningLab = new JLabel();
        input_code_waiting_error_warning_panel
            .add(input_code_waiting_error_warningLab);
        JPanel message_panel = new JPanel();
        final JLabel messageLab = new JLabel(err);
```

```
        message_panel.add(messageLab);
        JPanel input_panel = new JPanel();
        JButton input = new JButton("Input");
        input_panel.add(input);

         // Add Action to the button
        input.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                     Input_regNumber_Waiting backWindow = new
                        Input_regNumber_Waiting("Input_regNumber_Waiting");
                    backWindow.setSize(400, 300);
                    backWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    backWindow.setLocationRelativeTo(null);
                    backWindow.setVisible(true);
                    dispose();
            }
        });
        // Set Layout Manager
        setLayout(new GridLayout(3, 0));

        // Add Panels to the Window
        this.add(input_code_waiting_error_warning_panel);
        this.add(message_panel);
        this.add(input_panel);
        }
    } // END OF CLASS
```

## Input_regNumber_Waiting

Listing B.15: *Input_regNumber_Waiting.java*

```
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Input_regNumber_Waiting extends JFrame {
    public Input_regNumber_Waiting(String winTitle) {
        super(winTitle);

        // Create Swing Components
        JPanel message_panel = new JPanel();
        final JLabel messageLab = new JLabel("Input your Data");
        message_panel.add(messageLab);
        JPanel regNumber_panel = new JPanel();
        JLabel regNumber_label = new JLabel("regNumber");
        final JTextField regNumberTxt = new JTextField(30);
        regNumber_panel.add(regNumber_label);
        regNumber_panel.add(regNumberTxt);
        JPanel input_panel = new JPanel();
        JButton input = new JButton("Input");
        input_panel.add(input);

         // Add Action to the button
        input.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {

                    String regNumber = regNumberTxt.getText();

                 if((regNumberTxt.getText().isEmpty())) {
                     Input_regNumber_Waiting_Error nextErrWindow = new
                        Input_regNumber_Waiting_Error("Input_regNumber_Waiting_Error ",
                         "Invalid Input. Please Try Again" );
                    nextErrWindow.setSize(400, 300);
                    nextErrWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    nextErrWindow.setLocationRelativeTo(null);
                    nextErrWindow.setVisible(true);
                    dispose();
                     }
                 else {
                     Input_code_Waiting nextWindow = new
                        Input_code_Waiting("Input_code_Waiting", regNumber);
                     nextWindow.setSize(400, 300);
                     nextWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                     nextWindow.setLocationRelativeTo(null);
                     nextWindow.setVisible(true);
                     dispose();
                     }
            }
        });
        // Set Layout Manager
```

```
        setLayout(new GridLayout(3, 0));

        // Add Panels to the Window
        this.add(message_panel);
        this.add(regnumber_panel);
        this.add(input_panel);
        }
    } // END OF CLASS
```

## Input_regNumber_Waiting_Error

Listing B.16: *Input_regNumber_Waiting_Error.java*

```
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Input_regNumber_Waiting_Error extends JFrame {
    public Input_regNumber_Waiting_Error(String winTitle, String err) {
        super(winTitle);

        // Create Swing Components
        JPanel input_regnumber_waiting_error_warning_panel = new JPanel();
        final JLabel input_regnumber_waiting_error_warningLab = new JLabel();
        input_regnumber_waiting_error_warning_panel
            .add(input_regnumber_waiting_error_warningLab);
        JPanel message_panel = new JPanel();
        final JLabel messageLab = new JLabel(err);
        message_panel.add(messageLab);
        JPanel initialise_panel = new JPanel();
        JButton initialise = new JButton("initialise");
        initialise_panel.add(initialise);

         // Add Action to the button
        initialise.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                     Input_regNumber_Waiting backWindow = new
                        Input_regNumber_Waiting("Input_regNumber_Waiting");
                    backWindow.setSize(400, 300);
                    backWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    backWindow.setLocationRelativeTo(null);
                    backWindow.setVisible(true);
                    dispose();
            }
        });

        // Set Layout Manager
        setLayout(new GridLayout(3, 0));

        // Add Panels to the Window
        this.add(input_regnumber_waiting_error_warning_panel);
        this.add(message_panel);
        this.add(initialise_panel);
        }
    } // END OF CLASS
```

## Create_Enrollment_Ready

Listing B.17: *Create_Enrollment_Ready.java*

```
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Create_Enrollment_Ready extends JFrame {
```

```java
    public Create_Enrollment_Ready(String winTitle, final String regNumber,
         final String code) {
        super(winTitle);

        // Create Swing Components
        JPanel regnumber_panel = new JPanel();
        final JLabel regnumberLab = new JLabel();
        regnumber_panel.add(regnumberLab);
        JPanel code_panel = new JPanel();
        final JLabel codeLab = new JLabel();
        code_panel.add(codeLab);
        JPanel message_panel = new JPanel();
        final JLabel messageLab = new JLabel("Are you sure to"+
            " proceed Creating ");
        message_panel.add(messageLab);
        JPanel create_panel = new JPanel();
        JButton create = new JButton("Create");
        create_panel.add(create);

         // Add Action to the button
        create.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                    boolean recordInserted = false;

                // Declare Connection properties
                Connection conn = null;
                Statement stmt = null;
                int recAffected = 0;

                // Declare Connection properties
                try {
                    Class.forName("com.mysql.jdbc.Driver").newInstance();
                    String connectionUrl = "jdbc:mysql://localhost:3306/testdb1";
                    String connectionUser = "root";
                    String connectionPassword = "";
                    conn = DriverManager.getConnection(connectionUrl,
                        connectionUser, connectionPassword);
                    stmt = conn.createStatement();

                    // Execute a query statement or a stored procedure
                    CallableStatement procnone = conn.prepareCall("{call createEnrollment(?, ?)}");

                    // Sets the input parameter
                    procnone.setInt(1, Integer.parseInt(regNumber));
                    procnone.setString(2, code);

                    // Registers the out parameters
                    recAffected =  procnone.executeUpdate();

                    // Check if there is an updated record
                    if (recAffected > 0)
                        recordInserted = true;
                    else
                        recordInserted = false;

                 } catch (Exception e1) {
                     e1.printStackTrace();
                 } finally {
                     try { if (stmt != null) stmt.close(); } catch (SQLException e1)
                        { e1.printStackTrace(); }
                     try { if (conn != null) conn.close(); } catch (SQLException e1)
                        { e1.printStackTrace(); }
                 }

              if(recordInserted) {
                    JOptionPane.showMessageDialog(null, "The business task"+
                        " is completed successfully.");
                    System.exit(0);
              }
              else {
                     Create_Enrollment_Ready_Error nextErrWindow = new
                        Create_Enrollment_Ready_Error("Create_Enrollment_Ready_Error ",
                            "Can not be Created. Please Try Again" );
                    nextErrWindow.setSize(400, 300);
                    nextErrWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    nextErrWindow.setLocationRelativeTo(null);
                    nextErrWindow.setVisible(true);
                    dispose();
              }
             }
        });
        // Set Layout Manager
        setLayout(new GridLayout(4, 0));

        // Add Panels to the Window
        this.add(regnumber_panel);
        this.add(code_panel);
        this.add(message_panel);
        this.add(create_panel);
        }
    } // END OF CLASS
```

## Create_Enrollment_Ready_Error

Listing B.18: *Create_Enrollment_Ready_Error.java*

```
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class Create_Enrollment_Ready_Error extends JFrame {
    public Create_Enrollment_Ready_Error(String winTitle, String err) {
        super(winTitle);

        // Create Swing Components
        JPanel create_enrollment_ready_error_warning_panel = new JPanel();
        final JLabel create_enrollment_ready_error_warningLab = new JLabel();
        create_enrollment_ready_error_warning_panel
            .add(create_enrollment_ready_error_warningLab);
        JPanel message_panel = new JPanel();
        final JLabel messageLab = new JLabel(err);
        message_panel.add(messageLab);
        JPanel exit_panel = new JPanel();
        JButton exit = new JButton("exit");
        exit_panel.add(exit);

         // Add Action to the button
        exit.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                     Input_regNumber_Waiting backWindow = new
                        Input_regNumber_Waiting("Input_regNumber_Waiting");
                    backWindow.setSize(400, 300);
                    backWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    backWindow.setLocationRelativeTo(null);
                    backWindow.setVisible(true);
                    dispose();
            }
        });
        // Set Layout Manager
        setLayout(new GridLayout(3, 0));

        // Add Panels to the Window
        this.add(create_enrollment_ready_error_warning_panel);
        this.add(message_panel);
        this.add(exit_panel);
        }
    } // END OF CLASS
```

# C

# Appendix: Models and MySQL Scripts of Experiment (3)

## C.1 Complete Models and Full results of Experiment (3)

This Appendix presents the XML representations of all $\mu$ML models In The University Administration System case study, discussed previously in chapter 10.

### Experiment (3a): Information Model Construction

Listing C.1: CaseStudy3.java

```java
package mde.example;

import java.io.File;
import java.io.IOException;
import mde.data.model.DDiagram;
import mde.database.gen.DumpFileGenerator;
import mde.mysql.gen.MySQLDumpFileGenerator;
import mde.database.gen.TreeException;
import mde.dbs.model.Schema;
import mde.dm2schem.rule.DDiagramToSchemaOnly;
import mde.inf2dm.rule.InfDiagramToDDiagram;
import mde.information.model.Association;
import mde.information.model.Composition;
import mde.information.model.Entity;
import mde.information.model.Attribute;
import mde.information.model.Generalisation;
import mde.information.model.Role;
import mde.model.Type;
import org.jast.ast.ASTWriter;

public class CaseStudy4 {

    public static void main(String[] args) throws IOException, TreeException,
        mde.gui.gen.TreeException {
        // TODO Auto-generated method stub

        // Construct the Information Model
        mde.information.model.Diagram informationModel = new
            mde.information.model.Diagram();

        Entity personEntity = new Entity("Person");
        Attribute attr2 = new Attribute("name", new Type("String"));
        Attribute attr3 = new Attribute("age", new Type("String"))
            .setLowerbound(18).setUpperbound(35);
        Attribute attr4 = new Attribute("gender", new Type("String"));
        personEntity.addAttribute(attr2);
        personEntity.addAttribute(attr3);
        personEntity.addAttribute(attr4);
        Entity studentEntity = new Entity("Student");
```

```
        Attribute attr5 = new Attribute("regNumber", new Type("Integer")).setIdentifier(true);
        Attribute attr7 = new Attribute("username", new Type("String"));
        Attribute attr8 = new Attribute("password", new Type("String"));
        studentEntity.addAttribute(attr5);
        studentEntity.addAttribute(attr7);
        studentEntity.addAttribute(attr8);
        Entity staffEntity = new Entity("Staff");
        Attribute attr16 = new Attribute("empNumber", new Type("Integer")).setIdentifier(true);
        Attribute attr17 = new Attribute("salary", new Type("Double"));
        staffEntity.addAttribute(attr16);
        staffEntity.addAttribute(attr17);
        Entity addressEntity = new Entity("Address");
        Attribute attr18 = new Attribute("postcode", new Type("String"));
        Attribute attr19 = new Attribute("street", new Type("String"));
        Attribute attr20 = new Attribute("city", new Type("String"));
        addressEntity.addAttribute(attr18);
        addressEntity.addAttribute(attr19);
        addressEntity.addAttribute(attr20);
        Entity moduleEntity = new Entity("Module");
        Attribute attr12 = new Attribute("code", new Type("Integer")).setIdentifier(true);
        Attribute attr13 = new Attribute("title", new Type("String"));
        Attribute attr14 = new Attribute("credit", new Type("Integer"));
        Attribute attr15 = new Attribute("desc", new Type("String"));
        moduleEntity.addAttribute(attr12);
        moduleEntity.addAttribute(attr13);
        moduleEntity.addAttribute(attr14);
        moduleEntity.addAttribute(attr15);
        Entity examEntity = new Entity("Exam");
        Attribute attr22 = new Attribute("id", new Type("String")).setIdentifier(true);
        Attribute attr27 = new Attribute("practical", new Type("Boolean"));
        Attribute attr24 = new Attribute("value", new Type("Integer"));
        Attribute attr25 = new Attribute("date", new Type("Date"));
        examEntity.addAttribute(attr22);
        examEntity.addAttribute(attr27);
        examEntity.addAttribute(attr24);
        examEntity.addAttribute(attr25);
        Entity projectEntity = new Entity("Project");
        Attribute attr31 = new Attribute("id", new Type("String")).setIdentifier(true);
        Attribute attr23 = new Attribute("title", new Type("String"));
        Attribute attr28 = new Attribute("group", new Type("Boolean"));
        Attribute attr29 = new Attribute("value", new Type("Integer"));
        Attribute attr30 = new Attribute("deadline", new Type("Date"));
        projectEntity.addAttribute(attr31);
        projectEntity.addAttribute(attr23);
        projectEntity.addAttribute(attr28);
        projectEntity.addAttribute(attr29);
        projectEntity.addAttribute(attr30);

        informationModel.addEntity(studentEntity);
        informationModel.addEntity(staffEntity);
        informationModel.addEntity(personEntity);
        informationModel.addEntity(moduleEntity);
        informationModel.addEntity(examEntity);
        informationModel.addEntity(addressEntity);
        informationModel.addEntity(projectEntity);

        informationModel.addAssociation(new Association().addName("Enrollment")
        .addRole(new Role("module", informationModel
            .getEntity("Module")).setMultiple(true))
        .addRole(new Role("student", informationModel
            .getEntity("Student")).setMultiple(true)));
        informationModel.addAssociation(new Association()
        .addRole(new Role("address", informationModel
            .getEntity("Address")).setMultiple(true))
        .addRole(new Role("student", informationModel
            .getEntity("Student")).setMultiple(false)));
        informationModel.addGeneralisation(new Generalisation().setDisjoint(false)
        .addRole(new Role("person", informationModel
            .getEntity("Person")).setGeneral(true))
        .addRole(new Role("student", informationModel
            .getEntity("Student")).setGeneral(false)));
        informationModel.addGeneralisation(new Generalisation().setDisjoint(false)
        .addRole(new Role("person", informationModel
            .getEntity("Person")).setGeneral(true))
        .addRole(new Role("staff", informationModel
            .getEntity("Staff")).setGeneral(false)));
        informationModel.addComposition((new Composition()
            .setTotal(false).addRole(new Role("module",
                informationModel.getEntity("Module")).setWhole(true))
        .addRole(new Role("exam", informationModel.getEntity("Exam")).setWhole(false))));
        informationModel.addComposition((new Composition()
            .setTotal(true).addRole(new Role("module",
                informationModel.getEntity("Module")).setWhole(true))
        .addRole(new Role("project", informationModel
            .getEntity("Project")).setWhole(false))));

        ASTWriter writer2 = new ASTWriter(new File("Uni_informationModel.xml"));
        writer2.usePackage("mde.information.model", "xmlns:inf");
        writer2.writeDocument(informationModel);
        writer2.close();

        System.out.println("(3) Information Model is Created by user.");
        // ------------------------------ //

```

```
        //Generate Data Model
        InfDiagramToDDiagram topRule4 = new InfDiagramToDDiagram();
        DDiagram  dataModel = topRule4.translate(informationModel);

        ASTWriter writer4 = new ASTWriter(new File("Uni_DataModel.xml"));
        writer4.usePackage("mde.data.model", "xmlns:data");
        writer4.usePackage("mde.model", "xmlns:mod");
        writer4.writeDocument(dataModel);
        writer4.close();

        System.out.println("(5) Data Model is Created (Information"+
            "Model --> Data Model).");
        // ---------------------------- //

        //generate Database Schema Model (Detailed_DFD"+
            " + Data Model ---> Schema)
        DDiagramToSchemaOnly topRule5 = new DDiagramToSchemaOnly();
        Schema schemaModel = topRule5.translate(null, dataModel);

        System.out.println(schemaModel.getName());

        ASTWriter writer5 = new ASTWriter(new File("Uni_SchemaModel.xml"));
        writer5.usePackage("mde.dbs.model", "xmlns:dbs");
        writer5.usePackage("mde.model", "xmlns:mod");
        writer5.writeDocument(schemaModel);
        writer5.close();

        System.out.println("(9) Database Schema Model is Created"+
            " (DataFlow + Data Model --> Database Schema Model).");

        // code generation
        DumpFileGenerator MySQLGenerator = new
            MySQLDumpFileGenerator(schemaModel);
        System.out.println("calling generate in mysql"+
            " package generator");
        MySQLGenerator.generate();
        System.out.println("Finished generating MySQL Schema OK");
    }
}
```

# Information Model Representation (3a)

Listing C.2: *Uni_informationModel.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<inf:Diagram xmlns:inf="mde.information.model" id="0">
  <inf:Entity id="1" name="Student">
    <inf:Attribute id="2" name="regNumber" identifier="true" size="0">
      <Type id="3" name="Integer"/>
    </inf:Attribute>
    <inf:Attribute id="4" name="username" size="0">
      <Type id="5" name="String"/>
    </inf:Attribute>
    <inf:Attribute id="6" name="password" size="0">
      <Type id="7" name="String"/>
    </inf:Attribute>
  </inf:Entity>
  <inf:Entity id="8" name="Staff">
    <inf:Attribute id="9" name="empNumber" identifier="true" size="0">
      <Type id="10" name="Integer"/>
    </inf:Attribute>
    <inf:Attribute id="11" name="salary" size="0">
      <Type id="12" name="Double"/>
    </inf:Attribute>
  </inf:Entity>
  <inf:Entity id="13" name="Person">
    <inf:Attribute id="14" name="name" size="0">
      <Type id="15" name="String"/>
    </inf:Attribute>
    <inf:Attribute id="16" name="age" upperbound="35" lowerbound="18" size="0">
      <Type id="17" name="String"/>
    </inf:Attribute>
    <inf:Attribute id="18" name="gender" size="0">
      <Type id="19" name="String"/>
    </inf:Attribute>
  </inf:Entity>
  <inf:Entity id="20" name="Module">
    <inf:Attribute id="21" name="code" identifier="true" size="0">
      <Type id="22" name="Integer"/>
    </inf:Attribute>
    <inf:Attribute id="23" name="title" size="0">
      <Type id="24" name="String"/>
    </inf:Attribute>
    <inf:Attribute id="25" name="credit" size="0">
      <Type id="26" name="Integer"/>
    </inf:Attribute>
    <inf:Attribute id="27" name="desc" size="0">
      <Type id="28" name="String"/>
```

```xml
      </inf:Attribute>
    </inf:Entity>
    <inf:Entity id="29" name="Assessment">
      <inf:Attribute id="30" name="id" identifier="true" size="0">
        <Type id="31" name="String"/>
      </inf:Attribute>
      <inf:Attribute id="32" name="title" size="0">
        <Type id="33" name="String"/>
      </inf:Attribute>
      <inf:Attribute id="34" name="group" size="0">
        <Type id="35" name="Boolean"/>
      </inf:Attribute>
    </inf:Entity>
    <inf:Entity id="36" name="Exam">
      <inf:Attribute id="37" name="date" size="0">
        <Type id="38" name="Date"/>
      </inf:Attribute>
      <inf:Attribute id="39" name="id" identifier="true" size="0">
        <Type id="40" name="String"/>
      </inf:Attribute>
    </inf:Entity>
    <inf:Entity id="41" name="Address">
      <inf:Attribute id="42" name="postcode" size="0">
        <Type id="43" name="String"/>
      </inf:Attribute>
      <inf:Attribute id="44" name="street" size="0">
        <Type id="45" name="String"/>
      </inf:Attribute>
      <inf:Attribute id="46" name="city" size="0">
        <Type id="47" name="String"/>
      </inf:Attribute>
    </inf:Entity>
    <inf:Entity id="48" name="Project">
      <inf:Attribute id="49" name="value" size="0">
        <Type id="50" name="Integer"/>
      </inf:Attribute>
      <inf:Attribute id="51" name="deadline" size="0">
        <Type id="52" name="Date"/>
      </inf:Attribute>
    </inf:Entity>
    <inf:Generalisation id="53">
      <inf:Role id="54" name="person" multiple="false" general="true">
        <inf:Entity ref="13"/>
      </inf:Role>
      <inf:Role id="55" name="student" multiple="false">
        <inf:Entity ref="1"/>
      </inf:Role>
    </inf:Generalisation>
    <inf:Generalisation id="56">
      <inf:Role id="57" name="person" multiple="false" general="true">
        <inf:Entity ref="13"/>
      </inf:Role>
      <inf:Role id="58" name="staff" multiple="false">
        <inf:Entity ref="8"/>
      </inf:Role>
    </inf:Generalisation>
    <inf:Generalisation id="59">
      <inf:Role id="60" name="assessment" multiple="false" general="true">
        <inf:Entity ref="29"/>
      </inf:Role>
      <inf:Role id="61" name="exam" multiple="false">
        <inf:Entity ref="36"/>
      </inf:Role>
    </inf:Generalisation>
    <inf:Generalisation id="62">
      <inf:Role id="63" name="assessment" multiple="false" general="true">
        <inf:Entity ref="29"/>
      </inf:Role>
      <inf:Role id="64" name="project" multiple="false">
        <inf:Entity ref="48"/>
      </inf:Role>
    </inf:Generalisation>
    <inf:Composition id="65">
      <inf:Role id="66" name="module" multiple="false" whole="true">
        <inf:Entity ref="20"/>
      </inf:Role>
      <inf:Role id="67" name="assessment" multiple="false">
        <inf:Entity ref="29"/>
      </inf:Role>
    </inf:Composition>
    <inf:Association id="68">
      <inf:Role id="69" name="module" multiple="true">
        <inf:Entity ref="20"/>
      </inf:Role>
      <inf:Role id="70" name="student" multiple="true">
        <inf:Entity ref="1"/>
      </inf:Role>
    </inf:Association>
    <inf:Association id="71">
      <inf:Role id="72" name="address" multiple="true">
        <inf:Entity ref="41"/>
      </inf:Role>
      <inf:Role id="73" name="person" multiple="false">
        <inf:Entity ref="13"/>
```

```
    </inf:Role>
  </inf:Association>
</inf:Diagram>
```

## Data Dependency Model Representation (3a)

Listing C.3: *Uni_DataModel.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<data:DDiagram xmlns:data="mde.data.model" id="0">
  <data:DEntity id="1" name="Person_Student">
    <data:DAttribute id="2" name="name" size="30">
      <mod:Type xmlns:mod="mde.model" id="3" name="String"/>
    </data:DAttribute>
    <data:DAttribute id="4" name="age" upperbound="35" lowerbound="18" size="30">
      <mod:Type id="5" name="String"/>
    </data:DAttribute>
    <data:DAttribute id="6" name="gender" size="30">
      <mod:Type id="7" name="String"/>
    </data:DAttribute>
    <data:DAttribute id="8" name="regNumber" unique="true" size="10">
      <mod:Type id="9" name="Integer"/>
    </data:DAttribute>
    <data:DAttribute id="10" name="username" unique="true" size="30">
      <mod:Type id="11" name="String"/>
    </data:DAttribute>
    <data:DAttribute id="12" name="password" unique="true" size="30">
      <mod:Type id="13" name="String"/>
    </data:DAttribute>
  </data:DEntity>
  <data:DEntity id="14" name="Person_Staff">
    <data:DAttribute id="15" name="name" size="30">
      <mod:Type ref="3"/>
    </data:DAttribute>
    <data:DAttribute id="16" name="age" upperbound="35" lowerbound="18" size="30">
      <mod:Type ref="5"/>
    </data:DAttribute>
    <data:DAttribute id="17" name="gender" size="30">
      <mod:Type ref="7"/>
    </data:DAttribute>
    <data:DAttribute id="18" name="empNumber" unique="true" size="10">
      <mod:Type id="19" name="Integer"/>
    </data:DAttribute>
    <data:DAttribute id="20" name="salary" unique="true" size="1">
      <mod:Type id="21" name="Double"/>
    </data:DAttribute>
  </data:DEntity>
  <data:DEntity id="22" name="Assessment_Exam">
    <data:DAttribute id="23" name="id" unique="true" size="30">
      <mod:Type id="24" name="String"/>
    </data:DAttribute>
    <data:DAttribute id="25" name="title" size="30">
      <mod:Type id="26" name="String"/>
    </data:DAttribute>
    <data:DAttribute id="27" name="group" size="1">
      <mod:Type id="28" name="Boolean"/>
    </data:DAttribute>
    <data:DAttribute id="29" name="date" unique="true" size="1">
      <mod:Type id="30" name="Date"/>
    </data:DAttribute>
  </data:DEntity>
  <data:DEntity id="31" name="Assessment_Project">
    <data:DAttribute id="32" name="id" identifier="true" size="30">
      <mod:Type id="33" name="String"/>
    </data:DAttribute>
    <data:DAttribute id="34" name="title" size="30">
      <mod:Type ref="26"/>
    </data:DAttribute>
    <data:DAttribute id="35" name="group" size="1">
      <mod:Type ref="28"/>
    </data:DAttribute>
    <data:DAttribute id="36" name="value" unique="true" size="10">
      <mod:Type id="37" name="Integer"/>
    </data:DAttribute>
    <data:DAttribute id="38" name="deadline" unique="true" size="1">
      <mod:Type id="39" name="Date"/>
    </data:DAttribute>
  </data:DEntity>
  <data:DEntity id="40" name="Enrollment">
    <data:DAttribute id="41" name="code" unique="true" size="10">
      <mod:Type id="42" name="Integer"/>
    </data:DAttribute>
    <data:DAttribute id="43" name="regNumber" unique="true" size="10">
      <mod:Type ref="9"/>
    </data:DAttribute>
  </data:DEntity>
  <data:DEntity id="44" name="Module">
    <data:DAttribute id="45" name="code" identifier="true" size="10">
```

```
      <mod:Type ref="42"/>
    </data:DAttribute>
    <data:DAttribute id="46" name="title" size="30">
      <mod:Type id="47" name="String"/>
    </data:DAttribute>
    <data:DAttribute id="48" name="credit" size="10">
      <mod:Type id="49" name="Integer"/>
    </data:DAttribute>
    <data:DAttribute id="50" name="desc" size="30">
      <mod:Type id="51" name="String"/>
    </data:DAttribute>
  </data:DEntity>
  <data:DEntity id="52" name="Address">
    <data:DAttribute id="53" name="postcode" size="30">
      <mod:Type id="54" name="String"/>
    </data:DAttribute>
    <data:DAttribute id="55" name="identity" identifier="true" unique="true" size="10">
      <mod:Type id="56" name="Integer"/>
    </data:DAttribute>
    <data:DAttribute id="57" name="street" size="30">
      <mod:Type id="58" name="String"/>
    </data:DAttribute>
    <data:DAttribute id="59" name="city" size="30">
      <mod:Type id="60" name="String"/>
    </data:DAttribute>
  </data:DEntity>
  <data:DDependency id="64">
    <data:DRole id="65" name="assessment_project" multiple="true">
      <data:DEntity ref="31"/>
    </data:DRole>
    <data:DRole id="66" name="module">
      <data:DEntity ref="44"/>
    </data:DRole>
  </data:DDependency>
  <data:DDependency id="67">
    <data:DRole id="68" name="enrollment" multiple="true">
      <data:DEntity ref="40"/>
    </data:DRole>
    <data:DRole id="69" name="module">
      <data:DEntity ref="44"/>
    </data:DRole>
  </data:DDependency>
  <data:DDependency id="67">
    <data:DRole id="68" name="enrollment" multiple="true">
      <data:DEntity ref="40"/>
    </data:DRole>
    <data:DRole id="69" name="module">
      <data:DEntity ref="44"/>
    </data:DRole>
  </data:DDependency>
  <data:DDependency id="70">
    <data:DRole id="71" name="enrollment" multiple="true">
      <data:DEntity ref="40"/>
    </data:DRole>
    <data:DRole id="72" name="student">
      <data:DEntity ref="1"/>
    </data:DRole>
  </data:DDependency>
  <data:DDependency id="73">
    <data:DRole id="74" name="address">
      <data:DEntity ref="52"/>
    </data:DRole>
    <data:DRole id="75" name="person">
      <data:DEntity ref="14"/>
    </data:DRole>
  </data:DDependency>
</data:DDiagram>
```

## Database and Query (DBQ) Model Representation (3a)

Listing C.4: *Uni_SchemaModel.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<dbs:Schema xmlns:dbs="mde.dbs.model" id="0" name="database">
  <dbs:Table id="1" name="Person_Student">
    <dbs:Column id="2" name="identity" automatic="true" size="10">
      <mod:Type xmlns:mod="mde.model" id="3" name="INTEGER"/>
    </dbs:Column>
    <dbs:Column id="4" name="name" size="30">
      <mod:Type id="5" name="VARCHAR"/>
    </dbs:Column>
    <dbs:Column id="6" name="age" upperbound="35" lowerbound="18" size="30">
      <mod:Type id="7" name="VARCHAR"/>
    </dbs:Column>
    <dbs:Column id="8" name="gender" size="30">
      <mod:Type id="9" name="VARCHAR"/>
    </dbs:Column>
    <dbs:Column id="10" name="regNumber" size="10">
```

```
        <mod:Type id="11" name="INTEGER"/>
      </dbs:Column>
      <dbs:Column id="12" name="username" size="30">
        <mod:Type id="13" name="VARCHAR"/>
      </dbs:Column>
      <dbs:Column id="14" name="password" size="30">
        <mod:Type id="15" name="VARCHAR"/>
      </dbs:Column>
      <dbs:PrimaryKey id="16">
        <dbs:Column ref="2"/>
      </dbs:PrimaryKey>
    </dbs:Table>
    <dbs:Table id="17" name="Person_Staff">
      <dbs:Column id="18" name="identity" automatic="true" size="10">
        <mod:Type id="19" name="INTEGER"/>
      </dbs:Column>
      <dbs:Column id="20" name="name" size="30">
        <mod:Type ref="5"/>
      </dbs:Column>
      <dbs:Column id="21" name="age" upperbound="35" lowerbound="18" size="30">
        <mod:Type ref="7"/>
      </dbs:Column>
      <dbs:Column id="22" name="gender" size="30">
        <mod:Type ref="9"/>
      </dbs:Column>
      <dbs:Column id="23" name="empNumber" size="10">
        <mod:Type id="24" name="INTEGER"/>
      </dbs:Column>
      <dbs:Column id="25" name="salary" size="5">
        <mod:Type id="26" name="DOUBLE"/>
      </dbs:Column>
      <dbs:Column id="27" name="Address.identity" size="10">
        <mod:Type id="28" name="INTEGER"/>
      </dbs:Column>
      <dbs:ForeignKey id="29">
        <dbs:Column ref="27"/>
        <dbs:Table id="30" name="Address">
          <dbs:Column id="31" name="identity" size="10">
            <mod:Type ref="28"/>
          </dbs:Column>
          <dbs:Column id="32" name="postcode" size="30">
            <mod:Type id="33" name="VARCHAR"/>
          </dbs:Column>
          <dbs:Column id="34" name="street" size="30">
            <mod:Type id="35" name="VARCHAR"/>
          </dbs:Column>
          <dbs:Column id="36" name="city" size="30">
            <mod:Type id="37" name="VARCHAR"/>
          </dbs:Column>
          <dbs:PrimaryKey id="38">
            <dbs:Column ref="31"/>
          </dbs:PrimaryKey>
        </dbs:Table>
      </dbs:ForeignKey>
      <dbs:PrimaryKey id="39">
        <dbs:Column ref="18"/>
      </dbs:PrimaryKey>
    </dbs:Table>
    <dbs:Table id="40" name="Assessment_Exam">
      <dbs:Column id="41" name="identity" automatic="true" size="10">
        <mod:Type id="42" name="INTEGER"/>
      </dbs:Column>
      <dbs:Column id="43" name="id" size="30">
        <mod:Type id="44" name="VARCHAR"/>
      </dbs:Column>
      <dbs:Column id="45" name="title" size="30">
        <mod:Type id="46" name="VARCHAR"/>
      </dbs:Column>
      <dbs:Column id="47" name="group" size="5">
        <mod:Type id="48" name="BOOLEAN"/>
      </dbs:Column>
      <dbs:Column id="49" name="date" size="5">
        <mod:Type id="50" name="DATE"/>
      </dbs:Column>
      <dbs:Column id="51" name="Module.code" size="10">
        <mod:Type id="52" name="INTEGER"/>
      </dbs:Column>
      <dbs:ForeignKey id="53">
        <dbs:Column ref="51"/>
        <dbs:Table id="54" name="Module">
          <dbs:Column id="55" name="code" size="10">
            <mod:Type ref="52"/>
          </dbs:Column>
          <dbs:Column id="56" name="title" size="30">
            <mod:Type id="57" name="VARCHAR"/>
          </dbs:Column>
          <dbs:Column id="58" name="credit" size="10">
            <mod:Type id="59" name="INTEGER"/>
          </dbs:Column>
          <dbs:Column id="60" name="desc" size="30">
            <mod:Type id="61" name="VARCHAR"/>
          </dbs:Column>
          <dbs:PrimaryKey id="62">
            <dbs:Column ref="55"/>
```

```
          </dbs:PrimaryKey>
        </dbs:Table>
      </dbs:ForeignKey>
      <dbs:PrimaryKey id="63">
        <dbs:Column ref="41"/>
      </dbs:PrimaryKey>
    </dbs:Table>
    <dbs:Table id="64" name="Assessment_Project">
      <dbs:Column id="65" name="id" size="30">
        <mod:Type id="66" name="VARCHAR"/>
      </dbs:Column>
      <dbs:Column id="67" name="title" size="30">
        <mod:Type ref="46"/>
      </dbs:Column>
      <dbs:Column id="68" name="group" size="5">
        <mod:Type ref="48"/>
      </dbs:Column>
      <dbs:Column id="69" name="value" size="10">
        <mod:Type id="70" name="INTEGER"/>
      </dbs:Column>
      <dbs:Column id="71" name="deadline" size="5">
        <mod:Type id="72" name="DATE"/>
      </dbs:Column>
      <dbs:Column id="73" name="Module.code" size="10">
        <mod:Type ref="52"/>
      </dbs:Column>
      <dbs:ForeignKey id="74">
        <dbs:Column ref="73"/>
        <dbs:Table ref="54"/>
      </dbs:ForeignKey>
      <dbs:PrimaryKey id="75">
        <dbs:Column ref="65"/>
      </dbs:PrimaryKey>
    </dbs:Table>
    <dbs:Table id="76" name="Enrollment">
      <dbs:Column id="77" name="identity" automatic="true" size="10">
        <mod:Type id="78" name="INTEGER"/>
      </dbs:Column>
      <dbs:Column id="79" name="code" size="10">
        <mod:Type ref="52"/>
      </dbs:Column>
      <dbs:Column id="80" name="regNumber" size="10">
        <mod:Type ref="11"/>
      </dbs:Column>
      <dbs:Column id="81" name="Module.code" size="10">
        <mod:Type ref="52"/>
      </dbs:Column>
      <dbs:Column id="82" name="Person_Student.identity" size="10">
        <mod:Type ref="3"/>
      </dbs:Column>
      <dbs:ForeignKey id="83">
        <dbs:Column ref="81"/>
        <dbs:Table ref="54"/>
      </dbs:ForeignKey>
      <dbs:ForeignKey id="84">
        <dbs:Column ref="82"/>
        <dbs:Table ref="1"/>
      </dbs:ForeignKey>
      <dbs:PrimaryKey id="85">
        <dbs:Column ref="77"/>
      </dbs:PrimaryKey>
    </dbs:Table>
    <dbs:Table ref="54"/>
    <dbs:Table ref="30"/>
  </dbs:Schema>
```

## Executable MySQL Code (3a)

Listing C.5: *database_MySQL.sql*

```
-- Database Creation

CREATE DATABASE sysDatabase;
USE sysDatabase;

-- Structure for table 'Person_Student'

CREATE TABLE Person_Student (
  identity INT(10) NOT NULL,
  name VARCHAR(30),
  age VARCHAR(30),
  gender VARCHAR(30),
  regNumber INT(10),
  username VARCHAR(30),
  password VARCHAR(30),
 PRIMARY KEY(identity));

-- Structure for table 'Person_Staff'
```

```

CREATE TABLE Person_Staff (
   identity INT(10) NOT NULL,
   name VARCHAR(30),
   age VARCHAR(30),
   gender VARCHAR(30),
   empNumber INT(10),
   salary DOUBLE(5),
   Address.identity INT(10) NOT NULL,
  PRIMARY KEY(identity),
  FOREIGN KEY(Address.identity) REFERENCES Address(identity));

-- Structure for table 'Assessment_Exam'

CREATE TABLE Assessment_Exam (
   identity INT(10) NOT NULL,
   id VARCHAR(30),
   title VARCHAR(30),
   group BOOLEAN(5),
   date DATE(5),
   Module.code INT(10),
  PRIMARY KEY(identity),
  FOREIGN KEY(Module.code) REFERENCES Module(code));

-- Structure for table 'Assessment_Project'

CREATE TABLE Assessment_Project (
   id VARCHAR(30) NOT NULL,
   title VARCHAR(30),
   group BOOLEAN(5),
   value INT(10),
   deadline DATE(5),
   Module.code INT(10),
  PRIMARY KEY(id),
  FOREIGN KEY(Module.code) REFERENCES Module(code));

-- Structure for table 'Enrollment'

CREATE TABLE Enrollment (
   identity INT(10) NOT NULL,
   code INT(10),
   regNumber INT(10),
   Module.code INT(10),
   Person_Student.identity INT(10) NOT NULL,
  PRIMARY KEY(identity),
  FOREIGN KEY(Module.code) REFERENCES Module(code),
  FOREIGN KEY(Person_Student.identity) REFERENCES Person_Student(identity));

-- Structure for table 'Module'

CREATE TABLE Module (
   code INT(10) NOT NULL,
   title VARCHAR(30),
   credit INT(10),
   desc VARCHAR(30),
  PRIMARY KEY(code));

-- Structure for table 'Address'

CREATE TABLE Address (
   identity INT(10) NOT NULL,
   postcode VARCHAR(30),
   street VARCHAR(30),
   city VARCHAR(30),
  PRIMARY KEY(identity));

-- Trigger: Applying Checking Constraints on table 'Person_Student'

DELIMITER //

DROP TRIGGER IF EXISTS 'person_studentCheck' //
CREATE TRIGGER 'person_studentCheck' BEFORE INSERT ON Person_Student
  FOR EACH ROW
    IF (NEW.age < 18 OR NEW.age > 35) THEN
      SET msg = 'INVALID DATA IN age';
      SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = msg;
    END IF;
//

-- Trigger: Applying Checking Constraints on table 'Person_Staff'

DELIMITER //

DROP TRIGGER IF EXISTS 'person_staffCheck' //
CREATE TRIGGER 'person_staffCheck' BEFORE INSERT ON Person_Staff
  FOR EACH ROW
    IF (NEW.age < 18 OR NEW.age > 35) THEN
      SET msg = 'INVALID DATA IN age';
      SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = msg;
    END IF;
//
```

## Experiment (3b): Impact Model Consturction

Listing C.6: CaseStudy12.java

```
package mde.example;

import java.io.File;
import java.io.IOException;
import mde.data.model.DDiagram;
import mde.database.gen.DumpFileGenerator;
import mde.database.gen.TreeException;
import mde.dbs.model.Schema;
import mde.dm2schem.rule.DDiagramToSchemaOnly;
import mde.impact.model.ImpBoundary;
import mde.impact.model.ImpConjunction;
import mde.impact.model.ImpCreateFlow;
import mde.impact.model.ImpDiagram;
import mde.impact.model.ImpObject;
import mde.impact.model.ImpReadFlow;
import mde.impact.model.ImpRole;
import mde.impact.model.ImpTask;
import mde.impact.model.ImpUpdateFlow;
import mde.impact2information.ImpDiagramToDiagram;
import mde.inf2dm.rule.InfDiagramToDDiagram;
import mde.information.model.Diagram;
import mde.mysql.gen.MySQLDumpFileGenerator;
import org.jast.ast.ASTWriter;

public class Full_example12 {
    public static void main(String[] args) throws IOException, TreeException,
        mde.gui.gen.TreeException {
        // TODO Auto-generated method stub

        // Construct the Impact Model
        ImpDiagram ImpactModel = new ImpDiagram();
        ImpBoundary impboundary = new ImpBoundary("Enrol_Boundary");
        ImpBoundary impboundary2 = new ImpBoundary("Set_Address_Boundary");
        ImpBoundary impboundary3 = new ImpBoundary("Manage_Assessment_Boundary");
        ImpTask impTask1 = new ImpTask("Enrol");
        ImpTask impTask2 = new ImpTask("Set Address");
        ImpTask impTask3 = new ImpTask("Set Assessment");
        ImpTask impTask4 = new ImpTask("Modify Value");
        ImpConjunction conj = new ImpConjunction("Set Address");
        ImpObject impObj1 = new ImpObject("Student");
        ImpObject impObj2 = new ImpObject("Staff");
        ImpObject impObj3 = new ImpObject("Enrollment");
        ImpObject impObj4 = new ImpObject("Address");
        ImpObject impObj5 = new ImpObject("Module");
        ImpObject impObj6 = new ImpObject("Assessment");

        // enrol boundary contents
        ImpReadFlow rf1 = new ImpReadFlow();
        ImpRole imprf1 = new ImpRole("enrol", impTask1);
        ImpRole imprf2 = new ImpRole("student", impObj1);
        rf1.addImpRole(imprf2);
        rf1.addImpRole(imprf1);
        ImpReadFlow rf2 = new ImpReadFlow();
        ImpRole imprf3 = new ImpRole("enrol", impTask1);
        ImpRole imprf4 = new ImpRole("module", impObj5);
        rf2.addImpRole(imprf4);
        rf2.addImpRole(imprf3);
        ImpCreateFlow cf = new ImpCreateFlow();
        ImpRole imprf5 = new ImpRole("enrol", impTask1);
        ImpRole imprf6 = new ImpRole("enrollment", impObj3);
        cf.addImpRole(imprf5);
        cf.addImpRole(imprf6);
        ImpReadFlow rf3 = new ImpReadFlow();
        ImpRole imprf7 = new ImpRole("withdraw", impTask2);
        ImpRole imprf8 = new ImpRole("student", impObj1);
        imprf8.setMultiple(true);
        rf3.addImpRole(imprf8);
        rf3.addImpRole(imprf7);

        impboundary.addImpTask(impTask1);
        impboundary.addImpObject(impObj1);
        impboundary.addImpObject(impObj5);
        impboundary.addImpObject(impObj3);
        impboundary.addImpReadFlow(rf1);
        impboundary.addImpReadFlow(rf2);
        impboundary.addImpCreateFlow(cf);

        // set address boundary contents

        ImpReadFlow rf11 = new ImpReadFlow();
        ImpRole imprf11 = new ImpRole("set_address_conj", conj);
        ImpRole imprf21 = new ImpRole("student", impObj1).setMultiple(false);
        rf11.addImpRole(imprf21);
        rf11.addImpRole(imprf11);
        ImpReadFlow rf21 = new ImpReadFlow();
        ImpRole imprf31 = new ImpRole("set_address_conj", conj);
        ImpRole imprf41 = new ImpRole("staff", impObj2).setMultiple(false);
        rf21.addImpRole(imprf41);
        rf21.addImpRole(imprf31);
```

```
        ImpReadFlow rf31 = new ImpReadFlow();
        ImpRole imprf81 = new ImpRole("set_address_conj", conj);
        ImpRole imprf71 = new ImpRole("set_address", impTask2);
        rf31.addImpRole(imprf81);
        rf31.addImpRole(imprf71);
        ImpCreateFlow cf1 = new ImpCreateFlow();
        ImpRole imprf51 = new ImpRole("set_address", impTask2);
        ImpRole imprf61 = new ImpRole("address", impObj4);
        cf1.addImpRole(imprf51);
        cf1.addImpRole(imprf61);

        impboundary2.addImpTask(impTask2);
        impboundary2.addImpObject(impObj1);
        impboundary2.addImpObject(impObj2);
        impboundary2.addImpObject(impObj4);
        impboundary2.addImpConjunction(conj);
        impboundary2.addImpReadFlow(rf11);
        impboundary2.addImpReadFlow(rf21);
        impboundary2.addImpReadFlow(rf31);
        impboundary2.addImpCreateFlow(cf1);

        // set assessment boundary contents

        ImpReadFlow rf22 = new ImpReadFlow();
        ImpRole imprf32 = new ImpRole("set_assess", impTask3);
        ImpRole imprf42 = new ImpRole("module", impObj5);
        rf22.addImpRole(imprf42);
        rf22.addImpRole(imprf32);
        ImpReadFlow rf32 = new ImpReadFlow();
        ImpRole imprf52 = new ImpRole("modify_value", impTask4);
        ImpRole imprf62 = new ImpRole("module", impObj5);
        rf32.addImpRole(imprf52);
        rf32.addImpRole(imprf62);
        ImpCreateFlow cf11 = new ImpCreateFlow();
        ImpRole imprf53 = new ImpRole("set_assess", impTask3);
        ImpRole imprf63 = new ImpRole("assessment", impObj6);
        cf11.addImpRole(imprf53);
        cf11.addImpRole(imprf63);
        ImpUpdateFlow uf = new ImpUpdateFlow();
        ImpRole imprf54 = new ImpRole("set_assess", impTask3);
        ImpRole imprf64 = new ImpRole("assessment", impObj6);
        uf.addImpRole(imprf54);
        uf.addImpRole(imprf64);

        impboundary3.addImpTask(impTask3);
        impboundary3.addImpTask(impTask4);
        impboundary3.addImpObject(impObj5);
        impboundary3.addImpObject(impObj6);
        impboundary3.addImpReadFlow(rf22);
        impboundary3.addImpReadFlow(rf32);
        impboundary3.addImpCreateFlow(cf11);
        impboundary3.addImpUpdateFlow(uf);

        ImpactModel.addImpBoundary(impboundary);
        ImpactModel.addImpBoundary(impboundary2);
        ImpactModel.addImpBoundary(impboundary3);

        ASTWriter writer1 = new ASTWriter(new File("impactModel.xml"));
        writer1.usePackage("mde.impact.model", "xmlns:imp");
        writer1.writeDocument(ImpactModel);
        writer1.close();

        System.out.println("(2) Impact Model is Created by user.");

        // Generate Information Model
        ImpDiagramToDiagram topRule = new ImpDiagramToDiagram();
        Diagram initial_dataModel = topRule.translate(ImpactModel);

        ASTWriter writer2 = new ASTWriter(new File("initial_InfoModel.xml"));
        writer2.usePackage("mde.information.model", "xmlns:inf");
        writer2.writeDocument(initial_dataModel);
        writer2.close();

        System.out.println("(2) Initial Information Model is Created by rules.");
        // ------------------------ //

        //Generate Data Model
        InfDiagramToDDiagram topRule4 = new InfDiagramToDDiagram();
        DDiagram  dataModel = topRule4.translate(initial_dataModel);

        ASTWriter writer4 = new ASTWriter(new File("Uni_DataModel.xml"));
        writer4.usePackage("mde.data.model", "xmlns:data");
        writer4.usePackage("mde.model", "xmlns:mod");
        writer4.writeDocument(dataModel);
        writer4.close();

        System.out.println("(5) Data Model is Created"(Information Model --> Data Model).");
        // ------------------- //

        //generate Database Schema Model (Detailed_DFD+Data Model ---> Schema)
        DDiagramToSchemaOnly topRule5 = new DDiagramToSchemaOnly();
        Schema schemaModel = topRule5.translate(null, dataModel);

        System.out.println(schemaModel.getName());
```

```java

        ASTWriter writer5 = new ASTWriter(new File("Uni_SchemaModel.xml"));
        writer5.usePackage("mde.dbs.model", "xmlns:dbs");
        writer5.usePackage("mde.model", "xmlns:mod");
        writer5.writeDocument(schemaModel);
        writer5.close();

        System.out.println("(9) Database Schema Model is Created"+
            " (DataFlow + Data Model --> Database Schema Model).");
                // ---------------------------- //

                DumpFileGenerator MySQLGenerator = new
                    MySQLDumpFileGenerator(schemaModel);
                System.out.println("calling generate in mysql package generator");
                MySQLGenerator.generate();
                System.out.println("Finished generating MySQL Schema OK");
    }
}
```

## Impact Model Representation (3b)

Listing C.7: *Uni_impactModel.xml*

```xml
<?xml version="1.0" encoding="UTF-8"?>
<imp:ImpDiagram xmlns:imp="mde.impact.model" id="0">
  <imp:ImpBoundary id="1" name="Enrol_Boundary">
    <imp:ImpTask id="2" name="Enrol"/>
    <imp:ImpObject id="3" name="Student"/>
    <imp:ImpObject id="4" name="Module"/>
    <imp:ImpObject id="5" name="Enrollment"/>
    <imp:ImpReadFlow id="6">
      <imp:ImpRole id="7" name="student" multiple="false">
        <imp:ImpObject ref="3"/>
      </imp:ImpRole>
      <imp:ImpRole id="8" name="enrol" multiple="false">
        <imp:ImpTask ref="2"/>
      </imp:ImpRole>
    </imp:ImpReadFlow>
    <imp:ImpReadFlow id="9">
      <imp:ImpRole id="10" name="module" multiple="false">
        <imp:ImpObject ref="4"/>
      </imp:ImpRole>
      <imp:ImpRole id="11" name="enrol" multiple="false">
        <imp:ImpTask ref="2"/>
      </imp:ImpRole>
    </imp:ImpReadFlow>
    <imp:ImpCreateFlow id="12">
      <imp:ImpRole id="13" name="enrol" multiple="false">
        <imp:ImpTask ref="2"/>
      </imp:ImpRole>
      <imp:ImpRole id="14" name="enrollment" multiple="false">
        <imp:ImpObject ref="5"/>
      </imp:ImpRole>
    </imp:ImpCreateFlow>
  </imp:ImpBoundary>
  <imp:ImpBoundary id="15" name="Set_Address_Boundary">
    <imp:ImpTask id="16" name="Set Address"/>
    <imp:ImpObject ref="3"/>
    <imp:ImpObject id="17" name="Staff"/>
    <imp:ImpObject id="18" name="Address"/>
    <imp:ImpReadFlow id="19">
      <imp:ImpRole id="20" name="student" multiple="false">
        <imp:ImpObject ref="3"/>
      </imp:ImpRole>
      <imp:ImpRole id="21" name="set_address_conj" multiple="false">
        <imp:ImpConjunction id="22" name="Set Address"/>
      </imp:ImpRole>
    </imp:ImpReadFlow>
    <imp:ImpReadFlow id="23">
      <imp:ImpRole id="24" name="staff" multiple="false">
        <imp:ImpObject ref="17"/>
      </imp:ImpRole>
      <imp:ImpRole id="25" name="set_address_conj" multiple="false">
        <imp:ImpConjunction ref="22"/>
      </imp:ImpRole>
    </imp:ImpReadFlow>
    <imp:ImpReadFlow id="26">
      <imp:ImpRole id="27" name="set_address_conj" multiple="false">
        <imp:ImpConjunction ref="22"/>
      </imp:ImpRole>
      <imp:ImpRole id="28" name="set_address" multiple="false">
        <imp:ImpTask ref="16"/>
      </imp:ImpRole>
    </imp:ImpReadFlow>
    <imp:ImpCreateFlow id="29">
      <imp:ImpRole id="30" name="set_address" multiple="false">
        <imp:ImpTask ref="16"/>
      </imp:ImpRole>
```

```xml
        <imp:ImpRole id="31" name="address" multiple="false">
          <imp:ImpObject ref="18"/>
        </imp:ImpRole>
      </imp:ImpCreateFlow>
      <imp:ImpConjunction ref="22"/>
    </imp:ImpBoundary>
    <imp:ImpBoundary id="32" name="Manage_Assessment_Boundary">
      <imp:ImpTask id="33" name="Set Assessment"/>
      <imp:ImpTask id="34" name="Modify Value"/>
      <imp:ImpObject ref="4"/>
      <imp:ImpObject id="35" name="Assessment"/>
      <imp:ImpReadFlow id="36">
        <imp:ImpRole id="37" name="module" multiple="false">
          <imp:ImpObject ref="4"/>
        </imp:ImpRole>
        <imp:ImpRole id="38" name="set_assess" multiple="false">
          <imp:ImpTask ref="33"/>
        </imp:ImpRole>
      </imp:ImpReadFlow>
      <imp:ImpReadFlow id="39">
        <imp:ImpRole id="40" name="modify_value" multiple="false">
          <imp:ImpTask ref="34"/>
        </imp:ImpRole>
        <imp:ImpRole id="41" name="module" multiple="false">
          <imp:ImpObject ref="4"/>
        </imp:ImpRole>
      </imp:ImpReadFlow>
      <imp:ImpCreateFlow id="42">
        <imp:ImpRole id="43" name="set_assess" multiple="false">
          <imp:ImpTask ref="33"/>
        </imp:ImpRole>
        <imp:ImpRole id="44" name="assessment" multiple="false">
          <imp:ImpObject ref="35"/>
        </imp:ImpRole>
      </imp:ImpCreateFlow>
      <imp:ImpUpdateFlow id="45">
        <imp:ImpRole id="46" name="set_assess" multiple="false">
          <imp:ImpTask ref="33"/>
        </imp:ImpRole>
        <imp:ImpRole id="47" name="assessment" multiple="false">
          <imp:ImpObject ref="35"/>
        </imp:ImpRole>
      </imp:ImpUpdateFlow>
    </imp:ImpBoundary>
  </imp:ImpDiagram>
```

## Generated Information Model Representation (3b)

Listing C.8: *initial_infoModel.xml*

```xml
<?xml version="1.0" encoding="UTF-8"?>
<inf:Diagram xmlns:inf="mde.information.model" id="0">
  <inf:Entity id="1" name="General"/>
  <inf:Entity id="2" name="Student"/>
  <inf:Entity id="3" name="Module"/>
  <inf:Entity id="4" name="Enrollment"/>
  <inf:Entity id="5" name="Staff"/>
  <inf:Entity id="6" name="Address"/>
  <inf:Entity id="7" name="Assessment"/>
  <inf:Association id="8">
    <inf:Role id="9" name="student" multiple="true">
      <inf:Entity ref="2"/>
    </inf:Role>
    <inf:Role id="10" name="general" multiple="false">
      <inf:Entity id="11" name="General"/>
    </inf:Role>
  </inf:Association>
  <inf:Association id="12">
    <inf:Role id="13" name="staff" multiple="true">
      <inf:Entity ref="5"/>
    </inf:Role>
    <inf:Role id="14" name="general" multiple="false">
      <inf:Entity ref="11"/>
    </inf:Role>
  </inf:Association>
  <inf:Association id="15">
    <inf:Role id="16" name="address" multiple="true">
      <inf:Entity ref="6"/>
    </inf:Role>
    <inf:Role id="17" name="general" multiple="false">
      <inf:Entity ref="11"/>
    </inf:Role>
  </inf:Association>
  <inf:Association id="18">
    <inf:Role id="19" name="enrollment" multiple="true">
      <inf:Entity ref="4"/>
    </inf:Role>
    <inf:Role id="20" name="student" multiple="false">
```

```
        <inf:Entity ref="2"/>
      </inf:Role>
    </inf:Association>
    <inf:Association id="21">
      <inf:Role ref="19"/>
      <inf:Role id="22" name="module" multiple="false">
        <inf:Entity ref="3"/>
      </inf:Role>
    </inf:Association>
    <inf:Association id="23">
      <inf:Role id="24" name="assessment" multiple="true">
        <inf:Entity ref="7"/>
      </inf:Role>
      <inf:Role id="25" name="module" multiple="false">
        <inf:Entity ref="3"/>
      </inf:Role>
    </inf:Association>
    <inf:Association id="26">
      <inf:Role id="27" name="assessment" multiple="true">
        <inf:Entity ref="7"/>
      </inf:Role>
      <inf:Role ref="25"/>
    </inf:Association>
  </inf:Diagram>
```

## Generated Data Model Representation (3b)

Listing C.9: *DataModel.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<data:DDiagram xmlns:data="mde.data.model" id="0">
  <data:DEntity id="1" name="General"/>
  <data:DEntity id="2" name="Student"/>
  <data:DEntity id="3" name="Module"/>
  <data:DEntity id="4" name="Enrollment"/>
  <data:DEntity id="5" name="Staff"/>
  <data:DEntity id="6" name="Address"/>
  <data:DEntity id="7" name="Assessment"/>
  <data:DDependency id="8">
    <data:DRole id="9" name="student">
      <data:DEntity ref="2"/>
    </data:DRole>
    <data:DRole id="10" name="general">
      <data:DEntity id="11" name="General"/>
    </data:DRole>
  </data:DDependency>
  <data:DDependency id="12">
    <data:DRole id="13" name="staff">
      <data:DEntity ref="5"/>
    </data:DRole>
    <data:DRole id="14" name="general">
      <data:DEntity ref="11"/>
    </data:DRole>
  </data:DDependency>
  <data:DDependency id="15">
    <data:DRole id="16" name="address">
      <data:DEntity ref="6"/>
    </data:DRole>
    <data:DRole id="17" name="general">
      <data:DEntity ref="11"/>
    </data:DRole>
  </data:DDependency>
  <data:DDependency id="18">
    <data:DRole id="19" name="enrollment">
      <data:DEntity ref="4"/>
    </data:DRole>
    <data:DRole id="20" name="student">
      <data:DEntity ref="2"/>
    </data:DRole>
  </data:DDependency>
  <data:DDependency id="21">
    <data:DRole ref="19"/>
    <data:DRole id="22" name="module">
      <data:DEntity ref="3"/>
    </data:DRole>
  </data:DDependency>
  <data:DDependency id="23">
    <data:DRole id="24" name="assessment">
      <data:DEntity ref="7"/>
    </data:DRole>
    <data:DRole id="25" name="module">
      <data:DEntity ref="3"/>
    </data:DRole>
  </data:DDependency>
  <data:DDependency id="26">
    <data:DRole id="27" name="assessment">
      <data:DEntity ref="7"/>
    </data:DRole>
```

```
      <data:DRole ref="25"/>
    </data:DDependency>
  </data:DDiagram>
```

## Generated Database and Query Model Representation (3b)

Listing C.10: *SchemaModel.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<dbs:Schema xmlns:dbs="mde.dbs.model" id="0" name="database">
  <dbs:Table id="1" name="General">
    <dbs:Column id="2" name="identity" automatic="true" size="10">
      <mod:Type xmlns:mod="mde.model" id="3" name="INTEGER"/>
    </dbs:Column>
    <dbs:Column id="4" name="Student.identity" size="10">
      <mod:Type id="5" name="INTEGER"/>
    </dbs:Column>
    <dbs:Column id="6" name="Staff.identity" size="10">
      <mod:Type id="7" name="INTEGER"/>
    </dbs:Column>
    <dbs:Column id="8" name="Address.identity" size="10">
      <mod:Type id="9" name="INTEGER"/>
    </dbs:Column>
    <dbs:ForeignKey id="10">
      <dbs:Column ref="4"/>
      <dbs:Table id="11" name="Student">
        <dbs:Column id="12" name="identity" automatic="true" size="10">
          <mod:Type ref="5"/>
        </dbs:Column>
        <dbs:Column id="13" name="Enrollment.identity" size="10">
          <mod:Type id="14" name="INTEGER"/>
        </dbs:Column>
        <dbs:ForeignKey id="15">
          <dbs:Column ref="13"/>
          <dbs:Table id="16" name="Enrollment">
            <dbs:Column id="17" name="identity" automatic="true" size="10">
              <mod:Type ref="14"/>
            </dbs:Column>
            <dbs:PrimaryKey id="18">
              <dbs:Column ref="17"/>
            </dbs:PrimaryKey>
          </dbs:Table>
        </dbs:ForeignKey>
        <dbs:PrimaryKey id="19">
          <dbs:Column ref="12"/>
        </dbs:PrimaryKey>
      </dbs:Table>
    </dbs:ForeignKey>
    <dbs:ForeignKey id="20">
      <dbs:Column ref="6"/>
      <dbs:Table id="21" name="Staff">
        <dbs:Column id="22" name="identity" automatic="true" size="10">
          <mod:Type ref="7"/>
        </dbs:Column>
        <dbs:PrimaryKey id="23">
          <dbs:Column ref="22"/>
        </dbs:PrimaryKey>
      </dbs:Table>
    </dbs:ForeignKey>
    <dbs:ForeignKey id="24">
      <dbs:Column ref="8"/>
      <dbs:Table id="25" name="Address">
        <dbs:Column id="26" name="identity" automatic="true" size="10">
          <mod:Type ref="9"/>
        </dbs:Column>
        <dbs:PrimaryKey id="27">
          <dbs:Column ref="26"/>
        </dbs:PrimaryKey>
      </dbs:Table>
    </dbs:ForeignKey>
    <dbs:PrimaryKey id="28">
      <dbs:Column ref="2"/>
    </dbs:PrimaryKey>
  </dbs:Table>
  <dbs:Table ref="11"/>
  <dbs:Table id="29" name="Module">
    <dbs:Column id="30" name="identity" automatic="true" size="10">
      <mod:Type id="31" name="INTEGER"/>
    </dbs:Column>
    <dbs:Column ref="13"/>
    <dbs:Column id="32" name="Assessment.identity" size="10">
      <mod:Type id="33" name="INTEGER"/>
    </dbs:Column>
    <dbs:ForeignKey ref="15"/>
    <dbs:ForeignKey id="34">
      <dbs:Column id="35" name="Assessment.identity" size="10">
        <mod:Type ref="33"/>
      </dbs:Column>
```

```
      <dbs:Table id="36" name="Assessment">
        <dbs:Column id="37" name="identity" automatic="true" size="10">
          <mod:Type ref="33"/>
        </dbs:Column>
        <dbs:PrimaryKey id="38">
          <dbs:Column ref="37"/>
        </dbs:PrimaryKey>
      </dbs:Table>
    </dbs:ForeignKey>
    <dbs:ForeignKey id="39">
      <dbs:Column ref="32"/>
      <dbs:Table ref="36"/>
    </dbs:ForeignKey>
    <dbs:PrimaryKey id="40">
      <dbs:Column ref="30"/>
    </dbs:PrimaryKey>
  </dbs:Table>
  <dbs:Table ref="16"/>
  <dbs:Table ref="21"/>
  <dbs:Table ref="25"/>
  <dbs:Table ref="36"/>
</dbs:Schema>
```

## Executable MySQL Code (3b)

Listing C.11: *database_MySQL.sql*

```sql
-- Database Creation

CREATE DATABASE sysDatabase;
USE sysDatabase;

-- Structure for table 'General'

CREATE TABLE General (
  identity INT(10) NOT NULL,
  Student.identity INT(10) NOT NULL,
  Staff.identity INT(10) NOT NULL,
  Address.identity INT(10) NOT NULL,
 PRIMARY KEY(identity),
 FOREIGN KEY(Student.identity) REFERENCES Student(identity),
 FOREIGN KEY(Staff.identity) REFERENCES Staff(identity),
 FOREIGN KEY(Address.identity) REFERENCES Address(identity));

-- Structure for table 'Student'

CREATE TABLE Student (
  identity INT(10) NOT NULL,
  Enrollment.identity INT(10) NOT NULL,
 PRIMARY KEY(identity),
 FOREIGN KEY(Enrollment.identity) REFERENCES Enrollment(identity));

-- Structure for table 'Module'

CREATE TABLE Module (
  identity INT(10) NOT NULL,
  Enrollment.identity INT(10) NOT NULL,
  Assessment.identity INT(10) NOT NULL,
 PRIMARY KEY(identity),
 FOREIGN KEY(Enrollment.identity) REFERENCES Enrollment(identity),
 FOREIGN KEY(Assessment.identity) REFERENCES Assessment(identity),
 FOREIGN KEY(Assessment.identity) REFERENCES Assessment(identity));

-- Structure for table 'Enrollment'

CREATE TABLE Enrollment (
  identity INT(10) NOT NULL,
 PRIMARY KEY(identity));

-- Structure for table 'Staff'

CREATE TABLE Staff (
  identity INT(10) NOT NULL,
 PRIMARY KEY(identity));

-- Structure for table 'Address'

CREATE TABLE Address (
  identity INT(10) NOT NULL,
 PRIMARY KEY(identity));

-- Structure for table 'Assessment'

CREATE TABLE Assessment (
  identity INT(10) NOT NULL,
 PRIMARY KEY(identity));
```

# D
# End-User Evaluation Experiment of ML Notation

## D.1 End-User Evaluation Experiment

### D.1.1 Description of the Online System

This section provides a description of business activity structure of an Online Hospital Booking System (OHBS). An admin can manage doctors in the system by either: adding a new doctor, updating a profile of an existing doctor, removing an existing doctor from the system. The admin is able to add a new doctor by first adding the personal detail of a new doctor to the system. Then, adding the login detail of the new doctor to the system.

- Draw an ellipse for each business activity (process) with a suitable name..
- Draw a white diamond and branches to represent options part processes (sub-processes) of a whole business process.
- Draw a white diamond and branches to represent all parts processes (sub-processes) of a whole business process.

**Time now (before drawing):**

**Time now (After drawing):**

### D.1.2 Description of Business Items (Entities)

This part describes information (business data) that is used within the online system to complete business activities. The system needs information about each doctor to be stored in the system by Adding personal detail of the new doctor process. There is a login information that has to be stored for each new doctor in the system by adding login detail process.

In addition, the system modifies the information about doctor profile by Update doctor profile process, and removes an existing doctor from the system by Remove doctor process.

- Draw a rectangle for each business entity and write a suitable name for it.
- Draw a line between each two entities that have a kind of relationship.

**Time now (before drawing):**

**Time now (After drawing):**

### D.1.3 Description of How the System Data interact with system

(Samilar text as the previous section) This part describes the interaction between business information (data) and business process within the online system. The system needs information about each doctor to be stored in the system by Adding personal detail of the new doctor process. There is a login detail that has to be stored for each new doctor in the system by adding login detail process.

In addition, the system modifies doctor profile by Update doctor profile process, and remove an existing doctor from the system by Removes doctor process.

- Draw an ellipse for each business activity (process) with a suitable name.
- Draw a rectangle for each business entity with a suitable name.
- Draw a suitable arrow between tasks and related entities (as explained).

**Time now (before drawing):**

**Time now (After drawing):**