UNIVERSITY OF SOUTHAMPTON

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

Electronics and Computer Science

Incremental and Rigorous Database Design and Code Generation Using UML-B and Event-B

by

Ahmed Al-Brashdi

Thesis for the degree of Doctor of Philosophy

April 2020

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES
Electronics and Computer Science

Doctor of Philosophy

INCREMENTAL AND RIGOROUS DATABASE DESIGN AND CODE GENERATION USING UML-B AND EVENT-B

by Ahmed Al-Brashdi

Correct operation of many critical systems is dependent on the consistency and integrity properties of underlying databases. Therefore, a verifiable and rigorous database design process is highly desirable. This research investigated and delivered a comprehensive and practical approach for modelling databases in a formal method and provide a tool that translates the verified model to a database implementation. The methodology was guided by a number of case studies, using abstraction and refinement in UML-B and verification with the Rodin tool. UML-B is a graphical representation of the Event-B formalism and the Rodin tool supports verification for Event-B and UML-B. Our method guides developers to model relational databases in UML-B through layered refinement and to specify the necessary constraints and operations on the database. The guidelines are supported by a tool we have developed called UB2DB that automatically generates a database system from a verified UML-B model. The tool generates both the structure to create the database in Oracle as well as the necessary operations on the database that has been modelled as events in UML-B model. The evaluation shows that the generated code from the models of the case studies preserves the constraints of the database and the performance of the operations is not very different from a hand written code.

Contents

List of Figures

List of Tables

Declaration of Authorship

I, Ahmed Al-Brashdi , declare that the thesis entitled *Incremental and Rigorous Database Design and Code Generation Using UML-B and Event-B* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as: [9],[10], [12], [11].

Signed:

Date:

Acknowledgements

I would like to express my gratitude to my family who supported me unconditionally during this PhD. My deep thanks goes to my supervisors whose without their help and insights, this thesis won't come to life. I would also like to thank the Ministry of Higher Education - Oman for their sponsorship and support for the whole PhD.

Chapter 1

Introduction

The design of database systems is an important discipline of software engineering [30] as databases carry critical data upon which important decisions rely. While designing a small database application might be simple and straightforward, current enterprises depend on very large and complex database systems. For every increase in complexity, the possibility of inconsistency and errors increases as well. One way of addressing this issue is to construct database applications using a rigorous design methodology. Using *formal methods*, a developer can specify complex systems and use formal verification to detect ambiguities and inconsistencies [26]. This thesis contributes an incremental design method for rigorous database design and provide a tool support for it with automated database code generation. The incremental pattern is derived from the investigation and modelling of different case studies.

1.1 Problem Statement

Unverified database systems might lead to critical consequences. Inconsistent or corrupt data in patient databases might result in a patient getting a wrong prescription which could harm his/her life [44]. If an enterprise derives its decisions from a non-accurate database, it may result in a loss of money. The conventional database design using Entity-Relational Diagram (ERD) to describe databases is restricted to modelling the structure of the databases without specifying the system behaviour. Modelling only the structure of the databases does not prove its consistency or unambiguity as these can be caused by an operation of the structure. A database user might try to execute statements on tables that do not exist in the database after database design. This might result from the separation of building the database from the program implementation that manipulate the database. Another error that affects the database consistency is when multiple statements that should be executed together are designed to execute independently. Commercial database systems such as Oracle and MySQL only prove

the consistency of database in run-time when trying to commit the transactions. This limits the discovery of inconsistency and ambiguity of requirements to the last stage of development and that increases the cost of development. This highlights the question of how to prove database consistency and integrity in the early stages of development in relation to the system requirements and ensure that operations on the database don't violate its constraints. To address the problem on a wide scale, we propose the usage of formal methods so that the requirements of the system can be proved to be consistent and unambiguous before any implementation of the database is created or generated.

Moreover, database systems evolve through time as new additions and extensions to the existing databases are often needed. For example, OSCAR (Open Source Clinical Application Resource) [98] evolved in ten years from 88 tables to 445 tables [27]. Many of the extensions also included adding new columns to existing tables. These extensions must preserve the existing database and do not violate any of its constraints as many inconsistencies accrue when the operations on databases do not reflect the changes made on these databases [71]. In order to address this, we need a methodology that supports incremental database design using formal methods where extended version of the specifications with its operations preserves the old specifications. This methodology should provide clear guidelines of how to abstract and refine database systems and provide support to them. The use of formal methods aims at making precise specifications and eliminating errors, while the incremental approach breaks the complexity of the system.

Finally, after successful design and specification of database application, there is a need to provide automation of code generation that is sound and preserve the requirements in the specification. Without this feature, the manual development of the code will be prone to errors and could produce incorrect implementation of some of the requirements specified in the design phase. Moreover, since most of the effort for the user will be spend on the specification, automated code generation will promote the productivity of the user by cutting the time needed for coding. The code generation should support the incremental development of the database and provide the facility to generate the code for extensions after proving they satisfy the existing database generated from the abstract model. The correctness of the generated code need to be addressed as well. Another issue with the code generation is its efficiency compared to a handwritten code. While the hand-written code can be optimised manually in ways that machines can not, the automatically generated code for applications should be in average 5 to 15 percent better or worse than the hand-written one [89].

The interest of this thesis lies on addressing the following:

1. Applying formal verification and specification on information systems,
2. providing an incremental design method to model information systems that help mastering the requirements by breaking the system into abstractions and refinements,

3. provide a tool support to automatically generate correct and efficient database code from the model.

The choice to address the problem on business information system as a target instead of embedded system is based on the scale and availability of such systems. While both might contains critical data, the business information systems are more related to decision making and customer information integrity. Moreover, business information systems are more diverse and rich in their components, so using them in the case studies addresses larger target. Adding to that, business information system is the target in which future extension of the system is more common than embedded system, hence the proposed incremental methodology. An example of such evolution of the system is the the Open Source Clinical Application Resource which evolved in ten years from 88 tables to 445 tables [27].

1.2 Aim and Objectives

This research developed a method and a tool to help developers rigorously design the structure of their database application and create the methods of manipulating and maintaining its data using abstraction and refinement. It defines guidelines and methods for modelling information systems using a UML-like graphical notation called UML-B in the Rodin platform. The Rodin platform provides support for modelling in Event-B and can be used to prove the consistency of the model specifications. The UML-B model are then be translated to Event-B specifications.

As UML-B is a graphical notation, it helps users and modellers to understand the system more clearly than modelling the system directly in Event-B. Moreover, since UML-B uses class diagrams, it is more aligned with the database as class diagrams are commonly used to describe databases. A further advantage of using UML-B is the auto generation facility provided by the UML-B tool in which the graphical notation is automatically translated to an Event-B model.

The objectives of this research were:

1. To discover the distinct types of classes, relations and associations relevant to databases to be modelled in UML-B; and to provide general guidelines that can be followed when modelling any database system in Event-B.

2. To propose an incremental approach for modelling information systems that respects its integrity and consistency. The approach should be applicable to different case studies.

3. To define translation rules to generate database code from formal specifications that can be automated by a tool.

4. To develop a methodology and a tool that automatically translates the formal model of an information system to database applications that create, read, update and delete the data efficiently.

5. To define a formal semantics for the generated database code so it can be shown that it is a correct implementation of the formal model defined in 2.

This PhD research delivers a comprehensive method and a supporting tool to translate UML-B models to database systems. The emphasis of this work is on relational database systems but it could be extended in the future to support other models of database systems such as NoSQL. It provides a method and a tool to model and translate different components of relational databases along with their constraints.

Contribution 1: Defining general guidelines and methodologies on how to model information systems that are data intensive in UML-B and prove them in the Rodin platform. These guidelines help the modeller to specify different constraints and invariants that satisfy the system requirements.

Contribution 2: Modelling different case studies of database systems in UML-B and Event-B and incorporate the methodologies defined in ***Contribution 1*** in their design.

Contribution 3: Defining translation rules from UML-B and Event-B models to database systems. These rules are used by a tool that generates the database implementation code from the model.

Contribution 4: Defining different design patterns for relational databases and incorporate these patterns in the process of modelling information systems in UML-B. The patterns include how to map object-oriented concepts such as inheritance in UML-B into the relational database model. It also defines solutions or common practice for common database needs.

Contribution 5: Building a tool that supports the above contributions and translates the UML-B model to a database system. The tool should generate the code that creates the database structure and manipulates its contents.

Contribution 6: Defining a formal semantics for SQL that covers all SQL statements for manipulating data. This also defines how to prove the correctness of the generated code as in ***Contribution 4***.

1.3 Research Method

This thesis follows a case study method when developing guidelines for database modelling in UML-B and Event-B. In order to identify different components and concepts for

the modelling practice, the modelling approach followed for the first case study is generalised and then is repeated for another case study after identify the common patterns to model databases using abstraction and refinement. By looking at each component in the model and how it can be translated to database code, we define a set of general translation rules that can be automated by a tool. The translation rules must follow a clear semantics that can be used also for a reverse translation (from database code to Event-B). When defining the semantics, we revisit the modelling approach and the translation rules and adjust them so they can be consistent with the semantics. The translation is then validated by a tool that follows the translation rules to generate the database code. We apply the case studies to the tool and automatically generate their database code. The generated code is then tested against the requirements of these case studies and defines any modifications and corrections needed. The modelling, semantics and translation rules presented in this thesis are the final result of this iterative process.

We have chosen case study method so that we validate the whole process on the same case studies staring from modelling to code generation and including applying the translation rules on the case studies. The case studies selected based on characteristics that needs to be covered. After the characteristics have been covered in all the case studies selected, we stop modelling further case studies. These characteristics are listed in the Table 1.1. The case studies do not relate to each other. The case studies are from different domains that covers business as in Car Sharing, and health care as in Emergency Room system.

We believe by covering these case studies with different characteristics and constraints, that the modelling, translation and the tool cover most of the cases. In case where a new system falls outside the scope of our work, the methodology and the tool will need to be extended. As the formal foundation has been defined, this should be a matter of extending the translation rules in the tool.

Characteristic	**Covered In**
Diverse types of relations and attributes	All case studies
Diverse types of constraints	All case studies
Highly relational and connected where some constraints required connecting more than two tables together.	SRES case study
Relational association for many-to-many relationship	SRES Case Study
Time constraints where fields time on one table must be contained within the time of another field at the same table or another table	Car Sharing system
Multi layer inheritance	Emergency Room system

Table 1.1: Characteristics of targeted case studies

1.4 Thesis Organisation

This thesis is divided into different chapters as follows:

- Chapter 2 gives a background about the topics that are related to this research, mainly formal methods and the relational database model. Section 2.5 provides an overview of related work that directly relates to this research.
- Chapter 3 describes how to model an information system using UML-B and Event-B in Rodin.
- Chapter 4 identifies translation rules from UML-B model to relational database. It provides rules to translate components that are modelled in Chapter 3.
- Chapter 5 defines the formal semantics of a subset of SQL and stored procedures that are targeted by the translation rules. They are used to show that the source of the translation and the target of the semantics of the generated code are equivalent.
- Chapter 6 outlines some case studies developed to establish guidelines and rules of translation and to evaluate the translation and the tool.
- Chapter 7 introduces the tool we developed for automatic code generation of a database system from UML-B and Event-B model. The chapter also outlines the relational database meta-model we defined to be used for model-to-model transformation by the developed tool.
- Chapter 8 presents a conclusion of achievements and contributions.

Chapter 2

Background

2.1 Introduction

In this chapter, different concepts that are related to our research are introduced. This chapter starts by introducing formal methods in general with emphasis on Event-B which is the formal method used in this research, followed by UML-B graphical notation. As the research deals with generating database application, a brief introduction is given to different models in database systems followed by more discussion on relational databases. The chapter covers the main features of relational databases that are related to this research.

2.2 Formal Methods

Formal methods are mathematical techniques used to rigorously specify, design and verify a system [26]. Formal specifications describe precisely the properties in which the system must have. They define *what* the system will do not *how* it does it. As formal specifications are independent of any programming language, they can be introduced at early stages in the development process. Formal verification enables a modeller to detect any ambiguities or inconsistencies early in the development process before any implementation is done. Examples of formal methods include *Event-B*, *Vienna Development Method (VDM)* [61], *Z notation* [93], and *B method* [4].

2.2.1 Overview of some formal methods

In this section, a brief overview of some of the formal methods is given. These formal methods are used in the related work that will be discussed in the next chapter. We separated Event-B with more details as it is the formal method we use in our research.

- **VDM** or Vienna Development Method is a formal method based on set theory and predicate logic and it was developed in the IBM Laboratory in Vienna [61]. VDM models are specified using a specification language called VDM-SL which has been standardised by the International Standards Organization [13]. VDM supports modelling systems at different level of abstractions. VDM++ extends VDM with support for object-oriented modelling by offering classes, objects and inheritance [39].

  A model in VDM consists of data types and operations that describe what the behaviour of the system. A variable in an operation is proceeded by an access modifier that specifies if the operation has a read-only or read-write access to the variable. Another feature in VDM is the function that calculates a result from numbers. A function is different from operation in VDM as it yields the same result for the same input, while the operation might give a different value [72]. VDM refinement, or *reification*, is supported by *data reification* which involves transiting abstract data types to concrete one for implementation. Operations in VDM are *decomposed* into a concrete implementation.

  Modelling in VDM is supported by *VDMTools* [46] which is a group of tools that support formal modelling using VDM and VDM++. The tools support *syntax* and *type checking* as well as *integrity checking* or proof obligation generation. Failed proof obligations can be checked by inspection of by external proof tool [99] as VDMTools does not discharge the proof obligations automatically.

- **Z**, which pronounced *Zed* is a formal notation that is also based on set theory and predicate logic. Unlike VDM, Z provides a way for decomposing a specification into small pieces called *schemas* [93]. The schemas are used to describe the static aspects of a system such as states and invariants, and the dynamic parts such as operations and changes to the system states. Object-Z extends Z notation to support specifications in an object oriented style [90].

  Refinement in Z is achieved by *operation refinement* and *data refinement*. For operation refinement, Z defines proof obligations for a concrete operation, *Cop*, to be a correct implementation of an abstract operation, *Aop*. Data refinement allows the mathematical data types to be refined into more concrete computer-oriented types for implementation [93].

  Z notation is supported by tools such as the *Community Z Tools* project [81, 66] which includes support for modelling in Z and is based on Eclipse. There is an Eclipse plugin for the *Z/EVES* [85] theorem prover that can be integrated with Community Z Tools but doesn't discharge the proofs automatically.

- **B-method** which also known as classical B (to distinguish it from Event-B) is yet another formal method based on set theory and predicate logic [4]. B method structure is based on the notion of *abstract machine* and *refinement* where the

system is first specified at an abstract level and then is refined to more concrete specifications that make it closer to the implementation. It supports data refinement similar to Z.

B method is supported by *Atelier B* tool that enables operational use of the B method. It includes automated syntax verification, proof obligation generation and automatic translation of B models to C or Ada languages [41].

2.3 Event-B

Event-B is a formal method for rigorous specification and verification of a digital system be it hardware or software and it is an extension to B method [3]. While the B method supports correct by construction software modelling, Event-B provides system level modelling by abstracting the entire system including software, hardware and environment. Another noted difference is that in B it is hard for the user to extend the language and the tool as most of them are not open source [53]. Event-B on the other hand is open for extensions. It is supported in an open tool platform called Rodin [5]. A model in Event-B consists of a static part in a *context* that defines the types, constants and axioms, and a dynamic *machine* part with all the *variables* and *invariants* as well as the *events* that change the variable state. An event in a machine may have *guards* that hold true before the execution of the event. Event *actions* change the state of variables. All events in a machine must satisfy all of its invariants.

2.3.1 Structure

Event-B is structured into two parts, static which is represented by a *context* and a dynamic which is represented by a *machine*. The context consists of *sets*, *constants* and *axioms*. The sets represent abstract types of entities [23]. The constants are logical variables whose values remain unchanged during execution of a machine. The axioms are assumptions about the constants. An Event-B machine sees a context and it consists of *variables*, *invariants*, and *events*. The machine variables can have their state changed in an event. The invariants are constraints on the variables in which must be held true by all events.

The following example illustrates the structure of a machine called *machine0* that sees the context *context0*. Machine0 has one variable *a* which is a set of integers from 1 to 100 and one event *addA* that adds a new element to a. We use the invariants to specify the type or constraints on variables as in *inv1*. The parameter *this_a* is a local variable to the event and the guards *grd1* must be true to enable the event.

CONTEXT context0
SETS
END
MACHINE machine0
SEES context0
VARIABLES

a
INVARIANTS

$\texttt{inv1} : a \subseteq 1 \mathinner{..} 100$
EVENTS
Initialisation

begin

$\texttt{act1} : a := \varnothing$
end

$addA \mathrel{\widehat{=}}$

any

$this\_a$
where

$\texttt{grd1} : this\_a \in 1 \mathinner{..} 100 \setminus a$
then

$\texttt{act1} : a := a \cup \{this\_a\}$
end

END

2.3.2 Refinement

Model refinement is a key concept in Event-B which enables a modeller to gradually specify the system so it becomes more precise and closer to reality [6]. Event-B provides a more flexible refinement approach than in Z, VDM and B [22] as it provides extensions to the abstract model. In a refinement, we start with an abstract model that describes the main functionality of a system. Then gradually we elaborate the system by adding further details in the specifications. Event-B refinement can be *horizontal* or *vertical*. The horizontal refinement includes adding extra details such as adding variables and events to the model while the vertical refinement or *data refinement* transforms the state of an abstract variable to make it closer to programming implementation.

Applying refinement to a context in Event-B can be done by adding new sets, constants and axioms. Machine refinement may include adding new variables, invariants and new events or refining existing events [3]. A new event in a refinement refines a *skip* event

Proof Obligation	**Description**
Invariant Preservation **(INV)**	This ensures that each event in the machine preserves all the invariants
Feasibility **(FIS)**	This ensure a non-deterministic action is feasible
Guard Strengthening **(GRD)**	This guarantees that the concrete event guards are stronger than the abstract event ones
Guard merging **(MRG)**	This ensures that a guard of an event that merges two abstract events is stronger than the disjunction of their guards
Simulation **(SIM)**	This ensures that an abstract event is correctly simulated in refinement
Well-definedness **(WD)**	This ensures that any element in the model such as axiom, theorem, invariant, guard, action, variant or witness is well defined

Table 2.1: POs in Event-B

that does nothing to the abstract model. Unlike VDM, Z and B method which have only one-to-one operation refinement [43], introducing new events in refinement is possible in Event-B as well as event splitting and merging.

When doing refinement we have to make some proofs so that the new refinement doesn't violate the abstract model. Moreover, doing a data refinement that removes an abstract variable and replaces it with another more concrete one introduces the *gluing invariant*. The gluing invariant is a predicate that links abstract variables and concrete variables [3].

2.3.3 Proof obligation

When modelling in Event-B, *Proof Obligations (PO)* define what is to be proved in the model [3]. They are used to prove the consistency of Event-B model. Proof obligations are automatically generated in the Rodin tool and need to be discharged either automatically or interactively by the modeller. Table 2.1 describes some of the proof obligation rules in Event-B. A complete list of these rules can be found in [3].

2.3.4 Rodin

Event-B is provided with an open tool called Rodin that integrates modelling and proving of Event-B models [5]. Rodin is built as an extension of the Eclipse [48] tool and is implemented in Java. The openness of the Rodin make it extensible and allow different parties to integrate their rigorous development tools to Rodin as plug-ins [24].

Rodin consists of three components: *static checker*, *proof obligation generator* and *proof manager*. The static checker analyses the model for any syntactical or typing errors [5].

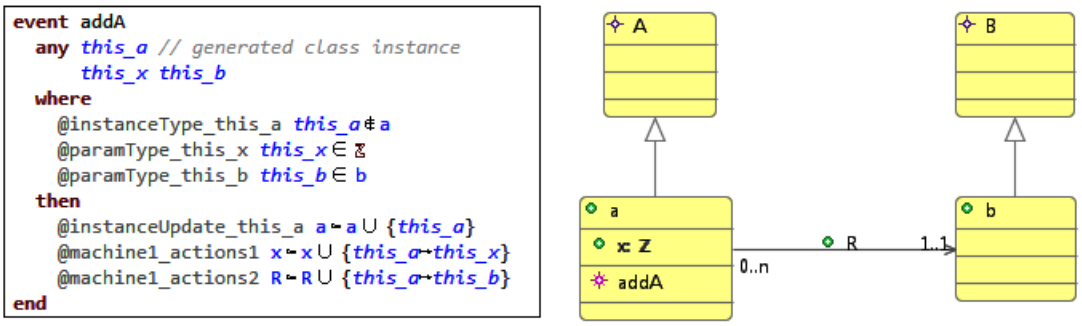

Figure 2.1: Example of UML-B modelling

The proof obligation generator generates the POs as in Table 2.1. The proof obligation manager also invokes the automated proof tactics of proofs such as discharged or not.

2.3.5 UML-B

The Unified Modeling Language (UML) is a graphical modelling language used to visualise, specify, construct and document a software system [84]. UML-B is a graphical notation for formal modelling in Event-B that is based on UML [91]. A tool, called iUML-B, is provided which supports building diagrams in UML-B and is integrated into an Event-B machine or context. The model is translated into Event-B for verification. UML-B supports modelling with class diagrams which are used to describe the model and state machine diagrams which partition a class into different states. The iUML-B tool is based on the Eclipse Modelling Framework (EMF) for Event-B [92]. Figure 2.1 shows a basic model in UML-B where A and B are supertypes for classes a and b and R is an association between them. The variable x is an attribute in a of type integer and *addA* is an event that modifies the state of a. It adds an element to a and to its attribute x and association R. The association, represented by an arrow between two classes, has a name and a multiplicity such as R with a multiplicity of one a to exactly one b which is represented by 1..1. The other end of the multiplicity, $0..n$, means that a b can be associated by zero to n as.

UML-B allows the modeller to choose one of three kinds when adding an event to a class. These kinds are *normal* event, *constructor* or *destructor*. A constructor event should be selected for events that aim to create an instance of a class. The destructor is used for the opposite. For other operations, the normal event is selected; it adds a guard automatically to check that the instance to select or update is an element of that class set.

UML-B provides support for the refinement of class diagrams. The class diagram can be refined by adding new attributes or new associations. New classes and events are also possible when refining a UML-B class diagram and new class invariants can be defined [86].

2.3.6 Comparison with other formal methods

Unlike Z, VDM and classical B, Event-B provides more flexible refinements in which we can introduce new events in refinements. This feature is important in our research as an information system includes different operations on its data and we need to introduce these operations on data when their variables are introduced in a refinement. Without the feature of extension refinement, one cannot extend the formal model of an information system with more details and provide operations on them. In a refinement, we can strengthen the guards in Event-B but not the pre-conditions in VDM, Z and B method. Strengthening guards are needed when we have cases such as case split where an abstract event is refined into two events for different cases. Another example is when a new event is executed before the refined event in which we need to strengthen the guards in the refined one so that it is enabled only when the new event completes.

Moreover, as we expect operations on the database to be called in a non-deterministic manner, we need to ensure that operation pre-conditions are well preserved by the operation itself. The guards in Event-B events are the pre-conditions that must hold for an event to be enabled. However, in classical B, the caller of an operation is the one to ensure that the pre-conditions are hold but not by the called operation itself [43]. We do not call an operation from within another operation in our research, but we model each operation separately.

As Rodin integrates modelling and proving in one tool set, it also provides an immediate feedback related to the model. The Rodin platform allows the proof obligation of a model to be traced, hence, easily fixed if failed [5, 18]. Tools such as Community Z Tools and VDMTools do not provide the same automation of proof obligations generation and discharge. The wide variety of open tools and integrations with Rodin add values to the research. As iUML-B is an open tool, the research can integrate with it and provide a database translation from the same model that is translated by iUML-B to Event-B and proved in Rodin. The continuous development and support for Rodin provide a solid background for this research to rely on. While Atelier-B provides an automatic discharge of proofs, it does not provide an integration with UML modelling and generation.

2.4 Database Systems

Database systems have different data models such as *hierarchical*, *object-oriented*, *graph*, *relational*, *key-value*, *document-oriented* and *column-oriented*. A data model describes the data for a system and is usually consists of *structure*, *operations* and *constraints* of data [51]. Hierarchical models construct data as a tree [97] with parent and child structure. Object data models bring the concept of object-oriented programming to databases [16] and represent the data as a collection of objects [15]. Graph models

of databases structure data as graphs and manipulate them by graph-oriented operations [15]. The key-value model stores data as key-value pairs where a client may add or retrieve the value by a key [96]. In document-oriented data models, each record in the database is considered as a document and is stored in a semi-structured format [58]. Column-oriented databases store and retrieve data by columns instead of rows [96]. The last three models are the ones that being used in NoSQL (Not only SQL) databases.

2.4.1 Relational database

In 1970, Codd [28] introduced the relational model of database systems, which became the most widely used database type. The relational model of a database is composed of several *relations*. Relation elements are represented as *tuples* which may be formed by one or many *columns* where each column takes it values from a set. Given sets $S_1, S_2, \ldots, S_n$, relation R is a set of n-tuples where each tuple has its first element from S_1, its second element from S_2, ...etc. Each column in a relation has a heading (name) and a domain of values such as character or integer. Relations are viewed as tables, tuples as rows and columns as attributes.

While database tables are stored physically in disk blocks, relational databases support *views* which are virtual tables that are formed by a query on one or more underlying tables. In other words, a view is just a stored query. The view can be used to hide the complexity of the underlying database from the user and application developer. A view may join multiple tables together and hide the complexity of these joins conditions to provide a simple interface for the data. Another advantage of using views is to add a level of security to the database by hiding some information from underlying tables and presenting only the necessary ones. Moreover, providing a logical data independence using views is very beneficial when developing a complex database system. If a modeller or database administrator wants to change the structure of a table, and there is an application which uses it, a view can be generated that preserves the current structure of that table, and the table can be changed without affecting the application.

2.4.2 Relational algebra

Relational algebra is a theoretical language that is used to define operations using mathematical notation from the set theory. An operation on a relation results in a new relation. For relations, A and B, we may have different operations performed on them where both relations need to have an identical set of attributes with the same domain. These operations are:

- **Union** $\cup$: $A \cup B = \{x | x \in A \vee x \in B\}$. The union between A and B results in a new relation that has all the element in A or B or both.

- **Intersection** $\cap : A \cap B = \{x | x \in A \wedge x \in B\}$. The intersection between relation A and relation B results in a new relation that has all the element that appears in both A and B.
- **Set difference** $- : A - B = \{x | x \in A \wedge x \notin B\}$. The difference between relation A and relation B give us a relation that has all A elements which are not in B.
- **Cartesian product** $\times : A \times B = \{xy | x \in A \wedge y \in B\}$. The cartesian product pairs the tuples of relations A with tuple x and B with tuple y and forms new tuples xy.
- **Join** $\bowtie$: $A \bowtie B = \{x, y, z | x, y \in A \wedge y, z \in B\}$. The join pairs relations A and B using common column.
- **Selection** $\sigma : \sigma_p(A) = \{a | a \in A \wedge p(a)\}$. Where a is a tuple in relation A and p is a predicate on a. The selection operator retrieves subset of A tuples based on the predicate.
- **Projection** $\Pi : \Pi_{c_1,c_2}(A) = \{a(c_1, c_2) | a \in A \wedge c_1, c_2 \in a\}$. Where a is a tuple in relation A and $c_1,_c 2$ are columns in a. The projection operator retrieves given columns of a relation. Selection operator deals with tuples while projection deals with columns.
- **Rename** $\rho : \rho_x(E) = \{x | x = E\}$. The rename operator returns the result of the expression E with the name x. E could be a relation or any of the above expressions.
- **Assignment** $\leftarrow$: $A \leftarrow E = \{a | a \in A \Rightarrow a = E\}$. The assignment operator assigns the result of the expression E to A. E could be a relation, a literal value or operation such as $A - B$.

2.4.3 Database Normalisation

Normalisation, which has been introduced by Codd in [28] is the process of organising databases and is built on the concept of normal forms [31]. The aim of normalisation is to eliminate data duplication which might cause data inconsistency. The term normalisation deals with the *functional dependency* between record's attributes except for the first normal form. The process of normalisation includes *decomposition* of relations by replacing one relation by several [51].

Consider attributes X and Y in relation R, the functional dependency is the expression $X \rightarrow Y$ where X determines Y or Y is dependent on X. For this, it is not possible to have two Xs with the same value that are mapped with different values of Y. Both X and Y can be more than one attribute. The word "functional" suggests there is a

function which takes a list of values for X in our example and produces unique values of Y for each X [51].

The first three normal forms were defined in [29]. The definitions of these normal forms are:

- **First normal form (1NF)**: Each attribute in a record contains only a single value. Every relational database must be at least in 1NF. This shows that first normal form deals with the *"shape"* of the records [62] but not with the functional dependencies.
- **Second normal form (2NF)**: Be in 1NF and each non-prime attribute must be dependent on the entire key (no partial dependency). A non-prime attribute is an attribute that is not part of any candidate key. A relation key is an attribute, or set of attributes, that uniquely identifies the relation. 2NF is violated when a non-key attribute is a dependent on a subset of the key. If there is no composite key in R, if R in 1NF then it is automatically in 2NF. Consider the following example:

student	module	semester	credit

  In this example, the key is {student, module}. While *semester* is dependent on both *student* and *module*, *credit* is determined by *module* only, hence this example is not in 2NF. For this to be in 2NF, the table needs to be split where *credits* attribute is moved to a new table where the *module* is the key as:

student	module	semester

module	credits

- **Third normal form (3NF)**: Be in 2NF and there is no attribute in R that is dependent on a non-key attribute. There is no transitive dependency on the key. Consider the following example:

lecturer	department	location

  The *location* attribute is determined by the *department* which is not a key attribute in this example. For this to be in a 3NF, the table should be split into two tables as:

lecturer	department

department	location

2.4.4 Transaction management

The database transaction is a single logical unit that may be formed by one or more CRUD (Create, Read, Update and Delete) operations. These operations must satisfy a set of properties in a multi-user environment. In [57], four properties of transactions are introduced. They are widely known as ACID properties for *Atomicity*, *Consistency*, *Isolation* and *Durability*.

Atomicity states that all the transaction must succeed or nothing at all. Consider the following example of transferring 200 from *account_1* to *account_2*:

```
update account_1 set balance = balance - 200;
update account_2 set balance = balance + 200;
```

The first statement will reduce the first account balance by two hundred and the second statement will add two hundred to the second account. If we consider this as one transaction unit, then both statements must execute correctly or none of them is executed. In case of a failure between the execution of the two statements, then the database must ensure that the first statement is rolled back so that no change is made to the database.

A transaction must preserve the database consistency, which means that each successful transaction must commit only the legal result [57] in which the transaction moves the database from one valid state to another. For the same example, if the sum of this two accounts before the two statements was 1000, then after the transaction it must be 1000 also.

Transaction events must be isolated from other transactions' events. The isolation concerns multiple transactions. Consider the following scenario where *account_1* originally has a balance of 500:

```
update account_1 set balance = balance - 200;
                                              select balance from account_1;
update account_2 set balance = balance + 200;
```

The select statement is executed by another transaction in between the two update statements. The isolation property specifies that the second transaction will get the balance of *account_1* as 500 even though the first update reduced the amount by 200. This is because that the first transaction hasn't committed the result yet and the second transaction only sees the committed result.

For any committed transaction, the system must guarantee its result durability. Without the durability property, a sudden system failure or shutdown might cause some recent updates to be lost. The update by the transaction in the previous example is permanent and any transaction that reads the balance of account_1 later will see it as 300.

Constraint	Description
Primary Key	Each row in the table is uniquely identified by this key and there are no two rows in the table that have the same primary key. The key could be formed by a single attribute or a combination of two or more attributes
Foreign Key	Is a primary key of table, say A, that is embedded in table B for *referential integrity* which means the value of this attribute in B must relate to an existing value of a primary key in A
Not Null	The value of this attribute must be provided
Unique	The value of this attribute (or a combination of two of more attributes) should be unique for the whole table and there are no two rows that have the same value for this attribute
Check	Used for domain constraint where the value of an attribute must be from a given set, It can be defined for one of more attributes in the same table or from different tables

Table 2.2: Constraint types in SQL

2.4.5 SQL language

Structured Query Language (SQL) is a language that facilitates and supports defining, manipulating, accessing and controlling relational databases [32]. The language takes its roots from set theory and predicate calculus like the formal methods mentioned earlier. SQL can be categorised into different sublanguages. The *Data Definition Language* (DDL) in SQL is used for defining data structure such as tables and views using `CREATE` statement. We use *Data Manipulation Language* (DML) in SQL to manipulate data in an existing database using `SELECT`, `INSERT`, `UPDATE` and `DELETE`.

Database security is delivered by SQL through two mechanisms, *Data Control Language* (DCL) and views. DCL in SQL is used for data access control by `GRANT`ing and `REVOKE`ing permission to access a particular object in the database to a user. The view, as mentioned earlier, can be used to hide sensitive information from particular users and is supported by SQL.

Transaction management in SQL is supported by the commands `COMMIT` and `ROLLBACK`. The commit command saves any changes to the database while the rollback returns the state of a database before the transaction was started or to a specified `SAVEPOINT` in the transaction that has been defined earlier.

SQL supports different *integrity constraints* that ensure the correctness and the consistency of the database. These constraints might relate to a table, an attribute in a table, or a relation between one table and another. Table 2.2 summarises the constraints supported by SQL.

2.4.6 Database programming technologies

SQL by itself does not provide a complex access and manipulation of data where there is a need for looping, branching, conditions, ...etc. Database programming can be achieved by different programming languages that provide libraries or integrations for database access and manipulation.

Some database management systems provide languages that are integrated with their database to overcome the limitations of SQL such as PL/SQL for Oracle database [45]. PL/SQL, which stands for "Procedural language for Structured Query language" provides a complete programming solution for Oracle database. It is not a stand-alone programming language as it is embedded language to the database. Its integration with the database makes it a high-performance language compared to the other stand-alone languages and interfaces for databases [45].

Java programming language provides an interface to interact with the database using Java Database Connectivity (JDBC) [17]. As JDBC is not embedded to the database, it first needs to establish a connection to the database before executing any SQL statements and retrieving results. While JDBC basically embeds SQL statements in Java programs, it also supports calling PL/SQL procedures.

Object-relational mapping is an automated persistent of program object to relational database application [20]. This is achieved by using a meta-model that describes how to map objects to the relational model. By using an ORM tool such as *Hibernate* [82], a developer can build the database application once for any dialect of SQL. While ORM has it own query language that independently accesses different relational databases, it also supports native SQL query if needed by the developer. This reduces the mismatch between the relational database and the object-oriented application and provides a powerful data access layer.

2.5 Related Work

In this section, we are going to review some of the literature that relate to our work. We will try to take a broader view of the related work without narrowing our search on one formal method, Event-B. First, we review the related work of similar research and tools that generate database code from a formal model. Following that is some literature defining semantics of SQL.

2.5.1 Database formalisation and code generation

Much existing literature covers the concept of formalising database specifications and translating them into implementation. A significant portion of this work addresses specific parts of database design.

The DAIDA [60] was a project on correct database applications with contributions from different countries and institutions. It provides a full engineering process from requirements to design to implementations. The researchers in DAIDA used DBPL for implementation which is a procedural language designed for database applications. Part of this work is the derivation of database code from formal specifications as by Gunther et al. [56] starting from TDL conceptual language that is coupled with the B method specification and DBPL implementation. The TDL model is refined toward an implementation in DBPL. The transformation from TDL to DBPL is not automated [87], hense, there is no automated code generation.

Schlatte and Aichernig in [88] present a method to create a VDM formal model for relational databases. The method transfers an entity-relation diagram to VDM-SL constructs along with additional constraints. The method includes formalising database structure, SQL data types, database operations and run-time checking using triggers. The approach presented by Schlatte and Aichernig is by translating ER diagrams to VDM. They do not provide a translation or a code generation from VDM to relational database implementation.

Barros in [37] and [19] translates Z specifications to relational databases. The work represents a partial description of the mapping process and describes the generated prototype. It covers different CRUD operations as well as transactions, sorting, aggregations and other database components. The work translates the textual specification in Z and there is no graphical model of the representation in which the process should start with. The work does not provide a mechanism for modelling databases systems in different abstractions and refinements. Moreover, the automatic code generation of the databases is not addressed in Barros work.

In [63], the authors present a method that translates Z notation to SQL and Delphi programming language by extending the Delphi library to support Z structures. While the work provides a formal basis to build a database prototype, it does not implement a tool for the translation. Only the formal definition of translation function is presented in the paper without the implementation of these functions. There is no tool provided in which modellers can use to automatically generates database code for the formal definitions.

In [80], Polack and Stepney investigate the usage of Z generic in developing information systems to separate the concern of structure and data types. They proposed the use of the generic types instead of refinement as the refinement requires considerable effort and

higher cost which is, to them, only justified for critical application. Their work does not address the development of information systems in full details.

Laleau and Mammar in [68] and[69] present a tool that refines a UML specification into a B model and then to a database application. The work supports a modeller to design a UML diagram and then translate that model to a B specification and on to Java and SQL code. They designed a dedicated prover for database refinement in B [67]. Only the first normal form is guaranteed in Laleau and Mammar's work and other normal forms are up to the UML designer to satisfy. The refinement process supported by Laleau and Mammar work is toward a database implementation of B specifications. Their research lacks an approach for gradually modelling relatively large and complex systems in layered refinement where in each refinement level, new concepts and requirements are introduced. The derived Java code in their approach is done outside any formal environment in which its correctness can not be ensured. Moreover, their tool does not provide full automation of code generation from the B specification. The translation of B implementation level into Java and SQL is performed manually. Laleau with Polack in [64] investigated the tractability between the UML model and B method to support the development of information systems with a tool to support rigorous information system specification.

Davies et al. in [36] explore how to formalise object database specifications and constraints using Booster notation [33]. They use a model-driven approach to automatically generate object-oriented databases. An extended version of B method and Object Constraint Language (OCL) [104] are used to implement the object database in which methods are implemented as functions in C language. There work is extended in [35] where in the final stage of the development process the object model in Booster is compiled to relational model with procedure and SQL. The specification language used in [35] is Z notation.

Wang and Wahls in [103] developed a Rodin plug-in that translates Event-B to Java and JDBC code to create and query a database. While, to the best of our knowledge, this is the only work that translates an Event-B to database applications, it has major limitations. The results shown in [25] identify major performance issues as well as the issues with preserving database integrity as in [8]. Their tool, Event-B2SQL, generates a single Java class for the entire system which becomes very large class (thousands of line of codes) which becomes impractical for generating code of systems with many operations. A user might have to increase the Java heap value above 512MB in order for the generated code to be written in one Java class. This has been concluded by experimenting Event-B2SQL tool on the case studies presented in Chapter 6.

The authors in [2] presented a process called Object2NoSQL that transfers UML class diagram to column-based NoSQL databases. The process first transforms the UML

model to logical model, then transforms the logical model to physical model that is independent on its implementation from the logical model. This can be used to study the extension of our work to support NoSQL databases. A round trip of this process which generates the UML diagram from NoSQL database is presented by Brahim et al. in [21].

Vega et al. [38] present Mortadelo, a framework that automatically generates NoSQL code from a data model. It offers a model-driven transformation, that starts from a data model and provides an automatically generated design and implementation for different NoSQL data store. While they provide a tool that support their framework, there is no formalism or formal semantics in the process.

The authors in [47] presents an approach to generate transactional web applications based on a conceptual model of the system. The code generated is for the user interface and business logic as well as data definition language for the databases. The SQL for manipulating databases such as insert, update and delete is not generated. similar to [38], there is no formalism, formal translation rules or formal semantics presented.

In [73], Musleh et al. target the generation of SQLite database in Android application using a tool they developed. The tool, Android SQLite Creator, generates automatically an SQLite database for android as well as the classes that perform read/write operations. A preliminary experiment evaluation was performed over small sized Android databases which showed that the tool can be usable.

Comparison to our approach

There exists some similarities between some of the reviewed work and our approach and design. Laleau and Mammar in [68] and[69] start the model from UML which is translated to B. As we start from UML-B with its semantics in Event-B, the invariants and events in UML-B are given as an Event-B. However, how to model static constraints such as not null and uniqueness remains the same. Similar to them, only the first normal form is guaranteed in this thesis, as satisfying the other normal forms is up to the modeller. The work by Jim Davies and his colleagues in [35, 34, 102] provide a similar approach as they use successive versions in development in which the next version may add or remove components to the initial one. However, our approach provides different concepts that can be applied to incrementally design database system and break the complexity into different refinement levels which is guided by different case studies. Some components and operations are not covered in a full scale in [35, 34, 102] such as using joins in select statements. While they represent the model as classes, attributes and associations, UML is a descriptive only in their papers and the model does not start from a UML diagram.

While DAIDA provides a solution for a correct database application, its implementation language is specific to DBPL with less usability for general audience who uses SQL and its procedural languages for the database application in current DBMS. Our approach tries to reduce the gap between verifiable software community and database community by providing clear guidelines to model databases in UML-B and Event-B and support that with automated code generation that the community can understand and utilise. Also, like others work in this area, DAIDA does not provide an incremental approach which can help to break the complexity of database systems in different abstract levels where refinements extend on the abstract model with more requirements but preserve the abstract invariants.

In general, none of these research provides general guidelines or an approach for how to model relational databases in formal methods using patterns. Moreover, they do not address layered refinement where in each refinement a modeller can introduce new operations on the newly introduced variables. The approach of modelling complex systems in different refinements is an important aspect and contribution of this research. There is still a gap in providing a clear methodology that is coupled with the formal specification which should simplify the process of rigorous database design. As stated by Gunther et al. [56], this point is always left unclear. A clear methodology of refinement steps is needed. There is also a gap in providing a tool that fully automates the process of database generation from a formal model and provides code that can be imported in major database management systems. While, admittedly, these formal methods are not used to evaluate the performance of database systems, however, the automatically generated code - when available - should be evaluated for its performance so that it can be used efficiently by the end users.

While this thesis build on the literature on how to model specific components such as attributes, associations and operations of database in formal models, it is centred toward incremental approach for database design and the automation of the generated code.

2.5.2 Formal Semantics of SQL

Defining formal semantics of SQL language is an important field of research that is needed to show the correctness proofs and the soundness of the generated SQL. Much of the literature define semantics to translate SQL queries to formal model but not all SQL statements.

Formal semantics for a subset of SQL has been defined in [70] using VDM. While the work does not tackle database views and built-in functions such as aggregation functions, it does consider basic SQL operations. The syntactic domain of their formal definition in VDM includes SQL commands: `create`, `expand`, `drop`, `select`, `update`, `insert` and `delete`.

In [74], the authors define set of rules to translate SQL queries to formal model using the *Extended Three Valued Predicate Calculus (E3VPC)*. This aims to address the issue that SQL is based on a three-valued predicate in which there is "unknown" predicate that represents a null value. The work gives a full definition of the formal semantics of SQL queries with special attention to the unknown or null values. The E3VPC has been used in [101] to provide a soundness proof for translating Event-B actions to SQL statements.

Similar to [74], Gegolla in [52] provides a formal semantics for SQL queries without considering other operations. He first defines the semantics of SQL core without the aggregation functions and `having` clause. The semantics are determined by a function `sql2erc`: `SQL-Query` $\rightarrow$ `CalculusQuery`. The semantics are limited to `select` statement only.

In [40], Edmond considers Z and SQL in conjunction and defines the relation between them as they share the same root. Both Z and SQL are based on set theory and predicate calculus. Edmond in [40] examines the mapping between Z specifications and SQL implementation.

Guagliardo and Libkin in [55] consider the basic subset of SQL queries for its semantics and applications. They validated their semantics on a randomly large number of generated queries and databases. They focused on the correctness of the queries on databases and with null values as in [54]. The manipulation on databases is not treated in [55].

These literature provide a basis to define the semantics of relational database code in Event-B. The semantics should be extended to cover all database operations defined in stored procedures and how they can be mapped to Event-B events.

2.5.3 Database patterns

There exists some literature that discuss design patterns for databases. These literature try to investigate different case studies of database systems and derive patterns used to model them. We aim to look at these and try to incorporate these patterns in Event-B when applicable. Gamma et.al in [50] define the design patterns as reusable solutions to common problems. Patterns are problems that occurs over and over in the environment and then describe a solution to that problem as described by Alexander in [14] for town and constructions. The same definition applies to software design patterns. While the design patterns are well understood and integrated into the development of object-oriented programming, we need to define some design patterns when modelling relational databases. There exist some literature that define different patterns for the relational database with no relation to formal modelling.

Vitacolonna in [100] introduced conceptual database design patterns including multiple roles and hierarchy patterns. A list of twenty-four design patterns for databases is identified in [59] from analysing open source applications.

Stathopoulou and Vassiliadis in [94] identify three main patterns for databases: *pivoting*, *materialisation* and *generalization\specialization*. The pivoting pattern separates similar attributes in one table to key-value pairs in a new entity. Materialisation defines a relation between abstract classes and concrete classes. The last pattern defines is-a relationship between entities.

All these patterns defined in the literature are useful when studying and modelling individual cases by providing a solution of how to model database systems. However, non of these deal with breaking a database system into incremental abstractions that can be used in any database system. We aim at providing a pattern of how to model databases incrementally and show that it can be applied to different database systems.

2.6 Gaps and Improvements

After reviewing the literature for database formalisation and code generations, we can outline the following possible improvements:

- **Incremental development of information systems**: The related work discussed the formalisation of the database components and operations, but not how to model them with abstraction and refinement. This includes defining patterns for modelling information systems in incremental manner.
- **Translation rules for incremental method**: As the modelling approach is incremental, the translation rules should reflect on the approach. It should defines how to translate and generate the new elements in the refinement while preserving the abstract.
- **Round trip semantics**: By defining the semantics of the generated code in the same language as the semantics of the source of the generation (Event-B), we can reverse the translation rules
- **All CRUD operations**: We need to tackle and generalise all CRUD operations in both modelling and semantics. The work by others omitted some operations such as update in [103], or joins in select in [68, 69]. The semantics was focused on select.

However, this research carries out some concepts from the literature. These includes:

- **Modelling individual components**: Modelling classes, attributes and association remains the same as in the literature. This includes how to model static constraints such as *null* or *not null* and *unique*.
- **Starting from UML**: We start the model from UML-B, similar to [68, 69, 35] were they started from UML.
- **Language for generated code**: We carry on using SQL as the language for the generated code as it is the used language to model relational databases. The operations are generated as stored procedures. The work in [103] and [68, 69] used Java methods to model operations, while we use stored procedures similar to [35].

2.7 Conclusion

In this chapter, we introduced the main concepts that relate to our research. Our interest is to focus on modelling information systems using graphical representation and translate it to relational database code. We choose relational database due to its strong foundations in literature and its relations and roots in set theory as well as its dominance as database model in the market. The independence between conceptual, logical and physical layered of relational database is an advantage to consider as the we don't need to worry about the underlying implementations in the database management systems. We use Event-B as a formal specification as it has simpler notations with an open toolkit and a range of supporting tools. The refinement mechanism offered by UML-B which is translated to Event-B enables the modeller to gradually specify the system using graphical representation. The possibility of having new events in a refinement provides an advantage for Event-B and UML-B to be used for the modelling as we can gradually introduce different operations on databases.

The subject of formalising database design has been studied in much of literature. All the reviewed work deals with formalising relational databases except for [36], which addresses object-oriented databases. Modelling tables and attributes and generating a database for them were commonly discussed in the reviewed literature, with some work on modelling CRUD operations. However, a clear methodology that accompanies the formal specifications is left-out. There is also a gap in the full code automation that supports such methodology. Where is the automated code generation provided as in [103], the analysis as in [25] and [8] show that the generated code performs slowly. The proposed research should answer the question of how to model databases in a UML-B like approach and preserve its consistency and integrity in its transactions? What are the steps should the users start with when specifying database system? How can the model be scaled and extended with further features that reflects new requirements? And how the code can be generated from the model and its extensions? Moreover, addressing

the semantics of generated code is an important aspect in our approach. It is essential to show that generated code is a refinement of the generated Event-B model from UML-B.

Chapter 3

Modelling Information System in UML-B

3.1 Introduction

In this chapter, we describe our approach of modelling information systems using UML-B class diagram and Event-B. We start by defining how to structure our model using abstraction and refinement in UML-B and then discuss modelling different CRUD operations as events in Event-B. When modelling the structure of a system, we use some patterns to address common issues. The modelling approach is guided by a case study in which we first model the case study in different refinements, then we derive general guidelines that are independent of any case study.

3.2 SRES case study

In this paper we use a Student Enrollment and Registration System (SRES) case study to illustrate our approach to modelling information systems using UML-B in different refinement levels and translating UML-B model to database implementation using SQL and PL/SQL. The case study addresses the issue of various departments in a university, each of which has different programs of study and modules. A student should register in various modules that are offered for the program in which they are enrolled. Each module is taught by one or more staff. The case study tries to cover a wide range of possible class types, relations and functions to determine a range of rules to model a database system in Event-B.We use this case study to show examples of the modelling approach, the case study is detailed in Chapter 6.

3.3 Modelling database through abstraction and refinement

This section shows how to structure a database model in UML-B using our approach of layered refinements. In order to illustrate our approach clearly, we need to introduce some concepts that we identified when modelling three case studies. These case studies concern a Student Enrollment and Registration System (SRES), a Car Sharing System and an Emergency Room System.

Following the modelling of the case studies, we can generalise guidelines for modelling information systems in layered refinements by extending each refinement with extra features and complexity. The guidelines define both the structure of the model as the operations on its variables. We define how to model CRUD (Create, Read, Update and Delete) operations in Event-B with minimal mathematical notations that can be later translated automatically to database code.

3.3.1 Primary, Secondary and Attribute Classes

Modelling these case studies introduces different class types that can be used in defining a refinement strategy when modelling database systems. These classes are *primary*, *secondary* and *attribute* classes. Primary classes are the classes that describe the main entities of the system and can be seen in classes that describe people such as *Student* and *Staff* in SRES, *Member* in Car Sharing, or *Patient* and *Doctor* in Emergency Room. Primary classes can also illustrate activities such as *Module* and *Treatment*, objects such as *Car* and *Room*, or an organisation such as *Department*.

Secondary classes are the classes that relate two or more primary classes together. Examples of such classes are *Registration* of a student in a particular model, *Booking* of a car by a member, and *Admission* of a patient by a doctor into a room.

Attribute class is a class that represents a complex attribute of a primary class that consists of multiple attributes. An example of such a class is the *Address* of a person which by itself has attributes such as street number and postcode. For each concept and refinement, this section will show it using an example from a case study.

The primary classes define the entities that can stand by themselves with no needs for other classes to define their meaning or objectives. As compared to that, the secondary classes are transactional in which they are fully dependant on primary classes and they can not hold meaning outside the primary classes. A hotel is a primary class that describe the main entity of hotel management system, but we can understand it by its own as an entity. On the other hand, booking is a secondary class in which we can not have any instance of it that does not depend and relate on a hotel and a room.

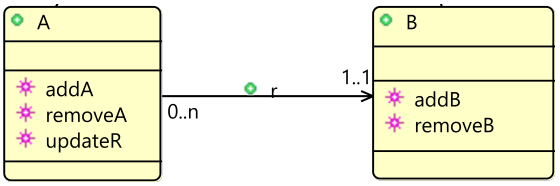

Figure 3.1: Abstract model of UML-B class diagram

The attribute class is just a complex data type that is an object by itself with many attributes to define it. A complex attribute is composed of sub attributes that together define a single attribute of another class. By defining these types of classes, we can easily break down a whole system into sub components and focus on modelling each component individually.

Data on the primary classes are less likely to change compared to the secondary classes as student information in SRES is less likely to change, unlike his/her registered modules. The number of instances in the secondary classes are more than in the primary one. An example is the single hotel with a hundred rooms. The hotel and room classes data remains static, while booking class keeps growing with thousands of records.

This provide a novelty of distinguishing between different classes and categorise them into three clear distinctions which helps in modelling large data intensive systems.

3.3.2 Modelling primary classes and their associations

Modelling information systems in our approach is done in different refinements that are defined in successive steps. The abstract model of the system may consist of different primary classes and the associations between them. Figure 3.1 shows an abstract UML-B model of two classes *A* and *B*. The diagram shows the classes and association between them, *r*.

The abstract model needs to include all required events that modify the state of variables introduced in that model. Such events are *add*, *update* and *remove* events, where add events insert new elements in the class, update events change the value of one or more of its attributes and remove events delete one or more records from it. Examples of primary classes in the SRES case study are *Student*, *Module*, *Staff*, *Program* and *Department*. Figure 3.2 shows our abstract UML-B model of the SRES case study.

The model includes all events that change the state of its classes instances, attributes and associations. Add events are set as constructor event types in UML-B. For each class such as *Program*, all associations from it to another class are added in its constructor event. For example, *addProgram* for the *Program* class has a parameter for *offeredBy* and an action to map it to *Program* instance as in action *act2*.

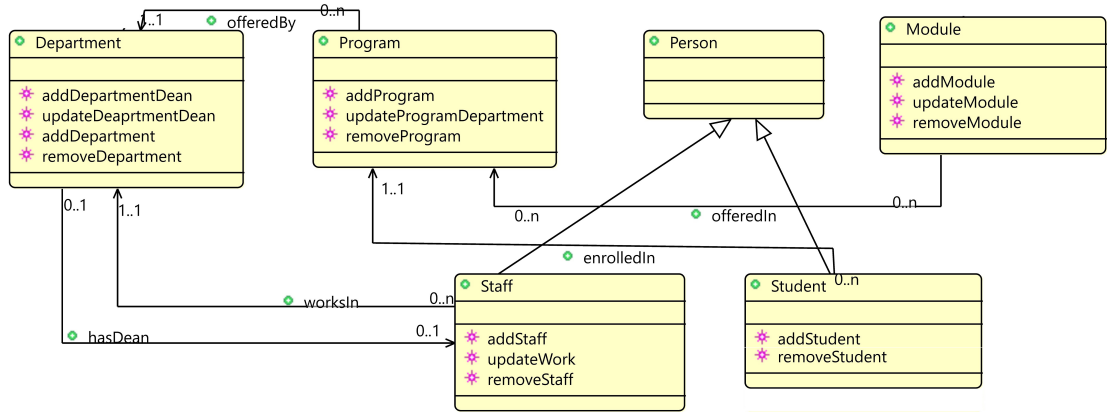

Figure 3.2: Abstract model of SRES entities and relations as a UML-B class diagram

$addProgram \mathrel{\widehat{=}}$

any

$this\_Program$

d

where

grd1 : $this\_Program \notin Program$

grd2 : $d \in Department$

grd3 : $this\_Program \in PROGRAM$

then

act1 : $Program := Program \cup \{this\_Program\}$

act2 : $offeredBy := offeredBy \cup \{this\_Program \mapsto d\}$

end

The model also includes inheritance between super and sub-classes. An example is the *Staff* and *Student* classes which are sub-classes of *Person* class. The subclasses could have some explicit associations for each that are not shared between them.

Modelling the relation between *Staff* and *Department* introduces a *circular dependency* in which each class relates to the other one forming a circle as a *Department* has a dean and a member of *Staff* works in a *Department*. By modelling this in Event-B and specifying each association as a total function, both adding *Staff* and adding *Department* events are not enabled as each requires an instance from the other class. To avoid this, we weakened one association, *hasDean*, by making the association optional, or partial function. Such issue (circular dependency) may not be detected in design phase using traditional database modelling approach in ERD. By using Event-B with model checking facilities provided by ProB in Rodin enables the modeller to detect and fix such issues in an early stage of the system development.

3.3.3 Adding attributes and extending events

In a refinement, each class will have attributes that add some details about the class such as *program_code* in class *Program*. After defining these classes and their associations, we refine the model by adding different attributes to each class and defining their constraints. The constraints such as *not null* and *unique* constraints can be defined by defining the attribute as *total* and *injective* functions when added in UML-B as in Figure 3.3 for the attribute *program_code* in *Program* class. Adding this as a refinement is because we prefer to have the general structure of the classes and associations between them first, then to add details to each individual class. Figure 3.4 shows this refinement in our approach where we added the attributes to the classes. In this level, data types such as date and variable characters are defined as carrier sets and used as types for different attributes in various tables. All events are extended to include the new attributes such as *program_code* and *progran_name* in *addProgram* event:

$addProgram \mathrel{\widehat{=}}$
extends *addProgram*

any

p_code
p_name

where

grd4: $p\_code \in PROGRAM\_CODE$

grd5: $p\_name \in VARCHAR$

then

act3: $program\_code := program\_code \cup \{this\_Program \mapsto p\_code\}$

act4: $program\_name := program\_name \cup \{this\_Program \mapsto p\_name\}$

end

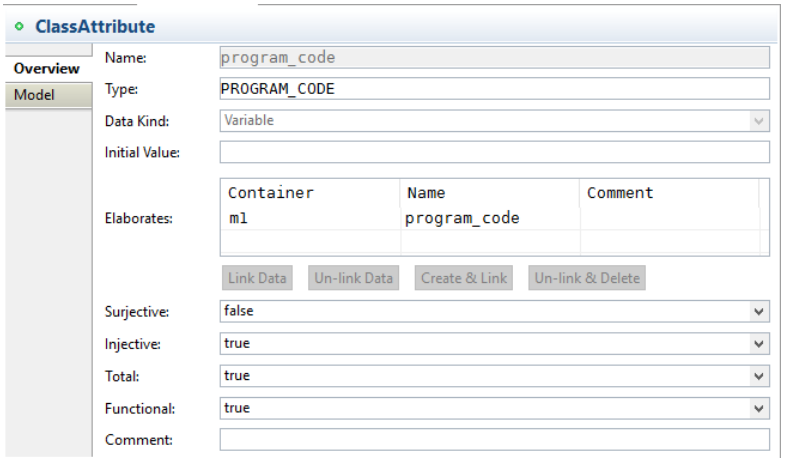

Figure 3.3: Setting class attribute in UML-B

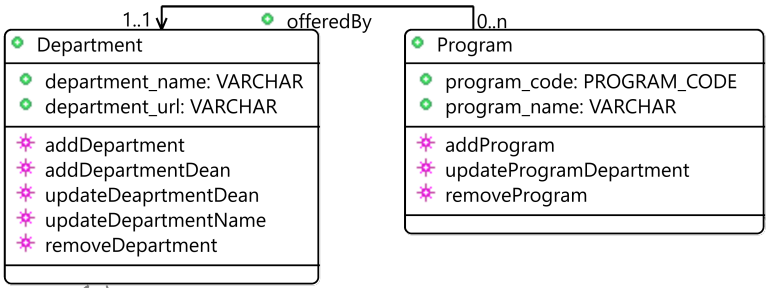

Figure 3.4: Adding attributes to the main classes

3.3.4 Modelling secondary classes

In a further refinement we introduce the secondary classes to the model in which they associate between primary classes or are instances of a primary class such as the *Registration* in Figure 3.5 which is a class that describes a Student taking a Module in a specific time and the *Module_Runs* which specifies modules running at given year and semester.

3.3.5 Modelling attribute classes

Another distinction is introduced in this model: the attribute classes. An example is *Address* class, which is an attribute type that is associated with *Person* as shown in Figure 3.6. The association is directed to and not from the Person class giving the assumption that each person might have $0..n$ addresses. This concept, the attribute class, can be introduced in any refinement. The association is defined from the primary/secondary class to the attribute class.

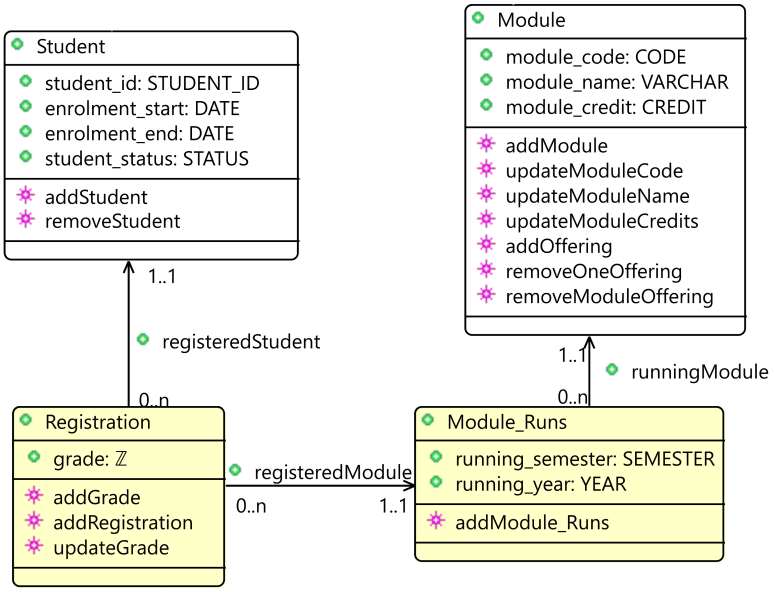

Figure 3.5: Adding secondary classes

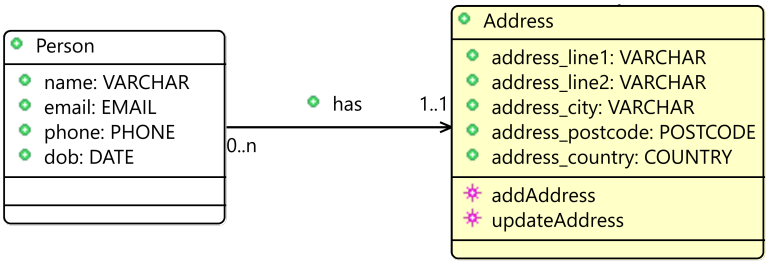

Figure 3.6: An example of an outer entity

3.3.6 Modelling historical data

In some systems such as SRES, there might be a need to move some historical data into different tables from which it can be retrieved later. Moving the data is necessary when the table becomes large and a full scan becomes very expensive. For example, after a student has completed and passes his/her registered modules, instead of keeping all the records in the original or live table, the completed records will be moved into a historical table. While the new table is a subtype of the same supertype as the live table, they are not bound to the live one. *Completed_Student* and *Completed_Registration* in Figure 3.7 are new classes that represent archives of the records of completed students. When a student finishes his or her degree, the information is moved to *Completed_Student* and the history of the registration is moved to *Completed_Registration*. We remove an instance from the *live* class and add it to the *historical* one in one event which is atomic in Event-B. The historical classes might have new attributes that are not in the live classes such as *d_date* which specifies the date of completion. This refinement can be introduced later in the system as it concerns archiving old data in which the modeller do not need to worry about it when the modelling starts. Similar refinement could include

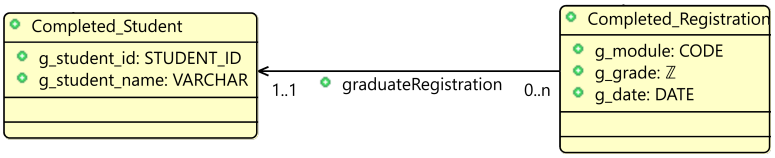

Figure 3.7: Historical data classes

classes that are used to track the changes in the database in which it keeps logs of all the changed data and by whom.

The authors in [59] mentioned *Archived Data* pattern which is similar to the historical pattern, where they represented it as a type of database tables that occur in some scenarios. This thesis present this as a refinement pattern in UML-B that preserves the constraints of the model that it refines.

3.3.7 Modelling Operations

UML-B model provides three kinds of events: *constructor*, *destructor* and *normal*. Our method and tool try to map these events to procedures that perform CRUD operations on the database.

For a constructor event in UML-B, an instance of the class is created along with its attributes and associations. A destructor removes an instance of a class with all its attributes and associations as in *removeA* using domain subtraction. Domain subtraction $\{this\_a\} \lhd x$ removes all pairs of x whose domain value is $this\_a$.

$removeA \;\widehat{=}$

any

$this\_a$

where

grd1 : $this\_a \in a$

then

act1 : $a := a \setminus \{this\_a\}$

act2 : $x := \{this\_a\} \lhd x$

act3 : $r := \{this\_a\} \lhd r$

end

The destructor event *removeStudent* removes the tuple of students who are enrolled in p program.

$removeStudent \;\widehat{=}$

any

ss, p

where

grd1: $p \in Program$

grd2: $ss = \{s \mid s \in Student \land enrolled(s) = p\}$

then

act1: $Student := Student \setminus ss$,
$student\_name := ss \lhd student\_name,\ enrolled := ss \lhd enrolled$

end

The *removeStudent* event removes all attributes and associations to the class instance that the event removes. The event guards enable the destructor event only if there is no association to that instance from another class. If the student has a registration associated with him/her, the event will not be enabled and the student will not be removed. Instead of removing a student, we first remove the registration associated with this student, then we remove the student from the set. This is preferable in our approach over removing all instances related to a student, or any class instance, so that a database user knows that there are still some associations targeted to that instance.

Normal events in UML-B can be used to update or override set elements as in *updateA* which use function override to update association r. Function override of r means that the range value that the domain $this\_a$ is mapped to is updated to the value new_r. Normal events can be used also to query information from the classes as in *getA* which retrieve all a's whose x value is z. Let's consider a *Rate* class in a car rental system

Event $updateA \mathrel{\widehat{=}}$
any
$this\_a$
new_r
where
guard1 : $this\_a \in a$
guard2 : $new\_r \in b$
then
action1 : $r := r \lhd\!\!+ \{this\_a \mapsto new\_r\}$
end

Event $getA \mathrel{\widehat{=}}$
any
a_list
$this\_x$
where
guard1 : $a\_list \in \mathbb{P}(a)$
guard2 : $this\_x \in \mathbb{Z}$
guard3 : $a\_list = x^{\sim}[\{this\_x\}]$
then
$skip$
end

as in Figure 3.8. The event *updatePerh* overrides the hourly rate of a car based on the kilometre rate by doubling the rate per hour for every tuple where the rate per kilometre is less than two.

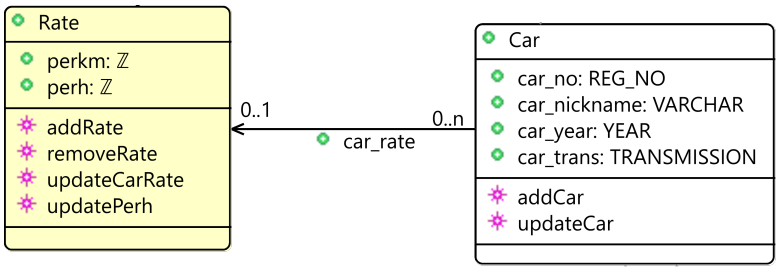

Figure 3.8: Car rental rate class with update event

updatePerh $\widehat{=}$

any

rs

where

grd1: $rs = \{r | r \in Rate \land perkm(r) < 2\}$

then

act1: $perh := perh \lhd\!\!+ \{t | t \in rs \cdot t \mapsto perh(t) * 2\}$

end

3.3.8 Introducing query events

While the events that modify the state of machine variables are introduced in that machine, we introduced *get* events in a later refinement because they might require a complete structure of different classes in order to retrieve valuable data. The event, *getDepartmentStaff*, shows a simple example of modelling queries in Event-B using equality and relational image inverse as in *grd3*. The event reports all the Staffs working in a given department. The query, or get, events do not have actions as they just report some data from the model.

getDepartmentStaff $\widehat{=}$

any

d

$staff\_list$

where

grd1 : $d \in Department$

grd3 : $staff\_list \in \mathbb{P}(Staff)$

grd3 : $staff\_list = worksIn^{\sim}[\{d\}]$

then

skip
end

Queries might not be as direct and simple as the example above, however, the pattern is still similar. A modeller models the result set as a local parameter to the event. Let's assume we want to get all students enrolled in any program offered by a department d. This query involves two queries; 1) getting all programs offered by a department, and 2) getting all students enrolled in that program. As 2) restrict the result of enrolled students to those in 1), we use the restriction for the second, while the first query uses the same pattern as in *getDepartmentStaff* event. This **sub-query** can be modelled as:

getDepartmentStudent $\widehat{=}$

any

d

$student\_list$

where

grd1 : $d \in Department$

grd3 : $student\_list \in \mathbb{P}(Student)$

grd3 : $student\_list = dom(enrolledIn \rhd offeredBy^{\sim}[\{d\}])$

then

skip

end

Another example of complex queries that can be modelled in Event-B is using set operations; union and intersection to group similar results from two queries. The union and intersection must be for the same types. An example is to list all staff of department d along with the deans of all departments. This includes two queries that both result a Staff type. It can be modelled as:

getStaffAndDean $\widehat{=}$

any

$staff\_list$

d

where

grd1: $staff\_list \in \mathbb{P}(Staff)$

grd2: $d \in Department$

grd3: $staff\_list = worksIn^{\sim}[\{d\}] \cup ran(hasDean)$

then

skip

end

To get the list of staffs who work for department d except for the deans, we replace the union by intersection operator. This forms a general pattern to model different queries in Event-B.

3.3.9 Modelling views

A database view is a *virtual* relation that is computed by operations on another relation(s). The difference between a view and a relation or table is that we only store the definition of the view instead of storing the actual values. Then each time a view is used inside an SQL database, it will be computed based on its definition. As the view can be computed like a normal relation, the relational algebra can be used to define the structure of a view and provide algebraic operations on it. As the view is a relation, it can operate on another view. Also, a view might be updatable in some cases. While it is very important to model the views correctly in order to support them, the reviewed literature and research do not provide any modelling and specifications for them in formal methods. In our approach we model the views as normal classes as there is no special representation for them, so far, in UML-B.

A view can project a single class or it can be a projection of multiple classes associated together. To distinguish them, we will call the first one a *simple* view and the other a *complex* view. If we think about deriving a whole class, then modelling a simple view should be a straightforward task as we can model it as a subtype of the main class. However, the derivation is done at an attribute level where attributes in the view are derived from attributes in the base class. To achieve this, we add an invariant that defines the derived attributes in the view.

Figure 3.9 shows an example of how this can be modelled where we defined *View* class as sub class of A. Then we added the attribute *derived_x* to be derived from x. The following invariant *viewInv1* specifies that *derived_x* is an x with value greater than or equal to seventy.

viewInv1 : $derived\_x = \{a \mapsto z | a \mapsto z \in x \wedge z \geq 70\}$

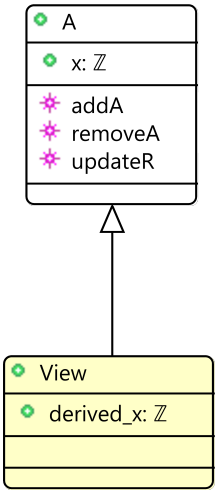

Figure 3.9: View with derived attribute

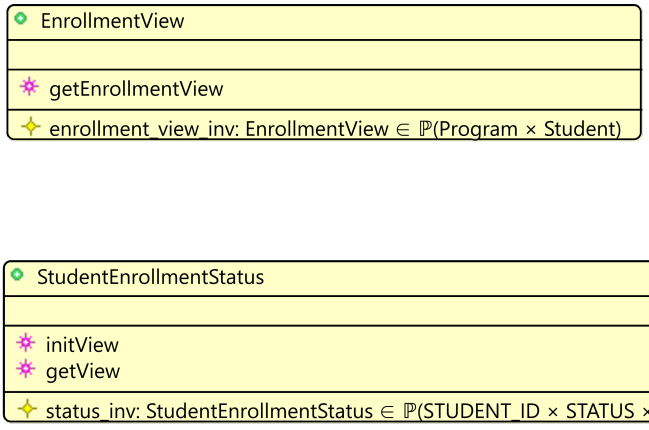

Figure 3.10: Modelling views in SRES

Another view, *EnrollmentView* is derived from two classes, *Program* and *Student*. We define the view as the Cartesian Product of the two classes as in the invariant *enrollment_inv*. Then in we create an event that gets the exact value of the view as in *getEnrollmentView*.

`enrollment_inv`: $EnrollmentView \in \mathbb{P}(Program \times Student)$

$getEnrollmentView \mathrel{\widehat{=}}$

where

`grd1` : $EnrollmentView = \{p \mapsto s | s \mapsto p \in enrolledIn\}$

then

$skip$

end

3.3.10 Association splitting

While the association between two classes in UML-B can be of a type relation which is a many-to-many association, relational database model does not support direct many-to-many relationships. We need association splitting to make the formal specification closer to the implementation. For any relation in Event-B that is a many-to-many association between two classes, we introduce a design pattern, *association splitting*, by refining it into a new class with two functions to the other two classes. This pattern as in Figures 3.11 and 3.12 shows the refinement of relation R to two functions $R1$ and $R2$ from a newly created intermediate class C to A and B. The following gluing invariant, *inv1*, specifies that R is equal to the relational composition of inverse $R1$ ($R1^{\sim}$) and $R2$:

`inv1`: $R = (R1^{\sim};R2)$

Since a relation does not have a duplication in pairs, the refined functions $R1$ and $R2$ must satisfy the same uniqueness of R as in *inv2*.

`inv2`: $\forall a, b, c1, c2 \cdot c1 \mapsto a \in R1 \wedge c2 \mapsto a \in R1 \wedge$
$c1 \mapsto b \in R2 \wedge c2 \mapsto b \in R2 \Rightarrow c1 = c2)$

The second invariant specifies that we cannot have two Cs that both refer to the same a and b. This forms a *composite* uniqueness in which the uniqueness is not about a single value, but the combination of multiple values.

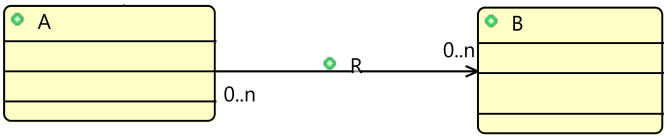

Figure 3.11: The abstract model of a relation R

An example of using such pattern is in the SRES case study where the offering of Module(s) in Program(s) is a many-to-many relation. The relation is refined into a new class which is the source of two functions that are directed to Module and Program classes as in Figure 3.13. The abstract *offeredIn* becomes a class, called *Offering* in this example, with two outward associations: *module_offering* and *program_offering*. These two invariants are included in the refinement:

`offeringInv2`: $\forall m, p, o1, o2 \cdot o1 \mapsto m \in module\_offering \wedge o2 \mapsto m \in module\_offering \wedge$
$o1 \mapsto p \in program\_offering \wedge o2 \mapsto p \in program\_offering \Rightarrow o1 = o2)$

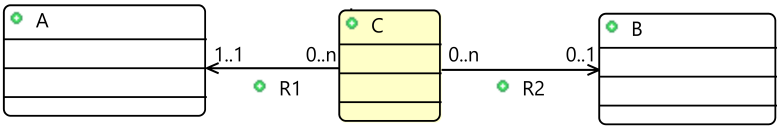

Figure 3.12: The refinement of relation R to R1 and R2

offeringInv1: $offeredIn = (module\_offering^{\sim}; program\_offering)$

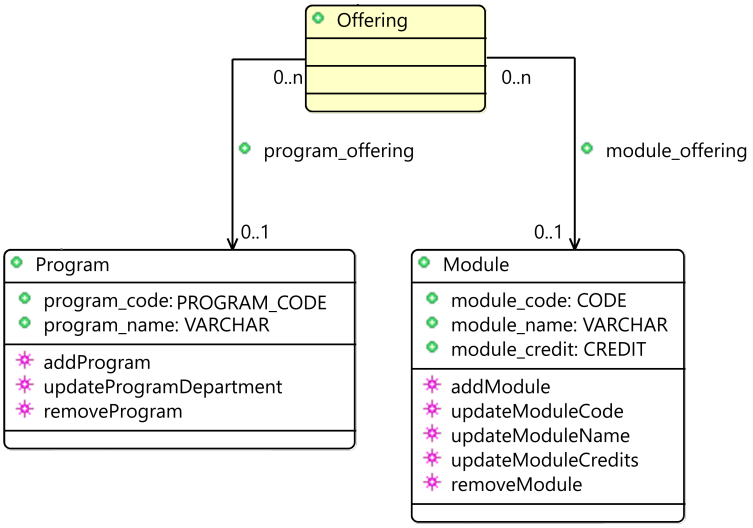

Figure 3.13: OfferedIn association split

Without the second invariant: *offeringInv2*, we could have two instances of Offering that are both associated with the same Module and Program which becomes a duplicate record.

The association splitting is a well known practice in relational database design. In databases, any relation between two tables that is many-to-many is mapped to an intermediate table with references to the previous tables. The novelty in this pattern is its definition and application in Event-B as a refinement which includes refining the events that used the relational association and providing the gluing invariant to prove the correctness of the refinement.

3.3.11 Model verification

By modelling databases in UML-B and Rodin, we introduce formal verification for our database models. Database elements such as tables, views or attributes have some properties that must always hold true. These properties are modelled as *invariants* in which they must be preserved by all events. Let's assume we have a requirement that Students in Figure 3.14 can only register in Modules offered by the same program of study they are enrolled in. This can be modelled by invariant *inv1* which applies to all instances of the registration class. The invariant becomes universally quantified in Event-B:

`inv1`: $\forall m, s \cdot s \mapsto m \in registeredStudent^{\sim}; registeredModule \Rightarrow$
$runningModule(m) \mapsto enrolledIn(s) \in offeredIn$

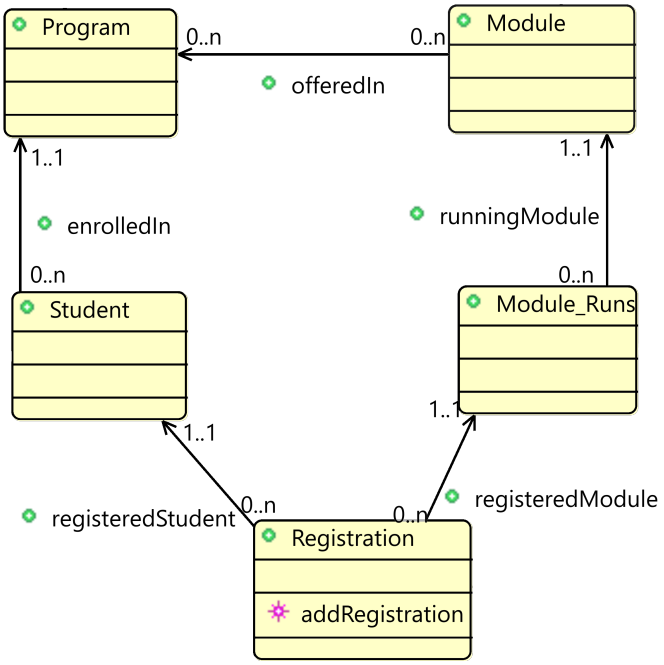

Figure 3.14: Student registration and enrollment

An event that adds a registration must preserve this invariant by having *grd4* that ensures the module to register the student for is offered in the student program of study:

addRegistration $\widehat{=}$

any

$this\_Registration$
m
s

where

grd1: $this\_Registration \notin Registration$

grd2: $m \in Module\_Runs$

grd3: $s \in Student$

grd4: $runningModule(m) \mapsto enrolledIn(s) \in offeredIn$

then

...

end

3.4 Summary of approach

By using our approach to model an information system at different refinement levels, we introduced different concepts and distinctions in which each concept could be modelled in a new refinement. This approach can help the modellers to gradually model a complex system using layered refinement where in each refinement they focus on modelling and verifying a subset of the requirements. The approach can be summarised in the following steps:

- Modelling primary classes, associations and relevant events. In this abstract model, only the primary classes identified for the case studies are modelled. Any association between these classes are given with their constraints such as total and injective. The associations are functional. Also, we introduce the events for constructing instances from the classes, removing instances from the classes, updating the classes such as changing associating instances. These events must be introduced in this model and later are refined with more parameters, guards and actions.

- Adding attributes to classes. This model extends on the abstract model by introducing the details of the classes. Each class will have its attributes added and events extended to reflect on the new attributes. The attributes have their constraints such as partial, total and or injective defined as typing invariant. Constructor, destructor and update events are extended so that when constructing new instance of a class, its attributes are provided as parameters in the event and associated to the class instance to be created.

- Introducing secondary classes, extending events and adding new events. This model adds secondary classes to the abstract model and introduce the associations between these classes and the refined primary classes. The events for modifying the secondary classes are also defined. As part of the refinement, the events in the primary classes are extended so that they reflect on the added associations from or to the secondary classes.

- Introducing attribute classes, extending events and adding new events. The attribute classes include attributes in them. All events that modifies their instances are defined in this model. It is also necessary to extend the events from previous models to reflect on the added associations from or to the attribute classes.

- Introducing historical data, extending events and adding new events. This model defines new classes that are of the same type of previously defined classes. The usage of these classes are to hold any historical data by moving some instances from the active classes to the historical classes for archiving. This is to separate the active instances of a class from inactive or historical instances.

- Introducing query events. This is the final refinement in this approach where the queries are defined for the entire model. After the whole structure of the model is specified in different refinement level, the queries are introduced. By introducing queries later in the refinement, we make sure that all the necessary data are modelled in the system and queries between classes can be defined. As this refinement includes only queries and no modification on the state of the variables are introduced, the model will not require any proof obligations.
- Modelling database views through derived classes. In this initial treatment of views, we defines how to model view that derives attributes from a single class or between classes that relate together. The view inclusion as a refinement is viewed the same as queries: introducing this refinement in later steps so that all the classes, attributes and associations are modelled for views to be useful.

3.5 Defined patterns

There are three patterns defined in this thesis. Each pattern is applied to at least one case study, but not necessarily all of them. Here are these patterns:

Pattern: Association splitting **Problem:**While associations between classes in UML-B can be relational, this is not supported in relational databases. We need to address the mismatch between object classes and relational table. **Solution:**Refine each relational association in UML-B into two functions and a new class. The two functions will have the new class as their domain.

The Association Splitting pattern is applied in the SRES case study for offeredIn between *Module* and *Program* of study.

Pattern: Historical **Problem:** Over time, many records becomes irrelevant to the active state of the system while they take part of their classes. **Solution:** Move these records to historical classes that they share the same structure as the live ones.

This pattern has been applied for the SRES case study for *Completed Students* and *Completed Registration*. The third pattern *Inheritance* is discussed in Section 4.5.

3.6 Alternative approach

An alternative approach to our modelling is to model the whole system in one go. This will make the system harder to understand compared to the separation of concepts we introduced. We can argue that it takes longer to understand a large system than a system that is broken down to smaller and clearer concepts. It is also much easier to rollback or change a small refinement than to do the changes in one complex model. Moreover, creating the model can be a hard task. For this, it is easier to break the model into smaller models and model them as refinements. As our approach is supported by a tool that can generate the extension of the model from the refinement as an alter command in SQL as discussed in Chapter 7, it will not be possible to provide such facility if the system is to be modelled in one go.

Where the stepwise refinement is introduced in the previous work such as of [68] using classical B, or [37] using Z, the refinement was toward implementation where the concrete model becomes closer to the programming language. Refinement such as ours where new components is being added in the refinement is also known in the literature such as in [23] and [3], our approach of refinement is by applying it to database design using distinct concepts such as *Primary* and *Secondary* classes, where in each refinement we introduce a concept. We provided clear guidelines on what to introduce in each refinement when modelling information system.

3.7 Conclusion

The previous sections show that modelling databases in UML-B is a straightforward task when examining different components of databases. Throughout the process of modelling the case study, there were some distinctions that can be made between different kinds of classes and events. These distinctions, including classes, inheritance, constructor events, destructor events, or normal events can be used to decide how to model the system. They might identify a refinement strategy or patterns for the model such as starting with classes and associations, then introduce attributes, then queries or get events, ... etc. We followed an approach of gradually modelling the database structure and its CRUD operations in different refinement levels as will be discussed more in Chapter 6. Undertaking the approach of specifying various components in different refinements enabled us to model each concept separately and verify its specifications.

Chapter 4

Translating UML-B Model to Database Code

4.1 Introduction

In this chapter, we discuss our translation rules for translating from UML-B class diagram to database code. First, we justify the choice of using PL/SQL as programming technology from the three technologies outlined in Section 2.4.6 to translate the model operations. The variables in Event-B are translated to SQL. Then we define the translation rules which generates the database structure and operations. Finally, we define how to preserve event atomicity using transaction management in our translation.

4.2 Programming technology

Before defining any translation rules from UML-B models to database applications we need to define the application language we will translate to. The translation to a particular language should not make the translation rules highly coupled with that language. Extending the translation to other languages should be achieved with minimal changes. However, this discussion only concerns the translation of events or operations as the classes, attributes and associations are translated to SQL which is universal in relational databases model.

Stored procedures might not be the only technology to translate to, but it can be called from JDBC or Hibernate. Using stored procedure increases the performance of an application as the code resides next to the data. They are compiled once to executable code and cached for all users [75]. Moreover, stored procedures are a defence against SQL injection in which all embedded SQL statements are parametrized. Adding to that, they can be used to enforce access control over the underlying tables. The stored procedures

might be generated from the model events in UML-B with event parameters being the stored procedure parameters. Application developers can write the application in any language that supports calling stored procedures and they are not bound to Java as in JDBC and Hibernate. For these reasons, stored procedures are used when translating events in UML-B and Event-B to database code.

4.3 An EBNF Syntax for UML-B

Before we show the translation rules, we need to highlight the EBNF syntax for UML-B so that we can easily relate each element to its translation. The extended Backus-Naur form (EBNF) is a notation used to formally describe a syntax of a language [79]. A UML-B class diagram corresponds to a diagram with one or more classes and events. The recursive syntax is as follows:

```
<class diagram>     ::= <diagram name> [refines <diagram name>] <class>*
<diagram name>      ::= String
<class>             ::= <class name> <class type> [inherits <class name>] [
    refines <class name>]
                        [<attribute>* <association>* <event>*]
<class name>        ::= String
<class type>        ::= set | constant | variable
<attribute>         ::= <attribute name><attribute type><constraint>[<initial
    value>]
<attribute name>    ::= String
<attribute type>    ::= <primitive type> | <user type>
<primitive type>    ::= integer | boolean | real
<constraint>        ::= <total> , <injective>
<total>             ::= true | false
<injective>         ::= true | false
<association>       ::= <association name><is a function><source class><target
    class><constraint>
<association name>  ::= String
<is a function>     ::= true | false
<source class>      ::= <class name>
<target class>      ::= <class name>
```

A class diagram contains classes and it may refine another class diagram. We assume at least one class is modelled in a class diagram. The star(*) indicates one or many classes. The square brackets indicate optional elements. The event syntax is outlined in Section 4.6. This is equivalent to how UML-B syntax is defined by its metamodel.

4.4 Translating Structure

The first set of translation rules concern the data structure of the model without the operations.

Each rule is represented with a name and a signature that defines the source and the target of the rule. A translation function *translate* is defined which takes the source of the translation rule and yields the target SQL and PL/SQL code.

The Function *translate_classdiagram(diagram_name)* translates a UML-B class diagram to a database. It generates the code to create the database with the diagram name and iterate through the translation rules of the diagram contents of classes and events as in Rule_1 in Table 4.1. In UML-B, events are part of a class, but we separate them when presenting translations as an event, hence a procedure, might relate to more than one class/table. Also, to distinguish structure from operations in the translation. The *translate_class* and *translate_event* functions are broken down to different rules as follows in this chapter.

Rule_1	**UML-B class diagram ⇒ SQL database**
Source	translate_classdiagram(dn, $c_1, ..., c_n, e_1, ..., e_m$)
Target	**create database dn ;** translate_class(c_1) ; ... translate_class(c_n) ; translate_event(e_1) ; ... translate_event(e_m) ;

Table 4.1: Mapping class diagram to SQL database

Each rule is represented in a tabular format that outlines the rule signature, the source and the target.

In *Rule_1* we iterate through diagram classes and events. Classes in UML-B could be of types *set*, *constant* or *variable*. In our modelling, *set* classes are only used to define the types of *variable* classes. For each variable class in a UML-B class diagram model, the class is translated to a table in SQL. *Rule_2* specifies that a variable class in UML-B is translated into a table in a relational model of the database. Each class is generated with default primary key with type integer to represent its instances as in *Rule_2*. Each class consists of many attributes and associations. We show two alternative rules, one for a class with no inheritance, and another in the case of inheritance. For each generated class, we generate a default primary key, *cn_id*, which is fresh identifier generated from *cn* to represent its instances. A sub class in UML-B is translated to an association from the subclass to the super class. More details for treating inheritance is discussed in Section 4.5.

Rule_2	**UML-B class ⇒ SQL table** /* no inheritance */
Source	translate_class (cn,attributes, associations)
Target	**create table(**cn, cn_id **int primary key,** translate_attributes, translate_associations)

Table 4.2: Mapping UML-B classes to SQL tables

Rule_3	**UML-B class ⇒ SQL table** /* with inheritance */
Source	translate_class (cn inherits an, attributes, associations)
Target	Alter table cn add i int references an ; Where i is a fresh identifier

Table 4.3: Mapping UML-B classes inheritance to SQL association

Following the translation of the class definition, we translate its attributes and associations. We iterate through the class attributes where we apply the *translate_attribute* function to each attribute. Any attribute in a class is translated to an attribute in the corresponding table. The attribute type is mapped to its equivalent type in SQL as in Table 4.4. The target SQL uses an `alter` command that changes the structure of the table by adding a new attribute to the table generated from the class.

If an attribute is defined in the UML-B model as a total function, it is translated to a *not null* constraint in SQL which means that the value of the attribute must be provided when inserting a new record in that table. If an attribute is defined in UML-B as an injective function, it is translated to a *unique* constraint when mapped to SQL which means that all the values of that attribute are distinct. An initial value of an attribute is translated to a *default* constraint in SQL. That means if the value of the attribute is not provided when inserting a new record, the initial value will be used. Table 4.4 shows the translation of the attribute. Table 4.5 shows the total to not null constraint translation, and Table 4.6 shows the injective to unique constraint translation rule.

Rule_4	Class attribute ⇒ Table attribute
Source	translate_attribute (cn, an, atype, pc, ic, [initial value])
Target	**alter table** cn **add** an atype translate_total, translate_injective, translate_initial;

Table 4.4: Mapping class attributes to table attributes

Rule_5	**total function ⇒ not null constraint**
Source	translate_total(true)
Target	**not null**;

Table 4.5: Mapping injective function to unique constraints

We assume that every attribute is defined as a function. *Rule_7*: *attribute type ⇒ attribute data type* is applied to every attribute where the attributes type is translated to the equivalent type in SQL.

Rule_6	**injective function ⇒ unique constraint**
Source	translate_injective(true)
Target	**unique**;

Table 4.6: Mapping injective function to unique constraints

Rule_7	**Functional association between classes ⇒ Referential integrity**
Source	translate_association(asn, source, target, total_constraint, injective_constraint)
Target	alter table source add asn **int references** target (target_id) total_constraint, injective_constraint;

Table 4.7: Mapping association to referential integrity

For the association between two classes that is defined as a function, it is translated to a *foreign key* constraint and is added as an attribute in the originating table. The SQL command in *Rule_7* adds an attribute *asn* to the source table *source*. The command specifies that *asn* `references` the attribute *target_id* which is the primary key of the target table.

The foreign key *asn* references the primary key *target_id*, follows from *Rule_2*, of the referenced table *target*. The constraints are translated the same way as in attributes. The associations can be defined as a *partial function*, a *total function*, a *partial injective* or an *total injective*. In relational databases we have *one-to-one*, *one-to-many* and *many-to-many* relationships, each could be mandatory or optional. The associations are mapped to relationships as follows:

- Partial function ⇒ optional one-to-many; following Rule_7 where Rule_5 and Rule_6 don't apply.
- Total function ⇒ mandatory one-to-many; following Rule_7 and Rule_5.
- Partial injective ⇒ optional one-to-one; following Rule_7 and Rule_6.
- Total injective ⇒ mandatory one-to-one; following Rule_7, Rule_5 and Rule_6.

Many-to-many associations are refined to two functions depending on the type of relation as in section 3.3.10. However, we defined a translation rule for many-to-many associations as in Table 4.8 where the association in the UML-B is not defined as a function. The target SQL *create*s a new table *asn* with two attributes *source_id* and *target_id* in which each `references` tables *source* and *target* respectively. The SQL command also specifies that the primary key of this newly created table is a composite of both attributes *source_id* and *target_id*. The translation can be further detailed depending on the relation type as follows:

Rule_8	**relational association ⇒ new table with two foreign keys**
Source	translate_relational(asn, source, target)
Target	`create table asn (source_id int references source, target_id int references target, primary key (source_id, target_id));`

Table 4.8: Mapping relational association to new table

- Any relation (many-to-many) association between two classes $asn \in source \leftrightarrow target$ should be mapped to a new table, *asn*, with the name of the association. The new table should have two foreign keys to *source* and *target*.
- If relation *asn* is total, the foreign key to the domain of the relation, *source*, should have not null constraint.
- If relation *asn* is surjective, the foreign key to the range of the relation, *target*, should have not null constraint.
- If relation *asn* is total surjective, both foreign keys should have not null constraints.

4.5 Translating inheritance

The relational database model does not have the notion of inheritance as in the object-oriented model. While we model the system in UML-B using class diagrams, we translate the model into tables in the relational database. In this section, we discuss the choices of translating a super class with two sub classes in which each of the sub class has some unique attributes. The translation choices are evaluated on their performance while preserving the database consistency.

For a class A with two subclasses, B and C, the literature as in [49] discuss three patterns when mapped to relational databases. The first pattern is *Single Table Inheritance* in which the super class is mapped to a table and all subclass attributes are included in this table to form a single table for all three classes. The second pattern is *Class Table Inheritance* where each class is mapped to a table to form three tables, A, B and C. The third pattern, *Concrete Table Inheritance*, maps only the concrete classes to form two tables, B and C where both have the class A attribute. We evaluated these patterns to test the performance for inserting new records and retrieving data using each pattern. The testing was done on the example of *Account* from the car sharing case study in Chapter 6 as seen in Figure 4.1.

1. **Single Table Inheritance**: We add sub type attributes in super type table to form one table for all. In this case, a check constraint is needed to ensure only data for one sub type is provided such as:

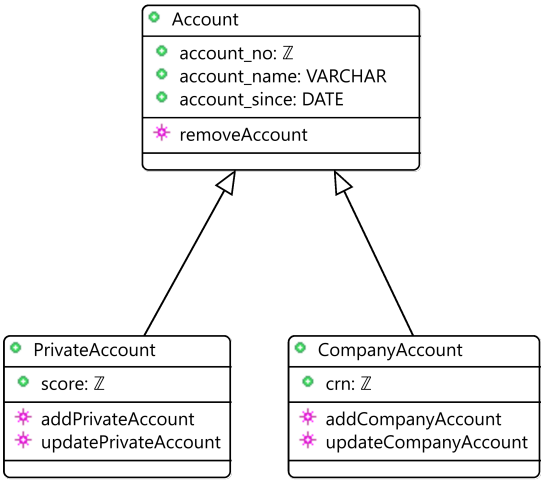

Figure 4.1: Account super class with two sub classes

```
alter table Account check (score is not null
 and crn is null) OR (score is null
 and crn is not null))
```

So that an *Account* cannot be both; company and private account.

2. **Class Table Inheritance 1**: In this pattern, we translate the classes to three tables where the sub tables reference the super. This way, a record of the super table might be referenced by both sub tables unless we have a constraint to enforce so that the super class is only referenced only by one of the sub tables but not both. This reference might be hard to set and evaluate as DBMSs don not support such check constraints. Also, this pattern enables a super table not to be referenced by any of its sub tables.

   We can solve the issue of referencing the same super instance by a trigger in the database. The trigger is a program that is fired or executed when an event occurs in the database. The trigger check each time a sub table try to reference a super table and ensure it is not already referenced by the other sub class. For the *Account* example, this can be done by the following trigger:

```
create or replace trigger check_supertype
before insert or update of Account_id on Private
for each row
declare
 ids integer;
begin
 select count(*)
 into ids
 from Company
 where :new.account_id = Company.account_id;
if (ids >0)
```

```
  then
        raise_application_error(-20000,
        'Account cannot be both a Private and a
        Company!');
  end if;
end;
```

The trigger is enforced before every insert or update the sub table *PrivateAccount*. If the *PrivateAccount* tries to reference an account that is already referenced by the *CompanyAccount* table, the trigger will raise an error and prevent the insert/update. As there might be cases where the super class can be referenced by both sub-classes such as *Person* who is a *Staff* and a *Student* at the same university, the constraint won't be generated unless explicitly specified.

3. **Class Table Inheritance 2**: In this pattern, we have a reference from super to both sub tables. A check constraint such as in 1 will enforce that a super table only references one of its sub tables. The super table, *Account* will have the following constraint:

```
alter table Account
  add constraint check ((pa_id is null
  and ca_id is not null)OR (pa_id is not null and
  ca_id is  null));
```

Where *pa_id* and *ca_id* are foreign keys from the super table *Account* to both sub tables *PrivateAccount* and *CompanyAccount* respectively.

4. **Concrete Table Inheritance**: This pattern forms two tables that are not related together (in a relational model sense) so we treat them like any other table. However, this pattern can violate the constraints and requirements of many systems especially when the shared attribute from the super class is required to be unique. Let us assume we have a *Person* class with *email* and *phone* attribute which are both unique. Two subclasses, *Staff* and *Student* inherit the super class with an additional attribute for each. If we followed this pattern and mapped *email* and *phone* to attributes in both tables, *Staff* and *Student*, we could have an instance of a *Student* and an instance of a *Staff* that have the same *email* or *phone* or both. Another reason for omitting this pattern is when a *Person* has a reference from another table. Mapping that reference to either *Student* or *Staff* is hard to accomplish.

Figure 4.2 shows the result of inserting and querying one thousand records using patterns 1 to 3 which shows that single table inheritance pattern performs better than the other patterns for insertion while no major difference is in query evaluation. As the first pattern will involve null values for 50% of each sub type records, this means possible space waste. To decide what pattern to support, we need to answer the question of how

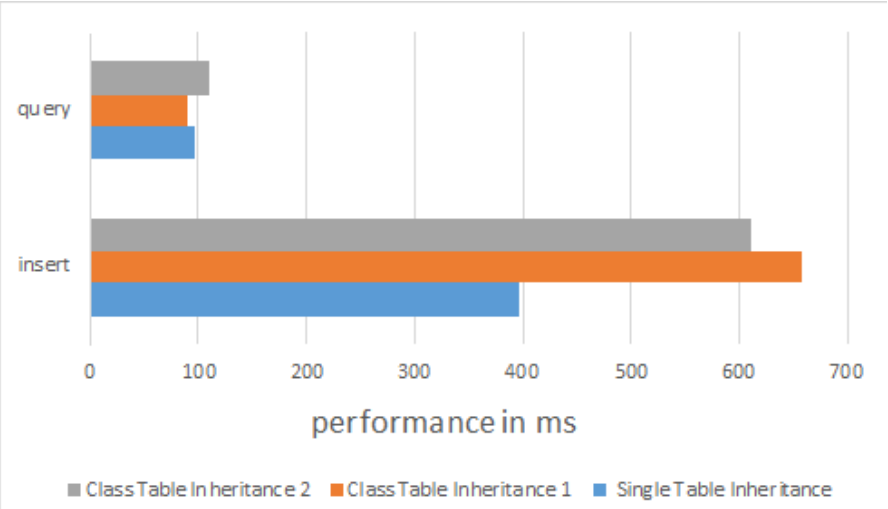

Figure 4.2: Performance evaluation of the two tested patterns

Pattern	**Used space in MB**	**Used blocks**
Single Table Inheritance	6 MB	768 blocks
Class Table Inheritance 1	7.875 MB	1008 blocks
Class Table Inheritance 2	9.875 MB	1264 blocks

Table 4.9: Used space and blocks of different patterns

DBMSs deal with null fields in terms of space usage. Will they optimise the space so that no space is set aside for fields that are unused in some rows? We populated the tables with 100000 records divided equally for sub tables and ran a query against DBMS to get the size of each table and the number of blocks used. Table 4.9 summarises the result which shows that single table inheritance takes less space and uses fewer blocks compared to the other patterns. However, as the associations can be added in later refinements, we can omit the first pattern for single table. We always translate to the pattern two for the references from sub classes to the super class.

The inheritance pattern is described in the literature as in [49]. This thesis provides a solution based on experimental evaluation on the performance of the mapping from inheritance to tables with associations and constraint preservation.

Pattern: Inheritance **Problem:** Relational databases do not support inheritance, thus we needs a way to map inheritance from UML-B to SQL. **Solution:** Generate tables for the super class and all sub classes. then associate each table of the sub class to the table generated for the super class.

This pattern has been applied for SRES case study for *Person* as super, and *Staff* and *Student* as sub classes. The generated code based on the translation rules generated the *Person*, *Staff* and *Student* tables with references from *Staff* and *Student* to *Person*. The same pattern has been applied to the Emergency case study.

Rule Name	Rule Signature: UML-B and Event-B ⇒ Database code
Rule_10	an event ⇒ stored procedure
Rule_11	event parameters ⇒ stored procedure parameters
Rule_12	event parameter type ⇒ stored procedure parameter type
Rule_13	guard relates to variable ⇒ ignored
Rule_14	equal operator ⇒ select statement
Rule_15	range restriction ⇒ IN condition with sub query
Rule_16	set operators ⇒ SQL set operator
Rule_17	cardinality function ⇒ count function
Rule_18	constructor event ⇒ stored procedure with an insert statement
Rule_19	destructor event ⇒ stored procedure with delete statement in SQL
Rule_20	override action ⇒ update statement
Rule_21	an event ⇒ one transaction unit

Table 4.10: Operation-related translation rule

4.6 Translating Operations

Class events in UML-B class diagram are translated into stored procedures in the database as per *Rule_10*. The stored procedure may have parameters and different logic and execution commands. Table 4.10 outlines the translation rules for Event-B events to different stored procedures where SQL commands are embedded in them. In this section, we outline the translation rules for class events including guards and actions. The EBNF for the UML-B events is as follows:

```
<event>             ::= <event name> <event type> <extended> <refines> <
    parameter>* <guard>* <action>*
<event name>        ::= String
<event type>        ::= constructor | destructor | normal
<extended>          ::= true | false
<refines>           ::= skip | <event name>
<parameter>         ::= <parameter name> <parameter type>
<parameter name>    ::= String
<parameter type>    ::= <primitive type>|<class type>
<class type>        ::= <class name>
<guard>             ::= <predicate>
<action>            ::= <expression>
```

The event parameters and parameters types are translated into procedure parameters and their types as in *Rule_11* and *Rule_12*.

4.6.1 Translating guards

For database verification, it is not enough to verify the structure of the database but we also need to verify the correctness of the database operations. Before executing any operation on the database, we need to ensure that the operations will not cause any

Rule_10	**an event ⇒ stored procedure**
Source	translate_event(en, $p_1, ..., p_n, g_1, ..., g_m, a_1, ..., a_j$)
Target	**create procedure** en { translate_parameter(p_1) ; ... translate_parameter(p_n) ; translate_guard(g_1) ; ... translate_guard(g_m) ; translate_action(a_1) ; ... translate_action(a_j) ;}

Table 4.11: Mapping event to procedure

Checked by DBMS	**Corresponding guard**
Primary key constraint	$a \notin A$ (where A is the class name)
Foreign key constraint	$r \notin B$ (where B is the target of the association)
Unique constraint	$this\_x \notin ran(x)$ (where x is an injective function)

Table 4.12: Checks performed by the DBMS and the corresponding guards

inconsistency on the database. This validation is modelled by the guards that enables an event in which it satisfies all invariants. However, some validation will be very expensive to perform in our translation when we have a large data set. When we pass a parameter in a constructor event and try to check that its value is not already in the set, this requires scanning the whole set of data for uniqueness. Meanwhile, the uniqueness property is translated from the injective function in invariants as by the rule *Rule_6*, and if the injective function is translated correctly, DBMS should check it before making the insert. This introduces a clear distinction of what guards to translate. The distinction is based on examining the guards we have in our modelling of different case studies.

When a guard is related to a variable in the model as in *Rule_13*, the guard will be ignored in the translation. Guards that check for the value of the parameter passed with no relation to variables will be translated to a program code in stored procedure and will be evaluated before executing any SQL statement. Examples are guards that check parameter types, or their range of values. Table 4.12 shows the checks that are performed by the DBMS and the corresponding guard for each of them. These guard patterns will not be translated and their validation will be performed by the DBMS.

Table 4.21 shows what guard patterns are we translating to SQL or application code, and what patterns are ignored. Third and fourth guards in Table 4.21 are similar in the pattern but different in which third guard checks for the existence of an element in a class where guard four checks for a data type.

The event *addA* shows a simple example of this distinction.

m

$addA \mathrel{\widehat{=}}$

any

$a\ b\ v$

where

grd1 : $a \notin A$

grd2 : $b \in B$

grd3 : $a \in A\_SET$

grd4 : $v \in type$

then

...:

end

- The guard *grd1* ensures that the newly generated instance is not an element in A. As the instances are translated into keys in the generated database based on Rule_2, the DBMS will always check this for every entry and prevent the insert if it is violated. Translating this guard explicitly in the code will result in doing the same verification twice; by the DBMS and by the program code in stored procedure. This double checking will result in slower performance.
- Guard *grd2* ensures that b is an element of B in which is used for the association between the two classes, A and B. As the association is translated to referential integrity, its validation will always be checked by the DBMS, so *grd2* is not translated in the generated code.
- Guards *grd4* and *grd5* checks the type of the parameters without reference to a variable. These validations will be translated in the generated code by specifying the type of the parameters passed.
- Guard *guard6* is used to check the uniqueness of a range of a function which is defined by an injective function invariant. This check will be done by the DBMS as *email* function is translated to uniqueness constraint in the generated code based on Rule_3_2 and the DBMS will always check for its uniqueness.

In UML-B, we model the data retrieval in a guard with equality sign that assigns a result to a local variable in a normal event as in *Rule_14*. In this case, the guard is translated to *select* statement in SQL and the event parameters that are used in this guard will be translated in the *where* clause in the *select* statement. The expression for select can be of different forms following the case studies modelled in this research.

Rule_14	**equal operator ⇒ select statement**
Source	a_list = r˜[{b}] Where r is an association from or an attribute in class A, b is a parameter in the event to represent an element of the association, and a_list is a parameter for a set of type A.
Target	`select a_id into a_list from A where r = b;` Where a_id is a key for A table translated by Rule_2.

Table 4.13: Mapping equal operator to select statement

```
<query result>    = expression
```

We used a subset of the expressions in the Event-B language based on the observations on our case studies. These expressions are as follows:

```
expression      ::= expression <operator> expression
                  | dom(expression)
                  | card(expression)
                  | expression~[{expression}]
```

We can show different cases for the query retrieval as in *getAs* example event which gets all As of a given Bs based on the relationship r.

$getAs \mathrel{\widehat{=}}$

any

$a\_list\ r\ b$

where

grd1 : $a\_list \in \mathbb{P}(A)$

grd2 : $b \in B$

grd3 : $a\_list = r^{\sim}[\{b\}]$

then

skip

end

We translate *grd3* to a *select* statement in which *p_list* is the result set of all As that mapped to b. This is translated to a select statement as in Table 4.13. If b is a set instead of an element, `r = b` is replaced with `r in (b)`. The relational inverse and image is currently the only pattern supported by this rule.

The query result can be used in any event as a guard where the result of the query is a condition for the update.

Rule_15	**range restriction ⇒ IN condition with sub query**
Source	$a\_list = dom(a \rhd b)$ Where a is an association from or an attribute in class A, b is an expression such as r˜[{b}], and a_list is a parameter for a set of type A.
Target	`select a_id into a_list from A where a IN (select a_id from A where r = b);` Where a_id is a key for A table translated by Rule_2.

Table 4.14: Mapping range restriction to sub query

Rule_16	**set operators ⇒ SQL set operator**
Source	list = a ∪ b Where a and b are expressions, and list is a parameter for a set of type a and b where a and b are of the same type.
Target	`select a UNION select b` Where a and b are select statements mapped from expressions in earlier rules.

Table 4.15: Mapping set operators

Rule_17	**cardinality function ⇒ count function**
Source	a = card(b). Where a is of type integer and b is a variable or an expression.
Target	`select count(b) from A` if b is a variable and `select count(*) from A where b` if b is an expression. Where A is the class name.

Table 4.16: Mapping cardinality

Translating guards or ignoring them at the translation process is derived by the analysis of the three case studies. By modelling all case studies events with guards and translating them to SQL, we can argue that the rules defined for the guard translation cover the common cases. The guards that are ignored appears in most of the events and they are checked by the DBMS as in Table 4.12. All the guards for the case studies are addressed in the translation rules of guards. In case a system has guards that do not fall in the defined guards for all the case studies, new translation rules will need be introduced.

4.6.2 Translating actions

The actions represent the actual operations on database except for data selection. In our case studies, we tried to follow patterns when modelling these operations in actions, and use minimal mathematical notation. This section shows what notation we used for each event and operation and how we translate them to stored procedures with SQL statements. The examples show only the SQL statement without the whole declaration of stored procedures. Table 4.22 summarises the actions that are used in our models.

Constructor events in UML-B class are translated to stored procedures with SQL insert statements as in *Rule_18*. The example event *addA* is defined as a constructor event for the A class. It is translated to a stored procedure named *addA* as shown below.

$addA \mathrel{\widehat{=}}$

any

$a\ b\ v$

where

grd1 : $a \notin A$

grd2 : $b \in B$

grd3 : $a \in A\_SET$

grd4 : $v \in type$

then

act1 : $A := A \cup \{a\}$

act2 : $r := r \cup \{a \mapsto b\}$

act3 : $x := x \cup \{a \mapsto v\}$

end

As an *insert* statement adds more than one attribute to a table, actions *act1-act3* are grouped together in the translation process to form one *insert* statement for all actions as in *addA* procedure as in *Rule_18*.

Rule_18	**constructor event ⇒ stored procedure with insert statement**
Source	$A := A \cup \{a\}$ $r_i := r_i \cup \{a \mapsto b_i\}$ $x_j := x_j \cup \{a \mapsto v_j\}$ Where r_i is an association from class A to class B_i and $i \in 1..m$. The x_j is an attribute of type v_j and $j \in 1..n$.
Target	`insert into A` $(a\_id, r_i, .., r_m, x_j, .., x_n)$ `values` $(a, b_i, .., b_m, v_j, .., v_n)$`;` Where a_id is the key that represent tuple in A as by Rule_2

Table 4.17: Mapping constructor to insert statement

Destructor events in UML-B class are translated to stored procedures with SQL delete statements. Destructor events work similar to constructors as all attributes in the database for that class are affected when deleting an instance of it as in the example event *removeA*.

removeA $\widehat{=}$

any

a

where

grd1 : $a \in A$

then

act1 : $A := A \setminus \{a\}$

act2 : $r := \{a\} \lhd r$

act3 : $x := \{a\} \lhd x$

end

The delete statement will be translated from *act1* only as deleting a record from a table correspond to deleting its attributes and relations. The event will be translated to a *removeA* procedure where the class instance, *a*, is used as the value of the primary key of *A* table which is *a_id*.

Rule_19	**destructor event ⇒ stored procedure with delete statement**
Source	$A := A \setminus \{a\}$ $r_i := \{a\} \lhd r_i$ $x_j := \{a\} \lhd x_j$ Where r_i is an association from class A to class B_i and $i \in 1..m$. The x_j is an attribute in A and $j \in 1..n$.
Target	`delete from A where a_id = a;` Which will delete all associations and attributes with a key equal to a.

Table 4.18: Mapping destructor to delete statement

Both constructors and destructors are translated by default to insert and delete statements. While destructors will remove all attributes and associations of the class instance, constructors might provide some but not all attributes and associations of a class. This is possible when the omitted attributes and association are defined as partial functions. For this reason, the translation of a constructor will iterate through all actions of the event, while destructor translation does not need to do that.

Normal events are used for two kinds of operations in our modelling, updating class or querying information from the class. Normal events in a class with override operators are translated to a stored procedure SQL update statements as by *Rule_20*. This rule applies to all override actions in an event. The following example event override more than one attribute from the A class but will be translated to one update statement as in *Rule_20*.

Rule_20	**override action ⇒ stored procedure with an update statement**
Source	$r_i := r_i \lhd\!\!+ \{a \mapsto b_i\}$ $x_j := x_j \lhd\!\!+ \{a \mapsto v_j\}$ Where r_i is an association from class A to class B_i and $i \in 1..m$. The x_j is an attribute in A and $j \in 1..n$.
Target	`update A set` $r_i = b_i$, $x_j = v_j$ `where a_id = a;`

Table 4.19: Mapping override to update statement

updateA $\widehat{=}$

any

$a\ b\ v$

where

grd1 : $a \in A$

grd2 : $b \in B$

grd4 : $v \in \mathbb{Z}$

then

act1 : $r := r \lhd\!\!+ \{a \mapsto b\}$

act2 : $x := x \lhd\!\!+ \{a \mapsto v\}$

end

4.6.3 Transaction rules

While each event in Event-B is atomic and one event can have actions that translate to multiple SQL statements, DBMS such as Oracle supports *statement-level atomicity*. This means each statement is an atomic unit [76] and if one event is mapped to more than one statements, its atomicity is violated. To guarantee event atomicity when translated to stored procedures with multiple SQL statements, a transaction management must be defined as per *Rule_21*. A simple solution is to define a *savepoint* before executing any statement. A savepoint is a point in the transaction in which we can roll back to [77]. If all statements execute correctly, then the transaction is committed, otherwise, the transaction returns the state of the database to the savepoint. For example, let us consider an event that moves a student between two classes by removing its record from one class and adds it to another. A real case will be graduating students where all graduate students are moved to an *archive* or a *history* class. This can be modelled as the following event:

graduateStudent $\widehat{=}$

any

Rule_21	**an event ⇒ one transaction unit**
Source	event ev action* end

Target	`procedure ev`
	start transaction
	statement*
	end transaction
	`end`

Table 4.20: Mapping event to single transaction

s

where

grd1 : $s \in Active\_Student$

then

act1 : $Active\_Student := Active\_Student \setminus \{s\}$

act2 : $Graduated\_Student := Graduated\_Student \cup \{s\}$

end

Both *Active_Student* and *Graduated_Student* are of the same type. The event is atomic in which both *act1* and *act2* are executed correctly or the whole event is not executed. The procedure *graduate_student* ensures that both *insert* and *delete* statements are executed correctly or none of them as the exception will rollback to the state when *before_graduate* savepoint is defined.

```
create procedure graduate_student(sid in int)
    as
begin
    savepoint before_graduate;
    insert into graduated_student
      select * from active_student where student_id = sid;
    delete from active_student where student_id = sid;
    exception
        when others then
            rollback to before_graduate;
end;
```

4.7 Conclusion

In this chapter, we outlined and discussed our translation rules from UML-B class diagram model to relational database code. The UML-B model is mapped to SQL that generates the structure of the database that persists the data of the system. Different

events in the models are mapped to stored procedures that manipulate the data stored in the database. The choice of stored procedures as a translation code should add a layer of protection to the database against attacks such as SQL injection.

Translating class inheritance in UML-B to a relational database is not a very straightforward task compared to translating normal classes to tables. It becomes clear that different translation choices might have some consequences that might affect the consistency of the database. While the time taken to retrieve data is close for the three tests, the result shows that the *Single Table Inheritance* perform better when inserting new records than the *Class table Inheritance*. However, depending on different cases, the first pattern might not be applicable such as when we need to reference one concrete table to other tables in the database. The same can be said for the other patterns which means each case study or system may require different pattern. Currently, UB2DB supports the second pattern only.

Guard pattern	**Translated to**	**Event kind**
$r \notin ran(R)$	Ignored	Constructor
$a \notin A$	Ignored	All
$a \in A$	Ignored	All
$x \in \mathbb{Z}$	x in integer	Constructor and normal for update
$x \in 0..100$	$if(x \geq 0 \&\& x \leq 100)$	Constructor and normal for update
$a\_list = r^{-1}[\{b\}]$	select * from <type of a_list> where r = b	Normal for select
$a\_list = r^{-1}[b]$	select * from <table of a_list> where r in (b)	Normal for select

Table 4.21: Guard patterns to translate

Action pattern	**Translated to**	**Event kind**
$x := x \cup \{this\_a \mapsto this\_x\}$	insert into <class name> (x) values (this_x)	Constructor
$x := x \Leftarrow \{this\_a \mapsto this\_x\}$	set x = this_x where <id of this_a type> = this_a	Normal for update
$A := A \setminus \{this\_a\}$	delete from A where <id of A> = this_a	Destructor

Table 4.22: Translated action patterns

Chapter 5

Semantics of Generated Code

5.1 Introduction

While relational databases are crucial aspect of today's systems, their languages, such as SQL, do not have a clear formal semantic. Formal semantics provide precise description and meaning for a language that are implementation independent. In this research, semantics of a subset of a relational database language is defined using Event-B. The semantics is targeted to the subset of SQL and stored procedures that is generated by the tool UB2DB [12] as defining a full semantics for all of SQL syntax is beyond the scope of this research. First we outline some related work on formal semantics of SQL. Then we start defining the SQL semantics for the database structure mainly for `create` and `alter` commands with the semantics of key constraints. The semantics for database operations are given for the stored procedures syntax in PL/SQL language for Oracle databases. Any other syntactic variations for other vendors should be semantically equivalent. The semantics of database operations includes different aspects such as operations parameters, transactions and different statements on databases.

5.2 Formal semantics of SQL and PL/SQL code

The UB2DB tool generates SQL and PL/SQL code from UML-B model for both structure and operations respectively. To show that the generated code is an equivalent or a refinement of the original Event-B code, we define the semantics of SQL and PL/SQL in Event-B. The Event-B model that results from this process should be a refinement of the Event-B semantics of the source UML-B model. Only the subset of SQL and PL/SQL which is generated by UB2DB is treated here.

First we show the SQL semantics that is used to `create` or `alter` the structure database tables. This involves the semantics of tables, attributes, associations and constraints. Then we outline the semantics for operations that query or modify the database.

5.2.1 Semantics for table structure

The relational model of a database is composed of several *tables*. Table elements are represented as *tuples* which may be formed by one or many *attributes* where each attribute takes its value from a set. Given sets $S_1, S_2, \ldots, S_n$, table R is a set of n-tuples where each tuple has its first element from S_1, its second element from S_2, ...etc. Each attribute in a table has a heading (name) and a domain of values that is taken from a set such as character or integer.

There are at least two approaches in which we can model database tables in Event-B. One approach is to model tables as records in Event-B [42]. For example, if we have *Person* table with some attributes, we can model it in Event-B as follows:

$Person = name \times dob \times address$

This is a simple to encode individual tables and gives one variable for each table with all its attribute. Attributes such as *name* have their types specified in invariants such as $name \subseteq NAME$ where *NAME* is a carrier set. However, refinement in this approach will need to change the table if we need to add new attribute, e.g:

$Person2 = name \times dob \times address \times phone$

In this approach, we will need to refine the table in each event it occurs in, plus providing a gluing invariant for each refined table.

Another approach is by using function projection. This approach is used for this work. Each attribute is a function from the table to the attribute type. This allows extension refinements as adding more attribute means adding another function without affecting the abstract model (no data refinement). Also, compared to the previous approach, this approach makes it easier to encode the constraints of attributes in its function as not null is a total function, null is a partial function and unique is an injective function.

A table is defined in SQL by stating its name and list of attributes as follows:

```
Create table T (pk P,
                fk1 F references T1,..., fkn F references Tn,
                a1 A [c*],..., an A [c*]);
```

Where *pk* is a primary key of type *P*, *fk* is a reference of type *F* to a table T_i, and *a* is an attribute of type *A* with *c* constraint(s). The constraints are:

```
[not null]
[unique]
[default = v] where v is an initial default value.
```

The T is defined in the SQL as a table type. This can be presented in Event-B as follows:

$$T \subseteq TABLE$$

In Event-B, we define the table as a set variable and the *TABLE* is represented as a carrier set. As variables must be of a type in Event-B, we specify T as of type *TABLE*, which is a carrier set defined in Event-B context. However, in Event-B we abstract the type of a set of variable such as *Person* in a case study example to an abstract type of *PERSONS* for example. This implies that to add any element to *Person*, the element must be of a type *PERSON*. For this, each table is given its own super-type in Event-B.

We showed in Chapter 4 in *Rule_2* of our translation scheme that a class diagram is mapped to a table in the SQL syntax. For a variable class, T in UML-B, the UML-B tool generates an Event-B syntax such as:

$$T \subseteq T\_SET$$

Where T_SET is the superclass for T of type carrier set. The source of the translation $T \subseteq T\_SET$ is equivalent to the target of the semantics.

The table must have one, and only one, primary key that distinguishes each tuple. Any table may have n number of of foreign keys that associate the table to the primary set of itself or another table. These constraints are discussed further in Section 5.2.2

Let us assume that attribute a in an SQL statement is associated to the T table and its value is from the set A. In the database, every tuple in T may have a value for attribute a, and that value must be of type A. This is represented by:

$$a \in T \nrightarrow A$$

In a relational database, each attribute must be atomic which means it has only single value for each instance and there is no repeating groups for attribute values. The attribute, a, in the table is a function in Event-B and can't be a relation.

A relation between two tables in SQL is an attribute in the source table that references the target table primary key.

A foreign association, fk_i, between tables T and T_i can be specified in Event-B as a function from T to T_i. The following shows the association in Event-B:

$$fk_i \in T \nrightarrow T_i$$

The mapping of primary key is discussed in Section 5.2.2.1.

5.2.1.1 Structure modification

SQL allows the structure of a table to be altered using the `alter` command. This allows the modeller to add new attributes to an existing table or modifying existing attributes. The same semantics of attributes in `create` can apply for the SQL that extends the structure of a table by adding a further attribute or association as in the statement below.

```
Alter table T
            Add b B [c*];
```

This SQL statement adds a new attribute, b, to the table T of a type B with constraint c. This extension results in a new variable and invariant in the refinement of the model in which the table was originally defined.

$$b \in T \nrightarrow B$$

The database can also be extended by adding a new association between two tables such as:

```
            Add d D references T2 [c*];
```

Where d is the association name of the type B that references *T2*, given that neither b nor d are already existing in the table T and c is a constraint.

5.2.2 Constraints and keys

5.2.2.1 Primary key

A primary key in a relational database is an attribute or composition of attributes in a table that are defined as identifiers for that table. The property of a primary key is that by knowing its value, we can get the value of its tuple attributes. This means that attributes in the table are functionally dependant on the primary key. As tables are

sets in relational database, their elements are presented as tuples or rows. Each tuple is identified by a primary key for that table. The tuple identifiers can be represented in Event-B as the instances of its table set which is the domain of each attribute function in Event-B. This complements the semantics of attributes as functions from the table set. As the primary key identifies the value of attributes for each tuple, the function domain identifies the value of its range.

For the table T that is translated to set T in Event-B, the following is true:

$$\forall\, t_1, t_2, x_1, x_2 \;.\; t_1, t_2 \in T \land t_1 \mapsto x_1, t_2 \mapsto x_2 \in a \land x_1 \neq x_2 \Rightarrow t_1 \neq t_2$$

Where t_1 and t_2 are tuples in T and x_1 and x_2 are values of attribute a in T. As a is a function from t to x, if x_1 and x_2 are not equal, then their domain t_1 and t_2 must be different.

5.2.2.2 Integrity constraint

The integrity constraint is managed by the foreign key that references two tables in SQL. We showed earlier that the reference is specified in Event-B as a function between the source and target tables. For a foreign key that reference one table to another, we can show its semantics as:

$$ran(fk_i) \in T_i$$

Here fk_i is the key that references T_i in T. For every event that adds elements to this function, the range of the maplet of the new element must always be an existing element of T_i.

5.2.2.3 Unique constraint

The uniqueness constraint means that the value of this attribute cannot be repeated in this table. This is equivalent to the following Event-B invariant.

$$b \in T \rightarrowtail \mathbb{B}$$

The injective function represent the uniqueness in SQL. The unique attributes in a relation are candidate keys in that relation, including the primary key. For a table T, for any candidate key in this table, we can say:

$\forall t1,\ t2 \in T.\ t1.canKey = t2.canKey \Rightarrow t1 = t2$

Here $t1$ and $t2$ are tuples in the table. This constraint applies for all unique keys in which they could be a single attributes or multiple attributes. This semantics is represented in Event-B as an injective function as above. This is equivalent to the source of the translation rule *Rule_6*.

5.2.2.4 Not null constraint

An attribute in a relation with a `not null` constraint means the value of that attribute must be given. The following command specifies d as a not null attribute.

```
c B not null;
```

As we specified that attributes in tables are mapped to a function from the table to the attribute type, the following Event-B invariant specifies the type of c in T:

$c \in T \rightarrow \mathbb{B}$

Since the attribute must have a value given from `not null` constraint, the function must be total. This is equivalent to the source of the translation of total function as in *Rule_5*.

5.2.3 Semantics for stored procedure blocks

Database operations can be handled by stored procedures in PL/SQL. The stored procedures can be broken down into procedure declaration and procedure body. The declaration contains the optional parameter list, while the body performs the tasks expected from the procedure. A procedure that performs queries or/and data manipulation tasks can be generalised as follows (anything between square brackets is optional):

Procedure p ($[par_1\ type_1, ...,\ par_n\ type_n]$)

Begin

<statement*>

End

The parameter list in p specifies each parameter name and type which is the set in which its value is from. When a procedure is specified to have a parameter from a given type, the procedure will not run unless the passed parameter is from that type. This semantic serves as a pre-condition for the procedure to be enabled. To ensure that a procedure

is not executed with incorrect parameter, we define the parameter list as guards in the Event-B event which must be true for the event to be enabled. This can be translated to this equivalent event:

Event p

Any $par_1, ..., par_n$

Where $par_1 \in type_1, ..., par_n \in type_n$

Then <action*>

End

Each element of this semantics can be linked to the translation rules of Chapter 4. In translation definition we showed that an event is translated to a procedure as in *Rule_10*, its parameters to procedure parameters as in *Rule_11*, etc. The semantics here are inverse translation from procedure to events.

The statements section in the procedure may include any number of SQL statements such as *insert*, *delete*, *update* and *select* and they executed sequentially. The SQL statements, by default, are not atomic; as the failure of one statement does not cause a rollback or a termination of another statement. However, the list of statements can be enclosed within a transaction unit to make them atomic. This means that either all statements inside the transaction unit execute successfully, or non of them.

As the statement block in the stored procedure can have a sequence of statements, each statement could be mapped to an event. The atomicity in Event-B is different than the sequential behaviour of procedures. While events that modifies many tuples (such as transferring between different accounts) or multiple tables (such as moving records from one table to another) are atomic if they are modelled in one event in Event-B, they are not in stored procedures unless they are grouped in one transaction unit. As, by default, statements in procedure are not atomic, each statement in a multi-statements procedure is semantically equivalent to an event. The events will have the same parameter list and guards for those parameters. For the procedure p with m statements, it is mapped into an m events as follows:

Procedure p ($[par_1\ type_1, ..., par_n\ type_n]$)

Begin

$statement_1, ..., statement_m$

End

Event p_1

Any $par_1, ..., \ par_n$

Where $par_1 \in type_1, ..., \ par_n \in type_n$

Then $action_1$

end

$\vdots$

Event p_m

Any $par_1, ..., \ par_n$

Where $par_1 \in type_1, ..., \ p_n \in type_n$

Then $action_m$

end

Let us first explore the approach of non-atomic statements to illustrates why it is complicated, then conclude our approach of atomic operations. Since the procedure is sequential, the order of statements is important. To map this correctly to Event-B semantics, we need to introduce a control Boolean variable(s) that controls the order of events execution. Let us take the event *ev1* as an example which assigns values to two variables as follows:

Event Ev1

Begin

...

Then $x1 := E, y1 := F$

End

The event *Ev1* illustrates an example consists of two sequential statements. For a sequential procedure that changes these values, it is equivalent to a sequence of two events as there are two actions in *ev1*. To control the order of the events, we introduce a new flag variable, say *cont* which is a Boolean type. We set the variable as false and specify that as a guard in the event for the first action then flag it as true. The guard for the second event ensures that the flag is true before changing the value for the second action

and then resetting the flag to false.

var:

$x2, y2, cont$

inv:

$y2 = y1, cont \in Bool, cont = False$

$cont = False \Rightarrow x2 = x1$

Events:

Ev2 refines skip

When $cont = False$

Then $x2 := E, cont := True$

End

Ev1 refines Ev1

When $cont = True$

Then $y2 := F, cont := False$

End

The event *Ev2* makes *x2* equals to *x1* in the abstract event by assigning E to it and flagging *cont* to True. It is possible for *x2* not to be equal to E when the flag is False, hense adding the invariant that when the *cont* is false, this implies that *x2* is equal to *x1*. Otherwise, *x2* is equal to E, the same as *Ev1* do for *x1*.

This approach of refining a procedure into different events and providing flag(s) when having multiple statement becomes more complex to refine manually when the number of statements grows. For an n statements, we will need a $n - 1$ flags to control the event sequence. For a translation that starts from Event-B model to stored procedure, it is important to make the procedure atomic as the events that generate them. Our translation scheme of Chapter 4 only generates atomic statements. In our translation we set all statements translated from one event as atomic, i.e., as one transaction unit. The statements that are bound in one transaction unit in a stored procedure can then have their semantics as a single event instead of multiple events. The following procedure performs multiple statements in one transaction:

Procedure p ([par list])

Begin

start transaction;

statement$_1$; ...; statement$_n$;

commit;

End

As the transaction is an atomic unit, the procedure is semantically equivalent to one atomic event in Event-B:

Event p

...

action$_1$, ..., action$_n$

End

5.2.4 Semantics of CRUD operations

After discussing the main structure of procedures, we need to specify the semantics for each individual statement. The following examples shows procedures that *insert*, *update*, *delete* and *select* records from the database and their equivalent events. The UML-B source for a procedure is a combination of CRUD operations on distinct sets of tuples. This means the operations of a generated procedure are sequentially independent of each other. While multiple statements can be in one procedure, we show them here as separate procedures to distinguish the semantic of different types of statements. The composition of statements is explained in Section 5.2.4.5.

5.2.4.1 Insert procedure

Procedure p (par_1 $type_1$, ..., par_n $type_n$)

Begin

insert into T ($att_1, \ldots,$ att_n) values ($par_1, \ldots,$ par_n);

End

The insert procedure adds element(s) to the table set and its attribute values. Each element/tuple in the table has a key that identified that tuple and all its attribute values. The following event shows how this procedure is mapped in Event-B.

$p \mathrel{\widehat{=}}$

any

$t,\ par_1, ...,\ par_n$

where

`grd1`: $t \notin T$

`grd2`: $par_1 \in type_1, ...,\ par_n \in\ type_n$

then

`act`: $T := T \cup \{t\},\ att_1 := att_1 \cup \{t \mapsto par_1\}, \ldots,\ att_n :=\ att_n \cup \{t \mapsto par_n\}$

end

Where t represent an element of the table which is the newly added tuple that is translated as primary key as in 5.2.2.1. If the primary key is automatically incremented, it does not need to be in the attribute list in `insert` statement. For any attribute att_i in the table that is defined to be unique, the following guard must hold:

$par_i \notin ran(att_i)$

As inserting a tuple in a table, T, might requires inserting a foreign key to another table, T_i, the insert procedure needs to satisfy the constraint of a foreign key. The value for the foreign key is passed as a parameter, par_i, and the following guard must hold:

$par_i \in T_i$

Then the action for insert should include:

$fk := fk \cup \{t \mapsto par_i\}$

Where fk is an association from R_1 to R_2.

5.2.4.2 Delete procedure

The delete procedure removes all tuples from a table or a subset of its tuples based on a condition. The condition returns a subset of the tuples that satisfy it in which they are identifiable by the tuple key. The procedure should look like:

Procedure p ($[par_1\ type_1, ...,\ par_n\ type_n]$)

Begin

delete from T [where Cond];;

End

Such statement results in subtracting a tuple from the table set. This procedure is then equivalent to:

$p\ \widehat{=}$

any

$t,\ par_1, ...,\ par_n$

where

grd1: $t = \{x \mid Cond(x)\}$

grd2: $par_1 \in type_1, ...,\ par_n \in\ type_n$

then

act: $T := T \setminus t,\ att_1 := t \lhd att_1,\ ...,\ att_n := t \lhd att_n,$
$assoc_1 := t \lhd assoc_1,\ ...,\ assoc_m := t \lhd assoc_m$

end

The event action subtracts the tuple from the table set that its key is equivalent to the parameter passed. It removes all attributes and associations from this tuple. However, we need to consider when a table, A references another table, B, and how to deal with removing a tuple from B that is referenced by A. The modellers can specify in SQL some delete restrictions on associations from a table as follows:

1. **Cascade**: This means that if a parent table has tuple(s) deleted, any tuples from child tables that reference the deleted one will be deleted also.

2. **No action/restrict**: This will prevent the action. The delete will rejected by the database engine and the state of the database will remain the same as before the statement. This is the default option if the user did not specify one.

3. **Set null**: As the name suggest, this option sets the value for any child that references the deleted tuples in the parent to null value.

4. **Set default**: This will set the value for the child to the default value provided in the association definition. This is not supported by some DBMS (Oracle for example).

These options might become inconsistent with the structure specified for a table. Modellers could specify a foreign key attribute as a *not null* but defining the `on delete` option as `set null`. The following table structure is valid in some DBMSs:

```
create table a (id int primary key);

create table b(b_id int primary key, a_id int not null,
                constraint a_in_b foreign key (a_id)
                references a(id) on delete set null);
```

The `on delete set null` means that if an *id* instance in *a* that was referenced in *b* by *a_id* got deleted from *a*, then its reference in *a_id* will be set as null. However, this is semantically wrong as we should not be able to specify *a_id* in *b* as *on delete set null* while we have it as *not null* attribute.

As the delete statement itself does not imply any foreign key restriction, the semantics is to follow the default option which is to restrict the delete if the tuple is referenced by another table. For any association, *toAssoc*, that is targeting the table in which we want to delete from, the following guard must hold for *restrict* option before the event is enabled:

$$t \notin ran(toAssoc)$$

This is repeated for every association in which t is in its range. This means that the deletion is restricted which represent the semantics of the default option (restrict) above. This means in SQL that the deletion should start from the parent to the children, assuming there is no circular references.

5.2.4.3 Update procedure

The update procedure replaces the value of attributes in a table with new values. The record(s) in the table to be updated must exist in the table. The update can be conditional which means that the update is only performed if a condition or a set of conditions are met. The condition is a guard in the event in which it defines the set of tuples that get updates. The following shows the structure of an update procedure:

Procedure p ($[par_1\ type_1, ...,\ par_n\ type_n]$)

Begin

update T set $att_1 = exp_1, ..., att_n = exp_n$ [where Cond];

End

The update only affects the records where the condition is true. The above procedure is mapped semantically to Event-B event as:

$p\ \widehat{=}$

any

$t,\ par_1, ...,\ par_n$

where

`grd1`: $t = \{x \mid Cond(x)\}$

`grd2`: $par_1 \in type_1, ...,\ par_n \in\ type_n$

then

`act`: $att_1 := att_1 \Leftarrow (t \times \{exp_1\}), \ldots,\ att_n := att_n \Leftarrow (t \times \{exp_n\})$

end

The condition in SQL update can be formed by three parts:

attribute operator expression

The operator that can be used to compare the result of attribute value to an expression are: $= \neg$, $>$, $\geq$, $<$ and $\leq$. The update will update the tuples of the table for which the condition holds. As t in the event represent the tuples to be updated, the condition defines the set of t to be updated. A condition such as att_m `= expression` is mapped to $att_m(t) = expression$ in the set comprehension of t in *grd1*.

The semantics for an update is defined as function overriding for the effected attributes. The function override means that the range of the functions for att_1 to att_n are replaced by new values passed in the parameters par_1 to par_n.

5.2.4.4 Select procedure

The select operation is different from the other three operations in that it does not change the state of the database. It only extracts data from the database to a temporary variable. Then the application can show the result to the user by iterating through the temporary variable if the variable is a set or just showing the variable value if it is an element. In a stored procedure, a select operation can be modelled as follow:

Procedure p (par_1 *out* $type_1[, ..., \ par_n$ *out* $type_n]$)

Begin

select $att_1, \ ..., att_n$ *into* $par_1, ..., par_n$ [where <condition>];

End

Where par_1 to par_n are output variables in the procedure parameters used to store the result of the query in select statement. The *into* clause stores the attribute value into the parameter value. The condition part restricts the selections for records that satisfy it. This can be easily shown in Even-B as:

$p \ \widehat{=}$

any

$t, \ par_1, ..., \ par_n$

where

`grd1`: $t = \{x \mid P(x)\}$

`grd2`: $par_1, ..., \ par_n \in type_1, ..., \ type_n$

`grd3`: $par_1 = att_1(t), ..., \ par_n = att_n(t)$

then

`act`: *skip*

end

Here the guard *grd1* represents all tuples that satisfy the condition. In guard *grd3*, storing of values is represented by an equation specifying the value of each parameter par_i.The predicate in *grd1* for select can be used in other CRUD operations such as update to restrict the update in tuples that satisfy this predicate such as *grd4* in *updateCarRate* in Section 5.2.4.3.

5.2.4.5 Complex operations

As noted before, an operation on databases can be formed by many statements or operations. An example is when moving data from one table to another table. This includes deleting a record from the first table and inserting a record in the second table as one operation. The deletion could also involve a selection of records to be deleted. The semantics of individual statements in such operations is the same as described for *insert*, *delete*, *update* and *select* discussed earlier. Procedures with multiple statements but without a transaction unit that unify all the statements should be mapped to multiple events, each event for each statement as described in Section 5.2.3. However, a transaction with multiple statements is mapped to a single event in Event-B for atomicity.

For a transaction T with an S_n statements, its atomicity means that all statements, S_1, ..., S_n are executed correctly or non of them. Its semantic is:

$$T < S_1; ...; S_n >= atomic < S_1; ...; S_n >, skip$$

The atomic is the event with a list of actions, each represent a statement in the transaction. If the transaction is rolledback as a reason for a failure of a statement, it is equivalent to *skip*ing the atomic event. The atomic event can be described further as follows

$$atomic < a := A; b := B >= a := A \parallel b := B$$

Given that B is independent of a. This emphasises the difference between SQL transactions or procedure statements in general and the actions in Event-B event. The $\parallel$ indicates the two statements execute in parallel. While the actions are parallel in Event-B event, the statements are sequential in procedures. We assume that statements in single transaction are independent of each others and that there are no two statements that assign to the same variable. This, in our model and translation, can be satisfied as we start from Event-B to SQL and stored procedures, and it is not allowed in Event-B for two actions to assign to the same variable in one event.

5.3 Exception handling

Modelling exceptions for procedures in Event-B is not a straightforward task as Event-B does not have an explicit mechanism to specify what happens in an event if its guards are not satisfied (the event is not enabled). However, different approaches might be used to handle exceptions in events. In PL/SQL procedures, users can specify a pre-defined exceptions for different possible scenarios, define their own exceptions, or provide one exception handling for all unsuccessful operations. In Event-B events, we can specify an exception for every guard negation in an event as another event such as:

evt1 $\widehat{=}$

any

$par_1, ..., par_n$

where

grd1: $predicate_1$

grd2: ...

grd3: $predicate_n$

then

act: *action*

end

This event, *evt1*, represent the semantic of successful completion of operation. Exception events should be modelled by the user to represent the exceptions of *evt1*. A message set is defined in the context to hold the exception massage and is partitioned into different constants for pre-specified error types such as:

CONTEXT c
SETS

$MESSAGES$

CONSTANTS

wrong_type null_value higher_range general_error

AXIOMS

axm1: $partition(MESSAGES, wrong\_type, null\_value, higher\_range, general\_error)$

END

The machine is then specifies with a variable for the message and used within an event that model the exception.

MACHINE m
SEES c
VARIABLES

$message$
INVARIANTS

inv1: $message \subseteq MESSAGE$

$exEvt1 \ \widehat{=}$

any

par_1
where

grd1: $\neg predicate_1$

then

act: $message := general\_error$

end

The *grd1* can be repeated for all exceptions in the original event *evt1*. In this approach there will be at least one counter event for exception, e.g $exEv_i$. Each exception event deals only with one exception. For example, an event, $exEv_i$, sets the value of a result variable to have the message of the exception. The modeller is responsible for modelling these exceptions one by one. We can assume that an event for a correct behaviour with n guards, there will be a maximum of n exception events, each for a negation of one guard.

For procedures where only one general exception is defined such as in *pExc*, the guard *grd1* in *exEvt1* will have alternatives of negations of predicates $1..n$ that shows all possible exceptions such as:

$grd1 : \neg predicate_1 \vee \ ... \ \vee \ \neg predicate_n$

This shows a single handling for all exceptions such as:

Procedure *pExc* ()

Begin

. . .

where others then

message

End

Another approach is to generate the exceptions automatically for each event by the UB2DB tool. The exception message is retrieved from the DBMS and it should be meaningful enough for the end user. This is easy to provide and it does not require extra modelling in Event-B. We can assume that the events in Event-B are well preserved and they represent the correct behaviour of the system.

5.4 Soundness of semantic

The soundness of the SQL semantics provided here can be verified by providing a round trip translation from SQL to semantics in Event-B and then from Event-B to SQL translation. For every translation rule, a mapping from the translation to the semantics can be achieved to provide the soundness of the translation. By reversing the formal semantics of SQL and stored procedures defined here, we should get an equivalent to the translation rules defined in Chapter 4. The following procedure:

Procedure p ([par list])

Begin

start transaction;

statement_1; ...; statement_n;

commit;

End

Is semantically defined as:

Event p

[par list]

action$_1$, ..., action$_n$

End

And following the translation rules *10,11,12 and 21* in Chapter 4, the event p is mapped to the following procedure:

Procedure p ([par list])

Begin

start transaction;

statement$_1$; ...; statement$_n$;

commit;

End

The tool can be extended so that the generated SQL from the model can be translated by the tool to its Event-B semantics. The tool then, and by using proofing utilities in Rodin, can show that the semantics is a refinement of the source model and provide the proof obligations for it.

5.5 Conclusion

In this chapter we showed how SQL and PL/SQL procedures can be mapped semantically to Event-B. While the semantics do not cover all SQL and PL/SQL features, they show the equivalent Event-B notations for the code generated by UB2DB. The related work for the semantics has been covered in this thesis. While the literature provides basis for the semantics of SQL, it does not provide any formal semantics for parameterised operations and conditions. Moreover, the focus on the literature was on providing semantics for SQL queries rather than database manipulation. This chapter contributes a formal semantics in Event-B for all aspects of the database language that is targeted by our tool UB2DB. This includes handling key constraints, providing semantics for single operations and complex ones, and providing semantics for exception handling. Also, the notion and treatment of refinement and database extension is provided as semantics for `alter` command in SQL.

Chapter 6

Case Studies

6.1 Introduction

In this chapter, we present three case studies that have been modelled during the development of the method and the guidelines. These case studies vary in components and nature. The aim of these case studies is to validate the incremental approach and code generation using a range of example to demonstrate the generality of the approach. Undertaking the case study provided the ability to evaluate the literature and identify the missing pieces when formalising a database in Event-B and UML-B. This chapter shows our approach to model information systems in UML-B based on the analysis of the case studies. The full Event-B model of the case studies can be found in [1].

6.2 Student Enrollment and Registration System (SRES)

A case study of a Student Enrollment and Registration System (SRES) has been modelled in UML-B in different refinement levels. The case study addresses the issue of various departments in a university, each of which has different programs of study and modules. A student should register in various modules that are offered for the program in which s/he is enrolled. Each module is taught by one or more staff. The case study covers a wide range of possible class types, relations and functions to determine a range of rules to model a database system in Event-B. While the system is not safety critical by its domain, we can argue that it is business critical for the student and university in general. If students information or registration was missing or corrupted, this might affect their entire career and possibly mental health. A university can not run properly without a consistent system that manage students in different programs and faculties.

[1] http://users.ecs.soton.ac.uk/azab1g14/CaseStudies/

Failing to provide correct information to students in term of offered programs and modules might harm the reputation of the university and affect its state among different academic institutions. For this case study, some of the requirements are as follows:

1. The university has multiples faculties/departments.
2. Each department has a dean.
3. A staff is working in a department, and can be the dean of that department.
4. A student enrols in only one program.
5. A student is registered in various modules.4
6. Students can only register in modules offered by their program of study.

6.2.1 Incremental Development Through Refinement

The SRES case study was built through different refinements in which each refinement adds new variables and events to the abstract model. This section outlines how we modelled the case study and the generated code from it. As much of the modelling of SRES is outlined in Section 3.3, we will briefly mention the modelling steps in this section without many details. The figures in this sections are repeated to save the reader from going back to Section 3.3. To remind the reader, the SRES case study has been modelled as follows:

- M0: Modelling primary classes and their associations along with the necessary events. Figure 6.2 shows our abstract UML-B model of the SRES case study.
- M1: Adding attributes for classes and extending events. Figure 6.3 shows this refinement for SRES. All events are extended to include the new attributes.
- M2: Modelling secondary classes. Figure 6.4 shows this refinement.
- M3: Modelling attribute classes such as *Address* in Figure 6.5.
- M4: Modelling historical data as in Figure 6.6.
- M5: Splitting many-to-many associations by refining it to a class with two association functions that are directed to the source and the target of the refined relation as in Figure 6.7. The *Offering* class is a refinement of *offeredIn* association in Figure 6.2.
- M6: Modelling queries as we introduced the *get* events that query existing data from the classes.

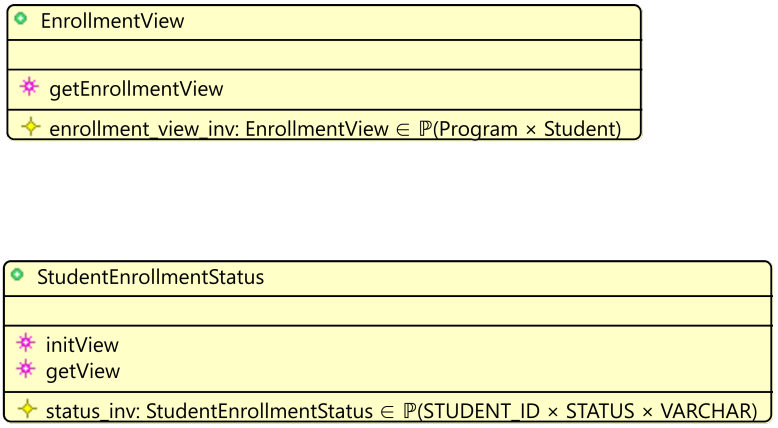

Figure 6.1: Modelling views in SRES

- M7: Modelling database views. This refinement includes defining examples of views in SRES case study. The examples in Figure 6.1 shows two different views that are modelled differently. The first view, *EnrollmentView*, lists programs of study and students enrolled in that program. It is defined as in the invariant as an element of the power set of two classes Cartesian product. The event *getEnrollmentView* list all view elements as maplet between Program and Student where for all Students that are enrolled in that Program.

$getEnrollmentView \;\widehat{=}$

where

$$\texttt{SRES\_guards8} : EnrollmentView = \{p \mapsto s | s \mapsto p \in enrolledIn\}$$

then

skip

end

The second view *StudentEnrollmentStatus* is defined differently in which its type is generic to the type of some specific attributes that are needed for the view. It defines the view as power set between three carrier sets that are the types of the attributes; *student_id*, *student_status* and *program_name*. Then two events are defined in which the first *initView* initializes the value of this view and the second event, *getView* list all the view values. The action in *initView* event assigns the value for the view which means that this event must be executed first in order for the view to be useful.

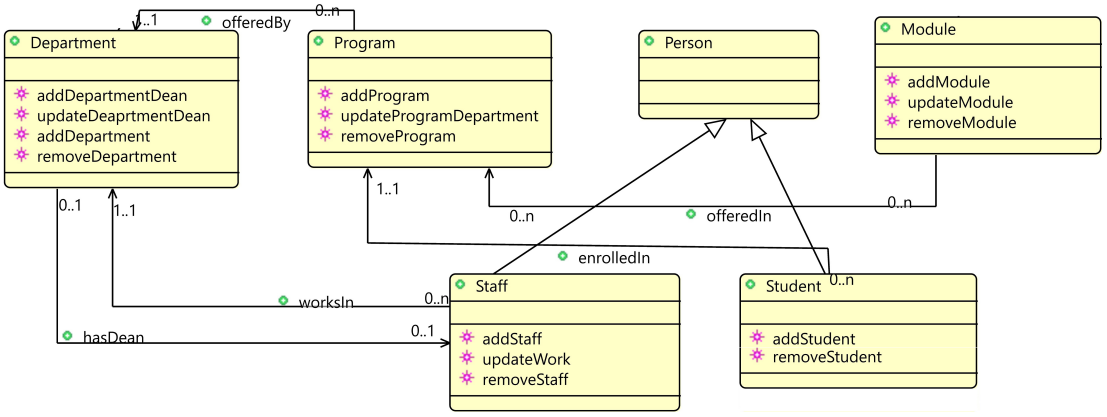

Figure 6.2: Abstract model of SRES entities and relations as a UML-B class diagram

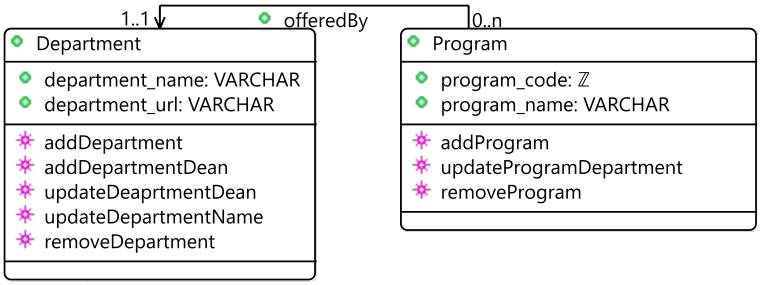

Figure 6.3: Adding attributes to the main classes

$initView \;\widehat{=}$

any

s

p

where

$\texttt{paramType\_s} : s \in Student$

$\texttt{paramType\_p} : p \in Program$

then

$$\texttt{viewAction} : StudentEnrollmentStatus := \{sid \mapsto status \mapsto title \mid student\_id(s) = sid \wedge student\_status(s) = status \wedge program\_name(p) = title \wedge s \mapsto p \in enrolledIn\}$$

end

6.2.2 Model validation

After modelling the case study, we validated the model against the requirements of the system. To validate the model, we used the ProB [65] which allows automated animation for Even-B specification. One animation for the abstract model of SRES is be executing the following sequence of events:

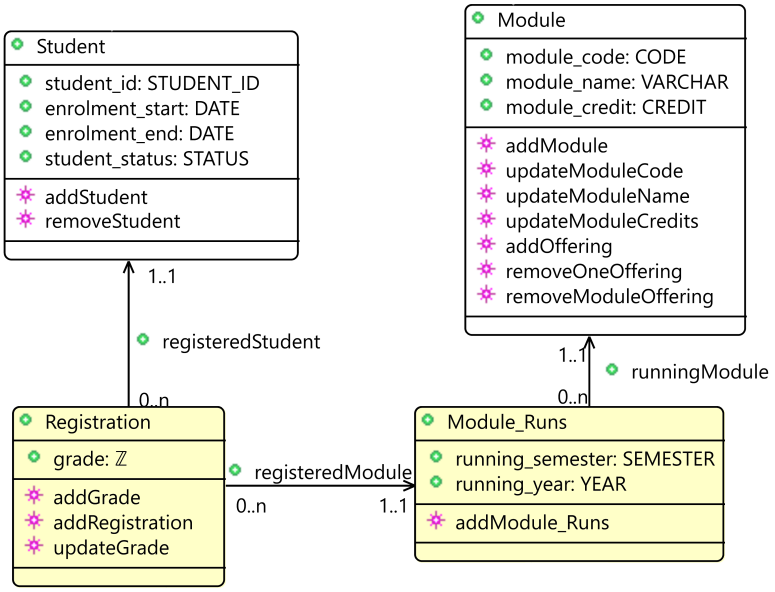

Figure 6.4: Adding secondary classes

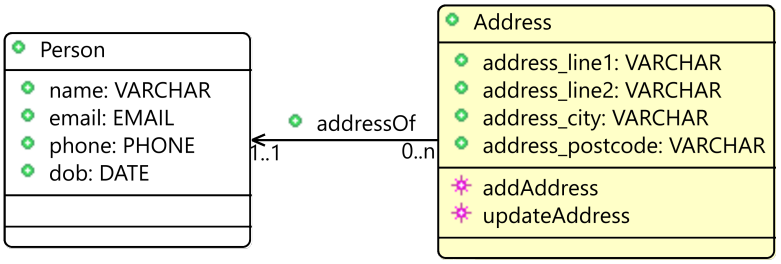

Figure 6.5: An example of an outer entity

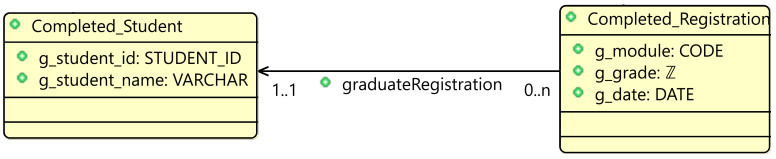

Figure 6.6: Historical data modelling

$$initialisation$$

$$addDepartment(d1)$$

$$addProgram(p1, d1)$$

$$addProgram(p2, d1)$$

$$addStaff(e1, d1)$$

$$addStaff(e2, d1)$$

$$updateDepartmentDean(d1, e1)$$

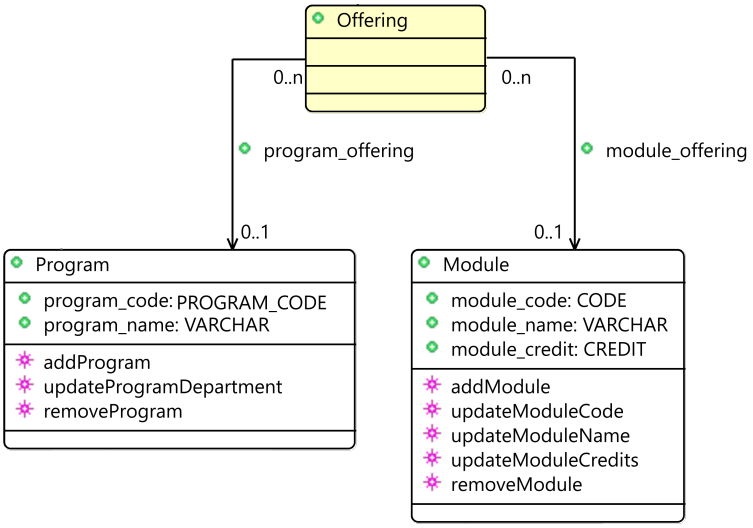

Figure 6.7: Association split

$$addStudent(s1, p1)$$

$$addStudent(s2, p2)$$

$$addModule(m1, p1)$$

$$addModule(m2, p1)$$

By using ProB animation, we changed the specification of requirements 1 and 2. The initial model was specified such that *hasDean* association from *Department* to *Staff* is totla, as well as *worksIn* from *Staff* to *Department*. When animating this model, neither *addDepartment* nor *addStaff* events are enabled. The reason is that as both are total, we could not create a department record without first having a staff record for the dean. Also, we could not add staff record as a department is needed for its *worksIn* value. We weakened *hasDean* association to be a partial so that we can add a department then add its staffs. The *updateDepartmentDean* then assigns the dean from staff to a department.

The case study model is fully verified and every event preserves all the invariants of the system. If we take the requirement 6 which states: Students can only register in modules offered by their program of study, the following invariant is specified in the model:

`inv1`: $\forall m, s \cdot s \mapsto m \in registeredStudent^{\sim}; registeredModule \Rightarrow$
$runningModule(m) \mapsto enrolledIn(s) \in offeredIn$

Along with *addRegistration* event outlined in Chapter 3, the *updateRegistration* also preserves this invariant as in *grd4* which ensures the new module to register instead of an old one is offered by this student program of study:

Machine	No of proofs	Automated	Manual
M0	37	37	0
M1	55	55	0
M2	33	28	5
M3	16	16	0
M4	1	1	0
M5	9	9	0
M6	0	0	0
M7	9	9	0

Table 6.1: POs in SRES Case study

$updateRegistration \mathrel{\widehat{=}}$

any

$this\_Registration$
m
s

where

grd1: $this\_Registration \in Registration$

grd2: $m \in Module\_Runs$

grd3: $s \in Student$

grd4: $runningModule(m) \mapsto enrolledIn(s) \in offeredIn$

then

...

end

6.2.3 Proof obligations

Table 6.1 shows the statistical analysis for the proofs in the SRES case study. This shows that most of the proofs are automated. Some refinements such as M6 includes no proofs as it only introduces queries and there is no event which change the state of the variables.

6.2.4 Generated code from SRES

After modelling the SRES in different refinements in UML-B, we generated both the SQL statements to create the structure for the database and the stored procedures that manipulate or query the data. For each class, a sequence in the Oracle database

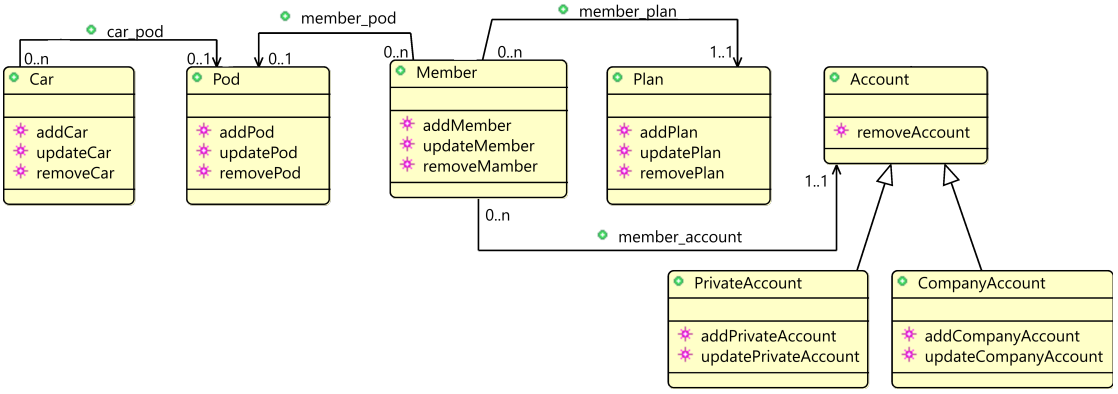

Figure 6.8: Abstract model of Car Sharing case study

management system is created to manage the the primary key generations for each class instance. A sequence in Oracle is a database object in which users can generate unique and incremental integers. Appendix A shows the SQL and stored procedures generated for SRES.

6.3 Car Sharing System

The second case study concerns car sharing system where different *members* get to share some *cars* that are placed in different locations called *pods*. The system manages the *booking* facility and records the *trips* taken in each booking. Members may be allocated their favourite pod. Members are charged based on distance and time travelled in a shared car. Many aspects of the system are time and location oriented such as pods, booking and trips. The system might be thought of as business critical in which invalid data might result in preventing bookings or charging incorrect fees. This section covers the layered refinement approach for this case study.

6.3.1 M0: Modelling primary classes and associations

In the abstract model of the Car Sharing case study, we modelled the primary classes and the associations between them. The model includes the *Car* class which is parked in a particular *Pod*. Members choose plans in which they are members for a month, a quarter or a year. Each member has an account which could be a private or a corporate account. Figure 6.8 shows the UML-B model for this abstraction. The model includes *add*, *remove* and *update* events for the classes.

6.3.2 M1: Adding attributes to classes

In the first refinement, we followed the guidelines of extending the model by adding attributes to each class. The events are extended so adding or removing a class instance will also add and remove its attributes. The attributes for *Member* class includes *name*, *password*, *dob*, *license*, and *e_date* for date of enrollment. Figure 6.9 shows the UML-B model of this refinement. The following event *addMember* extends the abstract one and adds all attributes for this class:

$addMember \mathrel{\widehat{=}}$
extends $addMember$

any

$this\_Member$
p
a
m_name
$m\_password$
m_dob
$m\_license$
e_date

where

grd1: $this\_Member \notin Member$

grd2: $p \in Plan$

grd3: $a \in Account$

grd4: $m\_name \in VARCHAR$

grd5: $m\_password \in VARCHAR$

grd6: $m\_dob \in DATE$

grd7: $m\_license \in \mathbb{Z}$

grd8: $e\_date \in DATE$

grd9: $m\_license \notin ran(member\_license)$

then

act1: $Member := Member \cup \{this\_Member\}$

act2: $member\_plan := member\_plan \cup \{this\_Member \mapsto p\}$

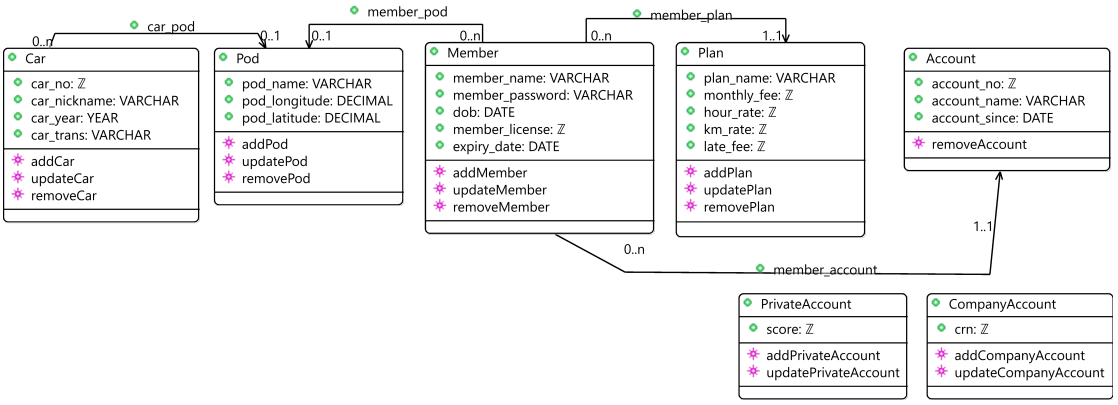

Figure 6.9: Adding attributes to the classes

act3: $member\_account := member\_account \cup \{this\_Member \mapsto a\}$

act4: $member\_name := member\_name \cup \{this\_Member \mapsto m\_name\}$

act5: $member\_password := member\_password \cup \{this\_Member \mapsto m\_password\}$

act6: $dob := dob \cup \{this\_Member \mapsto m\_dob\}$

act7: $member\_license := member\_license \cup \{this\_Member \mapsto m\_license\}$

act8: $expiry\_date := expiry\_date \cup \{this\_Member \mapsto e\_date\}$

end

6.3.3 M2: Modelling secondary classes

In this refinement, we extended the model by adding new classes and associations. The extension is done by the secondary classes that associate between two or more primary classes and some other classes that add details to the secondary class. The *Booking* class describes each a member booking a particular car at some time. The *Trip* class records all trips of a particular booking. Figure 6.10 shows this refinement.

The car sharing system requires some constraints towards timing for it to be accurate and consistent. The following invariants constrain the model:

tripInBooking: $\forall t, b \cdot t \mapsto b \in trip\_booking$
$\Rightarrow trip\_start(t) \geq booking\_start(b) \wedge trip\_end(t) \leq booking\_end(b)$

tripTime: $\forall t, ts, te \cdot t \in Trip \wedge t \mapsto ts \in trip\_start \wedge t \mapsto te \in trip\_end \Rightarrow ts < te$

bookingTime: $\forall b, t, s \cdot b \in Booking$
$\wedge\, b \mapsto t \in time\_booked \wedge b \mapsto s \in booking\_start \Rightarrow t < s$

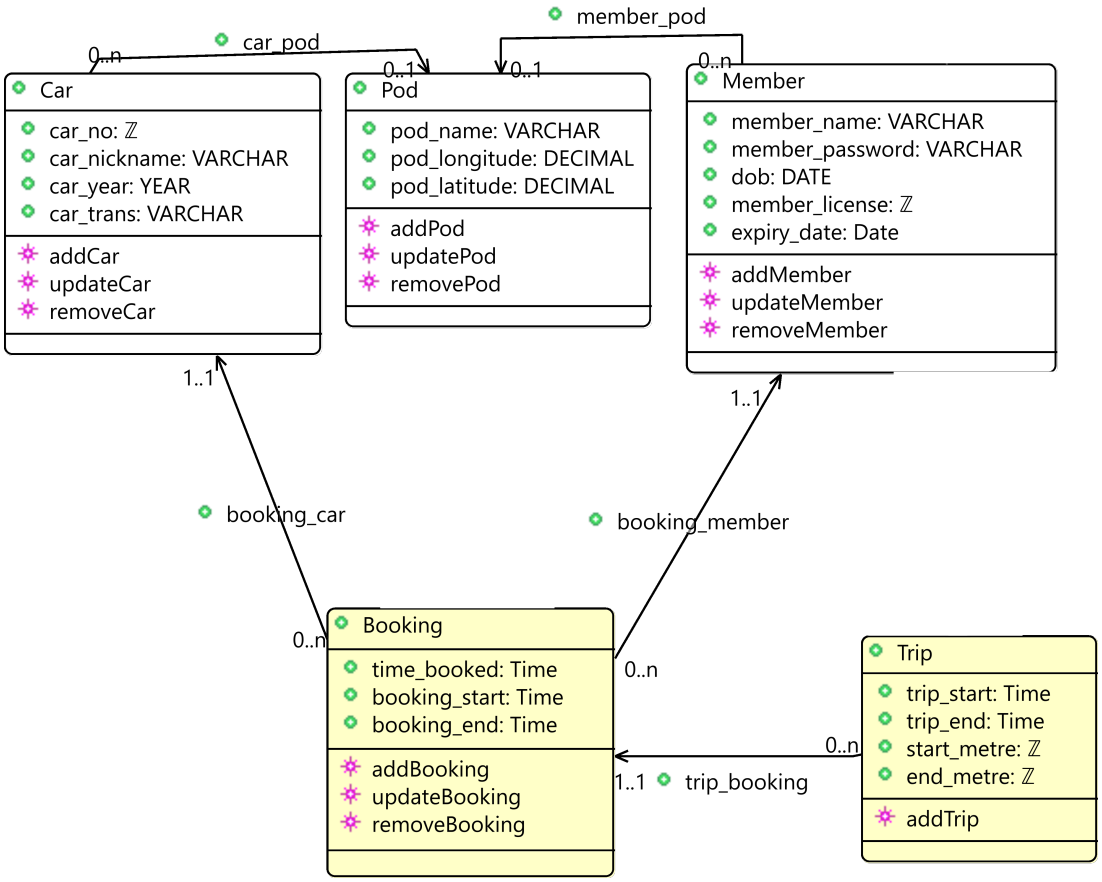

Figure 6.10: Adding secondary classes

The *bookingTime* and *tripTime* invariants specify that the end of a trip or a booking is greater, or later than its start. The *tripInBooking* invariant ensures that every trip associated with a booking happens within the time frame of that booking.

6.3.4 M3: Modelling queries

In this refinement, we modelled examples of queries from the classes as the structure of the model is almost complete. An example is querying cars parked in a particular pod as in the following event using relation inverse as in *grd3*:

$getPodCars \;\widehat{=}$

any

p

car_list

where

grd1: $p \in Pod$

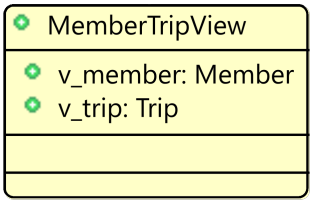

Figure 6.11: Modelling views over classes

grd2: $car\_list \in \mathbb{P}(Car)$

grd3: $car\_list = car\_pod^{\sim}[\{p\}]$

then

$skip$

end

memberTrip-inv: $\forall mv, m, t \cdot mv \in MemberTripView \wedge v\_trip(mv) = t \wedge v\_member(mv) =$

$$m \Rightarrow t \mapsto m \in trip\_booking; booking\_member$$

6.3.5 M4: Modelling simple views

This refinement experiences view modelling in UML-B and Event-B. The *MemberTripView* in Figure 6.11 list trips for each member based on the associations between Trip, Booking and Member classes. The following invariant is added to the model so that members are associated to their trips in the view:

6.3.6 Proof obligations

Table 6.2 shows the statistics of the Car Sharing case study. Similar to the observation in SRES, most of the proofs are discharged automatically. The *m3* machine for the queries includes no proofs.

6.3.7 Generated code for Car Sharing

Appendix B shows both the Event-B model and the generated code for the Car Sharing system using UB2DB. The UB2DB plugin generated the code for both MySQL and Oracle DBMSs as they use slightly different flavours of SQL. An example is the treatment

Machine	No of proofs	Automated	Manual
M0	29	29	0
M1	69	69	0
M2	43	39	4
M3	0	0	0
M4	12	12	0

Table 6.2: POs in the Car Sharing case study

of automatically incremented integer to represent the primary key as MySQL has an `auto increment` type while Oracle does not provide the same feature. As stated in SRES case study, for Oracle we generate a sequence for each class to be used as an incremental primary key integer.

6.4 Emergency Department System

The emergency department system manages patients coming to the emergency department for medical help. The patients are then seen by a doctor and if necessary treated by a nurse. They might be sent home, prescribed some medicines, or they might stay at the hospital. The system does not tackle shift management, or operations and surgeries. While the case study does not tackle every details in the real world, it includes a variety of classes and events that can be easily extended further to deliver a complete system. The aim of modelling this case is to further evaluate that our methodology of incremental development of information system can be shown it is valid for different case studies, hence not every details of the emergency system is modelled or presented here.

6.4.1 M0:Modelling primary classes

This model includes the primary classes of emergency department system. These classes are for *Person* and its sub-classes which are *Worker* and *Patient*. The *Worker* class has three sub-classes which are *Receptionist*, *Nurse* and *Doctor*. Other Primary classes are objects for *Medication*, *Bed* and *Room*. Figure 6.12 shows this abstract model.

6.4.2 M2: Adding attributes to classes

In this refinement, class attributes are added as *person_name* in Figure 6.14. The *person_name* becomes a super-attribute for *Patient* and *Worker* classes. As the *Worker* class is a super-class for *Receptionist*, *Nurse* and *Doctor*, they also inherit *person_name* attribute.

6.4.3 M1:Modelling secondary classes

This model includes the secondary classes that help making association between the primary classes and illustrate further requirements. This includes *Admission* class with association to three primary classes as in Figure 6.13. Each Admission instance records a a patient who is being admitted by a doctor in a given room. Each of the three associations *patient_admin*, *admitted_by* and *admitted_bed* is a total function, meaning each admission tuple must have these three values.

6.4.4 M3: Modelling attribute classes

Attribute classes are introduced in this refinement as *Phone*, *Email* and *Address* in Figure 6.14.

6.4.5 M4: Modelling queries

This refinement involves defining different queries on the model to retrieve valuable information. An example query is to get who prescribed a prescription for a patient as in the event *getWhoPrescribed*.

$getWhoPrescriped \mathrel{\widehat{=}}$

any

p

d

where

grd1: $p \in Patient$

grd2: $d = \{x | x \in presc\_by[patient\_prescription^{\sim}[\{p\}]]\}$

then

skip

end

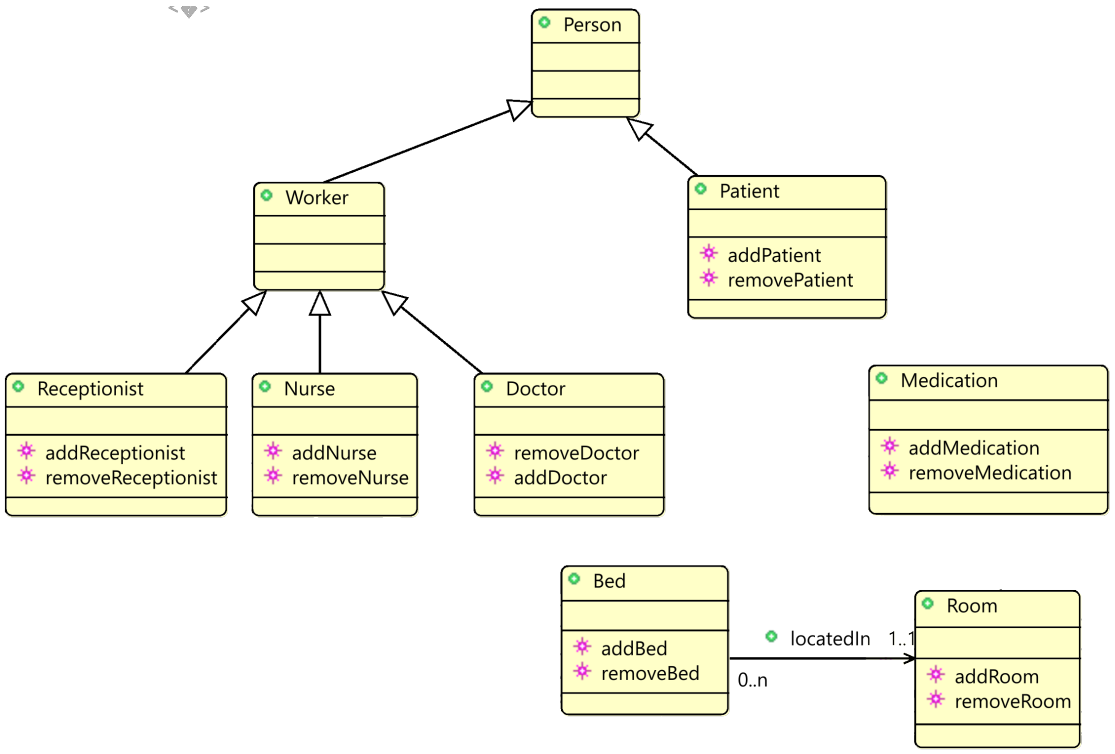

Figure 6.12: Emergency department abstract model

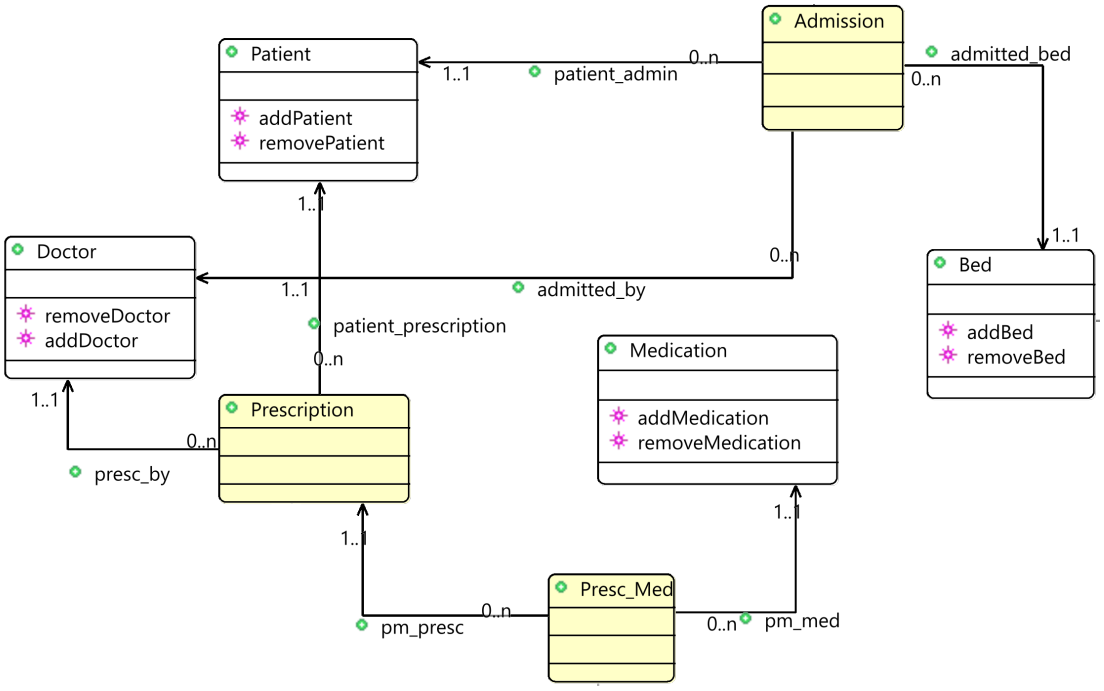

Figure 6.13: Adding secondary classes in Emergency system

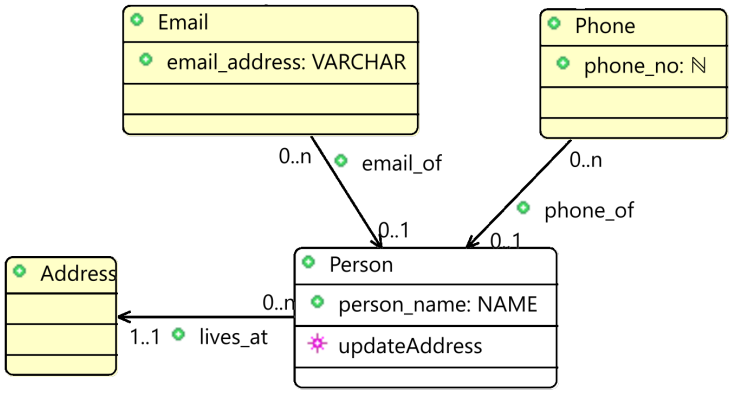

Figure 6.14: Attributes and attribute classes

6.4.6 Differences with previous case studies

The emergency department system shares the same methodology for the design as the other case studies in this thesis. The historical refinement is not covered in this case study.

In this case study, the inheritance between classes is in more than one layer as *Receptionist*, *Nurse* and *Doctor* inherit *Worker*, and *Worker* inherits *Person*. However, this difference has no effect for the translation and code generation.

6.5 Evaluation of modelling approach

The contributed modelling approach as in Chapter 3 has been applied to the previous three case studies. When modelling these case studies, we can observe the following:

- Identifying different class types such as primary and secondary helps clarifying the system complexity and focusing on one aspect in each refinement.
- Depending on how large the system is, some refinements may be combined together such as secondary classes and attribute classes.
- It is important to leave the query modelling to later refinement so that we can model queries from different related classes and attributes.

6.6 Conclusion

Modelling these case study enabled us to define how to model data intensive information system using UML-B. Modelling more than one case study with the same approach

shows that our method for modelling information systems using layered refinement it not coupled with a particular case study, but it can be applied for different case studies.

The case study models show that we can define general patterns for incremental development of information systems. The patterns include incremental refinement as well as how to model different operations and constraints. The generated code shows the application of the translation rules on different case studies as well as the practicality of the tool that is discussed in Chapter 7. The evaluation of the modelling approach on the case studies indicates the importance of the order of the refinement such as the importance of adding queries at later refinement to be able to retrieve data from different classes and attributes.

Chapter 7

Tool Support

7.1 Introduction

Rodin has a tool, called iUML-B, that generates Event-B from UML-B models so that models can be verified using Rodin proof and model-checking capabilities. It provides support to building UML-B diagrams in Rodin and is integrated in an Event-B machine or context. The iUML-B tool is based on the Eclipse Modeling Framework (EMF) for Event-B [92]. The Eclipse Modeling Framework (EMF) is a framework for tools development which provides modelling and code generation facilities [95]. As all these tools are open, this allows us to develop our translation tool, UB2DB, as a plug-in for Rodin using EMF and integrate it with the existing iUML-B tool.

We built a tool, UB2DB (UML-B to DataBase) [1], that provides an automatic generation of SQL code from a model defined in UML-B in the Rodin platform that implements the translation rules defined in Chapter 4. The generated SQL will create the relational database structure using *create* and *alter* commands and provide procedures that populate and manipulate the data. Developers can model their system in UML-B and verify it in the Rodin platform to detect any inconsistency or ambiguity. Then they can use UB2DB to translate the verified model to SQL code. UB2DB translates directly from UML-B and not from Event-B.

7.2 EMF Relational Databases Meta-Model

To translate from UML-B to a relational database, we designed our own meta-model for relational databases which is a model that describes the relational model of databases. Figure 7.2 shows the defined meta-model using the EMF framework. Each database is composed of tables, views, procedures, attributes, references and constraints. Each

[1] https://github.com/albarashdi/UB2DB

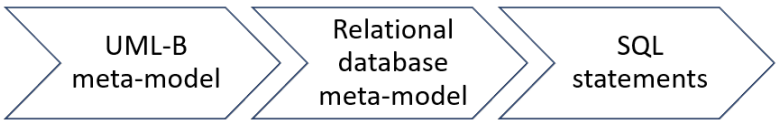

Figure 7.1: The translation process for UB2DB

table may have many attributes and many references that target a specific table which could be the same table. We also included the view component whic is a virtual table that is build from querying from one or more tables that are related together. The view is not related directly to a table but to attributes as it may project many attributes from different tables. Each element has a name associated with it that is reflected in the conceptual model of SQL of that element which identified it from other elements. Along with the name, attributes have types and constraints that specify if an attribute is *not null*, *unique* or has a *default* value when initialized or a check constraint that defines a constraint on its value such as a range of integer value for integer type attribute. The check constraints may be specified for table attributes or view attributes which means a view only has a selection of a table attribute such as only attributes whose values is greater than 70.

Our meta-model for the relational database is different from others such as in [83] as we include operations on the database as procedures. While database operations are the components that affect the database consistency and integrity, they haven not, as per our knowledge, been modelled in other meta-models. While the DBMSs do not explicitly associate a procedure to a table, we provided that explicit link so that we map class events to the corresponding table to represent CRUD operations. However, we also specified a direct connection between a database and procedure(s) so it can be used to translate events that are not associated with a particular class. Not all the entities in the meta-model are used in the current translation of UB2DB tool.

7.3 Translation Process and Approach

The first step in UB2DB is to translate the EMF representation of UML-B to the EMF representation of the database such as translating a class in UML-B to a table, or a class diagram to a database. A second step is to generate the textual SQL code from the database EMF representation as in Figure 7.1. For each UML-B model, there are two translations done, one to SQL by UB2DB, and another to Event-B by UML-B as in Figure 7.3. The Event-B by UML-B translation is done by the iUML-B tool which is already exist. These two translations are separate from each other.

UB2DB takes the translation rules defined in Chapter 4 to generate the database code from class diagram model in UML-B. Each element in UML-B such as class diagram or class has a unique name. UB2DB generates the SQL statements that create a database

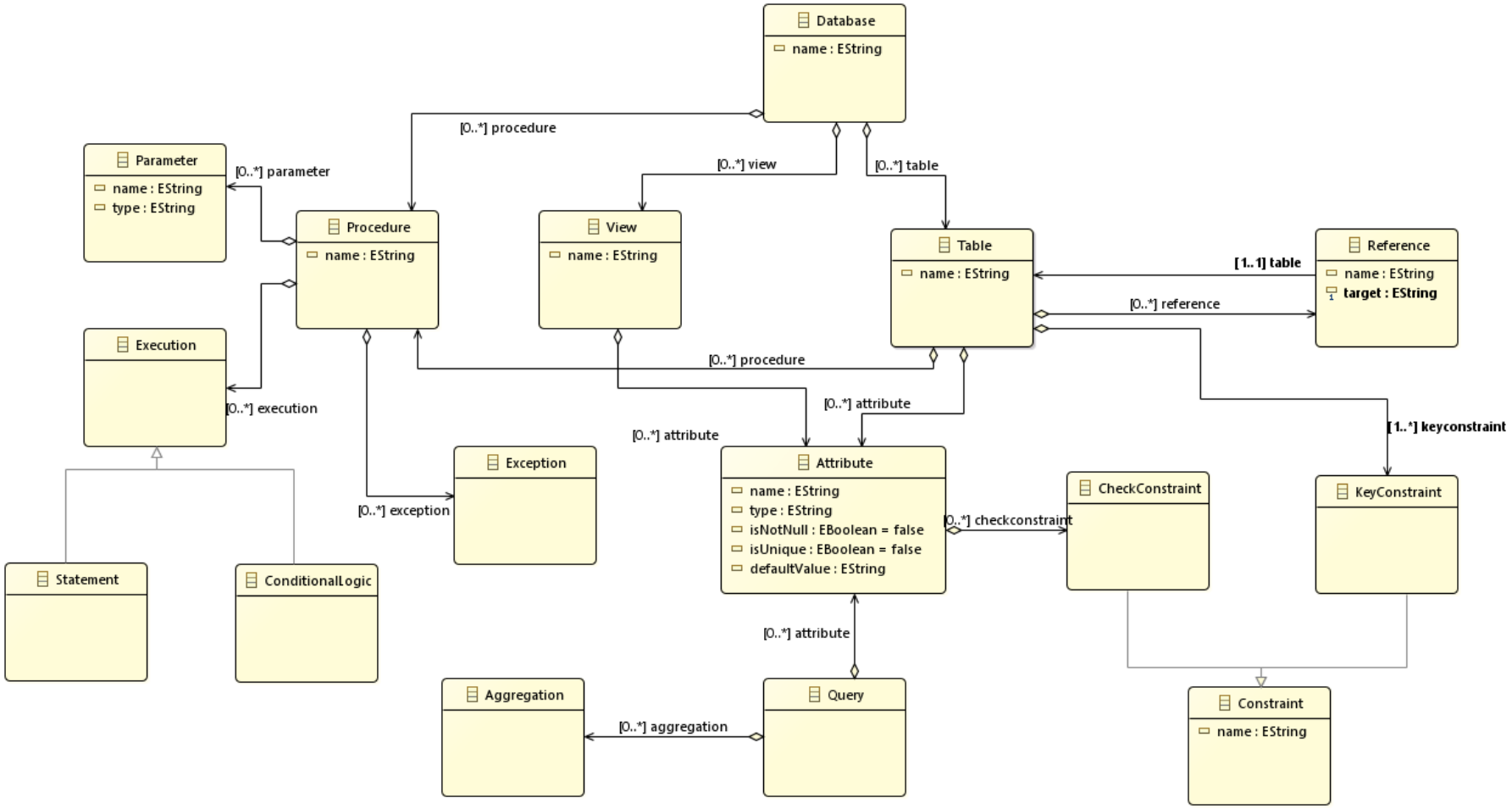

Figure 7.2: Database meta-model defined for UB2DB tool

whose name is given by the class diagram name following Rule_1 and Rule_7, and generates a table for each class in the model based on Rule_2. The associations between classes are translated to relations between tables. Each class attribute in the UML-B model will result in an attribute in the corresponding table. Each component such as association or attribute is translated into a separate statement in the generated SQL. This way, dealing with refinement will be easier as adding a new attribute for a class in a refined model will correspond to adding an SQL statement that adds that attribute to the table instead of going through the whole process of creating a table again.

UB2DB translates the inheritance between classes based on the association from the sub classes to other classes. The tool checks if sub classes have any outgoing association then it will generate the command to create the sub tables, otherwise, only the super table will be generated and all attributes of the sub class are added to the super class. This is based on the experiment result on different patterns for inheritance mapping to a relational model.

Views in relational database, do not have a special graphical representation in UML-B. To translate them to views we restrict the modeller to use a naming convention for them by including *View* in the class name. This is so that the tool can easily distinguish between normal classes and views. Simple views are supported by the tool and translated to views given that their naming convention includes *view*.

Each constructor event in UML-B is translated by UB2DB into a *procedure* with the `insert into table` statement in SQL. The procedure takes all class attributes and associations as parameters for the insertion. Destructor events are translated into procedure with the `delete from table` statement in SQL. Normal events are translated into procedures with an `update table` statement if the event has an override operator, or to a `select from` statement if the event does not have an action.

The UB2DB translation is implemented using a generic EMF translation plugin which is provided by researchers at the University of Southampton [1]. Translation and rules are contributed using the Eclipse extension mechanism. For each component in the database meta-model such as table or attribute, there is a rule defined by a Java class to translate UML-B to it. Each rule in UB2DB has *fire* and *dependencyOk* methods. The *fire* method does the mapping between UML-B elements to database elements. In the translation process, dependencies must be checked by the *dependencyOk* method before proceeding to the translation. A table is dependant on a database which means it cannot be generated before the database, and an attribute is dependant on a table as shown in Figure 7.4. This also ensures an ordering of the translation of different components.

To translate the UML-B model, the user needs to select the class diagram from Event-B explorer in Rodin then activate the translation by clicking on UB2DB icon in the tool bar as in Figure 7.5. The tool then will generate the SQL files for the whole model as well as separate SQL files for the new features only.

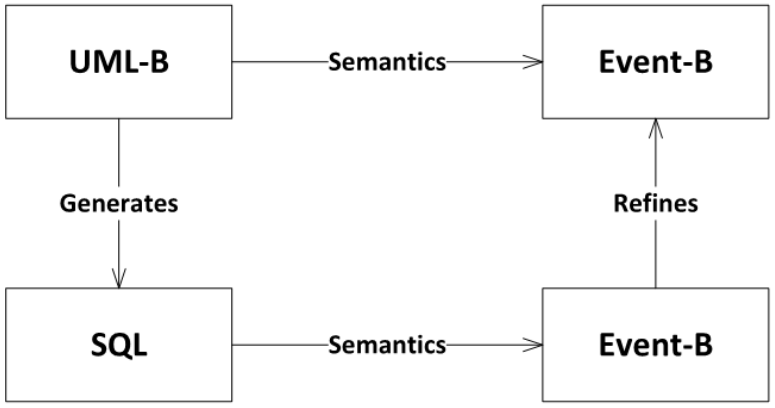

Figure 7.3: Translation from UML-B model to Event-B and database

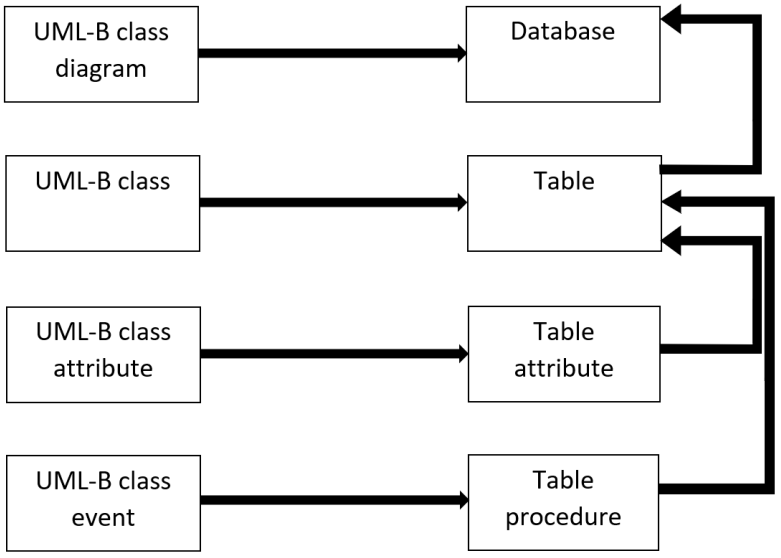

Figure 7.4: Dependencies in UB2DB

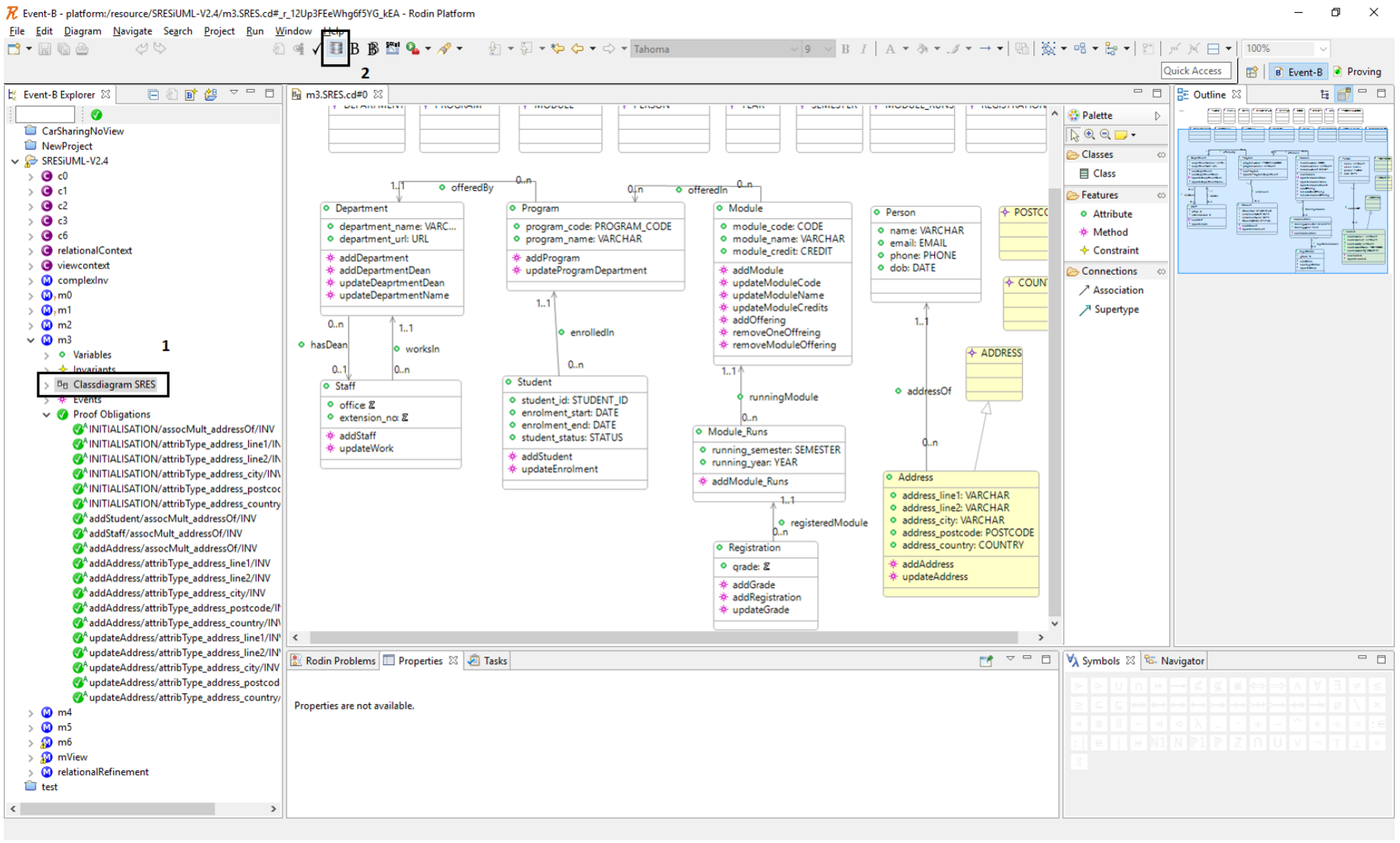

Figure 7.5: A demo for UB2DB

7.4 Tool limitations

While UB2DB generates the SQL and stored procedures following the rules 1-18 successfully, it does have some limitations in its current version.

One limitation is that the tool does not traverse back in the refinement chain. For a machine *m1* that refines *m0*, UB2DB will not see *m0* if it was executed for *m1*. This limitation has an effect on inheritance translation as the super type arrow does not appear in the refinement in iUML-B and UB2DB will not see the inheritance link which means that the sub classes will not be associated with the super class if translating to class table inheritance pattern. Moreover, the additional constraints that are defined in Section 4.5 are not automatically generated by UB2DB.

As UB2DB takes the UML-B model for translation and not the Event-B, any specifications that are specified in Event-B model outside UML-B will not be treated by UB2DB.

Only expressions and predicates that are used in Chapters 3 and 4 are translated by the tool. If the modeller modelled operations differently, they will not be translated by UB2DB and the modeller will not get a warning or an error in the current version.

For conditional delete and update where the condition is a complex one, the user need to manually translate the condition in `where` clause as the tool does not translate the special conditions yet. While the semantics and translation rules are defined for this, the tool support is not implemented yet.

7.5 Tool Evaluation

UB2DB is evaluated and executed against the case studies in Chapter 6. Starting from an abstract model, as in Figure 7.6 for the car sharing case study, where classes have associations but no attributes, the tool generates the tables with one attribute as a key for each table. Then the associations are added to the source tables as attributes that reference the target table.

Further refinement of the model might include adding attributes to different classes as in Figure 7.7 for the SRES case study. Another refinement could add more detail to the model by introducing new classes and associate them to classes in the abstract model such as the *Booking* class for car sharing in Figure 7.8. Attributes and relations added in later refinements are translated to the `alter table` command so that we can build on the previously generated database without rebuilding the database. The `alter` command can be used to modify the structure of a table by adding, modifying or deleting attributes or relations.

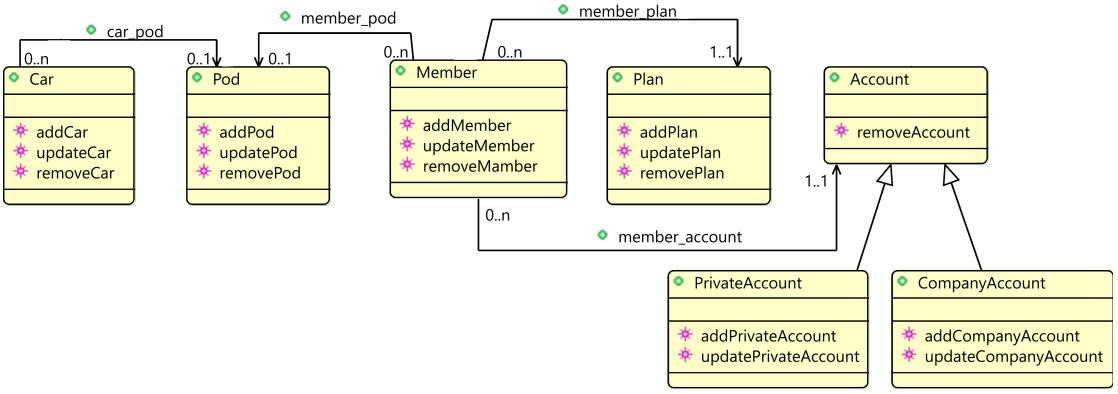

Figure 7.6: Abstract model for car sharing case study

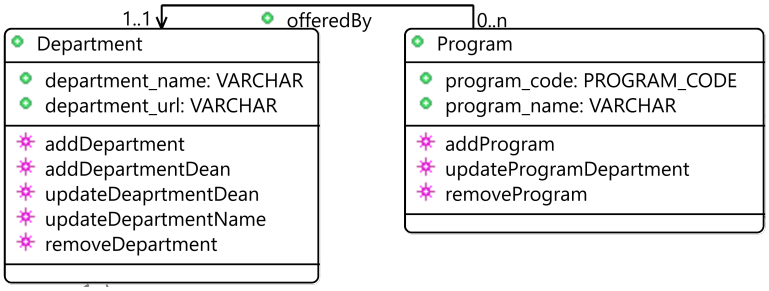

Figure 7.7: Refinement by adding attributes in Program and Department classes

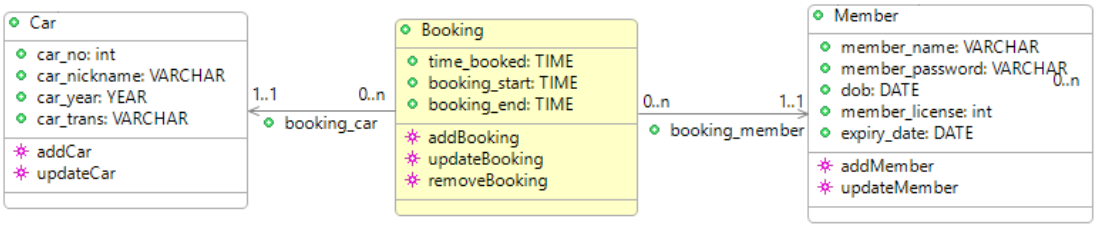

Figure 7.8: New classes in refinement model

UB2DB generates the code and saves it in SQL script files with sql extensions. The code is separated into different files for table script, attributes, association, . . . etc. The generated SQL code for two case studies was successfully imported in Oracle database management system and all the supported database structures and constraints were successfully generated. This includes generating intermediate tables and assigning different constraints such as primary key, foreign key, not null, uniqueness, default value and basic check constraints. Events were translated to procedures for constructors, destructor or normal events. For any constructor event, the tool generated a procedure with a name as the event name and took the class attributes and associations as parameters for the procedure. Destructor events were translated to procedures with one parameter corresponding to a key for the record to be deleted if the condition for the deletion is not given. Normal events were translated to either update or select procedures according to the rules. However, UB2DB does not yet deal with queries with aggregation functions

such as *max*, *min*, and *average*.

7.5.1 Generated code

UB2DB generated the code of the case studies as defined by the translation rules. We show here few examples on one of the classes in SRES case study. For class definition such as Department in SRES case study, the following code is generated:

```
CREATE TABLE Department(Department_id INT PRIMARY KEY);
```

This is generated based on *Rule_2* which takes the source function *translate class (cn,attributes, associations)* and generates *create table(cn, cn id int primary key, translate attributes, translate associations)*. At this refinement level as in Chapter 3, there is no attributes in the class. However, the following is generated for the association based on *Rule_7*:

```
ALTER TABLE Department
        ADD hasDean INT
        CONSTRAINT Department_Staff_FK REFERENCES Staff(Staff_id);
```

The tool concatenates the source and the target of the association with the suffix *FK* as the constraint name.

The constructor event *addDepartment* is translated to the following PL/SQL code based on rules 10, 11, 12, and 18:

```
CREATE OR REPLACE PROCEDURE addDepartment(v_department_name IN VARCHAR,
        v_department_url IN INT, v_hasDean IN INT) IS
BEGIN
        INSERT INTO Department VALUES (Department_seq.NEXTVAL,
        v_department_name, v_department_url, v_hasDean);
END;
```

An update event that assigns a dean to a department is translated to the following:

```
CREATE OR REPLACE PROCEDURE updateDeaprtmentDean(id IN INT, s IN INT)IS
BEGIN
        UPDATE Department
        SET hasDean = s
        WHERE Department_id = id;
END;
```

Appendix A and Appendix B show the generated code for two case studies.

The UB2DB plugin generated the code for both MySQL and Oracle DBMSs as they use slightly different flavours of SQL. An example is the treatment of automatically incremented integer to represent the primary key as MySQL has an `auto increment` type while Oracle does not provide the same feature. For Oracle, we generate a sequence for each class to be used as an incremental primary key integer.

7.5.2 Code performance

To evaluate the performance of the generated code, we tested the stored procedure for adding records of a table and compared it to the manual insert statement. The structure of both manual and automated are similar except that in the manual we used prepared statements with insert statement, while the generated code used stored procedure. The time is calculated using `System.currentTineMillies()` method in Java by taking the time before and after the execution of the stored procedures and insert statements. Adding 10000 Pods took in average 2720 ms using the *addPod* procedure generated by UB2DB. The same amount of the records took 2247 ms using the manual insert statement. While the generated code to perform as good as the manual code, it did not fall far behind as it was slower by around 21%.

7.6 Comparison with EventB2SQL tool

The EventB2SQL tool is the only tool available we could compare UB2DB with. The EventB2SQL tool translates abstract Event-B models to databases. It gives the user the option to generate Java with JDBC, or a mobile application with SQLite embedded database, or a PHP web site with the database code to generate it. The operations to populate the database are generated automatically even if these operations (insert, delete and get) are not modelled in the Event-B. By evaluating case studies, we found that some of the constraints such as total function attributes and injective attributes that are translated to not null and unique constraint respectively are not preserved in the code generated by EventB2SQL.UB2DB has a huge gain in performance compared to EventB2SQL as in the later, adding 100 records took 2 minutes [25]. This is compared to adding 10000 Pods in UB2DB took in average 2720 ms. The two records are of comparable complexity of the number and types of the attributes. Table 7.1 summarises the differences between UB2DB and EventB2SQL.

Another noticeable difference between the two tools is the size of a model they can translate. When applying EventB2SQL on the SRES case study, the tool hung during the process of creating the Java class. By investigating the issue, it turned that the maximum Java heap size is not enough to generation of the file as it is a big Java class with thousands of lines of code. We increased the maximum Java heap size to over 512MB in order for the code to be generated. This was not an issue with UB2DB as the tool generates small .sql files for the structure and operations of the model.

	UB2DB	**EventB2SQL**
Delivered as	Rodin plugin	Rodin plugin
Starting point	UML-B	Event-B
Generated code	SQL and PLSQL	Java and JDBC, PHP, Android
Performance	10000 records 2720 ms	100 records in 2 minutes
Supports refinement	Yes	No

Table 7.1: UB2DB verses EventB2SQL

7.7 Conclusion

UB2DB is a tool that translates UML-B models to relational databases by generating SQL statements that build the database and structure its tables and relations. The UML-B model is translated by the UML-B tool to Event-B for verification. UB2DB provides support to translate different components in UML-B model into code that reserves the constraints such as *not null* and *unique*. It also provides support for events that create new instances of classes, delete an existing one, update its attributes or select from one or more classes.

The evaluation of UB2DB shows that it can generate the database code for different refinement in different case studies and its performance is not far behind a hand written code. Comparing it to the other available demonstrates that UB2DB generates the code that preserves the constraints defined in the model while EventB2SQL failed to preserve all of them.

Chapter 8

Conclusions and Future Work

8.1 Summary of Research

Formal modelling and specification of database systems is an important concept which has been covered in much literature. The importance of verified database design lies in critical domains and decisions that depend on correct and consistent data.

Generating database code and applications from formal models has been achieved partially, as discussed in different literature. While most of them provide translation from formal methods directly, the authors in [69] developed a tool to transfer UML-like models to B specification and then to a database application, yet their work lacks some important aspects and properties of databases like views, performance and security.

This research provides a method and a tool for modelling information systems in UML-B and translating the model to SQL code. The model is translated by the existing tool, iUML-B, to Event-B for verification and by our tool UB2DB to database code. The research provides a practical approach and a supporting tool for modelling the databases with different constraints and through layered refinement.

The tool supports translating to different constraints in SQL such as primary key, foreign key, not null, unique and check constraints. The generated SQL code by the tool satisfies the system requirements and constraints and is validated against them for the case studies. A modeller can model different operations on an information system and generate stored procedures for them. An important assumption is that the user will only use the SQL and stored procedures generated by the tool. This means that any changes or extensions should be made to the UML-B model rather that to the generated code. The generated stored procedures can be called from any programming languages that provide a support for communicating with Oracle databases.

8.2 Contributions

We can summarise the contributions in this thesis compared to other similar works reviewed in Section 2.5, and in relation with the expected contributions listed in Chapter 1 as follows:

- **Contribution 1**: This research defines an approach for modelling information systems in UML-B and generating the database code from it. It advises guidelines for modelling such systems in different refinement levels using UML-B and Event-B. The reviewed literature does not tackle how to structure the model in different refinement levels where in each refinement the modeller introduces some concepts for specification and verification. Where the refinement is used in the reviewed work, such as in [68], it was a refinement for implementation where the concrete model becomes closer to an implementation language. While layered refinement is well used when modelling in Event-B as stated is [23] and [3], the contribution of this thesis is applying that to database design with distinction of different concepts such as *primary* and *secondary* classes.

- **Contribution 2**: Chapter 6 outlines different case studies that we modelled using UML-B and Event-B. It covers three unrelated case studies for student enrollment system, car sharing system and emergency department system. These case studies shows a consistent methodology when modelling information system in UML-B and Event-B. These case studies evaluated the methodology by showing its applicability to different systems and that it can be generalised for any case study.

- **Contribution 3**: Chapter 4 defines general rules to translate UML-B and Event-B models to database code. While [103] is the only other work, to our knowledge, that generates database code from Event-B, it does not include graphical representation using UML-B and the translation seems to violate constraints such as not null and uniqueness as concluded in [8]. The translation takes into account different refinement levels by using `alter` command which enables generating code for extra features or details that are introduced in further refinement. This feature is not available in the reviewed work in [68] which does not support adding and translating new classes and attributes in refinement. The contribution of this thesis is defining translation rules that support extension to the databases in later refinement.

  Preserving event atomicity is an important feature when translating events to database operations, yet it wasn not explicitly dealt with in the related work. For an event that might be translated to a procedure with more than one SQL statement, our translation defines a mechanism using transaction management in SQL to ensure that all the statements translated from a single event are atomic and within one transaction unit.

The defined translations in this thesis translates events to stored procedure instead of plain SQL statements. The stored procedures involve parameters, guards and actions of events in Event-B. Users of the generated code might use any application language that they are familiar with and call the stored procedures without worrying about affecting the consistency of the database as all that is preserved by the stored procedures. The reviewed works translate the specification to a combination of SQL and a programming language such as Java in [69] and [103] or to Delphi programs as in [63].

- **Contribution 4**: This research includes the definition of the splitting refinement and how to model that in Event-B. While the introduction of intermediate table is well known in relational database community, the contribution of this thesis is how to model that in Event-B in an abstraction and refinement approach. While inheritance representation in the relational model in not a new topic as presented in [49], its evaluation on performance and constraints was not fully discussed. This includes the need to define extra constraints or triggers in the database in order to preserve the correctness as in the object model in UML-B, then evaluate the performance after enforcing these constraints.

- **Contribution 5**: The UB2DB tool is developed in this thesis which translates UML-B models to database code for structure and manipulation. A contribution of this thesis is developing and providing this tool which automatically generates the SQL and PL/SQL code. As the tool can generate the database code from each refinement with an alter command for attributes and constraints, an extension to the database can be generated later from a new refinement while preserving the consistency of the current database that was generated from the abstract model. The new requirements or details will be modelled as a refinement of the last model. Then by using UB2DB, the user can generate the code from the new refinement and ensure it will not violate any constraints in the currently running database. However, this can be used with some limitations. If the new refinement does not include association or references to instances from the old classes, then the code of the refinement can be imported to the database without violating the consistency of the system. However, if the refinement includes adding association to the new class from a class in an abstract model, then we can not import the code of the refinement model in a populated database as the tables generated from the new classes will be empty. Exceptions to this will be if the new association is a partial function which is translated to attribute with a null value, or if the attribute has a default value that already exists in the populated database.

  The relational database meta-model that is defined using EMF ECore specifies procedures that operate on tables. Other meta-models in [83] only consider the structure of the data without the operations on them.

While Event-B is not used to model or prove performance property, it is important to evaluate the performance of the generated code in order to show its practicability for the intended user. We do not expect our generated code to be faster than a hand written one, or even as fast as. However, the benefits of correct implementation and consistency outweigh the lower performance. Evaluating the performance of inserting 10000 records using the generated code shows that our code performed around 21% slower than a hand written code. Also, the performance shows major difference compared to the other available tool, Event-B2SQL in which adding 100 records takes 2 minutes [25].

Compared to the code generated from another tool, our automatically generated code is minimal and easy to understand by the user. On the other hand, Event-B2SQL generates complex Java code that is almost impossible to understand and comprehend as the code generated from the whole model is wrapped into single Java class with no separation of structure and operations. To generate the code of SRES case study using Event-B2SQL, we needed to increase the Java virtual machine memory (heap) in order for the file to be generated.

- Representation and translation to **all CRUD operations**. UB2SQL work only consider, a part from queries, insert and delete [68]. Update was not dealt with neither as operation to translate to, nor as semantic from SQL to formal method.

- Our modelling and translation adds a level of support to database security. One of the primary defences against SQL injection attacks is using stored procedures as indicated by the Open Web Application Security Project (OWASP) [78]. We translate events in UML-B to stored procedures. The stored procedures are parameterised which then prevent an attacker from submitting malicious SQL. Such concerns were not dealt with in the reviewed work.

- **Contribution 6**: This research defines formal semantics for relational database and stored procedures in Event-B. The defined semantics shows the correctness of the generated code. While the formal semantics were defined in previous work such as in [52, 74], the focus was on queries, or did not provide full coverage for procedures with transaction management as in [70]. Our semantics focuses on operations that manipulate the database which could cause data inconsistency. When defining the semantics and soundness of the translation, we looked at the whole stored procedures rather than just the SQL statements. This means events with their parameters, guards and actions are correctly translated. None of the related work, according to our knowledge, considered defining semantic for parameterised database operations. This is important as each SQL statement is triggered and executed multiple times with different value for each.

8.3 Future Work

While this work provided a comprehensive methodology and tool support to incrementally modelling information systems in formal methods with automated code generation, it has the possibility of many improvements and further research which can build on it. Some further improvements can be specified as:

- **Automated normalisation**: In the future, the methodology and the tool support would benefit from studying the normalisation and functional dependencies and investigating how they can be automated in the process of relational database design. Then we can integrate the automation with the tool so that a database generated by UB2DB can automatically be normalised to a particular normal form. Akehurst et al. in [7] represented a method to automate the process of database normalisation via meta-modelling using UML and OCL. Their approach uses OCL to encode the definitions of normal forms (up to BCNF) over data modelling profile of UML. Then they use graph rewriting rules to reconstruct the data model from one normal form to a higher normal form with tool support. SQL definitions of the database are not generated by the tool. This can form a basis for an integration of normalisation rules in the system design within UML-B as invariants, then based of the desired normal form, the UML-B class diagram could be rewritten.

- **Support for complex views**: Further improvement could be by extending our support to the views to model and translate views over multiple base classes. This aims to simplify the complexity of common queries where the same data from multiple classes is retrieved very often. This is an extension to our support for modelling views over a single class that aim to hide some information from the base class.

- **Design patterns**: Further improvement is to investigate different design patterns for modelling relational database which can be derived from different diverse case studies. The patterns will define how provides a solution for common problems when modelling information systems that are data intensive.

- **Support for non-relational databases**: We aim to extend on the modelling, translation and the tool to support non-relational databases such as NoSQL to introduce the tool to wider applications. The tool then will give the user the ability to choose a target model for the translation of the same UML-B model.

- **Round-trip translation**: With the translation rules and the semantics of the generated code are all using Event-B, we can extend on the tool so that users can import databases code in SQL and PL/SQL to the Rodin platform and translated to UML-B class diagrams to prove the database and refine it.

Lastly, the UB2DB tool will be released as an open source to the public as well as the case studies. This will help others to build on the developed tool and extend its support to cover other database models such as NoSQL. This will also provides valuable feedback for both the methodology of modelling as well as the tool performance and efficiency.

Appendix A

Generated Code for SRES Case Study

A.1 Creating structure

```
CREATE TABLE Program(Program_id INT PRIMARY KEY);
CREATE SEQUENCE Program_seq
START WITH 1
INCREMENT BY 1;
ALTER TABLE Program ADD program_code INT NOT NULL UNIQUE;
ALTER TABLE Program ADD program_name VARCHAR2(255) NOT NULL UNIQUE;
CREATE TABLE Department(Department_id INT PRIMARY KEY);
CREATE SEQUENCE Department_seq
START WITH 1
INCREMENT BY 1;
ALTER TABLE Department ADD department_name VARCHAR2(255) NOT NULL UNIQUE;
ALTER TABLE Department ADD department_url INT NOT NULL UNIQUE;
CREATE TABLE Staff(Staff_id INT PRIMARY KEY);
CREATE SEQUENCE Staff_seq
START WITH 1
INCREMENT BY 1;
ALTER TABLE Staff ADD office INT;
ALTER TABLE Staff ADD extension_no INT;
CREATE TABLE Student(Student_id INT PRIMARY KEY);
CREATE SEQUENCE Student_seq
START WITH 1
INCREMENT BY 1;
ALTER TABLE Student ADD student_id INT NOT NULL UNIQUE;
ALTER TABLE Student ADD enrolment_start DATE NOT NULL;
ALTER TABLE Student ADD enrolment_end DATE NOT NULL;
ALTER TABLE Student ADD student_status INT NOT NULL;
CREATE TABLE Person(Person_id INT PRIMARY KEY);
```

```
CREATE SEQUENCE Person_seq
START WITH 1
INCREMENT BY 1;
ALTER TABLE Person ADD name VARCHAR2(255) NOT NULL;
ALTER TABLE Person ADD email INT NOT NULL UNIQUE;
ALTER TABLE Person ADD phone INT NOT NULL UNIQUE;
ALTER TABLE Person ADD dob DATE NOT NULL;
CREATE TABLE Module(Module_id INT PRIMARY KEY);
CREATE SEQUENCE Module_seq
START WITH 1
INCREMENT BY 1;
ALTER TABLE Module ADD module_code INT NOT NULL UNIQUE;
ALTER TABLE Module ADD module_name VARCHAR2(255) NOT NULL UNIQUE;
ALTER TABLE Module ADD module_credit INT NOT NULL;
CREATE TABLE Module_Runs(Module_Runs_id INT PRIMARY KEY);
CREATE SEQUENCE Module_Runs_seq
START WITH 1
INCREMENT BY 1;
ALTER TABLE Module_Runs ADD running_semester INT NOT NULL;
ALTER TABLE Module_Runs ADD running_year INT NOT NULL;
CREATE TABLE Registration(Registration_id INT PRIMARY KEY);
CREATE SEQUENCE Registration_seq
START WITH 1
INCREMENT BY 1;
ALTER TABLE Registration ADD grade INT;
CREATE TABLE Address(Address_id INT PRIMARY KEY);
CREATE SEQUENCE Address_seq
START WITH 1
INCREMENT BY 1;
ALTER TABLE Address ADD address_line1 VARCHAR2(255) NOT NULL;
ALTER TABLE Address ADD address_line2 VARCHAR2(255);
ALTER TABLE Address ADD address_city VARCHAR2(255);
ALTER TABLE Address ADD address_postcode INT NOT NULL;
ALTER TABLE Address ADD address_country INT NOT NULL;
CREATE TABLE Live_Student(Live_Student_id INT PRIMARY KEY);
CREATE SEQUENCE Live_Student_seq
START WITH 1
INCREMENT BY 1;
CREATE TABLE Graduated_Student(Graduated_Student_id INT PRIMARY KEY);
CREATE SEQUENCE Graduated_Student_seq
START WITH 1
INCREMENT BY 1;
CREATE TABLE Completed_Registration(Completed_Registration_id INT PRIMARY KEY);
CREATE SEQUENCE Completed_Registration_seq
START WITH 1
INCREMENT BY 1;
ALTER TABLE Student
```

```
        ADD enrolledIn INT
        CONSTRAINT Student_Program_FK REFERENCES Program(Program_id)
 NOT NULL;
ALTER TABLE Staff
        ADD worksIn INT
        CONSTRAINT Staff_Department_FK REFERENCES Department(Department_id)
 NOT NULL;
ALTER TABLE Program
        ADD offeredBy INT
        CONSTRAINT Program_Department_FK REFERENCES Department(Department_id)
 NOT NULL;
ALTER TABLE Department
        ADD hasDean INT
        CONSTRAINT Department_Staff_FK REFERENCES Staff(Staff_id)
;
ALTER TABLE Module_Runs
        ADD runningModule INT
        CONSTRAINT Module_Runs_Module_FK REFERENCES Module(Module_id)
 NOT NULL;
CREATE TABLE offeredIn(
        Module_id INT,
        Program_id INT,
        PRIMARY KEY ( Module_id, Program_id),
        FOREIGN KEY (Module_id) REFERENCES Module(Module_id),
        FOREIGN KEY (Program_id) REFERENCES Program(Program_id)
);
ALTER TABLE Registration
        ADD registeredModule INT
        CONSTRAINT Registration_Module_Runs_FK REFERENCES Module_Runs(
    Module_Runs_id)
 NOT NULL;
ALTER TABLE Address
        ADD addressOf INT
        CONSTRAINT Address_Person_FK REFERENCES Person(Person_id)
 NOT NULL;
ALTER TABLE Registration
        ADD registeredStudent INT
        CONSTRAINT Registration_Live_Student_FK REFERENCES Live_Student(
    Live_Student_id)
 NOT NULL;
ALTER TABLE Completed_Registration
        ADD graduatedRegistration INT
        CONSTRAINT Com_Gra_FK REFERENCES Graduated_Student(Graduated_Student_id
    )
 NOT NULL;
```

A.2 Translated Events to Stored Procedures

```
CREATE OR REPLACE PROCEDURE addProgram(v_program_code IN INT, v_program_name IN
     VARCHAR, v_offeredBy IN INT)
IS
BEGIN
         INSERT INTO Program VALUES (Program_seq.NEXTVAL,v_program_code,
    v_program_name, v_offeredBy);
END;

CREATE OR REPLACE PROCEDURE updateProgramDepartment(id IN INT, d IN INT)
IS
BEGIN
UPDATE Program
        SET offeredBy  = d
        WHERE Program_id = id;
END;
CREATE OR REPLACE PROCEDURE getProgramOffering()
IS
temp_var INT;
BEGIN
        SELECT Module_id INTO temp_var
        FROM Module
        WHERE  offeredIn = id;
END;
CREATE OR REPLACE PROCEDURE getMultiProgramOffering(array IN INT)
IS
temp_var INT;
BEGIN
        SELECT Module_id AS module_list
        FROM Module
        WHERE  offeredIn IN (array);
END;
CREATE OR REPLACE PROCEDURE addDepartment(v_department_name IN VARCHAR,
    v_department_url IN INT, v_hasDean IN INT)
IS
BEGIN
         INSERT INTO Department VALUES (Department_seq.NEXTVAL,
    v_department_name, v_department_url, v_hasDean);
END;

CREATE OR REPLACE PROCEDURE updateDeaprtmentDean(id IN INT, s IN INT)
IS
BEGIN
UPDATE Department
        SET hasDean  = s
        WHERE Department_id = id;
```

```
END;
CREATE OR REPLACE PROCEDURE updateDepartmentName(id IN INT, d_name IN VARCHAR)
IS
BEGIN
UPDATE Department
        SET department_name  = d_name
        WHERE Department_id = id;
END;
CREATE OR REPLACE PROCEDURE addStaff(v_office IN INT, v_extension_no IN INT,
    v_worksIn IN INT)
IS
BEGIN
         INSERT INTO Staff VALUES (Staff_seq.NEXTVAL,v_office, v_extension_no,
    v_worksIn);
END;

CREATE OR REPLACE PROCEDURE updateWork(id IN INT, d IN INT)
IS
BEGIN
UPDATE Staff
        SET worksIn  = d
        WHERE Staff_id = id;
END;
CREATE OR REPLACE PROCEDURE addStudent(v_student_id IN INT, v_enrolment_start
    IN DATE, v_enrolment_end IN DATE, v_student_status IN INT, v_enrolledIn IN
    INT)
IS
BEGIN
         INSERT INTO Student VALUES (Student_seq.NEXTVAL,v_student_id,
    v_enrolment_start, v_enrolment_end, v_student_status, v_enrolledIn);
END;

CREATE OR REPLACE PROCEDURE updateEnrolment(id IN INT, p IN INT)
IS
BEGIN
UPDATE Student
        SET enrolledIn  = p
        WHERE Student_id = id;
END;
CREATE OR REPLACE PROCEDURE addModule(v_module_code IN INT, v_module_name IN
    VARCHAR, v_module_credit IN INT, v_offeredIn IN INT)
IS
BEGIN
         INSERT INTO Module VALUES (Module_seq.NEXTVAL,v_module_code,
    v_module_name, v_module_credit, v_offeredIn);
END;
```

```
CREATE OR REPLACE PROCEDURE updateModuleCode(id IN INT, code IN INT)
IS
BEGIN
UPDATE Module
        SET module_code  = code
        WHERE Module_id = id;
END;
CREATE OR REPLACE PROCEDURE updateModuleName(id IN INT, m_name IN VARCHAR)
IS
BEGIN
UPDATE Module
        SET module_name  = m_name
        WHERE Module_id = id;
END;
CREATE OR REPLACE PROCEDURE updateModuleCredits(id IN INT, credit IN INT)
IS
BEGIN
UPDATE Module
        SET module_credit  = credit
        WHERE Module_id = id;
END;
CREATE OR REPLACE PROCEDURE getModuleOffering()
IS
temp_var INT;
BEGIN
END;
CREATE OR REPLACE PROCEDURE addModule_Runs(v_running_semester IN INT,
    v_running_year IN INT, v_runningModule IN INT)
IS
BEGIN
         INSERT INTO Module_Runs VALUES (Module_Runs_seq.NEXTVAL,
    v_running_semester, v_running_year, v_runningModule);
END;

CREATE OR REPLACE PROCEDURE addRegistration(v_grade IN INT, v_registeredModule
    IN INT, v_registeredStudent IN INT)
IS
BEGIN
         INSERT INTO Registration VALUES (Registration_seq.NEXTVAL,v_grade,
    v_registeredModule, v_registeredStudent);
END;

CREATE OR REPLACE PROCEDURE addRegistration(v_grade IN INT, v_registeredModule
    IN INT, v_registeredStudent IN INT)
IS
BEGIN
```

```
         INSERT INTO Registration VALUES (Registration_seq.NEXTVAL,v_grade,
    v_registeredModule, v_registeredStudent);
END;

CREATE OR REPLACE PROCEDURE updateGrade(id IN INT, g IN INT)
IS
BEGIN
UPDATE Registration
        SET grade  = g
        WHERE Registration_id = id;
END;
CREATE OR REPLACE PROCEDURE addAddress(v_address_line1 IN VARCHAR,
    v_address_line2 IN VARCHAR, v_address_city IN VARCHAR, v_address_postcode
    IN INT, v_address_country IN INT, v_addressOf IN INT)
IS
BEGIN
         INSERT INTO Address VALUES (Address_seq.NEXTVAL,v_address_line1,
    v_address_line2, v_address_city, v_address_postcode, v_address_country,
    v_addressOf);
END;

CREATE OR REPLACE PROCEDURE updateAddress(id IN INT, line1 IN VARCHAR, line2 IN
     VARCHAR, city IN VARCHAR, postcode IN INT, country IN INT)
IS
BEGIN
UPDATE Address
        SET address_line1  = line1, address_line2  = line2, address_city  =
    city, address_postcode  = postcode, address_country  = country
        WHERE Address_id = id;
END;
```

Appendix B

Generated Code for Car Sharing Case Study

B.1 Creating Structure

Tables and sequences:

```
CREATE TABLE Car(Car_id INT PRIMARY KEY);
CREATE SEQUENCE Car_seq
START WITH 1
INCREMENT BY 1;
CREATE TABLE Member(Member_id INT PRIMARY KEY);
CREATE SEQUENCE Member_seq
START WITH 1
INCREMENT BY 1;
CREATE TABLE Pod(Pod_id INT PRIMARY KEY);
CREATE SEQUENCE Pod_seq
START WITH 1
INCREMENT BY 1;
CREATE TABLE Account(Account_id INT PRIMARY KEY);
CREATE SEQUENCE Account_seq
START WITH 1
INCREMENT BY 1;
CREATE TABLE Plan(Plan_id INT PRIMARY KEY);
CREATE SEQUENCE Plan_seq
START WITH 1
INCREMENT BY 1;
CREATE TABLE CompanyAccount(CompanyAccount_id INT PRIMARY KEY);
CREATE SEQUENCE CompanyAccount_seq
START WITH 1
INCREMENT BY 1;
CREATE TABLE PrivateAccount(PrivateAccount_id INT PRIMARY KEY);
```

```
CREATE SEQUENCE PrivateAccount_seq
START WITH 1
INCREMENT BY 1;
CREATE TABLE Booking(Booking_id INT PRIMARY KEY);
CREATE SEQUENCE Booking_seq
START WITH 1
INCREMENT BY 1;
CREATE TABLE Trip(Trip_id INT PRIMARY KEY);
CREATE SEQUENCE Trip_seq
START WITH 1
INCREMENT BY 1;
CREATE TABLE Payment(Payment_id INT PRIMARY KEY);
CREATE SEQUENCE Payment_seq
START WITH 1
INCREMENT BY 1;
```

Adding Attributes

```
ALTER TABLE Car ADD car_no VARCHAR2(255) NOT NULL UNIQUE;
ALTER TABLE Car ADD car_nickname VARCHAR2(255) NOT NULL UNIQUE;
ALTER TABLE Car ADD car_year VARCHAR2(255) NOT NULL;
ALTER TABLE Car ADD car_trans VARCHAR2(255) NOT NULL;
ALTER TABLE Member ADD member_name VARCHAR2(255) NOT NULL;
ALTER TABLE Member ADD member_password VARCHAR2(255) NOT NULL;
ALTER TABLE Member ADD dob DATE NOT NULL;
ALTER TABLE Member ADD member_license VARCHAR2(255) NOT NULL UNIQUE;
ALTER TABLE Member ADD expiry_date DATE NOT NULL;
ALTER TABLE Pod ADD pod_name VARCHAR2(255) NOT NULL UNIQUE;
ALTER TABLE Pod ADD pod_longitude DECIMAL NOT NULL;
ALTER TABLE Pod ADD pod_latitude DECIMAL NOT NULL;
ALTER TABLE Account ADD account_no NUMBER NOT NULL UNIQUE;
ALTER TABLE Account ADD account_name VARCHAR2(255) NOT NULL;
ALTER TABLE Account ADD account_since DATE NOT NULL;
ALTER TABLE Plan ADD plan_name VARCHAR2(255) NOT NULL UNIQUE;
ALTER TABLE Plan ADD monthly_fee NUMBER NOT NULL;
ALTER TABLE Plan ADD hour_rate NUMBER NOT NULL;
ALTER TABLE Plan ADD km_rate NUMBER NOT NULL;
ALTER TABLE Plan ADD late_fee NUMBER;
ALTER TABLE CompanyAccount ADD crn VARCHAR2(255) NOT NULL UNIQUE;
ALTER TABLE PrivateAccount ADD score NUMBER;
ALTER TABLE Booking ADD time_booked TIMESTAMP NOT NULL;
ALTER TABLE Booking ADD booking_start TIMESTAMP NOT NULL;
ALTER TABLE Booking ADD booking_end TIMESTAMP NOT NULL;
ALTER TABLE Trip ADD trip_start TIMESTAMP NOT NULL;
ALTER TABLE Trip ADD trip_end TIMESTAMP NOT NULL;
ALTER TABLE Trip ADD start_metre NUMBER NOT NULL;
ALTER TABLE Trip ADD end_metre NUMBER NOT NULL;
ALTER TABLE Payment ADD bank_account NUMBER NOT NULL;
```

```
ALTER TABLE Payment ADD bank_code NUMBER NOT NULL;
ALTER TABLE Payment ADD card_type VARCHAR2(255) NOT NULL;
ALTER TABLE Payment ADD card_holder VARCHAR2(255) NOT NULL;
ALTER TABLE Payment ADD card_number NUMBER NOT NULL;
ALTER TABLE Payment ADD card_expire DATE NOT NULL;
```

Adding constraints

```
ALTER TABLE Booking
        ADD CONSTRAINT time_constraint CHECK (booking_start < booking_end);
```

B.2 Translated events to stored procedures

```
CREATE OR REPLACE PROCEDURE addMember(v_member_name IN VARCHAR,
    v_member_password IN VARCHAR, v_dob IN DATE, v_member_license IN VARCHAR,
    v_expiry_date IN DATE, v_member_account IN INT, v_member_pod IN INT,
    v_account_plan IN INT)
IS
BEGIN
         INSERT INTO Member (Member_id, member_name, member_password, dob,
    member_license, expiry_date, member_account, member_pod, account_plan)
    VALUES (Member_seq.NEXTVAL,v_member_name, v_member_password, v_dob,
    v_member_license, v_expiry_date, v_member_account, v_member_pod,
    v_account_plan);
END;

CREATE OR REPLACE PROCEDURE addPod(v_pod_name IN VARCHAR, v_pod_longitude IN
    DECIMAL, v_pod_latitude IN DECIMAL)
IS
BEGIN
         INSERT INTO Pod (Pod_id, pod_name, pod_longitude, pod_latitude) VALUES
     (Pod_seq.NEXTVAL,v_pod_name, v_pod_longitude, v_pod_latitude);
END;

CREATE OR REPLACE PROCEDURE addPlan(v_plan_name IN VARCHAR, v_monthly_fee IN
    VARCHAR, v_hour_rate IN VARCHAR, v_km_rate IN VARCHAR, v_late_fee IN
    VARCHAR)
IS
BEGIN
         INSERT INTO Plan (Plan_id, plan_name, monthly_fee, hour_rate, km_rate,
     late_fee) VALUES (Plan_seq.NEXTVAL,v_plan_name, v_monthly_fee, v_hour_rate
    , v_km_rate, v_late_fee);
END;

CREATE OR REPLACE PROCEDURE addCompanyAccount(v_crn IN VARCHAR)
IS
```

```
BEGIN
        INSERT INTO CompanyAccount (CompanyAccount_id, crn) VALUES (
    CompanyAccount_seq.NEXTVAL,v_crn);
END;

CREATE OR REPLACE PROCEDURE addPrivateAccount(v_score IN VARCHAR)
IS
BEGIN
        INSERT INTO PrivateAccount (PrivateAccount_id, score) VALUES (
    PrivateAccount_seq.NEXTVAL,v_score);
END;

CREATE OR REPLACE PROCEDURE addBooking(v_time_booked IN VARCHAR,
    v_booking_start IN VARCHAR, v_booking_end IN VARCHAR, v_booking_car IN INT,
     v_booking_member IN INT)
IS
BEGIN
        INSERT INTO Booking (Booking_id, time_booked, booking_start,
    booking_end, booking_car, booking_member) VALUES (Booking_seq.NEXTVAL,
    v_time_booked, v_booking_start, v_booking_end, v_booking_car,
    v_booking_member);
END;

CREATE OR REPLACE PROCEDURE addTrip(v_trip_start IN VARCHAR, v_trip_end IN
    VARCHAR, v_start_metre IN VARCHAR, v_end_metre IN VARCHAR, v_trip_member IN
     INT, v_trip_car IN INT)
IS
BEGIN
        INSERT INTO Trip (Trip_id, trip_start, trip_end, start_metre,
    end_metre, trip_member, trip_car) VALUES (Trip_seq.NEXTVAL,v_trip_start,
    v_trip_end, v_start_metre, v_end_metre, v_trip_member, v_trip_car);
END;

CREATE OR REPLACE PROCEDURE addPayment(v_bank_account IN VARCHAR, v_bank_code
    IN VARCHAR, v_card_type IN VARCHAR, v_card_holder IN VARCHAR, v_card_number
     IN VARCHAR, v_card_expire IN DATE, v_account_payment IN INT)
IS
BEGIN
        INSERT INTO Payment (Payment_id, bank_account, bank_code, card_type,
    card_holder, card_number, card_expire, account_payment) VALUES (Payment_seq
    .NEXTVAL,v_bank_account, v_bank_codev_card_type, v_card_holder,
    v_card_number, v_card_expire, v_account_payment);
END;

CREATE OR REPLACE PROCEDURE removeBooking( v_Booking_id IN INT)
IS
BEGIN
```

```
DELETE FROM Booking WHERE Booking_id = v_Booking_id;
END;

CREATE OR REPLACE PROCEDURE removePayment( v_Payment_id IN INT)
IS
BEGIN
DELETE FROM Payment WHERE Payment_id = v_Payment_id;
END;
```

References

[1] Emf framework for Event-B. http://wiki.event-b.org/index.php/EMF_framework_for_Event-B, 2018. [Online; accessed 21-July-2018].

[2] Fatma Abdelhedi, Amal Ait Brahim, and Gilles Zurfluh. Formalizing the mapping of uml conceptual schemas to column-oriented databases. *International Journal of Data Warehousing and Mining (IJDWM)*, 14(3):44–68, 2018.

[3] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[4] Jean-Raymond Abrial and Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.

[5] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International journal on software tools for technology transfer*, 12(6):447–466, 2010.

[6] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28, 2007.

[7] DH Akehurst, B Bordbar, PJ Rodgers, and NTG Dalgliesh. Automatic normalisation via metamodelling. *Declarative Meta Programming to Support Software Development*, page 45, 2002.

[8] Ahmed Al-Brashdi. Translating Event-B to database application. Master's thesis, University of Southampton, 2015.

[9] Ahmed Al-Brashdi. Model-based database design. In *Formal Methods 2016 Doctoral Symposium*, pages 1–6, 2016.

[10] Ahmed Al-Brashdi, Michael Butler, and Abdolbaghi Rezazadeh. Incremental database design using UML-B and Event-B. In Proceedings Joint Workshop on *Handling IMPlicit and EXplicit knowledge in formal system development (IMPEX)* and *Formal and Model-Driven Techniques for Developing Trustworthy Systems (FM&MDD),* Xi'An, China, 16th November 2017, volume 271 of *Electronic*

Proceedings in Theoretical Computer Science, pages 34–47. Open Publishing Association, 2018.

[11] Ahmed Al-Brashdi, Michael Butler, and Abdolbaghi Rezazadeh. UB2DB: Rodin plug-in for automated database code generation. In *Rodin Workshop 2018*, pages 1–2, 2018.

[12] Ahmed Al-Brashdi, Michael Butler, Abdolbaghi Rezazadeh, and Colin Snook. Tool support for model-based database design with Event-B. In *FM&MDD Workshop at ICFEM 2016*, pages 1–7, 2016.

[13] V. S. Alagar and K. Periyasamy. *Vienna Development Method*, pages 405–459. Springer London, London, 2011.

[14] Christopher Alexander. *A pattern language: towns, buildings, construction*. Oxford university press, 1977.

[15] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1, 2008.

[16] Malcolm P Atkinson, Francois Bancilhon, David J DeWitt, Klaus R Dittrich, David Maier, and Stanley B Zdonik. The object-oriented database system manifesto. In *DOOD*, volume 89, pages 40–57, 1989.

[17] Ying Bai. *Practical database programming with Java*. John Wiley & Sons, 2011.

[18] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

[19] Roberto Souto Maior Barros. On the formal specification and derivation of relational database applications. *Electronic Notes in Theoretical Computer Science*, 14:3–29, 1998.

[20] Christian Bauer and Gavin King. *Hibernate in action*. Manning Greenwich CT, 2005.

[21] Amal AIT BRAHIM, Rabah TIGHILT FERHAT, and Gilles ZURFLUH. Extraction process of conceptual model from a document-oriented nosql database. In *2019 11th International Conference on Knowledge and Systems Engineering (KSE)*, pages 1–5. IEEE, 2019.

[22] Michael Butler. Decomposition structures for Event-B. In *International Conference on Integrated Formal Methods*, pages 20–38. Springer, 2009.

[23] Michael Butler. Mastering system analysis and design through abstraction and refinement. In *Engineering Dependable Software Systems*. IOS Press, 2013.

[24] Michael Butler and Stefan Hallerstede. The Rodin formal modelling tool. In *Electronic Workshop In Computing*, 2007.

[25] Néstor Catano and Tim Wahls. A case study on code generation of an ERP system from Event-B. In *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*, pages 183–188. IEEE, 2015.

[26] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[27] Anthony Cleve, Maxime Gobert, Loup Meurice, Jerome Maes, and Jens Weber. Understanding database schema evolution: A case study. *Science of Computer Programming*, 97:113–121, 2015.

[28] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[29] Edgar F Codd. Further normalization of the data base relational model. *Data base systems*, pages 33–64, 1972.

[30] Thomas M Connolly and Carolyn E Begg. *Database systems: a practical approach to design, implementation, and management*. Pearson Education, 2005.

[31] Chris J Date. *An introduction to database systems*. Addison-Wesley, 2004.

[32] Chris J Date and Hugh Darwen. *A Guide to the SQL Standard*, volume 3. Addison-Wesley New York, 1987.

[33] Jim Davies, Charles Crichton, Edward Crichton, David Neilson, and Ib Holm Sørensen. Formality, evolution, and model-driven software engineering. *Electronic Notes in Theoretical Computer Science*, 130:39–55, 2005.

[34] Jim Davies, Jeremy Gibbons, James Welch, and Edward Crichton. Model-driven engineering of information systems: 10 years and 1000 versions. *Science of Computer Programming*, 89:88–104, 2014.

[35] Jim Davies, David Milward, Chen-Wei Wang, and James Welch. Formal model-driven engineering of critical information systems. *Science of Computer Programming*, 103:88–113, 2015.

[36] Jim Davies, James Welch, Alessandra Cavarra, and Edward Crichton. On the generation of object databases using Booster. In *Engineering of Complex Computer Systems, 2006. ICECCS 2006. 11th IEEE International Conference on*, pages 10–pp. IEEE, 2006.

[37] Roberto SM de Barros. Deriving relational database programs from formal specifications. In *FME'94: Industrial Benefit of Formal Methods*, pages 703–723. Springer, 1994.

[38] Alfonso de la Vega, Diego García-Saiz, Carlos Blanco, Marta Zorrilla, and Pablo Sánchez. Mortadelo: A model-driven framework for nosql database design. In *International Conference on Model and Data Engineering*, pages 41–57. Springer, 2018.

[39] E Durr and Jan Van Katwijk. VDM++, a formal specification language for object-oriented designs. In *CompEuro'92.'Computer Systems and Software Engineering', Proceedings.*, pages 214–219. IEEE, 1992.

[40] David Edmond. Refining database systems. In *International Conference of Z Users*, pages 25–44. Springer, 1995.

[41] CearSy System Engineering. Atelier b. http://www.atelierb.eu/en/, 2017. [Online; accessed 22-February-2017].

[42] Neil Evans and Michael Butler. A proposal for records in event-b. In *International Symposium on Formal Methods*, pages 221–235. Springer, 2006.

[43] Asieh Salehi Fathabadi. *An Approach to Atomicity Decomposition in the Event-B Formal Method*. PhD thesis, University of Southampton, 2012.

[44] Eduardo B Fernandez, Rita C Summers, and Christopher Wood. *Database security and integrity*. Addison-Wesley Longman Publishing Co., Inc., 1981.

[45] Steven Feuerstein and Bill Pribyl. *Oracle Pl/SQL Programming*. O'Reilly Media, Inc., 2005.

[46] John Fitzgerald, Peter Gorm Larsen, Shin Sahara, et al. VDMTools: advances in support for formal modeling in VDM. *ACM Sigplan Notices*, 43(2):3, 2008.

[47] Hector Florez, Edwarth Garcia, and Deisy Muñoz. Automatic code generation system for transactional web applications. In *International Conference on Computational Science and Its Applications*, pages 436–451. Springer, 2019.

[48] The Eclipse Foundation. Eclipse. https://www.eclipse.org/, 2018. [Online; accessed 13-July-2018].

[49] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[50] Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1994.

[51] Hector Garcia-Molina. *Database systems: the complete book*. Pearson Education India, 2008.

[52] Martin Gogolla. *Extended Entity-Relationship Model: Fundamentals and Pragmatics*. Springer-Verlag New York, Inc., 1994.

[53] Ali Ameer Gondal. *Feature-oriented Reuse with Event-B & Rodin*. PhD thesis, University of Southampton, 2013.

[54] Paolo Guagliardo and Leonid Libkin. Correctness of sql queries on databases with nulls. *ACM SIGMOD Record*, 46(3):5–16, 2017.

[55] Paolo Guagliardo and Leonid Libkin. A formal semantics of sql queries, its validation, and applications. *Proc. VLDB Endow.*, 11(1):27–39, September 2017.

[56] Thomas Günther, Klaus-Dieter Schewe, and Ingrid Wetzel. On the derivation of executable database programs from formal specifications. In *International Symposium of Formal Methods Europe*, pages 351–366. Springer, 1993.

[57] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[58] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on NoSQL database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE, 2011.

[59] Ramzi A Haraty and Georges Stephan. Relational database design patterns. In *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*, pages 818–824. IEEE, 2013.

[60] Matthias Jarke. *Database application engineering with DAIDA*, volume 1. Springer, 2014.

[61] Cliff B Jones. *Systematic software development using VDM*, volume 2. Citeseer, 1990.

[62] William Kent. A simple guide to five normal forms in relational database theory. *Communications of the ACM*, 26(2):120–125, 1983.

[63] Saeed Khalafinejad and Seyed-Hassan Mirian-Hosseinabadi. Translation of Z specifications to executable code: Application to the database domain. *Information and Software Technology*, 55(6):1017–1044, 2013.

[64] Régine Laleau and Fiona Polack. Coming and going from uml to b: a proposal to support traceability in rigorous is development. In *International Conference of B and Z Users*, pages 517–534. Springer, 2002.

[65] Michael Leuschel and Michael Butler. ProB: A model checker for B. In *FME 2003: Formal Methods*, pages 855–874. Springer, 2003.

[66] Petra Malik and Mark Utting. CZT: A framework for Z tools. In *International Conference of B and Z Users*, pages 65–84. Springer, 2005.

[67] Amel Mammar and Régine Laleau. Design of an automatic prover dedicated to the refinement of database applications. In *FME 2003: Formal Methods*, pages 834–854. Springer, 2003.

[68] Amel Mammar and Régine Laleau. From a B formal specification to an executable code: application to the relational database domain. *Information and Software Technology*, 48(4):253–279, 2006.

[69] Amel Mammar and Régine Laleau. UB2SQL: a tool for building database applications using UML and B formal method. *Journal of Database Management*, 17(4):70, 2006.

[70] Silvio Meira, Regina Motz, and Fernando Tepedino. A formal semantics for SQL. *International Journal of Computer Mathematics*, 34(1-2):43–63, 1990.

[71] Loup Meurice, Csaba Nagy, and Anthony Cleve. Detecting and preventing program inconsistencies under database schema evolution. In *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on*, pages 262–273. IEEE, 2016.

[72] Andreas Müller. VDM - the vienna development method. *Bachelor thesis in Formal Methods in Software Engineering, Johannes Kepler University Linz*, 2009.

[73] Iman Musleh, Samer Zain, Mamoun Nawahdah, and Norsaremah Salleh. Automatic generation of android sqlite database components. In *SoMeT*, pages 3–16, 2018.

[74] Mauro Negri, Giuseppe Pelagatti, and Licia Sbattella. Formal semantics of SQL queries. *ACM Transactions on Database Systems (TODS)*, 16(3):513–534, 1991.

[75] Oracle. Advantages of stored procedures. https://docs.oracle.com/cd/F49540_01/DOC/java.815/a64686/01_intr3.htm, 1999. [Online; accessed 08-March-2016].

[76] Oracle. Database concepts. https://docs.oracle.com/database/121/CNCPT/transact.htm, 2016. [Online; accessed 14-October-2016].

[77] Oracle. Database SQL reference. https://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_10001.htm, 2016. [Online; accessed 07-November-2016].

[78] OWASP. SQL injection prevention cheat sheet. https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet, 2016. [Online; accessed 08-March-2016].

[79] Richard E Pattis. Ebnf: A notation to describe syntax, 2013.

[80] Fiona Polack and Susan Stepney. Systems development using z generics. In *International Symposium on Formal Methods*, pages 1048–1067. Springer, 1999.

[81] Community Z Tools Project. CZT: Community Z tools. http://czt.sourceforge.net/, 2016. [Online; accessed 24-April-2017].

[82] RedHat. Hibernate. http://hibernate.org/, 2017. [Online; accessed 23-February-2017].

[83] Sonja Ristić, Slavica Aleksić, Milan Čeliković, and Ivan Luković. An EMF Ecore based relational DB schema meta-model. In *Proceedings of the 6th International Conference on Information Technology ICIT*, 2013.

[84] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004.

[85] Mark Saaltink et al. The Z/EVES 2.0 user's guide. *Ora Canada*, 1999.

[86] Maryah Said, Michael Butler, and Colin Snook. Language and tool support for class and state machine refinement in UML-B. In *International Symposium on Formal Methods*, pages 579–595. Springer, 2009.

[87] Klaus-Dieter Schewe, Joachim W Schmidt, and Ingrid Wetzel. Specification and refinement in an integrated database application environment. In *International Symposium of VDM Europe*, pages 496–510. Springer, 1991.

[88] Rudi Schlatte and Bernhard K Aichernig. Database development of a work-flow planning and tracking system using VDM-SL. In *Workshop Materials: VDM in Practice*, 1999.

[89] Bran Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19–25, 2003.

[90] Graeme Smith. *The Object-Z specification language*, volume 1. Springer Science & Business Media, 2012.

[91] Colin Snook and Michael Butler. UML-B and Event-B: An integration of languages and tools. In *Proceedings of the IASTED International Conference on Software Engineering*, SE '08, pages 336–341, Anaheim, CA, USA, 2008. ACTA Press.

[92] Colin Snook, Fabian Fritz, and Alexei Illisaov. An EMF framework for Event-B. In *Workshop on Tool Building in Formal Methods - ABZ Conference, Orford, Canada*, 2010.

[93] J Michael Spivey and JR Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

[94] Eugenia Stathopoulou and Panos Vassiliadis. Design patterns for relational databases. *Citeseer*, 2009.

[95] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[96] Christof Strauch, Ultra-Large Scale Sites, and Walter Kriha. NoSQL databases. *Lecture Notes, Stuttgart Media University*, 2011.

[97] DC Tsichritzis and Frederick H. Lochovsky. Hierarchical data-base management: A survey. *ACM Computing Surveys (CSUR)*, 8(1):105–123, 1976.

[98] OSCAR Canada users society. Oscar. http://oscarcanada.org/, 2018. [Online; accessed 31-July-2018].

[99] Sander Vermolen. *Automatically Discharging VDM Proof Obligations using HOL*. PhD thesis, Radboud University Nijmegen, 2007.

[100] Nicola Vitacolonna. Conceptual design patterns for relational databases. In *Proceedings of the 17th International Conference on Information and Software Technologies, Kaunas, Lithuania*, pages 239–246, 2011.

[101] Tim Wahls. Formal semantics and soundness of a translation from Event-B actions to SQL statements. *arXiv preprint arXiv:1606.02669*, 2016.

[102] Chen-Wei Wang and Jim Davies. Formal model-driven engineering: generating data and behavioural components. In *FTSCS*, pages 100–117, 2012.

[103] Qi Wang and Tim Wahls. Translating Event-B machines to database applications. In *Software Engineering and Formal Methods*, pages 265–270. Springer, 2014.

[104] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.