Incremental Database Design using UML-B and Event-B

Ahmed Al-Brashdi
University of Southampton
Southampton, UK
azab1g14@ecs.soton.ac.uk

Michael Butler
University of Southampton
Southampton, UK
mjb@ecs.soton.ac.uk

Abdolbaghi Rezazadeh
University of Southampton
Southampton, UK
ra3@ecs.soton.ac.uk

Correct operation of many critical systems is dependent on the data consistency and integrity properties of underlying databases. Therefore, a verifiable and rigorous database design process is highly desirable. This research aims to investigate and deliver a comprehensive and practical approach for modelling databases in formal methods through layered refinements. The methodology is being 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.

1 Introduction

Database systems hold large resources upon which critical decisions rely. These resources and decisions can be part of safety or business critical domains like health and patient systems or enterprise intelligent systems. This emphasises the fact that database systems are a very important field in software engineering [11] and thus require a verifiable and rigorous design and implementation. While the chances of inconsistency and ambiguity of specifications are low in small and simple systems, the chances increase as the database size and complexity grow. 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 database.

This research tries to address the question of how to gradually design databases and prove their consistency and integrity. To address this, we propose a method for model-based database design in formal methods using a UML-like notation that supports layered refinement of system models. We use Event-B which is a formal method for rigorous specification and verification of digital systems [1]. It has been supported in an open tool platform called Rodin [3]. We model our system using a UML-like notation in the Rodin tool called UML-B [18] which supports modelling in class diagrams and state machine diagrams. The UML-B tool translates UML-B models to Event-B models and the Rodin tool is used to verify their consistency. As class diagrams are commonly used to model database systems, using UML-B class diagrams will be more straightforward for database designers than modelling databases directly in Event-B.

This paper is structured as follows: In Section 2, we give background about the topics that are related to this research, mainly formal methods and relational databases. Section 3 describes how to model an information system using UML-B and Event-B in Rodin through abstraction and refinement following case studies. In Section 4, we outline our tool, UB2DB, which automatically generates database code from UML-B and Event-B model. Before concluding in section 6, we outline the most related work in Section 5.

R. Laleau, D. Méry, S. Nakajima, E. Troubitsyna (Eds): Joint Workshop on Handling IMPlicit and EXplicit knowledge in formal system development and Formal & Model-Driven Techniques for Developing Trustworthy Systems.
EPTCS 271, 2018, pp. 34–47, doi:10.4204/EPTCS.271.3

2 Background

2.1 Event-B

Event-B is a formal method for system modelling and verification. A model in Event-B consists of a static part in a *context* that defines the types, constants and axioms, and a dynamic *machine* component with all *variables* and *invariants* as well as *events* that change the variable state. An event may have *guards* that must hold before the execution of the event. Event *actions* change the state of a variable. All events in a machine must preserve all of its invariants.

Refinement in Event-B enables a modeller to gradually specify the system at different levels [1]. 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 [4]. 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 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 [1]. A new event in a refinement refines a *skip* event that does nothing to the abstract model. 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 *glueing invariant*. The glueing invariant links between an abstract state and a concrete one [1].

Compared to other formal notations and methods such as Z [19], VDM [14] and B method [2], Event-B provides more flexible refinements in which we can introduce new events in refinements. This feature is important in our research as databases include different operations on their data and we need to introduce these operations on data when their variables are introduced in refinements.

2.2 UML-B

UML-B is a graphical notation for formal modelling in Event-B that is based on UML [18]. 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 data structure and behavioural part of the system [18], and state diagram which are attached to a class, partitions the class instances into different states.

UML-B allows the modeller to choose one of three kinds when adding an event to a class. These kinds are *normal*, *constructor* or *destructor* events. 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.

A class diagram in UML-B 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. A state machine can be refined by adding new states ans transitions. A refinement of a state machine can also include adding nested states inside another state.

2.3 Relational Database

In 1970, Codd [10] 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* in which they may be formed by one or many *columns* where each column is 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.

Any committed state of a database must guarantee and preserve some pre-defined constraints and assertions. These constraints might relate to a table, an attribute in a table, or a relation between one table and another. It is important that database consistency is proved so that any requirement of the database system is not violated in any valid state of the system. It is also important to prove that database requirements don't contradict with each others. This can achieved by using formal methods in which invariants are used to model these requirements. The invariants are universally quantified in the formal model and preserved by all its operations.

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.

Modelling databases gradually using abstraction and refinement in Event-B should help the modeller breaks down the complexity of a system. Since the refinement proofs are generated when using refinement in Event-B, the modeller can make sure that the refinement does not violate the abstract model.

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

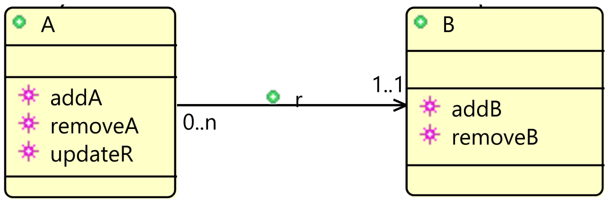


Figure 1: Abstract model of UML-B class diagram

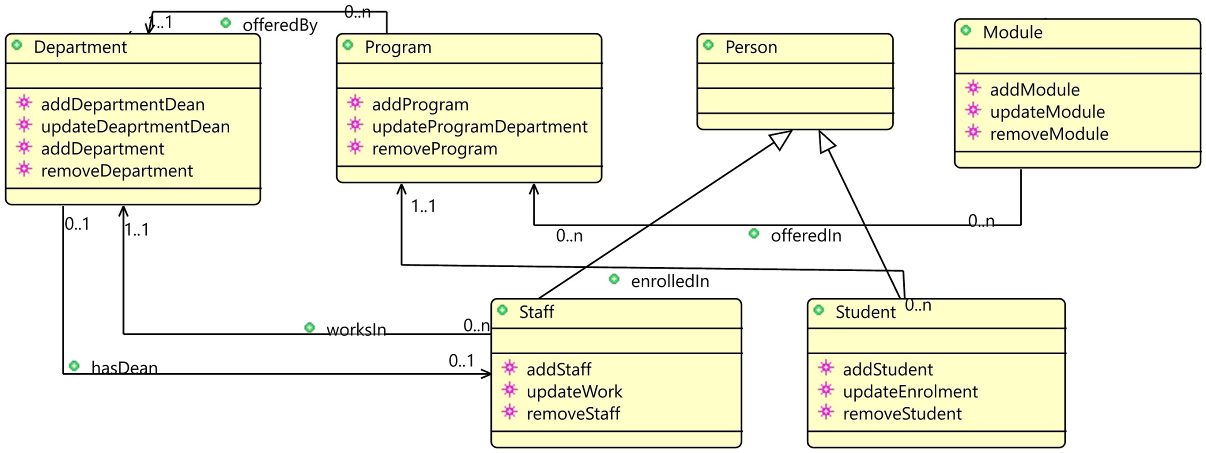


Figure 2: Abstract model of SRES entities and relations as a UML-B class diagram

street number and postcode. For each concept and refinement, this section will show it using an example of one of the case studies.

3.2 Modelling classes and their associations

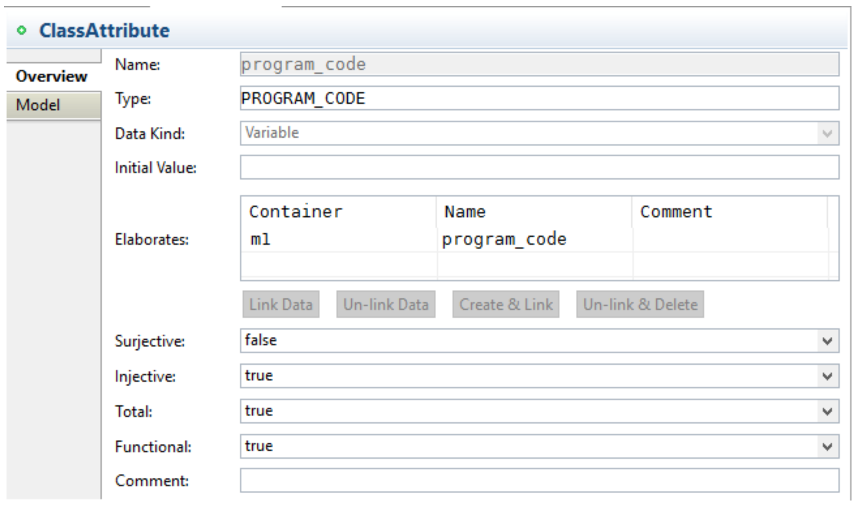
Modelling information systems in our approach is done in different refinements that are defined in successive steps. The abstract model of the system may consist of different classes and the associations between them. Figure 1 shows an abstract UML-B model of two classes *A* and *B*. The diagram shows 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 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*.

Event *addProgram* $\widehat{=}$

any



Name:	program_code
Type:	PROGRAM_CODE
Data Kind:	Variable
Initial Value:	
Elaborates:	
Surjective:	false
Injective:	true
Total:	true
Functional:	true
Comment:	

Container	Name	Comment
m1	program_code	

Figure 3: Setting class attribute in UML-B

$this\_Program, \; d$

where

grd1 : $this\_Program \notin Program$

grd2 : $d \in Department, \; this\_Program \in PROGRAM$

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.

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 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.

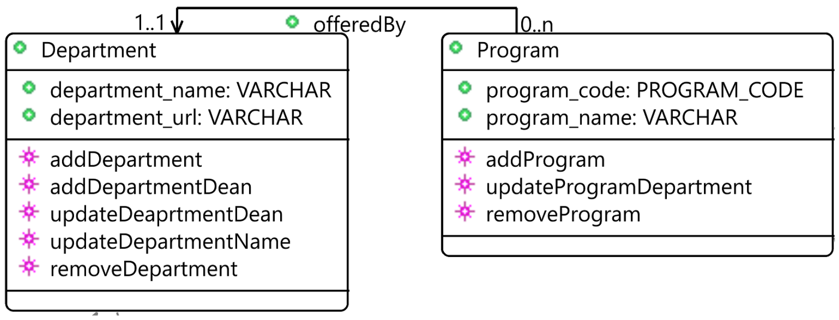


Figure 4: Adding attributes to the main classes

Figure 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:

```
Event  addProgram ≙
extends  addProgram
 any
          $p\_code,\ p\_name$
 where
          grd4: $p\_code \in PROGRAM\_CODE,\ p\_code \in PROGRAM\_CODE$
 then
          act3: $program\_code := program\_code \cup \{this\_Program \mapsto p\_code\}$
          act4: $program\_name := program\_name \cup \{this\_Program \mapsto p\_name\}$
 end
```

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 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.

3.5 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 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.

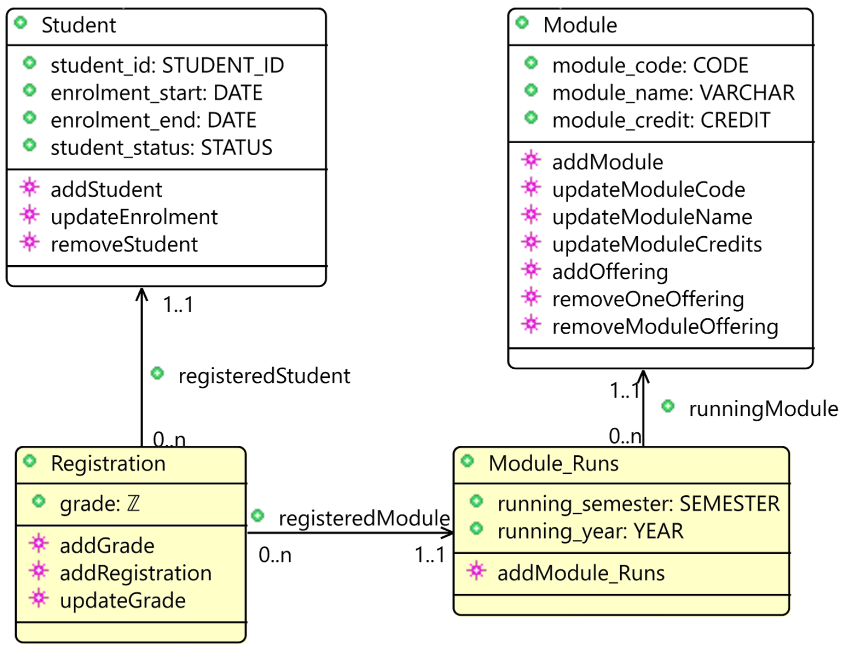


Figure 5: Adding secondary classes

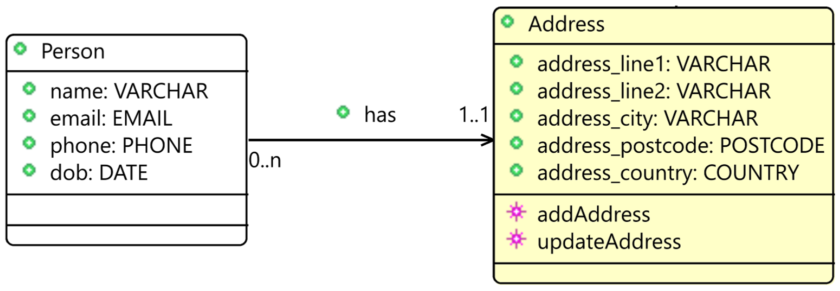


Figure 6: An example of an outer entity

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 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

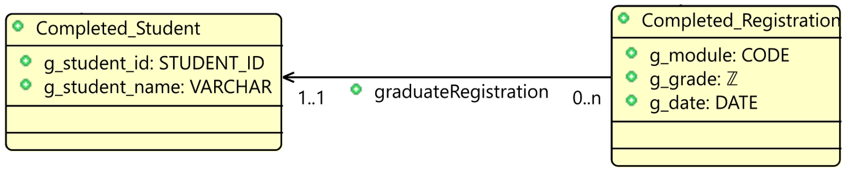


Figure 7: Historical data classes

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

3.7 Association splitting

While 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 8 and 9 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: $\forall a,b,c1,c2 \cdot c1 \mapsto a \in R1 \land c2 \mapsto a \in R1 \land$
$c1 \mapsto b \in R2 \land c2 \mapsto b \in R2 \Rightarrow c1 = c2)$

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.

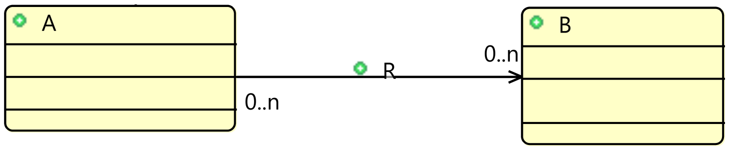


Figure 8: The abstract model of a relation R

3.8 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. The guards in Event-B events must hold for an event to be enabled and must satisfy the model invariants.

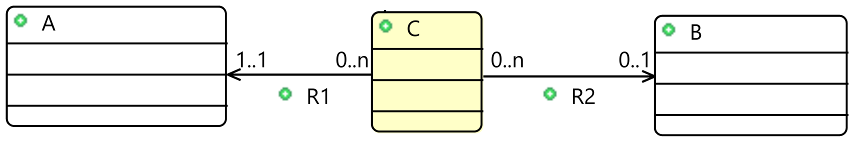


Figure 9: The refinement of relation R to R1 and R2

For a constructor event in UML-B, an instance of the class is created along with its attributes and associations such as *addprogram*. 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 $this\_a$.

```
Event removeA ≙
  any
      this_a
  where
      grd1 : $this\_a \in a$
  then
      act1 : $a := a \setminus \{this\_a\},\ x := \{this\_a\} \lhd x,\ r := \{this\_a\} \lhd r$
  end
```

Normal events in UML-B can be used to update or override set elements as in *updateA* which uses function override to update the 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*.

```
Event updateA ≙
  any
      this_a
      new_r
  where
      guard1 : $this\_a \in a$
      guard2 : $new\_r \in b$
  then
      action1 : $r := r \Leftarrow \{this\_a \mapsto new\_r\}$
  end
```

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

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*, reports, in *grd3*, 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.

Event *getDepartmentStaff* $\widehat{=}$
any
d
$staff\_list$
where
grd1 : $d \in Department$
grd3 : $staff\_list \in \mathbb{P}(Staff)$
grd3 : $staff\_list = worksIn^{-1}[\{d\}]$
then
$skip$
end

3.9 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.
- Introducing secondary classes, extending events and adding new events.
- Introducing attribute classes, extending events and adding new events.
- Introducing historical data, extending events and adding new events.
- Introducing query events.

The approach is extended further for extra features such as modelling database views. Two of our case studies with layered refinement can be found in [6].

3.10 Model verification

By modelling databases in UML-B and Rodin, we introduce formal verification for our database models. The database constraints are modelled as *invariants* in which they must be preserved by all events. Let's assume we have a requirement that Students can only register in Modules offered by the same program of study they are enrolled in as in Figure 10. This can be modelled by invariant *inv1* which applies to all instances of the registration class. For presentation and space, *inv1* is not shown in Figure 10 and is added directly to Event-B machine. The invariant becomes universally quantified in Event-B:

$$\text{inv1}: \forall m,s \cdot s \mapsto m \in registeredStudent^{-1}; registeredModule \Rightarrow runningModule(m) \mapsto enrolledIn(s) \in offeredIn$$

An event that adds a registration must preserve this invariant by having *grd2* that ensures the module to register the student for is offered in the student program of study:

Event *addRegistration* $\widehat{=}$

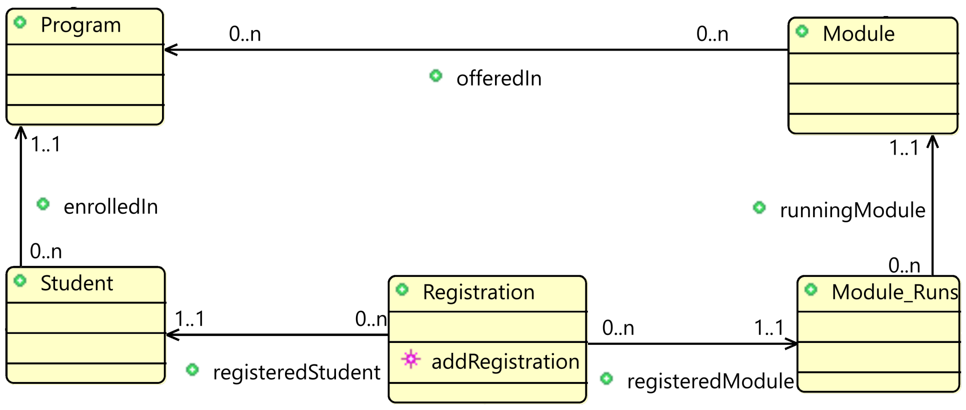


Figure 10: Student registration and enrollment

any

$this\_Registration,\ m,\ s$

where

`grd1`: $this\_Registration \notin Registration,\ m \in Module\_Runs,\ s \in Student$

`grd2`: $runningModule(m) \mapsto enrolledIn(s) \in offeredIn$

then

...

end

Without proving the consistency of this requirement among all operations on the database, a Students can register in a Module that is not offered by program of study in which they are enrolled in. The same verification applies to every invariant in the model in which it must be satisfied by every event in the model and any model that refines it. This draws an example for the importance of applying formal verification in the database design.

4 Tool support

We developed a tool, called UB2DB [7], as a plugin for the Roding platform which generates the SQL code for the database from the UML-B class diagram. The tool supports translating to different constraints in SQL such as primary key, foreign key, not null, unique and check constraints. The events in the model are translated to *stored procedures* in which event guards are validated and actions are executed in the stored procedure. A stored procedure is a program unit that is stored and validated in the database. The generated SQL code by the tool satisfies the system requirements and constraints and is validated against them for the case studies. We have generated the SRES database from the system and have been able to execute it successfully. Evaluating the performance of inserting 10000 records using the generated code shows that our code performed around 21% slower than a hand written code. Further evaluation and optimisation should improve the efficiency and performance of UB2DB.

5 Related Work

There is existing literature covering the concept of formalising database specifications. Schlatte and Aicherig present a database development of an industrial project using VDM-SL [17]. In [8] and [15], the authors formalise relational databases in Z specifications. Barros in [8] covers different CRUD operations as well as transactions, sorting, aggregations and other database components. There is no tool provided in which modellers can use to automatically generates database code for the formal definitions. Davies et al. in [13] shows how to formalise an object-oriented databases using UML and Object Constraint Language (OCL) [21] using Booster notation [12]. Mammar and Laleau in [16] have also specified relational database notions using UML-like notation. Their work supports modellers in designing databases using a UML diagram and then translate that model to a B specification and on to Java and SQL code. The refinement process supported by Laleau and Mammar work is toward a database implementation of B specifications. From Event-B, Wang and Wahls in [20] developed a Rodin plug-in that generates Java and JDBC code to create and query databases. However, the results shown issues with preserving database integrity from the code generated by their tool as in [5].

None of these research provides general guidelines for modelling relational database in formal methods. Moreover, they do not address layered refinement where in each refinement a modeller can introduce new classes, attributes, associations and operations. The approach of modelling database systems by gradual refinement steps is an important aspect and contribution of this research. While layered refinement is well used when modelling in Event-B as stated is [9] and [1], the contribution of this research is applying that to database design with distinction of different concepts such as *primary* and *secondary* classes and different events for different database operations.

6 Conclusion

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. 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 [16], it was a refinement for implementation where the concrete model becomes closer to an implementation language. Our research provides a practical approach for modelling the databases with different constraints through layered refinement. Throughout the process of modelling the case studies, we have identified the differences between different kinds of classes and events. These distinctions identify a refinement strategy or patterns for the model such as starting with classes and associations, then introduce attributes, then queries, etc. Undertaking the approach of specifying various components in different refinements enabled us to model each concept separately and verify its specifications. Our tool, UB2DB, which is developed to support our approach is validated against case studies and successfully generates the SQL and stored procedures code for database structure and operations from the case studies. While the presented approach for layered refinement are for guidance only, it is open for further extensions. Further extension patterns will be investigated using more and larger case studies. We will investigate different design patterns for modelling relational database which can be derived from different diverse case studies. The patterns will define how provide a solution for common problems when modelling information systems using UML-B and Event-B.

References

[1] Jean-Raymond Abrial (2010): *Modeling in Event-B: system and software engineering*. Cambridge University Press, doi:10.1017/CBO9781139195881.

[2] Jean-Raymond Abrial & Jean-Raymond Abrial (2005): *The B-book: assigning programs to meanings*. Cambridge University Press.

[3] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta & Laurent Voisin (2010): *Rodin: an open toolset for modelling and reasoning in Event-B*. International journal on software tools for technology transfer 12(6), pp. 447–466, doi:10.1007/s10009-010-0145-y.

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

[5] Ahmed Al-Brashdi (2015): *Translating Event-B to Database Application*. Master's thesis, University of Southampton.

[6] Ahmed Al-Brashdi (2017): *Case studies*. `http://users.ecs.soton.ac.uk/azab1g14/CaseStudies/`. [Online; accessed 24-August-2017].

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

[8] Roberto Souto Maior Barros (1998): *On the formal specification and derivation of relational database applications*. Electronic Notes in Theoretical Computer Science 14, pp. 3–29, doi:10.1016/S1571-0661(05)80226-9.

[9] Michael Butler (2013): *Mastering System Analysis and Design through Abstraction and Refinement*. In: Engineering Dependable Software Systems, IOS Press, pp. 49–78, doi:10.3233/978-1-61499-207-3-49.

[10] Edgar F Codd (1970): *A relational model of data for large shared data banks*. Communications of the ACM 13(6), pp. 377–387, doi:10.1145/362384.362685.

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

[12] Jim Davies, Charles Crichton, Edward Crichton, David Neilson & Ib Holm Sørensen (2005): *Formality, evolution, and model-driven software engineering*. Electronic Notes in Theoretical Computer Science 130, pp. 39–55, doi:10.1016/j.entcs.2005.03.004.

[13] Jim Davies, James Welch, Alessandra Cavarra & Edward Crichton (2006): *On the generation of object databases using Booster*. In: Engineering of Complex Computer Systems, 2006. ICECCS 2006. 11th IEEE International Conference on, IEEE, pp. 10–pp, doi:10.1109/ICECCS.2006.65.

[14] Cliff B Jones (1990): *Systematic software development using VDM*. 2, Citeseer.

[15] Saeed Khalafinejad & Seyed-Hassan Mirian-Hosseinabadi (2013): *Translation of Z specifications to executable code: Application to the database domain*. Information and Software Technology 55(6), pp. 1017–1044, doi:10.1016/j.infsof.2012.12.007.

[16] Amel Mammar & Régine Laleau (2006): *From a B formal specification to an executable code: application to the relational database domain*. Information and Software Technology 48(4), pp. 253–279, doi:10.1016/j.infsof.2005.05.002.

[17] Rudi Schlatte & Bernhard K Aichernig (1999): *Database development of a work-flow planning and tracking system using VDM-SL*. In: Workshop Materials: VDM in Practice, pp. 109–125.

[18] Colin Snook & Michael Butler (2008): *UML-B and Event-B: An Integration of Languages and Tools*. In: Proceedings of the IASTED International Conference on Software Engineering, SE '08, ACTA Press, Anaheim, CA, USA, pp. 336–341.

[19] J Michael Spivey & JR Abrial (1992): *The Z notation*. Prentice Hall Hemel Hempstead.

[20] Qi Wang & Tim Wahls (2014): *Translating Event-B machines to database applications*. In: Software Engineering and Formal Methods, Springer, pp. 265–270, doi:10.1007/978-3-319-10431-7_19.

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