Exercises for Teaching Logic in a Formal Methods Course: Formalizing ERDs

Guy Tremblay
Dépt. d'informatique — UQÀM
C.P. 8888, Succ. Centre-Ville
Montréal, Qué.
tremblay.guy@uqam.ca

Abstract

In a first course in formal methods for software engineers, the emphasis on the topic of logic should be on using (first-order predicate) logic to specify and formally describe properties. In this paper, we suggest using ERDs (Entity-Relationship Diagrams) as support for formalization exercises. Starting from graphical descriptions (ERDs) and textual informal specification of various constraints, students have to produce an equivalent textual and formal specification. We present the notation we use in our course (`Spec`), some heuristics to obtain the formal concepts from ERDs and a small example.

1 Introduction

Courses in formal methods are becoming more popular in undergraduate programs in computer science and software engineering. An important topic of such courses is logic, more precisely, propositional and (first-order) predicate logic.

The way in which logic is taught in a software engineering program should be different from how it is taught in a more theoretical computer science or mathematics program. For software engineers, the interest in logic is, first and foremost, to use logic to describe and express properties of software artifacts in a clear, succinct and non-ambiguous way; in other words, use logic for *specification*. A more advanced topic would be basic knowledge of rigorous reasoning and proofs, which might be useful for rigorous, systematic software development [6, 1]. On the other hand, topics such as as completeness, consistency, model theory, etc., which are usually the main emphasis of logic textbooks, are much less important for software engineers.

In a first formal methods course such as the one described in [7], which emphasizes the *description* aspect of formal methods, i.e., formal specification, little time is spent on formal proofs. On the other hand, a lot of time is devoted to the use of logic for expressing properties. In this paper, we present a class of problems that we use to introduce formal logic as a notation for expressing and formalizing various properties: the formalization of Entity-Relationship Diagrams (ERDs).

In section 2, we recall the basic notions of ERD modeling. Then, we briefly present (a subset of) the formal notation used in our course: the `Spec` notation [2]. In section 4, we present the general strategy used to translate ERDs into `Spec` concepts. This strategy, or the heuristics we present are informal rules ("*rules of thumb*") that can help produce a good first approximation of the formalization of an ERD. Of course, such heuristics are not algorithms, so some common sense is still required on the part of the specifier. Finally, an example is presented in section 5, followed by a brief assessment of our experience in section 6.

2 What are ERDs?

Entity-relationship diagrams (ERDs) are used to describe, using a graphical notation, the important concepts of an application domain, the notions useful to understand the basic entities of a problem and their relationships. Such diagrams may also allow the analyst to obtain an abstract conceptual model of the information to be stored in the information system being developed.

The basic concepts of ERD modeling are the notions of entity, attribute, relationship and associative object [4, 8]. [1]

Entity An entity represents a collection of objects or values, i.e., it denotes a type. Each entity has the following characteristics:

- It plays a key role in the problem being analyzed, and comes from the application domain of the system being developed.
- Each instance of the entity type is described by one or more attributes.
- The various instances can be distinguished from one another in a unique manner, using one or more attributes (often called the *primary key*).

[1] Subtyping can also be modeled with ERDs and in `Spec`. However, due to lack of space, we omit the presentation of this topic.

Attribute An attribute is a particular characteristic, or a piece of information that describes an entity; it is generally considered an *atomic* piece of information.

An entity is graphically represented by a rectangle: its name is indicated inside that box; the name of the attributes are indicated below or beside the box (see section 4.1).

Relationship A relationship is used to represent an association, or a link that can exist between two entities. In the same way that an entity describes a set of instances, a relationship describes a set of links, or associations: for each instance of the relationship, there exist specific instances of the appropriate entities which are related. A relationship is represented graphically by a diamond, with its name indicated below or above it (see section 4.2).

Associative object An associative object is an entity associated with a relationship, that is, an entity that describes, or gives information *about* a relationship. While an entity can exist without being in any relationship, an associative object can exist only when a relationship does exist between appropriate entities. Each associative object is related to some (direct) attributes that describe the particular instance of the relationship; furthermore, as each instance of an associative object requires that some specific entities be in a relationship, we also assume there are some (indirect) attributes that indicate the entities being related, i.e., those which give rise to the associative object. Graphically, an associative object is represented by an entity box with an arrow pointing from the box to the associated diamond/relationship (see section 4.3).

Derived attributes and relationships Derived attributes (or relationships) are attributes that can be derived, computed, inferred from existing entities and/or relationships. Such attributes are not included in an ERD. However, these can easily be formalized using appropriate concepts, as we will see in section 5.

3 What is `Spec`?

The notation chosen for introducing our students to formal specifications is neither `Z` nor `VDM`! Rather, we have chosen the `Spec` notation [2] for a number of reasons:

- Like `Z` and `VDM`, `Spec` uses an *abstract model* approach to software specification. In such an approach, the behavior of a software artifact, e.g., an object, is described by defining an abstract model for the state of this object and, then, showing how the various operations — the messages which the object can handle — transform this state, generally using pre-/post-conditions.
- Contrary to `Z`, `Spec` does not require special symbols or the use of special fonts or text editor/formatter; instead, specifications in `Spec` are written in plain ASCII text. This often makes it simpler to read, especially for novices — compare "`remove(x, m)`" with "$x \lhd m$". Furthermore, it also makes it possible to use the notation in program documentation and it allows students to write specifications using a plain text editor. Here are a few examples of (generalized) quantified expressions written in `Spec`:

```
ALL( x: nat SUCH THAT x ~= 0 ::  x > 0 )
SOME( x: nat SUCH THAT x ~= 0 ::  x > 0 )
SUM( x: nat SUCH THAT 1 <= x <= 100 ::  x )
```

- `Spec` has an interesting notion of `CONCEPT`. In the terminology of [5], we could say that `CONCEPT`s are used to introduce definitions and/or designations that define, clarify, make precise auxiliary concepts that help to understand the application domain or the properties of the system functionalities. Thus, `CONCEPT`s are not intended for defining system or software functionalities; these have to be specified with `MESSAGE`s. To formalize ERDs, it is `CONCEPT`s that are used. Here are some examples:

```
-- A type.
CONCEPT cat: type
-- An instance of the cat type.
CONCEPT garfield: cat
-- An attribute of a cat.
CONCEPT age(c: cat) VALUE(n: nat)
-- A relationship between cats.
CONCEPT larger(c1 c2: cat) VALUE(b: boolean)
```

  These concepts are all designations, that is, terms that name some phenomena without formally defining them. Such designations are like assumptions about the world, notions or concepts which cannot formally be explained or defined; they must be understood in order to understand the rest, like axioms in a formal system. Of course, designations can then be used to introduce formal definitions. For example, we could formally introduce the definitions below; in each case, the `WHERE` clause indicates the constraint that formally defines the concept:

```
CONCEPT older(c1 c2: cat) VALUE(b: boolean)
 WHERE b <=> age(c1) > age(c2)

CONCEPT smaller(c1 c2: cat) VALUE(b: boolean)
 WHERE b <=> larger(c2, c1)
```

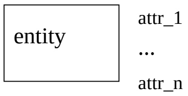
Other interesting features of `Spec`, not necessary for the present paper, are presented in [7]: explicit constructs for invariants, pre/post-conditions and state transitions; various module constructs; object model with asynchronous messages; exceptions; etc.

4 Heuristics for the formalization of ERDs

We now present some guidelines, or heuristics on how an ERD can be formalized using concepts and first-order predicate logic as found in `Spec`.

4.1 Entities and attributes

Suppose we have an entity which denotes something that *is not* an associative object:



This entity can be formalized by the following concepts:

```
-- Type defining the entity.
CONCEPT entity: type

-- Concepts (designations) for the various
-- attributes of the entity.
CONCEPT attr_1(e: entity) VALUE(a: type_1)
  ...
CONCEPT attr_n(e: entity) VALUE(a: type_n)
```

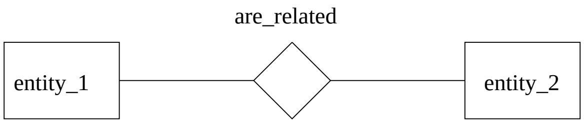
A few points to note:

- This entity must not be an associative object. This latter case will be handled later (section 4.3).
- There are various possibilities for the type of the attributes (`type_1`, ..., `type_n`):
  1. The type of the attribute is one of the base type, e.g., `nat`, `real`, etc. In this case, nothing further needs to be added.
  2. The type of the attribute is specified in another module, with its own associated attributes. In this case, this module can either be inherited or imported. If many attributes and relations need to be used and/or some of them must be refined, then it should be inherited. If only the name of the type is required, then it should be imported.
  3. The type of the attribute is not yet specified and is relevant only for the entity being specified. In this case, it is sufficient to introduce an informal designation, where the type name can be the same as the attribute name (overloading is allowed in `Spec`). For example, we could have the following concepts, where `student` is an entity type and `student_number` is an attribute of such an entity:

     ```
     CONCEPT student_number(s: student)
             VALUE(sn: student_number)
     ...
     CONCEPT student_number: type
     ```

4.2 Relationships

Suppose we have a simple relationship, that is, one which has no associative object related to it:



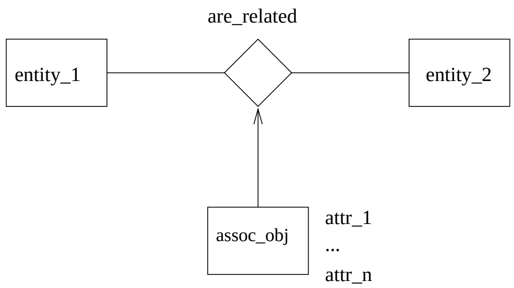
This relationship can be specified by the following informal designation:

```
CONCEPT are_related(e1: entity_1, e2: entity_2)
        VALUE(b: boolean)
  -- Comment giving an informal explanation
  -- of the relationship.
```

It is possible to associate various constraints with relationships, e.g., cardinality constraint or referential integrity constraint. Such constraints can sometimes (e.g., cardinality annotations), but not always (see section 5) be indicated directly on the ERD using annotations. Generally, they can be formalized in `Spec`, in which case the constraint has to be associated with one of the entity involved in the relationship. With which entity the constraint is associated will depend on the problem. An example is presented in section 5.

4.3 Associative objects

An associative object is an entity that *describes* a relationship and gives additional information about the relationship. An important characteristic of such an entity is that its existence depends on appropriate entities being in relationship. It is this aspect that we formalize here. Suppose we have the following ERD:



It can be formally specified with the following concepts:

```
-- Type defining the entity
CONCEPT assoc_obj: type
  WHERE
    -- Constraint for an associative object:
    -- It can exist only when appropriate
    -- entities are related.
    ALL( ao: assoc_obj ::
         SOME( e1: entity_1, e2: entity_2 ::
               are_related(e1, e2) &
               entity_1(ao) = e1   &
               entity_2(ao) = e2 ) )

-- The (direct) attributes of the entity.
CONCEPT attr_1(ao: assoc_obj)
        VALUE(a: type_1)
...
CONCEPT attr_n(ao: assoc_obj)
        VALUE(a: type_n)

-- The relationship.
CONCEPT are_related(e1: entity_1, e2: entity_2)
        VALUE(b: boolean)
```

```
-- The (indirect) attributes for obtaining the
-- related entities.
CONCEPT entity_1(ao: assoc_obj)
        VALUE(e1: entity_1)
  WHERE
    are_related(e1, entity_2(ao))
CONCEPT entity_2(ao: assoc_obj)
        VALUE(e2: entity_2)
  WHERE
    are_related(entity_1(ao), e2)
```

5 An example

Suppose we are given the ERD presented in figure 1 with the following constraints:

1. A student can take no more than 6 courses.
2. A course cannot be taken by more than 60 students.
3. A student can be taking a course which he/she has already taken only if the grade he/she has previously obtained was lower than `C`

The first two constraints could easily be indicated on the ERD by annotating it with cardinality constraints such as "`0:6`" or "`0:60`". However, the third constraint could not be expressed graphically, although it can be formalized using an appropriate concept. The `Spec` module formalizing this ERD is presented below:

```
DEFINITION courses
 IMPORT One_to_one
        FROM function{student, student_id}

 -- Student entity.
 CONCEPT student: type
   WHERE
    ALL(s: student :: nb_courses(s) <= 6),
    ALL(s: student ::
      ALL(c: course, r: result
           SUCH THAT
               is_taking(s, c) &
               student(r) = s  &
               name(course(r)) = name(c) ::
             grade(r) < #C))

 CONCEPT name(s: student)
         VALUE(n: name)
 CONCEPT student_id(s: student)
         VALUE(si: student_id)
  WHERE One_to_one(student_id)  -- Unique id.

 -- Course entity.
 CONCEPT course: type
  WHERE
   ALL(c: course :: nb_students(c) <= 60)
 CONCEPT name(c: course)
         VALUE(cn: course_name)
 CONCEPT section_number(c: course)
         VALUE(sn: section_number)

 -- Result entity.
 CONCEPT result: type
   WHERE
    ALL(r: result ::
      SOME(s: student, c: course ::
            has_taken(s, c) &
            student(r) = s &
            course(r) = c))
 CONCEPT term(r: result)
         VALUE(s: term)
 CONCEPT grade(r: result)
         VALUE(g: grade)
 CONCEPT student(r: result)
         VALUE(s: student)
           WHERE has_taken(s, course(r))
 CONCEPT course(r: result)
         VALUE(c: course)
           WHERE has_taken(student(r), c)

 -- Relationships.
 CONCEPT is_taking(s: student, c: course)
         VALUE(b: boolean)
 CONCEPT has_taken(s: student, c: course)
         VALUE(b: boolean)

 -- Derived attributes.
 CONCEPT nb_courses(s: student) VALUE(n: nat)
    WHERE
     n = NUMBER(c: course
                SUCH THAT is_taking(s, c) :: c)
  CONCEPT nb_students(c: course) VALUE(n: nat)
    WHERE
     n = NUMBER(s: student
                SUCH THAT is_taking(s, c) :: s)


 -- Basic types for the attributes.
 CONCEPT name: type
 CONCEPT student_id: type
 CONCEPT course_name: type
   -- Example: MT3143, CS1110, ...
 CONCEPT section_number: type
 CONCEPT term: type
 CONCEPT grade: type
  WHERE grade = enumeration{F, D, C, B, A}
END
```

Note the following interesting points:

- `One_to_one` is a library concept which, in this case, indicates that `student_id` is a *unique* identifier.
- A cardinality constraint on a relationship is generally associated, in a more natural manner, with one of the two entities. Clearly, the constraint on the maximum number of courses has to be associated with the `student` entity, while the other one has to be associated with the `course` entity.
- The third constraint has been associated with the `student` concept. It states (left column) that for any student `s`, and for any course `c` being taken by that student (`is_taking(s, c)`), then for any result obtained by that student (`student(r) = s`) for that course (`name(course(r)) = name(c)`), the

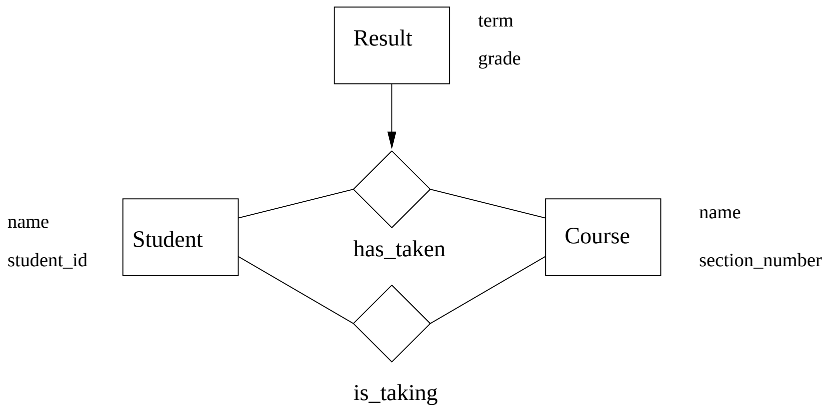


Figure 1: An ERD for a university registration system

grade associated with this result must necessarily be less than `C` (`grade(r) < #C`).

However, note that it could as well have been associated with the `course` entity: any of the students taking a course must obey the grade constraint.

6 Assessment

We have been teaching a formal methods course in our undergraduate program since January 1997. Such a course was introduced because of the students' dissatisfaction with a previously taught course in "Algebra and logic", course which was deemed too theoretical. In general, based on the informal students' comments and on the formal evaluation done by the *module* — at UQÀM, the *module* is the administrative entity responsible for managing a program of study — the formal methods course has been well received by the students.

As for the specific topic of ERD formalization, it is clear this is not a trivial topic for our students to grasp. The first reason is mainly that, at the time they take the formal methods course, most students have not *yet* been introduced to conceptual database modeling or to object-oriented analysis. Thus, the role of ERDs and the semantics associated with the diagrams is not familiar to them. We still think, however, that this topic is worth being presented because it introduces some important notions (type, attribute, instance, relationship, etc.) and allows for interesting modelization and formalization problems.

Finally, considering the time we spend on the topic of ERD formalization ($\approx 4\frac{1}{2}$ hours) and the fact that, for most students, this is a first encounter with the topic of conceptual modeling, not many students manage to understand well the notion of associative object. Thus, although we still spend some time explaining it, questions involving this notion are now considered as *bonus* marks.

7 Conclusion

We have presented a class of problems that can be used to introduce students to the use of logic for expressing formal properties. Our experience has been that interesting problems can be obtained, as ERDs (with or without associative objects) can be used to create abstract model for numerous set of concepts, from simple to complex ones. This makes it possible to use predicate logic in interesting ways, showing its expressiveness.

Finally, note that the ideas presented in this paper could also be applied to other graphical notations, for example, UML [3]. We are effectively considering such a transition for next year.

Acknowlegments

Thanks to Philippe Gabrini for his careful reading of a first draft of this paper. Thanks also to the anonymous referees for their useful comments.

References

[1] D. Andrews and D. Ince. *Practical formal methods with VDM.* McGraw-Hill, 1991.

[2] V. Berzins and Luqi. *Software Engineering with Abstractions.* Addison-Wesley Publishing Co., 1991.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* ACM Press, 1998.

[4] P. Chen. The entity-relationship model — toward a unified view of data. *ACM Trans. on Datab. Syst.*, 1(1):9–36, 1976.

[5] M. Jackson. *Software Requirements & Specifications — a lexicon of practice, principles and prejudices.* ACM Press & Addison-Wesley, 1995.

[6] C.B. Jones. *Systematic Software Development using VDM.* Prentice-Hall, 1986.

[7] G. Tremblay. An undergraduate course in formal methods: "Description is our business". In *Proceedings of the Twenty-ninth ACM SIGCSE Technical Symposium*, pages 166–170. ACM, 1998.

[8] E. Yourdon. *Modern Structured Analysis.* Yourdon Press, 1989.