An Undergraduate Course in Formal Methods: "Description is our business"

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

Abstract

We present a new undergraduate course in formal methods which emphasizes software specification and description instead of formal program correctness or program derivation.

This course is motivated by the idea that specifications play a key role in the software development process and that precise and clear descriptions of software systems and artifacts are important. It wants to show students that some simple notions from logic and mathematics (predicates and quantifiers, set theory, maps), together with appropriate constructs for defining software components and systems, can be useful for this purpose.

The course is taught using the `Spec` notation [1] and we first discuss why we did not choose the ubiquitous `Z` notation. The content of the course is then presented, followed with a brief description of an on-line WWW system that was developed to help the students do the exercices presented in the course notes.

1 Introduction

There is a growing recognition of the importance of formal methods, both in academy and in industry [2]. For a long time, especially in North America, formal methods have been equated with proofs of correctness or with formal program derivation, thus the numerous courses covering mainly this aspect [9, 3, 7, 10]. However, it is also being recognized that formal proofs are of less importance than *formal specifications* [11]. And, of course, to perform any proof, you first need the specifications.

The key role of formal specifications can be explained by a number of reasons:

- They can provide a clearer, more precise and less ambiguous basis on which to design the software.
- Using formal languages allows the use of tools for the manipulation, analysis, simulation, etc., of those specifications.
- Producing a formal specification can play an important role in validation — to determine if the appropriate requirements are satisfied — as it requires that vague ideas be clarified, that implicit concepts or properties be made explicit. As expressed by J. Wing [11], who calls it the "*formalization effort*":

> *Gaining a deeper understanding of the specificand by forcing yourself to be abstract yet precise about desired system properties can be more rewarding than having the specification document alone.*

This latter aspect is particularily important and is one of the major motivation for the course in formal methods ("MAT3143 Méthodes formelles") which has been introduced at UQAM (Université du Québec À Montréal). We describe this course in the following sections.

2 Why a New Course

For historical reasons, our program is entitled "*Informatique de gestion*" (Business-, MIS-style computer science). In recent years, however, it really has become a program with a software engineering orientation, where students have to take courses in programming (Ada, assembler, C/C++), networks, computer architecture, systems analysis, software design, project management and various business/management course. Furthermore, it is a coop program, so that students also have to do three (3) work terms in industry.

The mathematical background of our students is rather light, consisting in one course in each of the following: discrete mathematics, linear algebra and statistics. In addition, until last year, they also had to take a course in "*Algèbre et logique*" (algebra and logic), course which dealt with topics such as formal logic, coding theory, program correctness, etc. However, the appreciation of this course was not very good, as students found that it was too theoretical, lacking software engineering content. Thus, about a year ago, when we revised the program, we decided to replace it with a course with some mathematics, but with a more practical, software engineering orientation; this gave rise to a new course entitled "*Méthodes formelles*" (formal methods).

The goal of this new course *is not* to study logic or mathematics (e.g., set theory) for itself nor to use logic for proving the correctness of programs. Rather, it aims at having students *use* logic and basic mathematical concepts (combined with an appropriate notation) to *describe* software artifacts and properties. In other words, the aim of the course is essentially on the specifications part of formal methods, viewing formal specification as a means through which software systems, modules, components, etc. can be described. Thus, we want to instill the idea that, as once put by M. Jackson [5], "Description is our business".

3 Choice of Notation

The notation chosen for introducing our students to formal specifications is neither `Z` nor `VDM`! Rather, we have chosen the `Spec` notation [1] 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. This is generally a natural, constructive approach to specification.

  Although some people claim that an abstract model approach leads to specifications with an *implementation bias*, this can generally be avoided [8]; in fact, this is one of the topic discussed in the course.

- Contrary to `Z`, `Spec` does not require special symbols — does not look like a *cabalistic* notation :-) — or the use of special fonts or text editor/formatter; instead, specifications in `Spec` are written in plain ASCII. This often makes it simpler to read, especially for novices — compare "`remove(x, m)`" with "$x \lhd m$". Also, it makes it possible to use the formal notation directly within program documentation and allows students to write specifications using a simple text editor.

- In `Z`, almost every entity is defined through schemas, i.e., schemas are the sole means of modularisation. This may make it difficult to distinguish schemas used for defining abstract machines vs. abstract types vs. object classes. In `Spec`, on the other hand, various module constructs are provided to distinguish these different kinds of software artifacts: `FUNCTION`, `MACHINE` and `TYPE`; the latter kind can be used to define either mutable types (i.e., classes of objects) or immutable types (i.e., sets of abstract values).

  This aspect seems particularily important and, often, is not stressed enough in early programming courses. For example, in `Smalltalk` everything is an object, in `Ada` everything is defined using packages. But not all kinds of entity are naturally modeled by objects, and packages can be used to implement both machines, abstract data types or classes of objects. Using the various `Spec` modules, these different kinds of entity can then be presented and discussed in a more explicit manner.

- `Spec` has an interesting notion of `CONCEPT`. In the terminology of [6], we could say that `CONCEPT`s are used to introduce definitions and/or designations, but not to define system functionalities, these being specified with `MESSAGE`s. Thus, `CONCEPT`s are used to define, clarify, make precise auxiliary concepts helpful to understand the application domain or the properties of the functionalities being specified.

- In `Spec` (and in `VDM`) important properties such as invariants, pre- vs. post-conditions, state transitions have explicit constructs to identify them. In `Z`, on the other hand, these properties are left implicit and can only be identified by examining the details of the conditions associated with the schema. We believe the former, explicit approach should make it simpler for students to understand these various concepts. Of course, this means having to learn some additional notation (keywords).

- Our students, before taking this course, needs to complete two programming courses in `Ada`. A number of features from `Spec` have been inspired by a desire to facilitate the mapping to `Ada` programs: modules, generic, exceptions, etc. This makes it possible to illustrate some of the important notions — e.g., abstract machine — by showing the correspondence between an abstract `Spec` specification and some appropriate `Ada` package implementing the specification, thus rendering the concepts more concrete to students.

  Note also that although the `Spec` notation predates the object-oriented `Ada-95`, it does include a form of inheritance.

4 Course Description

The contents of the course is the following:

- Introduction: This section recalls the basic notions of software lifecycle, the place and role of analysis and specification vs. design. It also briefly presents what are formal methods, why formal specifications are important, what are the claimed advantages of using formal methods, examples of industrial applications, etc.

- The Spec notation (part 1): Propositions, quantifiers and CONCEPTs.

  The basic notions of predicate logic are recalled (from the discrete mathematics course) and presented using the Spec notation. Various forms of generalized predicates are also introduced, e.g., SUM, MININUM, NUMBER, thus showing that the ubiquitous universal and existential quantifiers — ALL and SOME, i.e., $\forall$ and $\exists$ — can easily be generalized without any special notation or symbol.

  The notions of CONCEPT and DEFINITION module are also introduced: such modules can be used to group related concepts, which is particularily useful to describe part of the application domain [6].

  Finally, examples of Entity-Relationships Diagrams (ERD) are presented and heuristics are given on how they can be formalized using appropriate CONCEPTs. This makes it possible to show that the formal version can represent complex constraints which cannot be described by the basic graphical formalism, e.g., constraints involving multiples entities and/or relations.

- The Spec notation (part 2): Abstract data types.

  Abstract modeling approaches are based on the use of a number of basic mathematical types — boolean, integer, sets, sequences, maps, etc. — which are presented in this section. Sets are *supposed* to be known from a previous course in discrete mathematics; sequences and, especially, maps are new for most students.

- The Spec notation (part 3): Functions, machines and types.

  This part forms the core of the course. The various kinds of modules used for defining software artifacts are presented through a number of examples:

  - FUNCTIONs = Modules with no notion of state: Numerical calculation (generic square root and polynomial roots), sorting functions (with vs. without duplicate).
  - MACHINEs = Modules defining abstract state machines: Stack, control system for vending machine and microwave oven, systems for managing loans in a video store and for dealing with bank accounts. A more complex example of a system for managing a small university library is also presented; here, a detailed informal, narrative description is first given, and then a formal specification is developed in a stepwise manner.
  - Mutable TYPEs = Modules that define classes of objects: Accumulators, stacks, various types of queues.
  - Immutable TYPEs = Modules that define abstract data types: Intervals, stacks, strings, rational numbers.

  In a number of cases, Ada definition and, sometimes, implementation packages corresponding to the abstract Spec module are presented. This seems particularily helpful in the cases where the same notion (e.g., stack) is modelized using multiple kinds of modules.

- Algebraic specifications.

  Earlier in the course, (part 2 of the Spec notation), the basic type constructors used for abstract modeling (sets, sequences and maps) are introduced informally, through simple examples. Here, the algebraic approach to the specification of abstract data types is introduced — using the Larch notation [4] — and formal definitions of these abstract data types are now presented.

- A glimpse of a rigorous method of development.

  In this part, the various *proof obligations* associated with a formal specification are presented [8], but in a rather informal and intuitive manner. The emphasis is put mainly on the proof obligations related directly with a specification — checking that it is satisfiable and the invariant is preserved — and on the notion of implementation bias. Thus, program correctness *per se* (in the programming-in-the-small sense) is not discussed.

- Regular languages, finite automata and transition diagrams.

  The notion of formal language is introduced, together with the notions of regular languages/expressions. Then, the various types of finite automata are presented (e.g., Moore vs. Mealy machines, deterministic vs. non-deterministic). Formal definition of some machines are presented in the usual mathematical style (e.g., a machine is a five-tuple) together with equivalent definitions using the Spec MACHINE construct. This makes it possible to show the additional expressive power of Spec machines, which can define not only finite automata but also extended state machines. On the practical side, a small example illustrating the use of lex is also presented.

  Note that this topic was introduced in the course because our students do not have to take any course in formal languages and automata theory; yet, we wanted them to have a minimal knowledge of these notions.

5 Course Material

Because no existing book was deemed appropriate for the course we intended to develop — they either all concentrate on a single approach or notation, e.g., Z, or are mainly collection of research papers — course notes (*en français*) were developed.

Although a number of exercices were included at the end of each chapter, solutions were not included. Instead, they have been made available on the `Web`. [1] However, in order to encourage the students to do the exercices — "practice makes perfect" — and not only read through the solutions, various *filters* have been provided. For example, before a student can get access to a solution, he/she must answer a multiple choices question related to the initial question/problem. In other cases, the student must first provide its own solution. A simple analysis of this solution is then performed — certain key identifiers are searched for — and, if it seems minimally correct, the solution is made accessible; otherwise, a message is displayed — an indication of what seems to be missing with some brief explanation — and no access is given to the solution.

Also, for some of the more complex problems, hints and/or partial solutions have been provided. Thus, students which are unsure about how to approach a problem can have a look at the hint, enabling them to proceed further.

6 Assessment

At the time of final writing, the course has been taught once and is currently being taught for a second time. The feedback from the students after the first term, especially when compared with the previous course in algebra and logic, was positive, even though they find the material quite difficult.

We think the perceived difficulty stems from the following reasons:

- The `Spec` notation: Learning the `Spec` notation is not easy, as the language involves a large number of concepts and constructs. However, considering their previous acquaintance with `Ada` and considering the fact that `Spec` makes explicit a number of aspects often left implicit in other notations — e.g., `CONCEPT`, abstract machine vs. type vs. object, exception, etc. — we think it is still the best choice for our students.

- Difficulty with *non-operational* thinking: When students take the formal methods course, most have already taken at least three (3) programming courses (two in `Ada`, one in assembly programming) and some have also taken a "*Paradigmes de programmation*" (programming paradigms) course, where they are introduced to `SML`, `Smalltalk`, and `Prolog`. In other words, they have mainly been trained in *operational thinking*. But writing formal specifications, even using the abstract model approach, requires a non-operational way of thinking, where the end result of an operation must be described without describing the intermediate process leading to this result.

  For example, suppose we have to describe an operation where given a set `s1`, all the elements from a set `s2` have to be removed. Denote by "`*s1`" the state before the operation and by "`s1`" the state after. A common mistake made by students is to write something like the following, interpreting the `ALL` quantifier as a kind of `FOR` loop and interpreting "`=`" as a kind of assignment:

```
TRANSITION
  ALL( x: t SUCH THAT x IN s2 ::
          s1 = *s1 - {x} )
```

- Mutable vs. immutable types: The notions of object vs. value seem quite confusing for many students. First, it does not seem to be clearly stressed in the early programming courses. Also, the fact that the programming paradigms course introduces those concepts mainly implicitly by using *different* languages (`SML` vs. `Smalltalk`) may not help students understand that these notions are equally valid, and distinct, in any programming language.

7 Conclusion

We have presented a new undergraduate course in formal methods recently introduced at UQAM. This course differs from a number of previous courses in formal methods as it *does not* emphasize program derivation or program correctness. Instead, it focuses on the specification phase, trying to instill the importance of good descriptions and to introduce students to non-operational thinking.

At the time of final writing, the `Web` support for the exercices has been used only one, not yet complete, semester (it was installed during summer 1997). Evaluation and feedback on it will be obtained in the coming weeks: we look forward to how the students have appreciated — or not? — this support.

Acknowlegments

Financial support for the development of the "*Système interactif d'aide aux exercices*" (`Web` support for the exercices) briefly described in section 5 was provided by the *Fonds de Développement Pédagogique* de l'UQAM. The system was designed by the author and implemented by two undergraduate students, Myriam Pitre and Pierre Levasseur. Thanks also to Ivan Maffezzini for his comments on a first draft of the paper.

[1] The URL is the following:

`http://www.info.uqam.ca/~mat3143/Exercices`

But note that *tout ce matériel est en français!*

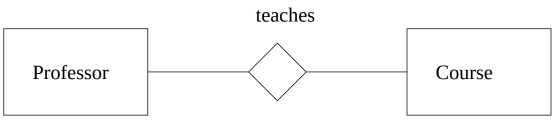

Figure 1: A small ERD

A Examples illustrating the `Spec` notation

A.1 Formalization of a simple Entity-Relationship Diagram

Suppose we have the ERD presented in fig. 1 and we want to indicate that a professor can teach at most three (3) courses and that a course is taught by a unique professor. This can be formalized in `Spec` using the following concepts:

```
DEFINITION courses_and_profs
 CONCEPT professor: type
  WHERE
   ALL( p: professor ::
           teaches_at_most{3}(p) )

 CONCEPT course: type
 CONCEPT professor( c: course )
         VALUE( p: professor )

 CONCEPT teaches( p: professor, c: course )
         VALUE( b: boolean )

 CONCEPT teaches_at_most{n: nat}( p: professor )
         VALUE( b: boolean )
   WHERE
    b <=> NUMBER( c: course SUCH THAT
                    teaches(p, c) :: c ) <= n
END
```

This `DEFINITION` module introduces a number of *designations* — concepts for which no detailed definition can be given (basic knowledge about the application domain); it also introduces a property (`teaches_at_most`) for which, using the designations, a formal (generic) *definition* can be given (`WHERE` clause).

A.2 Specification of a simple abstract machine

A simple abstract machine that models an accumulator which can be incremented and have its content tested is defined by the specification below. The specification also ensures that the content of the accumulator is always below a `maximum` value specified by the client of the module, thus defining a generic machine:

```
MACHINE accumulator{ max: nat SUCH THAT max > 0 }
 STATE( v: nat )
 INVARIANT v <= max
 INITIALLY v = 0

 MESSAGE add( x: nat )
  WHEN v + x <= max
   TRANSITION v = *v + x
  OTHERWISE
   REPLY EXCEPTION accumulator_overflow

 MESSAGE contains( x: nat )
  REPLY( b: boolean )  WHERE b <=> v = x
END
```

In this example, a `MESSAGE` indicates an operation, a functionality of the module, a `WHEN` clause indicates a pre-condition, a `TRANSITION` indicates a state transition relating the condition before vs. after the operation and a `REPLY` indicates a response returned by the operation.

References

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

[2] D. Craigen, S. Gerhart, and T. Ralston. Formal methods reality check: Industrial usage. *IEEE Trans. on Soft. Eng.*, 21(2):90–98, Feb. 1995.

[3] R. Denman, D.A. Naumann, W. Potter, and G. Richter. Derivation of programs for freshmen. In *Proceedings of the 25th SIGCSE Technical Symposium on Computer Science Education*, pages 116–120. ACM SIGCSE, March 1994.

[4] J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[5] M. Jackson. Description is our business. Keynote address at VDM '91, 1991.

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

[7] J.R. Jefferson Wadkins. Rigorous proofs of program correctness without formal logic. In *Proceedings of the 26th SIGCSE Technical Symposium on Computer Science Education*, pages 307–311. ACM SIGCSE, March 1995.

[8] C.B. Jones. *Systematic Software Development using VDM.* Prentice-Hall International Series in Computer Science, 1986.

[9] K.-K. Lau, V.J. Bush, and P.J. Jinks. Towards an introductory formal programming course. In *Proceedings of the 25th SIGCSE Technical Symposium on Computer Science Education*, pages 121–125. ACM SIGCSE, March 1994.

[10] H. McLoughlin and K. Hely. Teaching formal programming to first year computer science students. In *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education*, pages 155–159. ACM SIGCSE, March 1996.

[11] J.M Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, Sept. 1990.