in 4th Congress of Logic Applied to Technology (LAPTEC 2003). pages 9–18. Nov. 2003 © Ed. Pleiade

Rigorous Development of Functional Programs using Temporal Logic

Carlos H. C. Duarte
(carlos.duarte@computer.org)

BNDES, Av. Chile 100, Rio de Janeiro, RJ, 20001-970, Brazil
Universidade Estácio de Sá, Rua do Bispo 83, Rio de Janeiro, RJ, 20261-902, Brazil

Abstract. In this paper, we propose a new method, based on the use of temporal logic, for developing and reasoning about functional programs. Our software development method is rigorous and systematic: starting with a list of informal requirement descriptions, we initially derive a set of object-based specifications, which are later on transformed into modular monadic functional programs. The obtained specifications and programs are shown to consist in an effective basis for verification.

Keywords: Software Development, Formal Methods, Object-oriented approach, Temporal Logic, Functional Programming.

1 Introduction

For years, formal methods have been regarded as a promise for bridging the gap between software development processes and those adopted in engineering physical artifacts. Essentially, what is expected from the use of formal approaches is the development of reliable software artifacts through controlled and organised processes, using techniques and tools similar to those adopted in other engineering fields (e.g. the integral calculus and numerical approximation techniques used in the structural design of buildings within Civil Engineering).

Many formal methods and techniques exist today, each of which designed for treating specific domains. For instance, VDM [13] is a method based on a three-valued logic, which is particularly well-suited to specifying the pre and post-conditions of statements in sequential programs. Temporal logics, on the other hand, are appropriate for specifying and reasoning about reactive systems [16, 1].

So far, the adoption of formal development approaches has not obtained widespread acceptance, remaining as a research subject which is normally recalled only in specific practical situations. Some authors even suggest that software development is an informal activity by nature. In order to appempt to reconcile these viewpoints, the integration of formal and informal approaches has been proposed (e.g. [2]). The necessity of either justifying in rigorous terms the transition from application requirements to program code [3] or verifying a development once an executable program is obtained from specifications [11] also seem to count against the use of formal methods. These obligations, however, can be treated with formal software development environments. The consensus seems to be that it is not easy to define an effective formal software development process.

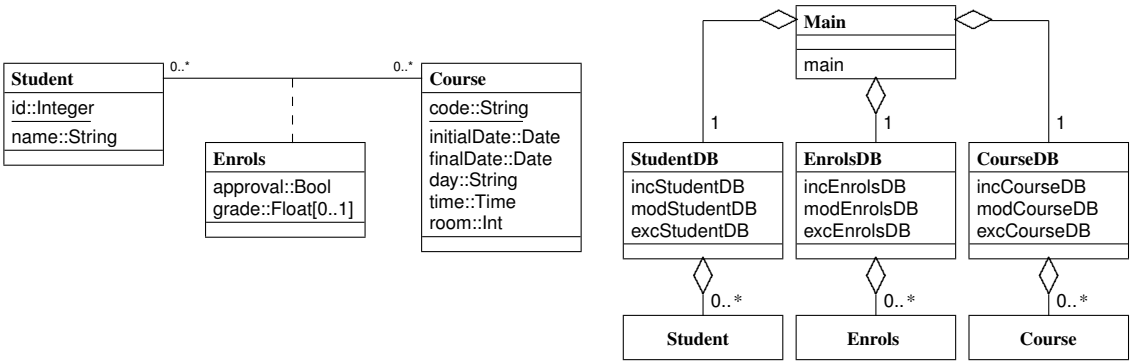

Figure 1: Specification of the academic system: (a) class diagram and (b) context diagram.

In this paper, as an exercise we develop the design, refinement, implementation and verification of a toy academic system. Our purpose here is to illustrate that, right now, we can propose the integrated application of a set of techniques and tools so as to define a new formal development method. This method, on the one hand, does not seem to be too distant from the current practises, but, on the other, enables the effective use of formal methods in software development activities.

Specifically, we describe in Section II the application requirements of our system. In Section III, we derive a set of UML diagrams [15], a set of temporal logic specifications [16], and, finally, an executable program written in the monadic functional language Haskell [12]. Next, in Section IV, we outline the formal verification of some properties of the described system. We conclude the paper with some comments on related and future research.

2 A Toy Example

Our problem consists in providing automated support for the academic activities of a university registrar. The system to be developed should provide support for the aspects of this problem related to students and their enrolment in specific courses. The following requirements are assumed to have been elicited after interviewing the actors in the problem domain:

- It should be possible to record, modify and erase students, courses and course enrolments;
- The attributes of interest regarding students are their personal id (key) and name;
- The attributes of interest regarding courses are the respective discipline code (key); initial and final date; the day of week, time and room of all classes;
- Concerning each enrolment relating a student to a course, it is necessary to record whether or not the student is approved in the course, with the respective final grade if appropriate.

As a constraint, a student is never allowed to enrol simultaneously in any pair of courses which have classes with conflicting date and time.

3 From Requirements to Specifications and Programs

3.1 Informal Design

The initial specification of our system is the class diagram presented in Figure 1 (a). That static structural model considers the existence of two object classes (with their respective attributes), **Student** and **Course**, and an association class, **Enrols**, to represent the many-to-many relationship between students and courses.

As we have already mentioned, our purpose here is to guide the development process towards the derivation of a program entirely written in a purely functional language. It is well known that purelly functional programs can only deal with state in an implicit way, considering state components as part of the environment which the program manipulates in a functional way [20]. Consequently, we are led to study the aforementioned classes within their utilisation context, which is depicted in Figure 1 (b). That implementation biased diagram is analogous to the context diagrams found in essential system analysis methods [14].

The diagram in Figure 1 (b) specifies our system as the **Main** class, an aggregation of three storage classes – dealing with students, courses and enrolments – which are manipulated as a single execution environment by method `main`. Storage classes themselves are aggregations that possess inclusion, modification and exclusion methods. With this organisation, the basic objects of our system are to be static, whereas those dealing with the environment will have dynamic behaviour. This guideline should be followed in the development of any information system.

3.2 Formal Design

Now, we transform each class specification into a theory presentation. We choose to define presentations here using the first-order temporal logical system described in [8], since the manipulated static objects may have arbitrary complex structure and the described system may present reactive behaviour. Each specification describing static objects gives rise to a new sort symbol, representing the population of objects of the corresponding type, and a construction function, which takes as arguments values of the respective attribute types and produces an object of the new type. Attribute names are translated into two different kinds of functional symbols: if the attribute is part of the object primary key, then it is mapped into a rigid symbol, which never suffers a value change, or else it is mapped into a flexible symbol, which may face value changes as time passes. Classes without an apparent primary key are refined into a new structure with an additional attribute to hold a generated unique id. Many of the properties stated in each presentation are derived from the requirements specification, but some already reflect the required information system operations to solve our problem.

In Figure 2, we present the vocabulary of symbols and the axioms derived from the student class. Students are constructed using the rigid function $STUDENT$, have their identity retrieved using the rigid function $idStudent$ and their currently adopted name recovered by the flexible function $nameStudent$. Two properties are specified: that two students with the same id are actually the same person (1.1), reflecting the fact that id is the primary key of **Student**, and that the identity of a constructed student is that mentioned in the construction function first argument (1.2), a closure property.

The derivation of theory presentations for basic object classes is relatively simple. On the other hand, the derivation of properties of storage classes is where the complexity lies, be-

Presentation STUDENT
imports INTEGER, STRING
sorts Student
rigid $STUDENT$ :: Integer $\rightarrow$ String $\rightarrow$ Student, $idStudent$ :: Student $\rightarrow$ Integer
flexible $nameStudent$ :: Student $\rightarrow$ String
axioms r, s :: Student, i :: Integer, n :: String
$idStudent\ r = idStudent\ s \rightarrow r = s$ (1.1)
$idStudent(STUDENT\ i\ n) = i$ (1.2)
End

Figure 2: Specification of students.

cause we are obliged to treat their behaviour over time. In Figure 3, for instance, we show the presentation corresponding to the student storage class. All the symbols defined in the presentation are flexible, reflecting their time dependant meaning. $StudentDB$ is a predicative symbol that takes a student as an argument and holds for this value as long as it is considered to be one of the stored objects. The other three symbols reflect the operations that may be performed. Note in axiom (2.1) that we are required to treat the flexible symbols of the basic class in the storage class specification: the axiom states that, if a student has some given name and identity, then these attribute values must yield the same student if used for construction, but observe that the converse is not the case because a student may adopt different names at different times. Axioms (2.2), (2.5) and (2.7) specify enabledness properties stating respectively that a student can only be recorded if this is not currently the case, or if the opposite is true for modification and exclusion. Axiom (2.3), similarly to axiom (2.8), specifies a causality property saying that a student is stored immediately after an inclusion and continues in this situation until excluded from the class. Axioms (2.4) and (2.6) specify the effect caused by a student inclusion or modification over student names. Finally, axiom (2.9) states a so-called locality property [9], that student names only change if an inclusion or modification happens.

The derivation of theory presentations corresponding to associations and association classes is a story on its own. The specification has to be successively transformed using the following process, which is very similar to that adopted in the logical design of database systems [6]:

- Each generalisation (or specialisation) hierarchy is treated as a one-to-many association;
- Each many-to-many relationship (association class or not) is treated as a class with a pair of one-to-many associations having as the many side that of the originally related classes;
- Associations of one-to-many kind imply in the inclusion of new attributes at the many side of the relation, representing the primary key of an object at the other side of the relationship;

After a class is derived using these rules, it is treated as any basic class. The presentation derived from **Enrols** is exhibited in Figure 5. Note that, because the primary key of **Enrols** is composed, the functional dependence of this object type in relation to both key components has to be captured as a conjunction (3.1).

Presentation STUDENTDB
imports STUDENT
flexible $StudentDB(\mathsf{Student})$,
$incStudentDB(\mathsf{Student}), modStudentDB(\mathsf{Student}), excStudentDB(\mathsf{Student})$
axioms $s :: \mathsf{Student}, i :: \mathsf{Integer}, n :: \mathsf{String}$

$$nameStudent\ s = n \wedge idStudent\ s = i \rightarrow s = STUDENT\ i\ n \tag{2.1}$$
$$incStudentDB(s) \rightarrow \exists r \cdot StudentDB(r) \wedge idStudent\ s = idStudent\ r \tag{2.2}$$
$$incStudentDB(s) \rightarrow \mathbf{X}((StudentDB(s))\mathbf{W}(excStudentDB(s))) \tag{2.3}$$
$$incStudentDB(s) \wedge (s = STUDENT\ i\ n) \rightarrow \mathbf{X}(nameStudent\ s = n) \tag{2.4}$$
$$modStudentDB(s) \rightarrow \exists r \cdot StudentDB(r) \wedge idStudent\ s = idStudent\ r \tag{2.5}$$
$$modStudentDB(s) \wedge (s = STUDENT\ i\ n) \rightarrow \mathbf{X}(nameStudent\ s = n) \tag{2.6}$$
$$excStudentDB(s) \rightarrow StudentDB(s) \tag{2.7}$$
$$excStudentDB(s) \rightarrow \mathbf{X}((\neg StudentDB(s))\mathbf{W}(incStudentDB(s))) \tag{2.8}$$
$$incStudentDB(s) \vee modStudentDB(s) \vee (nameStudent\ s = n \rightarrow \mathbf{X}(nameStudent\ s = n)) \tag{2.9}$$

End

Figure 3: Specification of the student storage class.

The derived presentation for the enrolment storage class possesses interesting characteristics. Inclusion can only happen if the respective student and course are recorded (4.1). Exclusion is only allowed if the relationship exists and the corresponding student has not received his final grade in the course (4.3), this last property implied by the problem requirements. Another relevant point to mention concerning this presentation is that it not only depends on the presentations of student and course storage classes, as discussed above, but it also regulates the occurrence of the operations defined therein: axioms (4.5) and (4.6) state that a student or course exclusion can only happen if the respective object does not participate in any relationship. This recalls us that the derived set of presentations can be seen together as yet another application of the categorical techniques described in [9].

3.3 Implementation Sketch

An implementation for the system in question, written in the monadic functional language Haskell, is obtained following the steps described below:

- Each derived theory presentation gives rise to the definition of a new abstract data type (inside a `module`), implemented in terms of algebraic and/or pre-defined basic types (defined with `data` or `type`);
- Each functional symbol becomes a field label of the respective algebraic type (defined between curly braces after `data`);
- Each predicative symbol denoting a data storage is implemented as a list of basic objects (defined using square braces after `type`);
- `main` is implemented as an interactive function, allowing the user to select a supported operation and subsequently inform any required data to perform the chosen method;

```
-- Student.hs
module Student (Student, STUDENT, idStudent, nameStudent) where

data Student = STUDENT {idStudent::Integer, nameStudent::String}

-- StudentDB.hs
module StudentDB (StudentDB, incStudentDB, excStudentDB, modStudentDB) where

import Student

type StudentDB = [Student]

selStudentDB :: StudentDB -> Integer -> [Student]
selStudentDB [] i = []
selStudentDB (h:t) i = if (idStudent h == i) then
                          h:(selStudentDB t i)
                       else
                          selStudentDB t i

incStudentDB::StudentDB -> Student -> StudentDB
incStudentDB l s = (s:l)

excStudentDB::StudentDB -> Student -> StudentDB
excStudentDB []    s = []
excStudentDB (h:t) s = if (idStudent s == idStudent h) then (excStudentDB t s) else
                          (h:excStudentDB t s)

modStudentDB::StudentDB -> Student -> StudentDB
modStudentDB []    s = []
modStudentDB (h:t) s = if (idStudent s == idStudent h) then (s:t) else
                          (h:modStudentDB t s)
```

Figure 4: Sketch of the implementation.

- Inclusion, modification and exclusion methods are implemented as functions with the type of the respective focus object inlined as the first formal parameter and also as the function result.

Following these steps, the implementation of **Student** and **StudentDB** are presented in Figure 4. It is important to say that some specified properties are not explicitly handled in this implementation process since they are ensured by the programming language semantics. For instance, this is the case for locality and closure properties. Enabledness and causality properties, on the other hand, are used in the derivation of the pre and post-conditions of execution of each storage class method, which appear in the derived main program.

4 From Specifications and Programs to Correctness

At this point, in order to attest the correctness of the derived program, which is guaranteed by the outlined method, we need to adopt a distinct (non-temporal) programming logic. It is also possible to continue using the same logical system and conservatively extend our specifications to capture the relevant properties of the derived implementation, in order to verify some other properties.

Following the second path, some new axioms are introduced in the specifications due to the chosen target programming language and its semantics. This is the case, for instance, regarding sequentially: in standard Haskell, since only one expression can be reduced at each time, we are ensured that always at most one object method is in execution. Note that this is enforced only at the implementation level: the concurrent occurrence of modification and ex-

Presentation ENROLS
imports INTEGER, STRING, BOOL, FLOAT
sorts Enrols
rigid $ENROLS$ :: Integer $\rightarrow$ String $\rightarrow$ Bool $\rightarrow$ Bool $\rightarrow$ Float $\rightarrow$ Enrols,
$idEnrols$ :: Enrols $\rightarrow$ Integer, $codeEnrols$ :: Enrols $\rightarrow$ String
flexible $approvalEnrols$ :: Enrols $\rightarrow$ Bool,
$existsGradeEnrols$ :: Enrols $\rightarrow$ Bool, $gradeEnrols$ :: Enrols $\rightarrow$ Float
axioms x, y :: Enrols, a, e :: Bool, i :: Integer, c :: String, f :: Float

$$idEnrols\ x = idEnrols\ y \wedge codeEnrols\ x = codeEnrols\ y \rightarrow x = y \tag{3.1}$$

$$idEnrols(ENROLS\ i\ c\ a\ e\ f) = i \tag{3.2}$$

$$codeEnrols(ENROLS\ i\ c\ a\ e\ f) = c \tag{3.3}$$

End

Figure 5: Specification of enrolments.

clusion operations would be admissible by our specifications, without any unexpected effect given that the observable result of the modification would be cancelled by the exclusion. Another example is that of causality properties, which only specify what would happen in specific situations but would not constrain these occurrences to happen only when caused. This shows that a completion process is necessary to capture all the implementation properties. The completed presentation is the appropriate artifact to provide as an input for automated software development environments comprising model checkers, for instance.

Following the first path above, we can perform the verification of some properties relying on the equational definitions of the pre-defined Haskell data types. For example, consider the implementation of the following interactive method, which appears in the `Main` module:

```
modStudent :: StudentDB -> IO (StudentDB)
modStudent d = do i <- getId
                  n <- getName
                  let r = selStudentDB d i
                  in  return (if r == [] then
                                 d
                              else
                                 let s = STUDENT i n
                                 in  modStudentDB d s)
```

At execution time, the function argument is always an alias of a manipulated state component and the function result definitely represents the new value of this part of the environment after the function execution terminates. This termination property is guaranteed by the absence of iteration or recursion. Moreover, this implementation preserves the validity of (2.5) and (2.6).

Assume that the execution of $modStudent$ is atomic. To verify (2.5), just note that the student database will never be modified if the student id is not in this storage. To verify (2.6), it is sufficient to ensure by structural induction that a student name will have been modified after the execution of $modStudentDB$. Therefore, both properties are guaranteed by our implementation. This kind of verification process can be better supported by interactive theorem provers.

Presentation ENROLSDB
imports ENROLS, STUDENTDB, COURSEDB
flexible $EnrolsD(\mathsf{Enrols})$,
$incEnrolsDB(\mathsf{Enrols}), modEnrolsDB(\mathsf{Enrols}), excEnrolsDB(\mathsf{Enrols})$
axioms $x, y :: \mathsf{Enrols}, n, wd :: \mathsf{String}, d_1, d_2 :: \mathsf{Date}, t :: \mathsf{Time}, o :: \mathsf{Int}$
...

$$incEnrolsDB(x) \rightarrow \exists n \cdot StudentDB(STUDENT(idEnrols\ x)\ n) \wedge \exists d_1, d_2, wd, t, o \cdot CourseDB(COURSE(codeEnrols\ x)\ d_1\ d_2\ wd\ t\ o) \wedge \exists y \cdot EnrolsDB(y) \wedge idEnrols\ x = idEnrols\ y \wedge codeEnrols\ x = codeEnrols\ y \quad (4.1)$$

$$incEnrolsDB(x) \rightarrow \mathbf{X}((EnrolsDB(x))\mathbf{W}(excEnrolsDB(x))) \quad (4.2)$$

...

$$excEnrolsDB(x) \rightarrow EnrolsDB(x) \wedge \neg(existsGradeEnrols\ x = \text{TRUE}) \quad (4.3)$$

$$excEnrolsDB(x) \rightarrow \mathbf{X}((\neg EnrolsDB(x))\mathbf{W}(incEnrolsDB(x))) \quad (4.4)$$

...

$$excStudent(s) \rightarrow \exists x \cdot EnrolsDB(x) \wedge idStudent\ s = idEnrols\ x \quad (4.5)$$

$$excCourse(c) \rightarrow \exists x \cdot EnrolsDB(x) \wedge codeCourse\ c = codeEnrols\ x \quad (4.6)$$

End

Figure 6: Specification of the enrolment storage class.

5 Concluding Remarks

In this paper, we have sketched the application of a method that not only relies on the use of temporal logic but also integrates formal and informal techniques aiming to develop functional programs. Based on a list of requirements, we initially construct a collection of UML diagrams, then systematically derive a set of temporal specifications, and finally implement them as a functional program. We believe that this is a relevant contribution since the outlined development method keeps many similarities with the current practises while taking advantage of the adopted formal techniques. This method was validated in the development of a complete academic system comprising 8 classes and 3200 lines of derived Haskell code.

Some other authors have defended similar ideas in the literature. Bekaert proposes the transformation of object-oriented specifications into logical theories [4]. Bose suggests the application of model checking in translating UML specifications into code [5]. Russel created new development techniques aiming at functional language implementations [17]. Thomsom [19], Shroeder and Mossakowski [18] have developed programming logics for functional languages. We are not aware of any proposal treating the whole software development process.

The described work highlights the relevance of associating a (temporal) specification logic to declarative languages, an idea initially proposed in [10]. One promising direction for future work is to attempt to generalise our method, first to other declarative paradigms and later to imperative languages. An orthogonal direction for future research is to investigate if the ideas discussed here can be transported to the development of concurrent and distributed systems. We foresee that, in order to deal with the open nature of these systems, a rely-guaranteed reasoning discipline will be required, as studied in [7].

References

[1] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable specifications of reactive systems. In G. Ausiello, M. Dezani-Ciancaglini, and S. R. D. Rocca, editors, *Proc. 16th Conference on Automata, Languages and Programming (ICALP'89)*, volume 372 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 1989.

[2] D. Andrew and D. Ince. *Practical Formal Methods with VDM*. McGraw-Hill, 1991.

[3] F. L. Bauer. From specifications to machine code: Program construction through formal reasoning. In *Proc. 6th Conference on Software Engineering (ICSE'82)*, pages 84–91, 1982.

[4] P. Bekaert, B. V. Nuffelen, M. Bruynooghe, D. Gilis, and M. Denecker. On the transformation of object-oriented conceptual models to logical theories. In *Proc. 21st Conference on Conceptual Modeling (ER2002)*, 2002.

[5] P. Bose. Automated translation of UML models of architectures for verification and simulation using Spin. In *Proc. 14th Conference on Automated Software Engineering*, pages 102–109. IEEE Computer Society, October 1999.

[6] P. P. Chen. The entity-relationship model - towards a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[7] C. H. C. Duarte and T. Maibaum. A rely-guarantee discipline for open distributed systems design. *Information Processing Letters*, 74(1–2):55–63, April 2000.

[8] C. H. C. Duarte and T. Maibaum. A branching-time logical system for open distributed systems development. *Electronic Notes on Theoretical Computer Science*, 67, 2002.

[9] J. Fiadeiro and T. Maibaum. Temporal theories as modularisation units for concurrent systems specification. *Formal Aspects of Computing*, 4(3):239–272, 1992.

[10] J. L. Fiadeiro, T. Maibaum, N. Martí-Oliet, J. Meseguer, and I. Pita. Towards a verification logic for rewriting logic. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Data Type Specificaiton: 14th Workshop on Specification of Abstract Data Types*, volume 1827 of *Lecture Notes in Computer Science*, pages 438–458. Springer-Verlag, 1999.

[11] C. A. R. Hoare. A calculus of total correctness for communicating sequential processes. *The Science of Computer Programming*, 1(1):49–72, 1981.

[12] P. Hudak. Mutable abstract datatypes. Technical Report YALEU/DCS/RR914, Department of Computer Science, Yale University, December 1992.

[13] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 2nd edition, 1990.

[14] S. McMenamin and J. Palmer. *Essential Systems Analysis*. Yourdon Press, 1984.

[15] O. M. G. OMG. *Unified Modelling Language Specification*. Object Management Group — OMG, June 1999. Version 1.3.

[16] A. Pnueli. The temporal logic of programs. In *Proc. 18th Symposium of Foundations of Computer Science*, pages 45–57. IEEE Press, 1977.

[17] D. Russell. *FAD: A Functional Analysis and Design Methodology*. Phd thesis, Computing Laboratory, University of Kent at Canterbury, January 2001.

[18] L. Schroeder and T. Mossakowski. Hascasl: Towards integrated specification and development of functional programs. In *Proc. 9th Algebraic Methodology and Software Technology (AMAST'02)*, 2002.

[19] S. Thompson. A logic for miranda. *Formal Aspects of Computing*, 1(1):339–365, July 1989.

[20] P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.