Teaching Formal Methods in a Third World Country: What, Why and How

Maximiliano Cristiá
U. N. de Rosario - U. N. de Córdoba
Flowgate Security Consulting
mcristia@fceia.unr.edu.ar

Abstract

We have been teaching formal methods for eight years at two major Argentinean universities. It is hard to find examples of the application of formal methods outside the most advanced industrial sectors. Then, why teach formal methods in a country that hardly produce software for advanced industries? Why formal methods in a country which buy technology instead of creating it? We were one of the first in teaching formal methods in Latin America, likely the first in teaching TLA and CSP. Our former students are now pursuing PhD's in several European countries and Argentina. Slowly but steadily our graduates are *infiltrating* the local industry.

Keywords: Formal methods, Argentina, Software Engineering, undergraduate course

1. INTRODUCTION

Argentina is a third world country. As such our industry heavily depends on the industry and technology of developed countries. Our country produces new technology only for a handful of sectors (biotechnology, oil piping, experimental nuclear reactors, and a couple more). Formal methods have been applied mostly to software systems to be used in advanced industrial sectors (avionics, air traffic control systems, medical devices, etc.) [1]. At a first glance, then, it looks like formal methods should be taught in courses attended by future software professionals of countries where these advanced industrial sectors are important.

However, we teach formal methods at two major Argentinean universities: Universidad Nacional de Rosario (UNR) since 1998 and Universidad Nacional de Córdoba (UNC) since 2003 –it is worth saying that formal methods are being taught in Argentina at Universidad de Buenos Aires since a couple of years before we started. Besides, there are other professors in these and other universities who are giving undergraduate or graduate courses on formal methods. They range from formal specification, to model checking, to architectural description, to formal verification and theorem proving.

This paper is not meant to be a report on the education of formal methods in Argentina. It is just to comment about our own experience since 1998 at UNR and with less emphasis on our more recent participation at UNC. In this sense, the paper starts, at section 2, by describing the context where we work, the formal methods we teach, and the reasons behind our choice. Section 3 explains how we teach formal methods and the resources we use. The conclusions of our experience are in section 4.

2. WHAT WE TEACH AND WHY

Before listing the methods we teach, we think that it is important to comment about the context where these courses are given. Once the context is clear we show the notations and techniques included in our course and the reasons behind this selection.

2.1. The Context

We teach formal methods in a subject named System Analysis of the fourth year of the Licenciatura on Computer Science (LCC) which is a degree part of the Facultad de Ciencias Exactas, Ingeniería y Agrimensura (FCEIA) of the UNR.

LCC was created in 1995 as part of the Mathematics Department. It is a five years degree with a strong emphasis on R+D. There are 28 subjects including calculus, algebra, physics, logic, computability and automata, data structures, operating systems, etc. Also there is a strong emphasis on functional programming and programming language semantics. For instance, in the last two years one introductory course on programming has been based on formal program development. All subjects but two are mandatory and students need to complete an industrial internship and a R+D thesis to get their degree. Nowadays, LCC has 300 students approximately with less than 30 graduates, although this number is growing rapidly in the last two years.

In the fourth year there are six subjects divided in equal numbers in both semesters. The course where we teach formal methods, System Analysis, is in the first semester and the complementary course, named Software Engineering, is in the last semester. Both subjects have an assignment of eight hours per week for sixteen weeks –totalizing 128 hours each.

Half a dozen of LCC's subjects are quite hard to pass. System Analysis is one of them. In the last three years we had an average of thirty students per semester: around 50% percent of them finished the course without having to take it again[1]. Most of the students who fail the first time, succeed in their second attempt, some having an outstanding performance in the second chance.

2.2. What Formal Methods we Teach

Since the course was given for the first time its contents have varied but the core remained the same. Many changes were introduced to correct errors, to improve the presentation, or to adjust the domains where some particular method is applied. Here, we just list the methods we teach making no reference to the form of the presentation, the intended use, or the resources we use –this is shown in section 3.

1. **Introduction**. In a four hours introduction we show that the Engineering part of Software Engineering is more of a desire than a reality. Many examples of disastrous software development projects are shown. Then, some of the best known examples of the successful application of formal methods are shown. Also, throughout the lecture, Software Engineering is compared with traditional engineering disciplines and its differences in success, efficiency and guaranties are remarked. At last, a cause for this difference is sought. Two possible answers are given: (a) the position of Brooks, and (b) the short history of Software Engineering and its fundamental epistemological difference with respect to the other branches of engineering. From Brooks's position we borrow his observation that silver bullets are always sought for the programming phase, leaving unattended the specification and design of the system.
2. **Z**. The Z formal notation is taught first because we think its one of the easiest languages to learn. We rest on the fact that students know first order logic and mathematics. This part of the course is quite standard with respect to classical text books on this matter. Types, state machines, schema, schema operations, schema promotion and schema composition are shown with detail in eight lectures summing up 32 hours.
   We deviate from the "standard" in that state invariants are not included in state schema. Instead we follow what we call the "TLA style", i.e. invariants are recorded in other schema and proof obligations are included for each operation. We proceed in this way due to two reasons: (a) we believe that operation specifications are more readable because implicit preconditions disappear, and (b) because this enables us to latter introduce formal verification (see below). A very simple example is a specification of an operation to decrement the value of a natural variable in one unit. The standard Z way is as follows:

[1] "Finish" does not mean to pass the final exam, it just mean that they fulfilled the conditions to continue with other subjects.

$$Sys \mathrel{\widehat=} [x : \mathbb{Z} \mid x \geq 0] \qquad Decr \mathrel{\widehat=} [\Delta Sys \mid x' = x - 1]$$

Why, if *x* must always be greater than or equal to 0, the specification of *Decr* does not include $x > 0$ as a precondition? The answer is that there is an implicit precondition hidden in the inclusion of ΔSys. Stating the invariants in this way, makes it impossible to write an operation over *Sys* that does not verify its invariant because that schema *defines* only states where *x* is greater than or equal to 0. Then, either the specifier gives the programmer the implicit precondition, the programmer calculates it or the programmer implements the wrong operation. Instead we propose the following specification:

$$Sys \mathrel{\widehat=} [x : \mathbb{Z}] \qquad Invariant \mathrel{\widehat=} [Sys \mid x \geq 0] \qquad Decr \mathrel{\widehat=} [\Delta Sys \mid x > 0 \land x' = x - 1]$$
theorem *Decr_Preserves_Invariant* : $Invariant \land Decr \Rightarrow Invariant'$

Thus, the programmer has all the information needed to implement *Decr* and the work of calculating and writing the implicit precondition *in a readable way* is changed by the (optional) effort of proving *Decr_Preserves_Invariant*.

3. **Statecharts**. We tell to the students that concurrent systems are hard, if possible, to be described in Z. Then, we need some formalisms to talk about concurrency and parallel systems. We start with a simple but clever and widely known graphical formalism. All aspects of Statecharts are shown including and-sates, or-states, extended transitions, synchronization by conditions, history, parametrized Statecharts, and so on. We spent 16 hours in Statecharts
4. **CSP**. Students found themselves the limitations of Statecharts. Then, we introduce a more powerful notation, CSP. As with the previous notations a rather practical course on CSP is given, including: events and processes, recursion and mutual recursion, algebraic operators, synchronous model, communication channels, determinism and non determinism, timers and simple real-time requirements, parametrized processes, etc. The failures/divergences semantics of CSP is also shown. CSP consumes 24 hours.
5. **TLA+**. We introduce TLA by first giving an semi-formal introduction to stuttering steps, the Alpern-Schneider theorem, safety, liveness, fairness and machine closure. Then, we introduce the language showing how it was designed based on all the previous ideas and results. It takes us 32 hours including some real-time specifications.

Also, a couple of classes are devoted to use a proof assistant to prove some state invariants of Z specifications. As the reader can see, the course is quite heavy and the schedule is very tight. Usually, some contingencies –such as strikes, holidays and exams– arise and the last topic is left to the students as homework.

2.3. Why we Teach Formal Methods

If formal methods are required only by the most advanced industrial sectors, why do we teach formal methods in a country where these sectors are not to relevant? Are we doing the right thing? Is it not a waste of time and money? Will our country benefit from our decision?

There are five reasons for which we teach formal methods and we believe that these give positive answers to all of these preceding questions.

1. **History.** By the time we had to design a course on Software Engineering for LCC there were only a handful of similar courses in Argentina and the Internet were not so common as today. Then, we asked advice to professors at Universidad de Buenos Aires who were teaching some formal methods. A couple of years after that we found on the Internet a course of the Master of Software Engineering at Carnegie-Mellon University named Models of Software Systems [2]. In that course professor D. Garlan proposed to teach various formal techniques to specify software systems. We borrowed from it some topics, ideas and references.
2. **Applicability and belief.** We teach formal methods because we believe they are the right way to build software. By using formal methods we think that the practice of software construction could be called Engineering. It is likely that there are other ways to get the same status but eventually we choose formal methods. We think that formal methods can

be applied more widely than they are today. Applications domains such as core banking systems, industrial control, ERPs and many more should benefit from formal methods. However, we believe that this benefit could become real only if formal methods are used in combination with traditional practices and are used mostly for specification and testing –in our opinion formal verification and formal refinement are still too immature for industry.

3. **Economics.** Here we see three different concerns: (a) to teach formal methods is cheap, (b) in the long run software produced with formal methods is less costly, and (c) software produced with formal methods has a huge added value. For the most cases to teach formal methods you do not need licensed software. Further, since we came from a Science faculty with a strong mathematical background it is more or less easy to learn them. We do not need to take expensive courses given by companies or foreign universities.
   The software industry can be set up at a rather low cost. Then it becomes a very appealing option for third world countries like Argentina. Moreover, the more added value a piece of software has, the more return it gives. High quality raises the chances of bigger exportation. Hence, if we want to help our software industry we have to form human resources as qualified as possible. Teaching formal methods produce such human resources. It has been said that the main economic barrier against the adoption of formal methods is the lack of professionals ready to apply them, and that the costs in training existing engineers can be prohibitive. Students attend UNR at no cost, then, by teaching formal methods, we become a sort of free training center for the local industry. We think this is an ideal situation in which the state transfers resources to the private sector at no cost.
4. **It is always easier to be informal than formal.** Hence, we need to take our students to a higher level of formality because, naturally, they will tend to be informal. It is very important to be rigid in the classroom with respect to the application of formal methods because once in the trenches they will be attracted by informality. By teaching formality we make no harm.
5. **We want to form rigorous Software Engineers.** We want our graduates to change the way software is produced in our region. UNR is the leading university in our region and its mission is to lead the way technology is produced. To reach this goal we need to form better software engineers than those who had graduated from other competing universities and private universities. Local companies suffer from all the oddities of traditional software production but at the same time they want to export software overseas. Then, by graduating rigorous Software Engineers we have the chance to change this situation.

2.4. Why we Choose this Combination

We think that a software engineer must master as many formal specification languages as programming languages or architectural styles or design patterns. Then we prefer to be shallow in the introduction of each method but wide in the menu. If they know several different notations they will be more prepared to approach different problems and situations.

Note that the chosen methods are not just apparently different but they differ in their very essence. We did not choose Z and VDM or B and Z, nor CSP and CCS. We believe that we are covering a rather complete set of attributes: Z is typed and based on first order logic and set theory, Statecharts is oriented towards concurrent systems and is graphical, CSP is for concurrent systems too but is textual and based on events, but TLA is not typed, is textual, is intended for concurrent systems but is an state-based formalism. If a student is lucky and can work with formal methods, then he or she can choose one of these or any other to become an expert.

3. HOW WE TEACH FORMAL METHODS

3.1. Some Pedagogical Concerns

Lectures are of two types: theory and practice. Theory is given by the head of the course while practice is given by some advanced students or graduates. Each kind of lecture has assigned a four hours class per week. Students are free to attend any lecture and are free to come and go in the middle of any of them. There are also office hours that any student can require once a week; in this classes students have another opportunity to ask specific questions to the teachers.

At the beginning we used slides for all the theory lectures but three years ago many students said that they did not like slides, that they rather blackboard and chalk. They argued that slides are boring and that they have problems in reading the slides and listening the teacher at the same time. Also, they said that slides give the sensation that the topic is easy, and that they want to see the teacher solving problems right before them. Hence, we abandoned slides and moved to traditional class notes. For now, we see a better performance of the students and they are more active in the classroom. All class notes follow a common pattern: a medium size model is described while concepts are introduced. In this way, students can write models quickly despite they do not have a deep knowledge of the formal semantics of the notation. We found that a lack of a detailed understanding of semantics is not an obstacle to write good models. In fact, despite our students have a good background in mathematics and logic, we discovered over the years that their main problem in writing formal models is abstraction. We found that they could understand complex semantics like temporal logic but they could not write good TLA+ models. Then, in the last years we focused in showing how to abstract, how to write models, how to forget thinking as a programmer. So, now, we spend more time in showing models than in teaching semantics. Formal semantics are taught just for CSP and TLA but we hardly require them in the exams.

Practice lectures have suffered less changes. In these classes students are asked to work on exercises sheets. Each exercise declares a set of functional requirements and the student is asked to write a formal specification in a particular notation. We do not ask them to find the right language. We hope they will be able to do so after solving many problems in different languages. Sometimes we ask them to use a notation not well suited for a particular problem. The intention is to show that, precisely, that notation is not good for that kind of problems. This kind of exercises are given at the end of the course when the students have learned at least three languages.

As the reader can see practice lectures are aimed to solve practical problems. We do not include theoretical exercises like showing the equivalence of two models or working with the semantics of a language. This is supported by theory lectures which, actually, are not so theoretical. All the course is oriented to form rigorous software engineers and not theoreticians of formal methods. The diversity of methods is there to give the students a wide menu of options and not to show many different theories. Our main objective is to give students the conceptual and intellectual tools to be able to apply formal methods routinely in their future positions. In the same sense, note that the whole course spent almost all the time in formal specification and barely nothing in formal verification. This comes from our belief that formal specification is economically feasible for most systems while formal verification is, for the near future, applicable in niche domains. In fact, the author of this paper regularly encourage the attendants to suggest the application of these methods in their future works or to use them silently and just show the results.

3.2. Resources

All the course material is maintained in a web page [3]. It consists of slides –even though they are not used in class anymore–, class notes, exercises sheets, software tools, papers and books. It should be noted that foreign books are quite expensive for us, then we cannot afford an important list of them; the library has only one copy of a handful of titles. Unless for some on-the-fly modifications students enjoy all the class material at the beginning of the course. Also there is a mailing list used mostly to communicate to the students news about the course. We have tried to make them to discuss technical topics through the list but we never succeeded.

The introduction is based on [4, 5, 6, 7] to show success and horror stories of software development. Jacky's book [8] is the recommended reading to learn Z but we also suggest [9, 10] and an on-line version of [11]. We also use class notes written by us [3] where we show how to write state invariants as was explained in section 2.2. Statecharts' mandatory readings are [12] and class notes where parametrized Statecharts are shown with more detail. The book of Hinchey and Jarvis [13] is the recommended reading for CSP and [14] is suggested as an advanced reference. We also use our own notes to show examples, standardize the notation and to introduce real-time modeling by means of timers [3]. Before introducing TLA, [15] is used to show some fundamental results, then Lamport's book [16] is used as the mandatory reading for

TLA+. Regarding software tools, we only require Z/EVES [17] for the formal verification section, but we suggest to use the TLA+ tools, too.

4. CONCLUSIONS

We are satisfied with what we have done so far. Former students of our courses are now pursuing PhD's at Europe, USA and Argentina in formal methods or related topics. Even PhD students who are not researching on formal methods sometimes add formality to their works. Nevertheless, it posses a threat: many of these students could never come back to Argentina giving their best to countries which did not spent to much in their education. Here, only a strong state policy oriented toward repatriation of high tech human resources could help.

At the same time, some graduates and students are working in the local industry timidly applying formal notations when possible. Some of them have told us that they feel so uncomfortable in their current position due to the way the work is done –informally, without discipline and without a clear methodology– that they want to move to another work. We do not know it for sure, but we believe we have some responsibility for those feelings: furthermore, after listening them we do not know if we did good or bad in instilling such a commitment to develop software in a formal manner.

We think that in the long term our students will modify the way software is produced in the local industry but we need to do more from the academy. In the last months, we set an arrangement with Polo Tecnológico Rosario to give a course to the staff of its member companies including Z and Statecharts along with Object Oriented Design, Design Patterns and Architectural Styles. The course is being attended by six developers, none of them being a former student of ours.

REFERENCES

[1] M. Hinchey and J. Bowen, *Industrial-Strength Formal Methods in Practice*, ser. Formal Approaches to Computing and Information Technology. Springer-Verlag, 1999.

[2] Master of Software Engineering, "Models of software systems," http://www.mse.cs.cmu.edy/Courses-17-651.html.

[3] M. Cristiá, "Material de la asignatura Análisis de Sistemas, LCC, FCEIA, UNR," http://www.fceia.unr.edu.ar/asist.

[4] W. W. Gibbs, "La crisis crónica de la programación," *Investigación y Ciencia*, pp. 72–81, Nov. 1994.

[5] R. Charette, "Why software fails," *IEEE Spectrum*, vol. 42, no. 9, pp. 36–43, Sept. 2005.

[6] N. Dershowitz, "Software horror stories," http://www.cs.tau.ac.il/ nachumd/horror.html.

[7] F. P. Brooks, "No silver bullet," in *Proceedings of the IFIP Tenth World Computing Conference*, H. Kugler, Ed., 1986, pp. 1069–1076.

[8] J. Jacky, *The Way of Z*. Cambridge University Press, 1997.

[9] B. Potter, J. Sinclair, and D. Till, *An Introduction to Formal Specification and Z*. Prentice Hall International, 1996.

[10] A. Diller, *Z: An Introduction to Formal Methods*. John Wiley Press, 1990.

[11] J. Woodcock and J. Davies, *Using Z: specification, refinement, and proof*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[12] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231–274, 1987.

[13] M. G. Hinchey and S. A. Jarvis, *Concurrent systems: formal development in CSP*. McGraw-Hill International Series in Software Engineering, 1995.

[14] A. W. Roscoe, *The Theory and Practice of Concurrency*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

[15] B. Alpern and F. B. Schneider, "Defining livennes," *Information Processing Letters*, vol. 21, no. 7, pp. 181–185, Oct. 1985.

[16] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, 2002.

[17] M. Saaltink, "The Z/EVES system," in *ZUM '97: The Z Formal Specification Notation*, J. Bowen, M. Hinchey, and D. Till, Eds., 1997, pp. 72–85.