Teaching Formal Methods for Software Engineering – Ten Principles

Antonio Cerone[1], Markus Roggenbach[2], Bernd-Holger Schlingloff[3], Gerardo Schneider[4], and Siraj Ahmed Shaikh[5]

[1] United Nations University — UNU-IIST, Macau SAR China
[2] Swansea University, Wales, United Kingdom
[3] Humboldt University and Fraunhofer FOKUS, Berlin, Germany
[4] University of Gothenburg, Sweden
[5] Coventry University, United Kingdom

Abstract. In this paper we report and reflect about the didactic principles underlying our endeavour to write a book on "Formal Methods for Software Engineering – Languages, Methods, Application Domains", and to teach its contents at international summer schools[6]. Target audience for the book are taught master students, possibly striving for a career in industry, and doctoral students in their early years, possibly in search of a suitable topic for their dissertation. We outline ten principles underlying the design of the book, coin a recommendation from each principle, and give appropriate examples. We report about the feedback from participants to the schools and lectures, and relate our principles to other pedagogical suggestions for teaching Formal Methods.

1 Introduction

Formal Methods are one means in Software Engineering that can help ensure that a computer system meets its requirements. They can help make descriptions precise. Such descriptions include requirements, specifications, and models, which appear at different levels and moments in the life cycle. They can also help in different kinds of analysis. The use of Formal Methods in Verification and Validation is wide and includes techniques such as static analysis, formal testing, model checking, runtime verification, and theorem proving.

In many engineering-based application areas of computer science, e.g., in the railway domain, Formal Methods have meanwhile reached a level of maturity that already enables the compilation of a so-called body of knowledge, i.e., a handbook compiling the key concepts, terms and activities that constitute the professional domain [FM-13]. In 2011, Barnes stated that "the application of Formal Methods is a cost effective route to the development of high integrity software" [Bar11]. In the hardware domain, Formal Methods are an essential part of the design processes [Lam05].

In Computer Science education, however, Formal Methods often play a minor role only. Typical questions raised in curriculum discussions include: Which of the many Formal

[6] The book "Formal Methods for Software Engineering – Languages, Methods, Application Domains" by the same authors is about to appear at Springer Verlag soon.

Methods shall be taught? Will the material taught be accessible to mainstream students? Will a Formal Methods course be attractive to students? Are there teachable case studies beyond the toy example level? Can students be involved in real projects where they can apply Formal Methods?

As a constructive and positive response to such questions, we are writing the book "Formal Methods for Software Engineering – Languages, Methods, Application Domains" [CRS+]. This paper offers an insight into the main principles adopted in the writing of this book. Teaching the contents of this book at summer schools associated with the SEFM conference series [SEF], and in various courses in different universities has added didactical insights. There were two groups of people present in these classes, to which the book is targeted:

- taught master students, possibly striving for a career in industry, and
- doctoral students in their early years, possibly in search of a suitable topic for their dissertation.

The rest of this paper is divided as follows. Section 2 presents the ten principles we propose for teaching Formal Methods with a view to making the subject fun and favourable. These serve as the guiding beacons for the writing of the book. Section 3 describes some of the student feedback we have received on delivering the contents of the book at a recent summer school. This provides a critical source of evaluation for us. Section 4 reviews some of the relevant literature in this area. Section 5 concludes the paper.

2 Principles

The inception of our book is due to 1st International School on Software Engineering and Formal Methods held in Cape Town, South Africa, in late October to early November of 2008. The two week school consisted of five courses on the application of Formal Methods to software design and verification delivered to an audience of graduate and research students. The focus of the book on addressing the fundamentals and facilitating learning through examples is largely owed to the instructional setting offered by the school. We have approached the subject of the book with a similar readership in mind and a strong desire to make the subject more widely accessible.

In the following, we state and exemplify our principles for teaching Fun With Formal Methods.

Principle 1: The field of Formal Methods is too large to gain encyclopaedic knowledge – choose representatives

The variety of Formal Methods is overwhelming. This might leave the beginner or potential user to be lost in the field. Which method shall be selected in a given context?

Here we recommend teaching a non-representative selection of methods with a solid introduction to each of them. Specialisation in one method is unproblematic, as the foundations of Formal Methods are well connected. For instance, concepts studied, say, in the

context of process algebra, are also to be found in temporal logics, which again are closely connected to automata theory, and are applied, e.g., in testing. Within a discipline, there are often attempts to unify structural insights. In logics, for example, the theory of institutions provides a general framework in which logical properties can be studied in a uniform way. The methodological approach to different Formal Methods often is comparable. Consequently, there is loads to gain by intensively studying selected few methods.

Principle 2: Formal Methods are more than pure/poor Mathematics – focus on Engineering

Although at the beginning computer science was strongly influenced by Mathematics, within the last 50 years it has become more and more of an engineering discipline. A fact is that many CS students quit because they cannot see the relevance of their first years' Mathematics classes to the engineering problems they want to solve. Thus, we encourage to motivate the introduction of Formal Methods by engineering challenges.

For example, we see Formal Methods as part of a Software Engineering curriculum. Here, it should be conveyed that the use of Formal Methods in a software development is not constrained to a specific process and life cycle model. That is, Formal Methods can be used with traditional as well as agile models. Moreover, Formal Methods should not constitute separate phases or sprints, but should be rather integrated as part of the general validation activities. Thus, teaching Formal Methods should frequently resort to other topics in Software Engineering.

Mishra and Schlingloff [MS08] evaluate the compliance of Formal Methods with process areas identified in CMMI-DEV, the capability maturity model integration for development. Their result is that out of 22 process areas from CMMI, six can be satisfied fully or largely with a formal specification based development approach. Notably, the process areas requirements management, product integration, requirements development, technical solutions, validation and verification are supported to a large extent. Mishra and Schlingloff also show the possibility of automation in process compliance, which reduces the effort for the implementation of a process model.

Formal Methods begin to play a significant role also in Software Engineering standards. For example, the RTCA standard DO-178C "Software Considerations in Airborne Systems and Equipment Certification", includes the DO-333 "Formal Methods" supplement addressing Formal Methods to complement testing. Using this example and the engineering challenges which led to its conception can greatly increase the students' motivation. Furthermore, success stories related to industrial applications and successful careers in academia and in industry can support this argument.

Principle 3: Formal Methods need tools – make them available

In the classroom, Formal Methods usually start on the "blackboard" only: Toy examples are treated in an exemplary way. With paper and pen one checks how a method works. For convincing case studies, however, Formal Methods need tool support in order to become

applicable. Tools for simulation of behaviour and visualisation of state space or traces are essential to allow students to understand the behaviour associated with their models.

This is the case as software systems are fundamentally different compared to Mathematical theories:

Numbers of axioms involved. The number of axioms when applying Formal Methods is by magnitudes larger than the number of axioms involved in Mathematical theories. Consequently, tool support is needed in Formal Methods for sheer book keeping.

Ownership and interest. Most software systems are the intellectual property of a company. This restricts access to the actual code; interest in their design is limited. Mathematical theories are part of the scientific process and publicly available. There is scientific interest in them. Therefore, proofs related to a specific software system are studied by few people only, and only when necessary – while Mathematical proofs are studied by many, over and over again. Consequently, tools play the role of "proof checkers" for quality control in Formal Methods.

Change of axiomatic basis. Requirements of software systems are bound to change in small time intervals. This means that design steps involving Formal Methods need to be repeated several times, sometimes already during the design phase, certainly when maintaining the system. Mathematical theories, however, are stable over centuries. Consequently, tools are needed to help with management of change in Formal Methods.

Thus, it is essential that students have access to powerful, even industrial-strength tools. They should also be referred to databases with "big" examples on which they can try these tools. There is a plethora of free Formal Methods tools available, and many tool providers will give students free access to expensive commercial tools, if certain legal restrictions are met. Usually, the negotiation process takes several month, so start well in advance.

Principle 4: Modelling versus programming – work out the differences

Models of software systems are different from programming code as programs are executable, while models can be executable. A model is a purposeful abstraction of either an existing system or a system still to be built. In the first case, the purpose of modelling is of analytical nature, in the second case the aim is to assist system construction.

Abstraction Models can be constructed at a high level of abstraction on which data and processes may often be described in an intuitive way. In programming it is necessary to spell out how to implement this data and processes by adding details. In fact, modelling can focus on the central ideas, irrelevant details are abstracted away. Here, the usability, i.e., the ease to express these ideas, is important when choosing a modelling language. Such languages often have a visual representation, e.g., automata, process algebras, Petri nets. Tool can support such visualisations allowing to design models in an intuitive way and to easily learn the semantics.

Validation This process relates the informal system description with the most abstract formal level. In this context, the aim of a model is to provide evidence that the modelled

system satisfies given properties. In a model-based approach informal properties may be formalized as processes or logic formulae. Although natural language descriptions are easily comprehensible, finding their correct formalisations is a hard task. Consequently, providing templates which correspond to well-known properties may assist students in their specification efforts. This also may help them to understand the correspondence between the natural language formulation and the formalisation of the property.

Refinement and Verification Refinement transforms an abstract model into a more concrete one, whereas verification gives a formal argument for the correspondence between the two. The more concrete a model is, the harder becomes the verification task. In this context it is important to convey two messages: tools are required to perform verification, see Principle 3; abstraction is an important precondition to work with Formal Methods tools.

Principle 5: Tools teach the method – use them

In the old days, a programming language was taught by going through its syntactic constructs and highlighting the (denotational or operational) semantics associated with each

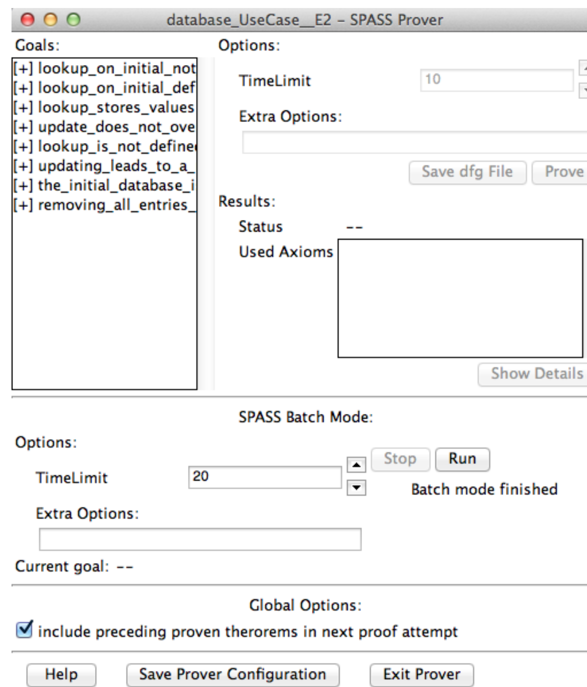

Fig. 1. Snapshot of Hets validating the database specification.

construct. Like it or not - this is no longer the way that programming is learned. Similar to children learning to speak, nowadays students learn to program by mimicking and modifying given examples, interacting with an IDE, and experimenting with the results.

We claim that the same should hold for Formal Methods. Instead of tediously going through the semantics of each construct in a formal language, allow the students to experiment with an appropriate tool to discover the semantics for themselves. Non-withstanding, the formal semantics needs to be available so students so that they can use it in case of need.

We illustrate this principle with an example written in the algebraic specification language CASL. The task is to specify an elementary database:

[Database in CASL] With the operation symbols *initial* and *update* we construct a table. The so constructed table can be inspected with *look_up*, manipulated with *remove*, and checked with *isEmpty*.

spec DATABASE =
  sorts *Database*, *Name*, *Number*
  ops $initial : Database;$
    $look_up : Database \times Name \to ? Number;$
    $update : Database \times Name \times Number \to Database;$
    $remove : Database \times Name \to Database$
  pred $isEmpty : Database$
  $\forall\ db : Database;\ name, name1, name2 : Name;\ number : Number$
  • $\neg\ def\ look_up(initial, name)$ %(non_def_initial)%
  • $name1 = name2$
   $\Rightarrow look_up(update(db, name1, number), name2) = number$
   %(name_found)%
  • $\neg\ name1 = name2$
   $\Rightarrow look_up(update(db, name1, number), name2)$
    $= look_up(db, name2)$ %(name_not_found)%
  . . .

Taking the operations *initial* and *update* as constructors, our axioms systematically cover how the operations *look_up* and *remove* and the predicate *isEmpty* interact with them: In the state *initial* the database has no entries. Consequently, observing this state with the *look_up* function, yields no result – see %(non_def_initial)%. When there is at least one entry in the database, we might obtain a result when observing it: Should the last entry match the look-up request, the number of the entry is to be returned – see %(name_found)%; otherwise, we have to inspect the previous entries – see the axiom %(name_not_found)%. . . .

This explanation is enough to grasp the "functional specification style" applied in this example – it's effects (including variations of the axioms) can then be observed by using a tool, in our case an automatic theorem prover. Populating the database with name constants *Erna*, and *Hugo* and number constants $N4711$ and $N17$, using the tool set

HETS with the automated theorem prover SPASS one can prove, e.g., that the following holds (see also Fig. 1):

[Database properties]

- $\neg$ $look_up(initial, Hugo) = N17$
  %(lookup on initial not equal to 17)%
- $\neg$ $def\ look_up(initial, Hugo)$ %(lookup on initial undefined)%
- $look_up(update(initial, Hugo, N4711), Hugo) = N4711$
  %(lookup stores values)%
- $look_up(update(update(initial, Hugo, N4711), Erna, N17), Hugo)$
  $= N4711$ %(update does not overwrite)%
- $\neg$ $def\ look_up(update(initial, Erna, N17), Hugo)$
  %(lookup is not defined without update)%
- $\neg$ $isEmpty(update(update(initial, Hugo, N4711), Erna, N17))$
  %(updating leads to a non empty database)%
- $isEmpty(initial)$ %(the initial database is empty)%
- $isEmpty(remove(update(initial, Hugo, N4711), Hugo))$
  %(removing all entries leads to an empty database)%

That is, students can actually experiment with the database and can see if their understanding of how it should behave is met. For instance, a database can have an overwriting semantics – two updates for the same name are possible, the last update determines the value (this is the way how the database is specified here); another design choice would be that the update function is only defined for names not yet present in the database.

This example shows that it is beneficial to study specifications by the means of tools.

Principle 6: Formal Methods need lab classes – create a stable platform

For students, carefully designed lab classes can be enormously motivating – see the student quotes in Section 3 concerning Reflections. Labs can offer hands-on experience with Formal Methods tools and practical examples. Such teaching style appeals to the plug-and-play mindset of a student generation who loves to play with gadgets of all kinds. Furthermore, it can provide the students with a sense of achievement: students use Formal Methods with a concrete, visible effect on their screen – rather than being lost or even frustrated in some semantical detail. Beyond such fun and achievement aspects, lab-classes allow students to judge how mature the tool support for a specific method is.

When preparing lab classes, to minimize the effort it is worthwhile – and we do so in the context of our book project – to work with frozen versions of tools compiled on a live-CD. Freezing the tools ensures that the effects one wishes to demonstrate are actually there (e.g., a time-out in an automatic theorem prover or a model checker, where a small change

of the tool's search heuristics can change the outcome). Moreover, the use of a live-CD circumvents installation problems.

For a first step, it is often enough to demonstrate elementary use of a tool, as exemplified in the following lab-exercise on "Simulating the Children-and-Candy Puzzle with ProBe" (see also Fig. 2):

[**Lab sheet for the process algebra CSP**]

1. Entering the command `probe` starts the simulator ProBe from a terminal. The simulator opens in a GUI.
2. One loads the file `childrensPuzzle.csp` by choosing the File option. This launches a new window.
3. Double clicking on the process `Children` allows one to expand the process as shown in the lectures.
4. Following any execution branch eventually yields a state where all three children hold 4 candies.

Here, the Children-and-Candy puzzle is an example already known from the lectures, where in the lab class the students shall explore and experience that (1) the system has many different execution paths and that (2) these paths are infinite.

The next step can then concern a change within the example (for example change the number of children, i.e., processes, involved in the puzzle). Only then students should be asked to develop own examples. Such an approach separates learning the (basic) functions of the tool, dealing with mistakes and error messages (when changing the example), and mastering the method (when working out an own example from scratch).

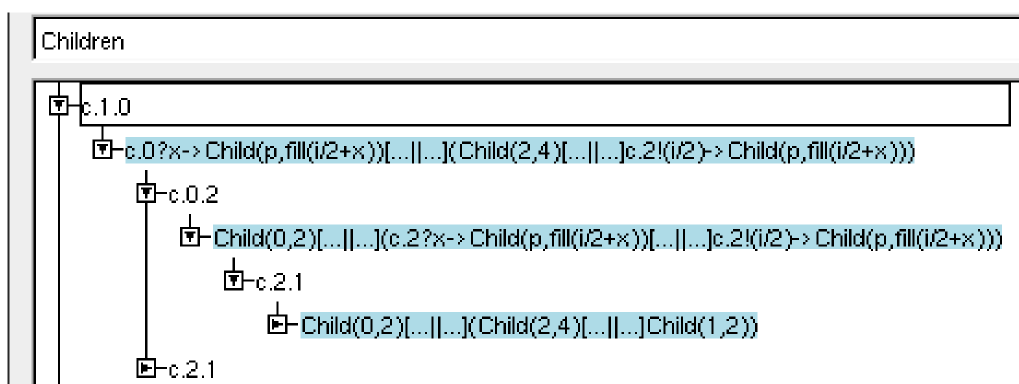

Fig. 2. Simulation in ProBe

Principle 7: Formal Methods are best taught by examples – choose from a domain familiar to the target group

An introduction to logic can read as follows:

> Today's your lucky day! You are the candidate in a big TV quiz show, and so far you have mastered all challenges. In the final round, there are three baskets, one of which contains the big prize, the other two being empty. The quizmaster gives you the following facts:
> 1. Either the prize is in the middle basket, or the right basket is empty.
> 2. If the prize is not in the left basket, then it is not in the middle.
>
> Which basket do you choose?

This example is typical for a number of "logic puzzles" which have been popular since the middle of the 19th century. One can formalize and solve such puzzles — not just for the sake of winning in TV quiz shows, but also to construct software solutions for all sorts of problems.

Seen from a hardware perspective, nowadays every computer is a binary device, built from switching elements which can be in one of two states each — on or off, high or low, zero or one etc. Therefore, hardware can be described and analysed with logical means.

From a software perspective, every programming language incorporates some logic, e.g., propositional logic for Boolean conditions in control flow decisions such as branches or loops. Thus, the study of logic is fundamental for programming. There even are special programming languages like, e.g., PROLOG, in which logic is turned into a programming paradigm. For many programming languages, there are special logics to express certain properties of program phrases, e.g., pre- and postconditions. Examples include ACSL for C, JML for Java, and Spark Ada.

Moreover, logic can be used as a means for specification of systems, independent of the particular programming language. Examples are CASL and the Z and B specification languages. Given a system specification, logic can be used to verify that a particular program is correct with respect to it. Specialised logics have been developed for this task, including temporal and dynamic logics.

From these examples it should be clear that there is not "one" logic in computer science, but many different logics are being used. Each of these can be considered to be a Formal Methods:

- The syntax of a logic usually is given by a small grammar which defines the well-formed formulae.
- For the semantics, notions like "model", "interpretation" and "validity" are defined.
- The method usually includes ways to show which formulae are valid in a given model or in all models of a certain class.

All the above examples are familiar to computer science students.

Principle 8: Each Formal Method consists of syntax, semantics and algorithms - focus uniformly on these key ingredients

To illustrate this principle, we discuss these ingredients in the context of text replacement:

[**Motivation**] Assume that we want to replace all occurrences of certain patterns in a text, e.g. remove all comments from an HTML document. In HTML, comments are marked beginning with "<!--" and ending with "-->". Most editors offer a facility for replacement based on *regular expressions*, that is, you may specify the symbol '*' as a wildcard in the search. With this, replacing "<!--*-->" by an empty string yields the desired result.

Starting from such an accessible example, one can discuss syntax, semantics, and method for regular expressions:

[**Syntax**] Syntactically, each formal method deals with objects denoted in a formal language. This gives a simple example for using Backus-Naur-Form [Bac59]:

$$\mathit{Regexp}_{\mathcal{A}} \;::=\; a \in \mathcal{A} \;\mid\; \lambda \;\mid\; (\mathit{Regexp}_{\mathcal{A}}\ \mathit{Regexp}_{\mathcal{A}}) \;\mid\; (\mathit{Regexp}_{\mathcal{A}} + \mathit{Regexp}_{\mathcal{A}}) \;\mid\; \mathit{Regexp}_{\mathcal{A}}^{*}$$

Semantically, regular expressions can be dealt with in denotational, operational, and axiomatic style.

[**Denotational semantics**] For any regular expression φ, we define the denoted language $[\![\varphi]\!]$ by the following clauses:

- $[\![a]\!] \triangleq \{a\}$ for any $a \in \mathcal{A}$. That is, the regular expression 'a' defines the language consisting solely of the one-letter word 'a'.
- $[\![\lambda]\!] \triangleq \{\}$. That is, λ denotes the empty language.
- ...

[**Operational semantics**] For any regular expression φ, we define an *automaton* $\mathbf{A}(\varphi)$, that is, a graph (N, E, s_0, S_F), where N is a nonempty set of *nodes*, $E \subseteq (N \times \mathcal{A} \times N) \cup (N \times N)$ is a set of (labelled) *edges*, $s_0 \in N$ is the *initial node* and $S_F \subseteq N$ is the set of *final nodes*.

- $\mathbf{A}(a) \triangleq (\{s_0, s_1\}, \{(s_0, a, s_1)\}, s_0, \{s_1\})$ for any letter $a \in \mathcal{A}$.
- $\mathbf{A}(\lambda) \triangleq (\{s_0\}, \{\}, s_0, \{\})$.
- ...

[**Axiomatic semantics**] Axiomatic systems for equality of regular expressions were given by Salomaa in [Sal66] and Kozen in [Koz94] and [KS12]. We call an equation $\alpha = \beta$ *derivable* and write $\vdash \alpha = \beta$, if it is either an instance of one of the following axioms or follows from a set of such instances by a finite number of applications of the following rules. Salomaa gives the following axioms and derivation rules:

- $\vdash (\alpha + (\beta + \gamma)) = ((\alpha + \beta) + \gamma)$, $\vdash (\alpha(\beta\gamma)) = ((\alpha\beta)\gamma)$
- $\vdash (\alpha(\beta + \gamma)) = ((\alpha\beta) + (\alpha\gamma))$, $\vdash ((\alpha + \beta)\gamma = ((\alpha\gamma) + (\beta\gamma))$
- ...

A formal language is described by an unambiguous syntax and a Mathematical semantics. For a Formal *Method* (as opposed to a formal language) it is essential that there are some *algorithms* or *procedures* which describe what can be done with the syntactic objects in practice. A Formal Method describes how to "work with the language", that is, perform some activity on its elements in order to achieve certain results.

Continuing with our example of regular expression, we define the following method:

[**Method**] A frequent task while writing scientific articles is to consistently replace certain text passages by others in the whole text. A replacement $[\alpha := \beta]$ consists of a regular expression α and a word β over $\mathcal{A}$. The word δ is the result of the replacement $[\alpha := \beta]$ on a word γ, denoted as $\delta = \gamma[\alpha := \beta]$ if one of the following holds:

- Either there exist γ_1, γ_2 and γ_3 such that
  1. $\gamma = \gamma_1\gamma_2\gamma_3$,
  2. $\gamma_2 \in [\![\alpha]\!] \setminus [\![\varepsilon]\!]$,
  3. ...
- or...

As an application of regular replacement, we note that
$(p \Rightarrow (q \Rightarrow p))[(p + q) := (p \Rightarrow q)] = ((p \Rightarrow q) \Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow q)))$

The Backus-Naur-Form being introduced this way, we take care to present other formal languages in exactly the same format, e.g., various logics and process algebra.

Principle 9: Formal Methods have several dimensions – use a taxonomy

In order to give students an orientation, it is important to provide a taxonomy. Formal Methods can, e.g., be categorized according to the following dimensions.

Application range – this dimension determines the application domain (e.g., avionics, railway, finance) and the specific needs of this domain (whether the systems are mainly reactive, interactive, real time, spatial, mobile, service-oriented, etc.)

Underlying technology – this dimension notes how the method can be realised. Typical methods include refinement, simulation and symbolic execution, model-checking,

automated or interactive theorem proving, static analysis etc., with technologies such as resolution, SAT solving, symbolic representation etc.

Properties under concern – here we categorize properties of the systems which are the subject of the Formal Method and which are supported by the method (safety, liveness, fairness, security, consistency, etc.)

Usability – this dimension describes how well fit the method is for actual use (universality, applicability, expressivity, maturity, learning-curve, intelligibility, etc.)

Other criteria can include lightweight vs. heavyweight Formal Methods, specification focussed vs. analysis focussed, correctness-by-construction vs. design-and-proof-methodology, etc.

Principle 10: Formal Methods are fun – shout it out loud!

Psychology tells us that the human learning capacity is highest when we enjoy what we are doing. Many researchers are attracted to Formal Methods just because it is fun to play with them – so the best way of teaching is to convey this fun to the students.

As an example, we start our book with the following paragraph:

> [**Fun with Formal Methods**] What is a Formal Method?
> You have just bought a book on Formal Methods and are making holiday plans in the Caribbean to read it on the beach. In order to guarantee the reservation, your travel agent requires a deposit. You decide to pay electronically via credit card.

With such a motivation, we describe some typical requirements for an electronic payment system.

A strong motivator are also competitions. There exist several competitions in the Formal Methods community, e.g., the VerifyThis Verification Competition, the Hardware Model Checking Competition, or the SAT competition. Of course, it is not easy for a beginner to enter such a competition, but you can set up an accessible competition amongst your class and provide a small price.

3 Reflections

We put our principles to test by teaching the material at three international summer schools (one of which was exclusively dedicated to the curriculum of the book). Furthermore, we used the material in several classes in our home universities. We collected results and opinions from the participants and evaluated them. Overall, the students were quite satisfied with the lectures and materials. Some remarks from students' survey were

- "The best part of the school is the way the courses are related."
- "The structure of courses is very good."
- "The school opened up new ways of thinking."

– (The school helped in) "showing that theory comes from practice and vice versa, but practice is the goal."
– (I liked) "the practical applications of Formal Methods."
– (Now) "I feel ready to approach logic from a knowledgeable point."
– (The required) "hands-on exercises were very good."

There was unanimous support (60% agreement, 40% strong agreement) for the statement "Teaching and learning methods used have enabled me to learn effectively". 80% of the participants agreed or agreed strongly to the statement "Teaching resources were adequate", reflecting the fact that some participants would have liked more practical examples and lab classes.

This shows the importance of our principles 3 and 5. Even though we dedicated quite a large portion (around one third) of the available time budget to lab classes, according to the participants they could have played an even bigger role. Some quotes from the section "Suggestions for Improvements" of the questionnaire were

– "More practical sessions with the tools."
– "More on applications, especially security."
– "More lab lessons."

Summarizing, we can conclude that our approach to teaching Formal Methods for Software Engineering is quite successful, and that the ten principles mentioned above have passed the on-road test of classroom use.

4 Related Work

The Formal Methods community has been well aware of the didactic challenges faced in teaching of the subject. We find a variety of approaches adopted in the teaching delivery of the subject.

Gibson and Méry [GM98] reveal some lessons from their case study-led approach to a course where fundamental problems, and the need for formality, is demonstrated over a set of case studies delivered to students. They also emphasise the need for tool support and the practical hands-on nature of the learning that such provision facilitates. This is evidenced to provide for rich interaction between the teacher and the learner, ultimately helping both. Our principles embrace both of these lessons. The use of tool support and practical learning has also been advocated by Heitmeyer [Hei98] and Liu *et al.* [LTHN09]. The latter, however, promote the value of writing formal specifications by hand to start with. This allows them to learn syntax and semantics as if it were a foreign language for adoption. The use of examples and case studies for their instructive value is acknowledged by several [Bur95,GM98,Hei98,RS04]. Moreover, their effective value for student assessment is recognised by Sotiriadou and Kefalas [SK00] while pointing out to the need for equal emphasis on both understanding and ability to construct formal specifications. A dearth of real world case studies is a challenge we face if such an approach for teaching and learning is to be pursued arguably [Bur95,Hei98]. Over the years, the situation has

improved as the uptake of Formal Methods in the industry progresses [WLBF09]. However, student motivation remains a concern and, as Reed and Sinclair [RS04] show, using smaller persistent examples is but only one way to address this problem, albeit to limited effect.

There is consensus amongst the community on the two-way relationship that exists between the state of adoption of Formal Methods within the industry and the quality of teaching of the subject [Hei98,Bur95,RS04,Bjø01]. While the former serves as a potential lack of motivation, the situation is not helped by rolling sets of graduates who come with limited knowledge or experience of the subject. Undoubtedly, any solution needs to address fundamental problems of student attitudes (to subjects such as Mathematics [Fin03] and the true cost of software errors in the industry [JMØ06]). Recent reforms to the teaching of computer sciences in the UK schools, where appropriate emphasis on programming and problem-solving nature of the subject is placed [BKC+13], lay down the foundation to a student body more able and equipped for Formal Methods in the future.

While the above issues inform the endeavours towards writing our book, including lessons from those who have strived to achieve the same [Bjø01], our approach promises a departure from tradition to a more fresh approach.

5 What next?

The challenge of Software Engineering has been acknowledged for decades now; the notion of software crisis has been related to the debate on software industry [Dij72,NR69] since early days. It is safe to presume the state of the industry will continue this course unless methods and practices are addressed. Our book is motivated by the very challenge and has set out to address the fundamental need for clear and coherent reasoning. We also acknowledge that a new era has dawned with breakthroughs in electronics and communications, and as such three trends have emerged over the recent years: (a) software is increasingly designed to support critical systems, which require safety and reliability guarantees to be predicated over execution, (b) modern computing is becoming pervasive [Sat01]. No longer is software sitting comfortably on desktops, but seamlessly dispersed across urban and household installations, needing to be open to cross-layer configuration and communication, and (c) user centricity is vital. As systems become more interactive, how they appear to and engage end-users has received more attention than ever before [Dix10].

We make a deliberate effort to encourage the influence of the above themes over several examples of applications spreading across the book. Such trends only add to the challenge of Software Engineering as they bring increased complexity in requirements and specification, added layers of abstraction for design, and the potential for subtle errors at the implementation stage. Equally, however, they make the examples motivating and fun. The book is due out. Watch this space!

Acknowledgement

We would like to express our sincere gratitude to colleagues and students who have contributed to our writing effort over discussions, lectures and the three summer schools. Amongst them, we especially want to thank Erwin R. Catesbeiana (Jr) for motivating classroom experiences. All this has helped with forming ideas and reflections over the content and style of the book. We remain grateful to the wider Formal Methods community who, over the years, has also informed the debate on how best to teach the subject and ensure that this carries on.

This work is partly supported by Macao Foundation.

References

[Bac59] J. W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In *Proceedings of the International Conference on Information Processing*. UNESCO, 1959. Available via the web site of the Computer History Museum's Software Preservation Group, `http://www.softwarepreservation.org`.

[Bar11] Janet Elizabeth Barnes. Experiences in the industrial use of Formal Methods. In Alexander Romanovsky, Cliff Jones, Jens Bendiposto, and Michael Leuschel, editors, *AVoCS'11*. Electronic Communications of the EASST, 2011.

[Bjø01] Dines Bjørner. On teaching software engineering based on formal techniques - thoughts about and plans for - a different software engineering text book. *Journal of Universal Computer Science*, 7(8):641–667, 2001.

[BKC+13] Neil Christopher Charles Brown, Michael Kölling, Tom Crick, Simon Peyton Jones, Simon Humphreys, and Sue Sentance. Bringing computer science back into schools: lessons from the UK. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 269–274. ACM, 2013.

[Bur95] Colin J. Burgess. The role of Formal Methods in Software Engineering education and industry. *University of Bristol, UK*, 1995.

[CRS+] Antonio Cerone, Markus Roggenbach, Bernd-Holger Schlingloff, Gerardo Schneider, and Siraj Ahmed Shaikh. *Formal Methods for Software Engineering – Languages, Methods, Application Domains*. Springer. To appear.

[Dij72] Edsger Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[Dix10] Alan Dix. Human-computer interaction: A stable discipline, a nascent science, and the growth of the long tail. *Interacting with Computers*, 22(1):13–27, 2010.

[Fin03] Kate Finney. Hello class – let me introduce you to Mr. Six. *Teaching Formal Methods*, 2003.

[FM-13] Towards a Formal Methods Body of Knowledge for Railway conrol and safety systems, 2013. `http://ssfmgroup.wordpress.com`.

[GM98] Jean-Paul Gibson and Dominique Méry. Teaching Formal Methods: Lessons to learn. In *IWFM. Workshops in Computing, BCS*, 1998.

[Hei98] Constance Heitmeyer. On the need for practical Formal Methods. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 18–26. Springer, 1998.

[JMØ06] Magne Jørgensen and Kjetil Moløkken-Østvold. How large are software cost overruns? A review of the 1994 CHAOS report. *Information and Software Technology*, 48(4):297–301, 2006.

[Koz94] Dexter Kozen. A completeness theorem for Kleene Algebras and the Algebra of Regular Events. *Information and Computation*, 110:366–390, 1994.

[KS12] Dexter Kozen and Alexandra Silva. Left-handed completeness. In Wolfram Kahl and Timothy G. Griffin, editors, *RAMiCS 2012*, LNCS 7560, pages 162–178. Springer, 2012.

[Lam05] William K. Lam. *Hardware Design Verification*. Prentice Hall, 2005.

[LTHN09] Shaoying Liu, Kazuhiro Takahashi, Toshinori Hayashi, and Toshihiro Nakayama. Teaching Formal Methods in the context of Software Engineering. *ACM SIGCSE Bulletin*, 41(2):17–23, 2009.

[MS08] Satish Mishra and Bernd-Holger Schlingloff. Compliance of CMMI process area with specification based development. In *SERA '08*, pages 77–84. IEEE Computer Society, 2008.

[NR69] Peter Naur and Brian Randell. Software Engineering Report of a conference sponsored by the NATO Science Committee Garmisch Germany 7th–11th October 1968. 1969.

[RS04] Joy N. Reed and Jane E. Sinclair. Motivating study of Formal Methods in the classroom. In *Teaching Formal Methods*, pages 32–46. Springer, 2004.

[Sal66] Arto Salomaa. Two complete axiom systems for the algebra of regular events. *J. ACM*, 13(1):158–169, January 1966.

[Sat01] M. Satyanarayanan. Pervasive computing: vision and challenges. *Personal Communications, IEEE*, 8(4):10–17, 2001.

[SEF] IEEE Conferences on Software Engineering and Formal Methods (SEFM). `http://sefm.iist.unu.edu/`.

[SK00] Anna Sotiriadou and Petros Kefalas. Teaching Formal Methods in computer science undergraduates. *Athens, World Scientific Engineering Society Press*, pages 91–95, 2000.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John S. Fitzgerald. Formal Methods: Practice and experience. *ACM Comput. Surv.*, 41(4), 2009.