Structuring Formal Methods into the Undergraduate Computer Science Curriculum

S. Ramnath[1], S.M. Walk[2]

[1] Dept of CSIT, St Cloud State University, Minnesota
[2]Dept of Math and Stat, St Cloud State University, Minnesota

Abstract

There is an urgent need to emphasize and integrate Formal Methods into the undergraduate curriculum in Computer Science in the United States. The core topics associated with Discrete Math courses tend to shy away from formal approaches to validate correctness of structures; most programming and data structures texts make no mention of the need to construct "provable" or at least "easily testable" code; and formal approaches have failed to gain a foothold in even the upper-level CS courses. It is further disheartening to note that the word "formal", as applied to curriculum or outcomes, appears only three times in the 2020 ACM IEEE Computing Curricula, and does not appear at all in the ABET CAC or ABET EAC criteria. We are entering an age defined by a highly interconnected, ubiquitous computing environment, with a large AI component. Knowing the precise capability of our systems is particularly vital to safeguard our safety-critical and mission-critical applications. In such an environment, the lack of a well-structured exposure to formal methods is a serious shortcoming in our computing curriculum.

Our article explores possible remedies for this situation. We discuss efforts (or the lack thereof) that have been made over the years to introduce these concepts into the courses at various levels. We examine the curricular, pedagogical and organizational challenges involved in bringing Formal Methods into the mainstream of computing disciplines. Finally, we recommend possible initiatives that need to be undertaken to integrate, emphasize, and help students gain an appreciation for formal methods throughout our computing curricula.

Keywords: Computer Science, Formal Methods, Software Engineering, Curriculum Development, Teaching and Learning

1 INTRODUCTION

The nature of the relationship between mathematical foundations and programming skills has a long history in the undergraduate Computer Science curriculum. The typical Computer Science program started out as an extension to mathematics, developed into a separate entity for which a strong mathematical foundation was still considered essential early in the program, and became one where a single Discrete Math course is the only prerequisite for most Computer Science courses. Over the years, the mathematical background considered essential to learn programming has been reduced from completion of Calculus to readiness for College Algebra. A couple of decades ago, it was not atypical for programs to require a semester of Calculus, followed by a semester of Discrete Math, before students were even admitted to the program.

This change in mathematics requirements has been accompanied by a shift in the approach taken towards programming: whereas earlier, programming was built on mathematical and logical foundations, it is now mostly built on example and intuition. This shift is also reflected in the process of admission to the BS in Computer Science at our own institution. Earlier, it was not uncommon to see students complete Calculus, Discrete Math and a semester of programming before applying for the major; now-a-days it is more common to see programs where students “directly declare” the major and complete two semesters of programming before they take Discrete Math. The Math requirement for the CS major has also changed: a second course in Discrete Math that dealt exclusively with logic, axioms and inference has disappeared. Although this describes the history of one program, it is fairly typical of the evolution of Computer Science majors.

The precise impact of all these changes is hard to assess. Many of the changes were carried out after a lot of deliberation; to some extent, the courses that were dropped were seen as losing their relevance and becoming hoops for students to jump through. Nonetheless, we are now faced with new challenges,

and it is essential that we reassess our priorities. Specifically, we are now dealing with a highly interconnected, ubiquitous computing environment, with a large AI component. Knowing precisely what our systems can and cannot do has become more important, and this awareness is particularly vital for safety-critical and mission-critical applications. Formal methods for verifying code are more important now than ever before. High personnel cost has often been cited as a reason for not embracing formal methods; this indicates that we need to improve the way we train our workforce and change the prevailing view of formal methods within our academic institutions.

In this article we discuss the challenge of structuring Formal Methods into the Computer Science curriculum and make some suggestions. The next section discusses efforts that have been made over the years to introduce these concepts into the courses at various levels. In Section 3 we look at the pedagogical challenges and possible approaches, and in Section 4 we suggest possible initiatives that need to be undertaken. Section 5 concludes the paper.

2 THE CURRENT SITUATION

To get a better understanding, it is useful to examine the existing situation in the field of formal methods from a few different viewpoints.

2.1 Developing Coursework and Pedagogy

Although there is a large body of literature that deals with teaching and learning of computing principles, relatively little attention has been paid to the teaching of formal methods. The reasons for a lack of interest can be difficult to pin down, but as pointed out in [1]:

> Formal methods are taught only in a limited number of computer science university programmes, mainly at postgraduate level, and are usually presented as such a difficult topic that university students keep away from them and the industry, in general, does not consider them as a worthy research and development investment. Even worse, most of the technicians (electrical or machine engineering) who design and build safety critical systems never had a course in formal methods during their studies.

There appears to be considerable variation in the prerequisites and the set of topics across all these courses. Although these courses in general seem to discuss some combination of notation, Hoare Logic and a specification language, one does find courses with outcomes/topics as follows, where a significant part of the course is devoted to automata theory and grammars.

> understand the concepts and relevance of logic, formal languages and automata theory, and computability.
> mechanical formal proofs, correctness for simple programs, solve problems in first-order logic.
> elementary machine models: designing finite-state automata, pushdown automata and Turing machines.
> problems in formal languages: writing regular expressions, regular grammars, and context-free grammars.

There appears to be some variation also in emphasis. Some courses seem to focus on formal specification alone; others seem to focus on the proof aspect. As another example, here is a course description

> Formal Methods are practical structuring and design patterns that encourage programming that is easy to understand and to maintain. They are only a part of the large body of "good programming practices," distinguished from that larger set because –for the methods we consider– the computer-science community has determined the science behind the structures: we know why they are effective.

A graduate student blogger, whose "…goal was to return to my workplace as a domain expert in formal methods…" offers an interesting point of view. A recent post refers to two kinds of formal methods approaches [2]:

> "Lightweight" is a term that's sometimes thrown around when discussing these kinds of model checking tools. I'd summarize lightweight formal methods as those built around the idea that "it's better to have an imperfect tool that's useful than a comprehensive tool that's unusable." Alloy (and, by extension, Forge) is probably the best example of this.
>
> Lean, Isabelle, and Coq are well-suited to working with systems at a lower level, and work as a mechanization of mathematical logic: a proof in a proof assistant is equivalent to (but typically more formal than) a proof that a mathematician or computer scientist might write on a piece of paper.

The variations are not necessarily a cause for concern, since all these approaches could be effective in improving the quality of our software engineers; but the variations point to a lack of consensus about such a course, in contrast to subjects such as Analysis of Algorithms or Theory of Computation, which have standard syllabi and textbooks. Further, several of the courses on formal methods spend a considerable amount of time on sets and logic, despite requiring Discrete Math as a prerequisite; this is an indicator that Discrete Math as widely taught is insufficient, so that math courses required for computer science may need revision if formal methods are to be integrated as mainstream computer science.

Another reason for the lack of interest in the subject is the perception that formal methods are all about "writing proofs" [3], and hence this field of study gets relegated to the "theoretical" side of computer science.

2.2 The Mathematics Connection in Computer Science Programs

As pointed out earlier, the relationship between Mathematics and Computer Science at the undergraduate level has changed in recent years. Although the connection remains very strong at the fundamental level, a lot has changed at the surface, giving the perception that Computer Science can be taught independent of Math. As pointed out in [4]

> Formal methods can be considered as the area of computer science that most effectively bridges the gap between mathematics and computer science. They are potentially a great educational tool for fostering mathematical reasoning skills and problem-solving abilities in a very wide audience of potential learners from university, industry, school and research.

Over the years Discrete Mathematics syllabi and textbooks have to focus on discrete structures like graphs, sets and relations. While this is useful, these courses need to place more emphasis on axiomatic systems/semantics and lay the groundwork for proofs of programs. To further strengthen the math connection, we need to ensure that [4]:

> ... new programmers connect program execution to proof, and that this connection is best made by the use of coordinated tools and examples that appear in both introductory programming and discrete mathematics.

As long as mathematics and mathematical reasoning courses are seen as hurdles to get over or experiences to survive—experiences that are seen as "other," as "a math course that we CS people have to take but that isn't really CS," there will be resistance to formal methods. The key is to help students see mathematics and computer science as co-existing, mutually beneficial curricular subjects.

2.3 Pedagogical Challenges and Approaches

For the most part, formal methods continue to be viewed as being on the "fringe", as far as undergraduate computer science programs are concerned. This attitude presents a formidable hurdle, and a culture shift is needed. Another barrier is the view that any kind of "proof writing" is difficult. We also see that students are increasingly impatient and seem less willing to spend time "thinking and planning" about code they have to write; it is easier (perhaps "quicker") for them to produce "quasi-correct" programs and test them, rather than spend time designing the logic. (Of course, some of the blame for that lies with easy access to computing resources, but curriculum and emphasis plays a big role.)

Fostering student appreciation of formal methods requires not only a change in attitude but also the learning of a lot of new concepts, and it cannot be achieved in one course. The process manifests itself in two ways: *connecting content modules across courses*, and *reinforcing concepts in later courses*. To deal with the first issue, formal methods concepts covered in later courses have to be tied back with material learned in earlier courses. The second issue arises when those skills are not used or required in later experiences. If students learn to prove programming assertions inductively in one class but then aren't asked to use that skill in later classes, then the skill atrophies; worse, cynicism sets in from the start, because they will of course hear from other students that "You learn that in that one class, but you never use it again." There is a spiral: Students do not take the skills seriously because they believe that they will never use them, so that instructors of later courses assume that students do not have the skills and structure their courses to accommodate that lack of skill.

Nonetheless, we believe that the spiral can be stopped and in fact can work positively in the opposite direction. Fortunately, some of the educational and pedagogical innovations from the last few decades can help us.

2.3.1 Working with Threshold Concepts

Threshold concepts were first recognized by Meyer and Land [5]

> A threshold concept can be considered as akin to a portal, opening up a new and previously inaccessible way of thinking about something.

Grasping such a concept causes the learner to engage with the discipline in a different way. As described in [5]:

> As a consequence of comprehending a threshold concept, there may thus be a transformed internal view of subject matter, subject landscape, or even world view

There is a lot of literature and research on the tailoring of pedagogical approaches specifically for such concepts. Fouberg[6] discusses these five ways to integrate threshold concepts into the classroom:

- Distinguish threshold concepts and spiral them into your course over the term.
- Ask students to reflect on what concepts functioned as threshold concepts for them.
- Teach your students about the importance of being uncertain.
- Write assignments that both integrate threshold concepts and scaffold knowledge usng Bloom's Taxonomy.
- Use formative assessment to get students thinking.

There is a lot of evidence to suggest that mastering the ability to our reasoning into the framework of a proof structure is a threshold concept. As pointed out in [7]

> One of the most difficult learning thresholds for students of mathematics is the concept of proof. The difficulty manifests itself in several ways: (1) appreciating why proofs are important; (2) the tension between verification and understanding; (3) proof construction.

In several ways, the teaching of formal methods mirrors all these challenges. As has been pointed out in research into the teaching of threshold concepts [8]:

> Sometimes the mastering of a concept involves both greater difficulty for the learner, and the benefit of increased maturity in the discipline. Such concepts have been recognized as "threshold concepts" in a variety of disciplines.

2.3.2 Using Different Kinds of Activities at Different Levels

It has been pointed out that a concept can be presented to learners at different levels using different strategies and different kinds of activities. [10]. These concepts have also been applied to the teaching of formal methods, over a wide spectrum of student abilities [9]. Choosing the kind of activity requires adaptation to the specific circumstance. As an example, in our mid-level course for mathematics and computing majors, described in a later section, the following sequence of activities are frequent features of the experience.

- Group activities lead students to proofs of famous results in mathematical folklore.

- Students are introduced to, and practice in class and on homework, the basic skills of setting up mathematical arguments for assertions with various structures ("for all . . . ," "if-then").
- From then on, students are asked to set up and carry out their own arguments in homework.
- Self-explanation with the reading of mathematical arguments is practiced—reading arguments is emphasized, with the observation that reading proofs makes one better at writing proofs and *vice versa*—and students are required to submit some written examples of their self-explanations.

This sequence has been designed to help students internalize their understanding of proofs in a gentle way. These are the four early experiences; the sequence is typically extended to cover other concepts as well, gradually ramping up the level of abstraction expected of the learner.

2.3.3 Employing the Whole-Part-Whole Approach

Another pedagogical (or rather, andrological) technique is the whole-part-whole approach[12], that has been particularly effective with adult learners. Here the learner is introduced to a "whole" problem, which sets the context for learning the "parts". Once the necessary parts have been learned, the students return to the "whole" and complete the exercise. The introduction of the whole at the beginning is effective in providing motivation and helps with retention. The effectiveness of this approach is documented in [13, 14]. Readers familiar with the "whole-part-whole" approach to teaching will no doubt have recognized hallmarks of that approach in the first three bullet points noted in the previous subsection, about a specific class; such an approach could be built into the curriculum of a Computer Science major program, with

- one or more entry-level courses introducing the need to verify program correctness (perhaps posing challenges to students to find test cases that would "break" each other's code) and the general idea of Formal Methods,
- one or more mid-level courses (like the one described earlier) to introduce the basic methods, and
- upper-level and capstone courses further refining and requiring the use of those methods with specific programs.

2.3.4 Employing Strategies that Promote Enduring Learning

It has been seen [11] that there are three key processes that reinforce learning: *retention, organization,* and *integration,* and these processes are triggered by the following practices:

- Ascertaining students' prior knowledge
- Engaging students in retrieval of previously learned information
- Promoting deep understanding through self-explanation

These practices, when consistently followed over a sequence of activities, can effect a transformation in the attitudes of learners.

3 EXPERIENCES AND OBSERVATIONS

A few reports are available on experimental observations about how students relate to reasoning about their own programs. All of these require students to demonstrate some understanding about a piece of code; such results are important, because these require students to engage with code beyond coding and testing, and these experiences can lay the foundation for abstract thinking needed for formal methods.

In [15] students were required to match code with given assertions, and conversely asked to match assertions with code. These experiments revealed that students often performed better when asked to choose code, rather than being asked to choose assertions that would match the given code. Generating proof for code usually requires the placement of assertions; activities such as those in [15] can help students build the necessary skills.

Research shows also that students succeed in producing correct programs without understanding why the program is correct. In [16] students to explain the structure and execution of their small programs after they had submitted them to a programming exercise. A third of the students struggled, but those who could explain their code were more likely to succeed in graduating from the program and entering the field.

A related issue is the tendency of students to jump straight into the code, putting together a design only as an afterthought. For instance, in our second-semester programming course at St. Cloud State University, a typical assessment is a written quiz (no computer allowed) on multi-way branching. Such a quiz follows multiple examples and assignments, and it generally presents the requirements for a simple problem, along with the design (a flowchart of a multi-way branch), and students need to translate the design into code. In our experience, students have to be repeatedly told that the code has to follow design and that they will lose points otherwise; and even then, we are not surprised if half the class still attempts to bypass the given design and write code from the functional requirements. One of the challenges we face is the need to wean students away from this habit of writing code directly, based on intuition and (limited) experience, and to help them replace it with the habits of planning and design.

A few other researchers [17, 18, 19, 20] have reported on their attempts to integrate and assimilate formal methods into undergraduate programs. They have been effective to varying degrees, and provide valuable insight for any program that seeks to place greater emphasis on formal methods. However, there remains a need for defining a a structured approach that has broad institutional recognition and can be replicated across programs with measurable results.

As mentioned earlier, there is a need to see math and CS courses as co-existing curricular subjects. The authors of [4] and [20] have described their experiences with tools and content to strengthen this connection, One way we have addressed the matter at our institution is to develop a sophomore/junior level class taught by mathematics faculty and populated by both mathematics and computer science majors, in which topics of mathematics and computer science are intermixed: In fact, the course's usual instructor tells students up front that "mathematics students will wonder why they're learning so much computer science stuff, and computing students will wonder why they're learning so much mathematics, but the answer is that the two disciplines contribute a lot to each other." Some of those two-way contributions seen in the class are the following:

- Computer numeration and pitfalls of floating-point arithmetic (topics required for computer science and software engineering accreditation) are encountered early on in the course. These explorations help both populations to understand that computers don't just "magically get" real numbers and that care is needed. Near the end of the course, students learn how the familiar number systems are constructed formally (a topic required for mathematics education majors), beginning with the natural numbers—and with the hope that the natural number system is consistent—helping both populations to see that the real numbers and the rational numbers and such aren't just magically "there" and that their use depends on proper definitions.
- Comparisons and contrasts between declarations of variables in programming and proof-writing are emphasized. Approaches to keeping track of assumptions in a proof are likened to those for keeping track of the depths of loops in programs.
- The consideration of axiom systems for the predicate calculus is compared with the programming of a computer, with proper axioms being the "macros to load."
- Algorithm analysis helps all students to see why they were required to learn techniques such as the Euclidean algorithm (even though "factoring numbers to find their gcd" seemed fine in junior high school) and synthetic division (even though "just plugging numbers into a polynomial" is the way they've always done it) as well as to expose them to some other methods for carrying out familiar tasks (such as multiplying integers) that are not as obvious as the familiar methods but are provably more efficient. A recurring theme is that "Mathematical thinking is not replaced by our use of technology; mathematical thinking improves our ability to use technology."

4 POSSIBLE INITIATIVES (RECOMMENDATIONS)

Recognizing the central importance of formal methods will clearly involve a culture-shift in the computing curriculum. As is to be expected, enabling such a shift will require a multi-pronged approach. We propose some initiatives that may need to be undertaken; this will obviously need a broader discussion.

4.1 A Framework for the Undergraduate Curriculum

There are clearly several aspects to this framework, and this section lists most of them; this list may not, however, be exhaustive.

4.1.1 The Role of Mathematics

There are three aspects to this role: *the content of math courses required for CS, the pedagogical approaches taken,* and *the connection between the contents of math and CS courses.*

The math content needs to promote “mathematical thinking and reasoning” rather than provide a long list of of definitional concepts. Typically, these courses should be such that the student completing the course feels much more comfortable with the need for proofs, their ability to work with proofs, and manage a higher level of abstraction.

Several pedagogical approaches have been listed earlier in this article. In general, these tend to focus on the idea of creating student experiences with associated outcomes, rather than focusing on the content. Courses tend to be created with content-based syllabi (and yes, outcomes), and it is not uncommon for faculty to get pressured by the need to cover the content and miss the wood for the trees. In this context, it is good to recall a quote from William Butler Yeats: "Education is not the filling of a pail, but the lighting of a fire."

Making a bridge between math and CS is critical if students are to be feel intrinsically motivated. The traditional idea of “stressing the importance” is helpful, but other, more concrete connections are needed. These can be made through share use of formalisms and notation, and the use of common technological tools. It is also useful to consider similar problems in both math and CS courses.

4.1.2 Building towards Formal Methods in the Computer Science Courses

There are several courses in the typical undergraduate computing major that can lay the foundation for grasping the more advanced concepts.

- CS1 and Data Structures. In these courses, students can be exposed to assertions, and where possible the idea of invariants. These efforts often fail if the assertions are added as “another requirement” in an assignment; it is essential that they be integrated into a problem-solving process. The use of models can be introduced during the process of problem definition.
- Logic circuit design. These courses have increasingly been removed from Computer Science, and topics folded into broader “computer architecture” courses; but sequential circuit design provides a great opportunity for students to model requirements as finite state machines.
- Theory of programming languages. These courses typically cover syntax of programming languages, along with a host of diverse topics. Learning about the role of formal grammars in the computing process, and the manner in which grammars can be suitably formalized, can help build an appreciation for formal methods.
- Computing Ethics. Special emphasis could be placed on the need to ensure correctness of code.
- Object-oriented software construction. This would be a course taken after Data Structures [13]. Creating a focus on modelling and going into the details of the Liskov Substitution Principle are topics that help lay the foundation for an advanced course on formal methods.[8]
- Software Engineering courses. Formal specification methods can be introduced in a Software Analysis course, and Software Quality courses can be expanded to include formal validation methods.
- Course dedicated to Formal Verification and Validation. It should be noted that if students have the background from discrete math and the courses listed above, such a course could be built around on a project-based learning model. It has been seen that such an approach results in better engagement [8, 14] and retention [10]. Such a course could potentially substitute for a course like Theory of Computation.

It is important to note here that the manner which all this is built should be such that overall program size does not increase, and does not result in a greater cognitive load on the students. This structure is similar in spirit to the one proposed in [17], but somewhat broader in scope.

4.2 Special Recognition for Higher Standards

This transformation will clearly require a high degree of commitment from both the programs and the students. We cannot expect this level of commitment without conferring some distinction upon the participants. These distinctions will have to be conferred in two ways:

- *Recognizing programs that provide the necessary structure for students.* This structure involves two parts. The curricular materials will need to be suitably tailored to meet the requirements, and students going through the program will need to be monitored regarding the expected outcomes.
- *Recognizing students who meet the higher standards.* We suspect that a significant fraction of graduating students will not have mastered the use of formal methods at the appropriate level. Therefore, graduating from a Computer Science program should not be enough, in and of itself, to confer any distinction. Students graduating from recognized programs could have been required to master such topics or had the opportunity to earn related badges and certificates.

4.3 Developing Instructional Materials

For academic departments to structure formal methods into multiple courses, providing educational materials would be critical. The materials would need to be developed and made freely available. Authors of standard texts could be brought on board, and the textbooks could themselves suggest places where such modules could be integrated into a course.

4.4 Exerting Influence through Prominent Bodies

As we mentioned earlier, the bodies that play a significant role in defining curriculum will have to be a part of the solution. Creating curriculum and enabling academic departments to come on board can contribute to the momentum on this.

The topic of Computing Ethics deserves special mention in this context. Clause 3.7 in the ACM code of ethics exhorts computing professionals to “Recognize and take special care of systems that become integrated into the infrastructure of society”[21]. Perhaps the ACM could consider clauses to the effect that “the level of verification be appropriate to the importance of the code (perhaps that it is directly proportional to the amount of damage that could occur if the code fails)” and “programmers ensure that they have taken all feasible steps to ensure correctness of their code”? As an aside, we note that a phrase equivalent to “formal methods” does not appear in the code.

5 CONCLUSIONS

In this paper we have attempted to capture some of the dilemmas that those in academia face in their attempts to turn out software engineers who are ready to join teams working on safety-critical and mission-critical systems. Based on the literature and our experiences with teaching formal methods, we see a connection between preparing these students and shaping their abilities for and attitude towards proofs and theorems. Further, we have presented some suggestions for some initiatives that we feel will be critical to success in this endeavour.

The effort really does need to start with the curriculum. We cannot expect graduates to become experts in program verification as professionals if they never encountered the ideas as students. The more we engage students with proofs and with axiomatic systems and semantics, the more their eyes will be open to the possibility of unexpected cases and unintended consequences of something as seemingly cut-and-dried as a computer program. Recognizing that a general principle can be false even if some specific instances are true, that familiar properties such as associativity and distributivity can fail for floating-point numbers, that a set of axioms can have surprising and conflicting models (i.e. “the preconditions do not guarantee the desired postcondition”), that a well-used set of axioms or beliefs might harbor a disastrous inconsistency—all of these can help to awaken students to the need for care in programming and thus to lay the groundwork for proofs of programs. Ideally, they will come to see that the unexpected can happen if possibilities are not considered carefully and rigorously—and come to see themselves as professionals responsible for checking for such possibilities.

Effecting such a culture change is a formidable undertaking and will need a lot of effort from everyone. Our objective in writing this paper is to place this issue before this forum of researchers and educators,

and hopefully trigger a movement in the right direction. Formulating a plan of action will clearly have to be a larger exercise involving the larger community of professionals who see value in such an initiative.

REFERENCES

[1] Cerone, A., Lermer, K.R. (2021). Adapting to Different Types of Target Audience in Teaching Formal Methods. In: Cerone, A., Roggenbach, M. (eds) Formal Methods – Fun for Everybody. FMFun 2019. Communications in Computer and Information Science, vol 1301. Springer, Cham. https://doi.org/10.1007/978-3-030-71374-4_5

[2] What I've Learned About Formal Methods In Half a Year. https://jakob.space/blog/what-ive-learned-about-formal-methods.html, 2023.

[3] Make Formal Verification and Provably Correct Software Practical and Mainstream. Comments posted in May 2022. https://news.ycombinator.com/item?id=31543953.

[4] Wonnacott, D. G., & Osera, P. M. (2019). A Bridge Anchored on Both Sides: Formal Deduction in Introductory CS, and Code Proofs in Discrete Math. arXiv preprint arXiv:1907.04134.

[5] J. Meyer and R. Land. 2003. Threshold concepts and troublesome knowledge: Linkages to ways of thinking and practising within the disciplines. ETL Project Occasional Report 4 (2003). http://www.ed.ac.uk/etl/docs/ETLreport4.pdf

[6] Erin H Fouberg. September 25, 2019. 5 Ways to Integrate Threshold Concepts Into Your Classroom. The Wiley Network (September 25, 2019). https://www.wiley.com/network/latest-content/5-ways-to-integrate-threshold-concepts-into-your-classroom

[7] David Easdown. Teaching proofs in mathematics. 1st Joint International Meeting between the American Mathematical Society and the New Zealand Mathematical Society December 12-15, 2007. http://at.yorku.ca/c/a/t/m/51.

[8] Sarnath Ramnath and Brahma Dathan. 2022. Crossing Learning Thresholds Progressively via Active Learning. In Proceedings of the 2022 ACM SIGPLAN International Symposium on SPLASH-E (SPLASH-E 2022). Association for Computing Machinery, New York, NY, USA, 14–23. https://doi.org/10.1145/3563767.3568128.

[9] Cerone, A., Lermer, K.R. (2021). Adapting to Different Types of Target Audience in Teaching Formal Methods. In: Cerone, A., Roggenbach, M. (eds) Formal Methods – Fun for Everybody. FMFun 2019. Communications in Computer and Information Science, vol 1301. Springer, Cham. https://doi.org/10.1007/978-3-030-71374-4_5.

[10] HIGH IMPACT TEACHING STRATEGIES. https://www.education.vic.gov.au/Documents/school/teachers/management/highimpactteachingstrat.pdf

[11] Help Students Retain, Organize and Integrate Knowledge https://tll.mit.edu/teaching-resources/how-to-teach/help-students-retain-organize-and-integrate-knowledge/

[12] Knowles, M.S., Holton, E.F. III, and Swanson, R.A., 2011. The Adult Learner, Sixth Edition: The Definitive Classic in Adult Education and Human Resource Development, Taylor and Francis, seventh edition, 2011.

[13] Sarnath Ramnath and Brahma Dathan. 2008. Evolving an integrated curriculum for object-oriented analysis and design. SIGCSE Bull. 40, 1 (March 2008), 337–341. https://doi.org/10.1145/1352322.1352252

[14] Sarnath Ramnath and John H. Hoover. 2016. Enhancing Engagement by Blending Rigor and Relevance. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16). Association for Computing Machinery, New York, NY, USA, 108–113. https://doi.org/10.1145/2839509.2844554A.A. Author, "Online Article Title," *Periodical Title*, vol. Volume, no. Issue, pp.-pp., Publication Year. Retrieved from URL

[15] Blankenship, S. (2022). Learning to Reason About Code with Assertions: An Exploration with Two Student Populations https://tigerprints.clemson.edu/all_theses/3950/

[16] Lehtinen, T., Lukkarinen, A., & Haaranen, L. (2021, June). Students Struggle to Explain Their Own Program Code. In Proceedings of the 26th ACM Conference on Innovation and

Technology in Computer Science Education V. 1 (pp. 206-212). https://arxiv.org/pdf/2104.06710.pdf

[17] S. Skevoulis and V. Makarov, Integrating Formal Methods Tools Into Undergraduate Computer Science Curriculum, Proceedings. Frontiers in Education. 36th Annual Conference, San Diego, CA, USA, 2006, pp. 1-6, doi: 10.1109/FIE.2006.322570

[18] M Sebern, H Welch, Formal Methods In The Undergraduate Software Engineering Curriculum. 2008 Annual Conference & Exposition, 2008 - peer.asee.org

[19] A Zamansky, E Farchi. Exploring the role of logic and formal methods in information systems education. - SEFM 2015 Collocated Workshops, 2015 – Springer

[20] Alabi, O., Vu, A., & Osera, P. M. (2022, March). Snowflake: Supporting Programming and Proofs. In Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 2 (pp. 1398-1398).

[21] The ACM Code of Ethics. https://www.acm.org/code-of-ethics

[22] Cognitive Load Theory – The Definitive Guide. https://www.educationcorner.com/cognitive-load-theory/